



UNIVERSIDAD DE MURCIA

Escuela Internacional de Doctorado

Desarrollo, Optimización y Autooptimización de
Algoritmos Paralelos para Análisis Cinemático de Sistemas
Multicuerpo Basado en Ecuaciones de Grupo

D. José Carlos Cano Lorente

2021

A mi hermano.

Agradecimientos

Quiero comenzar con un reconocimiento muy especial a los directores de esta tesis. En primer lugar, por su tutela en el máster que despertó mi interés por la investigación gracias a un proyecto apasionante, con aplicaciones prácticas en el ámbito de la ingeniería. En segundo lugar, por haberme animado con grandes dosis de entusiasmo a continuar con un proyecto de tesis doctoral. En este período he podido constatar su enorme talento, profesionalidad y capacidad de trabajo. Siempre disponibles, sin importar fines de semana y vacaciones, ante cualquier petición de ayuda, orientación o consejo. Su rigor guió los planteamientos de la investigación, mostrando la misma exigencia en las sucesivas revisiones de este documento. Sin su ayuda este trabajo no se hubiera podido completar.

A los profesores de la Facultad de Informática, que han venido contribuyendo a mi formación desde aquella lejana segunda promoción de la diplomatura, hasta la más reciente del máster. Cada uno de ellos ha dejado una huella, una pincelada de conocimientos y generado unas incansables ganas de aprender.

Gracias a mis padres, que siempre me proporcionaron el apoyo y los medios necesarios para mi educación. Y especialmente a Charo, mi esposa, y a mis hijas, Ana, Belén y Ángela, que han contribuido a este trabajo desde su inicio con un derroche sin igual de paciencia, creando en el hogar un entorno de estudio sin el que no es posible afrontar una tarea como la culminada aquí. Por no pedir nada a cambio del tiempo robado, gracias de corazón.

¡Gracias a todos!

Índice general

1	Introducción	5
1.1	Contexto	5
1.2	Estado del arte	9
1.2.1	Modelado, análisis y simulación de sistemas mecánicos . .	10
1.2.2	Simuladores para la dinámica de sistemas mecánicos . . .	15
1.2.3	Arquitecturas de hardware paralelo	23
1.2.4	Programación paralela	25
1.2.5	Librerías de álgebra lineal	28
1.2.6	Paralelismo en simuladores de sistemas mecánicos	30
1.2.7	Autooptimización	35
1.3	Objetivos	36
1.4	Metodología	39
1.5	Contribuciones	41
1.6	Estructura de la tesis	41
2	Cinemática computacional de sistemas multicuerpo	43
2.1	Modelado de sistemas multicuerpo	44
2.1.1	Conceptos básicos topológicos	45
2.1.2	Modelado en coordenadas dependientes	50
2.1.2.1	Coordenadas relativas	51
2.1.2.2	Coordenadas de punto de referencia	52
2.1.2.3	Coordenadas naturales	53
2.2	Análisis estructural de sistemas multicuerpo	55
2.2.1	Grafo estructural	56
2.2.2	Estructura cinemática	57

2.3	Cinemática computacional de sistemas multicuerpo	60
2.3.1	Análisis de posición	60
2.3.2	Análisis de velocidades	62
2.3.3	Análisis de aceleraciones	63
2.3.4	Rutina general para el análisis cinemático computacional .	64
2.3.5	Formulación cinemática basada en ecuaciones de grupo . .	66
2.4	Conclusiones	69
3	Herramientas computacionales	71
3.1	Herramientas hardware	71
3.2	Herramientas software	80
3.2.1	Compiladores	80
3.2.2	Librerías de álgebra lineal	81
3.2.3	Componentes auxiliares	86
3.3	Conclusiones	87
4	Optimización y Autooptimización	89
4.1	Ideas generales	89
4.2	Optimización de software científico	90
4.2.1	Supervisión de rendimientos	91
4.2.2	Modelado de rendimientos	93
4.2.3	Optimización	93
4.2.3.1	Optimización a nivel de aplicaciones	94
4.2.3.2	Optimización de compilación	95
4.2.3.3	Datos históricos como fuente de información . .	97
4.2.3.4	Parámetros ajustables y espacios de búsqueda . .	98
4.2.3.5	Algoritmos de búsqueda	99
4.2.4	Autooptimización de rutinas paralelas de álgebra lineal . .	99
4.2.5	Ejemplo de aplicación: optimización de un modelo de pre- dicción climático	101
4.3	Optimización de códigos de simulación de sistemas multicuerpo .	103
4.3.1	Características de los algoritmos	103
4.3.2	Autooptimización en el simulador	104
4.4	Conclusiones	106

5	Simulador	107
5.1	Motivación	108
5.2	Conceptos básicos	109
5.2.1	Funciones	110
5.2.2	Rutinas de usuario	116
5.2.3	Modelos	120
5.2.4	Grupos	121
5.2.5	Parámetros	124
5.2.6	Escenarios	125
5.2.7	Scripts	128
5.2.8	Resumen	130
5.3	Rutas	131
5.4	Base de datos	135
5.5	Árbol de rutas	137
5.5.1	Propiedades del árbol de rutas definido en el simulador . .	139
5.5.2	Generación del árbol de rutas	142
5.5.3	Tiempo de ejecución de una ruta	147
5.5.4	Estimación de tiempos	149
5.5.4.1	Estimación del tiempo de ejecución de un nodo con parámetros algorítmicos preestablecidos . . .	152
5.5.4.2	Estimación del tiempo de ejecución de un nodo con autooptimización de parámetros algorítmicos	158
5.6	Simulación	164
5.6.1	Configuración del simulador	164
5.6.2	Modos de simulación	167
5.6.2.1	Modo de entrenamiento	168
5.6.2.2	Modo simple	170
5.6.2.3	Modo múltiple	171
5.6.2.4	Modo autooptimizado	173
5.6.3	Informe de ejecución	174
5.7	Herramientas	176
5.7.1	Análisis de resultados: vista tabular	176
5.7.2	Análisis de resultados: vista gráfica	177

5.7.3	Autooptimizador interactivo	180
5.8	Arquitectura del software	183
5.9	Conclusiones	185
6	Experimentos	187
6.1	Aplicación del simulador al análisis cinemático de sistemas multi- cuerpo	188
6.1.1	Plataforma de Stewart	188
6.1.1.1	Plataforma de Stewart: resolución secuencial	194
6.1.1.2	Plataforma de Stewart: solución global	205
6.1.1.3	Plataforma de Stewart: resolución paralela	208
6.1.1.4	Plataforma de Stewart: ejecución autooptimizada	212
6.1.2	Modelo de suspensión de un camión	218
6.1.2.1	Suspensión de un camión: resolución secuencial	221
6.1.2.2	Suspensión de un camión: resolución paralela	224
6.1.2.3	Suspensión de un camión: escalado a un modelo multieje	228
6.2	Aplicación del simulador a la optimización de rutinas de álgebra lineal	234
6.2.1	Multiplicación de matrices por bloques	234
6.2.1.1	Experimentos en sistemas con CPU multicore	236
6.2.1.2	Experimentos en sistemas multiGPU	242
6.2.2	Multiplicación de matrices: algoritmo de Strassen	245
6.2.2.1	Experimentos en sistema con CPU monocore	250
6.2.2.2	Experimentos en sistema con CPU multicore	255
6.3	Conclusiones	257
7	Conclusiones, trabajo futuro y contribuciones	259
7.1	Conclusiones	259
7.2	Trabajo futuro	262
7.3	Difusión	267
A	Simulador: Manual de uso	271
A.1	Conceptos básicos	271

A.1.1	Funciones	271
A.1.2	Rutinas de usuario	272
A.1.3	Modelos	272
A.1.4	Grupos	272
A.1.5	Variables	273
A.1.6	Escenarios	273
A.1.7	Scripts	273
A.1.8	Rutas	273
A.2	Componentes del software	274
A.3	Requisitos del sistema	274
A.4	Interfaz gráfico de PARCSIM-MB: vista general	274
A.5	Barra de menús	276
A.5.1	Menú <i>File</i>	277
A.5.2	Menú <i>Model</i>	277
A.5.3	Menú <i>Results</i>	278
A.5.4	Menú <i>View</i>	279
A.5.5	Menú <i>Training</i>	279
A.6	Área de trabajo	280
A.6.1	Área de trabajo: <i>Variables</i>	280
A.6.2	Área de trabajo: <i>Groups</i>	282
A.6.3	Área de trabajo: <i>Scenarios</i>	284
A.6.4	Área de trabajo: <i>Routines</i>	285
A.6.5	Área de trabajo: <i>Scripts</i>	286
A.6.6	Área de trabajo: <i>Config</i>	287
A.7	Visor del grafo de modelos	290
A.8	Lección paso a paso	291
A.8.1	Preparación: elección de un problema	291
A.8.2	Creación del modelo	293
A.8.3	Creación del primer grupo del modelo	295
A.8.4	Creación del segundo grupo	297
A.8.5	Creación de una rutina	297
A.8.6	Asignar la nueva rutina ADD_SYS al Grupo1	300
A.8.7	Relación entre grupos: asignación de dependencias	301

A.8.8	Guardar los cambios	302
A.8.9	Creación del resto de grupos	303
A.8.10	Construcción del resto del grafo	306
A.8.11	Creación del resto de rutinas	308
A.8.12	Comprobación del modelo	308
A.8.13	Crear las variables del modelo	310
A.8.14	Crear las variables de tamaño	311
A.8.15	Asociar variables de tamaño a variables de modelo	314
A.8.16	Asignación de variables de modelo a argumentos	316
A.8.17	Rutas	318
A.8.18	Creación de un escenario	321
A.8.19	Validación de escenarios	325
A.8.20	Creación de un script	326
A.8.21	Preparación del simulador: Ubicación de los ficheros de configuración	331
A.8.22	Preparación del simulador: Control de la simulación	333
A.8.23	Preparación del simulador: Modos de simulación	334
A.8.23.1	Modo de entrenamiento	334
A.8.23.2	Modo simple	337
A.8.23.3	Modo múltiple	337
A.8.23.4	Modo autooptimizado	338
A.8.24	Guardar la configuración del simulador	339
A.8.25	Simulación	339
A.8.25.1	Primera ejecución: Modo múltiple con ruta seleccionada	340
A.8.25.2	Consulta de resultados	344
A.8.25.3	Consulta gráfica de tiempos de ejecución	346
A.8.25.4	Segunda ejecución: Modo múltiple con ruta calculada	347
A.8.25.5	Visor de rutas interactivo	349
A.8.26	Funciones avanzadas	352
A.8.26.1	Enlace de argumentos en rutinas	352
A.8.26.2	Ejemplo de anidamiento de rutinas	353
A.8.26.3	Importación de modelos	354

A.8.26.4 Inserción de modelos dentro de grupos	356
A.9 Preguntas frecuentes	360
A.10 Software de terceros	361
A.11 Datos de contacto	363
Bibliografía	365

Índice de Tablas

2.1	Grados de libertad que eliminan los pares cinemáticos que forman los eslabones del sistema mecánico según su grado k	50
3.1	Cluster <i>Heterosolar</i> : nodos de cómputo y especificaciones del hardware	79
5.1	Listado de librerías incorporadas en el simulador y funciones de álgebra lineal que implementan. Se incluye información sobre el tipo de matrices soportadas en cada caso.	111
5.2	Lista de funciones de álgebra lineal incorporadas en el simulador y sus identificadores.	113
5.3	Tipos de matrices manejados por el simulador.	125
5.4	Regla para determinar mediante los campos <code>modelId</code> , <code>groupId</code> y <code>functionId</code> en la base de datos <code>PARCSIM.db</code> el nivel (modelo, grupo o función) en el que se ha medido el tiempo de ejecución almacenado en un registro.	136
5.5	Contenido de la base de datos de entrenamiento para las funciones f_3 , f_4 y f_5 , obtenidos con la librería MKL operando sobre matrices de tamaño 100×100	157
5.6	Contenido de la base de datos de entrenamiento para las funciones f_3 , f_4 y f_5 , obtenidos con la librería MKL usando matrices de tamaños 50×50 y 200×200	157
5.7	Contenido de la base de datos de entrenamiento para las funciones f_3 , f_4 y f_5 con matrices de tamaños 100×100	162

5.8	Mejores parámetros algorítmicos <i>AP</i> aplicables al nodo que resuelve simultáneamente los grupos <i>Gr</i> ₁ , <i>Gr</i> ₂ y <i>Gr</i> ₄ . La información se obtiene desglosada por función ejecutada, a partir de la información de entrenamiento mostrada en la tabla 5.7.	162
6.1	Escenarios definidos para la simulación de una plataforma de Stewart que definen el tipo, factor de dispersión y los tamaños de las matrices asociados a la dimensión del terminal, nEQ_T, y a los grupos manivela-barra, nEQ_MB. El terminal, al estar unido siempre a seis manivelas, mantiene fijo el número de coordenadas en todos los escenarios.	195
6.2	Script definido para guiar el primer experimento de simulación de una plataforma de Stewart. Se asigna un único thread al primer nivel de paralelismo OpenMP al tratarse de una ejecución en secuencia de los grupos estructurales. El número de threads que se pone a disposición de la librería MKL se limita al número de cores físicos en JUPITER (12).	196
6.3	Configuración aplicable a una ejecución del simulador para el modelo de la plataforma de Stewart. El modo de ejecución múltiple genera varias ejecuciones en función del contenido del script. En este experimento la ruta preseleccionada es la que realiza la ejecución en secuencia de los grupos.	198
6.4	Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWART) obtenidos en la plataforma JUPITER con varios tamaños de matrices (nEQ_MB) y variando el número de hilos asignados a MKL. Matrices simétricas con dispersión del 80%.	198
6.5	Script definido para guiar el primer experimento de simulación de un modelo de Stewart en la plataforma SATURNO. Se limita el número de threads asignados al paralelismo interno de la librería MKL al número de cores físicos del hardware (24).	199

6.6	Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWARD) obtenidos en la plataforma SATURNO con varios tamaños de matrices (nEQ_MB) y variando el número de hilos asignados a MKL. Matrices simétricas con dispersión del 80%.	199
6.7	Script definido para la simulación de una plataforma de Stewart en JUPITER empleando la librería PARDISO. Se asigna un thread al primer nivel de paralelismo OpenMP (ejecución en secuencia de los grupos estructurales). El número de threads a disposición de la librería se limita al número de cores (12).	200
6.8	Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWARD) obtenidos en la plataforma JUPITER con varios tamaños de matrices (nEQ_MB) y variando el número de hilos asignados a PARDISO. Matrices simétricas con dispersión del 80%	201
6.9	Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWARD) obtenidos en la plataforma SATURNO con varios tamaños de matrices (nEQ_MB) y variando el número de hilos asignados a PARDISO. Matrices simétricas con dispersión del 80%	202
6.10	Script definido para guiar el experimento de simulación del modelo de Stewart con todas las librerías incluidas en la versión actual del simulador. Se reserva un único thread al primer nivel de paralelismo OpenMP al tratarse de una ejecución en secuencia de los grupos estructurales. Se habilita una GPU para ser usada por la librería MAGMA (valor 1) y se desactiva el segundo nivel de paralelismo del resto de librerías (valor 1).	202
6.11	Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWARD) obtenidos en la plataforma JUPITER con varios tamaños de matrices (nEQ_MB) y empleando diferentes librerías sin explotación de paralelismo implícito, y MAGMA explotando una GPU. Matrices simétricas con dispersión del 80%.	203

6.12	Escenarios definidos para la simulación de una plataforma de Stewart que especifican matrices banda, un factor dispersión del 30% y diferentes tamaños de las matrices asociados a la dimensión del terminal, <code>nEQ_T</code> , y a los grupos manivela-barra, <code>nEQ_MB</code>	204
6.13	Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (<code>MBS-STEWART</code>) obtenidos en la plataforma JUPITER con varios tamaños de matrices (<code>nEQ_MB</code>) y empleando diferentes librerías sin explotación de paralelismo implícito. Matrices simétricas con dispersión del 30%.	204
6.14	Escenarios definidos para la simulación de una plataforma de Stewart que definen el tipo, factor de dispersión y los tamaños de las matrices para la solución global, <code>nEQ_Global</code> . Se incluyen como referencia la dimensión del terminal, <code>nEQ_T</code> , y de los grupos manivela-barra, <code>nEQ_MB</code> , usados en la formulación por grupos estructurales.	206
6.15	Script definido para guiar el experimento de simulación de una plataforma de Stewart siguiendo una formulación global usando 4 cores de la plataforma JUPITER.	207
6.16	Comparación de los tiempos de ejecución de la solución global del modelo de la plataforma de Stewart (<code>MBS-STEWART</code>) obtenidos en la plataforma JUPITER con varios tamaños de matrices (<code>nEQ_Global</code>) y variando el número de hilos asignados a MKL. Matrices simétricas con dispersión del 80%.	207
6.17	Script definido para guiar la simulación del modelo de Stewart usando las librerías MKL y PARDISO. Se asignan tres threads al primer nivel de paralelismo para permitir la resolución simultánea de tres grupos estructurales. El número de threads que se pone a disposición de las librerías se limita al número de cores físicos de JUPITER (12).	209
6.18	Tiempos de ejecución obtenidos al combinar threads OpenMP y threads asignados a las librerías MKL y PARDISO, con varios tamaños de matrices (<code>nEQ_MB</code>) para resolver seis grupos estructurales Manivela-Barra en dos etapas. Hardware JUPITER con 12 cores, con matrices simétricas con dispersión del 80%.	210

6.19	Tiempos de ejecución obtenidos al combinar threads OpenMP y threads asignados a las librerías MKL y PARDISO, con varios tamaños de matrices (nEQ_MB) para resolver seis grupos estructurales Manivela-Barra en dos etapas. Hardware SATURNO con 24 cores, con matrices simétricas con dispersión del 80%.	210
6.20	Comparación de los tiempos de ejecución obtenidos con MKL, PARDISO y MA27 en SATURNO con 24 cores, para matrices simétricas de tamaños 30×30 y 3000×3000 y un factor de dispersión del 80%. Se muestran todas las combinaciones de threads asignados al primer y segundo nivel de paralelismo.	211
6.21	Script definido para guiar la construcción del conjunto de entrenamiento en la plataforma hardware SATURNO. Se obtendrán datos de ejecución con las siete librerías incluidas en el simulador, con asignaciones de threads desde 1 (sin paralelismo implícito) hasta 24 (cores físicos de la plataforma SATURNO).	213
6.22	Escenarios definidos para usar durante el entrenamiento de las funciones. Se definen la tipología de las matrices, su factor de dispersión y los tamaños de las matrices, nEQ	213
6.23	Preparación del simulador para una ejecución de entrenamiento en la plataforma SATURNO.	214
6.24	Configuración aplicable a la siguiente ejecución del simulador para el modelo de la plataforma de Stewart. El modo de ejecución autooptimizado selecciona de manera automática la rama de ejecución y los valores de los parámetros algorítmicos que ofrecen los menores tiempos de ejecución teóricos, buscando el máximo aprovechamiento de los recursos del hardware.	214
6.25	Mejores combinaciones de ejecución de la plataforma de Stewart en SATURNO para matrices de 3000×3000 y 80% de dispersión, frente a la obtenida en una ejecución autooptimizada.	216
6.26	Escenarios definidos para la simulación de la suspensión de un eje de un camión que describen los tamaños de las matrices nEQ , su tipología y factor de dispersión.	222

6.27	Script creado para el experimento de simulación de la suspensión de un camión. Se reserva un thread al primer nivel de paralelismo OpenMP para una ejecución en secuencia de los grupos. Los threads asignados a las librerías MKL y PARDISO se limitan al número de cores físicos de SATURNO (24).	222
6.28	Configuración del simulador para la ejecución del modelo de la suspensión de un camión en modo múltiple y con una ruta preseleccionada.	222
6.29	Comparación de los tiempos de ejecución del modelo de la suspensión (MBS-TRUCK) obtenidos en la plataforma SATURNO con varios tamaños de matrices (nEQ) y variando el número de hilos asignados a MKL. Matrices simétricas con dispersión del 80%. . .	223
6.30	Comparación de los tiempos de ejecución del modelo de la suspensión (MBS-TRUCK) obtenidos en la SATURNO con varios tamaños de matrices (nEQ) y variando el número de hilos asignados a PARDISO. Matrices simétricas con dispersión del 80%.	223
6.31	Comparación de los mejores tiempos de ejecución del modelo de la suspensión (MBS-TRUCK) obtenidos en la plataforma SATURNO con varios tamaños de matrices (nEQ) variando la asignación de threads a las librerías MKL y PARDISO frente a MA27 en modo secuencial. Matrices simétricas con dispersión del 80%.	224
6.32	Script creado para la simulación del modelo de la suspensión empleando las librerías MKL y PARDISO. Se asignan dos threads al primer nivel de paralelismo para permitir la resolución simultánea de dos grupos. El número de threads que se pone a disposición del paralelismo implícito de las librerías se limita al número de cores físicos de SATURNO (24).	225
6.33	Tiempos de ejecución obtenidos al combinar threads OpenMP y threads asignados a las librerías MKL y PARDISO, con varios tamaños de matrices (nEQ) para resolver el modelo MBS-TRUCK mediante dos rutas con paralelismo de grupos. Hardware SATURNO con 24 cores físicos. Matrices simétricas con dispersión del 80%. .	226

6.34	Tiempos de ejecución obtenidos al simular el modelo MBS-TRUCK con MKL mediante el cálculo simultáneo de cuatro grupos estructurales. Hardware SATURNO con 24 cores, con varios tamaños de matrices con dispersión del 80%.	228
6.35	Tiempos de ejecución del modelo MBS-TRUCK, suspensión de un eje, obtenidos con MKL en la plataforma SATURNO usando hasta 24 cores con varios tamaños de matrices y diferentes combinaciones de paralelismo. Matrices simétricas con dispersión del 80%.	230
6.36	Tiempos de ejecución del modelo MBS-TRUCK, suspensión de un eje, con MKL en SATURNO para varios tamaños de matrices (nEQ) y combinaciones de paralelismo hasta ocho cores. Matrices simétricas con dispersión del 80%.	233
6.37	Escenarios creados para la resolución de una multiplicación de matrices. Los escenarios definen la tipología de las matrices, dispersión y tamaños. nROWS y nCOLS indican el número de filas y columnas, respectivamente.	236
6.38	Comparación de los tiempos de ejecución del modelo de la multiplicación de matrices (MATMUL) obtenidos en JUPITER con varios tamaños de matrices (nROWS×nCOLS) y variando el número de hilos asignados a MKL.	236
6.39	Comparación de los tiempos de ejecución del modelo de la multiplicación de matrices (MATMUL) obtenidos en SATURNO con varios tamaños de matrices (nROWS×nCOLS) y variando el número de hilos asignados a MKL.	237
6.40	Configuración aplicable a la ejecución de tres modelos de multiplicación de matrices por bloques, con una ruta preseleccionada. . . .	241
6.41	Comparación de los tiempos de ejecución obtenidos en SATURNO al simular los modelos de la multiplicación de matrices tradicional (MATMUL) y por bloques (COLS50, COLS35, COLS20), para varios tamaños de matrices (nROWS).	242

6.42	Script creado para la simulación del modelo de la multiplicación de matrices por bloques empleando las librerías MKL y MAGMA. Se asignan tres threads al paralelismo OpenMP para la resolución simultánea de los grupos, tres GPUs para MAGMA y hasta cuatro threads para MKL. Plataforma JUPITER con 12 cores físicos. . .	244
6.43	Comparación de los tiempos de ejecución del modelo de la multiplicación de matrices por bloques (MATMULT_COLS_35) obtenidos en JUPITER con varios tamaños de matrices (nROWS×nCOLS), variando el número de hilos asignados a MKL y con tres GPUs explotadas con MAGMA.	244
6.44	Desglose por grupos de los tiempos de ejecución del modelo de la multiplicación de matrices por bloques (MATMULT_COLS_35) en JUPITER con varios tamaños de matrices (nROWS×nCOLS), variando el número de hilos asignados a MKL y con tres GPUs explotadas con MAGMA.	245
6.45	Configuración aplicable a la ejecución del modelo de la multiplicación mediante Strassen. En este experimento la ruta preseleccionada ejecuta en secuencia los grupos, asignando un único core a la librería MKL.	251
6.46	Tiempos de ejecución (en segundos) obtenidos con el simulador en una multiplicación sin bloques (modelo MATMUL) y mediante el algoritmo de Strassen (modelo STRASSEN). Matrices cuadradas con dispersión del 30% en las plataformas JUPITER y SATURNO, empleando un único core y MKL.	251
6.47	Configuración aplicable a la ejecución de los modelos de la multiplicación sin bloques y Strassen. Se preselecciona la ruta secuencial, sin paralelismo de grupos, y usando una GPU a través de la librería MAGMA.	252
6.48	Tiempos de ejecución (en segundos) obtenidos con el simulador en una multiplicación sin bloques (modelo MATMUL) y mediante el algoritmo de Strassen (modelo STRASSEN). Plataformas JUPITER y SATURNO empleando una GPU mediante la librería MAGMA.	252

6.49	Tiempos de ejecución (en segundos) obtenidos con el simulador en una multiplicación tradicional (modelo MATMUL) y mediante el algoritmo de Strassen sin recursión (R0) y con un nivel de recursión (R1). Matrices cuadradas con dispersión del 30% en las plataformas hardware JUPITER y SATURNO empleando un único core.	255
6.50	Configuración aplicable a la ejecución del modelo de la multiplicación mediante Strassen. En este experimento la ruta preseleccionada es la secuencial, sin paralelismo de grupos, empleando dos y tres threads en las llamadas a la multiplicación de matrices DGEMM de la librería MKL.	255
6.51	Tiempos de ejecución (en segundos) obtenidos con el simulador para una multiplicación tradicional sin bloques (modelo MATMUL) y mediante el algoritmo de Strassen (modelo STRASSEN). Plataforma hardware SATURNO, asignando dos y tres threads.	256
6.52	Tiempos de ejecución (en segundos) obtenidos con el simulador en una multiplicación tradicional sin bloques (modelo MATMUL) y mediante el algoritmo de Strassen (modelo STRASSEN). Plataforma hardware JUPITER, asignando dos y tres threads.	256
6.53	Tabla que recoge las mejores configuraciones de ejecución de una rutina que resuelve tres grupos para varios tamaños de matrices y factores de dispersión, obtenidas por medio de simulaciones usando las librerías MKL y PARDISO.	258
A.1	Variables del modelo para el algoritmo de la figura A.10(b). Dos variables de tamaño (sizeMat y sizeOne) son suficientes para establecer las dimensiones de las matrices. Se muestran dos escenarios con diferentes valores de sizeMat.	313
A.2	Combinaciones de valores obtenidos por el simulador a partir de la información introducida en el editor de PARCSIM en los parámetros OMP1 {1,...,4} y OMP2 {1,3}, threads OpenMP y threads asignados al paralelismo de las librerías, respectivamente.	329

Índice de figuras

1.1	Fases requeridas durante el proceso de modelado y simulación de un sistema multicuerpo.	14
1.2	Librerías matriciales clásicas y sus dependencias.	29
2.1	Ejemplo de sistema mecánico: mecanismo biela-cigüeñal.	47
2.2	Identificación de pares cinemáticos en un sistema biela-cigüeñal. .	47
2.3	Tipos de cadenas cinemáticas en función de la unión entre sus eslabones.	48
2.4	Tipos de cadenas cinemáticas según la clase de movimiento. . . .	49
2.5	Modelado empleando coordenadas relativas.	51
2.6	Modelado empleando coordenadas de punto de referencia.	53
2.7	Modelado empleando coordenadas naturales.	53
2.8	Ejemplo de sistema multicuerpo en 3 dimensiones.	54
2.9	Ejemplo de análisis estructural de un mecanismo.	55
2.10	Esquema de un cuadrilátero articulado.	57
2.11	Análisis de un cuadrilátero articulado: grafo estructural.	57
2.12	Análisis basado en el grafo: identificación de siguientes grupos sobre pares de mayor longitud.	59
2.13	Análisis de un cuadrilátero articulado: diagrama estructural. . . .	60
2.14	Mecanismo cuadrilátero articulado usado para mostrar las rutinas de análisis cinemático basado en análisis de grupo.	67
3.1	Diagrama de bloques de un sistema informático típico de CPU multinúcleo donde todos los núcleos comparten una caché L2. . .	72
3.2	Diagrama de bloques de un multiprocesador vagamente acoplado.	73

3.3	Diagrama de bloques de una arquitectura típica de CPU múltiple / núcleos múltiples (con dos CPU de cuatro núcleos cada una), donde las memorias distribuidas se ven desde las CPU como una memoria unificada, accesible con paradigma de compartición de memoria (arquitectura NUMA).	74
3.4	Vista esquemática de la arquitectura NVIDIA GPU, donde SM se refiere a <i>Streaming Multiprocessors</i> y SP a <i>Streaming Processors</i> . .	75
3.5	Vista esquemática del modelo de programación en CUDA.	76
3.6	Estructura del cluster <i>Heterosolar</i>	78
5.1	Representación mediante un grafo dirigido de la división de un sistema en un conjunto ordenado de subsistemas donde los nodos representan conjuntos de instrucciones. Según se observa en el grafo, los nodos 2 y 3 podrían resolverse simultáneamente.	108
5.2	Ejemplo de dos rutinas de usuario que ilustran el concepto de rutina simple (compuesta únicamente por funciones) y rutina anidada (compuesta por funciones y por otras rutinas). En tiempo de ejecución la rutina anidada aporta a la secuencia de cálculos las funciones que la forman.	116
5.3	Rutinas de ejemplo usadas para ilustrar el modo en el que se representan en el fichero <code>routines.rou</code>	119
5.4	Modelo de ejemplo formado por siete grupos. En cada grupo se muestran las instrucciones o funciones, ya desglosadas, a partir de las rutinas que definen la solución del subsistema o grupo al que representan. Las líneas dirigidas indican las dependencias entre los grupos.	121
5.5	Esquema de la relación grupo-rutina-parámetros. El grupo Gr_1 ejecuta la rutina RADDMUL que contiene dos funciones. Cada función requiere tres parámetros, por lo que Gr_1 , por incluir esta rutina, debe aportar los seis parámetros $p1, \dots, p6$	124
5.6	Vista esquemática de los conceptos usados en el simulador y sus interdependencias.	130
5.7	Grafo de resolución de un problema numérico mediante descomposición en subproblemas.	131

5.8	Cronograma de resolución del problema representado en la figura 5.7. Cada e_i representa una de las etapas en las que se divide la ejecución.	132
5.9	Cronograma de resolución del problema representado en la figura 5.7 retrasando la ejecución del grupo Gr_4 para resolverlo en paralelo con el Gr_3 . Cada e_i representa una de las etapas en las que se divide la ejecución.	132
5.10	Árbol que muestra las secuencias de cálculo válidas para el problema de la figura 5.7. Cada rama es una ruta que representa un modo de ordenar los cálculos capaz de resolver el sistema en su totalidad. Los nodos del árbol que contienen más de un grupo suponen la resolución simultánea de dichos grupos.	133
5.11	Ejemplo de codificación de los grupos en una ruta en PARCSIM. El símbolo + separa los identificadores de los grupos (groupId) que se calculan en paralelo en una misma etapa de tiempo, y el símbolo — separa etapas de tiempo. La imagen muestra la ruta 5 del árbol mostrado en la figura 5.10.	134
5.12	Estructura de la tabla Results contenida en la base de datos PARCSIM.db para la persistencia de los tiempos de ejecución obtenidos durante las simulaciones.	136
5.13	Resolución simultánea de tres grupos con diferente carga computacional asignando un thread a cada grupo. El tiempo total de ejecución corresponde al retraso provocado por el grupo Gr_3 , de cuatro unidades de tiempo.	138
5.14	Modificación de la estrategia de scheduling respecto a la mostrada en la figura 5.13. La ejecución aislada en otra etapa del grupo Gr_3 permite asignar un mayor número de threads a Gr_1 y a Gr_3 , consiguiendo mejoras en los tiempos de ejecución de ambos grupos, y una mejora global de una unidad de tiempo.	138
5.15	Modelo con cinco grupos, donde Gr_1 , Gr_2 y Gr_3 contienen la misma rutina para resolver una suma de matrices.	139
5.16	Árbol de rutas que permiten resolver el modelo de la figura 5.15 cuando los grupos Gr_1 , Gr_2 y Gr_3 operan sobre matrices de diferentes tamaños, de 100×100 , 200×200 y 300×300	140

5.17	Árbol de rutas que permiten resolver el modelo de la figura 5.15 en un escenario en el que los grupos Gr_1 y Gr_3 operan sobre matrices de tamaños 100×100 , mientras que el Gr_2 lo hace sobre matrices 200×200 . La rama 4 puede eliminarse, pues es equivalente a la rama 2.	140
5.18	Árbol de rutas que permiten resolver el modelo de la figura 5.15 en un escenario en el que los grupos Gr_1 , Gr_2 y Gr_3 operan sobre matrices de tamaños 100×100 . Las ramas 3 y 4 pueden eliminarse por ser equivalentes a la rama 2.	141
5.19	Búsqueda de nodos hijo para el nodo raíz que contiene al grupo <i>Start</i> . La zona sombreada muestra algunos de los nodos añadidos al árbol. En esta figura los nodos que contienen a Gr_3 y a Gr_5 son ejemplos de algunos nodos que se descartan por no considerarse válidos (necesitan que otros grupos se resuelvan antes que ellos). .	145
5.20	Búsqueda de nodos hijo para el nodo que contiene al grupo Gr_1 . Se descartan nodos que no cumplen con las precedencias del grafo del modelo.	146
5.21	Búsqueda de nodos hijo para el nodo que contiene al grupo Gr_2 . El nodo que contiene al Gr_5 no es válido pues dicho grupo necesita que se hayan resuelto antes Gr_3 y Gr_4 en esta rama.	146
5.22	Ejemplo de ruta que resuelve un determinado problema numérico. La ruta contiene cuatro nodos. El nodo N_2 incluye la resolución de dos grupos.	148
5.23	Modelo formado por siete grupos con las funciones ya desglosadas a partir de las rutinas que definen la solución de cada grupo. . . .	153
5.24	Representación de una de las rutas que resuelve el modelo de la figura 5.23 mediante la ejecución simultánea de los grupos Gr_1 , Gr_2 y Gr_4	154
5.25	Algoritmo de ejecución del simulador en el modo de entrenamiento. Se miden los tiempos de ejecución de todas las funciones para una serie de escenarios <i>SCN</i> y parámetros algorítmicos <i>AP</i> especificados por el usuario.	169

5.26	Algoritmo de ejecución del simulador en el modo simple sobre un conjunto de escenarios, donde el usuario fija los parámetros algorítmicos <i>AP</i>	170
5.27	Algoritmo de ejecución del simulador en el modo múltiple, con ejecuciones sobre un conjunto de escenarios <i>SCN</i> y con parámetros algorítmicos <i>AP</i> generados a partir de los ficheros de scripts. . . .	172
5.28	Algoritmo de ejecución del simulador en el modo autooptimizado, donde el software calcula los parámetros algorítmicos <i>AP</i> y la ruta R_x que mejor se adaptan al hardware disponible.	173
5.29	Información producida por el software durante la simulación, con los tiempos de ejecución ordenados de menor a mayor.	175
5.30	Consulta de información de los tiempos de ejecución en formato tabular.	176
5.31	Consulta de información de los tiempos de ejecución en formato tabular aplicando un filtro para obtener datos de una función específica.	177
5.32	Gráfico de tiempos de ejecución obtenidos al simular un modelo con un tipo de matrices seleccionado por el usuario. Se obtiene una serie para cada combinación de parámetros algorítmicos. . . .	178
5.33	Análisis del tiempo de ejecución de una función en el simulador. El gráfico muestra información de entrenamiento para diferentes parámetros algorítmicos y tamaños de matrices.	179
5.34	Representación en la interfaz gráfica GUI del simulador de una resolución de dos multiplicaciones de matrices.	180
5.35	Vista del modelo de la figura 5.34(a) incluyendo información de los escenarios.	181
5.36	Herramienta de autooptimización: mejor ruta y librería de cómputo propuestas para una plataforma con dos cores y un escenario seleccionado.	182
5.37	Herramienta de autooptimización: las dos mejores rutas para una plataforma con dos cores y un escenario seleccionado.	182
5.38	Herramienta de autooptimización: mejor ruta y librería de cómputo propuestas para plataformas con dos cores y una GPU, para un escenario concreto seleccionado.	183

5.39	Arquitectura a alto nivel del software PARCSIM.	184
6.1	Plataforma de Stewart.	189
6.2	Representación en el simulador de la rutina SG_KINEM_REC de resolución de un sistema mecánico con una configuración REC, formada por una junta de rotación (R), una esférica (E) y una cardan (C), válida para resolver un grupo manivela-barra de una plataforma de Stewart.	190
6.3	Representación en el simulador de la rutina SG_KINEM_6C, correspondiente a seis juntas de tipo cardan (6C), para la resolución de un sistema mecánico con la topología del terminal de una plataforma de Stewart.	191
6.4	Representación gráfica en el simulador del modelo MBS-STEWART para la resolución de una plataforma de Stewart.	192
6.5	Árbol de rutas que representa las alternativas de ordenación y agrupación de cálculos para resolver una plataforma de Stewart como la representada en el modelo de la figura 6.4.	193
6.6	Ruta que resuelve de manera secuencial todos los grupos en los que se descompone una plataforma de Stewart como la representada en el modelo de la figura 6.4.	194
6.7	Información obtenida por consola al finalizar las simulaciones en JUPITER, con tamaños de matrices (nEQ_MB) de 3000×3000 y variando el número de hilos asignados a MKL. El informe ordena de menor a mayor los tiempos de ejecución obtenidos.	197
6.8	Speed-up respecto a MKL secuencial variando el número de hilos asignados a MKL, para diversos tamaños de matrices. Matrices simétricas con dispersión del 80%.	200
6.9	Speed-up respecto a PARDISO secuencial variando el número de hilos asignados a la librería, para diversos tamaños de matrices. Plataforma hardware JUPITER. Matrices simétricas con dispersión del 80%.	201
6.10	Representación gráfica del modelo MBS-STEWART-GLOBAL para la resolución de una plataforma de Stewart sin división por grupos estructurales.	206

6.11	Speed-up de la formulación basada en ecuaciones de grupo frente a la global, variando el número de threads asignados a la librería MKL en JUPITER, para diversos tamaños de matrices. Matrices simétricas con dispersión del 80%.	208
6.12	La ruta resaltada introduce paralelismo en la resolución de una Plataforma de Stewart como la representada en el modelo de la figura 6.4.	208
6.13	Consulta de información de entrenamiento mediante el visor incorporado en la aplicación. Se muestran, filtrados, los tiempos de ejecución obtenidos en SATURNO por la librería 1 (MKL) al manipular matrices de tamaño 3000×3000 , simétricas y con un factor de dispersión del 80% y para cada valor de threads asignados al paralelismo interno.	215
6.14	Información incluida en el LOG de ejecución del simulador en modo autooptimizado en el que podemos encontrar, además del tiempo de ejecución obtenido, la ruta seleccionada y las rutinas asignadas a cada grupo. Escenario con matrices de tamaño 3000×3000 , simétricas y con un factor de dispersión del 80%, en la plataforma SATURNO.	217
6.15	Sistema mecánico que representa el sistema de suspensión de un camión.	218
6.16	Representación en el simulador del modelo MBS-TRUCK de resolución de un mecanismo de suspensión de un camión.	219
6.17	Árbol de rutas que representa las alternativas de ordenación y agrupación de cálculos para resolver un sistema mecánico de suspensión de un eje de un camión como el representado en el modelo de la figura 6.16.	220
6.18	Ruta que resuelve en secuencia todos los grupos en los que se descompone el modelo de la suspensión de un eje de un camión como el representado en la figura 6.16.	221
6.19	Las rutas resaltadas introducen paralelismo en la resolución de la suspensión de un camión como la representada en el modelo de la figura 6.16.	224

6.20	Ruta de resolución de la suspensión de un camión representada en el modelo de la figura 6.16 mediante el cálculo simultáneo de cuatro grupos.	227
6.21	Sistema de suspensión en una cabeza tractora de tres ejes.	228
6.22	Ruta para la resolución en secuencia de los tres ejes de un sistema de suspensión multieje.	229
6.23	Ruta para la resolución simultánea de los tres ejes de un sistema de suspensión multieje.	229
6.24	Ruta de solución en secuencia del modelo de tres ejes, resolviendo en paralelo los cuatro componentes paralelizables de cada eje. . .	231
6.25	Ruta de solución en secuencia del modelo de tres ejes, resolviendo en paralelo bloques de dos grupos dentro de cada eje.	232
6.26	Ruta de solución en paralelo del modelo de tres ejes, resolviendo en paralelo grupos de dos componentes de cada eje.	233
6.27	Varias aproximaciones a la multiplicación de matrices por bloques.	235
6.28	Representación en el simulador del modelo MATMULT de multiplicación de matrices $AB = C$	235
6.29	Speed-up respecto a MKL secuencial variando el número de hilos asignados a MKL, para diversos tamaños de matrices.	237
6.30	Representación en el simulador del modelo MATMULT_COLS_50 de multiplicación de matrices $AB = C$, dividiendo por columnas la matriz B en dos bloques de igual tamaño.	238
6.31	Árbol de rutas de ordenación y agrupación de cálculos para resolver una multiplicación de matrices por bloques mediante el modelo de la figura 6.30.	238
6.32	Representación del modelo MATMULT_COLS_35 de multiplicación de matrices $AB = C$, dividiendo por columnas la matriz B en tres bloques.	239
6.33	Representación del modelo MATMULT_COLS_20 de multiplicación de matrices $AB = C$, dividiendo por columnas la matriz B en cinco bloques.	239
6.34	Selección de la ruta que aplica paralelismo de grupos para resolver una multiplicación de matrices por bloques mediante el modelo de la figura 6.30.	240

6.35	Selección de la ruta que aplica paralelismo de grupos para resolver una multiplicación de matrices por bloques mediante el modelo de la figura 6.32.	240
6.36	Selección de la ruta que aplica paralelismo de grupos para resolver una multiplicación de matrices por bloques mediante el modelo de la figura 6.33.	240
6.37	Modelos MATMULT (arriba) y MATMULT_COLS_50 (abajo) para la multiplicación de matrices $AB = C$ en el simulador. Una multiplicación de 1000×1000 se reemplaza por dos multiplicaciones 1000×500	241
6.38	Grafo de los cálculos en el algoritmo de Strassen.	247
6.39	Representación gráfica en el simulador del modelo STRASSEN de multiplicación de matrices sin recursión mediante el algoritmo de Strassen. Además del nombre de las rutinas, se muestran las funciones que las componen.	248
6.40	Árbol de rutas que representa las alternativas de ordenación y agrupación de cálculos para resolver una multiplicación de matrices mediante el algoritmo de Strassen como el representado en el modelo de la figura 6.39.	249
6.41	Representación gráfica en el simulador de la ruta de ejecución del modelo STRASSEN de multiplicación de matrices mediante el algoritmo de Strassen, aplicable a sistemas con CPU monocore, donde todos los grupos se ejecutan en secuencia.	250
6.42	Representación del modelo STRASSEN_R1 para la multiplicación de matrices mediante el algoritmo de Strassen con un nivel de recursión.	253
6.43	Representación gráfica en el simulador de la ruta de ejecución del modelo STRASSEN_R1 de multiplicación de matrices mediante el algoritmo de Strassen con un nivel de recursión, aplicable a sistemas monocore.	254
A.1	Vista esquemática del interfaz gráfico que ofrece PARCSIM-MB (Model Builder).	275

A.2	Barra de menú del interfaz gráfico de PARCSIM-MB. La imagen muestra los submenús desplegados para mostrar los enlaces a las funcionalidades del software.	276
A.3	Vista de mantenimiento de las variables de tamaño y las variables del modelo en el interfaz gráfico PARCSIM-MB.	281
A.4	Vista del mantenimiento de grupos que conforman un modelo en el interfaz gráfico PARCSIM-MB.	282
A.5	Vista del mantenimiento de escenarios incluido en el interfaz gráfico PARCSIM-MB.	284
A.6	Vista del mantenimiento de rutinas en el interfaz gráfico PARCSIM-MB.	285
A.7	Vista del mantenimiento de scripts en el interfaz gráfico PARCSIM-MB.	286
A.8	Vista del editor de las opciones de configuración del simulador. . .	288
A.9	Representación de la información matricial manejada por el simulador para la resolución del problema numérico de este tutorial. .	291
A.10	Algoritmo de resolución del problema numérico de ejemplo usado en este tutorial.	292
A.11	Barra de estado de la aplicación PARCSIM-MB cuando no hay ningún modelo activo. Solo es posible acceder a la configuración general del simulador.	293
A.12	Barra de menú de la aplicación PARCSIM-MB. La imagen muestra los submenús desplegados para mostrar los enlaces a las funcionalidades del software que se encuentran habilitadas cuando no hay ningún modelo activo.	294
A.13	Edición del nombre de un nuevo modelo durante su creación. . . .	294
A.14	Interfaz gráfico PARCSIM-MB: Información del número de elementos al iniciar la creación de un nuevo modelo.	295
A.15	Interfaz gráfico PARCSIM-MB: Selección de la vista para la creación de un nuevo grupo.	295
A.16	Introducción del nombre de un grupo durante su creación.	295
A.17	Interfaz gráfico que refleja el nuevo grupo Inicio.	296
A.18	Proceso de creación del grupo etiquetado como Grupo1.	297
A.19	Activación de la vista de gestión de rutinas.	298

A.20	Proceso de creación de la rutina ADD_SYS.	298
A.21	Proceso de asignación de la función MADADD a la rutina ADD_SYS.	299
A.22	Interfaz gráfico actualizado para mostrar la rutina ADD_SYS que incluye a las funciones MATADD y SOLVESYS.	299
A.23	Asignación de la rutina ADD_SYS al grupo Grupo1.	300
A.24	Interfaz gráfico donde se muestra un grupo con una rutina asignada, detalle de las funciones asociadas y el grafo que muestra la secuencia de cálculos.	301
A.25	Creación de la dependencia de Grupo1 respecto a Inicio.	302
A.26	Proceso de grabación de un modelo y mensaje informativo.	302
A.27	Mensaje de error obtenido al comprobar la consistencia de un modelo no completado.	303
A.28	Creación de grupos mediante el procedimiento de clonado.	304
A.29	Cambio del nombre y borrado de la rutina de un grupo.	305
A.30	Herramienta para añadir un hijo al grupo Grupo1.	305
A.31	Grafo que muestra un modelo aún incompleto del algoritmo mostrado en la figura A.10(b).	306
A.32	Grafo con todos los grupos que componen el modelo del ejemplo de la figura A.10(b), a falta de la creación de un grupo final.	307
A.33	Procedimiento para crear un grupo final.	307
A.34	Grafo completo con todos los grupos que componen el modelo del algoritmo de la figura A.10(b).	308
A.35	Grafo con todos los grupos y las rutinas que componen el modelo del ejemplo.	309
A.36	Error obtenido al comprobar los argumentos de las funciones que se ejecutan en los grupos.	309
A.37	Visor del grafo ampliado que permite visualizar los parámetros de las funciones.	310
A.38	Procedimiento para la creación de variables.	311
A.39	Proceso de creación de variable de tamaño.	314
A.40	Vista de las variables de tamaño y de modelo creadas.	314
A.41	Asistente para la asignación de variables de tamaño a variables de modelo.	315

A.42 Lista completa de las variables de modelo con la especificación de su tamaño.	316
A.43 Argumentos de las funciones que se ejecutan en un grupo y sus controles desplegados con las variables del modelo disponibles. . .	316
A.44 Proceso de asignación del primer argumento de la función MATADD (la variable A1) al Grupo1.	317
A.45 Grupo1 con todos sus parámetros y variables asignadas.	317
A.46 Mensaje de error por falta de parámetros en la rutina que ejecuta el Grupo2.	318
A.47 Modelo completo donde todas las funciones tienen asignados sus argumentos.	318
A.48 Rutas de resolución para el modelo de la figura A.10(b). Se ha resaltado la ruta 2, que incluye el cálculo simultáneo de los grupos Grupo3 y Grupo4.	319
A.49 Activación de la vista de gestión de escenarios.	321
A.50 Solicitud de un identificador para un nuevo escenario durante su fase de creación.	322
A.51 Vista de mantenimiento de un nuevo escenario. Se observan las dos variables de tamaño que definimos para especificar el tamaño de las variables del modelo.	322
A.52 Vista del grafo del modelo que muestra el escenario recién creado.	323
A.53 Edición de los valores de un escenario.	323
A.54 Escenario incorrecto debido a los tamaños de las matrices.	325
A.55 Mensaje de escenario correcto.	326
A.56 Activación de la vista de gestión de scripts.	326
A.57 Captura del nombre de un nuevo script durante su proceso de creación.	327
A.58 Editor preparado para introducir datos de un nuevo script.	327
A.59 Captura de rangos de valores e individuales de parámetros algorítmicos durante la edición de un script.	328
A.60 Proceso para la creación de constantes durante la edición de los scripts.	330

A.61 Validación de los valores de los scripts mediante fórmulas. El número de threads combinados a partir de los del primer nivel (valor de OMP1) y del segundo nivel (valor de OMP2) se limitan al valor 4 de la constante <code>maxthreads</code>	331
A.62 Activación de la vista de configuración del simulador.	331
A.63 Selección del directorio de ubicación del ejecutable del simulador.	332
A.64 Selección de los parámetros que determinan la ejecución del simulador.	333
A.65 Ventana de captura del conjunto de entrenamiento en el simulador (scripts y escenarios).	334
A.66 Proceso de captura de un nuevo escenario de entrenamiento.	335
A.67 Visualización del contenido de una matriz en el simulador.	336
A.68 Captura de la configuración y los parámetros algorítmicos aplicables al modo de simulación simple.	337
A.69 Captura de la configuración y las especificaciones del hardware necesarios para el modo de simulación autooptimizado.	338
A.70 Ventana de simulación.	339
A.71 Datos de configuración para una simulación en <code>Multiple mode</code>	340
A.72 Datos de escenario para una simulación en <code>Multiple mode</code>	341
A.73 Datos del script para una simulación en <code>Multiple mode</code>	342
A.74 Selección de una ruta para una simulación en <code>Multiple mode</code>	342
A.75 Resultados de la primera simulación en <code>Multiple mode</code>	343
A.76 Consulta de resultados.	344
A.77 Información obtenida en la consulta de resultados.	345
A.78 Uso de filtros en la consulta de resultados.	346
A.79 Consulta gráfica de los tiempos de ejecución en función de los parámetros algorítmicos. Se muestran datos obtenidos en la primera simulación en modo múltiple.	347
A.80 Configuración del simulador para el modo de entrenamiento.	348
A.81 Consulta de resultados de entrenamiento.	348
A.82 Configuración del simulador para la ejecución en <code>Multiple mode</code> con selección automática de rutas.	348

A.83 Resultados obtenidos en una simulación múltiple con selección automática de rutas. El software ha detectado tres posibles rutas en relación con los parámetros algorítmicos aplicados.	349
A.84 Acceso a la funcionalidad interactiva de cálculo de la ruta con menor tiempo de ejecución estimado en función de unas determinadas especificaciones de hardware y tamaños de problema.	350
A.85 Obtención de la mejor ruta mediante la funcionalidad interactiva incorporada en el software.	351
A.86 Obtención de las tres rutas teóricamente más eficientes. El cuadro de texto ofrecido en el visor de rutas muestra los tiempos obtenidos con cada ruta, ordenados de menor a mayor, y las diferencias entre ellas.	351
A.87 Rutina de ejemplo para el proceso de enlace de argumentos. . . .	352
A.88 Enlace de argumentos mediante el procedimiento de arrastrar desde el origen y soltar en el destino en el grafo de una rutina. . . .	353
A.89 Rutina de ejemplo que muestra dos enlaces entre parámetros. . .	353
A.90 Ejemplo de rutina anidada con enlace entre argumentos propios y heredados de la rutina anidada.	354
A.91 Selección de archivos durante la importación de modelos.	355
A.92 Grupos importados desde otro modelo e insertados en el modelo activo.	355
A.93 Modificación del modelo Tutorial que incluye un nuevo Grupo6 que resuelve una multiplicación de matrices.	356
A.94 Selección de un modelo para ser ejecutado en el Grupo6.	357
A.95 Interfaz que muestra el modelo MATMUL_COLS50_G1 para ser ejecutado en el Grupo6.	357
A.96 Selección de la rama del modelo MATMUL_COLS50_G1 que se va a ejecutar en el Grupo6.	358
A.97 Grafo del modelo Tutorial con el Grupo6 ejecutando una rama del modelo MATMUL_COLS50_G1.	358
A.98 Arbol de rutas del modelo Tutorial con el Grupo6 ejecutando el modelo MATMUL_COLS50_G1.	359

Índice de algoritmos

1	Rutina general de análisis cinemático computacional.	65
2	Cinemática computacional basada en ecuaciones de grupo.	69
3	Algoritmo para crear el árbol de rutas de resolución de un determinado sistema mediante un proceso iterativo en el que se van añadiendo nodos hijo en cada rama.	142
4	Función para descartar grupos que dependen en orden de ejecución de otros grupos que aún no han sido calculados.	143
5	Función que selecciona nuevos nodos hijo con combinaciones válidas de un determinado orden sobre el conjunto de <i>remainingGroups</i> durante el proceso de creación del árbol de rutas que resuelve un determinado problema.	144
6	Procedimiento para asignar un coste (tiempo de ejecución) a todos los nodos de un árbol de rutas previamente creado. Una vez calculado el coste, el sistema busca en el árbol otros nodos que tengan la misma carga computacional (es decir, la misma rutina y tamaño de datos) y les asigna el mismo coste.	151
7	Función para obtener el tiempo de ejecución estimado, <i>nodeCost</i> , necesario para calcular un determinado nodo, <i>node</i> , en un escenario <i>SCN</i> cuando el número de threads del primer y segundo nivel de paralelismo (<i>th.omp₁</i> , <i>th.omp₂</i>), librería (<i>l</i>) y cantidad de GPUs instaladas (<i>n_g</i>) son fijados por el usuario. La función devuelve también el número de GPUs con las que se obtiene el menor tiempo de ejecución.	152

8	Estimación del tiempo de ejecución de un nodo, <i>node</i> , cuando se aplican los parámetros algorítmicos <i>AP</i> en un escenario <i>SCN</i> . Si no hay información de entrenamiento ajustada al escenario <i>SCN</i> , se busca el escenario <i>SCN_nearest</i> que más se aproxime.	156
9	Proceso autooptimizado para obtener los mejores parámetros algorítmicos, <i>Best_AP</i> , y el tiempo de ejecución, <i>nodeCost</i> , de un nodo, <i>node</i> , y escenario <i>SCN</i> . <i>n_c</i> y <i>n_g</i> contienen el número de cores y de GPUs respectivamente. La función obtiene <i>Best_AP.n_c</i> , <i>Best_AP.n_g</i> y <i>Best_AP.library</i> para cada grupo de <i>node</i>	159
10	Obtención de los parámetros algorítmicos <i>AP</i> que consiguen el mejor tiempo de ejecución, <i>cost</i> , de una determinada rutina <i>R</i> en un escenario <i>SCN</i>	161

Índice de Listados

5.1	Extracto del fichero <code>functions.fun</code> que recoge la descripción de las funciones disponibles en el simulador. Se muestra la información relativa a la suma de matrices <code>MATADD</code>	112
5.2	Extracto del fichero <code>functions.fun</code> con información relativa a las librerías que implementan la resolución de sistemas de ecuaciones <code>SOLVESYS</code> en el simulador.	115
5.3	Extracto del fichero <code>routines.rou</code> que muestra la información relativa a la rutina <code>RADDMUL</code> creada por el usuario y compuesta por las funciones del simulador <code>MATADD</code> y <code>MATMUL</code> , suma y multiplicación de matrices respectivamente.	117
5.4	Extracto del fichero <code>routines.rou</code> que muestra la información relativa a la rutina <code>RSYSADDMUL</code> creada por el usuario y que está compuesta por la función de solución de un sistema de ecuaciones <code>SOLVESYS</code> , y de la rutina <code>RADDMUL</code> , compuesta a su vez de una suma, <code>MATADD</code> , y una multiplicación de matrices, <code>MATMUL</code>	120
5.5	Extracto del fichero <code>ModeloEjemplo.mdl</code> que describe el modelo de la figura 5.4. Se muestra información de los grupos <code>Start</code> y <code>Gr1</code> , y las constantes <code>ModelRows</code> y <code>ModelCols</code>	122
5.6	Extracto del fichero que describe uno de los escenarios que se emplearán para simular el modelo de la figura 5.4. Se muestra información de las dos variables que corresponden a las matrices con identificadores 1 y 2.	126
5.7	Sección del fichero <code>SCRIPT_ScriptDemo.scp</code> que describe uno de los scripts que servirá para generar los parámetros algorítmicos durante la simulación del modelo de la figura 5.4.	128

5.8	Sección del fichero <code>ModeloEjemplo.mdl</code> que describe el árbol de rutas de la figura 5.10. Se muestra información de la ruta identificada como 5. TRANS, ADD_SYS, LINSYS y ADD son ejemplos de rutinas creadas por el usuario y asignadas a los grupos correspondientes.	133
5.9	Contenido del fichero <code>treelimits.cfg</code> que contiene los valores límites de tiempo y tamaño para el proceso de creación de un árbol de rutas.	147
5.10	Contenido del fichero <code>config.cfg</code> que contiene la información de configuración del simulador.	165
6.1	Resultado del comando <code>magma_print_environment()</code> de la librería MAGMA para recopilar información del hardware.	243

Resumen

El modelado es la disciplina que permite analizar y simular el comportamiento de un determinado sistema mediante una representación numérica de sus propiedades. Entre sus áreas de aplicación se encuentran el estudio de sistemas naturales, climáticos, poblacionales o mecánicos. Las técnicas de modelado disponibles en la actualidad permiten abordar el estudio de sistemas cada vez más complejos que requieren del uso eficiente de sistemas computacionales para su resolución en unos tiempos aceptables dentro de los límites de las asignaciones de recursos informáticos. Por este motivo, los científicos plantean los modelos con un enfoque que permite su traducción a algoritmos susceptibles de ser ejecutados por un ordenador. Por ejemplo, en el campo de la ingeniería que estudia los sistemas multicuerpo, una formulación topológica facilita el modelado automático de dichos sistemas y permite una resolución computacional eficiente de su análisis cinemático. La información obtenida de este proceso se puede aplicar al diseño de nuevos mecanismos, y engloba aspectos tales como el análisis de la posición de los elementos que componen el sistema o el rango de desplazamiento de las piezas móviles.

Dado el interés por la implementación computacional de numerosos problemas de naturaleza científica, es posible acceder a paquetes de software elaborados por diversos grupos de investigación que resuelven determinados subproblemas y que se pueden reutilizar en la resolución de problemas de mayor complejidad. Un ejemplo de estos paquetes son las librerías de álgebra lineal, muy usadas habitualmente en este tipo de problemas científicos y que son objeto de estudio continuo para su adaptación a la incesante evolución del hardware.

Las plataformas de hardware actuales incorporan más de una unidad de procesamiento, bien integrando varios procesadores en sus CPUs, bien añadiendo otras unidades de arquitectura masivamente paralela, como las GPUs, para conformar nodos de computación híbridos. La existencia de este tipo de hardware paralelo motiva el interés en explotar la ejecución simultánea de cálculos, con la consiguiente reducción de los tiempos de resolución de modelos complejos. Ahora bien, una aplicación óptima de técnicas paralelas requiere de un conocimiento profundo del hardware y de las librerías de cómputo disponibles, muchas de las cuales requieren un ajuste mediante parámetros para aprovechar todo su potencial. Por este motivo, no es frecuente que investigadores en áreas científicas concretas sean a la vez expertos conocedores de los diversos paradigmas de programación paralela existentes.

Esta tesis investiga un enlace entre la disciplina de la ingeniería mecánica y la computación, ofreciendo a usuarios no expertos en paralelismo un software que incorpora el estudio y optimización de algoritmos mediante una adecuada selección de librerías y su configuración. Partiendo del modelo de un sistema multicuerpo expresado en forma de grafo acíclico de los cálculos (básicamente operaciones de álgebra matricial) y sus dependencias, el software permite a un usuario ajustar los parámetros de paralelismo y librerías, y realizar simulaciones del modelo para analizar la influencia que cada configuración tiene en los tiempos de resolución. Además, una ejecución autooptimizada recomienda al usuario la manera más eficiente de paralelizar los cálculos, la librería a usar y los ajustes teóricos óptimos que se recomiendan para los parámetros algorítmicos en cada etapa de resolución del algoritmo.

Por último, esta metodología se puede aplicar a otras disciplinas ajenas a la ingeniería mecánica, en concreto a aquellas donde los problemas se pueden plantear con un enfoque similar, es decir, como agrupaciones de cálculos que incluyan operaciones matriciales realizadas en una determinada secuencia. Un ejemplo de aplicación en este sentido lo encontramos en algunas rutinas básicas de álgebra lineal, como la multiplicación de matrices.

Abstract

Modeling is the discipline that allows the analysis and simulation of the behavior of a certain system through the numerical representation of its properties. Applications include among others the study of natural, climatic, population or mechanical systems. The modeling techniques available nowadays make it possible to afford the study of increasingly complex systems, usually requiring the efficient use of computer systems to obtain its numerical resolution in times within the limits fixed by the computer resources. For this reason, scientists develop these models focusing on its subsequent translation into algorithms that can be executed in a computer. For example, in the field of engineering that studies multibody systems, a topological formulation facilitates automatic modeling and allows the efficient computational resolution of the system kinematics. The information obtained from this process can be applied afterwards to the design of new mechanisms, and covers aspects such as the analysis of the position of the elements that make up the system and the range of movement of the moving parts.

Given the growing interest in the computational implementation of many scientific problems, it is possible to access software packages already developed by several research groups aiming to solve certain sub-problems in a particular area. Those can be reused afterwards for solving more complex problems. An example of those packages are the linear algebra libraries, widely used in these types of scientific problems, which have been subject to continuous study for their adaptation to the constant evolution of modern hardware configurations.

The new hardware platforms incorporate more than one processing unit, either by integrating several processors in their CPUs, or by adding other units of massively parallel architecture, such as GPUs, or a combination of them, to make up hybrid computing nodes. The availability of this type of parallel hardware leads to an interest in exploiting the execution of simultaneous calculations, getting the benefit in the reduction of execution times. However, the optimal fit of parallel techniques requires a wide knowledge of the hardware and the available software libraries, many of which require the adjustment of a set of parameters for high performance. For this reason, it is not common for expert researchers in a specific scientific area to also be knowledgeable of the various existing parallel programming paradigms.

This thesis focuses on covering the gap between the discipline of mechanical engineering and computing, providing to non expert users in parallelism a simulator that includes the analysis and optimization of algorithms through the appropriate selection and configuration of libraries. Based on the model of a multi-body system in the form of a graph that collects the calculations (basically matrix algebra operations) and their dependencies, this software lets the user to adjust the parallelism and libraries' parameters and then to carry out simulations to analyze the influence of this adjustment on the execution times. A self-optimized execution will also find the most efficient way to group the calculations, as well as the appropriate library to be used and the optimal settings that should be applied in each resolution stage of the algorithm.

Additionally, this thesis shows how this methodology can be applied to other disciplines outside the mechanical engineering field, specifically to those where problems can be posed with a similar approach, that is, as groupings of calculations consisting on matrix operations performed in a certain sequence. An example is the application to some basic linear algebra routines, such as the matrix multiplication.

Capítulo 1

Introducción

En este capítulo inicial de la tesis se ofrece una visión general del contenido y estructura del presente documento. Se describe el contexto en el que se enmarca la investigación realizada, introduciendo para ello los conceptos teóricos necesarios, y llevando a cabo un repaso general del estado del arte en las áreas objeto de estudio. A continuación se presenta el objetivo principal de la tesis y su desglose en objetivos específicos, así como la metodología empleada en la consecución de los mismos. Por último se enumeran las contribuciones realizadas a raíz de este trabajo, y se explica la estructura de la tesis con un breve extracto del contenido de cada capítulo.

1.1 Contexto

En puertas de lo que se considera ya como una cuarta revolución industrial, o Industria 4.0, la simulación es una técnica que permite a las empresas ensayar el funcionamiento de determinados sistemas y anticiparse a problemas derivados de un diseño ineficiente de sus instalaciones o equipos. Si a esto le sumamos los avances en el desarrollo de sistemas computacionales de alto rendimiento, obtenemos como resultado la posibilidad de realizar simulaciones del funcionamiento de sistemas y mecanismos cada vez más complejos, y en un tiempo menor.

En el caso particular del diseño de máquinas, la simulación computacional de su cinemática permite predecir el comportamiento de todos sus elementos móviles, así como la interacción de los mismos con su entorno, permitiendo identificar y corregir fallos de funcionamiento antes de su comercialización, con el consiguiente ahorro de costes que puede suponer para la empresa que las produce.

En la actualidad, las plataformas computacionales han evolucionado hacia arquitecturas heterogéneas, compuestas por múltiples unidades de procesamiento paralelo que pueden integrar a su vez diferentes tipos de procesadores, como por ejemplo CPU (procesadores de propósito general con unos pocos cores) y GPU (coprocesadores gráficos con gran potencia de cálculo paralelo). Este nivel de heterogeneidad en el hardware conlleva la no uniformidad en los sistemas de desarrollo, haciendo que su programación pueda combinar más de un paradigma, invitando a seguir un modelo de programación híbrida capaz de explotar todos los recursos que ofrezca la plataforma hardware empleada.

Sin embargo, a pesar del avance en las prestaciones y en los recursos de las plataformas computacionales, incluidas las de coste medio-bajo utilizadas normalmente por los investigadores y diseñadores en general, las herramientas que se utilizan para la simulación y diseño de máquinas apenas explotan un porcentaje muy reducido de su potencial de cómputo; en la mayoría de los casos, los desarrolladores de software propio centran sus esfuerzos en mejorar las técnicas de modelado y las formulaciones cinemáticas y dinámicas en las que se basa la simulación, mientras que los diseñadores que utilizan software comercial, simplemente delegan en él su capacidad de explotar los recursos computacionales de forma eficiente.

Esta tesis doctoral profundiza en el estudio de técnicas y algoritmos de computación paralela y autooptimización con la intención de ofrecer a desarrolladores de software propio y comercial una herramienta capaz de explotar al máximo las prestaciones de la plataforma computacional utilizada para el análisis cinemático computacional de sistemas multicuerpo orientado al diseño eficiente de sistemas mecánicos. Además, la aplicación de estas técnicas y algoritmos se puede extender a la resolución de problemas de álgebra lineal en general, mejorando la eficiencia de las librerías de cálculo disponibles actualmente.

Las principales motivaciones que nos han llevado a emprender este estudio son las siguientes:

- En primer lugar, identificar la librería de cálculo adaptada a la naturaleza de un problema a resolver.

Los algoritmos de simulación de mecanismos se componen de rutinas que incluyen operaciones de álgebra lineal sobre matrices. Dichas matrices presentan diferentes tamaños y factores de dispersión en función de la topología del sistema mecánico en estudio. En esta rama de la ingeniería los cálculos con mayor demanda computacional son las resoluciones de los sistemas de ecuaciones que describen la morfología de los sólidos en que se dividen los mecanismos, así como las interacciones y dependencias entre ellos.

En general, el uso de métodos de álgebra lineal es habitual en la resolución de muchos problemas científicos y de ingeniería. Por este motivo es posible encontrar librerías de cómputo desarrolladas y optimizadas para el tipo de matrices predominante en un dominio científico concreto. Algunas de dichas librerías incorporan rutinas capaces de aprovechar los recursos ofrecidos por los sistemas computacionales paralelos, mediante la adecuada selección de unos parámetros que el usuario debe conocer.

Por todo ello es crucial, en términos de rendimiento, la selección de la librería que mejor se adapta al tipo de matrices a manejar y el ajuste óptimo de los parámetros que permita asegurar un uso eficiente de los recursos.

- En segundo lugar, explotar al máximo la topología del mecanismo.

El modelo matemático que se elabore a partir de un sistema mecánico determinará el algoritmo necesario para su implementación computacional y, por tanto, su rendimiento. En un software comercial de simulación de mecanismos es habitual abordar los sistemas de manera global, lo que implica que las matrices crecerán en tamaño hasta poder incluir las coordenadas de todos los cuerpos que componen el sistema y sus restricciones cinemáticas. Este enfoque global presenta el inconveniente de manejar matrices de gran tamaño. Por el contrario, una formulación topológica basada en la fragmentación de los mecanismos en grupos cinemáticos permitirá obtener la solución del sistema completo mediante resoluciones independientes de cada

grupo por separado y en un orden determinado, ofreciendo esta sistemática una doble ventaja: en primer lugar el uso de matrices de menores dimensiones (al manejar subsistemas de tamaño menor) y, en segundo lugar, la aplicación de paralelismo en la resolución de los grupos. En este sentido, serán necesarias estrategias de identificación de los componentes susceptibles de ser calculados de manera simultánea.

- En tercer lugar, maximizar el aprovechamiento de la capacidad de cómputo disponible.

La simulación de mecanismos mediante métodos numéricos requiere gran cantidad de recursos computacionales, por lo que el uso eficiente de los mismos garantiza una reducción en los tiempos de ejecución. En el ámbito científico es habitual encontrar plataformas de hardware heterogéneas que combinan en un único nodo de computación varias unidades multinúcleo con varios coprocesadores, pudiendo ser estos últimos de distinta naturaleza. Por tanto, tendríamos nodos híbridos compuestos de una CPU (n cores) + m coprocesadores.

Cada arquitectura ofrece un tipo de memoria subyacente, que determina el modo en que ésta puede ser utilizada por los nodos que constituyen la plataforma de cómputo y que, por tanto, es muy relevante a la hora de seleccionar el paradigma de programación a seguir, que puede no ser único debido a la naturaleza heterogénea de los componentes de los sistemas computacionales paralelos. Por ejemplo, la programación de un coprocesador de tipo GPU con CUDA [87] seguiría el paradigma SIMD (*Simple Instruction Multiple Data*), pero un coprocesador de tipo MIC (*Many Integrated Core Architecture*) podría programarse siguiendo el estándar OpenMP [53, 233, 234] de memoria compartida, al igual que se haría con una CPU multinúcleo. Sin embargo, un esquema de memoria distribuida demanda el uso de un estándar de paso de mensajes, como MPI [213], si queremos maximizar su rendimiento.

Por tanto, la programación de un software optimizado para un determinado sistema paralelo, especialmente si es híbrido, requiere de unos conocimientos muy específicos de la arquitectura del hardware y de las técnicas de programación paralela aplicables. En el caso particular de un software de simulación de mecanismos es imprescindible una formación en formulaciones cinemáticas y álgebra lineal, pero también un conocimiento de las librerías computacionales existentes. Debido a esta complejidad, será difícil encontrar la mejor configuración sin la ayuda de un proceso de autooptimización que determine los parámetros de paralelismo y el tipo de librería adecuados para una determinada configuración del hardware disponible.

- Por último, generalizar la investigación a otros ámbitos de naturaleza científica.

Las técnicas desarrolladas para la optimización de un simulador de sistemas multicuerpo se pueden aplicar a otros problemas computacionales con estructura similar, en concreto a aquellos consistentes en la resolución simultánea de problemas de álgebra lineal.

De manera similar al proceso que se sigue durante la formulación topológica de los sistemas multicuerpo, se buscará una descomposición del problema a resolver en un conjunto de unidades independientes cuyas instrucciones puedan ejecutarse de manera simultánea, probablemente mediante asignaciones a varios elementos de proceso.

1.2 Estado del arte

En esta sección se ofrece una recopilación de los últimos avances en las áreas de conocimiento relacionadas con la elaboración de esta tesis, en concreto el modelado y simulación de sistemas mecánicos, las arquitecturas de hardware paralelas y su programación, las librerías de álgebra lineal y, por último, las estrategias de autooptimización de software.

1.2.1 Modelado, análisis y simulación de sistemas mecánicos

Tradicionalmente, el modelado de sistemas mecánicos ha consistido en la recreación de los mismos mediante la construcción de modelos físicos, bien a escala o usando prototipos. Dependiendo del tipo y complejidad del sistema que se quiere modelar, las recreaciones físicas pueden ser costosas en tiempo y en dinero o, simplemente, imposibles de realizar (por ejemplo, el efecto sobre la estructura ósea del cráneo que produce un cambio en el material de un casco para motocicleta ante un impacto lateral). La alternativa a este enfoque tradicional es el análisis de modelos, en este caso matemáticos, que tienden a reproducir una situación real. Antes de la llegada de las computadoras, estos modelos matemáticos se resolvían mediante métodos grafo-analíticos que todavía se enseñan en las universidades por su sencillez y su didáctica. Estos métodos están muy limitados, ya que solo permiten resolver mecanismos sencillos, con pocos eslabones y uniones básicas entre elementos, y en un instante o posición concreta. Además, para analizar el mecanismo en otra posición hay que volver a desarrollar todos los cálculos de nuevo partiendo de cero. Pero en la actualidad, gracias al uso de las computadoras, estos modelos se resuelven mediante métodos numéricos avanzados que permiten su análisis en muchas situaciones (simulaciones) realmente complejas y con escasa intervención del analista. De la teoría e implementación práctica de estas simulaciones avanzadas se encarga la dinámica de sistemas multicuerpo.

La dinámica de sistemas multicuerpo se puede definir como un área de la mecánica computacional que reúne varias disciplinas, tales como la dinámica estructural, mecánica multifísica, matemática computacional, teoría del control y ciencia de la computación, con la finalidad de proporcionar métodos y herramientas para el prototipado virtual de sistemas mecánicos complejos [104]. La dinámica de sistemas multicuerpo juega hoy en día un papel fundamental en el modelado, análisis, simulación y optimización de sistemas mecánicos en una amplia variedad de campos y para un elevado rango de aplicaciones industriales: máquinas, mecanismos, robótica, vehículos, estructuras espaciales o biomecánica, por citar algunos.

El área de la dinámica estructural se centra en el estudio de los métodos para el análisis y la síntesis de los sistemas multicuerpo. Más concretamente, el campo del análisis, en el que se centra este trabajo, pretende ofrecer soluciones a los problemas de índole cinemática y dinámica.

En los problemas de análisis cinemático se estudia la posición y el movimiento del sistema multicuerpo. Se trata de problemas puramente geométricos, independientes de las fuerzas causantes del movimiento y de las características inerciales de los cuerpos que componen el sistema. Los problemas más comunes son:

- Posición inicial. Se trata de determinar la posición de todos los elementos de un sistema multicuerpo, a partir de la posición de los elementos de entrada. Es un problema de difícil solución que conlleva la resolución de un sistema de ecuaciones algebraicas no lineales con varias soluciones.
- Desplazamiento finito. Se trata de encontrar la posición final de los elementos de un sistema que evoluciona, desde una posición inicial, mediante un desplazamiento finito y no diferencial de los elementos de entrada. Pese a que es una variación del problema de posición inicial, tanto conceptualmente como desde el punto de vista de la herramienta matemática utilizada para resolverlo, su resolución es más sencilla dado que la posición inicial del sistema, conocida, puede ser utilizada como punto de partida para el proceso iterativo de resolución del sistema de ecuaciones algebraicas no lineales. La multiplicidad de soluciones no supone un problema en este caso, puesto que solo resulta de interés la solución más cercana a la inicial del sistema.
- Análisis cinemático directo. Consiste en relacionar el movimiento (posición, velocidad y aceleración) de los sólidos que componen el sistema multicuerpo con el movimiento del conjunto de actuadores encargado de controlarlo [120]. En el problema directo se define el movimiento de los actuadores y se debe determinar el movimiento de todos los sólidos del sistema.
- Análisis cinemático inverso. Similar al anterior, pero en el que se especifica la posición de alguno de los sólidos, normalmente un terminal encargado de ejecutar una trayectoria concreta, y se deben determinar los movimientos de los actuadores necesarios para conseguir que el terminal describa esa trayectoria.

- Simulación cinemática. Se trata de predecir el comportamiento cinemático del sistema, y engloba todos los problemas definidos anteriormente. Se utiliza para detectar posibles colisiones entre elementos, estudiar trayectorias de puntos y determinar las secuencias de las posiciones de los elementos.

Los problemas de análisis dinámico son mucho más complejos de resolver que los cinemáticos, que deben resolverse previamente (o simultáneamente) [189, 268]. Los problemas dinámicos involucran a las fuerzas que actúan sobre el sistema y, además, la masa, el tensor de inercia y la posición del centro de masas de todos los eslabones que lo forman. Los problemas más relevantes de índole dinámica que aborda esta disciplina son los siguientes:

- Posición de equilibrio estático. Se trata de determinar la posición en que se equilibran las fuerzas exteriores, las fuerzas de enlace y las reacciones que actúan sobre el sistema. Se engloba aquí, a pesar de no tratarse de un problema dinámico, ya que al menos depende de la masa y la posición del centro de masas de los eslabones. La solución al problema exige la resolución de un sistema de ecuaciones mediante procedimientos iterativos.
- Dinámica linealizada. Se linealizan las ecuaciones del movimiento en una posición particular del sistema para, mediante un estudio paso a paso de la historia del sistema o un análisis de valores propios, determinar los modos y las frecuencias de vibración natural que tienen lugar a partir de una posición de equilibrio estático o dinámico del sistema. De esta forma se conoce la rigidez dinámica del sistema.
- Dinámica inversa. Se trata de determinar la forma en que deben evolucionar las fuerzas de entrada para que el movimiento del sistema tenga unas características determinadas. Conocido el movimiento, pueden determinarse las aceleraciones de los centros de masas, por lo que las posiciones y sus primera y segunda derivadas son conocidas. El sistema que se debe resolver es, por tanto, lineal.
- Dinámica directa. Es el problema más complejo dentro de la dinámica. En este caso son conocidas las fuerzas externas que actúan sobre los eslabones y debe determinarse el movimiento que producen sobre ellos a partir de unas

condiciones iniciales de contorno conocidas. Exige formular las ecuaciones diferenciales (ordinarias o algebraicas) del movimiento, como en el caso anterior, pero ahora lo que se puede evaluar son las aceleraciones de los centros de masas. Éstas deberán integrarse dos veces para determinar las velocidades y las posiciones de los eslabones. Supone, por tanto, la utilización de métodos numéricos para integrar sistemas de ecuaciones diferenciales cuya complejidad dependerá del tipo de coordenadas utilizado para modelar el problema.

- Simulación dinámica. Se refiere, en la práctica, a la resolución del problema dinámico directo y engloba la simulación cinemática.
- Percusiones e impactos. Se trata de determinar el efecto que las percusiones e impactos que actúan sobre el sistema tienen sobre la distribución de velocidades del mismo. Las percusiones, de limitada importancia práctica, producen discontinuidades en la distribución de velocidades que pueden ser calculadas introduciendo una ecuación de naturaleza experimental que incorpore el tipo de impacto y la naturaleza de las superficies en contacto durante el mismo.

Centrándonos en la simulación de sistemas multicuerpo, se exponen a continuación las fases implicadas en la modelización y simulación de un sistema mecánico cualquiera (figura 1.1):

- ① El mecanismo real se representa mediante un gráfico en el que se incluyen todos los cuerpos y las uniones que lo componen.
- ② Se añaden al gráfico suficientes entidades, puntos y vectores, necesarios para identificar las posiciones y orientaciones de todos los sólidos en cualquier instante. De esta forma, las coordenadas cartesianas y, en su caso, angulares de esas entidades forman el conjunto de variables que define al modelo.
- ③ Las variables se relacionan entre sí mediante un conjunto de ecuaciones de restricción que obligan a que el sistema mecánico se encuentre siempre ensamblado en posiciones compatibles con las ligaduras entre los sólidos. La forma que adoptan estas ecuaciones de restricción depende del tipo de

coordenadas seleccionado. Se definen propiedades físicas de los componentes (longitudes, masas, tensores de inercia, etc.) y modelos matemáticos de fuerzas que producen movimiento (actuadores) y de fuerzas que lo detienen (resistencias) según el tipo de problema a resolver. En este punto termina la etapa de modelado.

- ④ Previo a la simulación por computador, y dependiendo del software o el lenguaje de programación elegido, será necesario reescribir el modelo (las ecuaciones) en un formato determinado.
- ⑤ Se procede a la simulación, ejecutando los algoritmos implementados para la solución computacional del modelo matemático elaborado. Como resultado se obtienen datos sobre el comportamiento teórico del mecanismo.
- ⑥ En el caso de que los resultados no sean los esperados, se introducirían modificaciones en el modelo formal para mejorar su validez. Esto puede ocurrir, bien porque no se cumplen criterios de diseño, bien porque se encuentran desviaciones sobre el comportamiento real del mecanismo mediante observación de un prototipo físico, si se dispone de él.

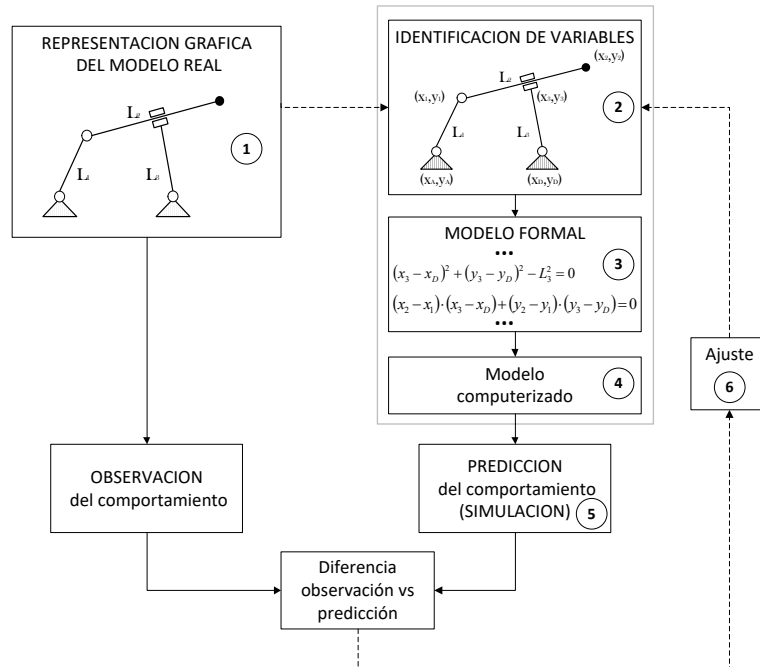


Figura 1.1: Fases requeridas durante el proceso de modelado y simulación de un sistema multicuerpo.

En esta tesis se pretende optimizar los algoritmos necesarios para la simulación cinemática de sistemas multicuerpo (fase 5 de la figura 1.1), de acuerdo con las motivaciones expuestas en el apartado 1.1. Para una mejor respuesta tanto a estas motivaciones como a los objetivos de la presente tesis, se profundizará en el capítulo 2 sobre el modelado (fases 1, 2 y 3) y la simulación (fases 4 y 5) de sistemas mecánicos, haciendo especial énfasis en el modelado basado en ecuaciones de grupo, el cual consiste en una solución modular que pretende explotar al máximo la topología del sistema mecánico a simular.

Cabe destacar que, aunque la optimización de algoritmos paralelos se ha aplicado en esta tesis a la simulación cinemática de sistemas multicuerpo, la verdadera potencialidad de la metodología que se presenta reside en que se puede aplicar, igualmente, a la optimización de las rutinas de análisis de cualquier otro método computacional, como puede ser la síntesis óptima de mecanismos, o la integración de las ecuaciones del movimiento, la detección de colisiones o la representación gráfica 3D en dinámica computacional, solo por mencionar algunos campos de interés para la comunidad multibody.

Debido a la necesidad de acudir a la simulación dinámica para el diseño de sistemas mecánicos, han proliferado el desarrollo y la comercialización de simuladores dinámicos con diferentes ámbitos de aplicación y prestaciones. En el siguiente apartado se recogen las características de algunos de los simuladores más reconocidos en la literatura.

1.2.2 Simuladores para la dinámica de sistemas mecánicos

Existen en el mercado herramientas computacionales, tanto comerciales como gratuitas, para la simulación de sistemas mecánicos. Algunas de ellas emulan el comportamiento de equipos industriales, principalmente robots disponibles en el mercado. Este tipo de simuladores se suelen emplear en tareas de formación del personal que va a manejar o instalar los equipos reales.

Por otro lado, existen herramientas más generales que permiten modelar cualquier tipo de mecanismo, añadiendo sólidos al sistema y especificando las juntas que representan las uniones de dichos sólidos con el resto de componentes. La

tipología de estas juntas o uniones restringe los movimientos relativos posibles de cada componente y del mecanismo en su conjunto. Con toda esta información, estos simuladores realizan el análisis cinemático calculando los movimientos posibles que puede adoptar el sistema bajo estudio. Si, además, se permite especificar el tipo de material, masa y rigidez de los elementos que componen el sistema, las simulaciones pueden incluir análisis de comportamiento dinámico y de resistencia mecánica.

Estos simuladores normalmente incluyen un conjunto de utilidades que facilitan su uso y permitan un ajuste del software a las necesidades de los usuarios. Entre ellas se encuentran:

- Entornos gráficos para el diseño de los mecanismos y la visualización de los resultados de las simulaciones.
- Librerías de objetos y piezas que se pueden incorporar al modelo que se está editando. Cada objeto está definido por su geometría y su comportamiento.
- Utilidades para la creación de módulos que implementen situaciones o comportamientos no contemplados en el propio simulador. En este caso el sistema emplea algún tipo de API (*Application Programming Interface*) para permitir la programación de estos añadidos en algunos de los lenguajes más conocidos, como C++ [279], Python [243], MATLAB [196], Java [235], etc.

Entre las herramientas de simulación más conocidas podemos encontrar las siguientes:

- Gazebo [122]: Cuenta con un completo entorno visual de diseño y simulación en 3D basado en el sistema de renderizado gráfico OGRE [229]. Incluye por defecto algunas figuras básicas, como cilindros, esferas y cubos, y también permite importar modelos creados en diversos formatos gráficos, como el estándar SVG [301] (*Scalable Vector Graphics*), o mallas 3D creadas en aplicaciones como Blender [41]. Gazebo permite especificar las limitaciones de movimiento de los componentes del sistema modelado, los límites de fuerza y velocidad soportadas y datos relativos a los materiales, como la viscosidad y la fricción.

Permite también el estudio de la dinámica de los mecanismos, siendo compatible con varios de los motores físicos más conocidos, como BULLET [46], DART [124], ODE [256] y SIMBODY [273]. Se trata de librerías que ofrecen estructuras de datos y algoritmos para la cinemática y dinámica en robótica.

Gazebo dispone de una API de programación en lenguaje C++ que permite incorporar funciones complementarias para modificar el comportamiento del robot. Se ofrece con licencia Apache 2 [19] y, por tanto, libre.

- MORSE [212]: Es un simulador académico orientado a la simulación 3D que se puede controlar por completo desde la línea de comandos. Las escenas de simulación y el comportamiento de cada elemento se generan a partir de simples conjuntos de instrucciones de Python.

MORSE incorpora un paquete de sensores estándar (cámaras, escáneres láser, GPS) y actuadores (controladores de velocidad, controladores de puntos de referencia, articulaciones y uniones genéricas).

La representación gráfica que ofrece MORSE se basa en Blender, lo que permite utilizar fácilmente la librería BULLET para la simulación de la física del movimiento del mecanismo.

- V-REP [66]: Dispone de un simulador 3D propio, tanto para editar los modelos que incorpora como para importar y exportar modelos grabados en un amplio abanico de formatos. Además, el sistema dispone de un conjunto de modelos básicos que permiten al usuario realizar un gran número de simulaciones.

Para llevar a cabo el estudio de la dinámica, dispone de integración con las ya mencionadas librerías BULLET y ODE, y también con NEWTON [164] y VORTEX [59]. Para la cinemática dispone de sistemas de cálculo que contemplan tanto la resolución inversa (IK) como la directa (FK).

V-REP puede trabajar con una API estándar que ofrece más de 500 funciones programadas en C++ y LUA [187] desde las que se pueden controlar la escena y los modelos. Además, incorpora APIs externas con cerca de 100 funciones que permiten enlazar la simulación con otras librerías. Admite programación en C/C++, Python, Java, MATLAB, Octave [163] y LUA.

Este software se ofrece con licencia comercial, con una modalidad gratuita con fines educativos. En el año 2019, V-REP ha sido mejorado y renombrado como CoppeliaSim.

- RobotStudio [1]: Es un simulador desarrollado por la empresa ABB. Aunque es un software que requiere licencia, existe una versión de evaluación, funcional durante 30 días.

Dispone de un gran conjunto de brazos robóticos propios y de un sistema de renderización muy conseguido. Al utilizar sus propios robots, el comportamiento de los mismos dentro del entorno simulado es casi idéntico al entorno real, lo que dota de efectividad a la simulación. Por el contrario, no se dispone de una opción para realizar pruebas con elementos ajenos a los ofrecidos por el fabricante y que no estén previamente cargados en el sistema.

Incluye la posibilidad de escribir programas en el lenguaje RAPID [2], creado por ABB. Estos programas pueden ser ejecutados posteriormente en entornos de producción mediante la funcionalidad de comunicación de robots simulados con robots reales.

- ROBODK [255]: Es un software industrial con el que se pueden simular alrededor de 500 robots industriales y herramientas de unos 40 fabricantes, como ABB, Fanuc, KUKA, Yaskawa/Motoman, Universal Robots, etc.

Este simulador resuelve la cinemática inversa, permitiendo manejar cualquier eslabón o articulación, y propagando el movimiento a sus uniones de forma correcta y respetando todas las restricciones.

En cuanto a la programación, como punto fuerte ofrece la creación de instrucciones en Python para modificar el comportamiento del brazo robótico. Dispone de la posibilidad de generar un código en lenguaje RAPID para ser cargado en los robots físicos que realizarán el trabajo real.

Aunque no dispone de un sistema de física o dinámica parametrizable, la posibilidad de programar el entorno en Python permite superar esta limitación, ofreciendo la alternativa de procesar los cálculos en herramientas externas como MATLAB.

- Actin Simulation [100]: Permite la creación de modelos en formato CAD (*Computer Aided Design*), y es capaz de realizar simulaciones que calculan la cinemática directa e inversa de sistemas robóticos de alto grado de libertad en tiempo real. Actin puede simular la coordinación entre múltiples robots de cualquier tipo o fabricante, así como accesorios para otros robots, herramientas y objetos simulados, ideal para simular sistemas de manipulación de objetos. También incluye sensores y cámaras que permiten dotar a la simulación de comportamientos similares al mundo real.

Los componentes de las simulaciones de Actin son configurables usando el estandar XML (*Extensible Markup Language*), y puede conectar fácilmente su código con componentes del kit de herramientas Actin para construir objetos C++. Dicho kit incluye funciones para cálculo geométrico (como por ejemplo matemática vectorial tridimensional) y rutinas matriciales (como las transformaciones).

- Webots [83]: Es un entorno de simulación de robots creado por Cyberbotics. Permite usar un conjunto de robots prediseñados y también crear nuevos modelos mediante un editor propio o con herramientas de diseño externas, como Blender.

Para el cálculo y parametrización de la dinámica de los sistemas se utiliza ODE, incluyendo en su versión “Pro” la posibilidad de definir comportamientos avanzados, como dinámicas de fluidos, sistemas sin fricción, etc.

En cuanto a las posibilidades de programación, incorpora una API con cerca de 200 funciones disponibles en varios lenguajes de programación que dan acceso a otros sistemas como OpenCV [232] para el procesamiento de imágenes o MATLAB para cálculos complejos. A la hora de realizar la simulación es necesario especificar un controlador para cada uno de los elementos que se desea manejar, el cual puede estar escrito en C/C++, Java, Python o MATLAB.

- EASY-ROB [82]: Dispone de una librería estándar con robots de varios fabricantes y ofrece la posibilidad de adquirir librerías para robots de fabricantes no incluidos en la versión estándar. Incorpora un sistema de modelado 3D para primitivas (cilindros, conos, esferas y cubos) y es capaz de importar y exportar modelos complejos en varios formatos.

Dispone de un lenguaje de programación propio para los robots, ERPL (*Easy-Rob Program Language*), y también incluye una API para integrar nuestros propios algoritmos en C para cinemática inversa, planificación e interpolación de movimientos, control dinámico y desarrollo de diálogos de usuario e interfaz de sensores.

- Marilou Robotics Studio [18]: Este entorno de simulación de la empresa anyKode está diseñado tanto para robots humanoides como para brazos articulados o sistemas de robots paralelos.

Dispone de una interfaz donde es sencilla la inclusión de robots y de otros elementos participantes en la simulación, incluyendo el ensamblado de cuerpos rígidos que disponen de masa e inercia. Mediante la unión de los cuerpos rígidos se pueden definir el número de grados de libertad del sistema y calcular la cinemática inversa.

La programación del robot se realiza mediante una SDK propia, la cual dispone de librerías válidas para varios lenguajes, incluido C++.

- MATLAB® [197]: Es una conocida herramienta de software matemático que ofrece un entorno de desarrollo integrado con su propio lenguaje de programación, especialmente orientado a problemas de cálculo matricial. Es posible usarlo para la simulación numérica de sistemas mecánicos como el que nos ocupa, ya que incluye herramientas específicas para diseñar aplicaciones de robótica a través de su librería *Robotics System Toolbox* [200].

Incorpora un ambiente de modelado denominado SimMechanics [201], orientado a trabajar con diagramas de bloques para la simulación de mecanismos de cuerpos rígidos. Además dispone de un entorno de simulación Simulink® [202] con las herramientas necesarias para estudiar el comportamiento dinámico de los modelos.

MATLAB es software propietario de MathWorks®.

- MapleSim™ [193]: Es una herramienta de modelado y simulación creada por MapleSoft®, que surge como una evolución del sistema de computación simbólica Maple™ [192] para ofrecer funcionalidades similares a las herramientas anteriores, incluyendo un entorno gráfico donde se puede observar el movimiento de los sistemas multicuerpo.

Se puede crear un robot a partir de la librería de componentes estándar. El sistema es capaz entonces de generar automáticamente las ecuaciones que describen el sistema completo y realizar su visualización. Para añadir nuevos elementos basta con crear nuevas ecuaciones que definan su comportamiento.

- **SOLIDWORKS** [85]: Es un software de diseño 3D creado en 1995 por la empresa SOLIDWORKS Corp. que permite modelar piezas y ensamblajes en 3D y planos en 2D. Ofrece un abanico de soluciones para cubrir los aspectos implicados en el proceso de desarrollo del producto: crear, diseñar, simular, fabricar, publicar y gestionar los datos del proceso de diseño.

SOLIDWORKS Simulation Professional [86] permite optimizar el diseño, determinar la resistencia mecánica del producto, la durabilidad del producto, la topología y realizar simulaciones secuenciales de física múltiple.

- **Abaqus** [84]: Es un conjunto de herramientas desarrollado por la compañía Abaqus Inc. que emplea el método de elementos finitos para realizar cálculos aplicables al ámbito de la ingeniería, como análisis estructurales, tanto estáticos como dinámicos, problemas de contacto entre sólidos, térmicos, mecánica de fluidos y otros.

Consta de varios módulos, siendo Abaqus/CAE (*Complete Abaqus Environment*) el utilizado tanto para el modelado y análisis de componentes mecánicos y ensamblajes (preprocesamiento) como para visualizar el resultado del análisis de elementos finitos llevados a cabo por los componentes Abaqus/Standar y Abaqus/Explicit. El módulo CFD proporciona capacidades avanzadas de dinámica de fluidos computacional.

- **SolidEdge** [271]: Es un programa desarrollado por SIEMENS que permite la creación de prototipos 3D utilizando lo que se denomina tecnología sincrónica. Este método de modelado 3D le da al usuario la libertad de cambiar entre modelado directo y paramétrico, en el que se tiene en cuenta el historial de diseño.

También proporciona varias herramientas de simulación, como es el caso de Solid Edge Simulation [270], que permiten procesar los modelos creados para simulaciones de esfuerzo, movimiento completo o vibración.

- Adams [214]: Es un software propiedad de MSC Software que ofrece, además de un editor gráfico para el diseño de sistemas multicuerpo, una herramienta para el estudio dinámico de las piezas móviles y la distribución de cargas y fuerzas en los sistemas mecánicos.
- Ansys [17]: Está dividido en tres herramientas principales llamados módulos: preprocesador (creación de geometría y mallado), procesador y postprocesador. Tanto el preprocesador como el postprocesador están provistos de una interfaz gráfica. El procesador permite la solución de problemas mecánicos e incluye el análisis de estructuras dinámicas y estáticas (ambas para problemas lineales y no lineales).

Del estudio de las herramientas anteriormente citadas se observa que las más avanzadas (Ansys, Adams, SOLIDWORKS, SolidEdge, Abaqus) facilitan la creación y representación de sistemas mecánicos de carácter general (lazos abiertos / cerrados) con sólidos rígidos o flexibles y mediante interfaces de usuario muy bien desarrolladas. Resuelven bastante bien la cinemática y la dinámica de sistemas no demasiado complejos y generan informes de resultados de gran calidad para uso académico y comercial. Tienen un gran público, ya que integran varias fases del desarrollo del producto, desde su modelado hasta su fabricación en máquinas de control numérico. Sin embargo, estas aplicaciones son comerciales, con licencias costosas solo al alcance de empresas con suficiente facturación en diseño de maquinaria. Son herramientas cerradas con las que no se puede interactuar para mejorar un motor de cálculo o incorporar nuevas funcionalidades, como por ejemplo optimización o síntesis cinemática.

Otras aplicaciones, como Gazebo, MORSE, V-REP y ROBODK, son de acceso abierto pero los interfaces de usuario o las capacidades de análisis no están tan desarrollados como en las aplicaciones comerciales. Por ejemplo, algunas solo resuelven cinemática inversa de sistemas mecánicos en lazo abierto (orientados a robótica) con modelos de brazos robóticos desarrollados por los propios desarrolladores o modelos ya creados sobre los que se pueden introducir pequeñas modificaciones. Las más avanzadas incluyen motores de dinámica y representación 3D mediante comunicación con software de terceros. La ventaja de estas aplicaciones reside en que son abiertas y un analista/desarrollador puede incluir sus propias rutinas de análisis.

Muchos grupos de investigación, en vez de conformarse con las limitaciones del software que se ofrece bajo licencia libre, prefieren desarrollar sus propias aplicaciones para simulación dinámica de MBS, por la completa flexibilidad que les permite su adaptación al tipo de problemas que les interesa resolver, utilizando rutinas cada vez más eficientes y sin depender de desarrolladores externos.

En este sentido, la herramienta PARCSIM que se presenta en esta tesis no está, en su estado actual, orientada específicamente a la simulación cinemática o dinámica de sistemas mecánicos, sino a la optimización de las rutinas, en función del hardware disponible, que tanto el software comercial como el de desarrollo propio utilizan para el modelado, diseño, análisis y simulación de sistemas multicuerpo y, aún en un sentido más amplio, para la optimización de rutinas empleadas en la resolución de problemas basados en el álgebra lineal. A modo de ejemplo de aplicación, en esta tesis se describe el uso PARCSIM para la optimización de rutinas de simulación cinemática de sistemas multicuerpo, tratando de sacar el máximo partido a su estructura cinemática.

1.2.3 Arquitecturas de hardware paralelo

Durante años el avance en el rendimiento de los ordenadores se producía de la mano del incremento de la frecuencia del reloj interno y del número de transistores integrados en los procesadores. En este sentido, la ley de Moore [210] ha sido el referente al predecir en el año 1965 que la complejidad de los circuitos integrados se duplicaría cada año, al menos durante dos décadas. Sin embargo el propio Moore, cofundador de Intel©, modificó en el año 1975 el enunciado de la ley para adecuarla al ritmo de evolución que mostraba la industria [211], afirmando que el número de transistores por unidad de superficie en los circuitos integrados se duplicaría cada 24 meses aproximadamente. Sirva como ejemplo el dato de que el primer microprocesador tal y como lo entendemos hoy en día fue el Intel 4004, que constaba de 2200 transistores, y el mismo fabricante presentó en 2003 el super-chip Power4+, que incorporaba 184 millones de transistores.

Sin embargo, la física impone límites en la escala de integración de los chips. Por un lado, surge la dificultad a la que se enfrentan los electrones al tener que circular por conexiones a las que la escala de integración obliga a que sean cada

vez más finas. Por otro lado, a medida que aumenta el número de transistores en el chip y la frecuencia a la que trabajan, también crece la potencia que necesitan y el calor que deben disipar. A raíz de estas consideraciones, se han producido estudios tendentes a revisar el fin de la validez de esta ley [188, 264, 285].

La alternativa a la escala de integración, que ha permitido mantener un aumento de la velocidad de cómputo, ha consistido en aumentar el número de unidades de proceso, tanto en grandes *mainframes* como en ordenadores de consumo. Primero fueron los coprocesadores matemáticos. Después Intel© diseñó la tecnología Hyper Threading [153] para simular dos microprocesadores lógicos dentro de uno físico. Esto permitió procesar hilos o subprocesos en paralelo en el procesador para aprovechar mejor sus recursos. La siguiente gran revolución fue la tecnología de doble núcleo en un mismo chip. Inicialmente AMD lanzó la serie Athlon X2 [12], seguido por Intel© con la serie Core [157]. Esta tendencia ha continuado y los ingenieros trabajan en incrementar el número de cores en los procesadores, permitiendo a los programadores el aprovechamiento de todos ellos mediante el uso compartido de la memoria del ordenador.

Por otro lado, la llegada de las tarjetas gráficas o GPUs [169, 237] ha permitido el uso de sus coprocesadores, originariamente optimizados para el procesamiento gráfico, para realizar tareas más generales en el ámbito del álgebra lineal, operaciones con matrices y, en general, en áreas dominadas por operaciones sobre datos en paralelo. Por este motivo se han convertido en lo que actualmente se denominan como GPUs de propósito general, o GPGPUs. Además, es posible la virtualización de GPUs [272], de manera que varias máquinas virtuales pueden acceder directamente a la capacidad de procesamiento de gráficos de una única GPU física.

Intel© dio un paso más en la evolución hacia el procesamiento paralelo con la introducción del concepto de Many Integrated Core (*MIC*) [154], en el que los núcleos del procesador son, en realidad, otros procesadores multinúcleo. Por ejemplo, los procesadores Intel Xeon Phi [156, 253] de la serie Knights Landing tienen en su interior 72 procesadores Intel Atom [152] de cuatro núcleos cada uno. AMD, a su vez, avanzaba con su estrategia de sistemas APU (*Accelerated Processing Units*) en su serie Fusion [11], consistente en integrar una CPU y una GPU en el mismo componente.

Finalmente, cabe mencionar la posibilidad que existe de interconectar varios multicores compartiendo sus recursos de memoria y computación [21, 173], entrando así en un escenario en el que problemas científicos y de ingeniería complejos, o simplemente con un elevado coste computacional, se pueden resolver en tiempos razonables.

En resumen, podemos encontrar plataformas que incluyen CPUs multicore, GPUs, Intel Xeon Phi, sistemas masivamente paralelos y con la posibilidad de virtualización de GPUs. Un listado actualizado con las más potentes se puede consultar en la lista TOP500 [25].

1.2.4 Programación paralela

Uno de los principales retos a los que se enfrenta la computación de altas prestaciones, o *HPC*, es el desarrollo de algoritmos eficientes que aprovechen el potencial que ofrecen las actuales arquitecturas heterogéneas paralelas. El mejor aprovechamiento comienza por poder descomponer los problemas en tareas independientes cuyas instrucciones puedan ejecutarse de manera simultánea en varios elementos de proceso. Pero el aumento de complejidad en el hardware lleva emparejada una dificultad a la hora de programar de manera adecuada para obtener las máximas prestaciones de las modernas plataformas.

Por ello, de la mano de la evolución del hardware, se ha producido una evolución en las técnicas de programación paralela, especialmente desde los años 90, y que ha sido recogida en publicaciones que han abordado el asunto tanto desde un punto de vista general [7, 91, 137, 248, 250, 269, 298], como en su aplicación a problemas numéricos y científicos [34, 109, 178, 265] y, más específicamente, en la aplicación del paralelismo al álgebra lineal y computación matricial [26, 209], ampliamente utilizadas en la resolución de muchos problemas de alto coste computacional.

Un factor clave que determina la diversidad de tipos de entornos de programación es la arquitectura de la memoria subyacente en el hardware y el modo en el que puede ser accedida. En este sentido podemos diferenciar entre modelos de memoria compartida y distribuida.

En el caso de memoria compartida, se dispone de la librería Pthreads de C [223] y OpenMP [6], que ofrecen una interfaz estándar para la programación multiproceso en este tipo de sistemas. Ambas tienen como base el modelo fork-join, consistente en dividir tareas pesadas en varios hilos (fork) y esperar a recoger los resultados de todos ellos en un resultado consolidado (join). En concreto OpenMP, además de poder especificar el número de hilos de ejecución, implementa el método denominado Thread Affinity [155], que permite al programador enlazar explícitamente cada hilo o thread a una unidad de procesamiento física.

Para sistemas con memoria distribuida encontramos las librerías PVM [123] y MPI [238, 274]. Este último proporciona el estándar actual para la programación mediante paso de mensajes y de él existen múltiples implementaciones, siendo OpenMPI [231] la más difundida. Este método de programación paralela permite la portabilidad a una gran variedad de sistemas, desde máquinas con memoria compartida hasta redes de estaciones de trabajo. A cada nodo de procesamiento se le asigna un número de identificación único, denominado rango. Todos los nodos tienen una copia del mismo programa, pero cada proceso ejecuta distintas sentencias del código según bifurcaciones introducidas que hacen referencia al rango. Este modelo presenta particularidades adicionales según la rutina de comunicación utilizada, pudiendo tener escenarios donde el transmisor y el receptor necesiten estar *on-line* para que un proceso espere respuesta del otro o, por el contrario, es posible que el emisor envíe la información a un *buffer* para su entrega posterior, permitiendo que el emisor pueda seguir ejecutando código sin preocuparse de si el receptor ha recibido o no los datos.

En cuanto a las GPUs, al comienzo de su incorporación a la computación paralela, los programas tenían que usar extensiones de conocidas APIs de desarrollo de gráficos (como OpenGL [170] y DirectX [207]), lo que obligaba a los programadores a tener conocimientos en el manejo de gráficos para aprovechar todo su potencial. Sin embargo, al abrigo de la investigaciones iniciadas en este campo, se han ido desarrollando varios lenguajes de programación y plataformas para realizar cómputo de propósito general sobre GPUs. Es el caso del entorno de desarrollo CUDA [206, 224] para la programación de las GPUs de NVIDIA, y OpenCL [171, 280], que se pretende erigir como el estándar para GPUs de diferentes fabricantes.

El avance en este área continúa y ya se están ofreciendo soluciones para las plataformas con GPUs virtualizadas con rCUDA [51, 97], y optimizaciones sobre los entornos de programación paralela existentes, como en la mejora de las transferencias CPU \longleftrightarrow GPU [294] o el nuevo MPI 3.0 con comunicaciones en una dirección [88]. Este sistema permite que las operaciones de comunicación no sean de bloqueo, de manera que un proceso remoto puede seguir computando sin tener que esperar la llegada de nuevos datos. De esta manera un proceso puede tener acceso directo al espacio de direcciones de memoria de otro remoto sin la intervención de aquél.

Aparte de su aplicación en la simulación de sistemas multicuerpo que se trata en esta tesis, el rendimiento que ofrece la computación paralela en problemas con un alto coste computacional ha hecho que sea muy demandada en el ámbito científico. Podemos encontrar casos de uso en un gran número de disciplinas:

- En bioinformática: Se ha empleado en la predicción de energías y modos de enlace entre ligandos y proteínas, técnica conocida como *docking* de moléculas [145, 204, 257]. También se encuentran aplicaciones en *virtual screening* [144], analizando grandes bases de datos o colecciones de compuestos para identificar posibles candidatos para un determinado fin, y en especial su aplicación en el descubrimiento de fármacos [105, 203, 260]. Otros ejemplos los podemos encontrar en el análisis de código genético [174, 267], en la búsqueda de similitudes entre proteínas [54] y en el estudio del comportamiento del cerebro humano [172, 286].
- En ingeniería: Se ha aplicado paralelismo al análisis en las áreas de electricidad y electromagnetismo [10, 177], para la representación del terreno [30], en el modelado hidrodinámico de costas y océanos [186] o en ingeniería agrícola [89] para la representación de relieves. El manejo de información climática (como por ejemplo la temperatura) y la gestión de recursos hídricos también son áreas que se han beneficiado de las técnicas de paralelismo [190].
- En economía y estadística: Se encuentran aplicaciones en el análisis envolvente de datos [20, 129], autoregresión vectorial para la búsqueda de interdependencias lineales entre múltiples series de tiempo [71], en modelos lineales [179], así como en modelos de ecuaciones simultáneas [128].

- En problemas de optimización: En optimización combinatoria [282] y en paralelización de métodos metaheurísticos [81, 98, 102, 283].
- En Big-Data: Donde la computación paralela se hace imprescindible para acelerar el procesamiento de enormes cantidades de datos [22, 252].
- En el campo de la inteligencia artificial, de cara a abordar la importante carga computacional del *deep learning* [31].
- Por último, con carácter general, existen trabajos sobre paralelización de rutinas básicas que se utilizan en otras disciplinas, como son la transformada Wavelet [32], la transformada rápida de Fourier [113] y el trazado de rayos en procesamiento gráfico [246].

1.2.5 Librerías de álgebra lineal

En el ámbito científico es habitual encontrar algoritmos de resolución de problemas cuyas operaciones básicas de computación se llevan a cabo con rutinas de álgebra lineal [161] operando sobre matrices [127]. Estas rutinas se encuentran disponibles en librerías que han evolucionado a partir de un conjunto de bloques básicos de computación que definen un estándar conocido como BLAS (*Basic Linear Algebra Subroutines*) [37, 90, 94] y que consta de 3 niveles: BLAS en su nivel 1 se encarga de las operaciones del tipo vector-vector, en su nivel 2 de las operaciones matriz-vector y en el nivel 3 de operaciones entre matrices. Gracias a su portabilidad y eficiencia, BLAS se ha usado en el desarrollo de otras librerías de álgebra lineal, como es el caso de LAPACK (*Linear Algebra PACKage*) [14, 38].

Con objeto de facilitar la portabilidad hacia cualquier tipo de plataforma de memoria distribuida se desarrolló BLACS, *Basic Linear Algebra Communication Subroutines* [36, 95], una librería de comunicaciones para aplicaciones de álgebra lineal que se ejecutan en multiprocesadores con paso de mensajes. BLACS, junto a BLAS, permitieron el desarrollo de PBLAS, *Parallel BLAS* [40, 57], para ofrecer un conjunto de versiones paralelas de las rutinas para memoria distribuida similares a las que proporciona BLAS para memoria compartida, con los mismos 3 niveles de BLAS: nivel 1 para operaciones vector-vector, nivel 2 para operaciones matriz-vector y nivel 3 para operaciones matriz-matriz. PBLAS se usa en

ScaLAPACK [35, 184], con el apoyo de BLACS para las operaciones de comunicación, ofreciendo un rendimiento optimizado en este tipo de plataformas. Esto último se consigue básicamente al minimizar los movimientos de datos gracias a la implementación de algoritmos que trabajan por bloques.

Trabajos adicionales han buscado adaptaciones de esas librerías a los nuevos sistemas de computación, como en el caso de PLAPACK (*Parallel LAPACK*) [9, 28, 290], que aporta una variante a la hora de distribuir los datos cercana al problema físico a resolver cuando se trabaja en sistemas de memoria distribuida. También se encuentra una optimización de ScaLAPACK para sistemas heterogéneos [251] y alguna propuesta de librerías para computación en grid [240].

La evolución seguida en el desarrollo de las librerías clásicas puede verse de manera jerárquica a través del esquema representado en la figura 1.2, donde se muestran las dependencias descritas.

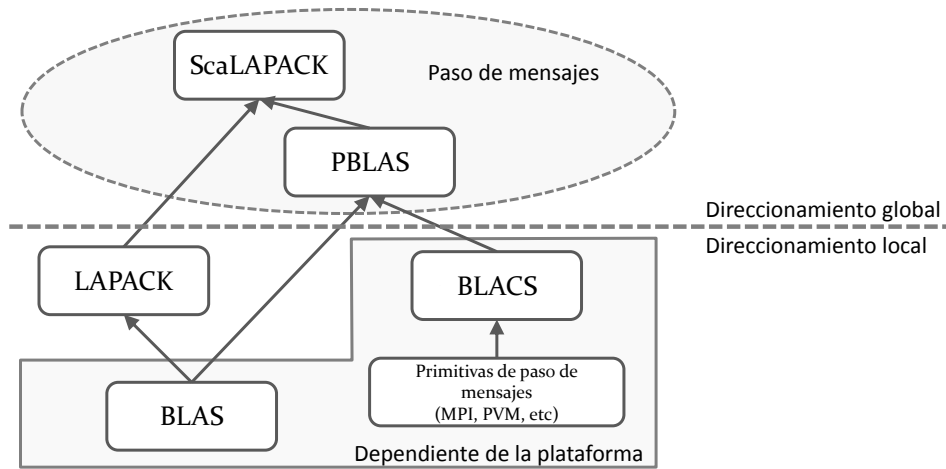


Figura 1.2: Librerías matriciales clásicas y sus dependencias.

Algunos fabricantes, como es el caso de Intel©, han realizado adaptaciones de algunas de estas librerías para una ejecución optimizada en su familia de procesadores multicore y manycore. Es el caso de la librería MKL [150] (*Math Kernel Library*), creada en 2003 tomando como base BLAS, LAPACK y ScaLAPACK, pero que incluye también solucionadores dispersos, transformadas rápidas de Fourier y matemática de vectores.

Igual sucede con NVIDIA©, que ha creado una arquitectura de computación paralela denominada CUDA™ [226], sobre la que han evolucionado nuevas librerías adaptadas a sus modelos de GPU. Es el caso de cuBLAS [225, 227, 228], desarrollada por el propio fabricante. Pero también las surgidas como iniciativas promovidas por terceros, como es el caso de la adaptación de LINPACK [39] para CUDA [106, 293], o de terceros en colaboración con la propia NVIDIA, como es el caso de la creación de CULA [99].

Las recientes plataformas de hardware híbridas han favorecido nuevos desarrollos, como es el caso de PLASMA (*Parallel Linear Algebra Software for Multicore Architectures*) [147] y MAGMA (*Matrix Algebra on GPU and Multicore Architectures*) [5, 146] para multicore y multiGPU, que incorpora los recientes desarrollos en algoritmos híbridos y de planificación para minimizar los costes de sincronización y comunicación. Se encuentran adaptaciones de MAGMA para Xeon Phi [92], y otras propuestas nuevas, como Chameleon, para virtualización de recursos en entornos multicore+multiGPU [4, 142, 300].

Otros estudios se enfocan en algún tipo concreto de operación matricial. Un caso notable es la multiplicación de matrices, que constituye el componente principal sobre el que se implementan otras rutinas de álgebra lineal de mayor funcionalidad. De ahí que esta operación haya sido revisada para su optimización en diversas plataformas, como multicore [160], empleando GPUs [181, 218, 230] y en sistemas heterogéneos y de altas prestaciones [8, 289]. Además de la multiplicación, otros casos de operaciones básicas optimizadas para distintos sistemas son la evaluación de polinomios matriciales [43], la multiplicación de Strassen [125, 134, 135] y los sistemas de ecuaciones [205, 236].

1.2.6 Paralelismo en simuladores de sistemas mecánicos

Esta tesis está centrada principalmente en el estudio de algoritmos de análisis cinemático de sistemas multicuerpo, los cuales se basan en la resolución numérica de los sistemas de ecuaciones que describen sus modelos, siendo habitual que se realicen los cálculos durante varias iteraciones con objeto de explorar todas o un conjunto de las posiciones que pueden adoptar los elementos que constituyen los sistemas simulados. Dada la evidente carga computacional de estos algoritmos,

se busca su optimización a través de un uso completo de todos los recursos del hardware. El modo más directo consiste en usar todos los cores disponibles cuando se emplean plataformas multicore.

Los entornos de simulación de sistemas mecánicos disponibles han mostrado algunos avances recientes hacia el uso eficiente de los recursos del hardware por medio del uso de paralelismo. Por ejemplo, Gazebo maneja el concepto de *Isla* [121], el cual simboliza una agrupación de componentes de un sistema (los cuerpos y uniones que lo forman) que se pueden separar del resto, y que por tanto se pueden calcular en paralelo. En V-REP solamente es posible ejecutar en threads diferentes los procesos que simulan sensores de distancia, de colisión y de proximidad. Sin embargo se permite tener varias instancias de V-REP ejecutándose en modo síncrono, donde cada paso de la simulación se activa desde una aplicación externa mediante el uso de una API para llamadas remotas [65], lo que permite abordar un enfoque modular [110] a la hora de simular varios sistemas multicuerpo.

Por otro lado, MATLAB® proporciona directivas de programación paralela a través del *Parallel Computing Toolbox* [198] sin necesidad de programación MPI o CUDA. Eso se consigue gracias a la ejecución de aplicaciones en *workers* (los motores de cálculo de MATLAB), encargados de activar los cores disponibles. También está disponible el entorno *MATLAB Parallel Server* [199] que permite, sin modificar el código, ejecutar las mismas aplicaciones en clusters o nubes, y también realizar cálculos matriciales que no caben en la memoria de un único equipo. Algunas funciones de MATLAB® se ejecutan automáticamente en paralelo en presencia de estos entornos. Algunos módulos completos, como los estadísticos, los de optimización y los de visión y procesamiento de imágenes, ya están diseñados para aprovechar el paralelismo.

En el caso de MapleSim™, MapleSoft® ha añadido a su software un modo paralelo [194] siguiendo un esquema de memoria compartida. La programación sigue el modelo denominado *Task Programming*, donde se crean tareas en lugar de threads. Una tarea es una llamada a una función que se puede ejecutar en un thread. El software asigna automáticamente las tareas a threads. Recientemente se ha introducido el *Grid Package* [195], que permite el cómputo paralelo multiproceso distribuido en un cluster o en una red.

En este mismo sentido, el simulador académico MORSE ofrece la posibilidad de implementar la simulación en una infraestructura distribuida de múltiples nodos por medio de un programa de servidor cuya tarea es sincronizar los eventos que ocurren entre ellos [24]. Pero esos nodos de simulación solo pueden resolver un sistema completo, por lo que está enfocado en la simulación de una escena donde existen varios robots trabajando colaborativamente.

Ansys ofrece el paquete Ansys HPC [16], *High Performance Computing*, que incluye paralelismo tanto en su resolutor disperso directo como en el de ecuaciones iterativas, y Adams incorpora en su módulo de análisis estructural MSC Nastran [215] técnicas de paralelización, tanto de memoria compartida como distribuida. Por contra, los algoritmos usados por SOLIDWORKS y por SolidEdge no incorporan en este momento técnicas de paralelismo.

Por lo tanto, los entornos de simulación de sistemas multicuero comerciales no han extendido de manera generalizada el uso de paralelismo en sus motores físicos para el cálculo cinemático. Esto se debe principalmente a que este tipo de software, o bien implementa una formulación global, obteniendo y resolviendo las coordenadas y ecuaciones del sistema en su conjunto, con lo que no se aprovecha la topología del sistema a analizar, o bien, aun utilizando una formulación topológica (por ejemplo, en coordenadas relativas) no se explota esa topología en un paralelismo de primer nivel.

Es en el entorno académico donde se observa un mayor interés y desarrollo en la aplicación de la computación paralela a la dinámica de sistemas multicuerpo. Una revisión detallada de estos avances escapa al ámbito de esta tesis debido fundamentalmente a dos razones. La razón de mayor peso es que la mayoría de los trabajos consultados enfocan las técnicas de paralelización al problema dinámico, que tiene particularidades muy distintas a las del problema cinemático que se trata en esta tesis. Por ejemplo, en el problema dinámico, dependiendo de la formulación utilizada, el sistema mecánico se debe abrir por una o varias uniones entre sólidos y se deben introducir términos adicionales en las ecuaciones del movimiento relacionados con las fuerzas de ligadura que, actuando en esas uniones, se encargarían de mantener el mecanismo ensamblado en todo instante de tiempo [120]. Las formulaciones globales abren el sistema por todas las uniones del mecanismo y algunas, más elaboradas, utilizan el algoritmo conoci-

do como *divide and conquer* (DCA) para aprovechar, en la formulación global, la eficiencia demostrada de técnicas recursivas propias de formulaciones topológicas [15, 108, 191]. Dado que son estas fuerzas de ligadura las que mantienen ensamblado el mecanismo, no es preciso resolver el problema de posición, que es el que más tiempo consume en un análisis cinemático. Otras particularidades del análisis dinámico que no aparecen en el cinemático y que condicionan las técnicas aplicables de paralelismo son la selección y configuración del integrador de las ecuaciones del movimiento, problemas de disipación de la energía mecánica, o el paso del sistema mecánico por posiciones singulares.

La segunda razón es que la conclusión a la que llegan la mayoría de los trabajos consultados, tras analizar las mejoras en el rendimiento de sus algoritmos de computación paralela al problema dinámico, es que el mejor rendimiento de una arquitectura paralela depende de un número muy elevado de factores y que, en cada problema a resolver, se debería seleccionar la combinación de recursos que ofrezca la mejor solución posible en términos de coste computacional, robustez y precisión en los resultados. Se habla, por tanto, de la necesidad de una simulación inteligente [69] y este es, en efecto, el principal objetivo de esta tesis.

Aun sin profundizar en la bibliografía sobre la aplicación de algoritmos de computación paralela a la dinámica de sistemas multicuerpo, sí se han podido extraer varias consideraciones de interés para esta tesis. Algunos autores [69] proponen como mejoras en el rendimiento la introducción de librerías de álgebra lineal para matrices dispersas combinado con la opción de paralelismo implícito cuya responsabilidad cae sobre el compilador.

Un análisis más exhaustivo del uso de diferentes librerías de álgebra lineal para matrices densas y dispersas combinadas con paralelismo basado en OpenMP se puede encontrar en [130]. Los autores prefieren OpenMP a MPI por ser menos intrusivo a la hora de su aplicación a código propio desarrollado por otros grupos de investigación. Experimentos aplicando un determinado tipo de formulación dinámica a cuadriláteros escalables con tamaños de problema de 100 a 4000 variables y matrices con diferentes porcentajes de dispersión permiten concluir que los speed-ups que se pueden conseguir dependen principalmente del tipo de problema, de la formulación y de la librería empleada.

Con la misma formulación dinámica se implementa paralelización mediante OpenMP en dinámica de sistemas multicuerpo en [191]. Los autores utilizan el mencionado algoritmo DAC y en relación con la implementación de paralelismo comentan que su eficiencia depende fuertemente de la generación de la topología, siendo la verdadera clave de la eficiencia de la paralelización, una buena elección del árbol de ensamblaje-desensamblaje, de acuerdo con la arquitectura específica del hardware tanto en CPU como en GPU. Se citan algunas recomendaciones y bibliografía relacionada con los aspectos a considerar para una buena elección de esta topología.

Otros autores prefieren MPI por su portabilidad a diferentes arquitecturas paralelas [247]. Estos autores basan sus técnicas de paralelismo en la generación de subestructuras que se pueden resolver en paralelo en diferentes CPUs, dejando un sistema de ecuaciones remanente (interface problem) que se deberá resolver finalmente para completar la solución de todo el problema. Para generar estas subestructuras desarrollan un algoritmo encargado de reorganizar las matrices del sistema generando estos subproblemas de menor tamaño y paralelizables [247, 295]. Aplican este algoritmo, entre otros, al modelo de un rotor activo de helicóptero con tamaños de matrices de entre 584 y 5228 elementos. El algoritmo trata de dividir el problema en un conjunto de N_p subproblemas, donde N_p es el número de procesadores disponibles, y se comprueba que los resultados más eficientes no siempre se obtienen subdividiendo en el mayor número de procesadores disponibles. No se analizan otros parámetros que puedan influir en la búsqueda de la solución más eficiente.

Un enfoque distinto a los indicados, de interés para la optimización de rutinas de cinemática computacional es el de aplicar paralelismo en el dominio del tiempo, en vez de en el dominio del espacio. Se trata de reescribir las ecuaciones del movimiento (o del análisis cinemático) considerando que, en vez de resolver un mecanismo durante un intervalo de tiempo, se replica ese mecanismo un número de veces igual al número de subintervalos en el que se ha dividido el tiempo total de análisis y se resuelven en paralelo todos esos mecanismos en sus subintervalos correspondientes [67]. Esta metodología se propone para arquitecturas basadas en cientos o miles de CPUs y, llevado al extremo, se resolvería el mecanismo en todos los posibles instantes de tiempo simultáneamente.

A la luz de estas consideraciones parece justificado el interés en estudiar problemas de diferente tamaño, con diversas librerías densas y dispersas, con capacidad de paralelismo implícito y explícito basado en OpenMP (menos intrusivo) y con diferentes arquitecturas paralelas. También se observa que una herramienta como PARCSIM es de gran utilidad a la hora de identificar cuál es la configuración de recursos que permite el mejor aprovechamiento de los recursos disponibles, dependiendo del tipo de problema a resolver, es decir, dado un sistema multicuerpo concreto, cuál es la simulación más inteligente que se puede lanzar en un equipo determinado.

1.2.7 Autooptimización

Dada la heterogeneidad en las plataformas hardware y la gama de librerías de computo disponible, se puede concluir que la búsqueda de los mejores tiempos de ejecución en la resolución de un determinado problema por medios computacionales puede ser una tarea compleja. Con carácter general, un software desarrollado para manejar un determinado juego de datos sobre un hardware concreto, y haciendo uso de unas determinadas librerías, puede ofrecer un rendimiento muy diferente al ejecutarse en otros sistemas, o simplemente cambiando el tipo y tamaño de los datos.

A veces, disponer de alguna técnica de predicción de las prestaciones de los algoritmos [115] puede ser suficiente para decidir la tecnología óptima a emplear. Pero una solución más ambiciosa, que garantice siempre los mejores resultados para un determinado hardware, requiere de alguna técnica de autooptimización. Esta técnica de naturaleza empírica busca maximizar el rendimiento de una aplicación software en una gran variedad de arquitecturas de ejecución.

En general, el proceso de autooptimización lleva a cabo una exploración del espacio de valores posibles para los parámetros ajustables del software, de manera exhaustiva o aplicando alguna heurística. A continuación activa sobre ellos las optimizaciones conocidas y preimplantadas en el código desarrollado [217, 291]. Un trabajo pionero en este área fue presentado por Brewer en su tesis doctoral de 1994 [44].

Se pueden encontrar estudios con propuestas de aplicación al ámbito de los resolutores de naturaleza numérica [168, 259], pero más abundantes lo son en el caso de su aplicación a rutinas del álgebra lineal [70, 73, 74, 76, 80, 49, 216], motivado por el uso generalizado de este tipo de librerías en la resolución de numerosos problemas de naturaleza científica. Algunas iniciativas han conducido a implementaciones de librerías con capacidad de autooptimización en sistemas concretos, como es el caso de ATLAS (*Auto-Tuning Linear Algebra Subroutines*) [297], que suponen una mejora del conocido conjunto de rutinas de BLAS al adaptarse automáticamente a las características del sistema con una serie de experimentos exhaustivos en la instalación. Otros trabajos analizan adaptaciones a algún tipo de hardware, como multicore [74, 77], manycore [119] o sistemas heterogéneos [165, 182], o se especializan en algún tipo especial de datos, como las matrices dispersas [258].

1.3 Objetivos

El objetivo principal de esta tesis consiste en el desarrollo, optimización y aplicación de propuestas de computación paralela para la resolución de sistemas multicuerpo en el campo de la ingeniería mecánica, así como la extensión de estas propuestas a otros tipos de problemas computacionales que cuenten con una estructura similar, esto es, que conlleven la resolución simultánea de un conjunto de subproblemas de álgebra lineal.

Para ello se han establecido los siguientes subobjetivos específicos:

- Análisis de técnicas que permitan optimizar la computación de las simulaciones de sistemas multicuerpo, MBS (*Multibody Systems*), en plataformas de computación de memoria compartida (CPU multicore), así como en plataformas híbridas conformadas por una CPU multicore junto a una o varias GPUs actuando como aceleradores de cómputo.
- Desarrollo de un conjunto de técnicas de autooptimización que permitan aumentar la eficiencia computacional de las simulaciones sin que sea necesaria la intervención del usuario.

- Desarrollo de una herramienta que facilite las simulaciones en las plataformas computacionales anteriormente mencionadas. Esta herramienta deberá permitir el análisis de las técnicas de optimización y autooptimización propuestas en los subobjetivos anteriores, en su aplicación a algunos MBS de ejemplo.
- Análisis de la aplicación de las ideas anteriores y del uso del simulador a otros ámbitos de aplicación, como es el caso de rutinas de álgebra lineal con una descomposición similar a la de la computación que encontramos en simulaciones de MBS.

El éxito de las optimizaciones propuestas será fruto de una combinación de tres factores: el planteamiento del problema, las librerías de cómputo empleadas y el tipo de plataforma computacional. En este sentido, algunas consideraciones a tener en cuenta por su influencia en la consecución del objetivo de esta tesis son:

- En referencia al planteamiento del problema, es necesario emplear alguna metodología que permita modelar el sistema mecánico multicuerpo en forma de una unión de subsistemas que se puedan resolver de manera independiente. En teoría de mecanismos se recurre al análisis estructural para conseguir un modelo formal del sistema mecánico. En otro tipo de problemas científicos se buscarán algoritmos que contemplen la resolución de los mismos mediante la división en un conjunto de rutinas independientes. En problemas así planteados, el simulador podrá analizar las diferentes estrategias de cálculos paralelos.
- En cuanto a las librerías de álgebra lineal, cabe mencionar que en el ámbito de la computación científica es posible disponer actualmente de un amplio conjunto de ellas, con rendimientos dispares en función del tipo y tamaño de las matrices y del hardware paralelo subyacente. Por este motivo, la selección de la librería que mejor se adapta a un determinado tipo de problema será un factor determinante en la consecución de mejoras en los tiempos de ejecución.

- En cuanto al hardware, el uso de librerías que aprovechen todos los cores y GPUs disponibles y la realización de cálculos simultáneos puede resultar la opción preferida a priori con vistas al mejor aprovechamiento de los recursos. Sin embargo, se deben contemplar escenarios en los que puede ser más conveniente, en cuanto a rendimiento, utilizar librerías que no incorporen paralelismo frente a otras que sí lo hagan. Por ejemplo, matrices de tamaño reducido y alta dispersión se manipulan con mucha eficiencia por algunas librerías especializadas que no usan paralelismo, como comprobaremos mediante la experimentación.

Estos aspectos deben ser tenidos en cuenta para el desarrollo del simulador. Este software será un producto final que nos ayudará a realizar algunos de los análisis planteados para los sistemas multicuerpo y a estudiar su posible extensión a otros ámbitos de aplicación. Por este motivo constituirá una parte fundamental del trabajo a desarrollar en esta tesis, y deberá incorporar las siguientes funcionalidades:

- Un interfaz gráfico que permita capturar un grafo representativo del algoritmo de resolución del sistema mecánico o problema de álgebra lineal que se quiere simular. Dicho algoritmo mostrará una secuencia ordenada de grupos compuestos por operaciones sobre matrices.
- Un módulo de análisis de los citados algoritmos que permita identificar las secciones que pueden ser ejecutadas de manera simultánea. Para ello, basándose en las dependencias entre los grupos que se deriven del análisis del grafo del algoritmo, el simulador construirá un árbol de opciones donde cada rama representará una manera diferente de ordenar y agrupar los cálculos.
- Un módulo de simulación que realice los cálculos empleando un conjunto preinstalado de librerías de álgebra lineal optimizadas para diferentes tipos de matrices y arquitecturas de hardware. Seleccionada una librería, y conocida la configuración del hardware (número de cores y número de GPUs instaladas), el simulador ejecutará los cálculos siguiendo las diferentes opciones de ordenación marcadas por el usuario, registrando los tiempos de ejecución obtenidos en cada simulación.

- Una herramienta de análisis de los resultados de la experimentación que permita comparar los tiempos de ejecución obtenidos usando diferentes librerías, y siguiendo diferentes ramas de ejecución. Esta información ayudará al usuario a la selección de la mejor instalación del software de cara a futuras ejecuciones en entornos de producción.
- Desarrollar una estrategia de autooptimización que determine de manera teórica la mejor agrupación de los cálculos en una determinada arquitectura de hardware y para un determinado tamaño y tipo de datos. Se usarán como base para las estimaciones los tiempos de ejecución que ofrecen las librerías de álgebra lineal instaladas, los cuales se habrán obtenido variando los parámetros de paralelismo y los tamaños y dispersión de las matrices. El proceso de selección autooptimizado usará esa información para recomendar la librería a emplear y los recursos hardware (cores y GPUs) que se deben asignar en cada etapa de la simulación.

1.4 Metodología

La presente tesis amplía la investigación realizada en el trabajo de fin de máster de título *Optimización de algoritmos paralelos para análisis cinemático de sistemas multicuerpo basado en Ecuaciones de Grupo* [50], en el que se analizaba el software desarrollado por el Departamento de Ingeniería Mecánica de la Universidad Politécnica de Cartagena para el control de un robot manipulador paralelo de seis articulaciones denominado plataforma de Stewart.

Dicho software es un ejemplo de la aplicación práctica de los métodos computacionales para la obtención de la estructura cinemática de sistemas multicuerpo [262], así como de formulaciones cinemáticas [52, 263] y dinámicas [266] basadas en ecuaciones de grupo. Mostraba la implementación computacional modular que permite este tipo de formulaciones, donde los problemas de posición, velocidad y aceleración del sistema multicuerpo completo se pueden resolver mediante la solución de los subsistemas que lo componen. El resultado de la investigación desarrollada durante el máster fue un simulador, una versión modificada del mencionado software de control al que se le incorporaba paralelismo.

Sin embargo, el simulador resultante seguía siendo válido únicamente para la resolución de ese tipo específico de sistema mecánico. La metodología seguida en esta tesis nos permitirá salvar esta limitación y alcanzar el objetivo de generalizar el simulador, no solo a otros tipos de sistema multicuerpo, sino permitiendo su extensión a otros problemas más generales cuyo planteamiento pueda ser de naturaleza similar.

Comenzaremos con la generalización del simulador, eliminando la restricción de la resolución de un tipo concreto de sistema multicuerpo. Será necesario desarrollar un motor de ejecución que pueda resolver un problema genérico, que deberá ser especificado mediante un grafo donde los vértices sean bloques que contengan secuencias de cálculos y los arcos indiquen el orden en que deben ejecutarse.

Probaremos el funcionamiento del simulador trabajando inicialmente con el manipulador paralelo de Stewart ya mencionado. Comenzaremos confirmando que todas las funcionalidades están cubiertas y ofrecen resultados correctos. Para ello realizaremos experimentos con varias librerías y parámetros ajustables en una plataforma hardware simple, un sistema con únicamente una CPU multicore. Una vez verificado el correcto funcionamiento probaremos la simulación sobre un sistema hardware más complejo, compuesto de una CPU multicore y un conjunto de GPUs.

A continuación trabajaremos con un problema diferente, un sistema multicuerpo más complejo, siguiendo la misma metodología que en el caso anterior, es decir, realizando simulaciones en plataformas de hardware cada vez más heterogéneas. Tras ello, activaremos la funcionalidad de autooptimización y comprobaremos mediante experimentos la validez de la misma.

Por último, extenderemos el uso del simulador a otro tipo de problemas de álgebra lineal, como por ejemplo la multiplicación de matrices, en su variante por bloques y mediante el algoritmo de Strassen [281].

1.5 Contribuciones

Durante el período de investigación necesario para la elaboración de la presente tesis, se han realizado contribuciones en forma de presentaciones en congresos y publicaciones en revistas. Además se han desarrollado dos aplicaciones de software que permiten, por un lado, la representación en un entorno gráfico de los algoritmos de resolución de problemas científicos planteados en términos de operaciones de álgebra lineal y, por otro, la simulación del algoritmo de resolución de dicho problema. Este software se encuentra disponible en la dirección http://luna.inf.um.es/grupo_investigacion/software. El Anexo A contiene información detallada de las funcionalidades del simulador y una guía de aprendizaje paso a paso basada en un ejemplo de uso.

1.6 Estructura de la tesis

Este documento se ha estructurado de la siguiente forma:

- En el Capítulo 2 se comentan aspectos relacionados con la cinemática computacional de los sistemas multicuerpo. En él se realiza una introducción teórica al modelado de dicho sistemas y al análisis estructural de los mismos. Además se describen las formulaciones cinemáticas para el análisis de una cadena cinemática cualquiera (grupo estructural) en los problemas de posición, velocidad y aceleración, y se presenta la rutina de análisis cinemático de un sistema mecánico complejo a partir de la resolución de la cinemática de sus grupos estructurales.
- En el Capítulo 3 se detallan las herramientas de hardware y software que se han necesitado durante la investigación y elaboración de la tesis. En las herramientas hardware se describe la plataforma computacional sobre las que se han realizado los experimentos. En el apartado de herramientas de software se hace referencia a los lenguajes de programación usados en el desarrollo del software, y los paradigmas de programación paralela, librerías de álgebra lineal y rutinas consideradas en el estudio experimental.

- En el Capítulo 4 se discuten las técnicas de optimización y autooptimización de software utilizadas en la actualidad en el ámbito científico. Asimismo se indican las técnicas básicas que se proponen en el simulador para la búsqueda de la secuencia de cálculos y parámetros algorítmicos óptimos.
- En el Capítulo 5 se describe el software desarrollado en esta tesis. Se muestran sus características principales y funcionalidades, se analizan en detalle las técnicas incorporadas para guiar las simulaciones y gestionar el proceso de autooptimización, y finalmente se hace una introducción a la arquitectura y los componentes del simulador.
- Para validar el funcionamiento del simulador y de las técnicas de optimización empleadas se han realizado una serie de experimentos que se describen en el capítulo 6. Entre ellos se encuentran simulaciones de sistemas multi-cuerpo, tanto en modo directo como a través de un proceso de autooptimización, y experimentos que muestran un caso de extensión de su uso a otros problemas del álgebra lineal, como el algoritmo de Strassen. En todos ellos se incluye un resumen de los resultados experimentales obtenidos y las interpretaciones derivadas de ellos.
- En el Capítulo 7 se recogen las principales conclusiones obtenidas y posibles líneas de trabajo futuro, así como la difusión de resultados (en congresos y publicaciones) llevada a cabo durante el período de investigación requerido por esta tesis.
- Finalmente, el anexo A contiene información técnica sobre el software simulador que se ha desarrollado, los componentes que lo integran, su estructura lógica, tecnologías empleadas y el manual de instalación y de usuario.

Capítulo 2

Cinemática computacional de sistemas multicuerpo

El estudio de sistemas mecánicos ha experimentado una gran evolución gracias al avance de los sistemas computacionales modernos, lo que ha permitido abordar problemas con una complejidad cada vez mayor. Un elevado número de sólidos, superficies de geometría compleja entre elementos que están en contacto, el carácter espacial del movimiento o un elevado número de grados de libertad del sistema son solo algunos de los factores que, considerados individualmente, harían poco práctica una solución basada en los métodos tradicionales. Si se unen varios de estos factores, como es habitual en sistemas mecánicos reales, entonces los métodos computacionales son absolutamente necesarios.

Tal y como se ha descrito en el apartado 1.2.1, la simulación por ordenador se ha convertido en una herramienta crucial en el entorno industrial actual, especialmente dinámico y competitivo, y tiene por objeto el análisis eficiente del comportamiento de un sistema mecánico, principalmente con finalidades de diseño y de control de nuevos sistemas o de sistemas ya existentes.

La simulación eficiente permite reducir los tiempos de diseño, ya que hace posible asignar diferentes valores a un elevado número de parámetros que definen el mecanismo y evaluar cómo repercuten esas modificaciones en el diseño final. Por otro lado, también permite reducir los tiempos de prueba, ya que solo será

necesario crear un prototipo real una vez que las simulaciones han encontrado la solución óptima. Todo ello lleva emparejado un enorme ahorro de costes de diseño, producción, e incluso posventa. Igualmente ocurre a la hora de implementar sistemas de control de mecanismos y máquinas. En estos casos se simulan modelos que deben recibir señales de sensores externos y recalcular, en tiempo real, los controles a ejercer sobre sus actuadores para corregir las mínimas desviaciones que se hayan podido encontrar en el sistema real.

Esta tesis se centra en el campo de la simulación cinemática de sistemas multicuerpo basada en ecuaciones de grupo, quedando fuera de su alcance la dinámica computacional. Para desarrollar los principales objetivos de la tesis se introducen en este capítulo los conceptos fundamentales sobre modelado, análisis estructural y simulación de sistemas mecánicos. El apartado de modelado introduce los conceptos topológicos que describen qué elementos componen los sistemas multicuerpo, así como los tipos de coordenadas y las correspondientes ecuaciones de restricción comúnmente utilizadas para abordar el modelado de dichos sistemas. Seguidamente se introduce el análisis estructural como herramienta para dividir un sistema mecánico en subsistemas, permitiendo explotar al máximo la topología del mecanismo para una simulación más eficiente y, por último, se introduce una formulación cinemática computacional que se puede resolver de forma global o mediante ecuaciones de grupo, y se detallan los algoritmos a implementar para llevar a cabo simulaciones cinemáticas mediante ambos métodos de análisis.

2.1 Modelado de sistemas multicuerpo

Con carácter general, la simulación de cualquier sistema comienza con una fase inicial de modelado en la que se pretende obtener un conjunto de ecuaciones que, desde el punto de vista de lo que se quiere simular, se considera que lo representa de manera adecuada. La exactitud que ofrece el modelo matemático respecto al sistema físico casi nunca se alcanza [47]. La aproximación será mejor en la medida que se hagan menos simplificaciones o, lo que es lo mismo, cuando se considere un mayor número de variables involucradas en el sistema físico, lo cual conlleva ineludiblemente a una mayor complejidad de las ecuaciones del modelo matemático en cuestión.

En la disciplina de los sistemas mecánicos, el modelado implica seleccionar las coordenadas que especifican de forma unívoca la posición y orientación de todos los cuerpos que forman parte de un mecanismo y establecer el conjunto de ecuaciones que expresan las restricciones en el movimiento relativo entre los cuerpos que forman ese sistema.

Las metodologías empleadas para obtener los modelos matemáticos permiten emplear formulaciones bastante precisas que exigen un gran esfuerzo computacional por su carácter de no linealidad. Es por ello que en aplicaciones multicuerpo es de especial interés la eficiencia computacional a efectos de simulación y control de los mecanismos. Esto ha llevado a invertir grandes esfuerzos en el desarrollo de algoritmos que permitan simular mecanismos en tiempo real [107], siendo este un campo de constante estudio debido a la creciente complejidad de los sistemas que se pretende simular [42].

2.1.1 Conceptos básicos topológicos

La teoría de máquinas se ocupa de la descripción de los elementos de las máquinas, de cómo están unidos entre sí y del estudio de las relaciones entre las fuerzas y los movimientos que aparecen entre sus elementos. Dos de los campos de estudio tradicionalmente vinculados a la teoría de máquinas son el análisis y la síntesis de mecanismos. La síntesis consiste en escoger y dimensionar un mecanismo que cumpla o que tienda a cumplir, con un cierto grado de aproximación, unas exigencias de diseño dadas.

Por contra, el análisis consiste en estudiar la cinemática y la dinámica de un mecanismo según las características de los elementos que lo constituyen. Por tanto, el análisis de un mecanismo permitirá, por ejemplo, determinar la trayectoria de un punto de una barra o una relación de velocidades entre dos miembros. Se recurre al estudio topológico de mecanismos para analizar los elementos que los constituyen en cuanto a sus formas, número de elementos, uniones entre ellos, los tipos de movimientos que pueden realizar y las leyes que los gobiernan, en definitiva sobre la configuración geométrica que determinará, a la postre, la movilidad del mecanismo.

A continuación se repasan los conceptos básicos relacionados con la teoría de máquinas que son necesarios en los estudios topológicos de los mecanismos:

- *Pieza* es cualquiera de los elementos indivisible que, junto a otros, permiten fabricar una máquina.
- *Eslabón* es un conjunto de piezas unidas rígidamente entre sí, lo que significa que no puede haber movimiento relativo entre dos puntos arbitrariamente seleccionados de un mismo eslabón, independientemente de la carga externa a que se encuentre sometido.

En los mecanismos planos (2D) cada eslabón puede poseer tres movimientos independientes: dos de traslación, según los ejes x e y de un sistema de coordenadas libremente elegido, y uno de rotación alrededor de un eje z , perpendicular a dicho plano. En los mecanismos 3D, un eslabón puede poseer hasta seis movimientos independientes, los de traslación a lo largo de los ejes x , y , z , y los de rotación alrededor de los mismos ejes. A cada uno de estos movimientos independientes se le llama grado de libertad.

Los eslabones transmiten el movimiento del impulsor o conductor (el eslabón de entrada) al seguidor o conducido (el eslabón de salida).

- *Par cinemático* es un conjunto formado por dos eslabones de tal forma que entre ellos exista la posibilidad de movimiento relativo.

Para describir este concepto se presenta el ejemplo ilustrado en la figura 2.1(a). Se trata de un mecanismo real en el que una biela (2) se encarga de transmitir el movimiento de un pistón (1) que circula dentro de un cilindro por lo que únicamente puede ser un desplazamiento longitudinal (L), a un cigüeñal (3) cuyo movimiento es rotatorio (R). La figura 2.1(b) representa una vista simplificada del mismo sistema, con objeto de facilitar su estudio.

En este mecanismo es posible identificar cuatro parejas de eslabones, como muestra la figura 2.2(a): 0 - 1, 0 - 3, 1 - 2 y 2 - 3. Estos pares representan las conexiones de miembros entre los que puede haber algún tipo de movimientos relativo (figura 2.2(b)). Si tomamos como ejemplo el par 2 - 3 y establecemos que el elemento 2 quede fijo, vemos que el único movimiento posible es el de rotación alrededor suyo.

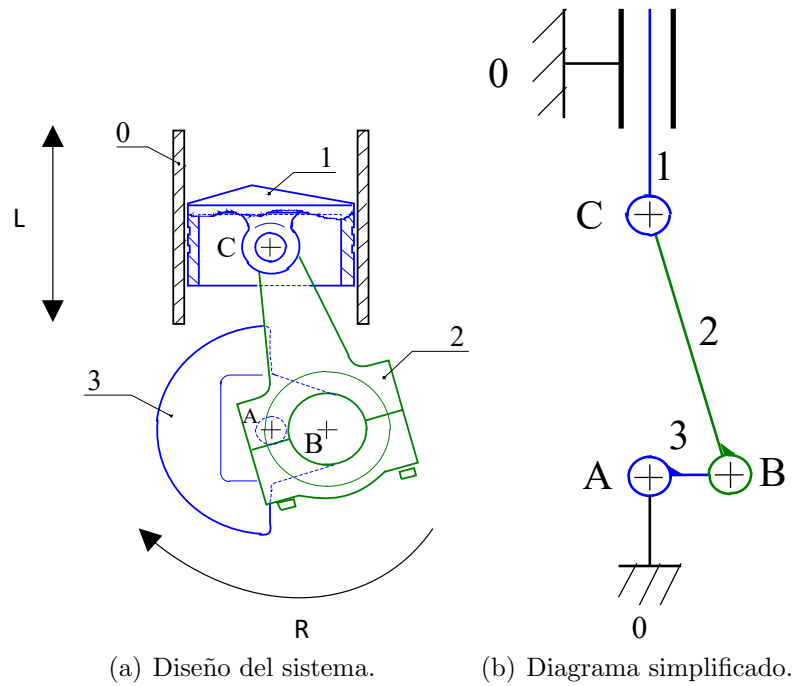


Figura 2.1: Ejemplo de sistema mecánico: mecanismo biela-cigüeñal.

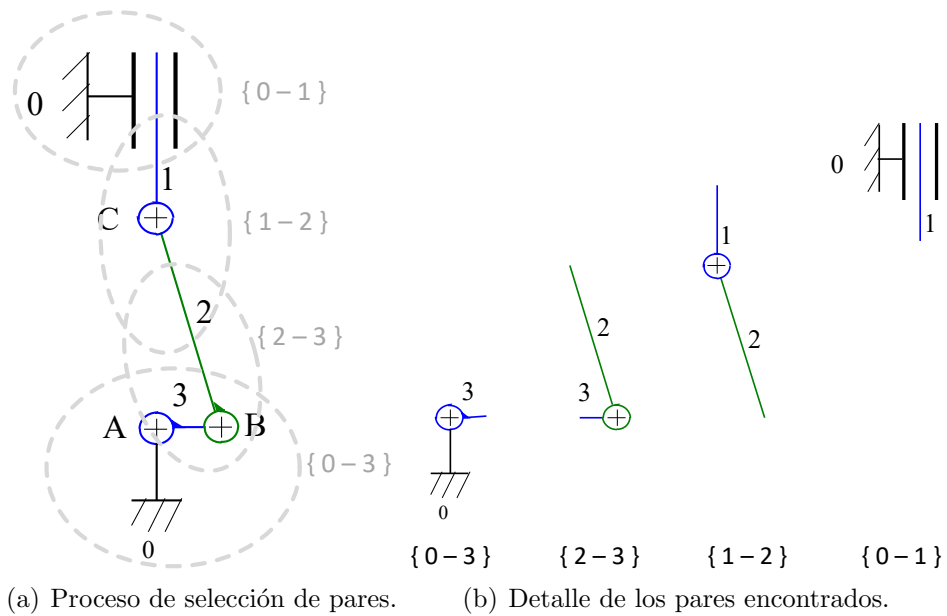


Figura 2.2: Identificación de pares cinemáticos en un sistema biela-cigüeñal.

Por tanto, el hecho de estar unidos los dos eslabones limita el movimiento de uno de ellos respecto al otro. Estas limitaciones se denominan condiciones de enlaces, y su formulación analítica se conoce como ecuaciones de restricción.

Los pares se pueden clasificar atendiendo al grado del par. Se define el *grado* de un par cinemático como el número de grados de libertad que permite el par, es decir, el número de movimientos independientes que un eslabón tiene respecto al otro. Como vimos en el ejemplo anterior del mecanismo biela-cigüeñal, el procedimiento para identificar el grado de un par consiste en separar los eslabones del resto del mecanismo y estudiar su movimiento relativo fijando uno de ellos. El número de movimientos independientes que tenga el eslabón móvil respecto al que se considera fijo establece el grado k del par, siendo e_k el número de grados de libertad que restringe la unión entre los dos sólidos del par. Dado que $e_k \neq 0$ se entiende que, en mecanismos planos, los pares cinemáticos solo pueden ser de grado $k = 1, 2$ y en mecanismos espaciales, de grado $k = 1, \dots, 5$. En la literatura se pueden consultar tablas que muestran todos los tipos de pares cinemáticos que forman los eslabones de mecanismos planos y espaciales [176].

- Una *cadena cinemática* es la formada por la unión de pares cinemáticos. Cuando cada eslabón o miembro pertenece al menos a dos pares, se habla de una cadena cerrada (figura 2.3(a)). En caso contrario tendremos una cadena abierta (figura 2.3(b)).

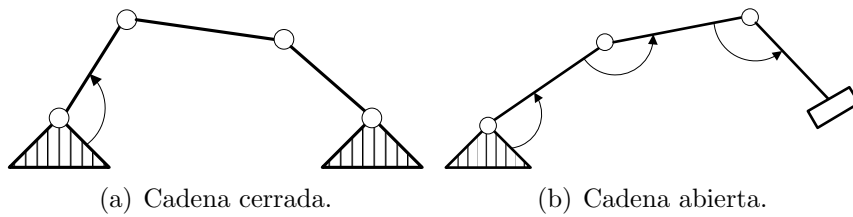


Figura 2.3: Tipos de cadenas cinemáticas en función de la unión entre sus eslabones.

Atendiendo al tipo de movimiento entre sus eslabones, se pueden clasificar en: bloqueada, cuando no es posible el movimiento relativo (figura 2.4(a)), desmodrómica o de cinemática determinada, cuando a un movimiento controlado de sus grados de libertad corresponden movimientos perfectamente

determinados de sus eslabones (figura 2.4(b)). Por último, la cadena es libre si los movimientos relativos de sus eslabones están permitidos, pero no perfectamente determinados (figura 2.4(c)).

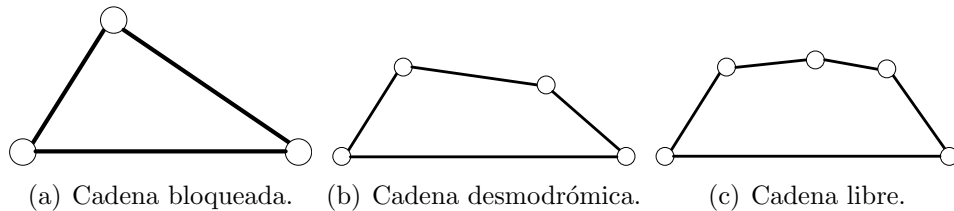


Figura 2.4: Tipos de cadenas cinemáticas según la clase de movimiento.

- Un *mecanismo* es una cadena cinemática en la que se considera como fijo uno de sus miembros (bastidor), al que se elige como marco de referencia para definir, respecto de él, el movimiento de todos los demás sólidos.
- Una *máquina* es la combinación de cuerpos rígidos o resistentes agrupados y conectados de tal modo que permitan transmitir esfuerzos desde la fuente de energía hasta el lugar donde han de ser vencidas las resistencias.

En general, y de forma simplificada, se puede decir que toda máquina está formada por tres elementos principales:

- El elemento motriz: dispositivo que introduce la fuerza o el movimiento en la máquina. Suele tratarse de un motor, un esfuerzo muscular o una fuerza natural (como viento, corriente de agua de un río, etc).
 - Uno o varios mecanismos: encargados de trasladar el movimiento del elemento motriz al elemento receptor.
 - Elemento receptor: recibe el movimiento o la fuerza para realizar la función de la máquina.
- La *movilidad* de un mecanismo es el número de variables independientes que hay que controlar para que su cinemática esté determinada. El cálculo de la movilidad de un mecanismo se puede realizar empleando el criterio de Grübler, que consiste simplemente en realizar una diferencia entre los grados de libertad de los eslabones del mecanismo y las restricciones impuestas por los pares cinemáticos:

$$L = BN_m - \sum_{k=1}^{B-1} e_k P_k \quad (2.1)$$

donde B es el número de grados de libertad introducidos por cada sólido rígido: 3 en sistemas planos, 6 en sistemas espaciales; N_m es el número de sólidos móviles libres; k es el grado del par; P_k el número de pares de grado k y e_k el número de grados de libertad que elimina un par de grado k . El valor e_k se puede obtener consultando la tabla 2.1.

Hay muchas consideraciones que aparecen en los mecanismos y modifican su movilidad que no vienen recogidas en la fórmula de Grübler (ec. 2.1). Casos como la rodadura pura, articulaciones múltiples, grados de libertad redundantes, etc., exigen la introducción de correcciones a esta fórmula. Estas consideraciones no son objeto de la presente tesis y nos limitaremos al estudio de sistemas multicuerpo que sí satisfacen esta ecuación.

k	$e_k(2D)$	$e_k(3D)$
1	2	5
2	1	4
3	-	3
4	-	2
5	-	1

Tabla 2.1: Grados de libertad que eliminan los pares cinemáticos que forman los eslabones del sistema mecánico según su grado k .

2.1.2 Modelado en coordenadas dependientes

Dado un sistema mecánico definido por un conjunto de n coordenadas dependientes y g grados de libertad, el número de ecuaciones de restricción necesarias para relacionar esas coordenadas dependientes entre sí es: $r = n - g$.

Dado que las coordenadas dependientes son las más utilizadas en el modelado de los sistemas multicuerpo, es de interés realizar una breve descripción de los tres tipos que existen: relativas, de punto de referencia y naturales.

2.1.2.1 Coordenadas relativas

Las coordenadas relativas sitúan cada elemento del mecanismo con respecto al anterior en la cadena cinemática. Las ecuaciones de restricción procederán de las condiciones de cierre de los lazos que componen el mecanismo. En la figura 2.5 observamos un solo lazo.

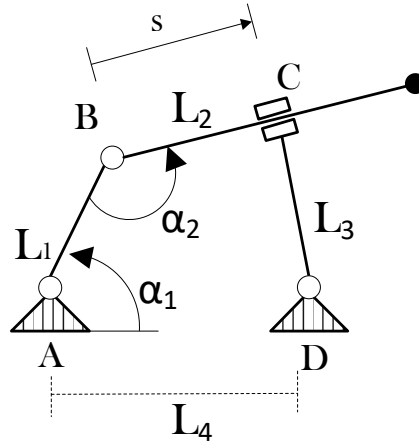


Figura 2.5: Modelado empleando coordenadas relativas.

La condición de cierre será:

$$\vec{AB} + \vec{BC} + \vec{CD} + \vec{DA} = \vec{0}$$

Esta ecuación vectorial se traduce, en el plano, en dos escalares:

$$L_1 \cos \alpha_1 + s \cos(\alpha_1 + \alpha_2 - \pi) + L_3 \cos(\alpha_1 + \alpha_2 + \frac{\pi}{2}) - L_4 = 0$$

$$L_1 \sin \alpha_1 + s \sin(\alpha_1 + \alpha_2 - \pi) - L_3 \sin(\alpha_1 + \alpha_2 + \frac{\pi}{2}) = 0$$

El número de coordenadas necesarias para modelar el mecanismo suele coincidir con el de movimientos permitidos por los pares cinemáticos; en este caso, $n = 3$ y son: $\mathbf{q} = [\alpha_1 \ \alpha_2 \ s]$. Como este mecanismo tiene un grado de libertad, $g = 1$, el número de ecuaciones de restricción es el más bajo posible: $r = 3 - 1 = 2$. Sin embargo, como vemos, las formulaciones a que dan lugar estas coordenadas son complicadas y la evaluación de los términos presentan funciones trigonométricas, lo que incrementa el coste computacional.

2.1.2.2 Coordenadas de punto de referencia

Las coordenadas de punto de referencia sitúan a cada elemento con independencia de los demás. Normalmente se eligen las coordenadas cartesianas de un punto cualquiera del elemento, que suele ser su centro de masas, y la orientación del sólido, lo que da un total de $n = BN_m$ coordenadas dependientes. Las ecuaciones de restricción surgen de imponer las uniones entre los elementos, ya que se han definido inicialmente como si estuvieran libres los unos de los otros. En la figura 2.6 se observa un modelado como el descrito.

En este caso hemos elegido $n = 3 \cdot 3 = 9$ coordenadas dependientes, que son: $\mathbf{q} = [x_1 \ y_1 \ \alpha_1 \ x_2 \ y_2 \ \alpha_2 \ x_3 \ y_3 \ \alpha_3]$, de modo que hay que imponer $r = 9 - 1 = 8$ ecuaciones de restricción, las cuales se detallan a continuación [68]:

$$\begin{aligned}
 (x_1 - x_A) - \frac{L_1}{2} \cos \alpha_1 &= 0 \\
 (y_1 - y_A) - \frac{L_1}{2} \sen \alpha_1 &= 0 \\
 \left(x_1 + \frac{L_1}{2} \cos \alpha_1 \right) - \left(x_2 - \frac{L_2}{2} \cos \alpha_2 \right) &= 0 \\
 \left(y_1 + \frac{L_1}{2} \sen \alpha_1 \right) - \left(y_2 - \frac{L_2}{2} \sen \alpha_2 \right) &= 0 \\
 \alpha_3 - \left(\alpha_2 + \frac{\pi}{2} \right) &= 0 \\
 (x_2 - x_3) \cos \alpha_3 + (y_2 - y_3) \sen \alpha_3 - \frac{L_3}{2} &= 0 \\
 (x_3 - x_D) - \frac{L_3}{2} \cos \alpha_3 &= 0 \\
 (y_3 - y_D) - \frac{L_3}{2} \sen \alpha_3 &= 0
 \end{aligned}$$

Observamos dos ecuaciones de restricción para cada par cinemático (igual a su grado $k = 2$), empezando por el par referenciado como A y siguiendo hasta el D.

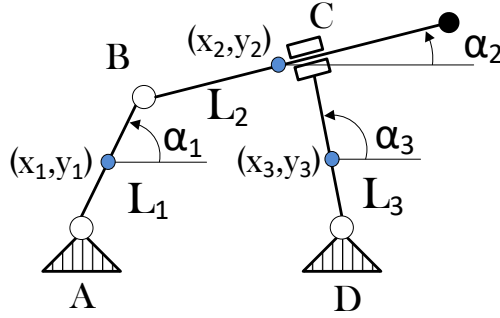


Figura 2.6: Modelado empleando coordenadas de punto de referencia.

Este tipo de coordenadas tiene como inconveniente principal el elevado número de las mismas, especialmente en sistemas 3D, lo que supone un número muy elevado de ecuaciones de restricción, que siguen siendo complejas por la presencia de funciones trigonométricas.

2.1.2.3 Coordenadas naturales

Al igual que las anteriores, las coordenadas naturales también sitúan cada elemento con independencia de los demás. Es una evolución de las coordenadas de punto de referencia, donde los puntos emigran a los pares, contribuyendo a la definición simultánea de los dos elementos que se unen en el par. Por tanto ya no son necesarias variables de tipo angular para definir la orientación de cada elemento. Observamos este proceso de emigración en la figura 2.7.

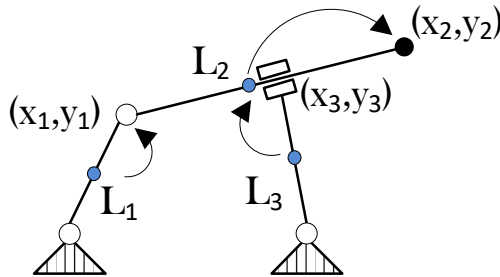


Figura 2.7: Modelado empleando coordenadas naturales.

Tenemos, por tanto, $n = 6$ coordenadas: $\mathbf{q} = [x_1 \ y_1 \ x_2 \ y_2 \ x_3 \ y_3]$, y debemos definir las $r = 5$ ecuaciones de restricción derivadas del carácter rígido de los sólidos y de ciertos movimientos impedidos por los pares [68].

Las tres primeras son condiciones de distancia constante entre puntos, para asegurar el carácter rígido de las barras móviles:

$$\begin{aligned}(x_1 - x_A)^2 + (y_1 - y_A)^2 - L_1^2 &= 0 \\ (x_2 - x_1)^2 + (y_2 - y_1)^2 - L_2^2 &= 0 \\ (x_3 - x_D)^2 + (y_3 - y_D)^2 - L_3^2 &= 0\end{aligned}$$

La siguiente impone que las barras 2 y 3 se mantengan perpendiculares:

$$(x_2 - x_1)(x_3 - x_D) + (y_2 - y_1)(y_3 - y_D) = 0$$

Y la última se encarga de que el punto especificado por el par (x_3, y_3) se halle siempre sobre la barra 2, es decir, alineado con los puntos (x_1, y_1) y (x_2, y_2) :

$$(x_3 - x_1)(y_2 - y_1) - (y_3 - y_1)(x_2 - x_1) = 0$$

El número de variables y ecuaciones de restricción crece rápidamente con la complejidad del mecanismo, especialmente en sistemas 3D, como sucede en el mecanismo mostrado en la figura 2.8.

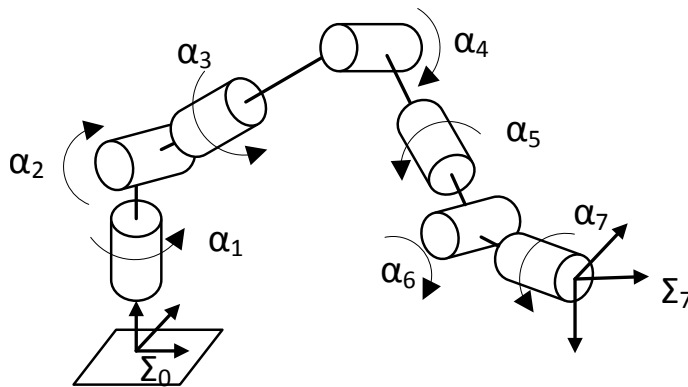


Figura 2.8: Ejemplo de sistema multicuerpo en 3 dimensiones.

2.2 Análisis estructural de sistemas multicuerpo

El análisis de sistemas multicuerpo puede ser abordado según una formulación denominada *Global*. En ella se seleccionan tantas coordenadas como sean necesarias para definir las posiciones de cada cuerpo con independencia del resto de cuerpos y, a continuación, se definen las ecuaciones de restricción asociadas a cada tipo de unión. Este método usa un gran número de variables y ecuaciones, que tiende a crecer rápidamente conforme aumenta la complejidad del sistema. Las coordenadas naturales y de punto de referencia son las más utilizadas en este tipo de formulaciones.

Una formulación alternativa a la global, denominada formulación *topológica*, permite abordar el proceso de modelado de sistemas mecánicos complejos mediante la descomposición del mismo en problemas más pequeños. Para ello, en Teoría de Mecanismos, el análisis estructural se encarga de estudiar la estructura cinemática de un mecanismo, es decir, los grupos o subsistemas en los que se podría dividir, y el orden en el que se tienen que analizar. La figura 2.9 muestra un ejemplo de esta técnica.

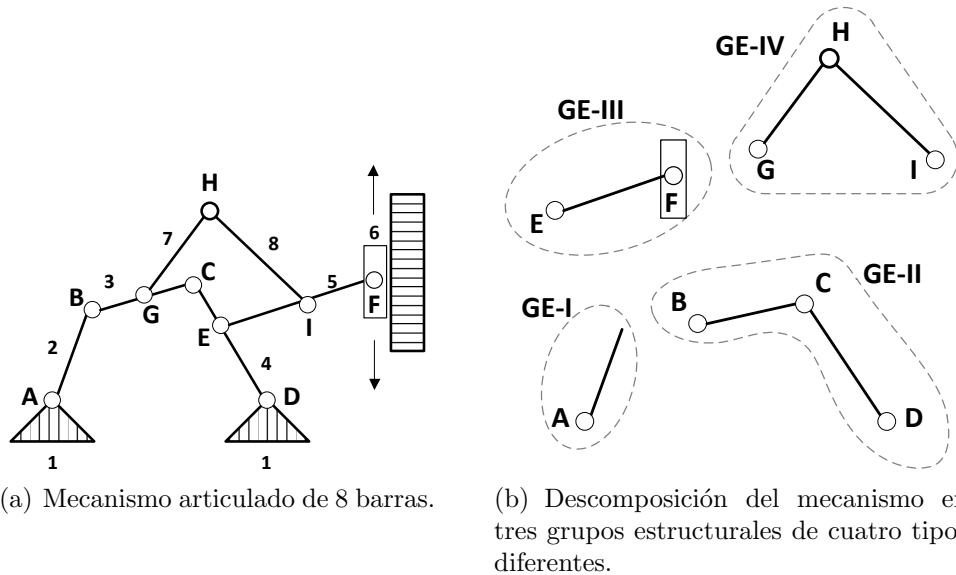


Figura 2.9: Ejemplo de análisis estructural de un mecanismo.

En la figura 2.9(a) se muestra un mecanismo articulado formado por ocho ba-

rras junto a su descomposición ordenada en los subsistemas o grupos estructurales (GE) en los que se puede dividir (figura 2.9(b)). En este ejemplo, la estructura cinemática la formarían cuatro grupos estructurales, de los cuales GE-I, GE-II y GE-III son de topología diferente, mientras que el grupo GE-IV presenta la misma topología que el GE-II.

2.2.1 Grafo estructural

Como complemento al esquema cinemático de un mecanismo, y como base para su análisis estructural, se suele dibujar el *Grafo Estructural*, que es un esquema en el que se muestran los pares existentes entre eslabones, su tipo y entre qué eslabones existe movimiento independiente de entrada. Además, el grafo estructural es una herramienta muy útil que permite sistematizar el proceso de obtención de grupos estructurales con objeto de llevar a cabo el análisis estructural.

Un grafo estructural representa la topología de un sistema siguiendo la siguiente convención:

- Cada nodo del grafo representa un eslabón. Estos nodos se etiquetan con un número que corresponde con el del eslabón.
- Si dos eslabones forman par, se trazan entre ellos tantas líneas de trazo fino como grado tenga el par.
- De las líneas descritas en el punto anterior, se trazarán con trazo grueso aquellas que el analista defina como movimientos de entrada, que se pueden identificar por una variable independiente y que corresponden, en el sistema real, al movimiento controlado por un actuador.

Tomemos como ejemplo un sistema sencillo, de un grado de libertad, como el mostrado en la figura 2.10, con los eslabones numerados del 1 al 4. Observamos que los eslabones 1 y 2 están conectados, de ahí la línea de unión que aparece reflejada en el grafo estructural de la figura 2.11.

Si consideramos que el grado de libertad que controlamos es el giro α_{21} entre estos dos eslabones (por ejemplo mediante un servomotor), el trazo de esa línea

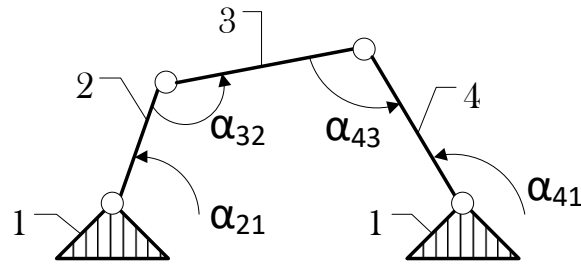


Figura 2.10: Esquema de un cuadrilátero articulado.

es grueso. El resto son uniones simples de una sola línea (solo es posible un movimiento entre ellos) y de trazo fino.

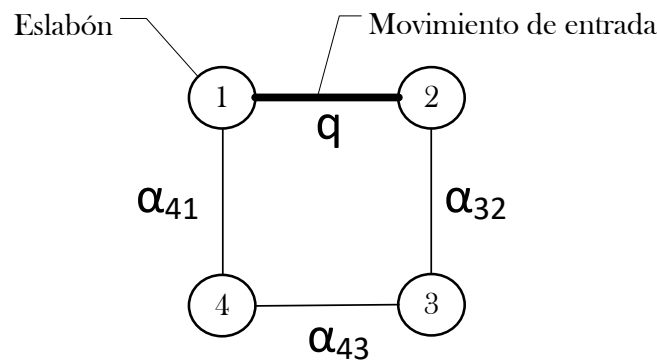


Figura 2.11: Análisis de un cuadrilátero articulado: grafo estructural.

2.2.2 Estructura cinemática

El análisis estructural de sistemas multicuerpo consiste en obtener su estructura cinemática; esto es, identificar un conjunto ordenado de grupos estructurales en que se puede dividir ese sistema.

Un *grupo estructural* se define como una cadena cinemática cuya movilidad coincide con el número de movimientos de entrada definidos sobre los eslabones de dicha cadena. Los movimientos de entrada serían las coordenadas o grados de libertad de dicha cadena cinemática. Debe cumplir, además, que ese grupo no puede dividirse en otros grupos con menor número de sólidos.

Para analizar si una cadena cinemática es un grupo estructural, se utiliza la ley de formación de grupos estructurales:

$$S_c - n_c = B(P - N_m)$$

donde:

- n_c es el número de movimientos de entrada o grados de libertad controlados.
- P es el número de pares activos en la cadena cinemática bajo estudio, suma de los pares internos que forman los eslabones de nuestra cadena y de los externos, aquellos que se forman con el bastidor y con sólidos de otros grupos previamente formados.
- S_c es la movilidad que dejan los P pares cinemáticos activos. Para calcularla podemos aplicar la siguiente fórmula, cuyos términos ya han sido definidos previamente:

$$S_c = \sum_{k=1}^{B-1} e_k \cdot P_k$$

Esta fórmula se puede extender, según tratemos con mecanismos planos o mecanismos espaciales, como sigue:

- 2D: $S_c = 1 \cdot P_1 + 2 \cdot P_2$
 - 3D: $S_c = 1 \cdot P_1 + 2 \cdot P_2 + 3 \cdot P_3 + 4 \cdot P_4 + 5 \cdot P_5$
- N_m es el número de eslabones móviles.

Con ayuda del grafo estructural y la ley de formulación de grupos estructurales se puede establecer un algoritmo que permite obtener la estructura cinemática de cualquier mecanismo plano. Este algoritmo consta de tres etapas:

1. Aislar el bastidor. El bastidor es un sólido fijo, y por lo tanto no formará parte de ninguna cadena cinemática.
2. Asignar enlaces a eslabones. Los grados de libertad permitidos entre el bastidor y los sólidos que forman par con él son movimientos que se adjudicarán a los eslabones que forman par cinemático con el bastidor.

3. Formar grupos estructurales. Para ello se buscan las cadenas cinemáticas que cumplen con la ley de formación de grupos estructurales. Se empieza con cadenas cinemáticas con el menor número posible de eslabones con objeto de encontrar primero los grupos de menor tamaño. Una vez identificado un grupo, volveremos al punto 2, para volver a asignar enlaces a eslabones que formaban pares cinemáticos con los eslabones del grupo recién identificado. Este bucle terminará cuando se haya conseguido la estructura cinemática de todo el mecanismo, es decir, hasta haber descompuesto todo el mecanismo en grupos estructurales.

Aplicando el método descrito al modelo representado por el grafo estructural de la figura 2.11 se obtiene la estructura cinemática mostrada en la figura 2.12.

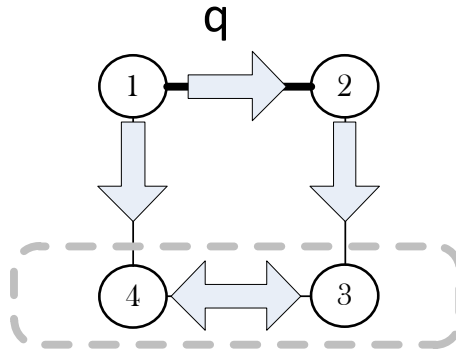


Figura 2.12: Análisis basado en el grafo: identificación de siguientes grupos sobre pares de mayor longitud.

La estructura cinemática obtenida mediante el grafo se representa en el *Diagrama Estructural*, donde cada círculo representa un grupo estructural. El identificado como 0 se añade artificialmente siempre, e identifica a la base fija. El resto se etiquetan con dos dígitos, el número de sólidos móviles (N_m) y los movimientos de entrada (n_c) al grupo. Se dibuja una flecha de unión entre dos grupos cuando hay conexión entre ambos, y el sentido de la misma indica el orden en que se han obtenido y, por tanto, define la secuencia en que se deberá resolver el análisis cinemático y dinámico.

En nuestro ejemplo identificamos 2 grupos estructurales (el 2 y el 3-4) que, junto al bastidor, suman los 3 grupos que vemos en la figura 2.13.

En el caso del modelado de mecanismos complejos, la tarea de identificar los

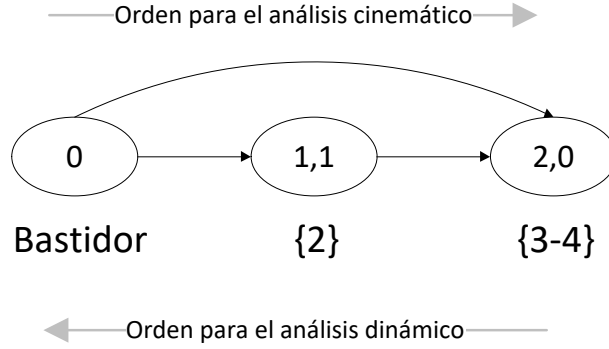


Figura 2.13: Análisis de un cuadrilátero articulado: diagrama estructural.

grupos estructurales y la posterior elección de las variables y las ecuaciones de restricción puede requerir un elevado nivel de formación por parte del analista. Por este motivo se están estudiando algoritmos para modelar los sistemas multicuerpo de forma automática, obteniendo su estructura cinemática mediante métodos computacionales tomando como base los métodos grafo-analíticos descritos anteriormente [261].

2.3 Cinemática computacional de sistemas multicuerpo

En este apartado se introduce la formulación comúnmente utilizada para el análisis cinemático computacional de sistemas multicuerpo y se describe la rutina necesaria para llevarlo a cabo. Una vez comprendida la formulación general se introduce su adaptación para la solución basada en ecuaciones de grupo en la que se basa esta tesis.

2.3.1 Análisis de posición

Supongamos un sistema mecánico con L grados de libertad en el que son necesarias \mathbf{q} coordenadas para definir la posición y la orientación de todos los sólidos. Este conjunto de coordenadas se puede dividir en dos subconjuntos: $\mathbf{q}^i \in \mathfrak{R}^L$ de coordenadas independientes, con las que se controla el mecanismo, y $\mathbf{q}^d \in$

\mathfrak{R}^m de m coordenadas dependientes, desconocidas pero relacionadas entre sí, con las coordenadas independientes y con la variable tiempo (t) por medio de un conjunto de r ecuaciones de restricción $\Phi(\mathbf{q}, t) \in \mathfrak{R}^r; r \geq m$ (ec. 2.2).

$$\Phi(\mathbf{q}, t) = \mathbf{0} \quad (2.2)$$

El número y tipo de coordenadas \mathbf{q} , así como de las ecuaciones de restricción que las relacionan entre sí, depende del método utilizado para modelar el sistema mecánico, según se ha visto en el apartado 2.1.2.

En el análisis de posición se controlan los valores de las coordenadas independientes y los grados de libertad del mecanismo y se determinan, mediante el sistema (ec. 2.2), el valor de las m coordenadas dependientes, lo que nos permitirá ensamblar el mecanismo completo. Si se quiere analizar el mecanismo en nuevas posiciones, se incrementan de forma controlada los valores de las coordenadas independientes y se vuelve a determinar el valor de las dependientes. La resolución del problema de posición para diferentes valores de los grados de libertad se conoce como simulación cinemática.

Cada análisis de posición exige, fijados unos valores de las coordenadas independientes, resolver el sistema no lineal de ecuaciones (ec. 2.2), para lo que se puede recurrir al método iterativo de Newton-Raphson:

$$\Phi_{\mathbf{q}^d(i-1)}(\mathbf{q}_{(i)}^d - \mathbf{q}_{(i-1)}^d) = -\Phi_{(i-1)} \quad (2.3)$$

En este método se obtiene, en cada iteración i , una nueva aproximación al valor exacto de las coordenadas dependientes $\mathbf{q}_{(i)}^d$, restándole al valor que tenían estas coordenadas en la iteración anterior ($i-1$), el producto de la inversa de la matriz Jacobiana $\Phi_{\mathbf{q}^d}$ por la matriz columna de las funciones de posición Φ , evaluadas estas dos últimas matrices, con los valores asignados a las coordenadas independientes y los de las coordenadas dependientes de la iteración anterior.

La matriz $\Phi_{\mathbf{q}^d}$ es la matriz Jacobiana de las funciones de posición, derivadas respecto a las variables dependientes (ec. 2.4):

$$\Phi_{\mathbf{q}^d} = \frac{\partial \Phi(\mathbf{q}, t)}{\partial \mathbf{q}^d} = \begin{bmatrix} \frac{\partial F_1}{\partial q_1^d} & \frac{\partial F_1}{\partial q_2^d} & \dots & \frac{\partial F_1}{\partial q_m^d} \\ \frac{\partial F_2}{\partial q_1^d} & \frac{\partial F_2}{\partial q_2^d} & \dots & \frac{\partial F_2}{\partial q_m^d} \\ \dots & \dots & \dots & \dots \\ \frac{\partial F_r}{\partial q_1^d} & \frac{\partial F_r}{\partial q_2^d} & \dots & \frac{\partial F_r}{\partial q_m^d} \end{bmatrix}_{r \times m} \quad (2.4)$$

Este método exige que la matriz Jacobiana sea cuadrada y su determinante no nulo, ofrece una convergencia cuadrática y las iteraciones terminan cuando el valor del residuo (ec. 2.5) es inferior a la tolerancia definida para el tipo de análisis que se realiza.

$$Res = \sum_{j=1}^m |\Phi_j| < tol \quad (2.5)$$

Una vez resuelto el problema de posición se pueden abordar los problemas de velocidades y de aceleraciones.

2.3.2 Análisis de velocidades

El análisis de velocidades consiste en determinar, en cada posición del mecanismo, las velocidades de las coordenadas dependientes $\dot{\mathbf{q}}^d$, conocidas las velocidades de los grados de libertad $\dot{\mathbf{q}}^i$ y la posición $\mathbf{q} = [\mathbf{q}^i, \mathbf{q}^d]$ de todos los sólidos, obtenida como resultado del análisis de posición.

La solución a este problema se plantea derivando respecto al tiempo las ecuaciones de restricción (ec.2.2), obteniéndose el sistema lineal de ecuaciones:

$$\Phi_{\mathbf{q}} \dot{\mathbf{q}} = \mathbf{0} \quad (2.6)$$

El sistema (ec. 2.6) se puede escribir, utilizando el método de partición de coordenadas [296], como:

$$\Phi_{\mathbf{q}^d} \dot{\mathbf{q}}^d + \Phi_{\mathbf{q}^i} \dot{\mathbf{q}}^i = -\Phi_t \quad (2.7a)$$

$$\Phi_{\mathbf{q}^d} \dot{\mathbf{q}}^d = -(\Phi_{\mathbf{q}^i} \dot{\mathbf{q}}^i + \Phi_t) \quad (2.7b)$$

Donde $\Phi_{\mathbf{q}^i}$ es la matriz Jacobiana de las funciones de posición, derivadas respecto a las variables independientes \mathbf{q}^i :

$$\Phi_{\mathbf{q}^i} = \frac{\partial \Phi(\mathbf{q}, t)}{\partial \mathbf{q}^i} = \begin{bmatrix} \frac{\partial F_1}{\partial q_1^i} & \frac{\partial F_1}{\partial q_2^i} & \dots & \frac{\partial F_n}{\partial q_k^i} \\ \frac{\partial F_2}{\partial q_1^i} & \frac{\partial F_2}{\partial q_2^i} & \dots & \frac{\partial F_n}{\partial q_k^i} \\ \dots & \dots & \dots & \dots \\ \frac{\partial F_n}{\partial q_1^i} & \frac{\partial F_n}{\partial q_2^i} & \dots & \frac{\partial F_n}{\partial q_k^i} \end{bmatrix}_{n \times k} \quad (2.8)$$

Para poder resolver el sistema (ec. 2.7), la matriz Jacobiana $\Phi_{\mathbf{q}^d}$ debe ser cuadrada. Para ello, el número de ecuaciones de restricción debe ser igual al de variables dependientes: $r = m$. En caso de no ser así, existen $r - m$ ecuaciones redundantes y la solución se puede plantear con dos estrategias: una consiste en identificar y eliminar las ecuaciones redundantes, y otra en utilizar una solución basada en mínimos cuadrados [120].

2.3.3 Análisis de aceleraciones

El análisis de aceleraciones consiste en determinar las aceleraciones de las coordenadas dependientes, $\ddot{\mathbf{q}}^d$, para cada instante de tiempo. Para ello, controlados los grados de libertad: \mathbf{q}^i , $\dot{\mathbf{q}}^i$, $\ddot{\mathbf{q}}^i$ en cada instante de tiempo, se obtienen las \mathbf{q}^d resolviendo el problema de posición, las $\dot{\mathbf{q}}^d$ resolviendo el problema de velocidad y las $\ddot{\mathbf{q}}^d$ resolviendo el problema de aceleración. Se puede formular la solución al problema de aceleraciones, derivando temporalmente la ecuación (ec. 2.6), obteniéndose:

$$\Phi_{\mathbf{q}} \ddot{\mathbf{q}} + \dot{\Phi}_{\mathbf{q}} \dot{\mathbf{q}} = \mathbf{0} \quad (2.9)$$

Utilizando el método de partición de coordenadas se puede reescribir (ec. 2.9)

como:

$$\Phi_{\mathbf{q}^d} \ddot{\mathbf{q}}^d + \Phi_{\mathbf{q}^i} \ddot{\mathbf{q}}^i + \dot{\Phi}_{\mathbf{q}} \dot{\mathbf{q}} + \dot{\Phi}_t = \mathbf{0} \quad (2.10a)$$

$$\Phi_{\mathbf{q}^d} \ddot{\mathbf{q}}^d = -(\Phi_{\mathbf{q}^i} \ddot{\mathbf{q}}^i + \dot{\Phi}_{\mathbf{q}} \dot{\mathbf{q}} + \dot{\Phi}_t) \quad (2.10b)$$

que representa un sistema lineal de ecuaciones que permite determinar las aceleraciones de las coordenadas dependientes buscadas.

2.3.4 Rutina general para el análisis cinemático computacional

La formulación cinemática computacional que se ha introducido en esta sección es la más extendida, con independencia del tipo de coordenadas seleccionadas para modelar el mecanismo: naturales, de punto de referencia o relativas. El algoritmo 1 muestra la rutina necesaria para la solución computacional de los problemas de posición, velocidad y aceleración del sistema multicuerpo.

Se ejecuta el bucle de tiempos, desde t_0 hasta t_f con incrementos de tiempo especificados (*timeStep*). Para cada instante de tiempo se actualizan los valores de las coordenadas independientes \mathbf{q}^i y de sus derivadas temporales, que son conocidas: $\dot{\mathbf{q}}^i$, $\ddot{\mathbf{q}}^i$.

Seguidamente, se resuelve el problema de posición; se llama a una función externa *fPos()* que devuelve los valores de las funciones de posición (ec. 2.2). Se evalúa el residuo (ec. 2.5) y se ejecuta el bucle iterativo de Newton-Raphson (ec. 2.3) hasta que este sea inferior a la tolerancia asignada. Una vez resueltas las posiciones, se actualiza la matriz Jacobiana completa $\Phi_{\mathbf{q}}$, de la que se extraen las correspondientes Jacobianas dependientes e independientes para formular la solución a las velocidades (ec. 2.7).

Finalmente, se evalúa el término $\dot{\Phi}_{\mathbf{q}} \dot{\mathbf{q}}$ necesario en la ecuación 2.10 para resolver el problema de aceleraciones. Hay que observar que, en esta rutina general, los sistemas de ecuaciones correspondientes a los problemas de posición, velocidad y aceleración están formulados considerando todas las coordenadas \mathbf{q} dependientes

e independientes del modelo. De esta forma, a mayor complejidad del sistema mecánico a analizar, mayor número de ecuaciones r forman esos sistemas, aumentando el esfuerzo computacional en la forma $\mathbf{O}(r^3)$.

Algoritmo 1: Rutina general de análisis cinemático computacional.

```

1  for  $t = t_0 : \text{timeStep} : t_f$  do
2       $\ddot{\mathbf{q}}^i(t); \dot{\mathbf{q}}^i(\ddot{\mathbf{q}}^i, \dot{\mathbf{q}}_0^i); \mathbf{q}^i(\ddot{\mathbf{q}}^i, \dot{\mathbf{q}}_0^i, \mathbf{q}_0^i)$     /* Evalúa coord. indep. */
3      %% I. Problema de posición %
4      CALL fPos()                                           /* Evalúa  $\Phi(\mathbf{q})$  */
5      CALL fResiduo()                                       /* Evalúa el residuo */
6      while  $\text{Residuo} > \text{tolerance}$  do
7          CALL fJacob()                                     /* Evalúa  $\Phi_{\mathbf{q}}$  */
8          Extrae  $\Phi_{\mathbf{q}}^d$                                 /* Jacobiana dependientes */
9          solve:  $\Phi_{\mathbf{q}_{k-1}}^d(\mathbf{q}_k^d - \mathbf{q}_{k-1}^d) = -\Phi_{k-1}$ 
10         CALL fPos()
11         CALL fResiduo()                                   /* Evalúa el residuo */
12     end
13     %% II. Problema de velocidad %
14     CALL fJacob()
15     Extrae  $\Phi_{\mathbf{q}}^d$                                      /* Jacobiana dependientes */
16     Extrae  $\Phi_{\mathbf{q}}^i$                                      /* Jacobiana independientes */
17     solve:  $\Phi_{\mathbf{q}^d} \dot{\mathbf{q}}^d = -(\Phi_{\mathbf{q}^i} \dot{\mathbf{q}}^i + \Phi_t)$ 
18     %% III. Problema de aceleraciones %
19     CALL Fiqqpp()                                         /* Evalúa  $\dot{\Phi}_{\mathbf{q}} \dot{\mathbf{q}}$  */
20     solve:  $\Phi_{\mathbf{q}^d} \ddot{\mathbf{q}}^d = -(\Phi_{\mathbf{q}^i} \ddot{\mathbf{q}}^i + \dot{\Phi}_{\mathbf{q}} \dot{\mathbf{q}} + \ddot{\Phi}_t)$ 
21 end

```

El siguiente apartado introduce la formulación basada en ecuaciones de grupo, con la que se pretende reducir el esfuerzo computacional al dividir sistemas complejos en subsistemas de tamaño reducido. En esta formulación se basa el desarrollo del simulador implementado en esta tesis.

2.3.5 Formulación cinemática basada en ecuaciones de grupo

El análisis cinemático de un sistema multicuerpo (MBS) se puede llevar a cabo resolviendo, en cada instante de tiempo, la cinemática de cada uno de los grupos estructurales que lo forman, en el orden definido por su estructura cinemática. La solución de cada grupo estructural puede seguir el esquema mostrado en la formulación general descrita en el apartado anterior, con dos particularidades. Por un lado, en la solución general, las coordenadas independientes \mathbf{q}^i corresponden a los grados de libertad de todo el mecanismo, mientras que en la solución por grupos corresponden a coordenadas de puntos y vectores de sólidos de grupos ya analizados. Por otro lado, las matrices implicadas en la rutina de análisis general se deben particularizar para cada sistema multicuerpo a analizar, mientras que en la solución por grupos, como cada grupo estructural tiene una topología concreta, se puede desarrollar una librería de rutinas de análisis de un elevado número de grupos estructurales, de forma que la solución de todo el sistema se obtenga mediante la llamada a las rutinas de los grupos que forman la estructura cinemática del sistema a analizar. En este último sentido, los sistemas de ecuaciones para los problemas de posición, velocidad y aceleración de grupos estructurales sencillos admiten soluciones directas, permitiendo prescindir de métodos iterativos.

A modo de ejemplo, y sin pérdida de generalidad, se describe qué forma tienen las rutinas de análisis cinemático basado en ecuaciones de grupo aplicado al mecanismo cuadrilátero articulado que se ha presentado en la figura 2.10 y cuya estructura cinemática se ha mostrado en la figura 2.12. En la figura 2.14(a) se muestra el mismo mecanismo con el grado de libertad controlado en la solución general, y en la figura 2.14(b) se muestra su división en dos grupos estructurales: SG-I y SG-II.

Para aplicar la formulación basada en ecuaciones de grupo, se debe seleccionar un conjunto de coordenadas \mathbf{q} que permita modelar al grupo estructural y definir

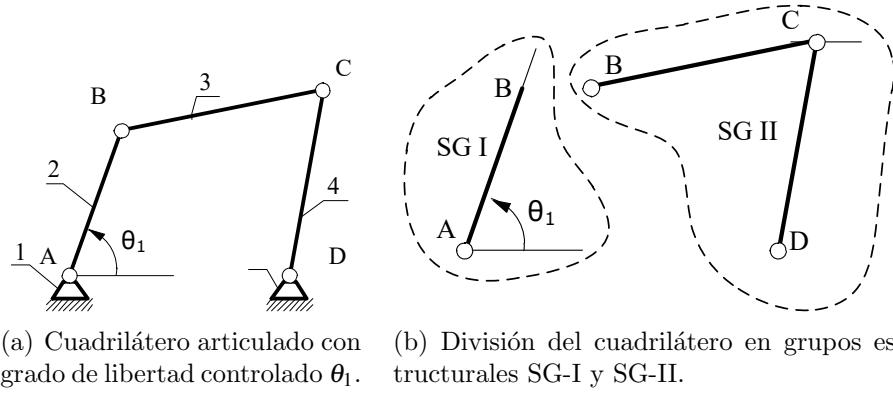


Figura 2.14: Mecanismo cuadrilátero articulado usado para mostrar las rutinas de análisis cinemático basado en análisis de grupo.

un conjunto de ecuaciones de restricción Φ que las relacionen. Se puede elegir cualquier tipo de coordenadas, y para este apartado utilizaremos coordenadas naturales.

Para el grupo SG-I de la figura 2.14(b), se tienen: $\mathbf{q}^d = [x_B, y_B]$ y $\mathbf{q}^i = [\theta_1]$ respectivamente. Las ecuaciones de grupo son muy básicas en este caso:

$$\Phi = \begin{bmatrix} x_B - \overline{AB} \cos \theta_1 \\ y_B - \overline{AB} \sin \theta_1 \end{bmatrix} = [\mathbf{0}]_{2 \times 1} \quad (2.11)$$

Para este grupo, la solución a las posiciones se obtiene por resolución directa de (ec. 2.11) y la solución a velocidades y aceleraciones, por derivación temporal directa de esas expresiones, como se muestra en (ec. 2.12) para la coordenada dependiente x_B :

$$\begin{aligned} \dot{x}_B &= -\overline{AB} \sin \theta_1 \dot{\theta}_1 \\ \ddot{x}_B &= -\overline{AB} \sin \theta_1 \ddot{\theta}_1 - \overline{AB} \cos \theta_1 \dot{\theta}_1^2 \end{aligned} \quad (2.12)$$

De este modo se podría preparar una rutina de análisis para este grupo que devuelva directamente la posición, velocidad y aceleración de sus coordenadas dependientes.

Para el grupo SG-II de la figura 2.14(b), se tienen: $\mathbf{q}^d = [x_C, y_C]$ y $\mathbf{q}^i = [x_B, y_B]$, dado que x_D, y_D son constantes. Las ecuaciones de grupo corresponden a la con-

dición de sólido rígido para los eslabones 3 y 4:

$$\Phi = \begin{bmatrix} \mathbf{r}_{BC}^T \mathbf{r}_{BC} - \overline{BC}^2 \\ \mathbf{r}_{CD}^T \mathbf{r}_{CD} - \overline{CD}^2 \end{bmatrix} = [\mathbf{0}]_{2 \times 1} \quad (2.13)$$

\mathbf{r}_{BC}^T es la traspuesta del vector \mathbf{r}_{BC} y se utiliza como notación compacta para la restricción constante entre dos puntos del modelo: $\mathbf{r}_{BC}^T \mathbf{r}_{BC} - \overline{BC}^2$ es lo mismo que $((XC - XB)^2 + (YC - YB)^2 - L^2 = 0$.

Las matrices necesarias para el análisis del grupo SG-II: Φ_{q^d} , Φ_{q^i} , $\dot{\Phi}_{q^i} \dot{\mathbf{q}}$, son fáciles de obtener y se pueden almacenar en una rutina específica de solución de este grupo. Los valores de las coordenadas dependientes se pueden calcular aplicando el método iterativo de Newton-Raphson (ec. 2.3) y los problemas de velocidades y aceleraciones se pueden formular de forma análoga al caso general (ecs. 2.7 y 2.10), teniendo en cuenta que, en muchos casos, la inversa de la matriz Jacobiana Φ_{q^d} se puede obtener de forma analítica, lo que permite su reutilización en la resolución de todos los sistemas de ecuaciones.

Una vez obtenida la información de los grupos estructurales en que está dividido el sistema multicuerpo (la estructura cinemática), el programa principal (algoritmo 2) se encarga de la rutina de análisis cinemático basado en ecuaciones de grupo mediante la ejecución de dos bucles anidados. El primero es el bucle de tiempos, en el que se incrementan los valores de los grados de libertad de todo el sistema. Para cada instante de tiempo, el segundo bucle llama a cada uno de los grupos estructurales que forman el sistema multicuerpo, en el orden definido en la estructura cinemática, y dentro de este bucle se identifica a qué rutina de solución debe llamar, dependiendo del tipo de grupo estructural que corresponda.

En el ejemplo de la figura 2.14(b) el fichero principal mostrado en el algoritmo 2 deberá llamar primero a la rutina de análisis del SG-I (*Solve_1RSG(*ARGS)*) y después a la rutina de análisis del SG-II (*Solve_3RSG(*ARGS)*).

Algoritmo 2: Cinemática computacional basada en ecuaciones de grupo.

```
1  for  $t = t_0 : timeStep : t_f$  do
2       $\ddot{\mathbf{q}}^i(t); \dot{\mathbf{q}}^i(\ddot{\mathbf{q}}^i, \dot{\mathbf{q}}_0^i); \mathbf{q}^i(\ddot{\mathbf{q}}^i, \dot{\mathbf{q}}_0^i, \mathbf{q}_0^i)$     /* Evalúa coord. indep. */
3      for  $ng = 2 : length(MGrupos)$  do
4          /* Resuelve todos los grupos estructurales */
5          switch  $MGroups(ng).kind$  do
6              case  $MGrupos(ng).kind == 1RSG$  do
7                  CALL Solve_1RSG(*ARGS)
8              case  $MGrupos(ng).kind == 3RSG$  do
9                  CALL Solve_3RSG(*ARGS)
10             case  $MGrupos(ng).kind == \dots$  do
11                 CALL ...
12         end
13     end
14 end
```

1RSG y *3RSG* son identificadores que se han dado a dos grupos estructurales para llamar a las correspondientes rutinas de su análisis cinemático. La notación es *1R* o *3R* dependiendo del número (1 o 3) de pares de rotación (R) del grupo y '*SG*' se refiere a Structural Group. De este modo, *1RSG* se refiere a un grupo estructural con un par de rotación; por ejemplo una manivela con giro controlado (SG I en la figura 2.14(b)) y *3RSG* se refiere a un grupo estructural con tres pares de rotación (SG II en la figura 2.14(b)).

2.4 Conclusiones

En este capítulo se han comentado aspectos relacionados con el análisis cinemático de los sistemas multicuerpo, con especial interés en la formulación topológica que permite, mediante un adecuado análisis estructural de un determinado mecanismo, hallar la división de un sistema en los subsistemas que lo componen y determinar el orden en que se deben analizar.

Este tipo de modelado, basada en ecuaciones de grupo, permite una imple-

mentación computacional modular en la que se pueden introducir técnicas de paralelismo con el objetivo de obtener mejores tiempos de ejecución en plataformas computacionales multicore o multicore+multiGPU, lo cual es objeto central de estudio de esta tesis.

También se han descrito dos tipos de rutinas para el análisis cinemático computacional de un mecanismo cuadrilátero de ejemplo. En ellas se encuentran resoluciones de sistemas de ecuaciones y el tratamiento de las matrices asociadas, cálculos habitualmente implementados en numerosas librerías de álgebra lineal. Dada la variedad de las mismas, la elección de la más eficiente para un problema concreto puede no ser trivial. En esta tesis estudiaremos un método de selección de librerías de cálculos, que tendrá en cuenta tanto el hardware computacional como el tamaño y tipo de las matrices.

Capítulo 3

Herramientas computacionales

Este capítulo recoge una descripción de las herramientas de hardware y software que se han empleado durante la elaboración del presente trabajo. Las herramientas hardware hacen referencia a las plataformas de computación empleadas, cuyas arquitecturas serán determinantes en la selección de los paradigmas de paralelismo a proponer, y que se usarán para la experimentación y comprobación de la eficiencia de las técnicas propuestas. Las herramientas software incluyen los compiladores usados para la creación del simulador desarrollado en esta tesis y las librerías de álgebra lineal en las que se han delegado los cálculos matriciales.

3.1 Herramientas hardware

En este trabajo se han estudiado técnicas de optimización aplicadas a algoritmos de resolución de ciertos problemas en los que se emplean técnicas de álgebra matricial. Un aprovechamiento óptimo del hardware requiere de un conocimiento de las unidades de computación elementales que constituyen las plataformas de cómputo habituales en el entorno científico.

Un ejemplo de componente informático es el procesador multinúcleo, el cual implementa multiprocesamiento en un solo paquete físico por medio de la inclusión de dos o más CPU que leen y ejecutan un determinado juego de instrucciones.

Al estar los procesadores instalados en un mismo componente, la conexión entre ellos es muy rápida. Existen dos formas de acoplar núcleos en un dispositivo multinúcleo, estrechamente o débilmente:

- En los multiprocesadores estrechamente acoplados (*tightly coupled multiprocessors*), también llamados multiprocesadores de memoria compartida, todos los procesadores comparten un único espacio de direcciones de memoria. Estos sistemas tienen unas tasas de transferencia de datos muy elevadas, pero deben implementar mecanismos de resolución frente a conflictos en el acceso a zonas de memoria comunes. Normalmente, cada núcleo tiene su propia memoria de instrucciones y datos (cachés L1) y todos los núcleos comparten una caché en chip de segundo nivel (L2). La figura 3.1 representa un diagrama de bloques de un sistema informático típico de CPU multinúcleo (un núcleo cuádruple en este caso) donde todos los núcleos comparten una caché L2. La CPU también está conectada a la memoria compartida externa.

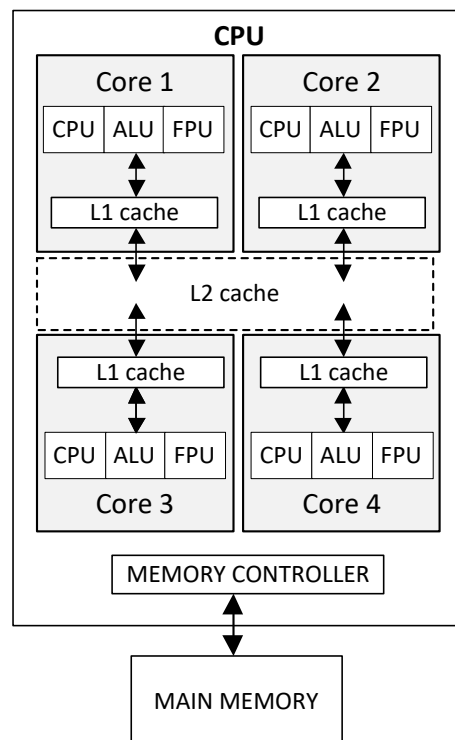


Figura 3.1: Diagrama de bloques de un sistema informático típico de CPU multinúcleo donde todos los núcleos comparten una caché L2.

Como particularidad, Intel© incorpora en algunos de sus procesadores la tecnología Hyper Threading [153] para simular dos microprocesadores lógicos dentro de un core físico, permitiendo así procesar dos threads o subprocesos en paralelo.

- En los multiprocesadores débil o vagamente acoplados (*loosely coupled multiprocessors*), también llamados multiprocesadores de memoria distribuida, cada procesador solo puede acceder a su propia memoria. Por este motivo se requiere una comunicación entre los nodos de proceso para coordinar las operaciones y mover los datos. Los módulos se conectan a través de la red MTS (sistema de transferencia de mensajes), por lo que la tasa de transferencia de datos es menor que en los sistemas multiprocesador estrechamente acoplados. La figura 3.2 muestra una arquitectura de este tipo.

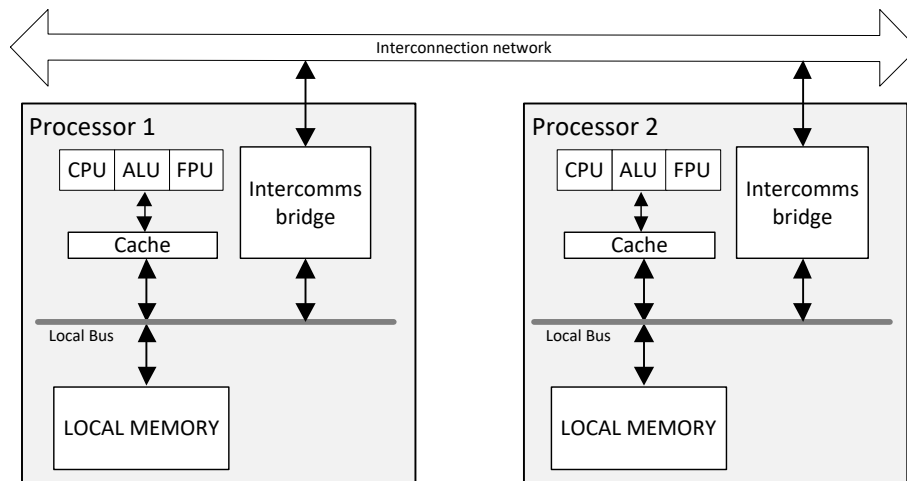


Figura 3.2: Diagrama de bloques de un multiprocesador vagamente acoplado.

Los sistemas con una gran cantidad de núcleos de procesamiento (decenas o cientos) a veces se denominan sistemas *many-core*. A partir de estos procesadores multinúcleo, es posible implementar arquitecturas de mayor nivel, por ejemplo, la conocida como NUMA (*Non-uniform Memory Access Architecture*) [13]. En tales sistemas, todas las CPU ven las memorias distribuidas como una memoria combinada; sin embargo, los tiempos de acceso y el rendimiento varían según la ubicación de la memoria y la CPU. Por ejemplo, los accesos a la memoria de la CPU ubicados en el lado opuesto de donde se encuentra la memoria objetivo incurren en una latencia mayor que los accesos a una memoria cercana.

La figura 3.3 muestra el esquema de un multicore formado por dos quadcores con un total de 8 cores físicos. En este tipo de sistemas, donde todos los cores tienen acceso compartido a la memoria, usaremos el estándar OpenMP [53, 233, 234] para explotar el paralelismo que ofrecen.

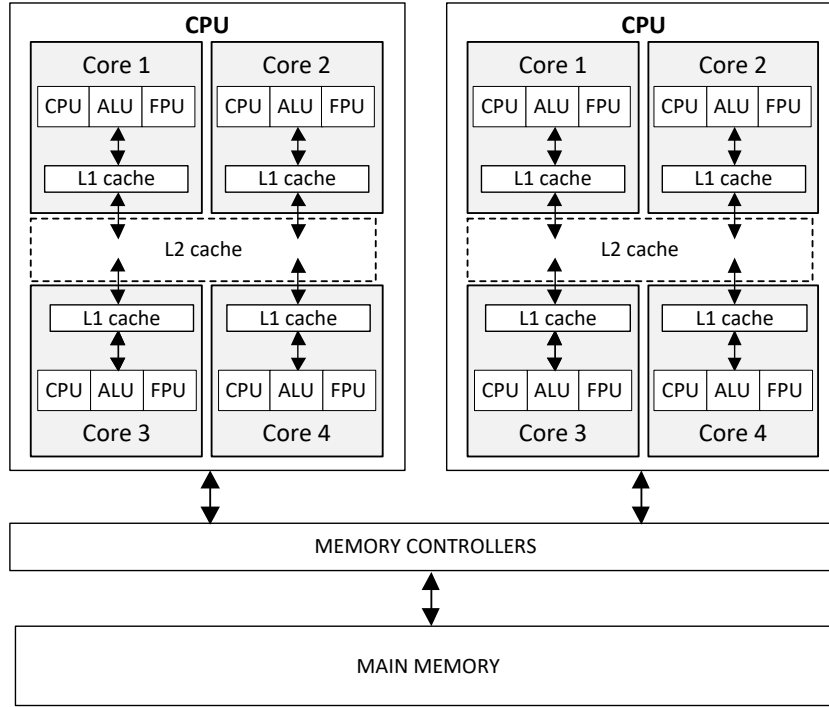


Figura 3.3: Diagrama de bloques de una arquitectura típica de CPU múltiple / núcleos múltiples (con dos CPU de cuatro núcleos cada una), donde las memorias distribuidas se ven desde las CPU como una memoria unificada, accesible con paradigma de compartición de memoria (arquitectura NUMA).

Otro elemento de procesamiento común son las GPU (*Graphic Processing Unit*). Estas unidades se diseñaron inicialmente para procesar tareas computacionalmente intensivas en aplicaciones de gráficos tridimensionales (3D), tales como texturizar conjuntos grandes de polígonos, calcular el sombreado y la iluminación, y renderizar las imágenes bidimensionales (2D) resultantes en pantalla. El hardware de una GPU contiene cientos o incluso miles de cores, y se organiza mediante un grupo de *Streaming Multiprocessors* (SM), que contienen a su vez una serie de *Streaming Processors* (SP) accediendo a una caché L1 de memoria compartida. La memoria global y la memoria de constantes son accesibles por todos los SM. Además, se proporciona una caché L2 de mayor capacidad para asegurar un acceso más rápido a la memoria global (figura 3.4).

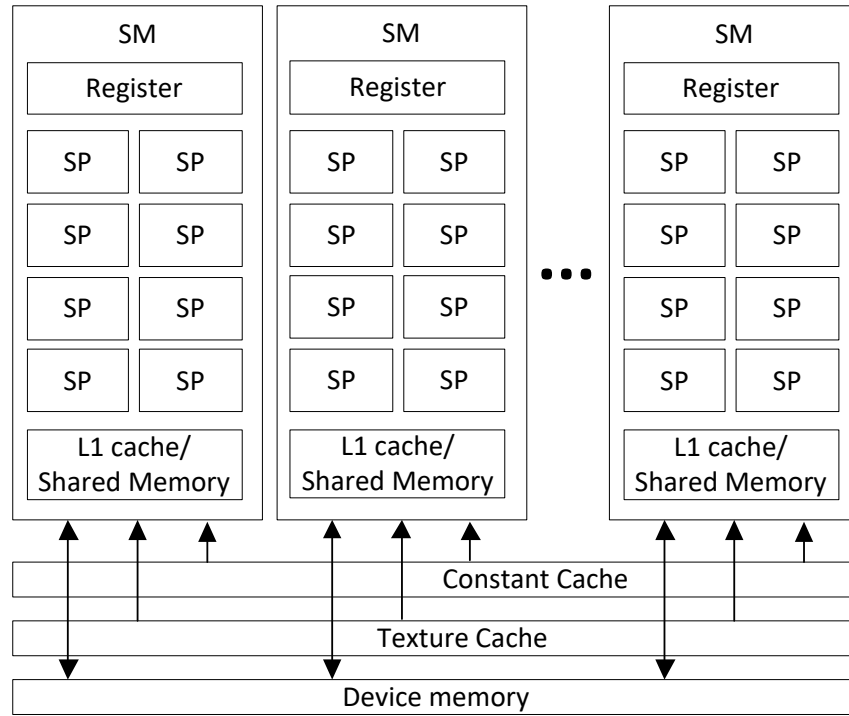


Figura 3.4: Vista esquemática de la arquitectura NVIDIA GPU, donde SM se refiere a *Streaming Multiprocessors* y SP a *Streaming Processors*.

Cuando las computadoras modernas comenzaron a incorporar tarjetas GPU, surgió el interés en usarlas como dispositivos de computación paralela de propósito general, originando la llamada programación de GPU de propósito general (GPGPU). Sin embargo, la programación GPGPU en ese momento era muy compleja porque el modelo de datos se basaba en una representación de vértices de una malla poligonal, y los datos de propósito general eran difíciles de representar de esa manera. Para mejorar este aspecto y hacer que la GPU sea un dispositivo programable real como una CPU se han propuesto primitivas de programación, como por ejemplo OpenCL [171] y NVIDIA© CUDA™ [206]. Esta última se ha convertido en la opción más extendida, ya que proporciona un interfaz de programación C/C++ y un modelo consistente en un código que se ejecuta en la CPU y que puede generar múltiples núcleos computacionales, o *kernels* CUDA, que realizarán en la GPU las tareas de mayor carga computacional (figura 3.5). Cada kernel se ejecuta mediante un conjunto de threads que están organizados en bloques (*Blocks*). A su vez, los bloques de threads están organizados en forma de malla (*Grid*).

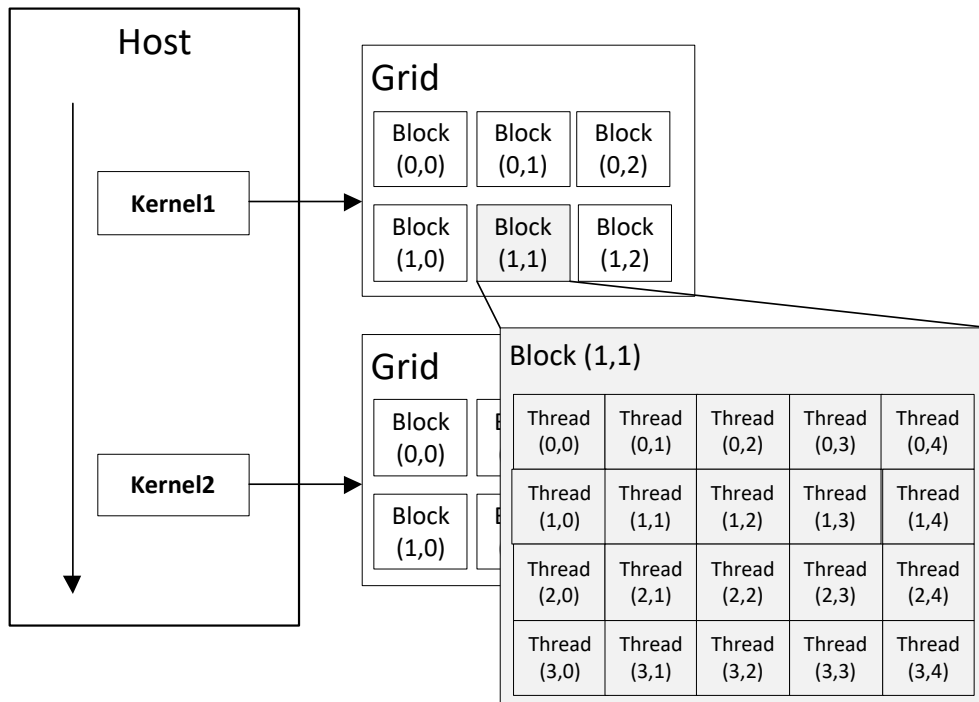


Figura 3.5: Vista esquemática del modelo de programación en CUDA.

De acuerdo a esta jerarquía, un problema paralelo se divide primero en subproblemas gruesos que se asignan a múltiples bloques, y luego cada subproblema se divide a su vez en piezas más finas que pueden ser procesadas por todos los threads dentro de cada bloque. Por lo tanto, los threads son la unidad básica de cálculo en CUDA. Todos los threads ejecutan el mismo código, pero tienen asignados identificadores diferentes que se usan para direccionar la memoria y tomar las decisiones de control, siguiendo el paradigma SIMD (*Simple Instruction Multiple Data*) [131].

La memoria presenta también una estructura jerárquica. En primer lugar, está la memoria de ámbito privado a cada thread, solamente accesible desde él mismo. Cada bloque de threads posee un espacio de memoria compartida por los threads del bloque (*Shared Memory*) y con un ámbito de vida igual que el del propio bloque. Todos los threads pueden acceder a una memoria global del dispositivo (*Device Memory*). Además, existen otros dos espacios de memoria de ámbito global, pero que son solo de lectura: la memoria constante y la de texturas. Todas las memorias de acceso global persisten más allá de la ejecución del propio kernel.

En esta tesis propondremos soluciones paralelas de uso combinado de computación en CPU multicore con el apoyo de una o más GPU. La estrategia consistirá en lanzar threads OpenMP en el multicore. Cada thread podrá realizar cálculos en la CPU y también gestionar llamadas a librerías de cómputo que hagan uso de una GPU. Las computaciones en CPU se harán a través de llamadas a librerías que podrán utilizar paralelismo implícito y, por tanto, crear nuevos threads. Este esquema genera un escenario de posibles combinaciones de cores+GPU que habrá que ajustar en función del rendimiento de las librerías, el tamaño del conjunto de datos y el número de cálculos que sean independientes y se puedan realizar en paralelo. Por ejemplo, en un problema donde se pueden resolver simultáneamente tres grupos de cálculos, pudiendo éstos tener diferente carga computacional, y usando una plataforma hardware de 12 cores y 2 GPU, una asignación equitativa podría reservar cuatro cores a cada grupo y una GPU a dos de ellos. Pero otra asignación podría reservar diez cores a uno de los grupos, y un core junto a una GPU para los otros dos. El reparto óptimo será aquel con el que se consiga obtener los menores tiempos de ejecución.

Los experimentos se ha realizado usando el cluster *Heterosolar*, perteneciente al Grupo de Computación Científica y Programación Paralela de la Universidad de Murcia [239], cuyo diagrama se muestra en la figura 3.6. El cluster está formado por un nodo de acceso, denominado *luna*, y cinco nodos de cómputo que se encuentran conectados entre sí por medio de una red Gigabit Ethernet y ejecutan la versión 3.13.0-33-generic (3.13.0-33.58) del kernel de Ubuntu Linux. *luna* actúa como *front-end* para permitir el acceso al cluster. Está virtualizado y no se puede utilizar como nodo de cómputo. Se encarga de exportar el sistema de ficheros al resto de nodos y habilita el acceso a ellos a través del protocolo ssh. Incorpora el sistema de colas TORQUE (*Terascale Open-source Resource and QUEue Manager*) [3] para la planificación de los trabajos que se envían a los nodos de cómputo solicitados. Las características de dichos nodos se pueden consultar en la tabla 3.1.

La diversidad de configuraciones en los nodos que forman el cluster *Heterosolar* nos va a permitir realizar experimentos sobre sistemas computacionales de diversa complejidad, con CPUs que incorporan diferente número de cores y con GPUs de varias capacidades computacionales.

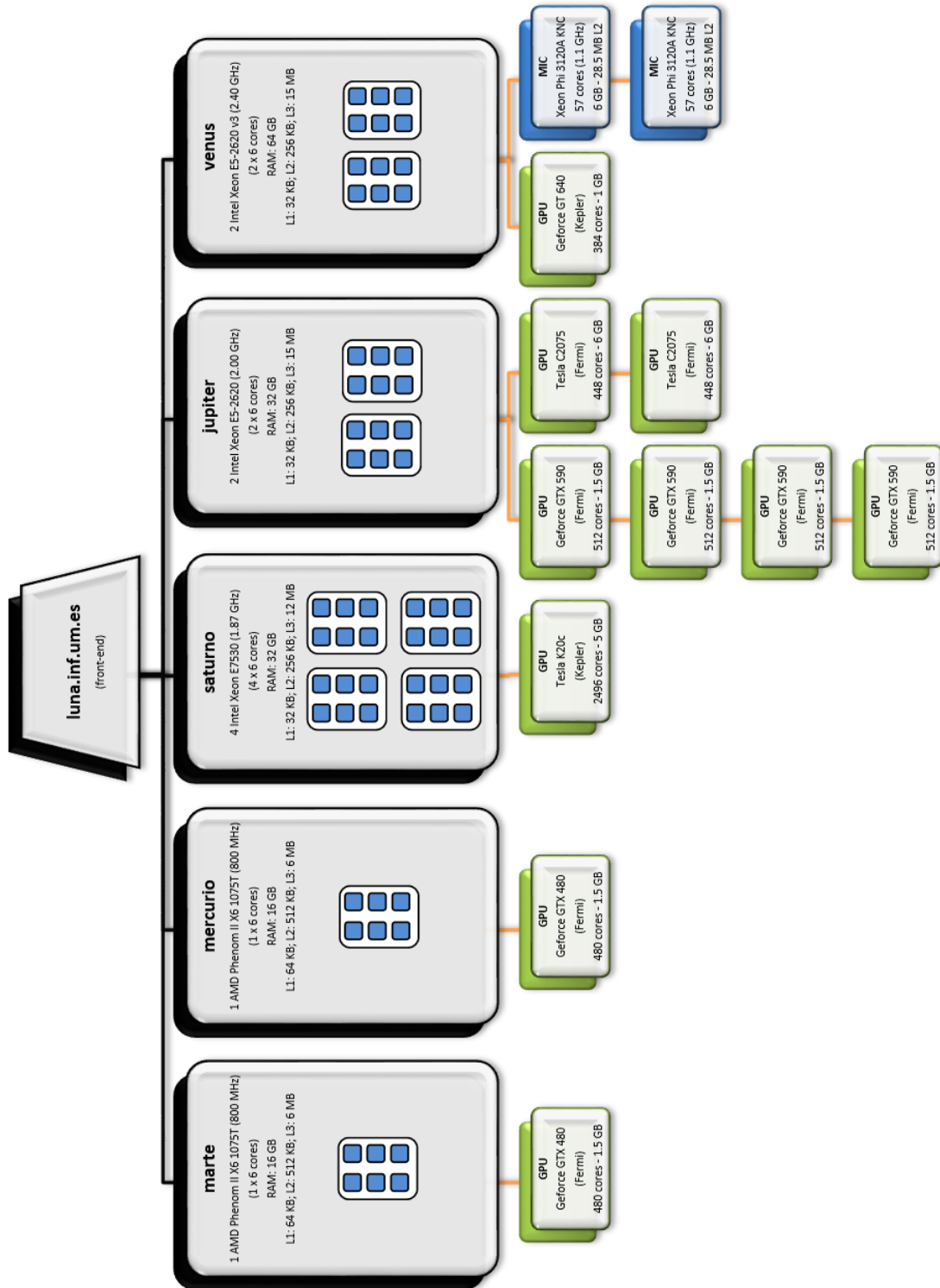


Figura 3.6: Estructura del cluster *Heterosolar*.

Nodo	Marte	Mercurio	Saturno	Jupiter	Venus
CPU					
Procesadores	Phenom II X6 1075T	Phenom II X6 1075T	Xeon E7530	Xeon E5-2620	Xeon E5-2620
Fabricante/Arquitect.	AMD/x86-64	AMD/x86-64	Intel/x86-64	Intel/x86-64	Intel/x86-64
Reloj	800 MHz	800 MHz	1.87 GHz	2.00 GHz	2.40 GHz
Cores por CPU	6	6	6	6	6
CPU por nodo	1	1	4	2	2
Cores físicos/lógicos	6 / 6	6 / 6	24 / 48	12 / 24	12 / 12
Memoria RAM	16 GB	16 GB	32 GB	32 GB	64 GB
Cache L1	64 KB	64 KB	32 KB	32 KB	32 KB
Cache L2	512 KB	512 KB	256 KB	256 KB	256 KB
Cache L3	6 MB	6 MB	12 MB	15 MB	15 MB
GPU					
Procesador NVIDIA	GeForce GTX 480	GeForce GTX 480	Tesla K20c	Tesla C2075	GeForce GT 640
Arquitectura	Fermi	Fermi	Kepler	Fermi	Kepler
GPU por nodo	1	1	1	2	1
Reloj	1401 MHz	1401 MHz	706 MHz	574 MHz	902 MHz
Memoria	1536 MB	1536 MB	4800 MB	5375 MB	1024 MB
Streaming MultiProcs.	15	15	13	14	2
Streaming Processors	32	32	192	32	192
CUDA cores	480	480	2496	448	384
Xeon Phi					
Modelo/Memoria RAM					3120A/6 GB
Unidades por Nodo					2
Cores físicos/lógicos					57 / 114

Tabla 3.1: Cluster *Heterosolar*: nodos de cómputo y especificaciones del hardware

3.2 Herramientas software

La elaboración de esta tesis ha requerido del uso de diversas herramientas de software, con diferentes propósitos. Estas herramientas son, por un lado, los compiladores necesarios para el desarrollo del simulador y, por otro, las librerías de computación que se han elegido para la realización de los cálculos de álgebra lineal.

3.2.1 Compiladores

Una parte importante del trabajo realizado en esta tesis ha sido la construcción de un simulador el cual, partiendo de un problema representado en forma de grafo, identifica los parámetros algorítmicos óptimos que minimizan los tiempos de ejecución aplicados a la resolución del problema, proponiendo ejecuciones paralelas cuando se identifican cálculos independientes, y seleccionando la librería de álgebra lineal que ofrece mejor rendimiento de acuerdo a las dimensiones del problema.

El simulador se ha desarrollado usando los lenguajes de programación que ofrecen mejores prestaciones para la funcionalidad que se quiere cubrir. Su estructura modular nos ha permitido trabajar con tres lenguajes de programación y sus correspondientes compiladores:

- JAVA [235]: Hemos seleccionado este lenguaje para el desarrollo del interfaz gráfico del simulador. El motivo ha sido la disponibilidad de librerías que implementan funcionalidades esenciales, tales como la edición de grafos (necesario para representar tanto el algoritmo a resolver como el árbol de soluciones), y las herramientas de representación gráfica y tabular para el análisis de los resultados obtenidos. Otro factor que nos ha hecho decidirnos por este lenguaje ha sido la portabilidad a diversos sistemas operativos, habiéndose probado con éxito tanto en entornos Windows [208] como Linux [185]. Para el desarrollo se ha usado el JDK (*Java Development Kit*) en su versión 8.

- C [159]: Se ha seleccionado este lenguaje para el desarrollo de la librería que gestiona la simulación, y que incluye tareas tales como la importación de los archivos que especifican el algoritmo de resolución y los juegos de datos. Además, gestiona la ejecución de los cálculos, introduciendo técnicas de paralelismo y realizando las llamadas a librerías de cálculo cuando se requiere. El motivo de seleccionar C como lenguaje de programación para esta parte del simulador radica en su rapidez de ejecución. Se ha empleado el compilador de C++ de Intel [148], generando un código compatible ANSI C, lo que permite su compilación en Windows (entorno usado durante la fase de desarrollo y para la ejecución en equipos de sobremesa y portátiles) y Linux (para su explotación en la fase de experimentación en el cluster *Heterosolar*).
- FORTRAN [158]: Todos los cálculos matriciales incorporados en el simulador se ejecutan por medio de librerías desarrolladas en FORTRAN. Es un lenguaje muy popular en el área de la computación de alto rendimiento y está especialmente adaptado al cálculo numérico y a la computación científica, siendo muy habitual su uso en áreas de la ingeniería, como es el caso del análisis computacional de mecanismos tratado en esta tesis. Dado que las llamadas a las funciones que ejecutan los cálculos se realizan desde la librería desarrollada en C, se ha creado un interfaz para permitir la comunicación, el uso compartido de memoria y el paso de argumentos entre ambos lenguajes. Se ha empleado el compilador de FORTRAN de Intel [149]. Al igual que ocurre con la librería desarrollada en C, el mismo código se puede compilar en entornos Windows y Linux.

3.2.2 Librerías de álgebra lineal

Como se comentó en la sección 1.2.5, las librerías de álgebra lineal se encuentran en continua evolución, siendo posible encontrar versiones adaptadas a diferentes tipos y tamaños de matrices. También se ofrecen optimizaciones para su ejecución en las modernas plataformas heterogéneas, que incluyen de manera habitual procesadores multicore en combinación con un número variable de coprocesadores, como por ejemplo GPUs.

Esta tesis profundiza en el análisis de un conjunto seleccionado de dichas librerías con el objetivo de conocer los rendimientos que ofrecen para diversos tamaños, factores de dispersión y tipología de matrices, y usar esta información para predecir cuál de ellas puede ofrecer mejores tiempos de ejecución en la resolución de un nuevo problema.

En los modelos de sistemas multicuerpo obtenidos empleando una formulación topológica es habitual obtener matrices simétricas con más del 70% de valores nulos. Además, estas matrices tienen la particularidad de que los valores no nulos se concentran alrededor de la diagonal. Este tipo de matrices tiene un especial interés en el marco de esta tesis por ser utilizadas en las rutinas de simulación de mecanismos. Sin embargo, el interés en extender las técnicas de optimización analizadas en este trabajo a otro tipo de problemas científicos hace que contemplemos también el uso de matrices no simétricas y densas en el simulador. Por ello, las librerías que hemos seleccionado, y que están incorporadas en el simulador, cubren un amplio rango de formatos matriciales. A continuación se ofrece un listado de las mismas:

- MKL [150] (*Math Kernel Library*) es una librería desarrollada por Intel© y está optimizada para su familia de microprocesadores. Incorpora rutinas BLAS [37], tanto para tipos de datos complejos como reales de simple y doble precisión. Incorpora también funciones de LAPACK [38], como son las factorizaciones LU, Cholesky y QR, usadas para la resolución de sistemas de ecuaciones. MKL permite paralelismo implícito, siendo posible indicar en tiempo de ejecución el número de threads a utilizar, o dejar a la propia librería esta elección.
- PARDISO [151] (*Parallel Direct Sparse Solver*) forma parte de la familia MKL de Intel© y está optimizado para ofrecer un rendimiento mejorado en la resolución de grandes sistemas de ecuaciones lineales dispersos. Al igual que MKL, puede manejar datos reales y complejos, tanto en matrices simétricas como no simétricas, y admite paralelismo implícito para el aprovechamiento de los procesadores multicore. Para el manejo eficiente de la memoria, Intel© MKL PARDISO puede manejar los datos de las matrices dispersas almacenados en el formato CSR (*Compressed Sparse Row*).

Este formato almacena una matriz M de dimensiones $m \times n$ dispersa utilizando tres *arrays*: V , COL_INDEX y ROW_INDEX . Si NNZ es el número de valores no nulos de la matriz M , entonces:

- El *array* V tiene una longitud NNZ y contienen los valores distintos de cero.
- El *array* COL_INDEX contiene los índices de las columna de los valores no nulos y, por tanto, también tiene una longitud NNZ .
- El *array* ROW_INDEX contiene un elemento por cada fila de la matriz indicando el índice en V donde comienza dicha fila.

Por ejemplo, consideremos la siguiente matriz M con ocho elementos no nulos:

$$M = \begin{pmatrix} 7 & 12 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 21 & 0 & 0 \\ 0 & 0 & 11 & 30 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Su codificación siguiendo el formato CSR requiere los siguientes tres *arrays*:

$$\begin{aligned} V &= (7 \quad 12 \quad 9 \quad 21 \quad 11 \quad 30 \quad 3 \quad 2) \\ COL_INDEX &= (0 \quad 1 \quad 1 \quad 3 \quad 2 \quad 3 \quad 4 \quad 5) \\ ROW_INDEX &= (0 \quad 2 \quad 4 \quad 7) \end{aligned}$$

- HSL (*Harwell Subroutine Library*) [60] es una librería de software para computación científica escrita y desarrollada por el *Computational Mathematics Group* en el Laboratorio STFC Rutherford Appleton. Entre sus rutinas más conocidas se encuentran las relacionadas con la resolución de sistemas de ecuaciones lineales dispersos. Todos estos paquetes pueden trabajar con reales de simple y doble precisión, y algunos de ellos ofrecen interfaces para MATLAB y para el manejo de números complejos. La librería se inició en 1963 y se utilizó originalmente en el Laboratorio Harwell en mainframes de IBM, en ejecuciones sobre sus sistemas operativos OS y MVS. Con los

años, la librería HSL ha evolucionado y ha comenzado a utilizarse en una amplia gama de plataformas, desde supercomputadoras hasta modernos ordenadores personales. Las versiones recientes incluyen soporte optimizado para procesadores multinúcleo. De entre todas las rutinas que forman parte de esta librería, hemos seleccionado las siguientes, por ofrecer diferentes optimizaciones según el tipo de matrices y aprovechamiento de paralelismo:

- MA27 [61] implementa un resolutor de sistemas de ecuaciones lineales simétricos y dispersos, con interfaces para trabajar tanto con valores de simple como de doble precisión. La librería necesita conocer únicamente los valores no nulos de las matrices, que se almacenarán en tres *arrays* A , IRN y ICN que contendrán los valores, índices de las filas e índices de las columnas que ocupan los valores respectivamente. Al tratarse de matrices simétricas, se pueden informar indistintamente los valores por encima o por debajo de la diagonal. Por ejemplo, en la siguiente matriz:

$$M = \begin{pmatrix} 2 & 3 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 & 6 \\ 0 & 4 & 1 & 5 & 0 \\ 0 & 0 & 5 & 0 & 0 \\ 0 & 6 & 0 & 0 & 1 \end{pmatrix}$$

Los *arrays* almacenarán los siguientes valores:

$$\begin{aligned} A &= (2 \ 3 \ 4 \ 6 \ 1 \ 5 \ 1) \\ IRN &= (0 \ 0 \ 1 \ 1 \ 2 \ 2 \ 4) \\ ICN &= (0 \ 1 \ 2 \ 4 \ 2 \ 3 \ 4) \end{aligned}$$

No hay versión paralela de la rutina MA27.

- MA57 [61] reemplaza a MA27 para resolver sistemas simétricos indefinidos. Además de ser más eficiente, ofrece características adicionales como una rutina que implementa el refinamiento iterativo y la posibilidad de reiniciar la factorización si se queda sin espacio.

El almacenamiento de los valores de las matrices sigue el mismo formato que la rutina MA27. Al igual que aquélla, tampoco dispone de

versión paralela.

- MA48 [62] es una rutina desarrollada para resolver un sistema asimétrico disperso de m ecuaciones lineales con n incógnitas utilizando la eliminación Gaussiana. Los valores no nulos de las matrices se especifican siguiendo el mismo formato que con MA27 y MA57 pero, dado que en esta ocasión las matrices no son simétricas, es necesario indicar todos los valores. Al igual que aquellas, tampoco dispone de una versión paralela.
- MA86 [63] utiliza un método directo para resolver grandes sistemas de ecuaciones lineales indefinidos, simétricos y dispersos. Este paquete usa OpenMP y está diseñado para arquitecturas multinúcleo. El número de threads se indica en tiempo de ejecución con el comando correspondiente de OpenMP. El usuario debe ingresar la parte triangular inferior de la matriz M siguiendo el formato CSC (*Compressed Sparse Column*), con las entradas dentro de cada columna ordenadas aumentando el índice de fila. Este formato es diferente al empleado en MA27, MA57 y MA48, aunque también usa tres *arrays*, que se describen a continuación:
 - PTR , de tamaño igual a $n + 1$, donde n es el rango de la matriz M . $PTR[j]$, donde $j = 0 \dots (n - 1)$, debe establecerse de modo que $PTR[j]$ sea la posición en ROW de la primera entrada con valores no nulos en la columna j . $PTR[n]$ contendrá el número total de valores no nulos en la parte triangular inferior de M .
 - ROW es un *array* de tamaño $PTR[n]$. Debe contener los índices de fila de los valores de la parte triangular inferior de M , con los índices de fila para los valores en la columna 0 precediendo a los de la columna 1, y así sucesivamente.
 - VAL , es un *array* de tamaño $PTR[n]$. Las entradas deben establecerse de modo que $VAL[k]$ almacene el valor de la entrada en la posición k de ROW .

Por ejemplo, consideremos la siguiente matriz M :

$$M = \begin{pmatrix} -3 & 1 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 1 \\ 0 & 1 & 3 & 2 & 0 \\ 0 & 0 & 2 & 4 & 0 \\ 0 & 1 & 0 & 0 & 2 \end{pmatrix}$$

En este caso los *arrays* necesarios para identificar la matriz en formato CSC contendrán los siguientes valores:

$$\begin{aligned} PTR &= (0 \ 2 \ 5 \ 7 \ 8 \ 9) \\ ROW &= (0 \ 1 \ 1 \ 2 \ 4 \ 2 \ 3 \ 3 \ 4) \\ VAL &= (-3 \ 1 \ 4 \ 1 \ 1 \ 3 \ 2 \ 4 \ 2) \end{aligned}$$

- MAGMA (*Matrix Algebra on GPU and Multicore Architectures*) [146], es una librería de álgebra lineal densa similar a LAPACK pero para arquitecturas heterogéneas/híbridas, comenzando con sistemas multicore + GPU. Hay dos tipos de interfaces de estilo LAPACK. El primero, denominado interfaz de CPU, toma una entrada y produce el resultado en la memoria de la CPU. El segundo, denominado interfaz de GPU, toma una entrada y produce el resultado en la memoria de la GPU. En ambos casos se utiliza un algoritmo de cálculo híbrido de CPU+GPU. Esta librería también incluye MAGMA BLAS, un complemento de las rutinas BLAS. Muchas de las funciones que incorpora tienen la posibilidad de explotar más de una GPU. Ofrece variantes de todas las funciones para diversos tipos de datos y precisión y tipos de matrices.

3.2.3 Componentes auxiliares

El simulador obtiene información del rendimiento de las funciones y almacena en una base de datos un registro de cada ejecución para permitir su análisis posterior. Como repositorio de información hemos seleccionado SQLite [278].

Las motivaciones de esta elección han sido las siguientes:

- SQLite está desarrollado íntegramente en C y es un motor de base de datos de código libre, por lo que se puede integrar en el código fuente del componente del simulador encargado de gestionar las simulaciones, proporcionándole la funcionalidad del acceso a los tiempos de ejecución almacenados.
- A diferencia de la mayoría de las bases de datos SQL, SQLite no requiere un proceso de servidor separado.
- SQLite lee y escribe directamente en un único archivo en disco, que contiene todas las tablas, índices y vistas de la base de datos.
- El formato de dicho archivo es multiplataforma y puede intercambiarse y ser usado entre diferentes sistemas operativos.

3.3 Conclusiones

En este capítulo se han descrito las herramientas utilizadas durante la investigación realizada en esta tesis. Las técnicas que permiten optimizar los algoritmos, especialmente en el caso de problemas con alta carga computacional, deben tener en cuenta el hardware disponible para buscar el mayor aprovechamiento posible de todos los elementos de cómputo. Por tanto, es necesario tener un conocimiento de los modernos sistemas heterogéneos, compuestos generalmente de procesadores multicore, acompañados de uno o más coprocesadores, como son el caso de las GPU. Como ejemplo de este tipo de plataformas, hemos presentado el cluster *Heterosolar* de la Universidad de Murcia, donde podemos encontrar los elementos de cómputo ya mencionados, y que nos ha servido para la experimentación y comprobación de las técnicas de paralelismo propuestas.

Por otro lado, son igualmente relevantes las librerías de cómputo científico, en especial las librerías de álgebra lineal necesarias para resolver los problemas de análisis cinemático de sistemas multicuerpo tratados en este trabajo. Hemos presentado alguna de estas librerías, las cuales están incorporadas en el simulador. Esto nos permite usarlas y estudiar los rendimientos que ofrecen al operar

sobre diferentes tipos y tamaños de matrices. Esta información será utilizada posteriormente a la hora de proponer el uso de una u otra librería en la resolución de determinados problemas que incorporen en sus algoritmos operaciones de álgebra lineal, bien en áreas de la ingeniería mecánica, o en otras disciplinas científicas. El proceso de autooptimización también hará uso de esta información para seleccionar, en cada etapa de resolución de un problema, la mejor librería y los parámetros de paralelismo, combinación que vendrá determinada por el hardware disponible y por el número y tipo de cálculos que se puedan realizar de manera simultánea.

Capítulo 4

Optimización y Autooptimización

En este capítulo se revisan los conceptos y las técnicas más recientes en el dominio de la optimización de software, con especial énfasis en su aplicación al ámbito científico, donde es habitual encontrar algoritmos con alta demanda computacional cuya ejecución debe completarse en un tiempo razonable, o dentro de los límites de las asignaciones de recursos informáticos disponibles para el usuario.

A continuación se describen alternativas a la hora de abordar la autooptimización, entendiendo por tal el proceso de adaptación automática del software al entorno hardware disponible en el momento de la ejecución. Se hablará de una aproximación pseudo-teórica y de una aproximación experimental, las cuales han sido incorporadas en el simulador desarrollado en el marco de esta tesis.

4.1 Ideas generales

Como vimos en la sección 1.2.3, los sistemas informáticos modernos de alto rendimiento engloban una amplia gama de diseños. En la actualidad, estos sistemas abarcan desde clusters personalizados de computadoras personales, utilizando componentes de red de bajo coste para la comunicación, hasta sistemas que involucran redes de servidores comerciales y sistemas diseñados a medida, muchos de los cuales se pueden encontrar en la lista TOP500 [\[25\]](#), como es el caso

del supercomputador Summit [141] desarrollado por IBM. Al mismo tiempo, se han ensamblado sistemas heterogéneos que emplean componentes relativamente convencionales como son las CPUs multicore, habituales en los ordenadores personales, junto con aceleradores basados en GPU [169, 237] o MIC [154, 156]. Esta proliferación sin precedentes de arquitecturas dispares plantea un futuro en el que la tarea de conseguir el rendimiento óptimo de una aplicación científica es más desafiante que nunca. Lo que es más, comprender por qué un cálculo se ejecuta con un nivel de rendimiento insuficiente rara vez es sencillo. Por estas razones, las técnicas para analizar y optimizar el código buscando las mejores prestaciones han sido durante mucho tiempo un elemento básico del campo de la informática de alto rendimiento.

4.2 Optimización de software científico

En las últimas décadas, y a medida que la informática científica se ha desarrollado, el campo del análisis de rendimiento ha evolucionado a la par. Algunos conceptos aplicables en esta disciplina son:

- La monitorización de una aplicación científica durante su ejecución. Con esta información se puede comparar el rendimiento de un cálculo científico en varios sistemas informáticos disponibles, o estudiar el rendimiento de un sistema informático en diversas aplicaciones científicas.
- El modelado de rendimientos, que incluye técnicas y herramientas para reflejar el comportamiento de las aplicaciones y las prestaciones del sistema informático en modelos relativamente simples pero precisos.
- La optimización del rendimiento, aplicando técnicas y herramientas para realizar, de manera manual o automática, los cambios de código necesarios, o aplicar parámetros algorítmicos adecuados para optimizar el rendimiento de una aplicación científica.

4.2.1 Supervisión de rendimientos

El objetivo del proceso de supervisión es monitorizar con precisión el rendimiento de una aplicación científica mientras se ejecuta, recurriendo para ello a diversos métodos, como el recuento de operaciones de punto flotante y de enteros, tiempos de ejecución de rutinas, etc. Esta supervisión precisa del uso y desarrollo de técnicas y herramientas, bien ofrecidas por el propio hardware, bien mediante aplicaciones especializadas de monitorización:

- A nivel de hardware se pueden usar los contadores de rendimiento, que son registros disponibles en las CPU modernas (aunque también en controladores de memoria e interfaces de red) que cuentan algunos eventos de bajo nivel dentro del procesador con una sobrecarga mínima. Cada familia de procesadores tiene un conjunto diferente de contadores de hardware, a menudo con diferentes nombres incluso para los mismos tipos de eventos. La adopción generalizada del uso de contadores se vio obstaculizada en el pasado debido a la escasa documentación y a la falta de interfaces multiplataforma. Por lo general se necesita un soporte especial del sistema operativo para acceder a los contadores, pero en las primeras versiones de algunos sistemas operativos populares (como Linux) faltaba este soporte, por lo que se tenía que recurrir a conjuntos de parches puestos a disposición de los desarrolladores, como por ejemplo el paquete `perfctr` [275] (desarrollado por Mikael Pettersson, de la Universidad de Uppsala) y los proyectos `perfmon` y `perfmon2` [101, 277] (desarrollados por Stéphane Eranian). En 2009, el soporte de contadores de rendimiento finalmente se fusionó con el kernel de Linux a través del proyecto de contadores de rendimiento para Linux (PCL) (desde entonces renombrado como “eventos perf”). Ahora bien, la naturaleza diferente de los contadores de rendimiento según las plataformas, arquitecturas y sistemas operativos llevó a la necesidad de trabajar en una capa de abstracción para ocultar estas diferencias, tarea que se abordó mediante el proyecto Performance API (PAPI) [45, 143] en la Universidad de Tennessee, que lanzó en el año 2000 la primera versión de un interfaz independiente de la plataforma para las implementaciones de contadores de hardware.

- En cuanto al software de análisis de rendimientos, estos se centran en recopilar los aspectos de la ejecución de un programa que sean medibles y correlacionarlos con los contextos del programa en el que ocurren. Este tipo de software se basa en el uso de eventos, operaciones que se hacen visibles para el sistema de medición. Un desafío fundamental para las herramientas de rendimiento es cómo recopilar información detallada que permita identificar problemas de rendimiento sin distorsionar excesivamente la ejecución de una aplicación. La naturaleza de las mediciones, el coste de recopilarlas y los análisis que con ellas se pueden realizar se derivan directamente del enfoque utilizado para recopilar datos, que suele ser de dos tipos:
 - Basado en instrumentación: Se realiza implementando directamente en el programa controles en múltiples puntos, conocidos como sondas, y a los que se les puede asignar un código de medición. Las sondas se pueden usar para medir diferentes aspectos del rendimiento del programa. Por ejemplo, se puede medir el tiempo transcurrido entre un par de sondas insertadas en la entrada y salida de un procedimiento para determinar la duración de una llamada a dicho procedimiento. Los microprocesadores modernos también contienen contadores de rendimiento de hardware que pueden ser utilizados por las sondas para recopilar información. La medición basada en sondas es un enfoque sólido que proporciona la base para muchas herramientas de rendimiento en paralelo, como Scalasca [284, 299] y Vampir [132].
 - Basado en muestreo asíncrono: Se realiza mediante un muestreo periódico basado en disparadores (*triggers*). Cuando se activa un evento, el sistema operativo envía una señal al programa en ejecución, donde un controlador de señales instalado por una herramienta de rendimiento registra la ubicación del contador del programa donde se recibió la señal. En aplicaciones paralelas se pueden configurar disparadores para monitorizar cada hilo independientemente. La naturaleza recurrente de un activador de muestreo significa que la ejecución de un hilo se rastrea muchas veces, lo que permite obtener un conjunto de datos con información sobre el contexto en el que se ha realizado la llamada a la ejecución del hilo. Este método de medición se ha usado en herramientas como PerfSuite [276] y HPCToolKit [254].

4.2.2 Modelado de rendimientos

El objetivo del modelado en este ámbito es comprender el rendimiento de un sistema informático mediante mediciones y análisis, y encapsular estas características en una fórmula compacta. El modelo resultante se puede utilizar para obtener una mayor comprensión de los fenómenos de rendimiento involucrados y para proyectar el rendimiento a otras combinaciones de sistemas y aplicaciones. Un modelo integral debería incorporar factores tales como la arquitectura del sistema, la velocidad del procesador, latencia de red y ancho de banda, eficiencia del software del sistema, tipo de aplicación, algoritmos utilizados, lenguaje de programación, tamaño del problema, etc. Debido a la dificultad de producir un modelo verdaderamente completo, se suelen acotar los mismos para que sean válidos para un solo tipo de sistema y aplicación, y trabajar sobre variaciones de los componentes hardware del sistema y el tamaño del problema a resolver.

La aplicación más común de un modelo de rendimiento es la estimación del tiempo de ejecución de un trabajo cuando se cambian los datos de entrada del algoritmo o cuando se usa un número diferente de procesadores en un sistema de computación paralelo. También se puede estimar el tamaño más grande del sistema hardware que se debe utilizar para ejecutar un problema determinado con un nivel de rendimiento aceptable.

Si bien no se ha planteado en esta tesis la inclusión de fórmulas que modelen los tiempos de ejecución teóricos, se considera como una línea de trabajos futuro para complementar la técnica basada en la consulta de datos de entrenamiento ya incorporada en el simulador.

4.2.3 Optimización

Las arquitecturas de hardware heterogéneas existentes en la actualidad requieren de la aplicación de estrategias de optimización sobre los códigos para lograr un alto rendimiento. Los programadores deben emplear un tiempo significativo en reescribir y ajustar dichos códigos, tarea que a menudo no resulta trivial, pues un software que funciona bien en una plataforma a menudo presenta cuellos de botella en otra. De ahí el interés en desarrollar software de autooptimización.

Con carácter general, la autooptimización se basa en probar un conjunto de posibles cambios que se pueden aplicar activando ciertas opciones algorítmicas que implementan los programadores en el código de sus aplicaciones, o modificando los valores de los parámetros ofrecidos por las diversas bibliotecas de cómputo disponibles, o por los propios compiladores.

La autooptimización puede tener un carácter estático cuando fija el valor de parámetros que se leen una vez cuando se inicia el programa y permanecen fijos durante la ejecución de la aplicación. Para un ajuste óptimo de este tipo de parámetros estáticos hay que esperar a obtener información de rendimientos que se recopila tras varias ejecuciones de la aplicación completa. En el caso de que los parámetros se puedan modificar durante el tiempo de ejecución, se puede abordar una optimización más agresiva, y tiene la ventaja de explotar datos de rendimiento actualizados y precisos que se pueden vincular directamente a secciones de código específicas, características de conjuntos de datos de entrada, características específicas de arquitectura y condiciones cambiantes.

Dado que es posible efectuar a la vez cambios sobre el código desarrollado por el usuario y sobre las bibliotecas, es importante coordinar el proceso de optimización para evitar que cada componente ejecute su propio ajuste automático. De lo contrario es posible que un cambio mejore el rendimiento y el siguiente lo perjudique, con un beneficio neto escaso o nulo. Para esta coordinación hay varios enfoques posibles, que van desde un arbitraje simple para garantizar que solo se ejecute un ajuste a la vez, hasta un sistema unificado que permita la búsqueda coordinada simultánea de parámetros por medio de diferentes sintonizadores.

4.2.3.1 Optimización a nivel de aplicaciones

Los desarrolladores de aplicaciones y bibliotecas a menudo exponen un conjunto de parámetros de entrada para permitir a los usuarios finales ajustar el comportamiento de la aplicación a las características del entorno. La selección de valores de parámetros apropiados es, por lo tanto, crucial para garantizar un buen rendimiento, y requiere una buena comprensión de las interacciones entre los parámetros de entrada, los comportamientos algorítmicos subyacentes destinados a controlar dichos parámetros y los detalles de la arquitectura de destino [29].

En este sentido, se habla de autooptimización cuando se automatiza la búsqueda empírica dentro de un conjunto de valores de parámetros propuestos por el programador de la aplicación, a menudo en base a información obtenida a partir de ejecuciones anteriores [103, 139].

En [222] se analizan técnicas empíricas para encontrar el mejor valor entero dentro de una lista de parámetros de entrada de la aplicación que los programadores han catalogado como críticos para el rendimiento. Además, proporcionan modelos de alto nivel de impacto de los valores de los parámetros, que luego son utilizados por el sistema de ajuste para guiar el proceso de búsqueda de los valores óptimos.

En [138] se trata el ajuste de parámetros de entrada para la rutina de multiplicación matriz-matriz PDGEMM de ScaLAPACK [93, 162]. PDGEMM es parte de PBLAS [40], que es la implementación paralela de BLAS (*Basic Linear Algebra Subroutines*) [37] para sistemas de memoria distribuida.

En [58] los autores emplean una serie de ejecuciones cortas para evaluar los aspectos de la aplicación que se ven influenciados por la elección de los parámetros de entrada. Los autores utilizan su algoritmo de búsqueda (una modificación de la búsqueda Nelder-Mead [221]) para navegar por el espacio de parámetros de entrada. De esta manera evitan el costoso proceso de realizar ejecuciones completas de la aplicación en la búsqueda del ajuste de los parámetros.

4.2.3.2 Optimización de compilación

Con carácter general, las decisiones de optimización tomadas por los compiladores tienden a ser de propósito general y conservadoras. Por lo tanto, para dar a los programadores más flexibilidad a la hora de tomar decisiones que afectan a la optimización de códigos complejos, se han propuesto una variedad de marcos de autooptimización que facilitan la exploración de un gran espacio de posibles opciones del compilador y sus valores. Existen también herramientas de autooptimización que centran su esfuerzo en la fase de compilación del software, generalmente mediante la gestión de un conjunto de implementaciones alternativas de un determinado cálculo en base al estudio del hardware subyacente [55, 126].

Hay muchos proyectos de investigación que trabajan en la optimización empírica basada en compiladores de núcleos de álgebra lineal. ATLAS [297] realiza la optimización durante el proceso de construcción, y elige entre un conjunto de núcleos parametrizados para determinar los parámetros correctos para su uso en una máquina concreta, generando con ello rutinas BLAS altamente optimizadas. Un enfoque similar se emplea en PHiPAC [33]. Tanto PHiPAC como ATLAS intentan una cantidad limitada de ajustes de las opciones de *flags* del compilador durante el proceso de compilación.

La biblioteca Oski (*Optimized Sparse Kernel Interface*) [292] proporciona núcleos computacionales para matrices dispersas optimizados automáticamente. FFTW [111, 112] optimiza las FFT dinámicamente durante el uso de la biblioteca (además de generar núcleos en el momento de la compilación), con una intervención limitada por parte del usuario. SPIRAL [242] genera bibliotecas de procesamiento de señal digital (DSP) ajustadas empíricamente.

Para ganar en generalidad y facilitar el trabajo de los desarrolladores, algunos proyectos han trabajado en el desarrollo de extensiones del compilador, como en el caso de MILEPOST GCC [114], o extensiones de lenguaje, como en ABCLibScript [167], Orio [133] y ATF [249]. El trabajo realizado en MILEPOST GCC, revisado posteriormente en [23], es uno de los varios que utilizan técnicas de *Machine Learning* para mejorar la optimización del compilador. Aunque este tipo de enfoque fue considerado desde el principio, se descartó inicialmente porque muchos datos de entrenamiento debían estar disponibles con antelación. Hay que tener en cuenta que estos datos dependen del compilador y la arquitectura, lo que en la práctica no lo hace intercambiable entre plataformas.

En el proyecto TACT (*Tool for Automatic Compiler Tuning*) [241] se utiliza un algoritmo genético para optimizar el software para varias plataformas que contienen chips ARM Cortex A9. El método consiste en generar la cadena *CFLAGS* a partir de un conjunto de alrededor de 200 opciones disponibles.

4.2.3.3 Datos históricos como fuente de información

Los datos obtenidos a partir de ejecuciones de la aplicación en entornos de producción pueden aportar información valiosa de cara a la optimización del código, pues hacen innecesaria la ejecución de configuraciones de parámetros que ya se han evaluado en el pasado. Aplicaciones como PerfExplorer [136, 288] y Prophesy [303] se pueden usar para manejar conjuntos de datos históricos de rendimiento, y proporcionan una variedad de técnicas de análisis (agrupamiento, resumen, ajuste de curvas, etc.). Sin embargo, la utilización de datos de rendimiento obtenidos en ejecuciones previas tiene que considerar cuidadosamente las circunstancias contextuales bajo las cuales se recopilaron (plataforma, datos de entrada, versión del código fuente, compilador, sistema operativo, otras aplicaciones que se puedan estar ejecutando en el mismo instante, el uso de la red, etc.). El método convencional para tratar la variabilidad de los rendimientos obtenidos es utilizar múltiples muestras. El operador más utilizado para agregar valores de rendimiento de múltiples muestras es el promedio. Para aplicar la información obtenida a contextos que no coincidan exactamente con aquéllos en los que se tomaron las muestras se pueden usar técnicas de interpolación [140].

También se pueden utilizar datos de entrenamiento obtenidos mediante ejecuciones con configuraciones ligeramente diferentes (por ejemplo, diferentes tamaños de entrada). Habitualmente los modelos se entrenan con un conjunto de datos formado por una serie de puntos distribuidos regularmente en el espacio de parámetros de entrada. Si se mantiene localmente un registro de tales evaluaciones (junto con alguna información del contexto de su ejecución), el tiempo de ajuste se puede reducir consultando dicho registro. En el caso de contar con modelos de rendimiento, como los descritos en la sección 4.2.2, éstos se pueden consultar para obtener predicciones de rendimiento para diferentes configuraciones de entrada.

De las técnicas mencionadas en este apartado, el software desarrollado en esta tesis recurre a la información de entrenamiento obtenida al ejecutar las rutinas básicas que el usuario puede utilizar para construir sus códigos, lo que permitirá estimar el tiempo de ejecución de la aplicación al variar el tipo de datos y los parámetros paralelos de las librerías de cómputo.

4.2.3.4 Parámetros ajustables y espacios de búsqueda

Un aspecto importante de la búsqueda de parámetros es realizar una especificación precisa de los valores que éstos pueden tomar. Dicha especificación podría ser tan simple como expresar los valores mínimos, máximos e iniciales de un parámetro. A veces no se deben buscar todos los valores dentro del rango, por lo que es útil la opción de especificar una función de paso. La cantidad de parámetros, junto con los valores que cada uno de ellos puede adoptar, generarán un espacio de búsqueda sobre el que hay que considerar los siguientes aspectos:

- Su tamaño: Si el espacio de parámetros es simple se puede abordar una búsqueda exhaustiva. Pero en sistemas que contienen demasiadas combinaciones se hace conveniente el uso de alguna heurística de búsqueda para evaluar solo un subconjunto de las configuraciones posibles.
- Su naturaleza: En el caso de que los parámetros puedan adoptar valores restringidos y discretos se deben utilizar métodos apropiados para acotar la búsqueda a regiones permitidas del espacio de búsqueda.
- Su interdependencia: Es necesario considerar los casos en que los parámetros no sean independientes entre sí.

Para que la autooptimización de naturaleza empírica sea un enfoque razonable, se deben abordar estrategias para reducir el espacio de búsqueda de parámetros, focalizando el esfuerzo en las regiones interesantes del espacio de opciones.

En [287] se combina el uso de modelos de interacción de parámetros y el uso de restricciones identificadas por el programador para eliminar del espacio de búsqueda las configuraciones menos favorables. Otros sintonizadores automáticos [56, 244], que trabajan en la optimización empírica basada en modelos, también han examinado una serie de vías alternativas para descartar variaciones en el código. En general, el ajuste en tiempo de ejecución puede aprovechar el conocimiento del conjunto de datos de entrada para reducir aún más el espacio de búsqueda.

4.2.3.5 Algoritmos de búsqueda

Son diversos los algoritmos de búsqueda que se han utilizado en diferentes sistemas de autooptimización. ATLAS [297] optimiza cada parámetro ajustable de manera independiente, manteniendo el resto fijo a sus valores de referencia. Los parámetros se ajustan en un orden predeterminado, y cada ajuste sucesivo utiliza los valores optimizados para los parámetros que le preceden en el orden. La desventaja de usar esta búsqueda en un marco de uso general es que requiere un orden predeterminado para los parámetros. ATLAS aprovecha años de experiencia en optimización en el campo del álgebra lineal densa para determinar dicho orden. Otros autooptimizadores [64, 180] han utilizado algoritmos genéticos. Un algoritmo genético comienza generando aleatoriamente una población inicial de configuraciones posibles. Cada configuración se representa como un genoma y la “aptitud” de la configuración indica su rendimiento. Según la idoneidad, cada iteración sucesiva del algoritmo produce un nuevo conjunto de configuraciones mediante el uso de operaciones genéticas: mutación, cruce y selección. La desventaja radica en su largo tiempo de convergencia. Además, el comportamiento transitorio de un algoritmo genético es impredecible y variable, lo que hace que este algoritmo no sea adecuado para autooptimización en tiempo real. El algoritmo Nelder-Mead Simplex [221] es uno de los métodos de búsqueda directa más utilizados en sistemas de autooptimización. Looptool [245], que es un framework de autooptimización basado en compilador, utiliza el método de búsqueda directa basado en patrones. Este método es muy fiable, pero en algunos casos se ha comprobado que el tiempo de convergencia del algoritmo es elevado [175].

4.2.4 Autooptimización de rutinas paralelas de álgebra lineal

En 4.2.3.1 se han presentado algunos planteamientos de optimización de aplicaciones y de rutinas de álgebra lineal, como es el caso de la multiplicación de matrices de la librería paralela PBLAS. Las rutinas de álgebra lineal son ampliamente utilizadas en el ámbito de la computación científica, de ahí que su optimización se haya convertido en un campo de estudio muy dinámico.

En concreto, el grupo de investigación de Computación Científica y Programa-

ción Paralela de la Universidad de Murcia [239] ha elaborado diversos trabajos sobre autooptimización de rutinas paralelas de álgebra lineal. Por ejemplo, en [216] se aborda un ajuste automático en base a un modelado teórico-experimental de los tiempos de ejecución de las rutinas aplicable en sistemas paralelos homogéneos de memoria compartida y distribuida. El modelo tiene en cuenta la influencia de la plataforma en los tiempos de ejecución mediante la definición de un conjunto de parámetros de sistema, y busca la selección automática de los parámetros algorítmicos que minimizan el tiempo de ejecución de cada rutina. En [73] se propone un sistema de compilación (*Poly-Compilation Engine*) para sistemas paralelos de memoria compartida, donde se genera una versión ejecutable de las rutinas para cada compilador instalado en la plataforma. Este sistema de compilación múltiple crea un modelo teórico-experimental del tiempo de ejecución de cada versión, que incluye el coste de creación de los threads, y que es usado en tiempo de ejecución para la selección de la mejor rutina y parámetros algorítmicos.

En [76] se emplea un enfoque experimental aplicado a sistemas heterogéneos compuestos de una CPU multicore y uno o más coprocesadores. Este estudio realiza una búsqueda de los mejores parámetros algorítmicos para un conjunto de tamaños de problema definidos durante la fase de instalación. En tiempo de ejecución se usarán los parámetros asociados al conjunto de instalación más cercano al problema a resolver. En [77] se estudia la selección de la mejor versión compilada de la rutina en función de la plataforma de ejecución. En [72] las rutinas se optimizan durante la instalación para cada plataforma concreta y, en tiempo de ejecución, los parámetros que definen a las plataformas se ajustan a las condiciones de carga de trabajo de los procesadores y de la red de interconexión.

En referencia a técnicas de modelado (sección 4.2.2), este grupo de investigación también ha elaborado propuestas, como en [75, 78, 116, 118], donde se diseñan modelos analíticos de rutinas paralelas para sistemas distribuidos y plataformas híbridas. En el caso de rutinas que presentan una estructura jerárquica, cuando una rutina contiene en su código llamadas a rutinas de librerías pertenecientes a un nivel inferior, se puede utilizar la información de autooptimización de las rutinas llamadas para optimizar la rutina que se está tratando en cada momento [79, 117].

En [48, 49] se desarrolla una metodología de autooptimización jerárquica que

puede ser aplicada en diferentes niveles hardware y software. En relación al hardware, esta metodología permite que la ejecución de las rutinas pueda ser optimizada para cualquier combinación de unidades de procesamiento paralelo que se agrupen de manera jerárquica como componentes de nodos de cómputo, agrupados a su vez en clusters. Del mismo modo, a nivel de software, el proceso de optimización se puede llevar a cabo de forma jerárquica entre rutinas de diferentes niveles, permitiendo que rutinas de nivel superior (como la multiplicación de Strassen o la factorización LU) puedan ejecutarse utilizando versiones optimizadas de las rutinas matriciales básicas a las que invocan, como por ejemplo la multiplicación de matrices.

4.2.5 Ejemplo de aplicación: optimización de un modelo de predicción climático

Como ejemplo de aplicación de las técnicas de optimización comentadas anteriormente a un software científico, citamos la descripción que en [27] se hace del proceso de optimización continua aplicado durante décadas a un software científico de alta demanda computacional, el modelo de simulación del clima denominado CCSM [220] (*Community Climate System Model*). El software está compuesto por cuatro componentes geofísicos que modelan la atmósfera, el océano, la tierra y el hielo marino, e intercambian datos de movimientos de masas, impulsos y energías a través de un quinto módulo de interconexión con objeto de crear, en su conjunto, un modelo climático global.

Desde el año 1983, en que se creó la primera versión de CCSM, este software ha venido siendo objeto de constante evolución tendente a mejorar su rendimiento, con un enfoque especial en la comunicación entre procesos, equilibrio de cargas y algoritmos paralelos. La concurrencia de los componentes de CCSM no es perfecta, ya que hay dependencias temporales entre algunos de ellos, lo que limita la fracción de tiempo en el que sus componentes se pueden ejecutar simultáneamente. Por ejemplo, el modelo de atmósfera, tierra y hielo marino se pueden ejecutar en un conjunto común de procesadores, mientras que el modelo oceánico se ejecuta simultáneamente en un conjunto separado de procesadores.

Con la versión CCSM4 [219], publicada en mayo de 2010, todos los componentes implementan códigos paralelos híbridos, que usan MPI para definir y coordinar el paralelismo de memoria distribuida, y OpenMP para el paralelismo de memoria compartida. Cada modelo de componente tiene sus propias características de rendimiento, y la integración entre ellos se suma a la complejidad a la hora de caracterizar el rendimiento [96]. El autor de este análisis resalta las técnicas que se han empleado con buenos resultados en la mejora de rendimientos del modelo CCSM [302]:

- Técnicas para reducir el tamaño del espacio de búsqueda de parámetros a la hora de determinar la configuración óptima del algoritmo.
- Adaptación a nuevas arquitecturas de hardware, con estudios detallados de la escalabilidad del rendimiento, con un seguimiento de las mejoras obtenidas en todos los módulos de la simulación, y para todos los tamaños de problemas.
- Avances en el modelado de rendimientos, necesidad derivada tanto de las investigaciones en el desarrollo de nuevas técnicas de programación paralela como de la creciente lista de plataformas computacionales. Y es que al comparar tanto algoritmos como plataformas, cada uno de los algoritmos alternativos debía optimizarse en las plataformas objetivo. En este proceso se comprobó la utilidad de una metodología experimental eficiente para determinar las opciones óptimas, ya que el mejor enfoque era una función del tamaño del problema, número de procesadores y sistema informático de destino.
- Soporte para varias capas de mensajes, algoritmos de comunicación y protocolos, y su implementación en determinadas secciones del código, pudiendo especificarse en tiempo de compilación o en tiempo de ejecución. En CCSM el protocolo de comunicación tiene un gran impacto en el rendimiento, y la mejor opción varía con el algoritmo paralelo, tamaño del problema, número de procesos, biblioteca de comunicación y arquitectura de destino. En consecuencia, el protocolo de comunicación utilizado varía entre los diferentes componentes de CCSM, admitiendo algunos de ellos la selección, tanto en

tiempo de compilación como en el momento de su ejecución en un algoritmo paralelo determinado.

- Posibilidad de modificar en tiempo de compilación o tiempo de ejecución ciertos parámetros algorítmicos, tales como la dimensión de determinadas estructuras de datos, el equilibrio de cargas y el uso de paralelismo MPI frente a OpenMP.

4.3 Optimización de códigos de simulación de sistemas multicuerpo

En esta sección se analiza la aplicación de algunas de las técnicas de optimización mencionadas en las secciones anteriores a los códigos de simulación de sistemas mecánicos multicuerpo, cuyos modelos computacionales quedaron descritos en la sección 2.3.5. Describiremos dos metodologías de autooptimización y su aplicación en el simulador que se ha desarrollado.

4.3.1 Características de los algoritmos

En el capítulo 2 se describió cómo, mediante un adecuado análisis estructural de los mecanismos, es posible hallar la división de un sistema multicuerpo en los subsistemas que lo componen y determinar el orden en que se deben analizar. Este tipo de modelado, basado en ecuaciones de grupo, admite una implementación computacional modular que permite la introducción de técnicas de paralelismo con el objetivo de obtener una reducción en los tiempos de ejecución. Además, los algoritmos de simulación de mecanismos se componen básicamente de rutinas que incluyen operaciones de álgebra lineal sobre matrices. Estas matrices pueden presentar diferentes tamaños y factores de dispersión en función de la topología del sistema mecánico.

Por tanto, una optimización de este tipo de códigos puede enfocarse en varias áreas:

- Aplicando algún paradigma de programación paralela a la hora de resolver los subsistemas que componen el sistema mecánico. El conjunto de parámetros algorítmicos a optimizar vendrá dado en función de la plataforma hardware. Por ejemplo, en plataformas dotadas de CPU multicore, y usando paralelismo OpenMP, se deberá estudiar la mejor selección del número de threads.
- Seleccionando la librería de álgebra lineal óptima, decisión que vendrá influenciada por la naturaleza y el tamaño del problema a resolver y su representación matricial. Además, ciertas librerías tienen versiones paralelas, con lo que es necesario ajustar también su asignación de threads.
- Dado que las optimizaciones anteriores pueden entrar en conflicto entre ellas, será necesario abordar una optimización global. Por ejemplo, en librerías con versión paralela habrá competencia por el número de threads. Y en librerías que ejecutan cálculos en GPUs, será necesario tener en cuenta el número de dispositivos instalados.

En el caso del simulador desarrollado junto a esta tesis, los parámetros algorítmicos considerados contemplan el número de threads del primer nivel (paralelismo explícito OpenMP), los threads del segundo nivel (aplicables al paralelismo interno de las rutinas de las librerías), la cantidad de GPUs y el identificador de la librería. Todos ellos son de naturaleza discreta, por lo que se pueden representar mediante números enteros especificados individualmente o como un rango definido entre un valor mínimo y un valor máximo.

4.3.2 Autooptimización en el simulador

En el capítulo siguiente se describirá el simulador desarrollado en esta tesis, el cual, a partir de un algoritmo elaborado por un usuario, permite identificar las instrucciones que se pueden ejecutar de manera simultánea, y con esa información construir un árbol en el que cada rama representa una ordenación válida de los cálculos. La rama óptima será aquella que, tras una adecuada selección de los parámetros de paralelismo y de la librería de cálculo, permita resolver el algoritmo completo en el menor tiempo de ejecución.

La búsqueda de la mejor rama, como parte del proceso de optimización de los algoritmos, se puede abordar en el simulador mediante dos técnicas que, atendiendo a la metodología empleada, se clasifican en:

- **Experimental:** En esta técnica se llevan a cabo ejecuciones exhaustivas de la simulación, variando los parámetros algorítmicos y el conjunto de datos de entrada, obteniendo de esta manera un conocimiento de las prestaciones del mismo en función del hardware y la naturaleza del problema a resolver. La información es registrada y ordenada para mostrar la mejor opción. En este caso se observa que no es necesario un conocimiento previo del algoritmo.
- **Pseudo-teórica:** En general esta aproximación trata de inferir el tiempo de ejecución de algún tipo de código científico basándose en un conocimiento del algoritmo subyacente y en los tiempos de ejecución reales de las rutinas básicas. Estas suposiciones vienen a conformar un modelo, explícita o implícitamente. Los tiempos de ejecución de las funciones elementales se sumarán para determinar el tiempo de ejecución de una determinada sección de un código de naturaleza secuencial que las contenga. El tiempo de ejecución de secciones del código que puedan calcularse de manera simultánea vendrá dado por el de mayor carga computacional. En el simulador este método decide, antes de iniciar la ejecución, cuál es la mejor rama para el hardware disponible. Para que esto sea posible será necesario haber entrenado el sistema realizando ejecuciones de las funciones elementales con varios tamaños y tipos de matrices, variando a su vez los parámetros algorítmicos ajustables (por ejemplo, en librerías paralelas, asignando diferentes números de threads). Para contemplar una posible variabilidad en las prestaciones del hardware, se puede decidir repetir el entrenamiento en diferentes instantes, en cuyo caso el autooptimizador utilizará los valores medios de los tiempos obtenidos.

Una vez entrenado el sistema, el simulador puede estimar el coste de cada rama sumando los tiempos de ejecución de los nodos que la componen. Cuando un nodo resuelve simultáneamente más de un grupo de cálculos, se reparten entre ellos los recursos disponibles (cores y GPUs), proceso que supone contemplar todas las distribuciones teóricas posibles. Finalmente se asocian librerías de cálculo a cada función dentro de los grupos, respetando

el hardware que se acaba de asignar (por ejemplo, no se usará una librería que requiera una GPU si el grupo no ha recibido la asignación de alguna de estas unidades de cómputo).

4.4 Conclusiones

En este capítulo se han descrito los conceptos más importantes que se aplican en el área de la optimización de aplicaciones científicas, y se han presentado los recientes avances en herramientas y metodologías disponibles en este campo. Además, se ha mostrado la aplicación de alguna de estas técnicas al campo de la optimización de librerías de álgebra matricial y a aplicaciones científicas más generales y de alto coste computacional, como es el caso de un modelo climático global.

Por último, se ha realizado una breve introducción a la funcionalidad de autooptimización incorporada en el simulador de sistemas multicuerpo desarrollado durante la elaboración de esta tesis, y se han explicado las dos aproximaciones, experimental y pseudo-teórica, que cubre dicho simulador.

En el siguiente capítulo se realiza una descripción completa del software, y se analiza la extensión de su uso a otros problemas más generales cuyo planteamiento pueda ser descrito en base a resoluciones paralelas de funciones de álgebra lineal. Por último, un capítulo dedicado a experimentos recogerá ejemplos concretos que mostrarán la aplicación real de las técnicas aquí presentadas.

Capítulo 5

Simulador

Este capítulo ofrece una descripción del software de simulación PARCSIM (PAR-allel C-omputations SIM-ulator). Este simulador es una aplicación software que permite a un usuario capturar, a través de un interfaz gráfico, algoritmos que resuelven problemas científicos mediante técnicas de álgebra matricial, y experimentar posteriormente los cálculos empleando diferentes parámetros de paralelismo y librerías de cómputo, analizando los tiempos de ejecución obtenidos en cada caso. Un modo de ejecución autooptimizado ayuda a usuarios no expertos a la selección de la mejor configuración de parámetros, proponiendo la ordenación temporal de los cálculos que permita una mejor explotación del paralelismo en un hardware concreto, conocidos el número de cores de la CPU y el número de GPUs instaladas.

Se comienza el capítulo recordando el contexto en el que se enmarca este simulador y las motivaciones que han llevado a su desarrollo. A continuación se realiza un recorrido por los conceptos que permiten explicar el diseño y ejecución del software. Finalmente se detallan las funcionalidades incorporadas y se describen la arquitectura y los componentes del mencionado software.

5.1 Motivación

Se pueden encontrar multitud de problemas científicos donde es posible representar en forma de grafo dirigido la división de un sistema (mecánico, físico, algebraico) complejo en un conjunto ordenado de subsistemas determinados (resolubles). En un grafo de este tipo los nodos corresponden a bloques de instrucciones que resuelven cada subsistema, y las líneas dirigidas entre los nodos indican el orden en que se deben calcular los bloques. Un estudio de las dependencias reflejadas en dicho grafo nos permite identificar los subsistemas que se pueden resolver en paralelo (lo que llamaremos primer nivel de paralelismo).

En el ejemplo mostrado en la figura 5.1 se representa la división de un determinado sistema en cinco subsistemas, representados por los nodos en el grafo. Aparentemente, los nodos 2 y 3 podrían resolverse de manera simultánea ya que no se muestran dependencias entre ellos.

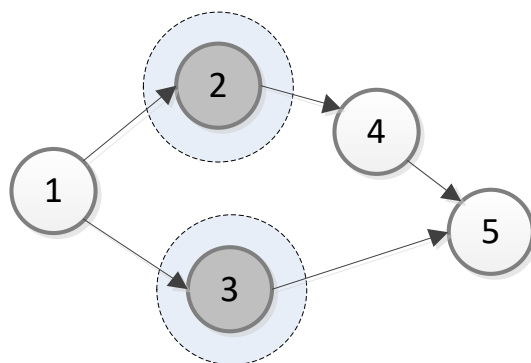


Figura 5.1: Representación mediante un grafo dirigido de la división de un sistema en un conjunto ordenado de subsistemas donde los nodos representan conjuntos de instrucciones. Según se observa en el grafo, los nodos 2 y 3 podrían resolverse simultáneamente.

Con carácter general, en implementaciones computacionales de problemas así planteados, el algoritmo con mayor potencial de eficiencia en términos de velocidad de ejecución sobre plataformas paralelas será aquel que identifique el mayor número de bloques de cálculos paralelizables. Y la implementación más rápida será la que asigne los recursos computacionales de manera que la ejecución paralela de bloques sea la más rápida de entre las posibles.

La búsqueda de la mejor implementación constituye una tarea compleja que puede ser abordada, como se describió en la sección 4.3.2, desde un punto de vista pseudo-teórico (estimando tiempos de ejecución teóricos en base a tiempos reales medidos en ejecuciones de las operaciones básicas) o desde un punto de vista experimental (donde se busca la mejor asignación de recursos analizando el comportamiento observado tras varias ejecuciones que exploren diferentes alternativas).

El simulador aquí presentado tiene como objetivo facilitar dicha tarea de búsqueda, ofreciendo un interfaz gráfico que permite guiar a un usuario no experto en paralelismo en la selección de la mejor implementación de su código, y conseguir de ese modo el aprovechamiento óptimo de una plataforma hardware concreta.

Por su diseño, PARCSIM permite realizar simulaciones computacionales en el campo del análisis cinemático de sistemas multicuerpo mediante la resolución de un conjunto ordenado de subestructuras de cinemática determinada (calculable) obtenidas mediante un análisis estructural del mecanismo (sección 2.3). En problemas de esta rama científica se requiere del uso de librerías de álgebra matricial para resolver la cinemática de los mecanismos, por lo que el simulador incluye el conjunto de operaciones de álgebra lineal que, con carácter general, son empleadas en este tipo de algoritmos de resolución.

Además de la aplicación directa en el ámbito de la ingeniería mecánica, su uso se puede extender a otras áreas científicas, en especial a aquellos problemas resolubles mediante la ejecución ordenada de rutinas de álgebra matricial.

5.2 Conceptos básicos

En esta sección se describen los conceptos que definen la estructura de la información manejada por el simulador, así como aquellos relacionados con el modo en que el software representa el algoritmo del usuario y con los parámetros que gestionan la simulación del mismo.

5.2.1 Funciones

En PARCSIM se denominan Funciones aquellas operaciones de álgebra lineal incorporadas en el simulador y que el usuario puede usar en un algoritmo de resolución de un problema concreto. Según su complejidad, se pueden distinguir dos grupos de funciones:

- Operaciones algebraicas básicas y transformaciones de matrices:
 - MATADD: Suma de dos matrices del mismo orden:
 $C = \text{MATADD}(A, B) \rightarrow C = A + B$, con $A, B, C \in \mathfrak{R}^{m,n}$
 - MATSUB: Resta de dos matrices del mismo orden:
 $C = \text{MATSUB}(A, B) \rightarrow C = A - B$, con $A, B, C \in \mathfrak{R}^{m,n}$
 - NORMA: Norma de un vector $a = [a_1, a_2, \dots, a_n]$
 $\text{NORMA}(A) = \sqrt{\langle a, a \rangle} = \sqrt{\sum_{i=1}^n a_i a_i}$
 - MATCHG: Cambio de signo de los elementos de una matriz $A = (a_{ij})$
 $\text{MATCHG}(A) = -A$, $(a_{ij} = -a_{ij}, \forall_{i,j})$
 - MATTRN: Transposición de una matriz $A = (a_{ij})$
 $\text{MATTRN}(A) = A^t = (a_{ij}^t)$, $a_{ij}^t = a_{ji}$, $1 \leq i \leq m, 1 \leq j \leq n$
 - RESHAP: Cambio de las dimensiones de una matriz. Dada una matriz $A \in \mathfrak{R}^{m,n}$, esta función crea una lista con todos sus elementos y los reorganiza, por columnas, generando otra matriz $B \in \mathfrak{R}^{r,s}$
 $B = \text{RESHAP}(A, B)$, con $mn = rs$
- Funciones implementadas en librerías externas:
 - MATMUL: Multiplicación de dos matrices $A \in \mathfrak{R}^{m,n}$ y $B \in \mathfrak{R}^{n,p}$
 $C = \text{MATMUL}(A, B) \rightarrow C = AB$, $c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$, $1 \leq i \leq m, 1 \leq j \leq p$
 - DGETRF: Descomposición LU de una matriz cuadrada $A = (a_{ij})$
 $\text{DGETRF}(A) = \{L, U\}$, con L matriz triangular inferior y U matriz triangular superior unitaria tal que $A = LU$
 - SOLVESYS: Resolución de un sistemas de ecuaciones $Ax = b$
 $x = \text{SOLVESYS}(A, b)$, con $A \in \mathfrak{R}^{m,n}$, $b, x \in \mathfrak{R}^{m,1}$

Dada su relevancia en la resolución de numerosos problemas científicos, algunas de estas funciones han sido objeto de múltiples adaptaciones computacionales a tipos específicos de matrices y plataformas hardware. En ocasiones dichas versiones mejoradas se han incorporado a librerías de cálculo que se han puesto a disposición de la comunidad científica.

La tabla 5.1 muestra las librerías incorporadas en la versión actual del simulador. Junto a ellas se relacionan las funciones y el tipo de matrices que manejan. Cada librería se identifica de manera única mediante un identificador, `Id`. El `Id` ‘0’ corresponde a las funciones implementadas en PARCSIM. El resto son librerías de terceros cuyos detalles se pueden consultar en la sección 3.2.2.

Librería		Tipo de matrices		Funciones
Nombre	Id	Densidad	Topología	
PARCSIM	0	Densas	No simétricas	MATADD, MATSUB, MATCHG, MATTRN, NORMA, RESHAP
MKL	1	Densas	No simétricas	MATMUL, DGETRF, SOLVESYS
PARDISO	2	Dispersas	No simétricas	SOLVESYS
MA27	3	Dispersas	Simétricas	SOLVESYS
MA57	4	Dispersas	Simétricas	SOLVESYS
MA48	5	Dispersas	No simétricas	SOLVESYS
MA86	6	Dispersas	Simétricas	SOLVESYS
MAGMA	7	Densas	No simétricas	MATMUL, SOLVESYS

Tabla 5.1: Listado de librerías incorporadas en el simulador y funciones de álgebra lineal que implementan. Se incluye información sobre el tipo de matrices soportadas en cada caso.

En una próxima evolución del software se permitirá incluir funciones de cálculo adicionales, bien desarrolladas por el propio usuario de la aplicación, o mediante el enlace con librerías externas. Esta línea de trabajo futuro se encuentra descrita en la sección 7.2.

El simulador almacena la información de las funciones en un fichero de texto, llamado `functions.fun`, donde se recogen, de una manera jerárquica y estructurada, las configuraciones de cada una de ellas mediante pares nombre-valor. El listado 5.1 muestra una sección de dicho fichero con la información específica para la función MATADD.

Listado 5.1 Extracto del fichero `functions.fun` que recoge la descripción de las funciones disponibles en el simulador. Se muestra la información relativa a la suma de matrices MATADD.

```
    ...
},
{
  functionName    = "MATADD";
  functionId      = 5;

  functionParameters =
  (
    {Name="inMatrixA "; Direction="i "; Type="Mf";},
    {Name="inMatrixB "; Direction="i "; Type="Mf";},
    {Name="outMatrixC "; Direction="o "; Type="Mf";}
  );

  parametersValidations =
  (
    {
      condition="R1=R2";
      errortext="The matrices must have the same number of rows";
    },
    {
      condition="R2=R3";
      errortext="The matrices must have the same number of rows";
    },
    {
      condition="C1=C2";
      errortext="The matrices must have the same number of columns";
    },
    {
      condition="C2=C3";
      errortext="The matrices must have the same number of columns";
    }
  );
  functionPackages =
  (
    {packageName="PARCSIM";
      packageId=0;
      packageFunctionId=3010;
      packageMultiThread=0;
      packageMatrixFormat=1;}
  );
}
    ...
```

A continuación se describen los campos mostrados en el listado:

- `functionId`: Es un número entero que representa de manera única y absoluta a una función en el simulador, de acuerdo a los valores recogidos en la tabla 5.2.
- `functionName`: Contiene una cadena alfanumérica con la descripción de la función.

Id	Función
1	DGETRF
2	DGETRS
3	SOLVESYS
4	MATMUL
5	MATADD
6	MATSUB
7	MATCHG
8	MATTRN
9	NORMA
10	RESHAP

Tabla 5.2: Lista de funciones de álgebra lineal incorporadas en el simulador y sus identificadores.

- `functionParameters`: Contiene una terna (o una lista de ellas) con información de cada uno de los parámetros requeridos por la función. En este contexto, los parámetros corresponden a los argumentos de las funciones:
 - `Name`: Una cadena alfanumérica que contiene el nombre del parámetro.
 - `Type`: El tipo del formato numérico del parámetro. En la versión actual del simulador se permiten parámetros con formato de matrices de enteros (`Mi`) o de valores reales con doble precisión (`Mf`).
 - `Direction`: Indica si el parámetro corresponde a un argumento de entrada (`i-nput`) o de salida (`o-utput`) de la función.
- `parametersValidation`: Contiene un par (o lista de pares) que representan las validaciones que el simulador impone a los argumentos que recibe la función en el momento de su ejecución:
 - `condition`: Expresa una condición que debe satisfacerse, y que relaciona el número de filas o columnas de los parámetros. Ambos lados de la igualdad se forman con *C* o *R* (como referencia al número de columnas o filas respectivamente) seguido por el orden del parámetro afectado. Por ejemplo, la fórmula $C1 = C2$ comprueba que el número de columnas de la matriz que se recibe como primer (1) parámetro coincide con el número de columnas de la del segundo (2).
 - `errorText`: Contiene el mensaje que el simulador muestra al usuario en caso de no cumplirse dicha validación.

- `functionPackages`: Contiene información sobre las librerías que implementan la función, y especifica el formato de datos y el tipo de paralelismo implícito que admite. Se requiere una 5-tupla (o lista de 5-tuplas) con el siguiente desglose:
 - `packageName`: Contiene el nombre de la librería (que podrá ser cualquiera de las mostradas en la tabla 5.1).
 - `packageId`: Un número que representa de manera única y absoluta a la librería en el simulador, de acuerdo a los valores mostrados en la tabla 5.1.
 - `packageFunctionId`: Un número entero que identifica de manera única en el simulador la implementación específica de la función `functionId` en la librería `packageId`.
 - `packageMultiThread`: Su valor indica el tipo de paralelismo implícito de la función. Las opciones disponibles en el simulador son:
 - 0: La función no tiene implementación paralela.
 - 1: La función admite paralelismo OpenMP en sus rutinas internas.
 - 2: La función admite paralelismo MKL (aplicable solo en funciones integradas en la librería MKL de Intel).
 - 3: La función puede ejecutarse en una GPU.
 - `packageMatrixFormat`: Especifica el formato en el que se deben almacenar las matrices en memoria para poder ser usadas por la librería (en la sección 3.2.2 se encuentra una descripción ampliada de dichos formatos). Los valores admitidos son los siguientes:
 - 1: Las matrices se representan mediante *arrays* bidimensionales.
 - 2: Las matrices se representan utilizando el formato CSR (*Compressed Sparse Row*).
 - 3: Se emplea para representar matrices simétricas, y usa tres *arrays* unidimensionales que almacenan los valores no nulos, los índices de las filas y los índices de las columnas que ocupan dichos valores respectivamente. Se pueden informar indistintamente los valores por encima o por debajo de la diagonal.

- 4: Es el mismo formato que el anterior pero aplicable a matrices no simétricas, por lo que los *arrays* deben hacer referencia a todos los valores no nulos de la matriz, indistintamente de si están por encima o por debajo de la diagonal.
- 5: Las matrices se representan utilizando el formato CSC (*Compressed Sparse Column*).

Para ilustrar la relación que existe entre las funciones y las librerías en el fichero `functions.fun` podemos fijarnos en la suma de matrices `MATADD`. Actualmente esta función está implementada únicamente en el simulador, por lo que en la sección `functionPackages` del listado 5.1 solo hay una entrada, la referida al propio simulador. Por el contrario, la función `SOLVESYS` tiene implementaciones en todas las librerías, motivo por el que la sección `functionPackages` muestra varias entradas, como queda reflejado en el listado 5.2.

Listado 5.2 Extracto del fichero `functions.fun` con información relativa a las librerías que implementan la resolución de sistemas de ecuaciones `SOLVESYS` en el simulador.

```
functionPackages = (  
  {packageName="MKL";   packageId=1; packageFunctionId=1010;  
    packageMultiThread=2; packageMatrixFormat=1;},  
  {packageName="PARDS"; packageId=2; packageFunctionId=1020;  
    packageMultiThread=2; packageMatrixFormat=2;},  
  {packageName="MA27";  packageId=3; packageFunctionId=1030;  
    packageMultiThread=0; packageMatrixFormat=3;},  
  {packageName="MA57";  packageId=4; packageFunctionId=1040;  
    packageMultiThread=0; packageMatrixFormat=3;},  
  {packageName="MA48";  packageId=5; packageFunctionId=1050;  
    packageMultiThread=0; packageMatrixFormat=4;},  
  {packageName="MA86";  packageId=6; packageFunctionId=1060;  
    packageMultiThread=0; packageMatrixFormat=5;},  
  {packageName="MAGMA"; packageId=7; packageFunctionId=1070;  
    packageMultiThread=3; packageMatrixFormat=1;}  
);
```

En la versión actual de PARCSIM el fichero `functions.fun` se distribuye empaquetado con el resto del software y no puede ser modificado. Como se indicó anteriormente, en futuras versiones se podrán incluir otras implementaciones de funciones, eficientes y con posibilidad de paralelismo, desarrolladas por un tercero o por el propio usuario, proceso que modificará automáticamente el fichero.

5.2.2 Rutinas de usuario

En PARCSIM una rutina representa una secuencia de cálculos o funciones que, desde el punto de vista del usuario, puede resolver un determinado problema o una parte de él. Durante la simulación, una rutina supone la ejecución en secuencia de todas las funciones que la componen. El usuario puede crear todas las rutinas que necesite. Una vez creadas, se pueden reutilizar en la resolución de más de un problema. Se distinguen dos tipos de rutinas:

- Simples: las formadas únicamente por funciones.
- Anidadas: pueden contener funciones y referencias a otras rutinas.

Como ejemplo, la figura 5.2(a) muestra la rutina simple R_1 con un diseño que contiene tres funciones: f_1 , f_2 y f_3 que se ejecutan en secuencia. Sin embargo, si observamos la figura 5.2(b), la rutina anidada R_2 se compone de tres funciones y una referencia a la rutina R_1 : f_1 , R_1 , f_2 y f_4 . La ejecución de R_2 comienza desglosando la rutina R_1 en sus funciones componentes e insertándolas en el lugar que ocupa R_1 dentro de R_2 .

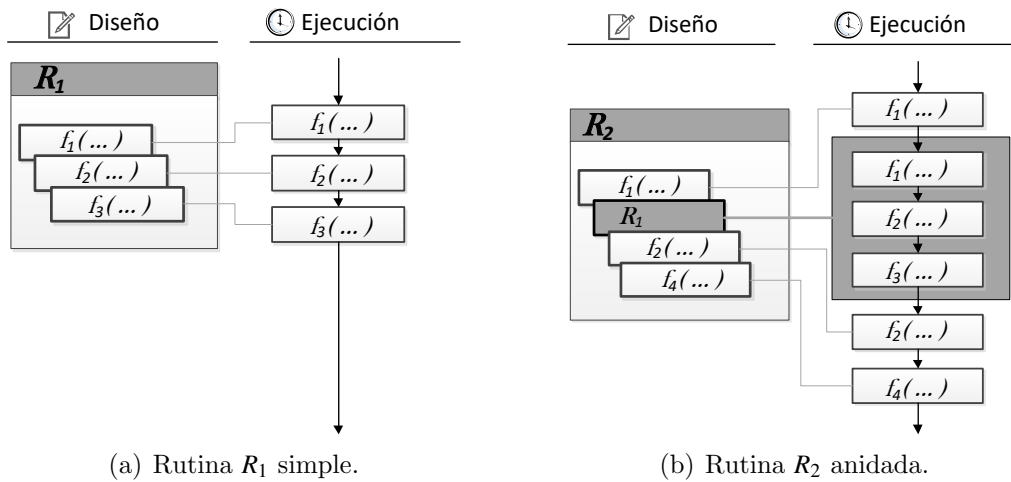


Figura 5.2: Ejemplo de dos rutinas de usuario que ilustran el concepto de rutina simple (compuesta únicamente por funciones) y rutina anidada (compuesta por funciones y por otras rutinas). En tiempo de ejecución la rutina anidada aporta a la secuencia de cálculos las funciones que la forman.

PARCSIM almacena la definición de todas las rutinas creadas por un usuario en el fichero de texto `routines.rou`, donde la información se presenta con una estructura jerarquizada. El listado 5.3 corresponde a la sección del fichero `routines.rou` con información relativa a una rutina simple compuesta por dos funciones, una suma y una multiplicación de matrices.

Listado 5.3 Extracto del fichero `routines.rou` que muestra la información relativa a la rutina `RADDMUL` creada por el usuario y compuesta por las funciones del simulador `MATADD` y `MATMUL`, suma y multiplicación de matrices respectivamente.

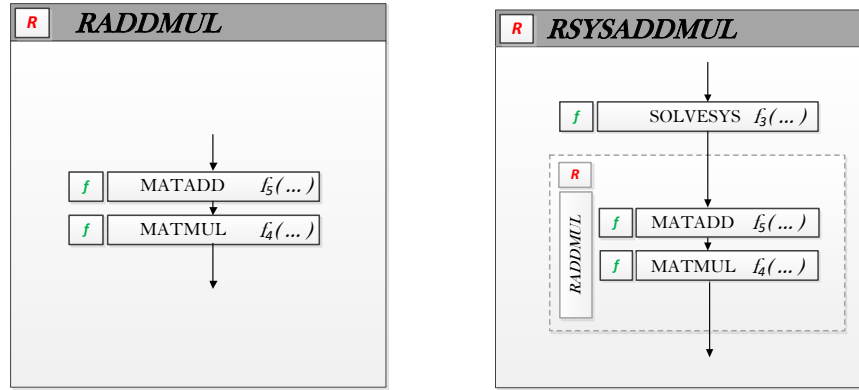
```
,
{
  routineId=31;
  routineName="RADDMUL";
  matrixComponentsList=(
    { ComponentType="F"; ComponentId=5; ComponentIterations=1;}
    ,
    { ComponentType="F"; ComponentId=4; ComponentIterations=1;}
  );
  matrixFunctionsList=(
    {
      functionId=5; routineId=31;
      subcontainer=0; container=0;
      functionIterations=1;
    }
    ,
    {
      functionId=4; routineId=31;
      subcontainer=1; container=1;
      functionIterations=1;
    }
  );
  ...
}
```

En dicho listado se pueden distinguir los siguientes campos:

- `routineId`: Un número entero que identifica de manera única a la rutina en el simulador. Es generado automáticamente cuando el usuario crea una nueva rutina.
- `routineName`: Una cadena alfanumérica que contiene el nombre dado por el usuario a la rutina.
- `matrixComponentsList`: Contiene una terna (o una lista de ellas) que representan los componentes que integran la rutina. Cada una contiene la siguiente información:

- `ComponentType`: El valor F o R según el componente sea una función u otra rutina, respectivamente.
 - `ComponentId`: Contiene el identificador de dicho componente, que podrá ser el `functionId` de una función, o el `routineId` de una rutina creada previamente.
 - `ComponentIterations`: Contiene el número de veces que se va a ejecutar este componente cuando el simulador calcula esta rutina. Esta funcionalidad es usada para simular procesos iterativos, como el de Newton-Raphson usado en la simulación de sistemas multicuerpo (sección 2.3.1).
- `matrixFunctionsList`: Esta sección contiene la lista de funciones que forman la rutina, una vez desglosadas las rutinas anidadas en sus componentes. Cada una de ellas se representa mediante la siguiente información:
- `functionId`: El identificador de la función (su `functionId`, descrito en la sección 5.2.1).
 - `routineId`: El identificador de la rutina al que pertenece la función.
 - `container`: Indica el orden que ocupa esta función en la lista de componentes de la rutina.
 - `subcontainer`: Cuando la función pertenece a una rutina anidada indica el lugar que ocupa en la secuencia de cálculos dentro de aquella. En caso contrario, como ocurre en este ejemplo, este campo no es relevante, y adopta el mismo valor que `container`.
 - `functionIterations`: Es el número de veces que se va a ejecutar esta función. Hereda su valor del campo `ComponentIterations`.

La figura 5.3(a) muestra una representación de la rutina simple a la que se hace referencia en el listado 5.3. La figura 5.3(b) muestra otra rutina, esta vez compuesta por una función (una resolución de un sistema de ecuaciones, `SOLVESYS`) y por una rutina anidada (en concreto la rutina `RADDMUL` del ejemplo anterior). Se observa cómo `RADDMUL` aporta las dos funciones que contiene: `MATADD` y `MATMUL`.



(a) Rutina compuesta por una suma y una multiplicación de matrices. El nombre RADDMUL es un acrónimo que hace referencia a R-utina, ADD-ition y MUL-tiplication.

(b) Rutina RSYSADDMUL compuesta por una función de resolución de sistemas de ecuaciones y por una rutina que incluye una suma y una multiplicación de matrices.

Figura 5.3: Rutinas de ejemplo usadas para ilustrar el modo en el que se representan en el fichero `routines.rou`.

El listado 5.4 corresponde a una sección del fichero `routines.rou` con la información relativa a la rutina anidada de la figura 5.3(b). En la sección `matrixComponentsList`, que muestra los componentes de dicha rutina, encontramos un primer elemento con `ComponentType="F"` y `ComponentId="3"`, que corresponde a la función `SOLVESYS`. El segundo elemento corresponde a la rutina `RADDMUL` y se representa con `ComponentType="R"` y `ComponentId="31"`. El desglose en funciones que se recoge en la sección `matrixFunctionsList` muestra las tres funciones que integran dicha función, la función "3" (`SOLVESYS`) aportada por la propia rutina, y las funciones "5" (`MATADD`) y "4" (`MATMUL`) aportadas por la rutina con Id "31" (`RADDMUL`). El orden de ejecución secuencial de las funciones definidas en una rutina, con rutinas anidadas o sin ellas, viene determinado por los parámetros `container` y `subcontainer`. En este ejemplo encontramos el `container` con valor "0" (la función, el primer componente) y el `container` con valor "1" (la rutina, el segundo componente). En el segundo contenedor el parámetro `subcontainer` toma los valores "0" y "1", indicando el orden en que se ejecutan sus funciones.

Listado 5.4 Extracto del fichero `routines.rou` que muestra la información relativa a la rutina `RSYSADDMUL` creada por el usuario y que está compuesta por la función de solución de un sistema de ecuaciones `SOLVESYS`, y de la rutina `RADDMUL`, compuesta a su vez de una suma, `MATADD`, y una multiplicación de matrices, `MATMUL`.

```
,
{
  routineId=3;
  routineName="Demo3";
  matrixComponentsList=(
    {ComponentType="F"; ComponentId=3; ComponentIterations=1;}
    ,
    {ComponentType="R"; ComponentId=31; ComponentIterations=1;}
  );
  matrixFunctionsList=(
    {functionId=3; routineId=3;
     subcontainer=0; container=0;
     functionIterations=1;}
    ,
    {functionId=5; routineId=31;
     subcontainer=0; container=1;
     functionIterations=1;}
    ,
    {functionId=4; routineId=31;
     subcontainer=1; container=1;
     functionIterations=1;}
  );
  ...
}
```

5.2.3 Modelos

Un Modelo en PARCSIM es la representación en forma de grafo dirigido del algoritmo de resolución de un determinado problema científico. En este grafo los nodos corresponden a bloques de instrucciones que resuelven determinados subsistemas que denominaremos Grupos, y las líneas dirigidas entre los nodos indican el orden en que se deben calcular. En la figura 5.4 se observa un modelo de ejemplo, compuesto por siete grupos. Dos de ellos (*Start* y *End*) no realizan cálculos, pero se incluyen para delimitar el inicio y el final del algoritmo. Los cinco restantes, cuyos nombres van desde *Gr*₁ hasta *Gr*₅, ejecutan rutinas compuestas por una o más funciones.

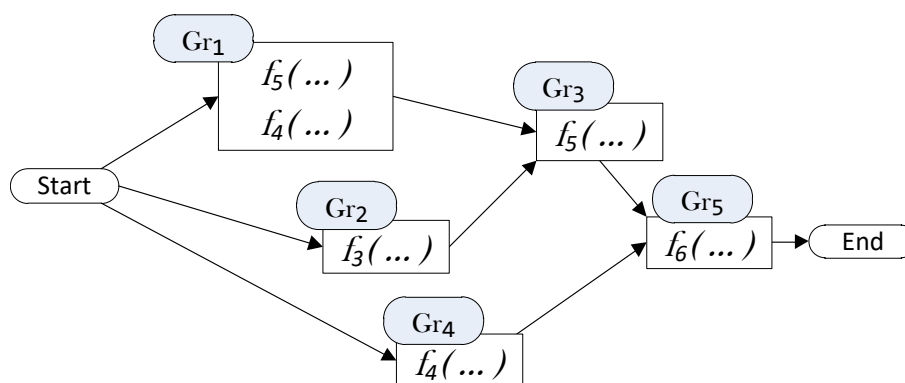


Figura 5.4: Modelo de ejemplo formado por siete grupos. En cada grupo se muestran las instrucciones o funciones, ya desglosadas, a partir de las rutinas que definen la solución del subsistema o grupo al que representan. Las líneas dirigidas indican las dependencias entre los grupos.

5.2.4 Grupos

En el simulador, los Grupos representan los subsistemas o módulos en los que se divide el sistema completo. Esta división la realiza el usuario como preproceso off-line (por lo que no interviene en el cómputo del tiempo de ejecución). También es el usuario el que define los algoritmos que se deben ejecutar para resolver cada uno de los grupos, así como las relaciones de dependencia entre ellos. En una representación en forma de grafo dirigido los grupos se corresponden con los nodos. La información de todos los grupos que forman un `Modelo` (sección 5.2.3) se almacena en un fichero de texto cuyo nombre se forma con el del modelo y con la extensión `.mdl`. En este archivo la información relativa a los grupos está recogida en la sección `modelGroupsList` (listado 5.5) donde, para cada grupo, se especifican los siguientes datos:

- `groupId`: Contiene un número entero generado automáticamente por el software en el momento de la creación del grupo y que sirve para identificarlo de manera única. Se comienza con el valor 1 para el primer grupo, y se asignan valores en orden secuencial conforme se va creando el resto de grupos, hasta llegar al último que tomará el valor correlativo más elevado.
- `groupName`: Es la cadena alfanumérica que contiene el nombre que el usuario asigna al grupo.

Listado 5.5 Extracto del fichero `ModeloEjemplo.mdl` que describe el modelo de la figura 5.4. Se muestra información de los grupos `Start` y `Gr1`, y las constantes `ModelRows` y `ModelCols`.

```
modelName="ModeloEjemplo";
modelId=4;
...
modelGroupsList=(
{
    groupName="Start ";
    groupId=1;
    groupNext=(2,3,4);
    groupisAModel=0;
    groupRoutine=-1;
    groupisAModelId=-1;
    matrixParameterList=({})
}
,
{
    groupName="Gr1 ";
    groupId=2;
    groupNext=(5);
    groupisAModel=0;
    groupRoutine=6;
    groupisAModelId=-1;
    matrixParameterList=(
        { matrixId=2; matrixName="A1";
          matrixRows="ModelRows"; matrixCols="1"; }
        ,
        { matrixId=1; matrixName="A2"
          matrixRows="1"; matrixCols="ModelCols"; }
        ,
        ...
    )
}
,
...
)
constants=(
    {name="ModelRows"; } ,
    {name="ModelCols"; } ,
    ...
);
```

- `groupNext`: Contiene un identificador (o lista de identificadores separados por comas) con los grupos que se calculan en secuencia después del grupo actual y que, por tanto, tienen dependencia temporal de este.
- `groupisAModel`: Toma el valor 0 si el grupo se resuelve mediante la ejecución de la rutina cuyo identificador se especifica en el parámetro `groupRoutine`. Un valor de 1 indica que el grupo se va a resolver ejecutando un segundo modelo embebido en el grupo. Esto representa una funcionalidad avanzada que se describe en el anexo (sección A.8.26.4).

- `groupRoutine`: Cuando `groupisAModel` es 0, este campo contiene el identificador de la rutina que resolverá el grupo. Como se describió en la sección 5.2.2, una rutina engloba una secuencia de cálculos (un algoritmo). Un valor -1 indica que no hay una rutina asignada a este grupo.
- `groupisAModelId`: Cuando el parámetro `groupisAModel` es 1, este campo contiene el identificador de un modelo que se inserta en el grupo. Dicho modelo estará formado a su vez por un conjunto de grupos, configurando de este modo un sistema de anidamiento de modelos. Para calcular el grupo será necesario la resolución del modelo insertado. Un valor -1 indica que este grupo no tiene asignado ningún modelo.
- `matrixParameterList`: Contiene un cuarteto (o una lista de ellos) que contiene información de los parámetros necesarios para ejecutar todas las funciones que integran la rutina ejecutada en este grupo. Dichos parámetros serán los argumentos de entrada y salida de las funciones. Dado que estas son operaciones de álgebra lineal, dichos parámetros tendrán formato matricial. Cada cuarteto contiene la siguiente información:
 - `matrixId`: Un identificador generado por el software en el momento en el que el usuario crea el parámetro.
 - `matrixName`: Una cadena alfanumérica que contiene el nombre dado por el usuario. Como se observa en el listado 5.5, el contenido de `matrixId` nos indica que primero se ha generado el parámetro A2 y después el A1. Esto no es relevante para el usuario pues el valor de `matrixId` tan solo se utiliza internamente para identificar de manera única a los parámetros durante la ejecución de la simulación.
 - `matrixRows`: Contiene el nombre de una constante que contendrá el número de filas de la matriz.
 - `matrixCols`: Contiene el nombre de una constante que contendrá el número de columnas de la matriz.Estas constantes, usadas para indicar las filas y columnas de las matrices usadas durante la simulación, deberán estar recogidas en el área `constants` de este mismo fichero.

5.2.5 Parámetros

Como se describió en la sección 5.2.2, las rutinas que resuelven subproblemas resultantes de la división de un problema complejo pueden estar compuestas por un número cualquiera de funciones y rutinas anidadas. Las funciones en PARCSIM, como se vio en 5.2.1, necesitan parámetros, que son los argumentos de entrada para realizar los cálculos y actualizar en consecuencia los de salida. Por tanto, una vez que se asigna una rutina a un grupo, es necesario indicar también qué parámetros o argumentos necesitan recibir o devolver sus funciones. En el simulador, el conjunto de parámetros de un modelo se representa mediante variables que el usuario define en tiempo de diseño y que representan a todas las matrices que se van a utilizar durante la simulación. La figura 5.5 muestra la relación entre los conceptos mencionados mediante un caso de ejemplo.

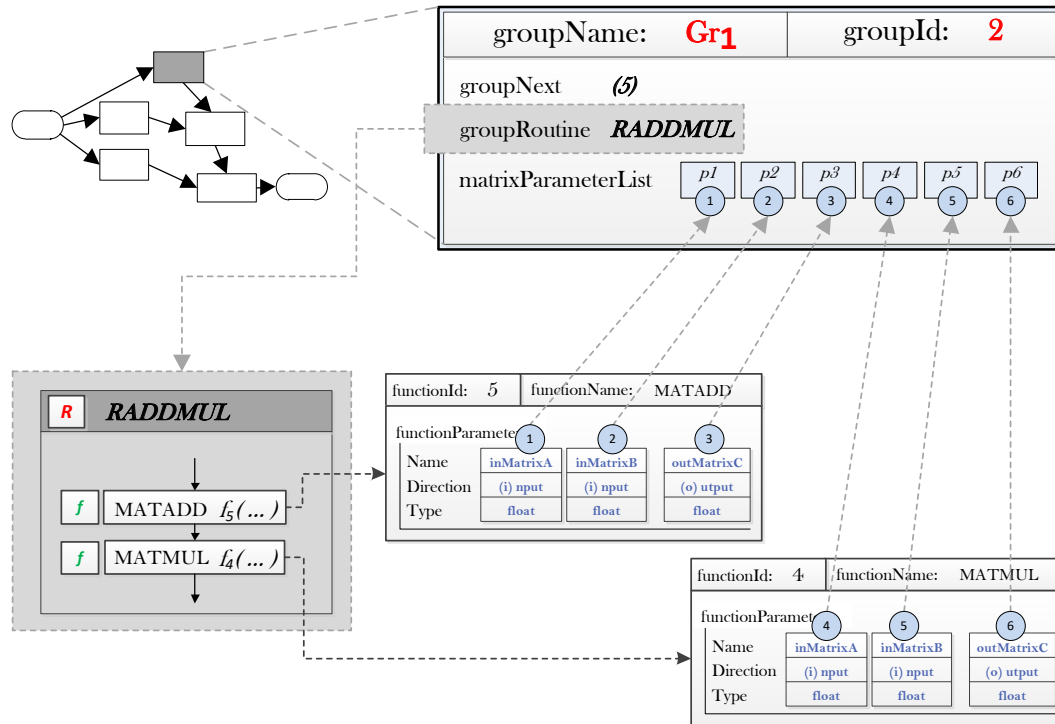


Figura 5.5: Esquema de la relación grupo-rutina-parámetros. El grupo **Gr1** ejecuta la rutina **RADDMUL** que contiene dos funciones. Cada función requiere tres parámetros, por lo que **Gr1**, por incluir esta rutina, debe aportar los seis parámetros $p1, \dots, p6$.

En la figura vemos una representación del grupo Gr_1 que forma parte del modelo que se describió en la figura 5.4. Dicho grupo ejecuta la rutina RADDMUL compuesta por una función suma MATADD y una función multiplicación MATMUL. Cada una de estas funciones requiere dos parámetros de entrada y uno de salida. Como consecuencia, el usuario debe asignar a RADDMUL, o bien seis parámetros a Gr_1 , o bien modificar la definición de la rutina para establecer una relación entre los parámetros de entrada de una función y el parámetro de salida de alguna otra, permitiendo de esta manera la comunicación de información entre funciones. Este método se describe en la sección A.8.26.1 del anexo.

5.2.6 Escenarios

Este término hace referencia a la naturaleza del problema a resolver, que quedará especificado mediante el tamaño y formato de las variables que se usarán como parámetros en las funciones de cálculo que componen las rutinas. Tal como se mencionó en el apartado 5.2.5, las variables tienen formato matricial, por lo que es necesario especificar sus dimensiones, tipo y factor de dispersión. Los tipos de matrices implementados en el simulador se muestran en la tabla 5.3. Las matrices banda simétricas son el formato que habitualmente se obtiene al modelar las ecuaciones de grupo que permiten resolver la cinemática de sistemas mecánicos. El factor de dispersión es un número entero que representa el porcentaje de valores nulos respecto al tamaño de la matriz.

valor	descripción
0	matriz no simétrica
1	matriz banda (con valores no nulos cercanos a la diagonal)
2	matriz simétrica con valores no nulos distribuidos aleatoriamente

Tabla 5.3: Tipos de matrices manejados por el simulador.

Un usuario puede crear tantos escenarios como desee, representando de esta manera tamaños de problema diferentes. El simulador resolverá el modelo con cada escenario en ejecuciones diferentes. Cada escenario, con la información de todas las variables que usa (y que por tanto deben haber sido creadas previamente), se guarda en un fichero de texto con la extensión `.sce` y con un nombre

que se compone con el del modelo, seguido de la cadena “SCENARIO” y de una descripción del escenario que el usuario asigna en el momento de su creación. El listado 5.6 muestra un ejemplo del formato de este tipo de fichero (por simplicidad se muestran únicamente dos variables).

Listado 5.6 Extracto del fichero que describe uno de los escenarios que se emplearán para simular el modelo de la figura 5.4. Se muestra información de las dos variables que corresponden a las matrices con identificadores 1 y 2.

```
...
,
scenarioList=(
{
  scenarioName="Escenario20 ";
  scenarioModelName="ModeloEjemplo ";
  scenarioModelId=4;
  scenarioMatricesList=(
    {
      scenarioMatrixId=1;
      scenarioMatrixRows=20;
      scenarioMatrixCols=20;
      scenarioMatrixType=1;
      scenarioMatrixSparsity=10;
      scenarioCreateRandom=1;
      matrixFileLocation="";
      scenarioreuseIfExist=0;
    },
    {
      scenarioMatrixId=2;
      scenarioMatrixRows=20;
      scenarioMatrixCols=20;
      scenarioMatrixType=1;
      scenarioMatrixSparsity=10;
      scenarioCreateRandom=1;
      matrixFileLocation="";
      scenarioreuseIfExist=0;
    }
  ),
},
...
,
```

En el listado podemos distinguir la siguiente información:

- **scenarioName:** Una cadena alfanumérica que almacena el nombre que el usuario asigna al nuevo escenario.
- **scenarioModelName:** Contiene el nombre del modelo al que pertenece el escenario.
- **scenarioModelId:** Contiene el identificador, generado por el simulador, del modelo al que pertenece el escenario.

- `scenarioMatricesList`: Almacena, mediante una lista de octetos, los detalles de formato y contenido de las variables que se usarán durante la simulación del modelo. Los campos son los siguientes:
 - `scenarioMatrixId`: Es un número entero que se genera de manera automática cuando el usuario crea una variable. En el listado 5.5, que contiene la configuración del modelo de ejemplo, podemos encontrar en el área de nombre `matrixParameterList`, el par `matrixId=2`, `matrixName="A1"`, que refleja la correspondencia entre el nombre y el identificador `scenarioMatrixId`.
 - `scenarioMatrixRows`: Contiene el número de filas de la matriz.
 - `scenarioMatrixCols`: Contiene el número de columnas de la matriz.
 - `scenarioMatrixType`: Contiene el tipo de matriz, cuyos valores posibles se muestran en la tabla 5.3.
 - `scenarioMatrixSparsity`: Es un valor de tipo entero que indica el porcentaje de valores nulos respecto al tamaño total de la matriz.
 - `scenarioCreateRandom`: Cuando este campo toma el valor 1, al inicio de la simulación el software crea la matriz con valores aleatorios. Si toma el valor 0, el sistema busca en el directorio de ejecución un fichero que contenga una matriz del mismo tamaño (filas y columnas), tipo y factor de dispersión especificados. En caso de no encontrarla, la crea con valores aleatorios.
 - `matrixFileLocation`: Si este campo contiene el nombre y ruta de un fichero, el simulador extraerá de dicho fichero los datos de la matriz.
 - `scenarioreuseIfExist`: Si toma el valor 1, el simulador busca en el directorio de ejecución un fichero que contenga una matriz del mismo tamaño (filas y columnas), tipo y factor de dispersión. En caso de no encontrarla, la crea con valores aleatorios.

5.2.7 Scripts

PARCSIM denomina *Script* a un conjunto de valores de los parámetros algorítmicos (*AP*) que se aplican durante la simulación de un modelo, en concreto, el número de threads generados, la cantidad de GPUs que se utilizan y la librería de cómputo empleada. Como se vio en la sección 4.2.3.4 la naturaleza de dichos *AP* determinará las estrategias de búsqueda de sus valores óptimos. En nuestro simulador, los valores de los parámetros se pueden especificar de dos formas: mediante una listas de valores individuales o como rangos. Si se especifican rangos, el simulador expande los valores individuales. Para limitar el número de combinaciones resultante, y por tanto el espacio de búsqueda, el usuario puede crear una fórmula que exprese una condición que relacione los valores de los parámetros. En caso de no cumplirse, dicha combinación de parámetros se descarta. Por ejemplo, en un sistema hardware multicore el usuario del simulador puede decidir imponer la restricción de que el número combinado de threads del primer y segundo nivel de paralelismo no excedan el número de cores de la CPU.

Se pueden crear varios scripts, que se guardan en ficheros de texto con la extensión `.scp` cuyo nombre se compone de la cadena “SCRIPT_” seguida de una descripción que el usuario asigna al script en el momento de crearlo. El listado 5.7 muestra un ejemplo del formato de este tipo de ficheros.

Listado 5.7 Sección del fichero `SCRIPT_ScriptDemo.scp` que describe uno de los scripts que servirá para generar los parámetros algorítmicos durante la simulación del modelo de la figura 5.4.

```
scriptName="ScriptDemo ";
scriptId=1;
scriptActive=1;
scriptTraining=0;
parameterList=(
    {parameterName="OMP1";    parameterValueRange=("1","4","1");}
    ,
    {parameterName="OMP2";    parameterValue=("1","3");}
    ,
    {parameterName="LOOP";    parameterValue=("1");}
    ,
    {parameterName="LIBRARY"; parameterValue=("1");}
    ,
    {parameterName="GPU";     parameterValue=("0");}
    ,
    ...
)
```

La información se almacena mediante los siguientes campos:

- `scriptName`: Es una cadena alfanumérica que contiene el nombre que el usuario asigna al script en el momento de crearlo.
- `scriptId`: Un número entero generado automáticamente por el sistema que identifica de manera única a un script dentro del simulador.
- `scriptActive`: Este campo puede tomar los valores 0 o 1. Cuando es 1 el script se considera activo, y el simulador hará uso de él. Esta opción permite al usuario crear un conjunto de scripts, cada uno de los cuales puede contener valores de los parámetros algorítmicos de acuerdo a un determinado hardware, y, de esta manera, poder activar unos u otros en función de la plataforma en la que va a realizar la simulación.
- `scriptTraining`: Este campo puede tomar los valores 0 o 1. Si el valor es 1 indica que el script será usado únicamente en ejecuciones cuyo objetivo sea el entrenamiento del sistema. Este valor es asignado de manera automática por el software cuando se crea un script de entrenamiento.
- `parameterList`: Contiene la lista de los parámetros algorítmicos y sus valores. Cada parámetro se identifica en el fichero como sigue:
 - `parameterName`: El nombre del parámetro asignado por el usuario.
 - `parameterValueRange`: Es una terna de números enteros que indica un rango de valores del parámetro. El primer número indica el valor inferior; el segundo número indica el valor superior; el tercer número indica el valor de salto.
 - `parameterValue`: Este campo indica un valor, o una lista de valores individuales de un parámetro, separados por comas. Los campos `parameterValueRange` y `parameterValue` son excluyentes ya que representan diferentes maneras de indicar los valores de los parámetros. Por ejemplo, en el listado 5.7 observamos que para el parámetro `OMP1` el campo `parameterValueRange` contiene un rango que va desde 1 hasta 4 con saltos de 1. Sin embargo, para el parámetro `LIBRARY` se indica únicamente el identificador 1, que corresponde a MKL, como refleja la tabla 5.1.

5.2.8 Resumen

La figura 5.6 muestra, mediante un diagrama, los conceptos introducidos en esta sección y la relación entre los mismos.

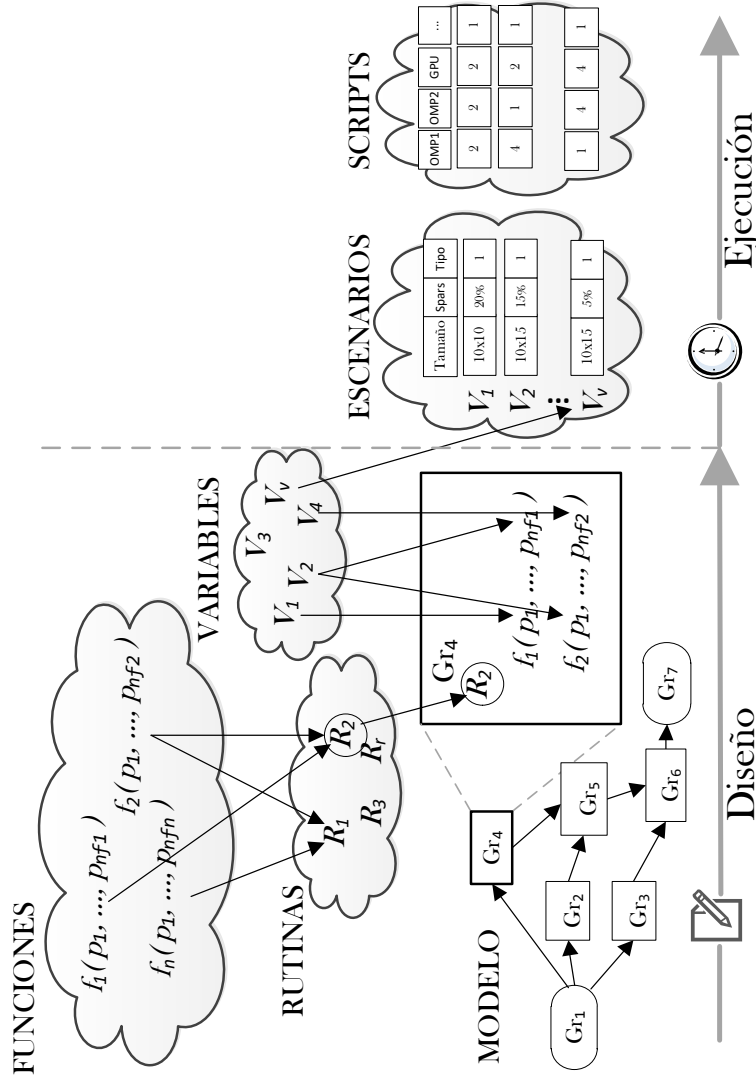


Figura 5.6: Vista esquemática de los conceptos usados en el simulador y sus interdependencias.

Inicialmente se parte de un conjunto de funciones incorporadas en el simulador. El usuario puede seleccionar una o varias de dichas funciones para crear sus propias rutinas, que representan una secuencia de operaciones que calculan un

determinado algoritmo. Los parámetros de las funciones, argumentos de entrada y salida de las mismas, se toman de una lista de variables creadas por el usuario y que representan a todas las matrices que han surgido del proceso de modelado del sistema real a simular. El modelo generado estará formado por una serie de grupos y sus dependencias, cada uno de los cuales tendrá asignada una rutina con los cálculos que resuelve dicho grupo. De cara a la ejecución será necesario, por una parte, indicar la dimensión y el formato de las matrices, lo que se denomina escenario y, por otra, el conjunto de parámetros algorítmicos que el software aplicará durante la simulación, denominado script.

5.3 Rutas

Como se vio en el apartado 5.1, los problemas numéricos resolubles empleando una estrategia de descomposición en subproblemas se pueden representar mediante grafos acíclicos dirigidos. Un grafo de este tipo se muestra en la figura 5.7, en el que se calculan siete grupos para resolver un sistema completo.

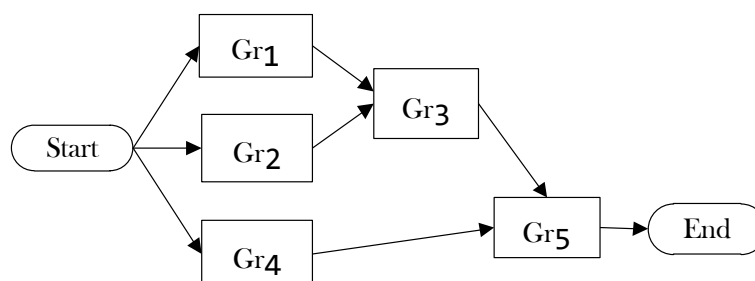


Figura 5.7: Grafo de resolución de un problema numérico mediante descomposición en subproblemas.

En este ejemplo encontramos una dependencia en el grupo *End*, el cual requiere la resolución del *Gr5*, que a su vez necesita los resultados obtenidos al calcular los grupos *Gr3* y *Gr4*. En este mismo grafo se puede observar que entre los grupos *Gr1*, *Gr2* y *Gr4* no hay ninguna dependencia. Esto significa que dichos grupos podrían resolverse de manera simultánea aplicando algún tipo de paralelismo. El cronograma mostrado en la figura 5.8 incluye en su etapa e_2 la resolución simultánea de dichos grupos.

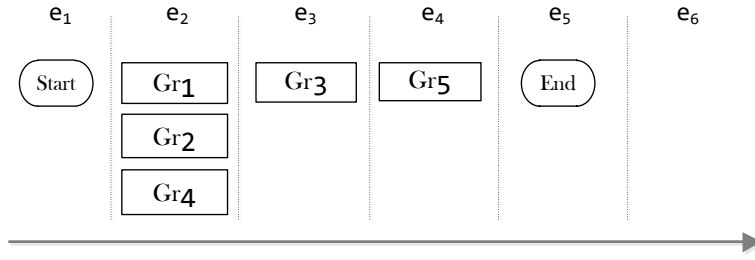


Figura 5.8: Cronograma de resolución del problema representado en la figura 5.7. Cada e_i representa una de las etapas en las que se divide la ejecución.

Ahora bien, la ejecución paralela puede no resultar la más eficiente, lo que dependerá de los cálculos a realizar, de los recursos del hardware disponibles y del tipo de librería de cómputo empleada. Por ejemplo, en una CPU que disponga de dos cores, la resolución simultánea de tres grupos puede obtener peores tiempos de ejecución que los conseguidos si uno de los grupos se calcula en una etapa posterior, siempre y cuando las dependencias entre grupos lo permitan (figura 5.9).

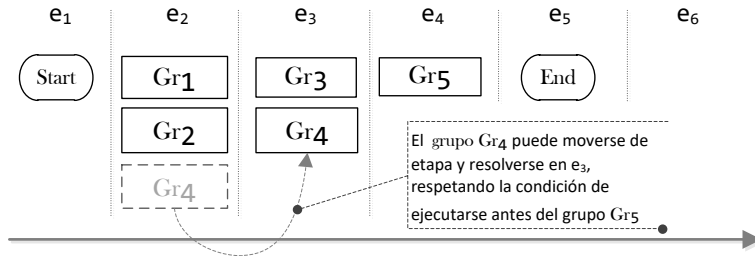


Figura 5.9: Cronograma de resolución del problema representado en la figura 5.7 retrasando la ejecución del grupo Gr_4 para resolverlo en paralelo con el Gr_3 . Cada e_i representa una de las etapas en las que se divide la ejecución.

Por tanto, para encontrar el algoritmo óptimo es necesario conocer previamente todas las alternativas de ordenación de los cálculos. El árbol representado en la figura 5.10 muestra todas las posibilidades que existen para resolver el problema del ejemplo de la figura 5.7. En dicho grafo se eliminan ramas que dan lugar a agrupaciones cuya ejecución es una permutación de otra ya generada.

En PARCSIM cada una de estas alternativas de ordenación y agrupación de los cálculos se denomina Ruta. Por ejemplo, se observa que la ruta 7 propone la ejecución en paralelo de los grupos de cálculos Gr_1 , Gr_2 y Gr_4 para, a continuación resolver los grupos Gr_3 , Gr_5 y End de manera secuencial.

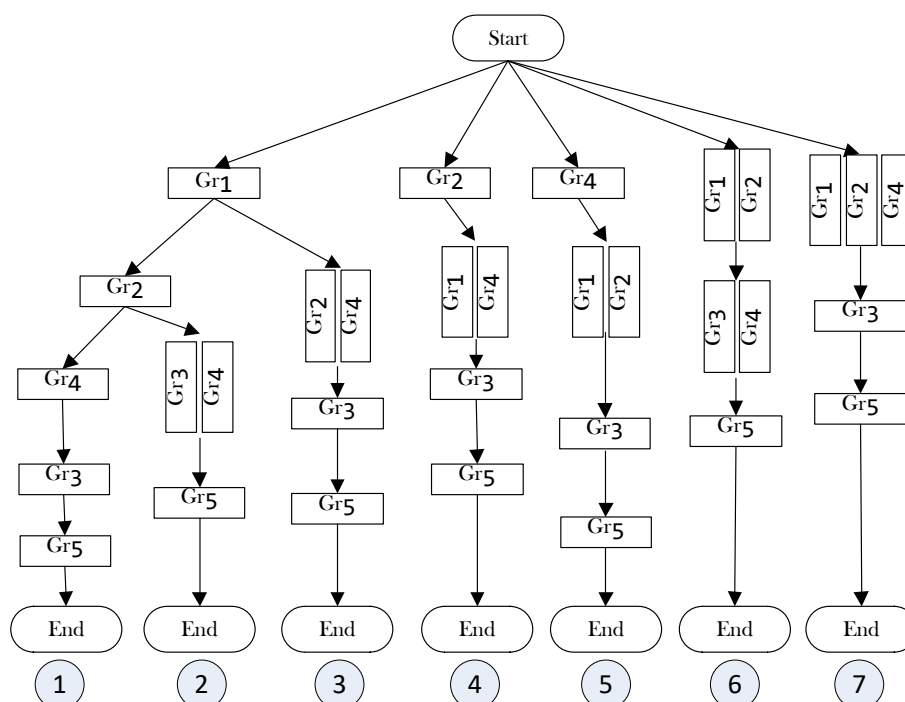


Figura 5.10: Árbol que muestra las secuencias de cálculo válidas para el problema de la figura 5.7. Cada rama es una ruta que representa un modo de ordenar los cálculos capaz de resolver el sistema en su totalidad. Los nodos del árbol que contienen más de un grupo suponen la resolución simultánea de dichos grupos.

Una vez creado el árbol de rutas, su estructura se recoge en una nueva sección, `branches`, en el mismo fichero que almacena los grupos descritos en 5.2.4. El listado 5.8 muestra la estructura de dicha sección.

Listado 5.8 Sección del fichero `ModeloEjemplo.mdl` que describe el árbol de rutas de la figura 5.10. Se muestra información de la ruta identificada como 5. `TRANS`, `ADD_SYS`, `LINSYS` y `ADD` son ejemplos de rutinas creadas por el usuario y asignadas a los grupos correspondientes.

```
branches=(
{
  branchpath="1-4-2+3-5-6-7";
  branchpathroutines="?-TRANS-ADD_SYS+LINSYS-ADD-LINSYS-?";
  branchpathnames="( Start ) ( Gr4 ) ( Gr1+Gr2 ) ( Gr3 ) ( Gr5 ) ( End ) ";
  branchid=36;
  numsteps=6;
  nodeslist="0-3-13-22-29-36";
  numnodes=38;
}
,
```

- **branchpath:** Es una cadena de texto que contiene los identificadores (`groupId`) de los grupos que forman esta ruta, su orden y agrupaciones. Para ello se emplea la siguiente codificación:
 - El símbolo `+` separa los códigos de los grupos que se calculan en paralelo en una misma etapa de tiempo.
 - El símbolo `-` separa etapas de tiempo.

La figura 5.11 muestra la codificación para la ruta 5.

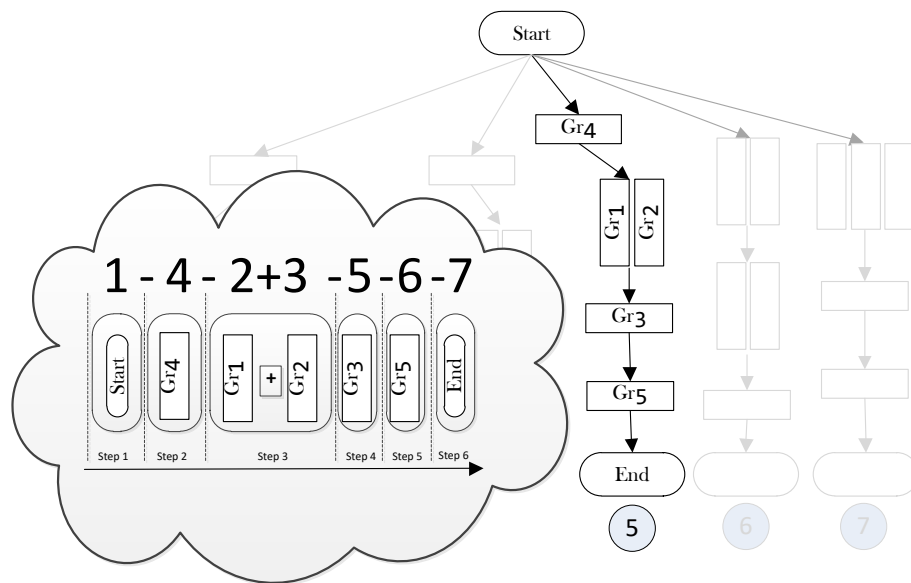


Figura 5.11: Ejemplo de codificación de los grupos en una ruta en PARCSIM. El símbolo `+` separa los identificadores de los grupos (`groupId`) que se calculan en paralelo en una misma etapa de tiempo, y el símbolo `-` separa etapas de tiempo. La imagen muestra la ruta 5 del árbol mostrado en la figura 5.10.

- **branchpathroutines:** Contiene una copia de la cadena de texto almacenada en **branchpath** donde se han sustituido los códigos de los grupos por los nombres de las rutinas que se ejecutan en cada uno, usando el símbolo `"?"` cuando un grupo no contiene ninguna rutina.
- **branchpathnames:** Al igual que **branchpath**, es una cadena de texto que identifica a los grupos contenidos en la ruta, pero en esta ocasión identificados por su nombre, y encerrados entre paréntesis los que se resuelven de manera simultánea.

- `branchid`: Un identificador único que el software asigna de manera automática a la ruta.
- `numsteps`: Contiene el número de etapas necesarias para resolver el problema siguiendo esta ruta.
- `nodeslist`: Almacena una cadena donde aparecen los identificadores de los nodos que forman esta ruta, separados por el signo `—`. PARCSCIM asigna automáticamente un identificador a cada nodo del árbol.
- `numnodes`: Contiene el número total de nodos que forman el árbol.

5.4 Base de datos

Como se ha descrito en las secciones precedentes, el simulador usa ficheros de texto para almacenar la información relativa a la representación del problema a resolver. Sin embargo los datos obtenidos durante la simulación se almacenan en una base de datos relacional SQLite (sección 3.2.3). Dicha base de datos utiliza un archivo de nombre `PARCSIM.db` que contiene una tabla `Results` donde se generan registros para guardar los tiempos de ejecución obtenidos al simular los modelos. Los campos de dicha tabla se detallan en la imagen 5.12, y se pueden agrupar en seis bloques:

- **Heading**: Aquí se encuentran los campos que almacena el identificador que el motor de base de datos asigna automáticamente a cada registro, el modo de simulación empleado, el nombre del hardware y las fechas y horas de inicio y fin de dicha simulación.
- **Model**: Engloba los campos que permiten identificar el modelo y la rama que se ha simulado.
- **Scenario**: Contiene los campos que describen el escenario (el tamaño, tipo y dispersión de las matrices).
- **Algorithmic Parameters**: Campos que almacenan los números de threads, el número de GPUs y la librería utilizada.

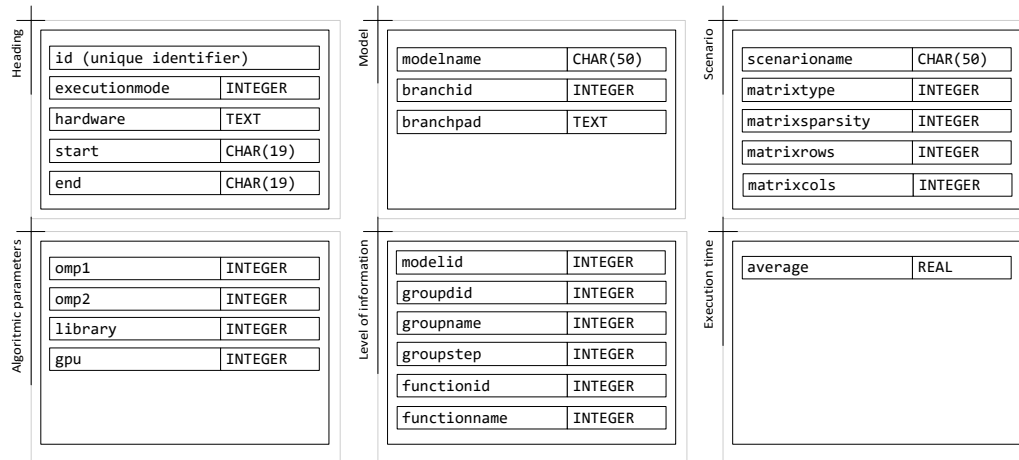


Figura 5.12: Estructura de la tabla Results contenida en la base de datos PARCSIM.db para la persistencia de los tiempos de ejecución obtenidos durante las simulaciones.

- Level of information: Estos campos indican el nivel en el que se han realizado las mediciones. Los tiempos de ejecución se pueden tomar de un modelo completo, un grupo o una función. La tabla 5.4 muestra el modo en el que se codifica dicha información. Cuando el tiempo de ejecución representa la simulación de un modelo completo se rellena únicamente el campo modelId con el identificador del modelo. Para la información relativa a un grupo, además del modelId, se introduce el identificador del grupo en groupId. Para representar los tiempos a nivel de función se completa también el campo functionId. En el caso particular de que la información haga referencia a un tiempo obtenido en el modo de entrenamiento solo se registra el código de la función en functionId.

modelid	groupid	functionid	Tiempo de ejecución que almacena
m	—	—	Modelo <i>m</i> completo
m	n	—	Grupo <i>n</i> en el modelo <i>m</i>
m	n	f	Función <i>f</i> en el grupo <i>n</i> del modelo <i>m</i>
—	—	f	Función <i>f</i> (modo de entrenamiento)

Tabla 5.4: Regla para determinar mediante los campos modelId, groupId y functionId en la base de datos PARCSIM.db el nivel (modelo, grupo o función) en el que se ha medido el tiempo de ejecución almacenado en un registro.

- Execution time: Almacena el tiempo de ejecución en segundos.

5.5 Árbol de rutas

Dado un problema científico concreto, uno de los objetivos del simulador es encontrar la ordenación de los cálculos que permite la resolución del modelo que lo representa en el menor tiempo posible. Como hemos estudiado en las secciones precedentes, una vez identificadas las funciones que se deben ejecutar, la elaboración de un grafo acíclico de los bloques de cálculos ayuda a encontrar los que son independientes entre sí y que, por tanto, pueden calcularse de manera simultánea. Con esa información el simulador puede componer un árbol donde cada rama representa un modo diferente de resolver el modelo y que sirve como base para determinar la mejor combinación de:

- Librerías: En el caso de que se disponga de más de una librería para la resolución de un determinado cálculo, la elección de la más eficiente en términos de rendimiento dependerá de varios factores:
 - Del tamaño del problema: En problemas matriciales, las dimensiones de las matrices.
 - De la naturaleza de los datos: Se refiere a la dispersión, simetría de la matrices, y el tipo numérico.
 - Del número de threads: Cuando las librerías implementan algoritmos paralelos en sus códigos, la asignación de threads a sus rutinas internas permite explotar arquitecturas hardware multicore o multicore con GPUs.
- Paralelismo: Incluso en el caso de que se identifique la posibilidad de realizar cálculos de manera simultánea, será necesario encontrar un *scheduling* adecuado, tarea nada trivial que debe tener en cuenta la librería de cómputo, los recursos hardware disponibles y la naturaleza de los cálculos de cada grupo. Con carácter general, el tiempo total necesario para calcular varios grupos en paralelo va a estar marcado por el tiempo de ejecución requerido por el de mayor coste computacional. Por ejemplo, en la figura 5.13 se representa la ejecución simultánea de tres grupos mediante la asignación de un thread a cada grupo. En este caso, los grupos Gr_1 y Gr_2 deben esperar a la finalización de Gr_3 , que consume cuatro unidades de tiempo.

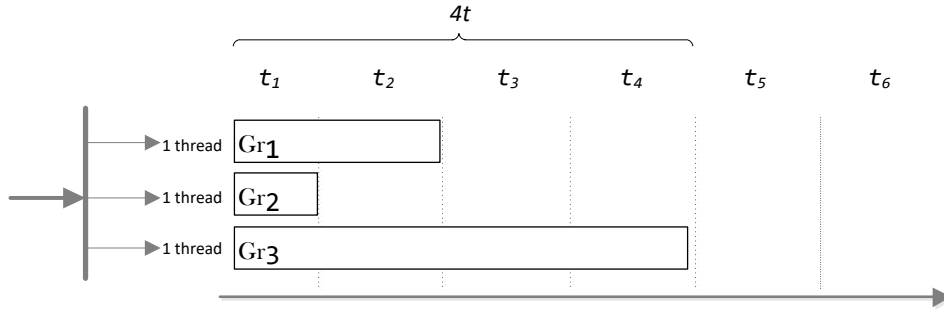


Figura 5.13: Resolución simultánea de tres grupos con diferente carga computacional asignando un thread a cada grupo. El tiempo total de ejecución corresponde al retraso provocado por el grupo Gr_3 , de cuatro unidades de tiempo.

Se puede plantear estrategia de *scheduling* diferente, como la mostrada en la figura 5.14, dividiendo los cálculos en varias etapas y aislando el grupo más costoso para poder asignarle una mayor cantidad de recursos y explotar el paralelismo implícito que puedan implementar las funciones de la librería. De este modo el grupo Gr_3 ya no se calcula junto a Gr_1 y Gr_2 . Esto permite asignar dos threads a Gr_1 , consiguiendo una reducción del tiempo de ejecución de dicho grupo en una unidad de tiempo. Igual ocurre con Gr_3 , donde, al poder asignarle los tres threads, se consigue una reducción de dos unidades de tiempo. La mejora global obtenida con esta reordenación de los cálculos, considerando el uso de librerías que admitan paralelismo implícito, es de una unidad de tiempo.

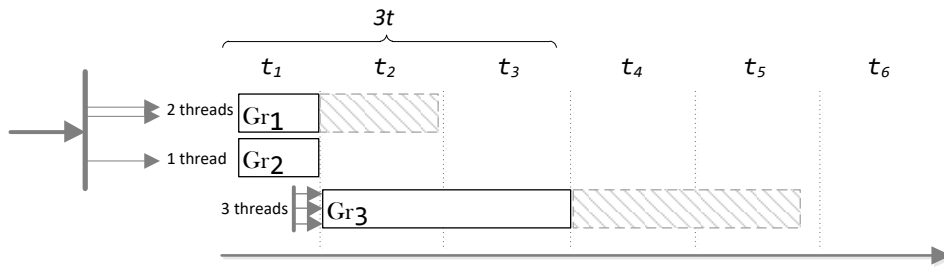


Figura 5.14: Modificación de la estrategia de scheduling respecto a la mostrada en la figura 5.13. La ejecución aislada en otra etapa del grupo Gr_3 permite asignar un mayor número de threads a Gr_1 y a Gr_3 , consiguiendo mejoras en los tiempos de ejecución de ambos grupos, y una mejora global de una unidad de tiempo.

5.5.1 Propiedades del árbol de rutas definido en el simulador

La elaboración de un árbol de rutas constituye el primer paso para la búsqueda automatizada de la versión del algoritmo que mejor se adapta a una plataforma hardware determinada. El simulador desarrollado en esta tesis crea árboles de rutas con las siguientes características:

1. Cada rama está compuesta por una serie de nodos ordenados, donde cada uno representa la resolución de uno o más grupos. El orden en el que se resuelven los grupos debe ser tal que se respete el orden de precedencia que refleja el grafo del sistema a resolver.
2. El árbol contiene solo una rama por cada agrupación de rutinas y tamaño de matrices. Ilustraremos esta propiedad mediante el modelo representado en la figura 5.15 que contiene tres grupos que ejecutan sumas de matrices.

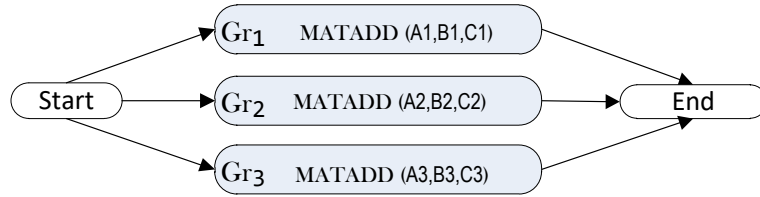


Figura 5.15: Modelo con cinco grupos, donde Gr_1 , Gr_2 y Gr_3 contienen la misma rutina para resolver una suma de matrices.

En un escenario en el que las matrices manejadas en cada grupo tienen tamaños diferentes, obtenemos un árbol de rutas como el mostrado en la figura 5.16. Cada rama del árbol, a pesar de resolver un total de tres sumas, tiene un coste computacional distinto debido al diferente tamaño de las matrices.

Pero si el grupo Gr_2 manipula matrices de un tamaño diferente al resto, entonces el árbol obtenido sería el de la figura 5.17. En este escenario se puede eliminar la rama 4, ya que se considera que tiene un tiempo de ejecución equivalente al de la rama 2 porque los nodos de ambas realizan los mismos cálculos sobre el mismo tipo de matrices.

Un último escenario posible en este modelo es aquel en el que las matrices son del mismo tamaño en todos los grupos. En este caso es posible eliminar

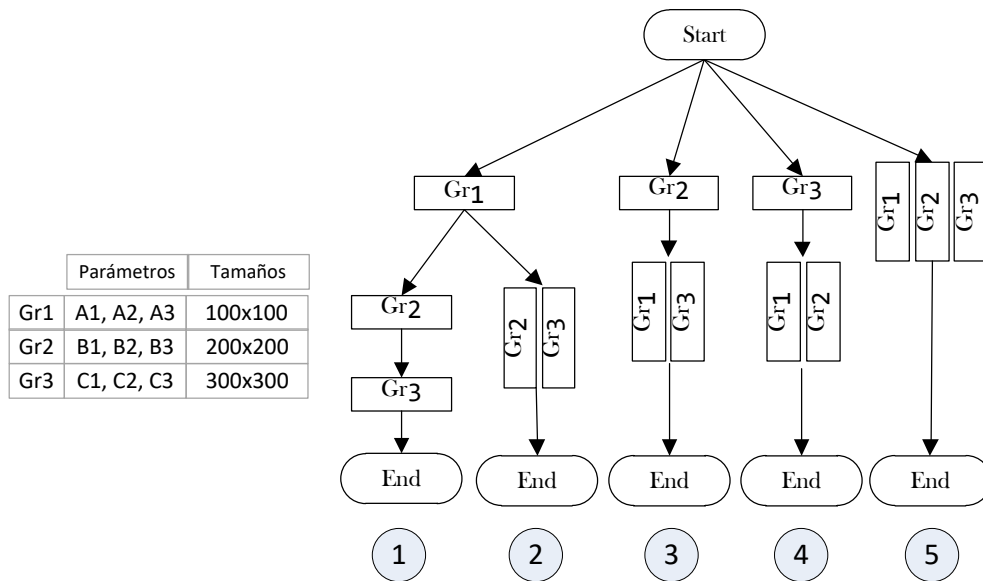


Figura 5.16: Árbol de rutas que permiten resolver el modelo de la figura 5.15 cuando los grupos Gr_1 , Gr_2 y Gr_3 operan sobre matrices de diferentes tamaños, de 100×100 , 200×200 y 300×300 .

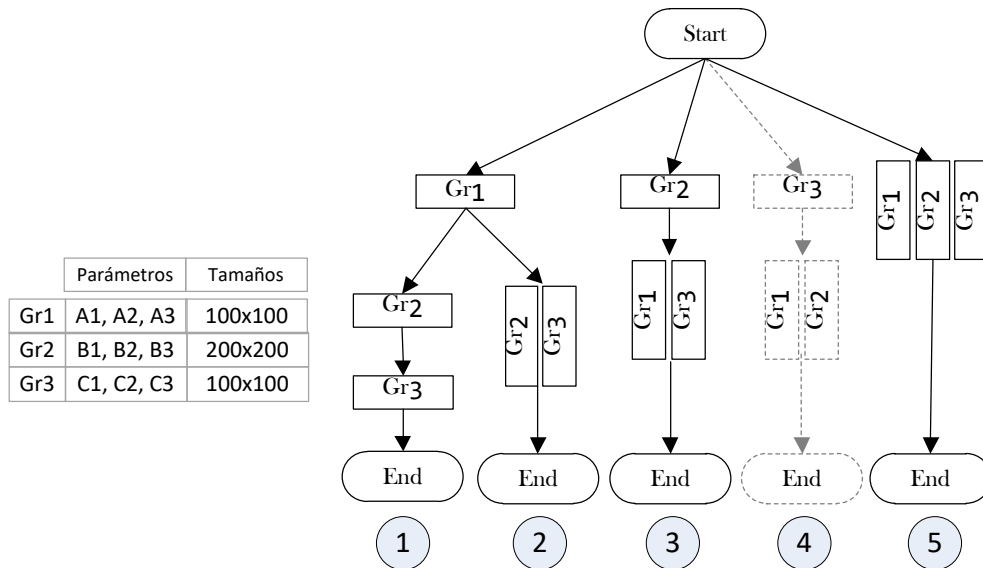


Figura 5.17: Árbol de rutas que permiten resolver el modelo de la figura 5.15 en un escenario en el que los grupos Gr_1 y Gr_3 operan sobre matrices de tamaños 100×100 , mientras que el Gr_2 lo hace sobre matrices 200×200 . La rama 4 puede eliminarse, pues es equivalente a la rama 2.

también la rama 3, como se muestra en la figura 5.18. El motivo es que las ramas 3 y 4 se consideran equivalentes a la 2 pues sus nodos ejecutan los mismos cálculos con matrices del mismo tamaño, por lo que los tiempos de ejecución serán los mismos.

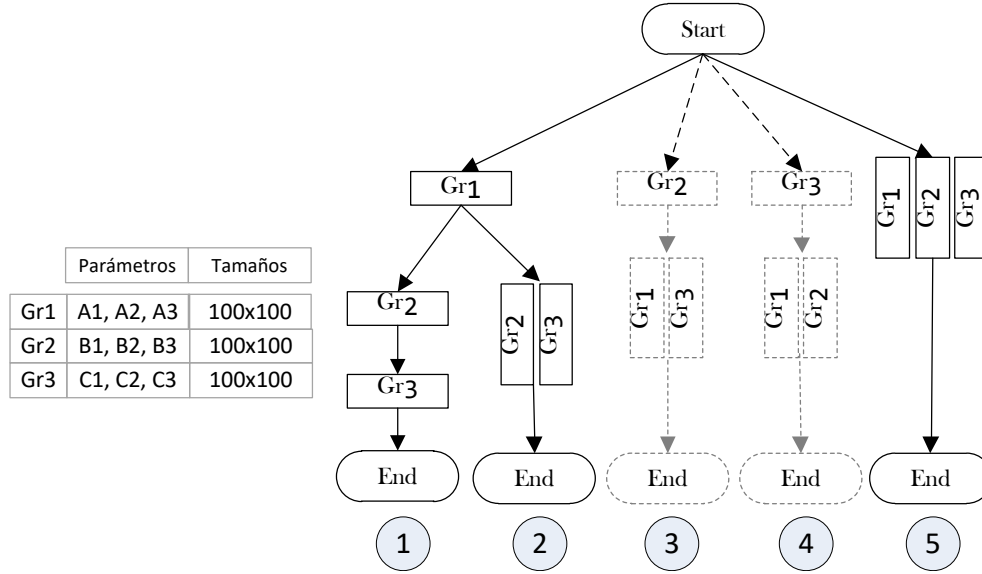


Figura 5.18: Árbol de rutas que permiten resolver el modelo de la figura 5.15 en un escenario en el que los grupos Gr_1 , Gr_2 y Gr_3 operan sobre matrices de tamaños 100×100 . Las ramas 3 y 4 pueden eliminarse por ser equivalentes a la rama 2.

3. Por último, en un árbol de rutas generado por el simulador, dos ramas cuyos nodos muestran las mismas agrupaciones de cálculos, pero en diferente orden, se considera que requieren el mismo tiempo de ejecución para su resolución. En este caso solo una de ellas será incluida en el árbol.

5.5.2 Generación del árbol de rutas

El algoritmo de creación del árbol de rutas implementado en el simulador es un proceso iterativo que queda descrito mediante el algoritmo 3. En cada etapa se selecciona un nodo del árbol que no tenga hijos y se le intentan agregar nodos que resuelven las combinaciones de grupos que aún no han sido resueltos en dicha rama.

Algoritmo 3: Algoritmo para crear el árbol de rutas de resolución de un determinado sistema mediante un proceso iterativo en el que se van añadiendo nodos hijo en cada rama.

```

1 GENERATE_TREES(MODEL, GROUPS[])
   Result: NODES[]
2 NODES[] = [];
3 NODES[] += root node;
4 number_of_nodes = 1;
5 while elapsed_time < maxTimeToGenerateTrees
6 and number_of_nodes < maxNumNodesInTrees do
7     for each node inside NODES[] do
8         if node has no children then
9             NV[: nodes in the same branch than node;
10            remaining[: groups not included in NV;
11            remaining[] = TRUNCATE_REMAINING(node, MODEL,
12            GROUPS[], NODES[], remaining[]);
13            n: number of groups in remaining;
14            for size ← 1 to n do
15                newNodes[] = CONSTRUCT_TREE(MODEL, GROUPS[],
16                NODES[], NV[], remaining[], size);
17                NODES[] += newNodes[];
18                number_of_nodes += size of newNodes[];
19            end
20        end
21    end
22 end

```

Describiremos el detalle de dicho proceso mediante un ejemplo práctico aplicado al modelo de la figura 5.7.

El procedimiento actúa como se describe a continuación:

- Paso 1: Se crea el nodo raíz para contener al grupo *Start*, que es el inicial en el modelo (línea 3 del algoritmo 3).
- Paso 2: Se selecciona un nodo sin hijos (línea 8), que en esta etapa inicial solo puede ser el nodo raíz. Buscamos los nodos que se han calculado en la misma rama donde se encuentra el nodo seleccionado (línea 9) y se buscan los grupos no resueltos en dichos nodos (línea 10). En esta primera iteración, dado que solo se ha calculado el grupo *Start*, quedarían por resolver todos los demás : *Gr₁*, *Gr₂*, *Gr₃*, *Gr₄*, *Gr₅* y *End*, en total seis grupos. Esta lista se asigna inicialmente a *remainingGroups*[].

En la línea 11 se ejecuta el algoritmo 4, encargado de eliminar de entre esos seis grupos a aquéllos que tienen una dependencia de otros aún no calculados en esta misma rama. Para ello, se recorre *remainingGroups*[] (línea 3) y, para cada grupo, se almacena en *precedents*[] la lista de sus precedentes según lo reflejen las dependencias del modelo *MODEL* (línea 4). Todos los elementos de dicha lista deben estar en algún nodo de la rama que se está analizando. En caso contrario, se descarta como candidato a formar parte de un nuevo nodo en esta etapa y se elimina de *remainingGroups*[] (línea 6). Esto ocurre en el ejemplo de la figura 5.7 con los grupos *Gr₃*, *Gr₅* y *End*, por lo que la lista inicial de seis grupos queda reducida a la formada por *Gr₁*, *Gr₂* y *Gr₄*.

Algoritmo 4: Función para descartar grupos que dependen en orden de ejecución de otros grupos que aún no han sido calculados.

```

1 TRUNCATE_REMAINING(node, MODEL, GROUPS[ ], NODES[ ],
  VisitedNodes[ ], remainingGroups[ ])
  Result: remainingGroups[ ]
2 groupsvisited[ ]: groups solved in VisitedNodes[ ];
3 for each group inside remainingGroups[ ] do
4   | precedents[ ]: precedents of group in MODEL;
5   | if all elements in precedents[ ] NOT in groupsvisited[ ] then
6   |   | remainingGroups[ ] – = group;
7   | end
8 end
```

El algoritmo 3 intenta entonces añadir al árbol los nodos que contengan cualquier combinación de los grupos Gr_1 , Gr_2 y Gr_4 . El bucle mostrado en la línea 13 gestiona este proceso, llamando al algoritmo 5 para añadir primero los nodos que contienen un solo grupo, y a continuación las combinaciones de dos y tres grupos. Este proceso supone un total de $2^3 - 1$ combinaciones.

Algoritmo 5: Función que selecciona nuevos nodos hijo con combinaciones válidas de un determinado orden sobre el conjunto de *remainingGroups* durante el proceso de creación del árbol de rutas que resuelve un determinado problema.

```

1 CONSTRUCT_TREE(MODEL, GROUPS[], NODES[], VisitedNodes[],
   remainingGroups[], sizeOfCombination)
   Result: newNodes[]
2 newNodes[] = [];
3 OriginDestinationRoutine[] = [];
4 for each combination of sizeOfCombination groups inside
   remainingGroups do
5     valid = true;
6     if any group in combination depends on others then
7         valid = false;
8         return
9     end
10    if other combination has same origin, destination and routines then
11        valid = false;
12    else
13        OriginDestinationRoutine[] += origin, destination and routine of
        combination;
14    end
15    if there is another branch in the tree that already contains the same
        type of groups then
16        valid = false
17    end
18    if valid = true then
19        newNodes[] = newNodes[] + combination
20    end
21 end

```

El algoritmo 5 recibe como argumentos, entre otros, el número de elementos de las combinaciones (*sizeOfCombination*) y los grupos a combinar (*remainingGroups*[]). Cada combinación obtenida en la línea 4 de dicho algo-

ritmo representa la resolución en paralelo de los grupos que la forman. Para que una combinación sea válida y pueda añadirse a un nodo del árbol como hijo del nodo seleccionado, se deben satisfacer las siguientes validaciones:

- En la línea 6 el software usa el grafo del modelo para comprobar que ningún grupo tenga dependencia de otro que esté dentro de esta misma combinación. Esta circunstancia impediría que se pudieran resolver en paralelo y, por tanto, estar en el mismo nodo del árbol. En este momento cualquier combinación formada por $\{Gr_1, Gr_2, Gr_4\}$ es válida pues, según indica el grafo 5.7, no hay dependencias entre ellos.
- En la línea 10 se busca otra combinación ya generada en la que los grupos que la constituyen tengan los mismos grupos precedentes y descendientes, y que a la vez ejecuten las mismas rutinas con matrices del mismo tamaño (se puede consultar el ejemplo mostrado al describir la propiedad 2 del árbol de rutas en la sección 5.5.1). En caso de encontrarse, se puede reducir el número de ramas descartando una de dichas combinaciones, ya que computacionalmente tienen el mismo coste.
- En la línea 15 se busca en el árbol otra rama que esté formada por combinaciones de grupos que tengan las mismas rutinas, pero en un orden diferente. En ese caso, la combinación actual se descarta para satisfacer la propiedad 3 del árbol de rutas descrita en la sección 5.5.1.

En este paso 2 algunos nodos válidos serían los que contienen las siguientes combinaciones de grupos $[\{Gr_1\}, \{Gr_2\}, \{Gr_4\}; \{Gr_1, Gr_2\}; \{Gr_1, Gr_2, Gr_4\}]$, como muestra la figura 5.19.

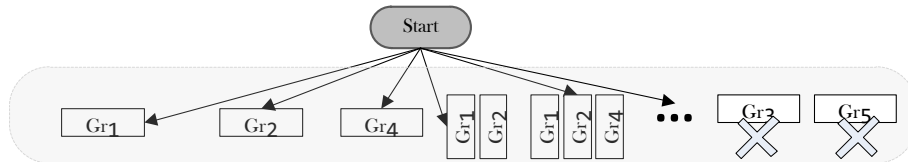


Figura 5.19: Búsqueda de nodos hijo para el nodo raíz que contiene al grupo *Start*. La zona sombreada muestra algunos de los nodos añadidos al árbol. En esta figura los nodos que contienen a Gr_3 y a Gr_5 son ejemplos de algunos nodos que se descartan por no considerarse válidos (necesitan que otros grupos se resuelvan antes que ellos).

- Paso 3: Se recorre al árbol para buscar un nuevo nodo sin hijos (línea 8 del algoritmo 3). En el ejemplo, se selecciona el que contiene al $\{Gr_1\}$, y se procede igual que en el paso 2. Los grupos que quedan por resolver serán (línea 10 del algoritmo 3): Gr_2 , Gr_3 , Gr_4 , Gr_5 y End . En la figura 5.20 observamos que, entre otros, los nodos formados por el grupo $\{Gr_2\}$ y por el par $\{Gr_2, Gr_4\}$ se consideran válidos y se añaden como dos nuevos nodos. Sin embargo, los que contienen a $\{Gr_3\}$ y $\{Gr_5\}$ se descartan porque dependen de grupos no calculados. Por ejemplo, Gr_3 necesita que Gr_1 y Gr_2 se hayan calculado con anterioridad, pero en la rama en la que se encuentra el algoritmo en este momento están el nodo raíz (con el grupo $\{Start\}$) y el nodo que contiene a $\{Gr_1\}$, pero ninguno con el grupo Gr_2 , lo que significa que aún no se ha resuelto. Lo mismo ocurre con el Gr_5 , que depende de Gr_3 y de Gr_4 , ninguno de los cuales se encuentran en esta rama.

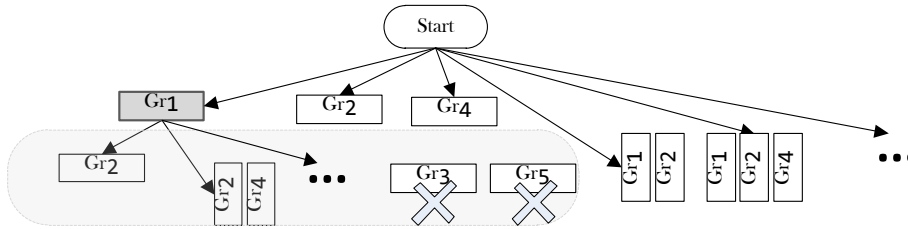


Figura 5.20: Búsqueda de nodos hijo para el nodo que contiene al grupo Gr_1 . Se descartan nodos que no cumplen con las precedencias del grafo del modelo.

- Paso 4: El siguiente nodo sin hijos es el que contiene al grupo Gr_2 , como refleja la figura 5.21. Procedemos de nuevo como en el paso 2.

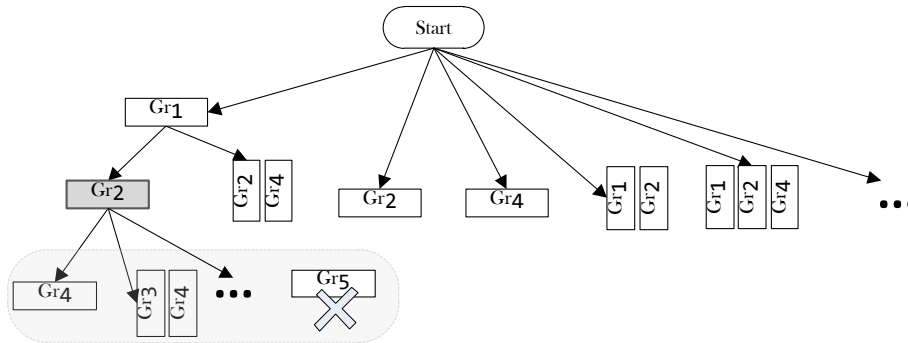


Figura 5.21: Búsqueda de nodos hijo para el nodo que contiene al grupo Gr_2 . El nodo que contiene al Gr_5 no es válido pues dicho grupo necesita que se hayan resuelto antes Gr_3 y Gr_4 en esta rama.

- Este proceso se repite hasta que todos los nodos tengan descendientes, excepto el nodo *End*, que marca el fin del algoritmo y que por tanto estará como último nodo en todas las ramas. El árbol obtenido, y que cumple con las propiedades descritas en la sección 5.5.1, se mostró en la figura 5.10.

Dependiendo de la cantidad de grupos y sus interconexiones, el proceso de creación del árbol de rutas puede llevar algún tiempo. Por este motivo el algoritmo utiliza dos variables que permiten detener su ejecución, bien superado un tiempo máximo (`maxTimeToGenerateTrees`), o bien alcanzado un número especificado de nodos en el árbol (`maxNumNodesInTrees`). Los valores de estas variables se leen del fichero `treeslimits.cfg` que se distribuye junto a la aplicación y que se puede modificar con un editor de texto. El listado 5.9 muestra un ejemplo del contenido de este fichero.

Listado 5.9 Contenido del fichero `treeslimits.cfg` que contiene los valores límites de tiempo y tamaño para el proceso de creación de un árbol de rutas.

```
maxNumNodesInTrees = 1000;  
maxTimeToGenerateTrees = 10; // seconds
```

5.5.3 Tiempo de ejecución de una ruta

Como vimos en 5.2.2, las funciones que forman parte de una rutina se ejecutan de manera secuencial, por lo que el tiempo de resolución de un grupo i que contiene k funciones, será la suma del tiempo de ejecución de todas las funciones f_{ik} , por tanto $T(Gr_i) = \sum_{j=1}^k T(f_{ij})$. Por otro lado, por el método descrito en 5.5.2 para construir el árbol de rutas, los n grupos contenidos en un nodo se ejecutarán en paralelo, por lo que el tiempo necesario para realizar los cálculos de un nodo N será igual al mayor de los tiempos necesarios para resolver los grupos que contiene, es decir, $T(N) = \max_{Gr_i \in N} \{T(Gr_i)\}$.

Con todo ello, el tiempo total necesario para resolver el problema siguiendo una determinada ruta R formada por x nodos será la suma de tiempos consumidos en los nodos que la forman:

$$T(R) = \sum_{i=1}^{nodos(R)} T(N_i) = \sum_{i=1}^{nodos(R)} \max_{Gr_j \in N_i} \left\{ \sum_{k=1}^{funciones(Gr_j)} T(f_{jk}) \right\} \quad (5.1)$$

Para ilustrarlo podemos utilizar el ejemplo mostrado en la figura 5.22, donde encontramos una determinada ruta R , formada por cuatro nodos N_x :

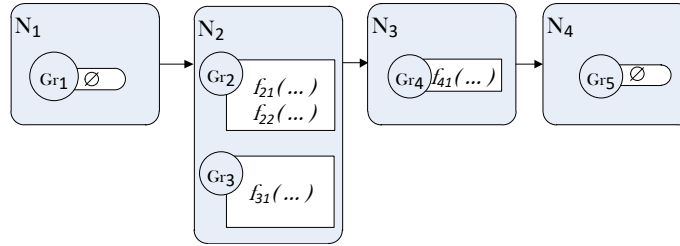


Figura 5.22: Ejemplo de ruta que resuelve un determinado problema numérico. La ruta contiene cuatro nodos. El nodo N_2 incluye la resolución de dos grupos.

- El nodo N_1 contiene el grupo $\{Gr_1\}$, que no ejecuta ninguna función.
- El nodo N_2 contiene los grupos $\{Gr_2, Gr_3\}$, que ejecutan rutinas que incluyen las funciones $\{f_{21}, f_{22}\}$ y $\{f_{31}\}$ respectivamente.
- El nodo N_3 contiene el grupo $\{Gr_4\}$, cuya rutina asociada ejecuta la función $\{f_{41}\}$.
- El nodo N_4 contiene el grupo $\{Gr_5\}$, sin ninguna función.

Si asumimos que el tiempo de ejecutar un grupo que no contiene rutinas de cálculo es 0, entonces $T(N_1) = 0$ y $T(N_4) = 0$. Por tanto, el tiempo de ejecutar la ruta representada en la figura 5.22 sería:

$$T(R) = T(N_1) + T(N_2) + T(N_3) + T(N_4)$$

$$T(R) = \max \{T(f_{21}) + T(f_{22}), T(f_{31})\} + T(f_{41})$$

5.5.4 Estimación de tiempos

Una vez elaborado el árbol de rutas asociado a un determinado modelo, se puede implementar en el simulador un proceso de estimación del tiempo de ejecución asociado a cada rama para encontrar la que teóricamente puede ofrecer una resolución más rápida en una determinada plataforma hardware.

Según la fórmula 5.1, el cálculo del tiempo de ejecución de cualquier rama se basa en los tiempos de ejecución conocidos de las funciones utilizadas. Ahora bien, de manera general estos tiempos son diferentes en función de los parámetros algorítmicos, AP , y el tipo de datos descrito en el escenario, SCN , sobre el que operan dichas funciones. En PARCSIM los AP están formados por el conjunto $\{n_c, n_g, l\}$, donde n_c es el número de cores, n_g la cantidad de GPUs (en la versión actual del simulador no se distingue entre diferentes tipos de GPUs instaladas) y l el tipo de librería. Un escenario SCN engloba al tamaño, topología y factor de dispersión de las matrices. Por tanto, podemos reescribir la ecuación 5.1 de manera que refleje la variabilidad de los tiempos de ejecución en función de los AP y SCN cuando se resuelve un problema siguiendo una determinada ruta R :

$$\begin{aligned}
 T(R, AP, SCN) &= \sum_{i=1}^{niveles(R)} T(N_i, AP, SCN) = \sum_{i=1}^{niveles(R)} \max_{Gr_j \in N_i} \{T(Gr_j, AP, SCN)\} \\
 T(R, AP, SCN) &= \sum_{i=1}^{niveles(R)} \max_{Gr_j \in N_i} \left\{ \sum_{k=1}^{funciones(Gr_j)} T(f_{jk}, AP, SCN) \right\} \quad (5.2)
 \end{aligned}$$

donde N_i representa los nodos en el nivel i de la ruta R .

De la fórmula anterior se deduce que un proceso de estimación pseudo-teórica del tiempo de ejecución de una rama necesita disponer de antemano de información sobre los tiempos de ejecución de las funciones obtenidos con diferentes parámetros algorítmicos (AP) sobre diferentes escenarios de datos (SCN). El simulador permite la obtención experimental de estos tiempos de ejecución en su modo de entrenamiento, que será descrito posteriormente en la sección 5.6.2.1.

En esta tesis proponemos una metodología que permite al simulador determinar los *AP* que minimizan los tiempos de ejecución para un determinado *SCN*. El proceso debe encontrar, para cada nodo del árbol, la mejor asignación de threads, GPUs y librería para resolver los cálculos incluidos en dicho nodo. La búsqueda de la mejor configuración de parámetros en PARCISM contempla los siguientes aspectos:

- El número de cores n_c : Establecerá un límite en el nivel de paralelismo que se puede aplicar, lo que permitirá descartar las ramas del árbol con nodos que demanden una cantidad de recursos paralelos no disponibles. PARCISM implementa actualmente el uso de paralelismo OpenMP en dos niveles. El primero para la resolución simultánea de grupos (paralelismo de grupos), y el segundo para la explotación del paralelismo implícito de las librerías o funciones. Dado que actualmente el simulador limita el número de threads al número de cores lógicos del hardware, se cumplirá que:

$$th.omp_1 \cdot th.omp_2 \leq n_c \quad (5.3)$$

siendo $th.omp_1$ el número de threads asignados al primer nivel de paralelismo OpenMP y $th.omp_2$ los asignados al segundo nivel.

- El número de GPUs n_g : PARCISM hace uso de las GPUs presentes en el hardware a través de librerías de cómputo preparadas para ello. En este caso, las llamadas a dichas librerías se hace desde el thread que está resolviendo un determinado grupo. Por este motivo, el número de GPUs máximo que se van a utilizar simultáneamente estará limitado por el número de threads del primer nivel de paralelismo en ejecución:

$$n_g \leq th.omp_1 \quad (5.4)$$

- Los escenarios *SCN*: Que describen el tamaño y dispersión de las matrices que se van a manejar, y que harán recomendable el uso de una u otra librería. Algunas de ellas, como hemos indicado anteriormente, puede ofrecer versiones paralelas que hay que tener en cuenta de cara a la asignación de threads.

El algoritmo 6 se encarga de calcular los costes o tiempos de ejecución requeridos para resolver cada uno de los nodos del árbol de rutas. Inicialmente se les asigna un valor *undefined* para indicar que están pendientes de calcular (línea 3) y un bucle recorre el árbol (línea 5) analizando los grupos que forman cada nodo. El caso en el que ninguno de los grupos tiene una rutina asignada significa que en dicho nodo no se realiza ningún cálculo, por lo que se le puede asociar un coste 0 (línea 7). De lo contrario se llama a la función **CALCULATE_NODE_COST**, que será descrita en las secciones siguientes.

Algoritmo 6: Procedimiento para asignar un coste (tiempo de ejecución) a todos los nodos de un árbol de rutas previamente creado. Una vez calculado el coste, el sistema busca en el árbol otros nodos que tengan la misma carga computacional (es decir, la misma rutina y tamaño de datos) y les asigna el mismo coste.

```
1 FILL_COSTS( $n_c, n_g, SCN, MODE, Tree$ )
2 for each node in Tree do
3   |  $cost(node) = \text{undefined};$ 
4 end
5 for each node in Tree do
6   | if node has no routine then
7     |  $cost(node) = 0;$ 
8   | else
9     | if  $cost(node) = \text{undefined}$  then
10    |    $tmpCost = \text{CALCULATE\_NODE\_COST}(node, th.omp_1,$ 
11    |    $th.omp_2, n_c, n_g, SCN, MODE);$ 
12    |    $cost(node) = tmpCost;$ 
13    |    $\text{COPY\_COSTS}(node, tmpCost);$ 
14    | end
15 end
```

Una vez obtenido el coste, y con objeto de no repetir los cálculos, la función **COPY_COSTS** (línea 12) busca a lo largo del árbol otros nodos que contengan grupos con la misma rutina y el mismo tipo de datos (mismo tamaño y tipo de las matrices). En este caso, el tiempo necesario para resolver ambos nodos se considera que es el mismo y se copia el coste obtenido a todos ellos.

Como se ha descrito anteriormente, el cálculo del tiempo de ejecución de un nodo lo realiza la función **CALCULATE_NODE_COST**, la cual presenta dos variantes, dependiendo de si el usuario especifica o no los parámetros algorítmicos *AP*. Los apartados siguientes describen ambos casos.

5.5.4.1 Estimación del tiempo de ejecución de un nodo con parámetros algorítmicos preestablecidos

Cuando el usuario fija el valor del número de threads asignados al primer y segundo nivel de paralelismo ($th.omp_1$, $th.omp_2$), el número de GPUs instaladas en el sistema (n_g) y la librería de cómputo a utilizar (l), entonces se ejecuta la función descrita en el algoritmo 7.

Algoritmo 7: Función para obtener el tiempo de ejecución estimado, *nodeCost*, necesario para calcular un determinado nodo, *node*, en un escenario *SCN* cuando el número de threads del primer y segundo nivel de paralelismo ($th.omp_1$, $th.omp_2$), librería (l) y cantidad de GPUs instaladas (n_g) son fijados por el usuario. La función devuelve también el número de GPUs con las que se obtiene el menor tiempo de ejecución.

```

1 CALCULATE_NODE_COST(node,  $th.omp_1$ ,  $th.omp_2$ ,  $n_c$ ,  $n_g$ ,  $l$ , SCN,
  MODE)
  Result: nodeCost, Best_AP[ ]
2  $n$ : number of groups in node;
3 if  $n > th.omp_1$  then
4   | nodeCost =  $\infty$ ;
5 else
6   | temp_AP = Best_AP;
7   | nodeCost =  $\infty$ ;
8   |  $C_{n_g}[]$  = possible combinations of  $n_g$ ;
9   | for each element in  $C_{n_g}$  do
10    | tmpCost = GET_NODE_COST(node,  $th.omp_2$ , element,  $l$ , SCN,
      | MODE);
11    | if tmpCost < nodeCost then
12      | nodeCost = tmpCost;
13      | Best_AP.n_g = element;
14    | end
15  | end
16 end

```

- La línea 4 asigna un coste ∞ cuando el número de grupos que contiene el nodo, y que por tanto se deben resolver simultáneamente, es mayor que el número de threads indicados para el primer nivel de paralelismo *th.omp1* (paralelismo de grupos). De esta manera, por la fórmula 5.2, la rama en la que se incluye dicho nodo tendrá también un coste ∞ y quedará fuera del proceso de selección de rutas.
- Si los grupos que contiene el nodo se pueden resolver en paralelo el algoritmo genera las posibles combinaciones C_{n_g} que reparten el total de n_g GPUs entre los grupos que forman el nodo (línea 9).
- En la línea 10 se llama a la función **GET_NODE_COST**, encargada de estimar el coste de ejecución de un nodo al aplicar ciertos parámetros algorítmicos (los threads y librería indicados por el usuario, y el número de GPUs que surge de cada combinación obtenida en el paso anterior). Si alguna combinación asigna el uso de GPU a una librería de cómputo que no lo admite, se devuelve un coste de valor ∞ para descartar dicha combinación. En las líneas 12 y 13 se establecen los valores que devolverá la función: el coste y la combinación de GPUs que ofrecen el menor tiempo de ejecución.

Ilustraremos este proceso con un ejemplo basado en el modelo que se utilizó en la sección 5.2.3 y que se muestra nuevamente en la figura 5.23.

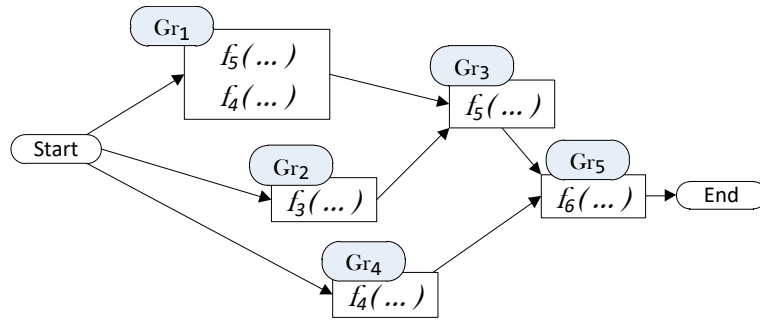


Figura 5.23: Modelo formado por siete grupos con las funciones ya desglosadas a partir de las rutinas que definen la solución de cada grupo.

Una de las posibles rutas que permiten su resolución se representa en la figura 5.24. En ella se observa que los grupos Gr_1 , Gr_2 y Gr_4 se calculan de manera simultánea en el segundo nodo. Las funciones que se ejecutan son f_3 , f_4 y f_5 .

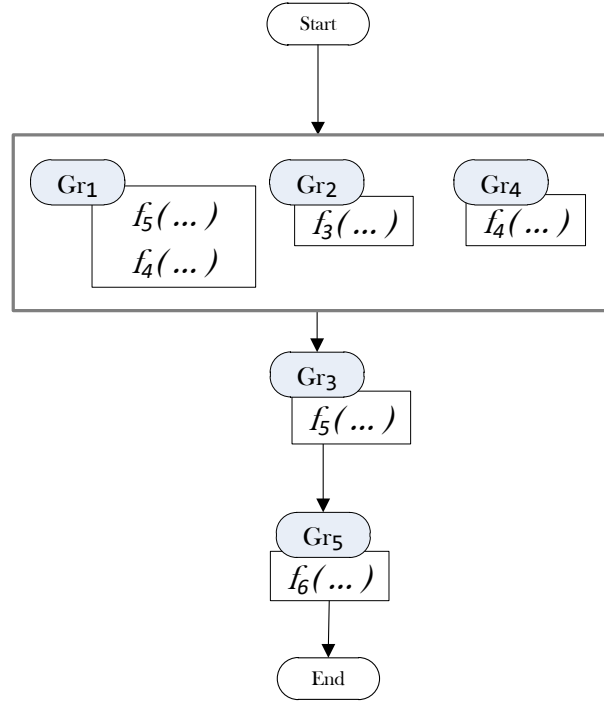


Figura 5.24: Representación de una de las rutas que resuelve el modelo de la figura 5.23 mediante la ejecución simultánea de los grupos Gr_1 , Gr_2 y Gr_4 .

Estudiaremos dos posibles casos. En el primero queremos calcular el coste del nodo que resuelve los grupos Gr_1 , Gr_2 y Gr_4 aplicando los siguientes valores de parámetros algorítmicos AP :

$$th.omp_1 = 1, th.omp_2 = 1, n_g = 0 \text{ y } l = 1$$

En este supuesto el número de grupos que contiene el nodo (tres) es mayor que el número de threads reservados para el paralelismo de grupos (uno). Por tanto, según la línea 4 del algoritmo 7, se asigna un coste ∞ al nodo, lo que hace que dicha ruta se descarte como válida para este juego de AP , independientemente del escenario SCN .

En un segundo caso, el usuario quiere obtener el tiempo de ejecución de ese mismo nodo pero con un nuevo conjunto de parámetros:

$$th.omp_1 = 3, th.omp_2 = 2, n_g = 1 \text{ y } l = 1$$

Como esta vez $th.omp_1 = 3$, es posible calcular en paralelo los tres grupos, y la línea 9 del algoritmo 7 recorre todas las combinaciones que utilizan un máximo

de $n_g = 1$ GPUs, obteniendo el coste del nodo asociado a cada una de ellas. Las combinaciones posibles son $\{(0,0,0), (1,0,0), (0,1,0), (0,0,1)\}$. Los tres valores en cada combinación indican la cantidad de GPUs asignadas a cada uno de los tres grupos. Ahora bien, la librería que corresponde con $l = 1$, según la tabla 5.1, es MKL. Esta librería no admite el uso de GPUs, por lo que la única combinación válida es $(0,0,0)$, y la función **GET_NODE_COST** llamada en la línea 10 se va a ejecutar solo una vez para obtener el tiempo de ejecución de las funciones cuando se asignan los parámetros $th.omp_2 = 2$, $n_g = 0$ y $l = 1$ y se manejan las matrices definidas en el escenario *SCN*.

El algoritmo 8 ejecuta la lógica de la función **GET_NODE_COST** y comienza en la línea 2 recorriendo todos los grupos que forman el nodo, en este caso tres. Para cada grupo se obtiene su rutina, y de cada rutina se obtiene el desglose en funciones, como se indica en las líneas 3 y 5. A continuación se consultan los registros almacenados en la base de datos de entrenamiento para cada una de dichas funciones y el tipo de matrices que defina el escenario *SCN* (línea 10). Si se encuentra un registro con los mismos parámetros algorítmicos, tamaño y factor de dispersión de las matrices que los buscados (línea 14), el coste almacenado en la base de datos se suma al coste del grupo (línea 15). El coste total del nodo (línea 33) será el mayor de los costes de los grupos que lo forman, como vimos en la sección 5.5.3.

Consideremos un escenario *SCN* de ejemplo con matrices cuadradas, de tamaños 100×100 , simétricas (es decir, de tipo 2, según indica la tabla 5.3), y un factor de dispersión del 50% y una base de datos de entrenamiento que contiene los registros que muestra la tabla 5.5.

Observamos que tenemos registros que corresponden a los mismos *AP* y *SCN* que se buscan, lo que permite a la función **GET_NODE_COST** obtener directamente los tiempos de ejecución T de cada grupo y, por tanto, del nodo:

$$T(Gr_1, AP, SCN) = T(f_5, AP, SCN) + T(f_4, AP, SCN) = 0.2 + 0.7 = 0.9;$$

$$T(Gr_2, AP, SCN) = T(f_3, AP, SCN) = 0.5;$$

$$T(Gr_4, AP, SCN) = T(f_4, AP, SCN) = 0.7;$$

$$T(N, AP, SCN) = \text{máx}\{0.9, 0.5, 0.7\} = 0.9$$

Algoritmo 8: Estimación del tiempo de ejecución de un nodo, *node*, cuando se aplican los parámetros algorítmicos *AP* en un escenario *SCN*. Si no hay información de entrenamiento ajustada al escenario *SCN*, se busca el escenario *SCN_nearest* que más se aproxime.

```
1 GET_NODE_COST(node, th2, ng, library, SCN, MODE)
   Result: cost, AP
2 for each group in node do
3   R: routine in group;
4   tmpCost = 0;
5   for each function in routine R do
6     cost = ∞;
7     m_size: matrix sizes for function in scenario SCN;
8     m_type: matrix types for function in scenario SCN;
9     m_sparsity: matrix sparsity for function in scenario SCN;
10    for each record in training database for function and m_type do
11      if record.matrixsize = m_size
12        and record.matrixsparsity = m_sparsity
13        and record.th2 = th2
14        and record.gpu = ng then
15        | tmpCost += record.cost;
16      end
17    end
18    if tmpCost = 0 then
19      SCN_nearest = GET_NEAREST_SCENARIO(SCN);
20      n_m_size: matrix sizes for function in SCN_nearest;
21      n_m_sparsity: matrix sparsity for function in SCN_nearest;
22      for each record in training database for function and m_type
23        do
24        | if record.matrixsize = n_m_size
25          and record.matrixsparsity = n_m_sparsity
26          and record.th2 = th2
27          and record.gpu = ng then
28          | tmpCost += record.cost;
29        | end
30      end
31    end
32    if tmpCost > cost then
33    | cost = tmpCost;
34  end
35 end
```

function	AP		SCN			Cost
	th.omp2	gpu	size	sparsity	type	execution time in seconds
f_3	2	0	100×100	50	2	0.5
f_4	2	0	100×100	50	2	0.7
f_5	2	0	100×100	50	2	0.2

Tabla 5.5: Contenido de la base de datos de entrenamiento para las funciones f_3 , f_4 y f_5 , obtenidos con la librería MKL operando sobre matrices de tamaño 100×100 .

Si el procedimiento no pudiera encontrar registros para el escenario *SCN* (línea 18 del algoritmo 8), entonces se buscaría el más próximo *SCN_nearest* (línea 19 del algoritmo 8) siguiendo un criterio de distancias mínimas, y se recuperaría el valor del coste asociado a ese escenario (línea 27). Supongamos, por ejemplo, un contenido de la base de datos de entrenamiento como el mostrado en la tabla 5.6.

function	AP		SCN			Cost
	th.omp2	gpu	size	sparsity	type	execution time in seconds
f_3	2	0	50×50	50	2	0.3
f_4	2	0	50×50	50	2	0.5
f_5	2	0	50×50	50	2	0.1
f_3	2	0	200×200	50	2	1.5
f_4	2	0	200×200	50	2	2.0
f_5	2	0	200×200	50	2	0.5

Tabla 5.6: Contenido de la base de datos de entrenamiento para las funciones f_3 , f_4 y f_5 , obtenidos con la librería MKL usando matrices de tamaños 50×50 y 200×200 .

Se puede comprobar que no se dispone de tiempos de ejecución para matrices de tamaños 100×100 , pero sí para 50×50 y 200×200 . Dado que el factor de dispersión coincide con el buscado, entonces el cálculo de las distancias tiene en cuenta únicamente al tamaño de las matrices:

$$d(SCN, SCN_{50 \times 50}) = \left(\frac{50 - 100}{100} \right)^2 = 0.25;$$

$$d(SCN, SCN_{200 \times 200}) = \left(\frac{200 - 100}{100} \right)^2 = 1;$$

Por este criterio, el escenario más próximo a 100×100 es 50×50 , con una distancia de 0.25. El coste (tiempo de ejecución estimado) del nodo para *SCN_nearest* será:

$$\begin{aligned} T(Gr_1, AP, SCN_nearest) &= 0.1 + 0.5 = 0.6; \\ T(Gr_2, AP, SCN_nearest) &= 0.2; \\ T(Gr_4, AP, SCN_nearest) &= 0.5; \\ T(N, AP, SCN_nearest) &= \max\{0.6, 0.2, 0.5\} = 0.6 \end{aligned}$$

5.5.4.2 Estimación del tiempo de ejecución de un nodo con autooptimización de parámetros algorítmicos

Si los parámetros algorítmicos, *AP*, no están prefijados por el usuario, entonces el simulador ejecuta una variante de la función **CALCULATE_NODE_COST** que se describe en el algoritmo 9, y que permite obtener los valores de *AP* que suponen el aprovechamiento óptimo del hardware disponible (número de CPUs y GPUs) cuando se trabaja con el tipo de matrices especificado en el escenario *SCN*:

- La línea 4 asigna un coste ∞ al nodo si el número de grupos que contiene, y que se deben resolver simultáneamente, es mayor que el número de cores n_c . Por la fórmula 5.2, la rama en la que se incluye dicho nodo tendrá también un coste ∞ , por lo que se descarta como opción para resolver el modelo.
- Se invoca a la función **GET_ROUTINE_BEST_PARAMETERS** (línea 15) que se encarga de buscar en la base de datos de entrenamiento los *AP* que ofrecen los menores tiempos de ejecución de las rutinas. Este proceso se repite para cada uno de los grupos que forman el nodo. Ahora bien, si todos ellos ejecutan la misma rutina (línea 6), entonces la llamada a **GET_ROUTINE_BEST_PARAMETERS** incluye como argumentos las variables *maxCores* y *maxGPUs*, cuyos valores son el cociente obtenido al dividir entre el número de grupos la cantidad de cores y GPUs disponibles respectivamente, lo que significa un reparto igualitario de los recursos, evitando la asignación de más recursos de los disponibles.

Algoritmo 9: Proceso autooptimizado para obtener los mejores parámetros algorítmicos, *Best_AP*, y el tiempo de ejecución, *nodeCost*, de un nodo, *node*, y escenario *SCN*. n_c y n_g contienen el número de cores y de GPUs respectivamente. La función obtiene *Best_AP.n_c*, *Best_AP.n_g* y *Best_AP.library* para cada grupo de *node*.

```

1  CALCULATE_NODE_COST(node, th.omp1, th.omp2,  $n_c$ ,  $n_g$ , l, SCN,
   MODE)
   Result: nodeCost, Best_AP[]
2  ng: number of groups in node;
3  if  $n_g > n_c$  then
4    | nodeCost =  $\infty$ ;
5  else
6    | if all groups execute the same routine then
7      | maxCores:  $n_c/n_g$ ; maxGPUs:  $n_g/n_g$ ;
8    | else
9      | maxCores:  $n_c$ ; maxGPUs:  $n_g$ ;
10   | end
11   m_size: matrix sizes for function in scenario SCN;
12   for each group in node do
13     | if group executes a Routine R then
14       | (tmpcost, AP) = GET_ROUTINE_BEST_PARAMETERS
15       | (R, SCN, maxCores, maxGPUs);
16     | end
17     | Best_AP.nc[group] += AP.nc;
18     | Best_AP.ng[group] += AP.ng;
19     | nodeCost += tmpcost;
20   end
21   if sum of Best_AP.nc[] >  $n_c$  or sum of Best_AP.ng[] >  $n_g$  then
22     | temp_AP = Best_AP;
23     | nodeCost =  $\infty$ ;
24     | reduce temp_AP.nc[] until sum of temp_AP.nc[] ≤  $n_c$ ;
25     | reduce temp_AP.ng[] until sum of temp_AP.ng[] ≤  $n_g$ ;
26     | for each possible combination CAP of temp_AP.nc[] and
       | temp_AP.ng[] do
27       | (tmpCost, temp_AP.library[]) = GET_NODE_COST(node,
        | CAP.nc, CAP.ng, SCN, MODE);
28       | if tmpCost < nodeCost then
29         | | nodeCost = tmpCost;
30         | | Best_AP = CAP;
31       | end
32     | end
33   end
34 end

```

- En el caso de que las rutinas que se ejecutan en cada grupo sean diferentes, no existe a priori un reparto de recursos óptimo, en cuyo caso la función **GET_ROUTINE_BEST_PARAMETERS** buscará los mejores, pudiendo darse el caso de que encuentre unos *AP* cuya suma exceda el límite del número de cores y GPUs que impone el hardware (línea 21 del algoritmo 9). Si los *AP* están dentro de los límites, entonces habremos encontrado también el coste teórico que se consigue aplicando dichos *AP*, y que resultará ser el menor tiempo de ejecución.
- Si los valores propuestos exceden las capacidades del hardware, el simulador inicia un proceso de reducción del número de threads y GPUs asignados a cada grupo, comenzando por los que tienen un valor mayor, hasta alcanzar una suma que esté dentro de los límites. Con ellos genera todas las combinaciones posibles de asignación de recursos a cada grupo, ejecutando entonces la función **GET_NODE_COST** (algoritmo 8) para estimar el coste de ejecución de dicho nodo para cada combinación de *AP* y escenario *SCN* (o el más cercano). En este modo autooptimizado, si dicha función no encuentra datos de entrenamiento para la librería que recibe como argumento, o bien encuentra otra con un mejor rendimiento para los mismos recursos de hardware, entonces propone el cambio de la librería.

GET_ROUTINE_BEST_PARAMETERS ejecuta el algoritmo 10, iterando por todas las funciones que componen la rutina (línea 2) y accediendo a la base de datos para buscar registros para dicha función (línea 6). Cuando algunos de esos registros hacen referencia al mismo escenario *SCN*, dentro del límite de cores y GPUs marcados por n_c y n_g (línea 9), entonces se busca entre ellos el que tenga un menor tiempo de ejecución (línea 10) y toma los valores de sus *AP*. Si no existen registros para el mismo escenario, se busca el más cercano (línea 18) siguiendo el mismo criterio de distancias mínimas descrito en la sección anterior.

Para ilustrar este procedimiento nos centraremos en el segundo nodo de la ruta que se muestra en la figura 5.24, en el que los grupos Gr_1 , Gr_2 y Gr_4 se calculan de manera simultánea. A diferencia de la sección anterior, donde se especificaban los valores de los *AP*, ahora el simulador calculará los mejores *AP* para dicho nodo, conocidos el número de cores, n_c , y el número de GPUs, n_g .

Algoritmo 10: Obtención de los parámetros algorítmicos AP que consi-
guen el mejor tiempo de ejecución, $cost$, de una determinada rutina R en
un escenario SCN .

```

1  GET_ROUTINE_BEST_PARAMETERS( $R, SCN, n_c, n_g$ )
   Result:  $cost, AP$ 
2  for each function in routine  $R$  do
3       $cost = \infty$ ;
4       $m\_size$ : matrix sizes for function in scenario  $SCN$ ;
5       $m\_type$ : matrix types for function in scenario  $SCN$ ;
6      for each record in training database for function function do
7          if record.matrixsize =  $m\_size$  and record.matrixtype =  $m\_type$ 
8              and record.cores  $\leq n_c$ 
9              and record.gpus  $\leq n_g$  then
10             if record.cost  $< cost$  then
11                  $cost = record.cost$ ;
12                  $AP.n_c = \max(AP.n_c, record.cores)$ ;
13                  $AP.n_g = \max(AP.n_g, record.gpus)$ ;
14                  $AP.library = record.library$ ;
15             end
16         end
17         if  $cost = \infty$  then
18              $SCN\_nearest$ : GET_NEAREST_SCENARIO( $SCN$ );
19              $n\_m\_size$ : matrix sizes for function in  $SCN\_nearest$ ;
20              $n\_m\_sparsity$ : matrix sparsity for function in  $SCN\_nearest$ ;
21             for each record in training database for function and  $m\_type$ 
22                 do
23                     if record.matrixsize =  $n\_m\_size$ 
24                         and record.matrixsparsity =  $n\_m\_sparsity$ 
25                         and record.th2 =  $th_2$ 
26                         and record.gpu =  $n_g$  then
27                             if record.cost  $< cost$  then
28                                  $cost = record.cost$ ;
29                                  $AP.n_c = \max(AP.n_c, record.cores)$ ;
30                                  $AP.n_g = \max(AP.n_g, record.gpus)$ ;
31                                  $AP.library = record.library$ ;
32                             end
33                         end
34                     end
35                 end
36 end

```

Como primer ejemplo, partimos de los datos de entrenamiento que se muestran en la tabla 5.7, y unos límites fijados inicialmente en $n_c = 8$ y $n_g = 2$.

function	AP			SCN			Cost
	library	th.omp2	gpu	size	sparsity	type	exec. time in seconds
f_3	1	2	0	100×100	50	2	0.5
f_3	3	1	0	100×100	50	2	0.1
f_4	1	1	0	100×100	50	2	1.0
f_4	1	2	0	100×100	50	2	0.9
f_4	2	1	0	100×100	50	2	1.5
f_4	1	3	0	100×100	50	2	0.7
f_5	7	1	1	100×100	50	2	0.3
f_5	1	1	1	100×100	50	2	1.1
f_5	2	1	1	100×100	50	2	1.4
f_5	1	2	0	100×100	50	2	0.8

Tabla 5.7: Contenido de la base de datos de entrenamiento para las funciones f_3 , f_4 y f_5 con matrices de tamaños 100×100 .

Con dicha información, la función **GET_ROUTINE_BEST_PARAMETERS** propone los **AP** que permiten obtener los mejores tiempos de ejecución para cada grupo, como los que se muestran en la tabla 5.8. Observamos que la suma de recursos propuesta para el total del nodo es de siete cores y una GPU, lo que se encuentra dentro de los límites establecidos de $n_c = 8$ y $n_g = 2$.

Function	Gr_1		Gr_2		Gr_4	Node
	f_5	f_4	f_3	f_4		
Library	7	1	3	1		
Cores	1	3	1	3		7
GPUs	1	0	0	0		1
Exec.time (seconds)	0.3	0.7	0.1	0.7		1.0

Tabla 5.8: Mejores parámetros algorítmicos **AP** aplicables al nodo que resuelve simultáneamente los grupos Gr_1 , Gr_2 y Gr_4 . La información se obtiene desglosada por función ejecutada, a partir de la información de entrenamiento mostrada en la tabla 5.7.

Por tanto, el mejor tiempo de ejecución para este nodo, en función de los datos de entrenamiento, es de 1 segundo.

Si repetimos el mismo proceso con un hardware más reducido, en concreto con $n_c = 4$ y $n_g = 0$, ya no será posible ejecutar el nodo aplicando los mejores **AP** identificados en el caso anterior, pues estos exceden los recursos disponibles. La

alternativa es buscar entre todas las combinaciones de cores y GPUs que respetan las limitaciones, e identificar la que ofrece un mejor rendimiento.

El simulador propuso como *AP* óptimos el uso de siete cores, desglosados en tres para resolver el grupo Gr_1 , uno para el Gr_2 y tres para el Gr_4 , lo que podemos representar como una terna $(3, 1, 3)$. El software inicia ahora un proceso en el que va reduciendo las asignaciones de cores, de uno en uno, para conseguir las ternas que respetan el límite de $n_c = 4$. En este ejemplo la terna $(1, 1, 2)$ se consigue reduciendo la asignación inicial de recursos de los grupos Gr_1 y Gr_4 . Otra terna válida es $(2, 1, 1)$.

Por otro lado, como $n_g = 0$, la propuesta inicial de resolver la función f_5 asignando una GPU ya no es válida en este hardware. Por tanto, la única terna de asignación de GPUs válida es $(0, 0, 0)$.

Con esta información el algoritmo 9 en su línea 26 calcula el coste estimado sin utilizar GPUs y realizando las asignaciones de cores que determinan las ternas $(1, 1, 2)$ y $(2, 1, 1)$:

- $(1, 1, 2)$: Esta terna impone la limitación de 1 core para resolver el grupo Gr_1 , aplicable a las funciones f_4 y f_5 que componen su rutina. El mejor tiempo de entrenamiento para la función f_4 usando un core y ninguna GPU es 1.0 segundos mediante la librería 1 (MKL). En el caso de f_5 , el tiempo es de 1.1, también con MKL. Por tanto el grupo tiene un tiempo estimado de resolución de $1.0 + 1.1 = 2.1$ segundos. De igual manera se obtienen los tiempos para el Gr_2 , para el que sigue siendo válida la opción de usar la librería 3 (MA27), con 0.1 segundos. El Gr_2 también mantiene su asignación de dos cores, con un tiempo de 0.7 segundos usando MKL. El coste total del nodo para la combinación $(1, 1, 2)$ es, por tanto, de 2.1 segundos, que corresponde con el mayor de los tiempos de resolución de sus grupos.
- $(2, 1, 1)$: Aplicando a la función f_4 la restricción del uso de dos cores y ninguna GPU que impone esta terna, se observa en la base de datos de entrenamiento que el mejor dato es el de 0.9 segundos empleando la librería 1 (MKL). Respecto a f_5 , usando MKL se obtiene un tipo de ejecución de 0.8 segundos. Por tanto el grupo Gr_1 tiene un tiempo estimado de resolución

de $0.9 + 0.8 = 1.7$ segundos. El Gr_2 no varía su asignación de recursos, por lo que se puede usar la librería 3 (MA27), consumiendo 0.1 segundos. El Gr_2 , que ejecuta la función f_4 , reduce su número de cores a 1. En la base de datos de entrenamiento, el mejor tiempo obtenido usando un core y ninguna GPU es de 1.0 segundos mediante la librería 1 (MKL). El coste total del nodo para la combinación $(2, 1, 1)$ es, por tanto, $\max(1.7, 0.1, 1.0) = 1.7$.

Por tanto, el proceso de autooptimización aplicado al nodo que resuelve en paralelo los grupos Gr_1 , Gr_2 y Gr_4 , con una limitación de recursos del hardware dado por las asignaciones $n_c = 4$ y $n_g = 0$, y donde las funciones f_3 , f_4 y f_5 manejan matrices de tamaños 100×100 con un factor de dispersión del 50%, determina que la manera óptima de resolver dicho nodo es usar la librería MKL asignando dos threads al Gr_1 , y un thread tanto a Gr_2 como a Gr_4 , como corresponde a la terna $(2, 1, 1)$.

5.6 Simulación

Una vez que el usuario ha introducido en PARCSIM el modelo que representa un determinado algoritmo y los datos que se van a manejar, el proceso de simulación realiza la ejecución efectiva de los cálculos incluidos en dicho algoritmo. El simulador se puede configurar para realizar múltiples ejecuciones, variando de una a otra la ruta, los parámetros de paralelismo, las librerías de cálculo y el juego de datos. Cada simulación genera información sobre los tiempos de ejecución y los almacena en una base de datos para ser estudiados posteriormente.

5.6.1 Configuración del simulador

El simulador necesita cierta información para gestionar su funcionamiento. Esta información se almacena en un archivo de configuración llamado `config.cfg`, generado automáticamente a partir de la información especificada por el usuario en un interfaz gráfico. El simulador accede al contenido del archivo de configuración al inicio de la ejecución. El listado 5.10 muestra un ejemplo del formato de este tipo de fichero.

Listado 5.10 Contenido del fichero `config.cfg` que contiene la información de configuración del simulador.

```
logFile="LOG";

simulatorExeDirectory = "../../*.mdl";
simulatorExecutionDirectory = "../../*.mdl";
simulatorDatabaseDirectory = "../../*.mdl";

History=1;
ExecutionType=3;
BranchSelection=1;

hardwareName="HWTest";
ListOfModels=(
    {
        modelFile="Strassen1.mdl";
        modelName="ModeloEjemplo";
        modelId=4;
    }
);

THREADSLEVEL1=4;
THREADSLEVEL2=4;
LIBRARY=1;
NUMGPU=1;

CORES=4;
GPUS=3;
```

A continuación se muestran los campos incluidos en dicho fichero de configuración:

- `logFILE`: Contiene una cadena de texto que almacena los primeros caracteres del nombre de los archivos de registro que el simulador genera durante su ejecución, y que recogen información relevante para el usuario. El software concatena el contenido del campo `logFILE` con el nombre del modelo y la descripción del escenario que se ejecuta en cada momento para crear el nombre del archivo.
- `simulatorExeDirectory`: Contiene la ruta del sistema de ficheros del ordenador donde se realiza la simulación, y que hace referencia a la ubicación del fichero ejecutable del simulador.
- `simulatorExecutionDirectory`: Contiene la ruta donde el simulador debe buscar los archivos de los modelos, escenarios y scripts. Este mismo directorio se usará como destino donde generar los archivos de salida.

- `simulatorDatabaseDirectory`: Contiene la ruta donde se sitúa el fichero PARCSIM.db que almacena la tabla de resultados con el formato de SQLite (sección 3.2.3).
- `History`: Es el número máximo de registros que se almacenarán en la base de datos para una determinada combinación de modelo, función, parámetros algorítmicos y escenario. Registrar más de un valor permite obtener tiempos medios de ejecuciones realizadas en diferentes momentos.
- `ExecutionType`: Identifica el modo de simulación. PARCSIM admite cuatro modos que se describirán en la sección 5.6.2 y que se codifican con un número entero entre 0 y 3.
- `BranchSelection`: Un valor 0 indica al simulador que debe seleccionar automáticamente una rama de ejecución. Por el contrario, con un valor de 1 las simulaciones usarán la (o las) ramas especificadas por el usuario para cada modelo.
- `hardwareName`: Es una cadena alfanumérica que el usuario introduce para identificar a la plataforma hardware en la que se ejecutarán las simulaciones. Este dato se almacena junto a los tiempos de ejecución obtenidos para poder realizar comparaciones entre los rendimientos obtenidos en diferentes plataformas.
- `ListOfModels`: Contiene una terna (o lista de ellas) con información relevante de los modelos que el simulador va a ejecutar:
 - `modelFile`: Nombre del fichero donde se encuentra almacenada la información del modelo.
 - `modelName`: El nombre del modelo.
 - `modelId`: Un número entero que lo identifica de manera única.
- `THREADSLEVEL1`: Contiene el número máximo de threads que se asignarán al primer nivel de paralelismo. Aplicable en el modo de simulación `Simple` (sección 5.6.2.2).

- **THREADSLEVEL2**: Contiene el número máximo de threads que se asignará al segundo nivel de paralelismo. Aplicable en el modo de simulación `Simple` (sección 5.6.2.2).
- **LIBRARY**: Contiene el identificador de la librería que se va a utilizar para los cálculos. Aplicable en el modo de simulación `Simple` (sección 5.6.2.2).
- **NUMGPU**: Contiene el número de GPUs que se van a poder utilizar para los cálculos. Aplicable en el modo de simulación `Simple` (sección 5.6.2.2).
- **CORES**: Contiene el número de cores instalados en el hardware (aplicable en el modo de simulación `autooptimizado`, descrito en la sección 5.6.2.4).
- **GPUS**: Contiene el número de GPUs instaladas en el hardware (aplicable en el modo de simulación `autooptimizado`, descrito en la sección 5.6.2.4).

5.6.2 Modos de simulación

El simulador se puede emplear tanto para la ejecución de funciones elementales como de modelos completos. La información de tiempos de ejecución que se obtiene en el primer caso sirve para conformar una base de datos de entrenamiento de las funciones, que podrá usarse posteriormente durante la ejecución de modelos cuando se requiera encontrar de forma automática la estrategia óptima de ordenación de cálculos y de asignación de recursos, siguiendo el proceso descrito en la sección 5.5.4.

En PARCSCIM se ofrecen cuatro modos de simulación:

- Entrenamiento
- Simple
- Múltiple
- Autooptimizado

5.6.2.1 Modo de entrenamiento

Este modo se emplea para obtener los tiempos de ejecución de las funciones incorporadas en el simulador, y está diseñado para realizar ejecuciones en batch recorriendo un conjunto de parámetros algorítmicos, *AP*, y juegos de datos, *SCN*, con los que interesa obtener información experimental del rendimiento de las funciones. El usuario especifica mediante una herramienta gráfica los mencionados *AP* y *SCN* los cuales, una vez almacenados en sus correspondientes ficheros, pueden ser usados posteriormente durante la ejecución.

Los valores de los parámetros algorítmicos *AP* que se emplean en el modo de entrenamiento se guardan en el archivo de texto `SCRIPT_Training.scp`. Su estructura es la misma que la mostrada en el listado 5.7 (sección 5.2.7). La diferencia radica en la cabecera ya que, en este caso, el software genera el archivo incluyendo la siguiente información de manera automática:

- `scriptName`: Contiene la cadena `Training`.
- `scriptId`: Contiene el valor 0.
- `scriptTraining`: Toma el valor 1, para indicar que este script será usado únicamente en las ejecuciones cuyo objetivo sea entrenar el sistema.

Los escenarios *SCN* de entrenamiento se guardan en ficheros de texto con la extensión `.sce_t`, y su nombre se forma con la cadena `SCENARIO` seguida de un número entero secuencial generado por el propio software. En el listado 5.6 (sección 5.2.6) se muestra un ejemplo del formato de este tipo de fichero.

La figura 5.25 muestra el proceso seguido por el simulador en el modo de entrenamiento. Se comienza con la lectura de los ficheros que contienen los parámetros algorítmicos y los escenarios de entrenamiento. Se recorren en un bucle los escenarios *SCN* leídos y, para cada uno, se recorren los diferentes parámetros algorítmicos *AP* que se obtienen a partir de la información almacenada en el script de entrenamiento. Para cada combinación de *SCN* y *AP*, se recorren en un bucle todas las funciones f_i incluidas en el simulador y se realizan ejecuciones de cada una de ellas, grabando en cada caso los tiempos de ejecución en la base de datos de resultados.

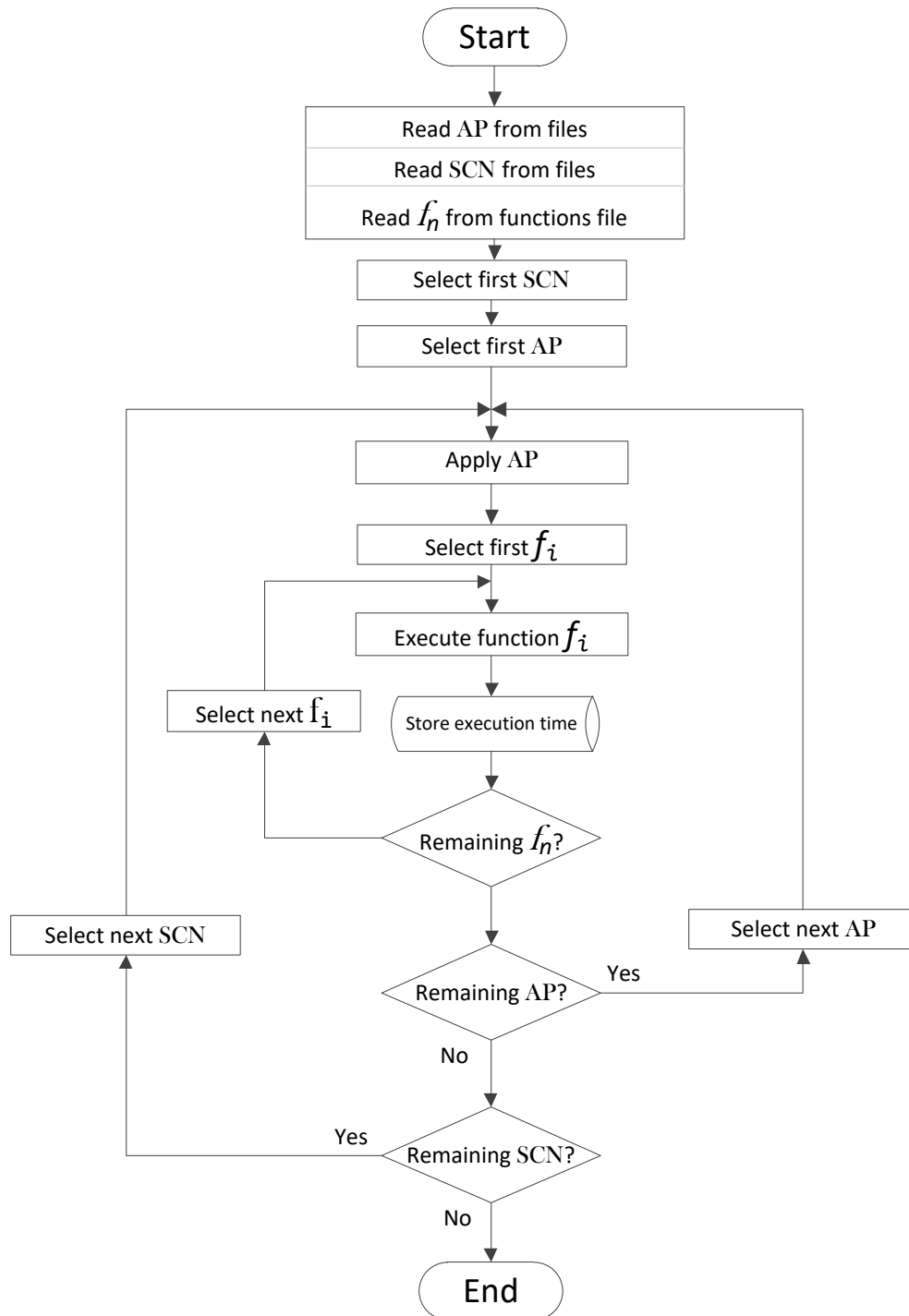


Figura 5.25: Algoritmo de ejecución del simulador en el modo de entrenamiento. Se miden los tiempos de ejecución de todas las funciones para una serie de escenarios *SCN* y parámetros algorítmicos *AP* especificados por el usuario.

5.6.2.2 Modo simple

Este modo permite la simulación de un modelo (o una lista de modelos) usando, para cada uno de ellos, sus propios escenarios *SCN* (como se describió en la sección 5.2.6), y tomando unos valores fijos y únicos de los parámetros algorítmicos *AP* especificados por el usuario. El proceso seguido por el simulador en este modo simple queda descrito en la figura 5.26.

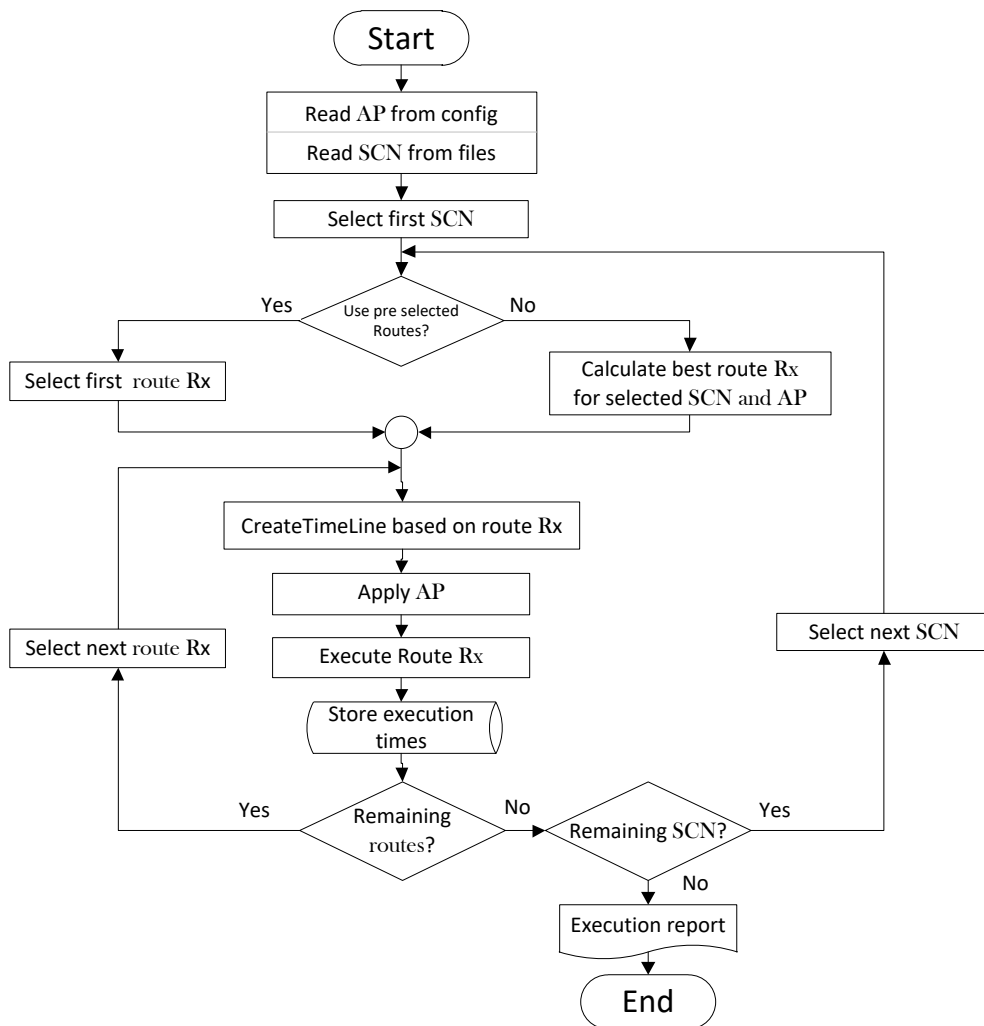


Figura 5.26: Algoritmo de ejecución del simulador en el modo simple sobre un conjunto de escenarios, donde el usuario fija los parámetros algorítmicos *AP*.

Los parámetros algorítmicos fijados por el usuario y guardados en el archivo de configuración de la aplicación son:

- La librería de álgebra lineal a usar.
- El número de threads asignados al primer nivel de paralelismo.
- El número de threads asignados al segundo nivel de paralelismo (su uso dependerá de si la librería admite o no el paralelismo implícito).
- La cantidad de GPUs disponibles (que se usarán o no en función del tipo de librería).

Como vimos en la sección 5.3, un modelo puede ser resuelto siguiendo diferentes estrategias de ordenación y agrupación de cálculos, cada una de las cuales se denominada ruta. En el modo simple PARCSIM puede realizar la simulación siguiendo una determinada ruta seleccionada por el usuario, o un conjunto de ellas, en cuyo caso se puede realizar una comparación empírica del rendimiento de cada una. Pero también es posible indicar al simulador que seleccione automáticamente la mejor ruta teórica en función de los *AP*, *SCN* y los tiempos de ejecución de las funciones obtenidos en el modo de entrenamiento, como se describió en la sección 5.5.4.

5.6.2.3 Modo múltiple

El modo múltiple permite la simulación en modo *batch* de uno o más modelos usando, para cada uno de ellos, sus propios escenarios *SCN*, al igual que ocurre en el modo simple pero, a diferencia de aquel, los valores de los parámetros algorítmicos *AP* se van a generar a partir de la información almacenada en ficheros de texto que representan scripts, como se describió en la sección 5.2.7. De este modo es posible generar una batería de ejecuciones que exploren un conjunto de parámetros algorítmicos, siendo el software el encargado de ordenar los resultados para mostrar los más favorables.

Al igual que en el modo simple, es posible especificar la ruta o conjunto de rutas que el software debe seguir durante la simulación o, alternativamente, indicar que sea el propio software el que calcule la óptima en cada iteración, de acuerdo a los valores de *AP* y *SCN* en cada momento. El proceso seguido por el simulador en este modo múltiple queda descrito en la figura 5.27.

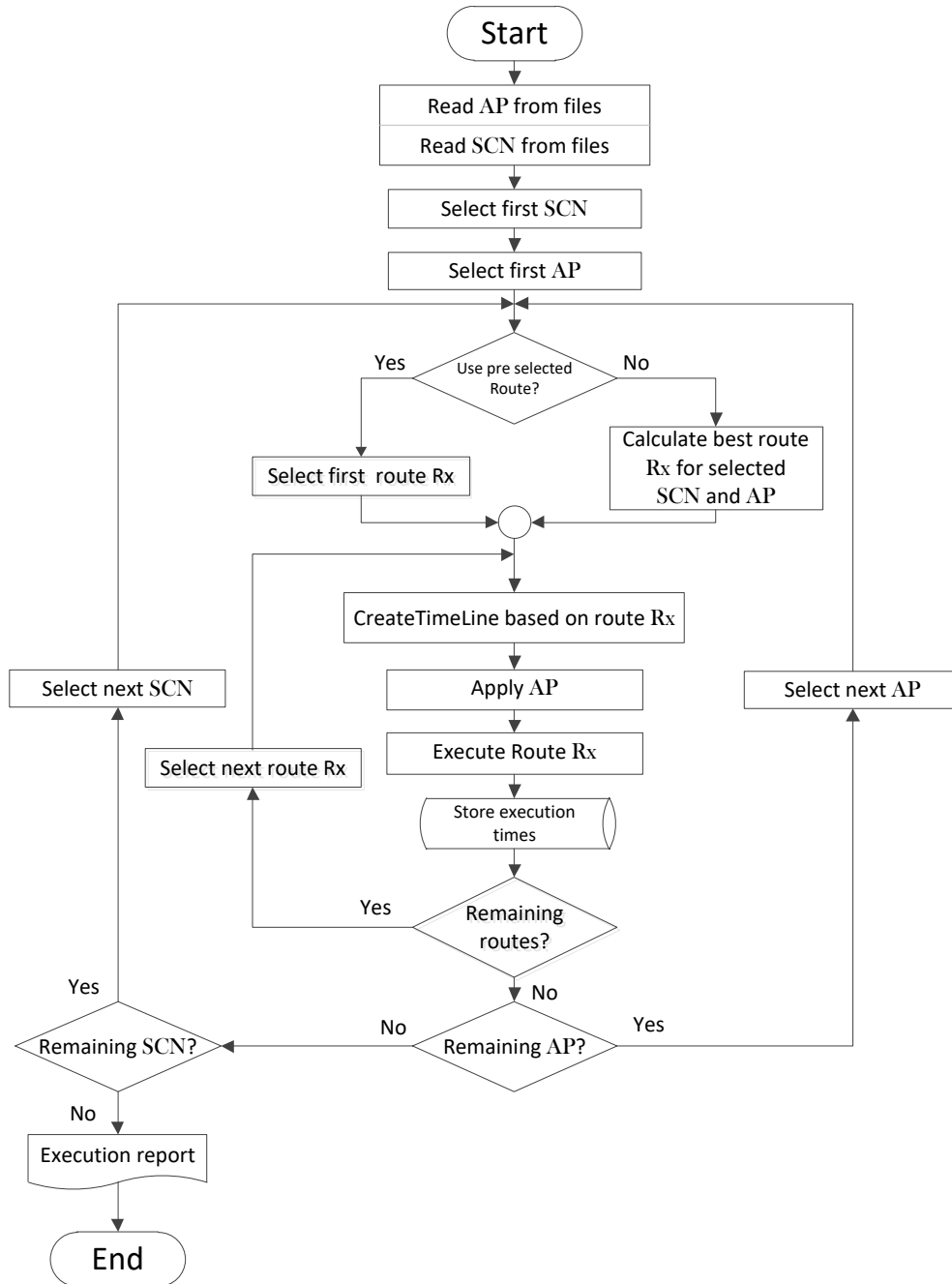


Figura 5.27: Algoritmo de ejecución del simulador en el modo múltiple, con ejecuciones sobre un conjunto de escenarios *SCN* y con parámetros algorítmicos *AP* generados a partir de los ficheros de scripts.

5.6.2.4 Modo autooptimizado

El modo autooptimizado permite simular un modelo, o una lista de ellos, usando en cada uno sus propios escenarios *SCN* (como se describió en la sección 5.2.6). Los parámetros algorítmicos y la rama de ejecución son seleccionados automáticamente por el propio software en función de la configuración del hardware, en concreto del número de cores y de GPUs. El proceso seguido por el simulador en este modo se describe en la figura 5.28.

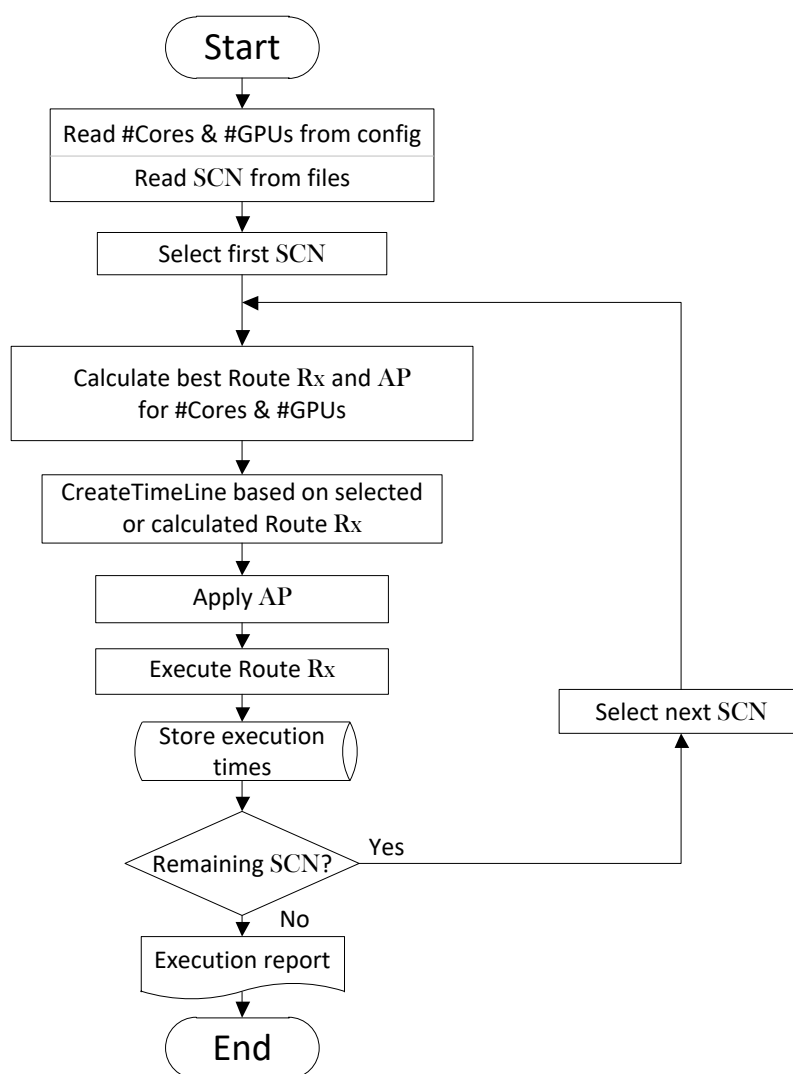


Figura 5.28: Algoritmo de ejecución del simulador en el modo autooptimizado, donde el software calcula los parámetros algorítmicos AP y la ruta R_x que mejor se adaptan al hardware disponible.

5.6.3 Informe de ejecución

El proceso de simulación genera información durante la ejecución que puede ser de interés para el usuario. Esta información se muestra en la consola del sistema operativo. Se trata de mensajes que informan de la etapa en la que se encuentra el proceso, los grupos que se están ejecutando en un momento determinado, los parámetros algorítmicos que se aplican o detalles del proceso de autooptimización, cuando éste se produce. En caso de no encontrarse errores, el software muestra, para cada tamaño de problema o escenario, un resumen de los tiempos de ejecución obtenidos en la simulación de cada modelo junto a los parámetros algorítmicos aplicados. Estos tiempos se muestran ordenados de menor a mayor, de manera que los primeros son los que ofrecen mejor rendimiento. La figura 5.29 muestra la sección final de un informe obtenido por consola en el que podemos identificar:

- ① Los escenarios, con detalles del tipo y tamaño de las matrices.
- ② Los tiempos de ejecución, ordenados de menor a mayor.
- ③ Los valores de los parámetros algorítmicos aplicados.
- ④ Las rutas con las que se han conseguido dichos tiempos. Se muestran los identificadores de los grupos dentro de una cadena que representa un orden de ejecución de los cálculos. Esta cadena sigue la codificación descrita en la sección 5.3, y que recordamos a continuación:
 - El símbolo + separa los códigos de los grupos que se han resuelto en paralelo, en una misma etapa de tiempo.
 - El símbolo – separa etapas de tiempo.
- ⑤ En caso de haberse completado la simulación de más de un escenario, se muestra un resumen con los mejores tiempos de ejecución obtenidos con cada uno de ellos, y los parámetros algorítmicos y la ruta utilizados.

Este informe también se graba en un fichero de texto cuyo nombre se forma con la cadena que se definió en el archivo de configuración (sección 5.6.1) seguido del nombre del modelo y del escenario.

SCRIPT_1	# cycles	Branch	steps	m_type	m_spars	m_rows	m_cols	(s)	ompth1	ompth2	library	gpu	groups	sequence
	6	1	48	5	2	10	50	0.023458	3	1	1	1	1	1-2+3-4+5-6+7-8
	1	1	52	6	2	10	50	0.023701	1	1	1	1	1	1-2-3-4-5+6+7-8
	7	1	52	6	2	10	50	0.024458	3	1	1	1	1	1-2-3-4-5+6+7-8
	5	1	52	6	2	10	50	0.025187	2	1	1	1	1	1-2-3-4-5+6+7-8
	3	1	52	6	2	10	50	0.026636	1	3	1	1	1	1-2-3-4-5+6+7-8
	4	1	48	5	2	10	50	0.027876	2	1	1	1	1	1-2+3-4+5-6+7-8
	2	1	48	5	2	10	50	0.028536	1	3	1	1	1	1-2+3-4+5-6+7-8
	8	1	48	5	2	10	50	0.032937	4	1	1	1	1	1-2+3-4+5-6+7-8
	9	1	52	6	2	10	50	0.034743	4	1	1	1	1	1-2-3-4-5+6+7-8
	0	1	48	5	2	10	50	0.039448	1	1	1	1	1	1-2+3-4+5-6+7-8

Best parameters are in SCRIPT_1.scip iteration 6 calculated in 0.023458 ms
Information has been saved in the database.

End: 0 ###
No errors Log BEST information will be saved in: ./BEST.txt

Model	scenario	Branch	steps	(s)	ompth1	ompth2	library	gpu	groups	sequence
demo	second	6	5	0.019892	3	1	1	1	0	(G1) (G2+G3) (G4+G5) (G6+G7) (G8)
demo	first	6	5	0.023458	3	1	1	1	0	(G1) (G2+G3) (G4+G5) (G6+G7) (G8)

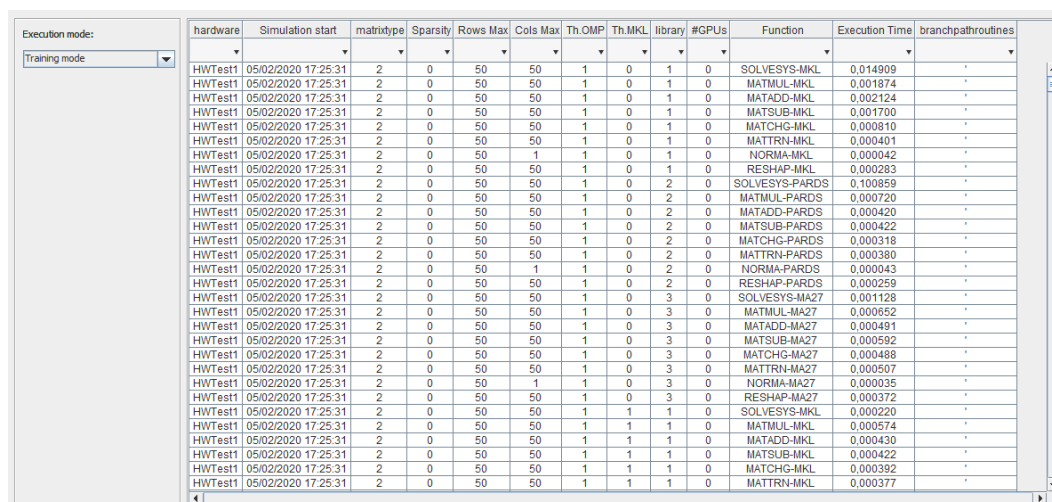
Figura 5.29: Información producida por el software durante la simulación, con los tiempos de ejecución ordenados de menor a mayor.

5.7 Herramientas

El software PARCSIM ofrece herramientas gráficas que permiten, por un lado, analizar los tiempos de ejecución obtenidos durante las simulaciones y, por otro, visualizar de manera interactiva el proceso de autooptimización implementado en el simulador.

5.7.1 Análisis de resultados: vista tabular

El interfaz gráfico del simulador ofrece una herramienta visual para acceder a los tiempos de ejecución almacenados en la base de datos y mostrar los registros en formato de tabla, como se muestra en la figura 5.30. Es posible seleccionar la información referida a un modo de simulación o un modelo concreto. Para ello se usan controles de selección desplegable donde se muestran los modos de simulación descritos en 5.6.2 y los modelos de los que hay información disponible.



hardware	Simulation start	matrixtype	Sparsity	Rows Max	Cols Max	Th.OMP	Th.MKL	library	#GPUs	Function	Execution Time	branchpaths/routines
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	1	0	SOLVESYS-MKL	0.014909	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	1	0	MATMUL-MKL	0.001874	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	1	0	MATADD-MKL	0.002124	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	1	0	MATSUB-MKL	0.001700	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	1	0	MATCHG-MKL	0.000810	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	1	0	MATTRN-MKL	0.000401	'
HWTest1	05/02/2020 17:25:31	2	0	50	1	1	0	1	0	NORMA-MKL	0.000042	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	1	0	RESHAP-MKL	0.000283	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	2	0	SOLVESYS-PARDS	0.100859	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	2	0	MATMUL-PARDS	0.000720	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	2	0	MATADD-PARDS	0.000420	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	2	0	MATSUB-PARDS	0.000422	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	2	0	MATCHG-PARDS	0.000318	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	2	0	MATTRN-PARDS	0.000380	'
HWTest1	05/02/2020 17:25:31	2	0	50	1	1	0	2	0	NORMA-PARDS	0.000043	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	2	0	RESHAP-PARDS	0.000259	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	3	0	SOLVESYS-MA27	0.001128	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	3	0	MATMUL-MA27	0.000652	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	3	0	MATADD-MA27	0.000491	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	3	0	MATSUB-MA27	0.000592	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	3	0	MATCHG-MA27	0.000488	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	3	0	MATTRN-MA27	0.000507	'
HWTest1	05/02/2020 17:25:31	2	0	50	1	1	0	3	0	NORMA-MA27	0.000035	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	3	0	RESHAP-MA27	0.000372	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	1	1	0	SOLVESYS-MKL	0.000220	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	1	1	0	MATMUL-MKL	0.000574	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	1	1	0	MATADD-MKL	0.000430	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	1	1	0	MATSUB-MKL	0.000422	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	1	1	0	MATCHG-MKL	0.000392	'
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	1	1	0	MATTRN-MKL	0.000377	'

Figura 5.30: Consulta de información de los tiempos de ejecución en formato tabular.

Una vez mostrados los datos, se pueden aplicar filtros adicionales sobre la información incluida en cada columna. Para ello se usan los campos de selección que aparecen debajo de la cabecera de la tabla, como muestra la figura 5.31.

hardware	Simulation start	matrixtype	Sparsity	Rows Max	Cols Max	Th.OMP	Th.MKL	library	#GPUs	Function	Execution Time	branchpathroutines
										MATADD-MA27		
HWTest1	05/02/2020 17:25:31	2	0	50	50	1	0	3	0	MATADD-MA27	0.000491	*
HWTest1	05/02/2020 17:25:46	2	50	50	50	1	0	3	0	MATADD-MA27	0.000914	*
HWTest1	05/02/2020 17:26:01	2	0	100	100	1	0	3	0	MATADD-MA27	0.001840	*
HWTest1	05/02/2020 17:26:18	2	50	100	100	1	0	3	0	MATADD-MA27	0.002137	*
HWTest1	05/02/2020 17:26:35	2	0	500	500	1	0	3	0	MATADD-MA27	0.055100	*
HWTest1	05/02/2020 17:26:56	2	50	500	500	1	0	3	0	MATADD-MA27	0.078766	*
HWTest1	05/02/2020 17:27:30	2	0	1000	1000	1	0	3	0	MATADD-MA27	0.290119	*
HWTest1	05/02/2020 17:28:26	2	50	1000	1000	1	0	3	0	MATADD-MA27	0.193544	*

Figura 5.31: Consulta de información de los tiempos de ejecución en formato tabular aplicando un filtro para obtener datos de una función específica.

5.7.2 Análisis de resultados: vista gráfica

El software permite generar gráficos de líneas que se pueden usar para comparar los tiempos de ejecución obtenidos con diferentes parámetros algorítmicos para un determinado tipo de matrices y hardware seleccionados por el usuario mediante los correspondientes controles desplegados. Como se observa en la figura 5.32, se muestra una serie gráfica para cada conjunto de parámetros algorítmicos. El eje X representa los tamaños de las matrices y el eje Y los tiempos de ejecución. Al seleccionar cualquiera de las series se muestra un grafo que representa la ruta de ejecución empleada en la obtención del dato escogido.

La información mostrada en el gráfico se detalla también en una tabla en la parte de información del visor. Es posible seleccionar con el ratón cualquiera de los valores de dicha tabla. Tras ello el gráfico se actualizará para resaltar la serie a la que pertenece el dato seleccionado.

También se dispone de herramientas que permiten ampliar cualquier parte del gráfico, modificar el formato y color de las series, así como generar una imagen en formato PNG (*Portable Network Graphics*) o enviar el gráfico a la impresora.

Una utilidad similar se encuentra disponible para visualizar en forma de gráfica los tiempos de ejecución obtenidos en el modo de entrenamiento. En este caso se deben seleccionar, además del hardware y el tipo de matriz, la función que se desea visualizar, como se muestra en la figura 5.33. Para todos ellos se dispone de un control de lista desplegable que muestran los valores disponibles.

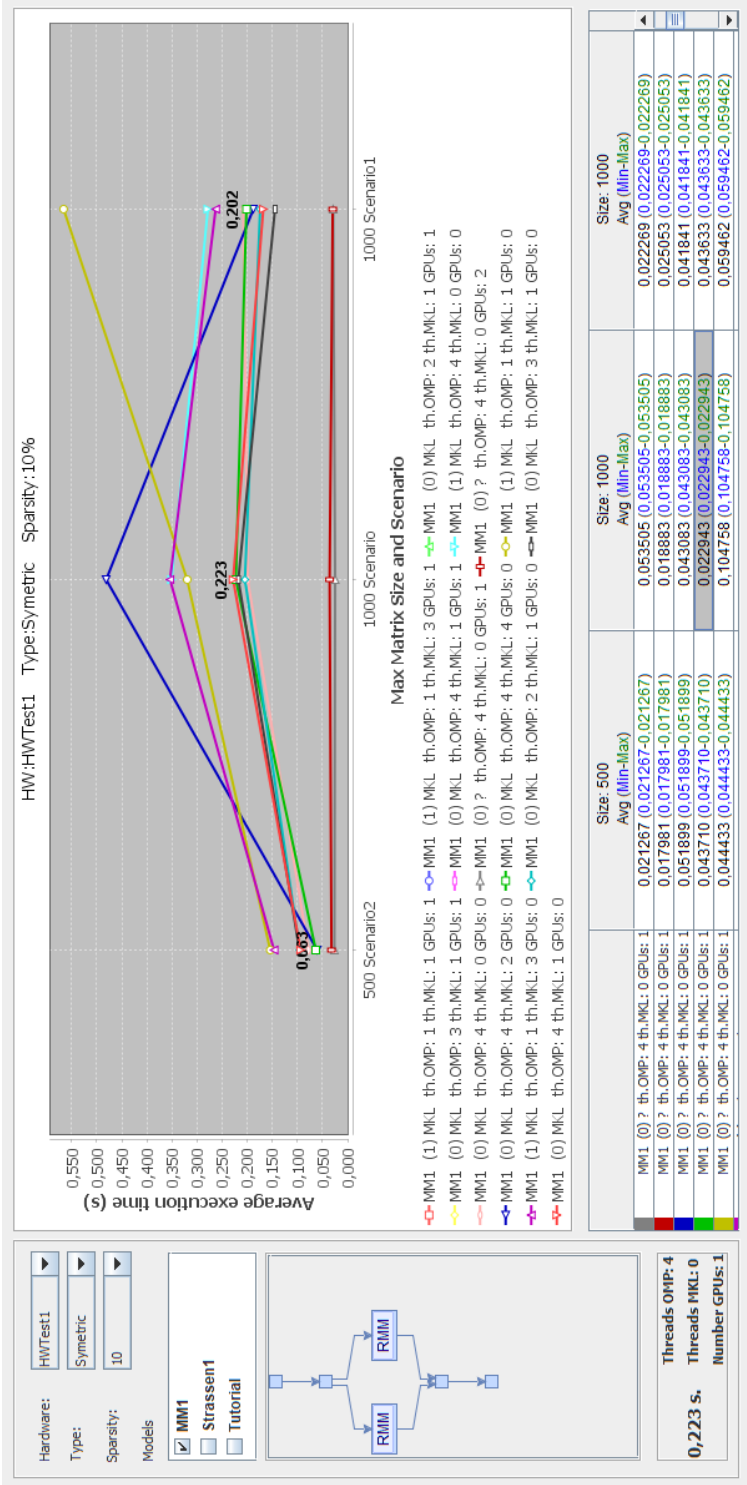


Figura 5.32: Gráfico de tiempos de ejecución obtenidos al simular un modelo con un tipo de matrices seleccionado por el usuario. Se obtiene una serie para cada combinación de parámetros algorítmicos.

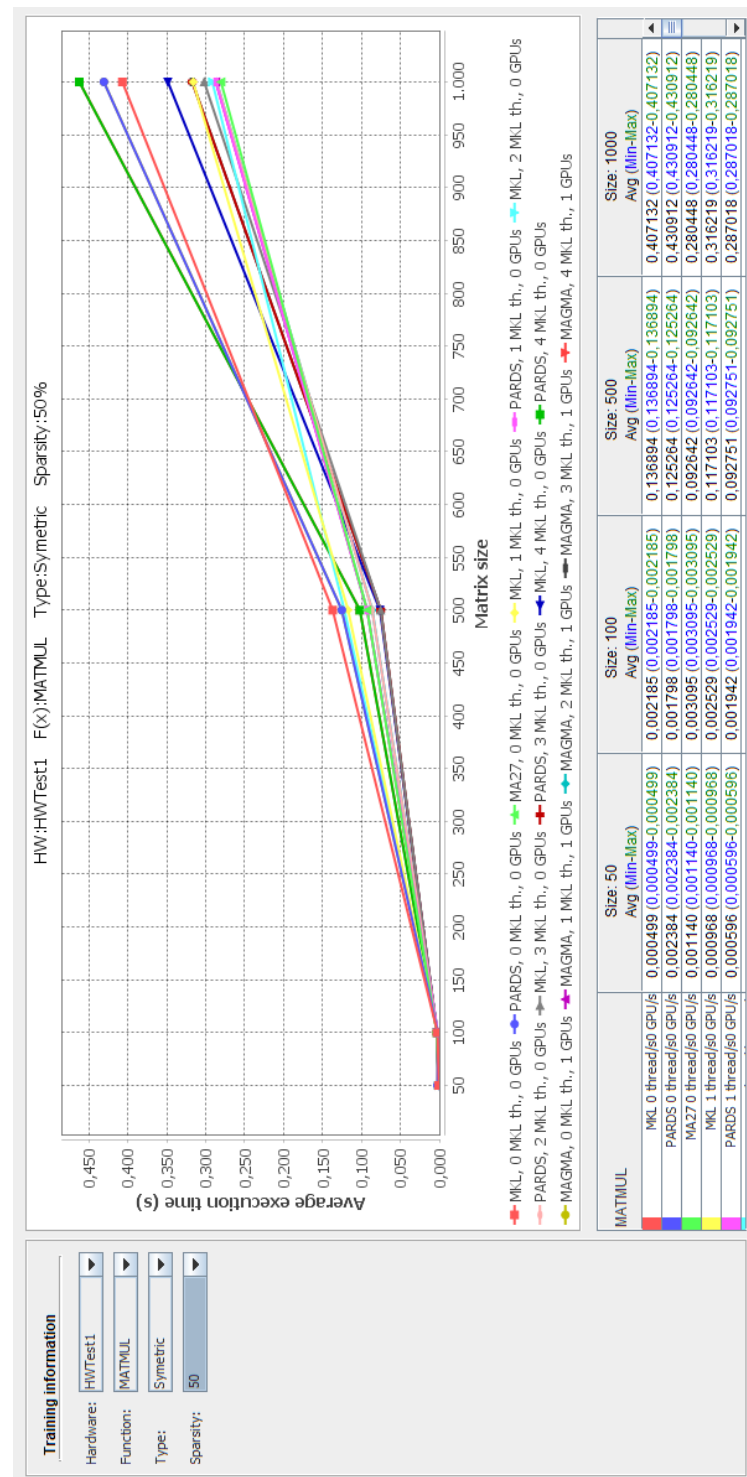
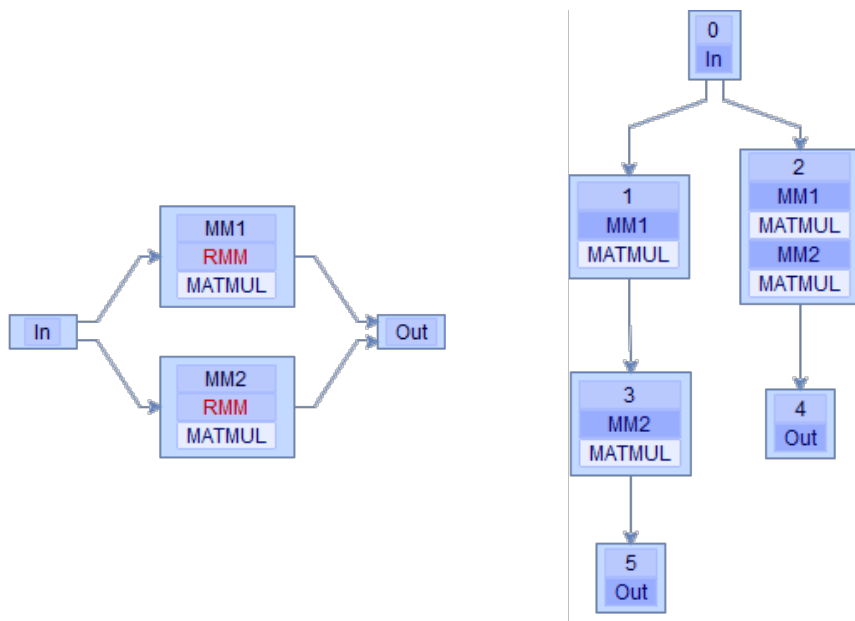


Figura 5.33: Análisis del tiempo de ejecución de una función en el simulador. El gráfico muestra información de entrenamiento para diferentes parámetros algorítmicos y tamaños de matrices.

5.7.3 Autooptimizador interactivo

En la sección 5.5.4 se describe el proceso implementado en el simulador en el que, partiendo del árbol de rutas correspondiente a un modelo, se estima el tiempo de ejecución asociado a cada rama, permitiendo con ello encontrar la que ofrece la resolución más rápida en una determinada plataforma hardware para un determinado tamaño de problema.

El interfaz gráfico de PARCSIM incorpora una herramienta interactiva que permite visualizar dicho procedimiento de estimación. De esta manera el usuario puede experimentar variando las prestaciones del hardware y comprobar cómo el software decide cuál es la ruta más rápida, y cuál es la librería de cómputo y los parámetros algorítmicos que permiten dicho rendimiento. A continuación se ilustra, mediante imágenes extraídas del interfaz gráfico del simulador, un ejemplo de uso de esta funcionalidad aplicada al modelo representado en la figura 5.34(a).



(a) Vista del modelo, con dos grupos *MM1* y *MM2*, cada uno de los cuales incluye una rutina de usuario *RMM* con una multiplicación de matrices.

(b) Vista del árbol de rutas. El nodo 2 propone la ejecución simultánea de *MM1* y *MM2*.

Figura 5.34: Representación en la interfaz gráfica GUI del simulador de una resolución de dos multiplicaciones de matrices.

Dicha imagen muestra la representación en PARCSIM de un modelo sencillo compuesto por cuatro grupos In, MM1, MM2 y Out. MM1 y MM2 ejecutan una multiplicación de matrices cada uno. Como se describió en la sección 5.5.2, el software genera un árbol que incluye todas las rutas que permiten resolver un determinado modelo. En este sencillo ejemplo, el árbol obtenido se muestra en la figura 5.34(b), y contiene únicamente dos rutas. La primera ejecuta en secuencia los dos grupos MM1 y MM2. La segunda ruta propone una ejecución en paralelo pues, como se observa en el grafo del modelo, no existe dependencia entre ellos.

Además del modelo, es necesario indicar el tamaño y tipo de las matrices que se van a multiplicar en los grupos MM1 y MM2. En PARCSIM esta información se denomina escenario. Es posible crear más de un escenario para cada modelo, y estos se pueden visualizar en el visor gráfico, como se muestra en la figura 5.35, donde se observan tres escenarios con diferentes tamaños de matrices.

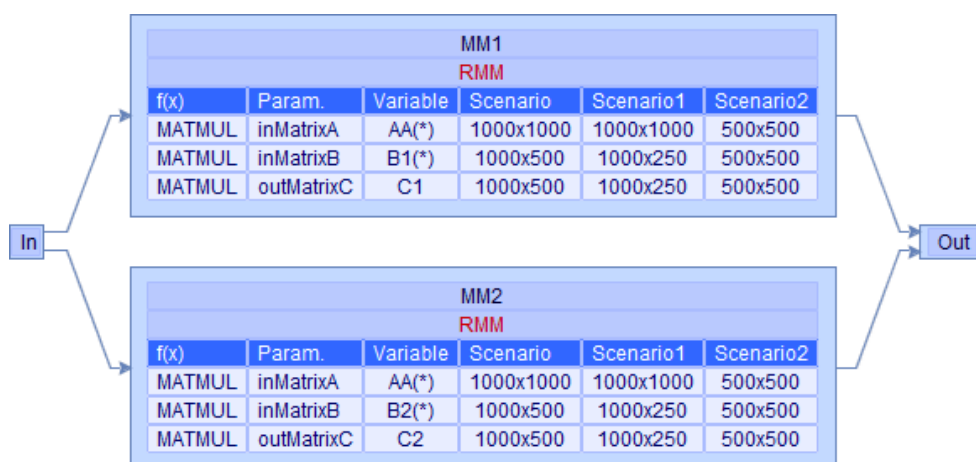


Figura 5.35: Vista del modelo de la figura 5.34(a) incluyendo información de los escenarios.

Una vez creado el árbol de rutas y uno o más escenarios, es posible ejecutar la herramienta interactiva de autooptimización, donde se debe indicar un escenario concreto y el número de cores y GPUs que configuran la plataforma hardware. En la figura 5.36 se observa la selección de dos cores y ninguna GPU. El campo Number of Paths se usa para indicar a PARCSIM el número de rutas que queremos obtener. El software lista las que pueda encontrar, comenzando por la mejor de ellas (la que ofrece el menor tiempo de ejecución).

En la ventana Autotuned fastest paths se visualiza en forma de texto la ruta más rápida y el tiempo de ejecución asociado. El visor gráfico resalta dicha ruta, y muestra dentro de cada grupo el número de cores usados y la librería de cómputo sugerida.

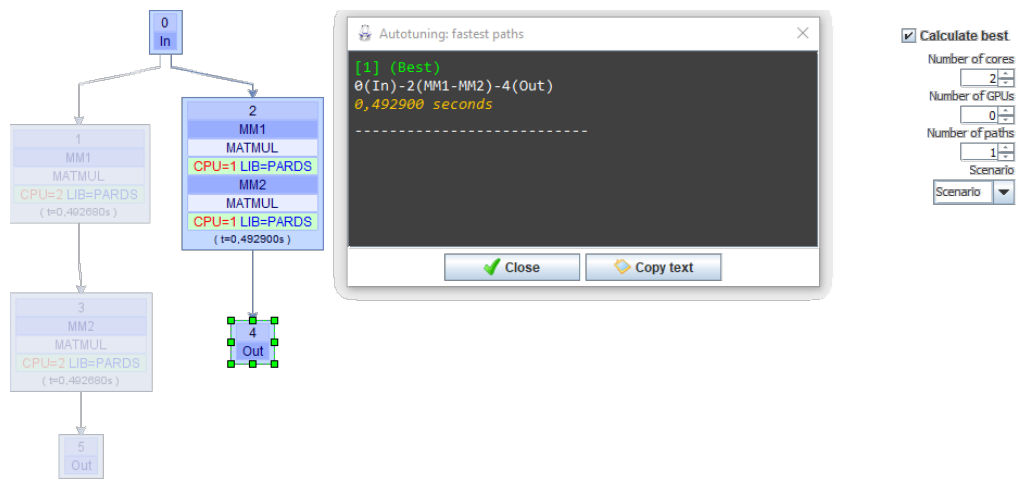


Figura 5.36: Herramienta de autooptimización: mejor ruta y librería de cómputo propuestas para una plataforma con dos cores y un escenario seleccionado.

Aumentando a dos el número de rutas a visualizar conseguimos el resultado de la figura 5.37.

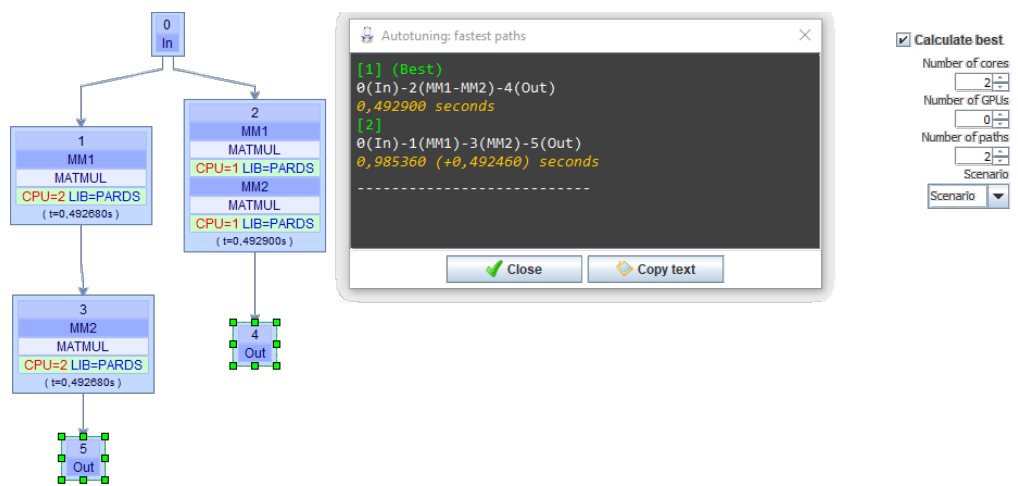


Figura 5.37: Herramienta de autooptimización: las dos mejores rutas para una plataforma con dos cores y un escenario seleccionado.

Observamos que en la ventana de texto aparecen las dos rutas encontradas junto a la diferencia de tiempos entre ellas (en este caso particular una ejecución en secuencia de los grupos supone 0.492460 segundos más que una ejecución paralela).

Si modificamos nuevamente el tipo de hardware para simular que disponemos de una GPU, el software comprueba si el uso de una librería que haga uso de dicha GPU permite obtener mejores resultados. En este caso encuentra una mejora en los tiempos de ejecución teóricos y sugiere emplear la librería MAGMA, como vemos en la figura 5.38.

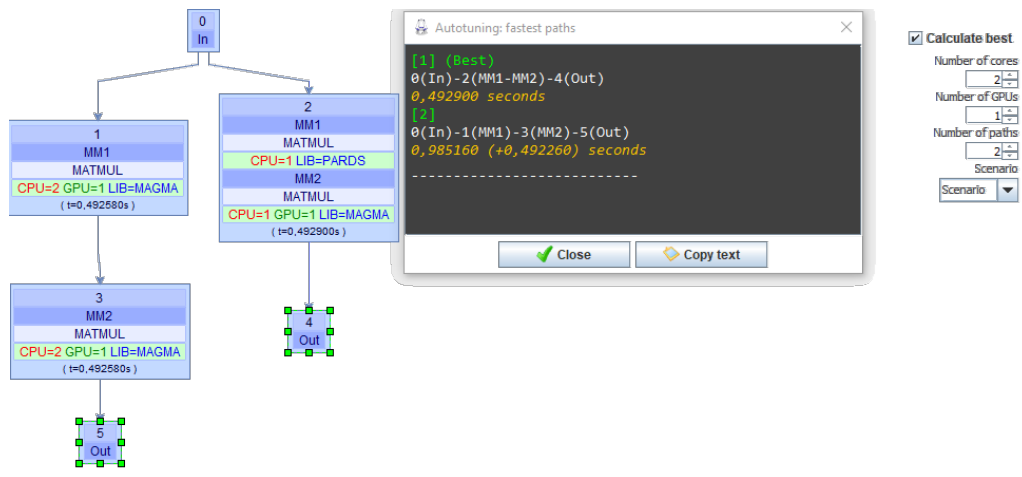


Figura 5.38: Herramienta de autooptimización: mejor ruta y librería de cómputo propuestas para plataformas con dos cores y una GPU, para un escenario concreto seleccionado.

Mediante esta utilidad, modificando los valores que representan las configuraciones del hardware y el tamaño del problema, un usuario puede obtener dinámicamente un análisis de la mejor asignación de cálculos a las unidades de cómputo, y de la librería que ofrece un mejor rendimiento en cada situación.

5.8 Arquitectura del software

La arquitectura del software está representada a alto nivel mediante el diagrama mostrado en la figura 5.39.

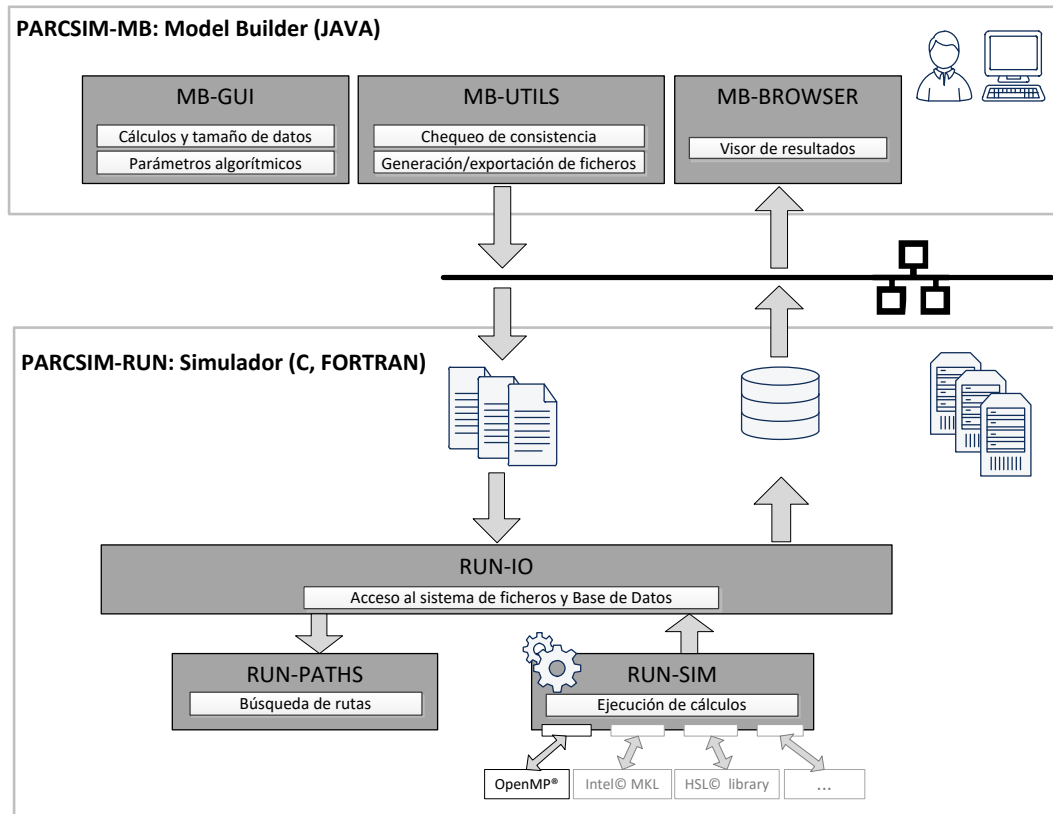


Figura 5.39: Arquitectura a alto nivel del software PARCSIM.

El simulador está formado por dos componentes:

- El componente PARCSIM-MB (Model Builder) ofrece un asistente gráfico para la gestión del simulador compuesto por:
 - El módulo MB-GUI, que facilita el uso del software a usuarios no expertos, evitando la creación manual de los ficheros que rigen el funcionamiento del simulador. Está desarrollado en JAVA y proporciona un editor visual para capturar los grupos de cálculos y sus dependencias. También permite especificar el tamaño del conjunto de datos que se utilizará en las simulaciones.
 - El módulo de utilidades MB-UTILS contiene la lógica que permite comprobar que el modelo introducido en el sistema está bien construido, como paso previo a la generación de los ficheros de texto necesarios para la simulación.

- El módulo de acceso a la base de datos que almacena los tiempos de ejecución, MB-BROWSER, permite al usuario comparar los rendimientos obtenidos en diferentes simulaciones.
- El componente PARCSIM-RUN es el núcleo del simulador y está formado por tres módulos con las siguientes funcionalidades:
 - RUN-IO: Acceso al sistema de archivos para la lectura de los ficheros de texto creados usando el módulo PARCSIM-MB, y que especifican el problema a simular, la dimensión de los datos y los parámetros algorítmicos. La finalidad y la estructura de estos ficheros han sido descritas en los diferentes apartados de la sección 5.2. El módulo RUN-IO también gestiona la grabación de los resultados en la base de datos y la generación del informe de ejecución.
 - RUN-PATHS: Elaboración, a partir del grafo del modelo, del árbol que representa las diferentes rutas que resuelven el problema. Incluye el proceso de estimación incluido en la herramienta de autooptimización que selecciona la ruta y los valores de los parámetros algorítmicos que ofrecen el menor tiempo de ejecución.
 - RUN-SIM: Simulación del modelo mediante la ejecución ordenada de las funciones incluidas en una o varias rutas que habrán sido seleccionadas previamente, bien por el usuario, bien por la función de autooptimización.

5.9 Conclusiones

En este capítulo se ha presentado el software de simulación PARCSIM (Parallel C-omputations SIM-ulator), una herramienta desarrollada junto a esta tesis, y que permite a un usuario capturar problemas científicos en forma de un conjunto ordenado de subsistemas determinados que se resuelven aplicando rutinas de álgebra matricial. Este es el caso de las implementaciones computacionales del análisis cinemático de sistemas multicuerpo, donde se trabaja con un conjunto ordenado de subestructuras de cinemática que se deben resolver.

El software ofrece al usuario herramientas de análisis que exploran la aplicación de técnicas de programación paralela y de elaboración de árboles que representan las diferentes alternativas de agrupación y ordenación de los cálculos. Incluye modos de ejecución puramente experimentales basadas en simulaciones que buscan las mejores combinaciones de parámetros algorítmicos, pero también ofrece una herramienta de autooptimización que selecciona la mejor configuración de la ejecución, realizando para ello una estimación teórica basada en los tiempos de ejecución reales de las funciones elementales y en los recursos del hardware disponibles.

Capítulo 6

Experimentos

El presente capítulo describe algunos ejemplos de aplicación del simulador desarrollado en esta tesis. Se muestran casos de uso agrupados en dos disciplinas: la simulación del análisis cinemático de sistemas multicuerpo y la aplicación a la optimización de rutinas de álgebra lineal. En cada caso se presenta una breve introducción teórica del problema a simular, su representación como un modelo manipulable por el simulador y, finalmente, el conjunto de experimentos encaminados a determinar el conjunto de parámetros de paralelismo y librerías que ofrecen los menores tiempos de ejecución para la plataforma hardware seleccionada.

En el área de sistemas multicuerpo se analizan un robot manipulador de cinemática paralela y un sistema de suspensión de un camión. Para su aplicación a rutinas de álgebra lineal se ha seleccionado la multiplicación de matrices, donde se estudiará su resolución mediante su descomposición por bloques y aplicando el algoritmo de Strassen.

6.1 Aplicación del simulador al análisis cinemático de sistemas multicuerpo

Como se describió en el capítulo 2, el análisis estructural es una herramienta de la ingeniería que estudia la división de un sistema mecánico en subsistemas. Esta metodología permite explorar alternativas de resolución en paralelo de problemas que tienen un tamaño más reducido que el original, consiguiendo simulaciones más eficientes, en especial cuando se aprovechan los recursos que ofrecen las modernas plataformas de hardware multicore+multiGPU. Además, empleando una formulación cinemática computacional basada en ecuaciones de grupo, se pueden elaborar los algoritmos que posteriormente se introducirán en el simulador, como veremos en las siguientes secciones.

No obstante, dado que el simulador trabaja con bloques de cómputo, este admite incluir cualquier algoritmo de resolución de MBS, con cualquier tipo de formulación cinemática, global o topológica, y cualquier tipo de coordenadas.

6.1.1 Plataforma de Stewart

Una plataforma de Stewart es un tipo de robot paralelo que consta de una superficie fija sobre la que se apoyan seis actuadores independientes, como muestra la figura 6.1(a). El terminal de la plataforma dispone de seis grados de libertad, por lo que se puede desplazar en las tres direcciones del espacio, y rotar respecto a esas tres mismas direcciones.

De las posibles configuraciones disponibles para este tipo de mecanismos, el elegido para su estudio en este trabajo utiliza actuadores rotatorios que proporcionan un movimiento rotacional a seis manivelas. El movimiento de las manivelas se traslada mediante juntas esféricas a unas barras conectadas, también con juntas esféricas, directamente a un punto fijo de la plataforma móvil. El mecanismo se resuelve mediante cinemática inversa, es decir, que a partir de la posición que se define para el terminal móvil se determinan los giros que deben describir cada una de las manivelas.

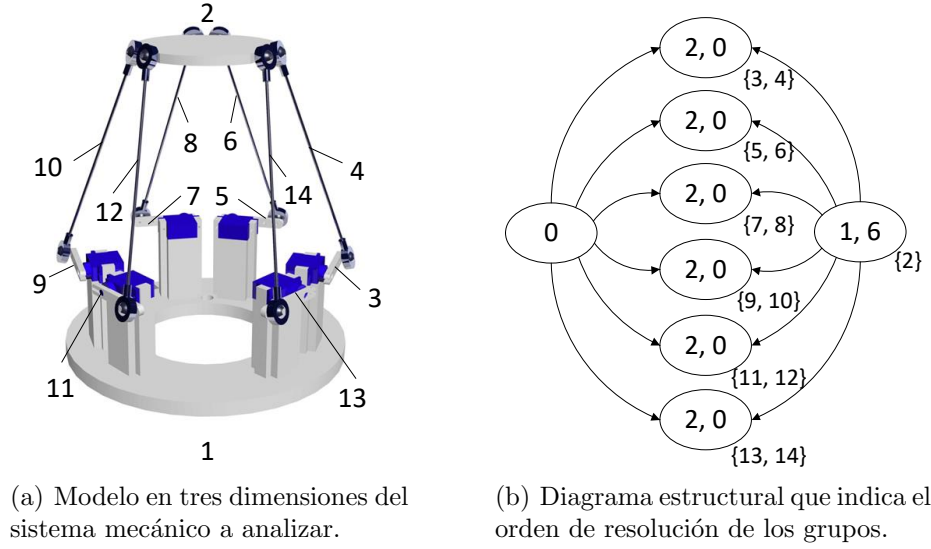


Figura 6.1: Plataforma de Stewart.

Un análisis estructural del mecanismo permite dividirlo en siete grupos estructurales: el terminal, elemento 2 en la figura 6.1(a), y seis grupos formados por conjuntos manivela-barra (los pares de elementos $\{3,4\}$, $\{5,6\}$, $\{7,8\}$, $\{9,10\}$, $\{11,12\}$ y $\{13,14\}$). La resolución de dichos grupos manivela-barra requiere que se haya resuelto el elemento del terminal, como muestra el diagrama estructural de la figura 6.1(b). El elemento 1 es el punto fijo de la plataforma, que no necesita resolución.

Dado que se trata de un robot de cinemática paralela, cada uno de los conjuntos manivela-barra presenta la misma topología y, por tanto, se resuelve de la misma manera. Por consiguiente, para su implementación en el simulador necesitamos dos tipos de rutinas de análisis, la que permite resolver el terminal y la que resuelve los grupos manivela-barra.

La figura 6.2 muestra una rutina para resolver un grupo compuesto por una junta de rotación, una esférica y una cardan, y que corresponde con la estructura que forman los grupos manivela-barra. Cabe indicar aquí que, para resolver la cinemática de este grupo, cada barra unida mediante juntas esféricas al terminal y a la manivela introduce un grado de libertad redundante que se ha eliminado bloqueando uno de los giros relativos entre la propia barra y el terminal, lo que equivale a modelar esa unión como una junta cardan.



Figura 6.2: Representación en el simulador de la rutina SG_KINEM_REC de resolución de un sistema mecánico con una configuración REC, formada por una junta de rotación (R), una esférica (E) y una cardan (C), válida para resolver un grupo manivela-barra de una plataforma de Stewart.

La creación de dicha rutina, SG_KINEM_REC, se realiza siguiendo el procedimiento descrito en la sección A.8.5. Muestra la estructura general del análisis cinemático computacional descrito en el algoritmo 1 de la sección 2, y se compone a su vez de otras tres rutinas encargadas de resolver cada uno de los problemas:

- Problema de la posición, mediante la rutina SG_POS. Esta se ejecuta en dos ocasiones, simulando el número de iteraciones que se estiman necesarias para converger cuando se ejecuta el método iterativo Newton-Raphson (ecuación 2.3) hasta alcanzar una tolerancia asignada.
- Una vez resultas las posiciones, se ejecuta la rutina SG_VEL, que formula la solución de las velocidades.
- Finalmente la rutina SG_ACE resuelve el problema de las aceleraciones.

Para resolver el terminal creamos la rutina SG_KINEM_6C, correspondiente a un mecanismo que incluye seis juntas tipo cardan. Como se puede observar en la figura 6.3, su estructura es similar a la anterior, con la solución de los problemas de posición, velocidad y aceleración, pero añadiendo una cuarta rutina (SG_EXTRA_POIS), encargada de resolver otros puntos de interés dentro del grupo estructural que representa al terminal.

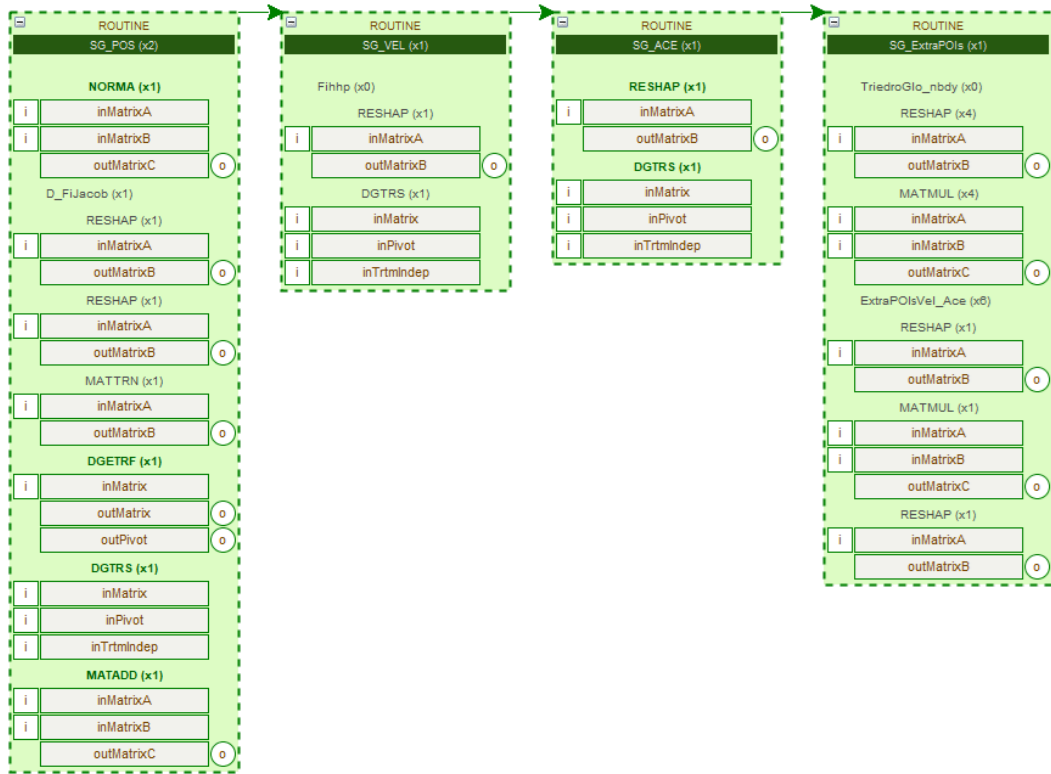


Figura 6.3: Representación en el simulador de la rutina SG_KINEM_6C, correspondiente a seis juntas de tipo cardan (6C), para la resolución de un sistema mecánico con la topología del terminal de una plataforma de Stewart.

Una vez creadas las rutinas, se pueden introducir en el simulador todos los grupos que forman el modelo de la plataforma de Stewart, al que hemos nombrado como MBS-STEWARD y que se encuentra representado en la figura 6.4. En el grafo se observan las mismas dependencias que las mostradas en el diagrama estructural de la figura 6.1(b), en el que los pares manivela-barra de aquel corresponden a los grupos SG_MB_1 al SG_MB_6 del modelo.

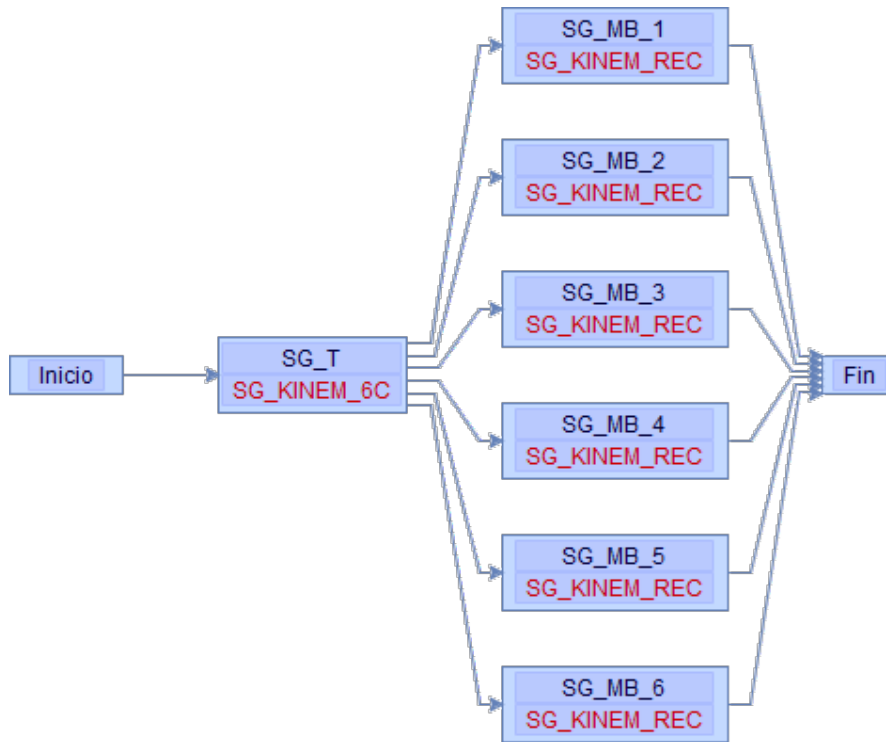


Figura 6.4: Representación gráfica en el simulador del modelo MBS-STEWART para la resolución de una plataforma de Stewart.

El simulador incorpora una herramienta que analiza las dependencias entre los grupos y construye un árbol que representa las diferentes alternativas de ordenación y agrupación de los cálculos. En la figura 6.5 se observan todas las posibilidades para la resolución de una plataforma de Stewart formada por un terminal y seis manivelas.

Las siguientes secciones muestran ejecuciones del simulador que permitirán encontrar las mejores combinaciones de parámetros algorítmicos aplicables a la resolución de la plataforma de Stewart. Cabe destacar que cada ejecución resuelve todos los grupos del modelo, y que es posible repetir el proceso un determinado número de veces que dependerá del tamaño del paso de tiempo y del tiempo total que se quiera simular. En el ejemplo de la biela-manivela unos valores habituales podrían ser los que simulan el giro de la misma entre 0 y 360° con incrementos de 0.1°. Es decir, unas 3600 ejecuciones solo en análisis. En síntesis óptima podrían ser más. Los tiempos de ejecución que se ofrecen en esta sección corresponden a 10 ejecuciones, y se expresan en segundos.

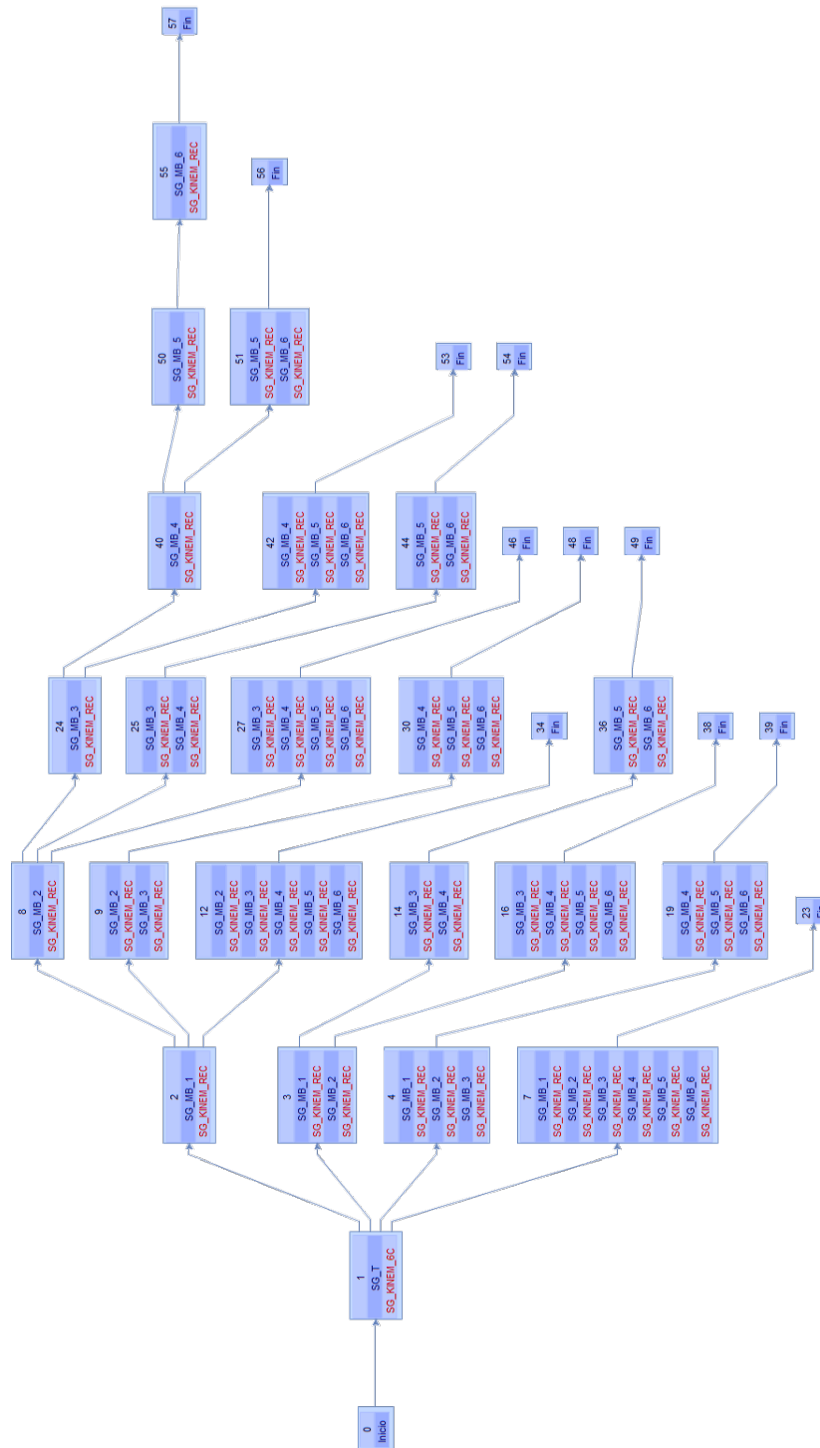


Figura 6.5: Árbol de rutas que representa las alternativas de ordenación y agrupación de cálculos para resolver una plataforma de Stewart como la representada en el modelo de la figura 6.4.

Del análisis estructural del mecanismo de la figura 6.1(a), además de obtener los grupos estructurales, se deriva que el grupo que representa al terminal cuenta con 12 coordenadas dependientes, mientras que los grupos de los conjuntos manivela-barra quedan definidos mediante 15 coordenadas dependientes cada uno. Además, las matrices que surgen de la formulación basada en ecuaciones de grupo para un modelo como el de Stewart son dispersas y simétricas, con más del 70% de valores nulos. A nivel de experimentación, y con objeto de generalizar el problema a otros problemas con la misma topología pero con grupos de mayor complejidad, se usarán las mencionadas matrices 12×12 y 15×15 , y se repetirán los experimentos para matrices de tamaños mayores, y con diferentes grados de dispersión.

6.1.1.1 Plataforma de Stewart: resolución secuencial

La figura 6.6 muestra resaltada la ruta que corresponde a una resolución en orden secuencial de los distintos grupos de la estructura cinemática de una plataforma de Stewart. Se trata de una de las rutas construidas por el simulador representadas en la figura 6.5. Al no aplicarse paralelismo explícito, los recursos hardware se pueden asignar por completo a la resolución de cada grupo estructural. Esto es especialmente útil con plataformas multicore+multiGPU cuando se utilizan librerías que implementan paralelismo en sus códigos.

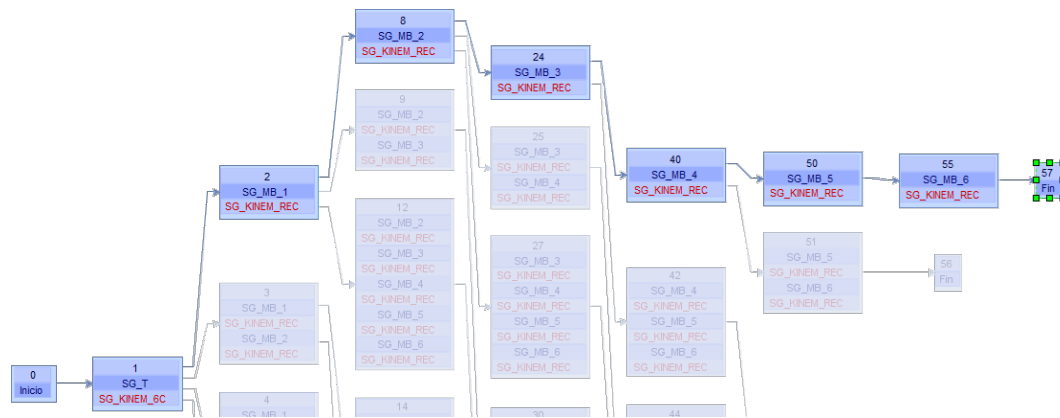


Figura 6.6: Ruta que resuelve de manera secuencial todos los grupos en los que se descompone una plataforma de Stewart como la representada en el modelo de la figura 6.4.

El primer grupo de ensayos corresponde a simulaciones de una plataforma de Stewart explotando únicamente paralelismo implícito de las librerías. Para las rutinas de multiplicación de matrices y resolución de sistemas de ecuaciones se empleará la librería Intel©MKL con varias asignaciones de threads, con objeto de determinar cuál de ellas ofrece mejores rendimientos. Obtendremos resultados para varios tamaños de matrices, donde el primero de ellos corresponde a las dimensiones del modelo real, $nEQ_T=12$ y $nEQ_M=15$. A partir de ahí se establecerán dimensiones diferentes para los grupos que resuelven `SG_KINEM_REC`, representando de esta manera grupos de mayor complejidad en los que sus sólidos se definen con un mayor número de coordenadas. En línea con el tipo de matrices obtenidas en las formulaciones basadas en ecuaciones de grupo, emplearemos matrices simétricas con sus valores distribuidos alrededor de la diagonal y con un factor de dispersión del 80%. El tipo de matriz y sus tamaños se indican en el simulador como un conjunto de escenarios (sección 5.2.6) que crearemos asignándoles nombres que van desde `MBS-STEWART1-80` hasta `MBS-STEWART8-80`, como mostramos en la tabla 6.1. El terminal mantiene su geometría invariable actuando como elemento al que se conectan las seis manivelas, por lo que mantiene fijo el número de coordenadas en todos los escenarios (12).

Escenario	Tipología	% Disp.	nEQ_T	nEQ_MB
<code>MBS-STEWART1-80</code>	Banda	80	12	15
<code>MBS-STEWART2-80</code>	Banda	80	12	30
<code>MBS-STEWART3-80</code>	Banda	80	12	60
<code>MBS-STEWART4-80</code>	Banda	80	12	120
<code>MBS-STEWART5-80</code>	Banda	80	12	360
<code>MBS-STEWART6-80</code>	Banda	80	12	1000
<code>MBS-STEWART7-80</code>	Banda	80	12	2000
<code>MBS-STEWART8-80</code>	Banda	80	12	3000

Tabla 6.1: Escenarios definidos para la simulación de una plataforma de Stewart que definen el tipo, factor de dispersión y los tamaños de las matrices asociados a la dimensión del terminal, nEQ_T , y a los grupos manivela-barra, nEQ_MB . El terminal, al estar unido siempre a seis manivelas, mantiene fijo el número de coordenadas en todos los escenarios.

A continuación seleccionaremos en el simulador la opción que permite obtener los tiempos de resolución por medio de una ejecución en secuencia de todos los grupos integrantes del modelo de Stewart, que será una de las ramas mostradas en el árbol de rutas del modelo. Siguiendo el proceso que se explica en la sec-

ción A.8.17, seleccionaremos la ruta mostrada en la figura 6.6. Una vez grabada, la información de dicha ruta queda asociada al modelo y será utilizada en la próxima ejecución del simulador.

El tipo de librería y los threads a utilizar se especifican mediante scripts (sección 5.2.7). Como hemos indicado anteriormente, en este primer experimento usaremos la librería MKL. Para el número de threads tenemos en cuenta el hardware, y en concreto el número de cores disponibles, 12 en el caso de la plataforma JUPITER del cluster *Heterosolar* (figura 3.6), perteneciente al Grupo de Computación Científica y Programación Paralela de la Universidad de Murcia. Con todo ello, el script almacena la información mostrada en la tabla 6.2.

Parámetro algorítmico	Valores
Número de threads del primer nivel (OpenMP)	{1}
Número de threads del segundo nivel (MKL)	{1, 2, 3, 4, 6, 10, 12}
Número de GPUs	{0}
Librería	{1 : <i>MKL</i> }

Tabla 6.2: Script definido para guiar el primer experimento de simulación de una plataforma de Stewart. Se asigna un único thread al primer nivel de paralelismo OpenMP al tratarse de una ejecución en secuencia de los grupos estructurales. El número de threads que se pone a disposición de la librería MKL se limita al número de cores físicos en JUPITER (12).

Como último paso, especificaremos en la configuración del simulador el modo de ejecución. En esta ocasión será el modo múltiple, descrito en la sección 5.6.2.3, para realizar tantas ejecuciones como combinaciones se obtengan con los valores de los parámetros algorítmicos especificados en el script. Además, marcamos la opción de simular con una ruta preseleccionada. La tabla 6.3 muestra de manera esquemática dicha configuración.

Una vez iniciada la simulación, el software realiza una ejecución por cada combinación de parámetros algorítmicos generada a partir de los valores especificados en el script. Los cálculos se realizan de acuerdo a la ruta preseleccionada (en este caso la que ejecuta los grupos en secuencia). La figura 6.7 muestra un extracto de la información mostrada por consola al finalizar el proceso de simulación (en concreto la simulación del escenario que contiene matrices de tamaño 3000×3000 y dispersión del 80%).

Model: MES-STEWART
Scenario: MES-STEWART8-80
Script: SCRIPT_ScriptTutorial1.scp

Branch	steps	m_type	m_spars	m_rows	m_cols	(s)	omph1	omph2	library	gpu	groups	sequence
57	9	1	80	3000	3000	26.613871	1	12	1	0	5-4-8-11-19-20-21-22-18	
57	9	1	80	3000	3000	27.178915	1	10	1	0	5-4-8-11-19-20-21-22-18	
57	9	1	80	3000	3000	35.875248	1	6	1	0	5-4-8-11-19-20-21-22-18	
57	9	1	80	3000	3000	50.784695	1	4	1	0	5-4-8-11-19-20-21-22-18	
57	9	1	80	3000	3000	58.910355	1	3	1	0	5-4-8-11-19-20-21-22-18	
57	9	1	80	3000	3000	74.818390	1	2	1	0	5-4-8-11-19-20-21-22-18	
57	9	1	80	3000	3000	134.093002	1	1	1	0	5-4-8-11-19-20-21-22-18	

Figura 6.7: Información obtenida por consola al finalizar las simulaciones en JUPITER, con tamaños de matrices (nEQ_MB) de 3000×3000 y variando el número de hilos asignados a MKL. El informe ordena de menor a mayor los tiempos de ejecución obtenidos.

Parámetro	Valores
Modelo a simular	{ MBS-STEWARD }
Modo de ejecución	Múltiple
Ruta	Preseleccionada

Tabla 6.3: Configuración aplicable a una ejecución del simulador para el modelo de la plataforma de Stewart. El modo de ejecución múltiple genera varias ejecuciones en función del contenido del script. En este experimento la ruta preseleccionada es la que realiza la ejecución en secuencia de los grupos.

Además de la información por consola, los tiempos de ejecución obtenidos se almacenan en una base de datos, como se describió en la sección 5.4, lo que permite un tratamiento posterior por medio de alguna herramienta de análisis (hojas de cálculo, software estadístico, etc.).

En la tabla 6.4 se pueden consultar los datos obtenidos en este primer experimento, con una única ejecución. En ella se comprueba la mejora de rendimiento debida a la paralelización de los cálculos matriciales de la librería MKL, siendo esta más significativa conforme aumenta el tamaño de las matrices.

nEQ_MB	MKL_sq	MKL_th2	MKL_th3	MKL_th4	MKL_th6	MKL_th10	MKL_th12
15	0.02341	0.02104	0.02110	0.02071	0.02081	0.02074	0.02103
30	0.02675	0.02679	0.02687	0.02670	0.02712	0.02685	0.02689
60	0.03471	0.04294	0.03999	0.03473	0.03452	0.03455	0.03481
120	0.06726	0.10049	0.07207	0.07473	0.07823	0.05967	0.07464
360	0.39681	1.05925	0.28366	0.27418	0.23914	0.23340	0.26520
1000	5.95198	4.38516	3.52734	2.83227	2.38782	2.08138	2.19284
2000	41.76047	25.11993	18.19453	15.12859	12.38607	10.85181	10.37525
3000	134.09300	74.81839	58.91036	50.78470	35.87525	27.17892	26.61387

Tabla 6.4: Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWARD) obtenidos en la plataforma JUPITER con varios tamaños de matrices (nEQ_MB) y variando el número de hilos asignados a MKL. Matrices simétricas con dispersión del 80%.

La misma ejecución sobre otra plataforma de hardware puede requerir cambios en algunos parámetros en el script. Por ejemplo, en SATURNO, que es otro nodo perteneciente al cluster *Heterosolar*, el número de cores físicos es de 24, por lo que tendremos que modificar el número de threads disponibles para la librería MKL como refleja la tabla 6.5.

Parámetro algorítmico	Valores
Threads primer nivel (OpenMP)	{1}
Threads segundo nivel (MKL)	{1, 2, 3, 4, 8, 16, 24}
Número de GPUs	{0}
Librería	{1 : <i>MKL</i> }

Tabla 6.5: Script definido para guiar el primer experimento de simulación de un modelo de Stewart en la plataforma SATURNO. Se limita el número de threads asignados al paralelismo interno de la librería MKL al número de cores físicos del hardware (24).

La nueva simulación consigue los resultados mostrados en la tabla 6.6, donde cada valor se obtiene como media de dos ejecuciones, una funcionalidad disponible en el simulador. Se observa que, a pesar de que los tiempos obtenidos en esta plataforma son inferiores a los obtenidos en JUPITER, también se cumple que el hecho de paralelizar los cálculos matriciales de la librería MKL consigue obtener notables mejoras de rendimiento.

nEQ_MB	MKL_sq	MKL_th2	MKL_th3	MKL_th4	MKL_th8	MKL_th16	MKL_th24
15	0.04792	0.04786	0.03179	0.03243	0.03761	0.02739	0.02778
30	0.06112	0.04946	0.06585	0.04007	0.04721	0.07527	0.10122
60	0.06883	0.09060	0.06579	0.07540	0.05325	0.06959	0.08722
120	0.13597	0.11488	0.09500	0.08720	0.09170	0.16040	0.16520
360	0.85828	0.63793	0.50359	0.49829	0.43655	0.57486	0.61038
1000	14.95429	10.12981	6.76745	7.55633	5.40775	5.03482	4.81100
2000	107.29968	62.34848	42.49197	36.96782	28.50321	27.09182	23.25768
3000	326.49321	191.43740	136.85202	114.41497	88.09155	61.94333	60.30540

Tabla 6.6: Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWART) obtenidos en la plataforma SATURNO con varios tamaños de matrices (nEQ_MB) y variando el número de hilos asignados a MKL. Matrices simétricas con dispersión del 80%.

En ambas plataformas, para tamaños de matrices inferiores a 1000×1000 , la sobrecarga que supone la gestión de threads hace que el aumento del número de estos no suponga mejora en el rendimiento. Pero es a partir de ese tamaño cuando se incrementa el speed-up, llegando a un valor de 5.04x para el tamaño de matrices de 3000×3000 en JUPITER y de 5.40x en SATURNO, como se puede comprobar en la figura 6.8. En las gráficas de evolución del tiempo de cómputo frente al tamaño de matrices empleadas se emplea escala logarítmica en el eje de las abscisas para resaltar los resultados que se obtienen con tamaños de matrices pequeñas, que son los que habitualmente se simulan en la comunidad multibody.

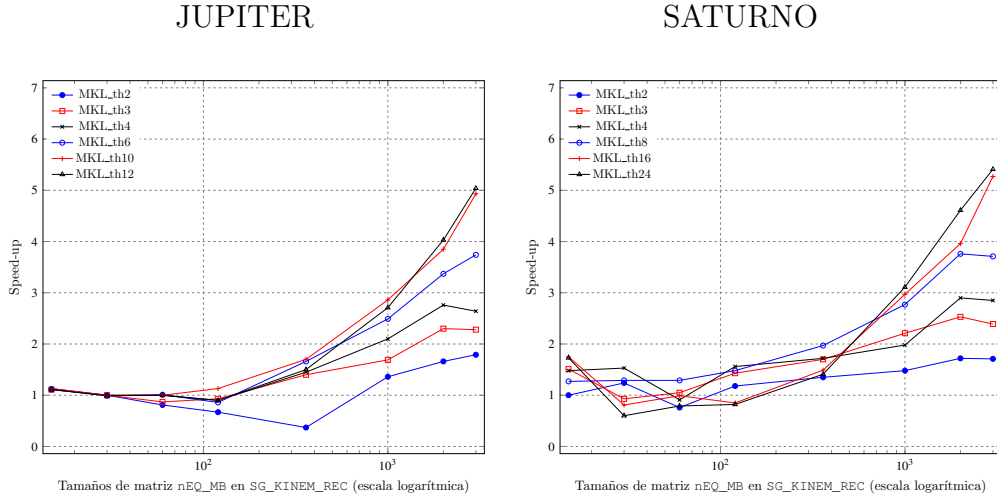


Figura 6.8: Speed-up respecto a MKL secuencial variando el número de hilos asignados a MKL, para diversos tamaños de matrices. Matrices simétricas con dispersión del 80%.

El mismo ejercicio se puede realizar con PARDISO, *Parallel Direct Sparse Solver*, una librería muy optimizada para trabajar con matrices dispersas simétricas y no simétricas. Para su ejecución en la plataforma JUPITER, se modifica el script para indicar el cambio de librería, quedando una configuración como la mostrada en la tabla 6.7.

Parámetro algorítmico	Valores
Threads primer nivel (OpenMP)	{1}
Threads segundo nivel (PARDISO)	{1, 2, 3, 4, 6, 10, 12}
Número de GPUs	{0}
Librería	{2 : PARDISO}

Tabla 6.7: Script definido para la simulación de una plataforma de Stewart en JUPITER empleando la librería PARDISO. Se asigna un thread al primer nivel de paralelismo OpenMP (ejecución en secuencia de los grupos estructurales). El número de threads a disposición de la librería se limita al número de cores (12).

Los resultados obtenidos quedan recogidos en la tabla 6.8. Se observa que los tiempos de ejecución que ofrece PARDISO son mejores que los de MKL cuando se manipulan matrices de dimensiones mayores de 1000×1000 . Y esto ocurre incluso sin paralelismo implícito y asignaciones de pocos threads (entre 2 y 4), lo que muestra la optimización de esta librería para trabajar con matrices dispersas.

nEQ_MB	PARD_sq	PARD_th2	PARD_th3	PARD_th4	PARD_th6	PARD_th10	PARD_th12
15	0.05032	0.06894	0.06295	0.06030	0.06359	0.06603	0.06801
30	0.06695	0.07506	0.08771	0.07662	0.07535	0.07837	0.07823
60	0.10100	0.11465	0.11430	0.10663	0.10976	0.11188	0.12063
120	0.16289	0.15360	0.15905	0.15021	0.14795	0.14796	0.15325
360	0.80643	0.64377	0.67087	0.65297	0.62256	0.64153	0.75040
1000	5.64075	4.03294	4.08755	3.66894	3.49814	3.54498	3.56475
2000	28.13423	19.39484	18.97692	16.27576	15.75483	15.46708	15.76543
3000	66.10252	47.47523	47.01676	38.71151	37.18689	36.11017	36.64083

Tabla 6.8: Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWART) obtenidos en la plataforma JUPITER con varios tamaños de matrices (nEQ_MB) y variando el número de hilos asignados a PARDISO. Matrices simétricas con dispersión del 80%

Por contra, los speed-up conseguidos al aumentar el número de threads en PARDISO son de magnitud inferior a los conseguidos con MKL, como se refleja en la figura 6.9, donde el mayor speed-up es de 1.83x para el tamaño de matrices de 3000×3000 .

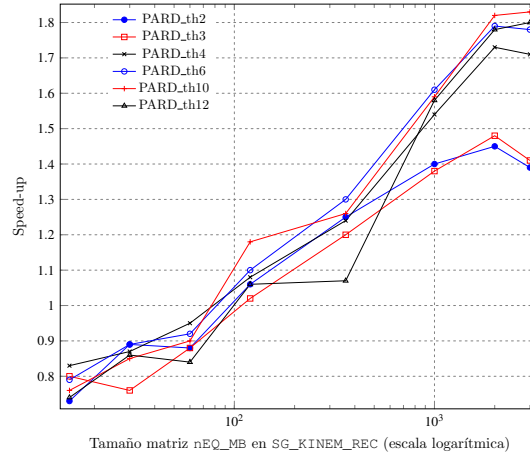


Figura 6.9: Speed-up respecto a PARDISO secuencial variando el número de hilos asignados a la librería, para diversos tamaños de matrices. Plataforma hardware JUPITER. Matrices simétricas con dispersión del 80%.

Se repiten ahora los mismos experimentos con PARDISO, pero en la plataforma SATURNO con 24 cores físicos. La tabla 6.9 muestra los resultados obtenidos con varias asignaciones de threads a la librería, donde nuevamente se comprueba la mejora de rendimiento aumentando el número de hilos, especialmente conforme aumenta el tamaño de las matrices tratadas.

nEQ_MB	PARD_sq	PARD_th2	PARD_th3	PARD_th4	PARD_th8	PARD_th16	PARD_th24
15	0.09151	0.10337	0.08323	0.09135	0.09367	0.09726	0.26302
30	0.10585	0.14141	0.13922	0.12336	0.11769	0.12869	0.25693
60	0.18529	0.18099	0.16894	0.13405	0.13973	0.18451	0.24645
120	0.30323	0.20309	0.19795	0.18121	0.19396	0.20614	0.28493
360	1.29694	1.03466	1.05293	1.01746	0.95169	1.14638	1.19874
1000	9.85990	7.05615	6.81412	6.26378	6.15644	6.66125	6.86684
2000	52.31130	34.74005	34.35743	30.21676	29.54983	28.93482	32.56781
3000	135.98866	90.65479	89.67818	72.70068	68.38662	69.85470	75.42628

Tabla 6.9: Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWART) obtenidos en la plataforma SATURNO con varios tamaños de matrices (nEQ_MB) y variando el número de hilos asignados a PARDISO. Matrices simétricas con dispersión del 80%

El siguiente experimento está encaminado a mostrar cómo el simulador puede ayudar a un usuario en la selección de la librería en sistemas moncore. En este caso podría interesar la utilización de librerías que, a pesar de no admitir paralelismo interno, puedan ofrecer buenos resultados debido a su implementación optimizada para un determinado tipo de matrices. Por tanto se deben comparar los rendimientos con otras que, admitiendo paralelismo, no lo puedan explotar debido a la limitación del hardware. Un ejemplo de implementación no paralela la ofrece MA27, que forma parte de la HSL, *Harwell Subroutine Library*, que está especializada en el manejo de matrices dispersas simétricas. En este experimento se analiza el comportamiento de todas las librerías incluidas en el simulador, descritas en la sección 3.2.2, en una ejecución en secuencia de los grupos y sin paralelismo implícito. Para ello el simulador nos permite incluir en los scripts una lista de librerías (tabla 6.10).

Parámetro algorítmico	Valores
Threads primer nivel (OpenMP)	{1}
Threads segundo nivel	{1}
Número de GPUs	{1}
Librería	{1 : MKL, 2 : PARDISO, 3 : MA27, 4 : MA57} {5 : MA48, 6 : MA86, 7 : MAGMA}

Tabla 6.10: Script definido para guiar el experimento de simulación del modelo de Stewart con todas las librerías incluidas en la versión actual del simulador. Se reserva un único thread al primer nivel de paralelismo OpenMP al tratarse de una ejecución en secuencia de los grupos estructurales. Se habilita una GPU para ser usada por la librería MAGMA (valor 1) y se desactiva el segundo nivel de paralelismo del resto de librerías (valor 1).

Estableciendo el número de threads del segundo nivel con un valor fijo de 1 se desactiva en el simulador el paralelismo implícito de aquellas librerías que lo admitan. De esta manera se comparan ejecuciones en modo secuencial de todas las librerías, simulando un hardware monocore en una plataforma multicore.

Además, si elegimos JUPITER para realizar las simulaciones, podemos utilizar la GPU que tiene instalada e incluir en este mismo experimento la simulación usando la librería MAGMA. En este caso el número de threads del segundo nivel que hemos fijado (1) no es relevante, y el thread de primer nivel creado en la CPU se utiliza para realizar la transferencia de datos a la GPU y lanzar los cálculos.

Los resultados, mostrados en la tabla 6.11 reflejan que, para matrices grandes (tamaños 3000×3000), la librería MAGMA presenta el mejor rendimiento por su aprovechamiento de la capacidad de cómputo de la GPU. Sin embargo, con matrices de menor tamaño los tiempos necesarios para el movimiento de información entre la memoria de la CPU y la GPU penalizan el uso de estos dispositivos. En ausencia de una GPU, los mejores resultados con matrices grandes los ofrece MA86, que muestran el mejor comportamiento para tamaños desde 1000×1000 en adelante.

nEQ_MB	MKL	PARDISO	MA27	MA57	MA48	MA86	MAGMA
15	0.02341	0.05032	0.02933	0.03567	0.02271	0.04751	0.07734
30	0.02675	0.06695	0.03032	0.02995	0.02922	0.04279	0.23196
60	0.03471	0.10100	0.03464	0.04228	0.03852	0.05692	0.24021
120	0.06726	0.16289	0.07008	0.09727	0.09390	0.09679	0.30023
360	0.39681	0.80643	0.55124	0.63214	0.85419	0.48040	2.01834
1000	5.95198	5.64075	5.93604	5.64270	9.73987	3.80918	6.86019
2000	41.76047	28.13423	35.81693	26.28010	68.16769	20.63315	22.86262
3000	134.09300	66.10252	109.05469	76.17988	212.16570	49.52427	47.04569

Tabla 6.11: Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWARD) obtenidos en la plataforma JUPITER con varios tamaños de matrices (nEQ_MB) y empleando diferentes librerías sin explotación de paralelismo implícito, y MAGMA explotando una GPU. Matrices simétricas con dispersión del 80%.

Como último experimento de esta sección de resolución secuencial de una plataforma de Stewart, sin paralelismo de grupos, realizaremos ejecuciones para comprobar la influencia del factor de dispersión de las matrices en las decisiones de

selección de la mejor librería. Para ello definimos un nuevo conjunto de escenarios que especifican valores del 30% para dicho factor. La tabla 6.12 muestra el nuevo conjunto de escenarios creados en el simulador.

Escenario	Tipología	% Disp.	nEQ_T	nEQ_MB
MBS-STEWART1-30	Banda	30	12	15
MBS-STEWART2-30	Banda	30	12	30
MBS-STEWART3-30	Banda	30	12	60
MBS-STEWART4-30	Banda	30	12	120
MBS-STEWART5-30	Banda	30	12	360
MBS-STEWART6-30	Banda	30	12	1000
MBS-STEWART7-30	Banda	30	12	2000
MBS-STEWART8-30	Banda	30	12	3000

Tabla 6.12: Escenarios definidos para la simulación de una plataforma de Stewart que especifican matrices banda, un factor dispersión del 30% y diferentes tamaños de las matrices asociados a la dimensión del terminal, nEQ_T, y a los grupos manivela-barra, nEQ_MB.

Los resultados obtenidos, recogidos en la tabla 6.13, comparados con los obtenidos con matrices con una dispersión del 80% (tabla 6.11) nos muestran que MKL y MAGMA ofrecen el mismo nivel de rendimiento en ambos experimentos, lo cual es debido a que ambas librerías implementan métodos para matrices densas. El resto de librerías, especializadas en el manejo de matrices dispersas, muestran un rendimiento notablemente peor en este escenario de matrices menos dispersas.

nEQ_MB	MKL	PARDISO	MA27	MA57	MA48	MA86	MAGMA
15	0.03511	0.06855	0.03703	0.03326	0.02809	0.05667	0.07550
30	0.04640	0.10206	0.04950	0.04357	0.03617	0.07420	0.23776
60	0.03990	0.18860	0.07807	0.06857	0.06117	0.08014	0.09795
120	0.06253	0.27707	0.17652	0.21410	0.12478	0.15367	0.26205
360	0.41613	1.35699	2.25658	1.37685	1.03515	0.89347	1.64686
1000	6.16760	11.55970	38.83014	19.65782	15.43737	7.25529	6.86989
2000	43.95146	61.34029	286.82495	109.51744	85.69672	36.33542	22.83841
3000	132.38977	164.91866	970.37451	332.06021	256.51447	99.15741	47.19025

Tabla 6.13: Comparación de los tiempos de ejecución del modelo de la plataforma de Stewart (MBS-STEWART) obtenidos en la plataforma JUPITER con varios tamaños de matrices (nEQ_MB) y empleando diferentes librerías sin explotación de paralelismo implícito. Matrices simétricas con dispersión del 30%.

Un análisis de los tiempos de ejecución nos permite concluir que, a pesar del factor de dispersión, para matrices de pequeño tamaños (15×15 y 30×30) la librería MA48 aún ofrece un buen rendimiento, mientras que con matrices grandes (2000×2000 y 3000×3000) MAGMA y las GPUs muestran toda su potencial. Para el resto de tamaños con los que se ha experimentado en este escenario de matrices densas, MKL ofrece el mejor rendimiento en una implementación con un único thread.

Podemos deducir de los experimentos mostrados en esta sección que, dado un problema a resolver en una plataforma dada, podemos optimizar el tiempo de ejecución decidiendo tanto la librería a utilizar, como el número de threads (cores de CPU) en base a las dimensiones del problema y dispersión de las matrices. Además se concluye que, de manera genérica, en programas diseñados con un enfoque puramente secuencial sin paralelismo explícito, podemos obtener mejoras de rendimiento usando librerías de álgebra matricial que implementen algoritmos paralelos en sus cálculos. Con la ayuda del simulador y la posibilidad que ofrece de realizar múltiples ejecuciones con diferentes parámetros algorítmicos, el usuario dispone de una herramienta que le permite identificar la mejor configuración, permitiendo a su vez obtener información para diferentes plataformas hardware.

6.1.1.2 Plataforma de Stewart: solución global

Antes de abordar la aplicación de paralelismo a la resolución simultánea de los grupos estructurales obtenidos mediante una formulación basada en ecuaciones de grupo, planteamos en esta sección un conjunto de experimentos que nos permitan validar la eficacia de dicha formulación frente al enfoque tradicional de modelado de sistemas multicuerpo (formulación global). Para ello se realizarán simulaciones de una plataforma de Stewart en la que todo el sistema multicuerpo se resuelve como si constara de un único grupo.

Un cálculo cinemático global debe ejecutar la rutina SG_KINEM_REC usada en la sección anterior, pero teniendo en cuenta las coordenadas del modelo completo, que incluye las correspondientes al terminal y a los grupos manivela-barra. Por tanto, los tamaños de las matrices se obtendrán a partir del tamaño del terminal, nEQ_T, y de los grupos manivela-barra, nEQ_MB.

Dado que este sistema consta de seis manivelas, los tamaños de las matrices se calcularán como: $nEQ_Global = 6 \cdot nEQ_MB + nEQ_T$.

La figura 6.10 muestra el modelo creado en el simulador correspondiente a la solución global.



Figura 6.10: Representación gráfica del modelo MBS-STEWART-GLOBAL para la resolución de una plataforma de Stewart sin división por grupos estructurales.

La tabla 6.14 recoge los escenarios que se usarán en los experimentos. Los tamaños de las matrices se muestran en la columna nEQ_Global . Se incluyen las columnas nEQ_T y nEQ_MB para reflejar su correspondencia con los tamaños derivados de la formulación basada en ecuaciones de grupo.

Escenario	Tipología	% Disp.	nEQ_T	nEQ_MB	nEQ_Global
MBS-STEWART1-80	Banda	80	12	15	102
MBS-STEWART2-80	Banda	80	12	30	192
MBS-STEWART3-80	Banda	80	12	60	372
MBS-STEWART4-80	Banda	80	12	120	732
MBS-STEWART5-80	Banda	80	12	360	2172
MBS-STEWART6-80	Banda	80	12	1000	6012

Tabla 6.14: Escenarios definidos para la simulación de una plataforma de Stewart que definen el tipo, factor de dispersión y los tamaños de las matrices para la solución global, nEQ_Global . Se incluyen como referencia la dimensión del terminal, nEQ_T , y de los grupos manivela-barra, nEQ_MB , usados en la formulación por grupos estructurales.

Para estos experimentos usaremos la librería MKL. Al no dividirse en grupos estructurales no se aplica OpenMP, por lo que se asigna solo un thread al primer nivel de paralelismo. También nos interesa comparar ambas formulaciones con diferentes asignaciones de threads a los cálculos realizados con MKL, 1 para la ejecución secuencial, 2 y 4 para ejecuciones paralelas. Con todo ello el script correspondiente al nuevo conjunto de experimentos almacena la información mostrada en la tabla 6.15.

Parámetro algorítmico	Valores
Número de threads del primer nivel (OpenMP)	{1}
Número de threads del segundo nivel (MKL)	{1,2,4}
Número de GPUs	{0}
Librería	{1 : <i>MKL</i> }

Tabla 6.15: Script definido para guiar el experimento de simulación de una plataforma de Stewart siguiendo una formulación global usando 4 cores de la plataforma JUPITER.

La tabla 6.16 muestra el resultado de la simulación con la formulación global, e incluye los obtenidos en la sección anterior (tabla 6.4) aplicando las mismas configuraciones paralelas y con los tamaños de matrices equivalentes de la formulación basada en grupos.

nEQ_Global	MKL_sq		MKL_th2		MKL_th4	
	Global	Grupos	Global	Grupos	Global	Grupos
102	0.02476	0.02341	0.02212	0.02104	0.02659	0.02071
192	0.03051	0.02675	0.02693	0.02679	0.03712	0.02670
372	0.09240	0.03471	0.10185	0.04294	0.06261	0.03473
732	0.54556	0.06726	0.41154	0.10049	0.22281	0.07473
2172	9.17419	0.39681	7.46936	1.05925	2.55019	0.27418
6012	179.40614	5.95198	92.36237	4.38516	52.17266	2.83227

Tabla 6.16: Comparación de los tiempos de ejecución de la solución global del modelo de la plataforma de Stewart (MBS-STEWARD) obtenidos en la plataforma JUPITER con varios tamaños de matrices (nEQ_Global) y variando el número de hilos asignados a MKL. Matrices simétricas con dispersión del 80%.

En términos generales se observa una mejora al trabajar con grupos estructurales respecto a los resultados obtenidos con una formulación global, ambas empleando MKL para los cálculos, consiguiendo speed-ups cada vez mayores conforme aumenta la dimensión del mecanismo, llegando hasta 30.1x para matrices de dimensiones 6012×6012 , como se refleja en la figura 6.11. También se observa que el speed-up es menor conforme aumenta el número de threads debido a que el paralelismo de MKL se explota mejor en la solución global, donde se tratan matrices más grandes. No obstante, la formulación basada en grupos permite usar otro tipo de paralelismo, como se analiza en la sección siguiente.

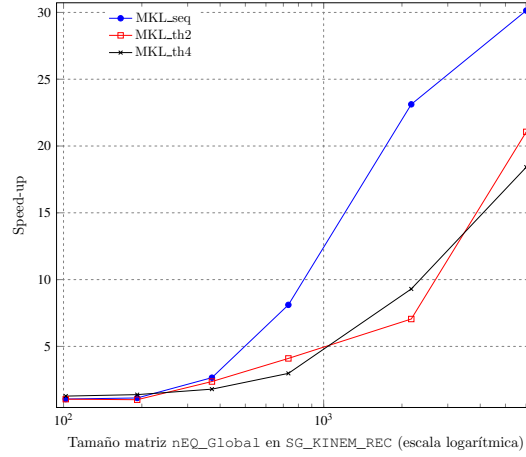


Figura 6.11: Speed-up de la formulación basada en ecuaciones de grupo frente a la global, variando el número de threads asignados a la librería MKL en JUPITER, para diversos tamaños de matrices. Matrices simétricas con dispersión del 80%.

6.1.1.3 Plataforma de Stewart: resolución paralela

Esta sección aborda la simulación de la plataforma de Stewart introduciendo paralelismo explícito en la solución de los grupos que representan las manivelas del mecanismo. La figura 6.12 muestra una ruta que propone la resolución simultánea de los grupos $\{SG_MB_1, SG_MB_2, SG_MB_3\}$ y posteriormente la de los grupos $\{SG_MB_4, SG_MB_5, SG_MB_6\}$. Otras rutas que introducen cálculos en paralelo se pueden encontrar en el árbol mostrado en la figura 6.5.

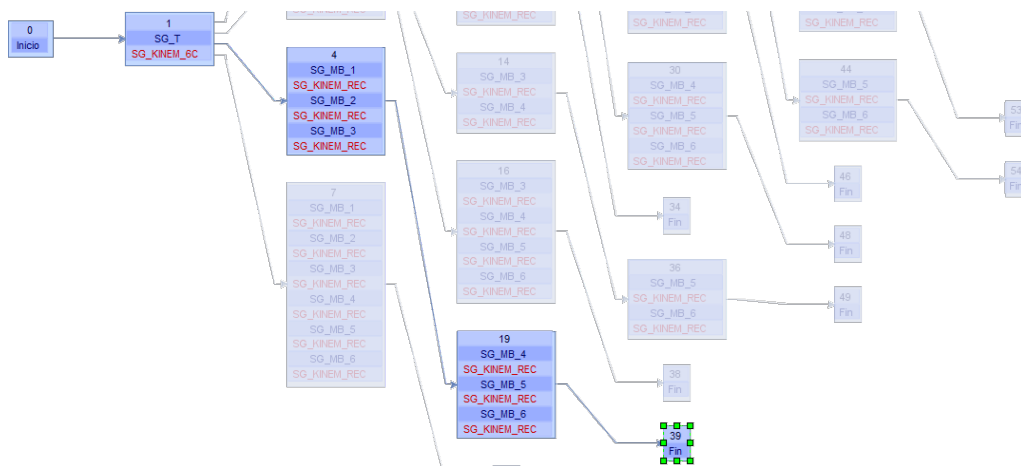


Figura 6.12: La ruta resaltada introduce paralelismo en la resolución de una Plataforma de Stewart como la representada en el modelo de la figura 6.4.

Debemos, por tanto, repetir los experimentos fijando una de esas rutas de ejecución paralela en el simulador. Siguiendo los pasos descritos en la sección A.8.17, seleccionamos la ruta mostrada en la figura 6.12 y grabamos dicho cambio. Esta selección queda almacenada junto al resto de información del modelo y será usada en las siguientes simulaciones.

Modificamos también el script para introducir el paralelismo en dos niveles. Dado que en la ruta elegida se resuelven tres grupos estructurales de manera simultánea, debemos añadir al menos un valor que asigne tres threads al primer nivel de paralelismo OpenMP. También indicaremos en dicho script que se van a usar MKL y PARDISO, como podemos ver en la tabla 6.17.

Parámetro algorítmico	Valores
Threads de primer nivel (OpenMP)	{1, 3}
Threads segundo nivel (MKL)	{1, 2, 3, 4, 6, 10, 12}
Número de GPUs	{0}
Librería	{1 : <i>MKL</i> , 2 : <i>PARDISO</i> }

Tabla 6.17: Script definido para guiar la simulación del modelo de Stewart usando las librerías MKL y PARDISO. Se asignan tres threads al primer nivel de paralelismo para permitir la resolución simultánea de tres grupos estructurales. El número de threads que se pone a disposición de las librerías se limita al número de cores físicos de JUPITER (12).

Esta nueva ejecución permite obtener los resultados que se muestran en la tabla 6.18. En ella, para cada dimensión de las matrices, se muestran los tiempos obtenidos al ejecutar en paralelo tres grupos estructurales aumentando los threads asignados a las librerías (MKL y PARDISO) hasta alcanzar un máximo de los 12 cores disponibles en JUPITER, es decir, aplicando las combinaciones 3×1 , 3×2 , 3×3 y 3×4 . Se incluyen dos columnas que recogen los mejores tiempos obtenidos en la sección anterior en una ejecución en secuencia de todos los grupos, y las asignaciones de threads correspondientes.

La misma ejecución en SATURNO obtiene los resultados que se muestran en la tabla 6.19. Dado que esta plataforma dispone de un máximo de 24 cores físicos, las combinaciones de threads a explorar son las que respetan dicho límite cuando se resuelven tres grupos de manera simultánea, es decir: 3×1 , 3×2 , 3×3 , 3×4 y 3×8 .

nEQ_MB	MKL		th. OMP \times th. MKL			
			3×1	3×2	3×3	3×4
15	1×4	0.02071	0.01315	0.01330	0.01336	0.01333
30	1×4	0.02670	0.01265	0.01297	0.01273	0.01339
60	1×6	0.03452	0.01511	0.01488	0.01561	0.01621
120	1×10	0.05967	0.02682	0.02395	0.03618	0.02820
360	1×10	0.23340	0.17734	0.17993	0.14890	0.13786
1000	1×10	2.08138	2.66616	1.74596	1.52307	1.81382
2000	1×12	10.37525	15.63358	10.02083	8.77579	7.86532
3000	1×12	26.61387	49.72425	30.33689	26.10797	23.40896
nEQ_MB	PARDISO		th. OMP \times th. PARDISO			
			3×1	3×2	3×3	3×4
15	1×1	0.05032	0.03500	0.03541	0.03544	0.03807
30	1×1	0.06695	0.02965	0.03583	0.03684	0.04110
60	1×1	0.10100	0.04200	0.04726	0.05062	0.06806
120	1×6	0.14795	0.06431	0.06267	0.06736	0.06947
360	1×6	0.62256	0.46196	0.40550	0.38145	0.34323
1000	1×6	3.49814	2.70817	2.23300	2.15049	2.32183
2000	1×10	15.46708	12.36325	9.78210	10.11429	8.99634
3000	1×10	36.11017	27.40348	25.44521	21.73778	21.49892

Tabla 6.18: Tiempos de ejecución obtenidos al combinar threads OpenMP y threads asignados a las librerías MKL y PARDISO, con varios tamaños de matrices (nEQ_MB) para resolver seis grupos estructurales Manivela-Barra en dos etapas. Hardware JUPITER con 12 cores, con matrices simétricas con dispersión del 80 %.

nEQ_MB	MKL		th. OMP \times th. MKL			
			3×1	3×2	3×3	3×4
15	1×16	0.02739	0.01810	0.01705	0.01742	0.01742
30	1×4	0.04007	0.02907	0.06895	0.05890	0.03320
60	1×8	0.05325	0.05290	0.03191	0.03136	0.03731
120	1×4	0.08720	0.09073	0.11892	0.06300	0.10932
360	1×8	0.43655	0.31309	0.29910	0.27771	0.27371
1000	1×24	4.81100	5.86757	4.89384	5.31004	4.07828
2000	1×24	23.25768	39.85024	26.14680	20.14455	21.73364
3000	1×24	60.30540	134.64927	84.95055	81.48293	50.86890
nEQ_MB	PARDISO		th. OMP \times th. MKL			
			3×1	3×2	3×3	3×4
15	1×3	0.08323	0.04165	0.04891	0.04812	0.05195
30	1×1	0.10585	0.06886	0.10432	0.09414	0.08114
60	1×4	0.13405	0.10844	0.07950	0.08531	0.10570
120	1×4	0.18121	0.14228	0.12726	0.13434	0.12006
360	1×8	0.95169	0.57615	0.53544	0.48909	0.58032
1000	1×8	6.15644	4.37001	5.27179	4.17358	4.01169
2000	1×16	28.93482	23.92102	19.35543	19.46898	18.06340
3000	1×8	68.38662	49.81405	42.49332	43.77798	35.39894

Tabla 6.19: Tiempos de ejecución obtenidos al combinar threads OpenMP y threads asignados a las librerías MKL y PARDISO, con varios tamaños de matrices (nEQ_MB) para resolver seis grupos estructurales Manivela-Barra en dos etapas. Hardware SATURNO con 24 cores, con matrices simétricas con dispersión del 80 %.

Un usuario que estudia la implementación computacional de un sistema multicuerpo concreto es conocedor de las dimensiones y topología de las matrices a utilizar. En este caso, interesa conocer la configuración de paralelismo y librerías que mejor se adapta a su escenario concreto. Por ejemplo, reordenando la información obtenida en las secciones anteriores se puede elaborar la tabla 6.20. En ella se comparan los rendimientos obtenidos empleando las librerías MKL, PARDISO y MA27 al operar sobre matrices de tamaños 30×30 y 3000×3000 . Buscando la mejor selección de parámetros algorítmicos en dicha tabla se puede observar que para matrices pequeñas (30×30), la librería MA27 obtiene los mejores resultados en la plataforma SATURNO con una asignación de threads 3×1 , es decir, tres threads asignados al primer nivel de paralelismo OpenMP y un thread asignado al segundo nivel. En el modelo de Stewart esta asignación corresponde a la ruta mostrada en la figura 6.12, que resuelve de manera simultánea los grupos $\{SG_MB_1, SG_MB_2, SG_MB_3\}$ y posteriormente los grupos $\{SG_MB_4, SG_MB_5, SG_MB_6\}$, sin explotación del paralelismo implícito de la librería, dado que MA27 no lo admite. Esta librería muestra un rendimiento optimizado para matrices dispersas de pequeño tamaño. Sin embargo, para matrices grandes con dimensión 3000×3000 , la librería que ofrece el mejor rendimiento es PARDISO, con una asignación de threads 3×8 , correspondiente a la ruta de resolución simultánea de tres grupos y asignando ocho threads al paralelismo interno de la librería.

th.Nivel 1 \times th.Nivel 2	30 \times 30			3000 \times 3000		
	MKL	PARDISO	MA27	MKL	PARDISO	MA27
3 \times 1	0.02907	0.06886	0.02459	134.64927	49.81405	76.11744
3 \times 2	0.06895	0.10432		84.95055	42.49332	
3 \times 3	0.05890	0.09414		81.48293	43.77798	
3 \times 4	0.03320	0.08114		50.86890	35.39894	
3 \times 8	0.11703	0.08498		48.29932	34.66773	
seq	0.06112	0.10585	0.04969	326.49321	135.98866	184.53877
1 \times 2	0.04946	0.14141		191.43740	90.65479	
1 \times 3	0.06585	0.13922		136.85202	89.67818	
1 \times 4	0.04007	0.12336		114.41497	72.70068	
1 \times 8	0.04721	0.11769		88.09155	68.38662	
1 \times 16	0.07527	0.12869		61.94333	69.85470	
1 \times 24	0.10122	0.25693		60.30540	75.42628	

Tabla 6.20: Comparación de los tiempos de ejecución obtenidos con MKL, PARDISO y MA27 en SATURNO con 24 cores, para matrices simétricas de tamaños 30×30 y 3000×3000 y un factor de dispersión del 80%. Se muestran todas las combinaciones de threads asignados al primer y segundo nivel de paralelismo.

En la siguiente sección se realizan experimentos en el modo autooptimizado en el que, además de la obtención de los parámetros algorítmicos, se determinará el orden óptimo de resolución de los grupos.

6.1.1.4 Plataforma de Stewart: ejecución autooptimizada

Como vimos en la figura 6.5, además de la ejecución en secuencia de los grupos que componen el modelo de la plataforma de Stewart, existen otras diez alternativas que resuelven el mismo problema, introduciendo en algún momento paralelismo de grupos. En la sección anterior se ha descrito el proceso que se sigue para realizar una simulación de una rama concreta seleccionada por el usuario. Pero, dependiendo de la complejidad de un modelo, la elección de una ruta puede no resultar sencilla a priori. El simulador facilita esta tarea mediante su modo de ejecución autooptimizado. En este modo, como se describió en la sección 5.5.4, se realiza un proceso de estimación del tiempo de ejecución asociado a cada rama del árbol de rutas con objeto de encontrar la que teóricamente puede ofrecer una resolución más rápida atendiendo al número de cores y GPUs del hardware. El cálculo del tiempo de ejecución de cualquier rama se realiza usando los tiempos de ejecución de las funciones que componen las rutinas usadas en el modelo. Dichos tiempos serán diferentes en función de los parámetros algorítmicos, AP , y el tipo de datos, SCN , sobre el que operan dichas funciones. Por este motivo es necesario construir un conjunto de datos de entrenamiento de las funciones para que esté disponible antes de lanzar una ejecución autooptimizada.

El simulador incorpora un modo de entrenamiento, descrito en la sección 5.6.2.1, encargado de realizar ejecuciones individuales de todas las funciones disponibles. Los parámetros algorítmicos y los tipos de matrices se especifican mediante un script y un conjunto de escenarios. En el script mostrado en la tabla 6.21 se seleccionan todas las librerías, variando en cada caso el número de threads asignados, con un rango desde 1 (sin paralelismo) hasta 24 (número de cores físicos de SATURNO). El número de GPUs a usar será como máximo de 1. El simulador descarta ejecuciones cuando la información del script no es coherente. Por ejemplo, la librería MAGMA no se ejecutará cuando el parámetro GPU tome el valor 0, dado que MAGMA requiere el uso de al menos una GPU.

Parámetro algorítmico	Valores
Número de threads del segundo nivel	$\{1, 2, \dots, 24\}$
Número de GPUs	$\{0, 1\}$
Librería	$\{1 : MKL, 2 : PARDISO, 3 : MA27, 4 : MA57\}$ $\{5 : MA48, 6 : MA86, 7 : MAGMA(GPU)\}$

Tabla 6.21: Script definido para guiar la construcción del conjunto de entrenamiento en la plataforma hardware SATURNO. Se obtendrán datos de ejecución con las siete librerías incluidas en el simulador, con asignaciones de threads desde 1 (sin paralelismo implícito) hasta 24 (cores físicos de la plataforma SATURNO).

En cuanto a los escenarios, definimos los recogidos en la tabla 6.22. Podemos observar que los tamaños de las matrices de entrenamiento están en consonancia con las usadas en las simulaciones del modelo de Stewart, recogidos en los escenarios mostrados en la tabla 6.1. Observamos que en algunos casos las dimensiones de las matrices coinciden, pero se han introducido intencionadamente otros escenarios que representan valores ligeramente diferentes. Cuando el simulador realiza los cálculos teóricos de tiempos de ejecución y necesita la información de la base de datos de entrenamiento, busca tiempos de ejecución obtenidos con matrices del mismo tipo y tamaño. En caso de no encontrar coincidencias, se inicia un proceso de búsqueda del escenario más cercano, tal y como se describió en el algoritmo 8 de la sección 5.5.4.

Escenario	Tipología	% Disp.	nEQ
SCENARIO1	Banda	80	15
SCENARIO2	Banda	80	30
SCENARIO3	Banda	80	50
SCENARIO4	Banda	80	100
SCENARIO5	Banda	80	400
SCENARIO6	Banda	80	500
SCENARIO7	Banda	80	1000
SCENARIO8	Banda	80	2000
SCENARIO9	Banda	80	3000

Tabla 6.22: Escenarios definidos para usar durante el entrenamiento de las funciones. Se definen la tipología de las matrices, su factor de dispersión y los tamaños de las matrices, nEQ.

Para activar el modo de ejecución de entrenamiento se accede a la configuración y se selecciona el valor correspondiente. También se puede indicar la plataforma de hardware, como se muestra en la tabla 6.23.

Parámetro	Valores
Hardware	SATURNO
Modelo a simular	{ MBS-STEWARD }
Modo de ejecución	Training

Tabla 6.23: Preparación del simulador para una ejecución de entrenamiento en la plataforma SATURNO.

Una vez ejecutado el entrenamiento, los tiempos obtenidos se almacenan en la base de datos y quedan accesibles para ser usados en las futuras ejecuciones autooptimizadas. La figura 6.13 muestra una consulta de los datos de entrenamiento realizada mediante el visor incorporado en el software, y cuyo uso se puede consultar en la sección A.8.25.2 del anexo. Se muestra la información que corresponde a los filtros aplicados: resultados obtenidos con MKL, variando los threads asignados al paralelismo interno de la librería al manipular matrices grandes de tamaño 3000×3000 .

Para simular el modelo de la plataforma de Stewart en el modo autooptimizado es necesario acceder de nuevo a la configuración del simulador para fijar dicho modo e indicar el número de cores y GPUs disponibles en el hardware. La tabla 6.24 muestra de manera esquemática dicha configuración en SATURNO.

Parámetro	Valores
Hardware	SATURNO
Modelo a simular	{ MBS-STEWARD }
Modo de ejecución	Autotuning
Number of cores	24
Number of GPUs	1

Tabla 6.24: Configuración aplicable a la siguiente ejecución del simulador para el modelo de la plataforma de Stewart. El modo de ejecución autooptimizado selecciona de manera automática la rama de ejecución y los valores de los parámetros algorítmicos que ofrecen los menores tiempos de ejecución teóricos, buscando el máximo aprovechamiento de los recursos del hardware.

Cuando iniciamos la ejecución, el simulador recorre uno por uno cada uno de los escenarios mostrados en la tabla 6.1. En cada caso, el software calcula la ruta y los parámetros algorítmicos asociados que suponen el tiempo de ejecución teórico más reducido, y resuelve el modelo siguiendo dicha ruta.

hardware	Simulation start	matrixtype	Sparsity	Rows Max	Cols Max	Th.OMP1	Th.O...▲	library	#GPUs	Function	Execution Time	matrixcolsk	numsamples	branchselection
SATURNO	02/11/2020 18:25:35	1	80	3000	3000	1	1	1	0	SOLVESYS-MKL	2.647096	3000	1	0
SATURNO	02/11/2020 18:26:15	1	80	3000	3000	1	2	1	0	SOLVESYS-MKL	1.422809	3000	1	0
SATURNO	02/11/2020 18:26:44	1	80	3000	3000	1	3	1	0	SOLVESYS-MKL	0.955696	3000	1	0
SATURNO	02/11/2020 18:27:08	1	80	3000	3000	1	4	1	0	SOLVESYS-MKL	0.723338	3000	1	0
SATURNO	02/11/2020 18:27:31	1	80	3000	3000	1	5	1	0	SOLVESYS-MKL	0.590991	3000	1	0
SATURNO	02/11/2020 18:27:52	1	80	3000	3000	1	6	1	0	SOLVESYS-MKL	0.511843	3000	1	0
SATURNO	02/11/2020 18:28:12	1	80	3000	3000	1	7	1	0	SOLVESYS-MKL	0.483474	3000	1	0
SATURNO	02/11/2020 18:28:31	1	80	3000	3000	1	8	1	0	SOLVESYS-MKL	0.385317	3000	1	0
SATURNO	02/11/2020 18:28:50	1	80	3000	3000	1	9	1	0	SOLVESYS-MKL	0.349610	3000	1	0
SATURNO	02/11/2020 18:29:09	1	80	3000	3000	1	10	1	0	SOLVESYS-MKL	0.316998	3000	1	0
SATURNO	02/11/2020 18:29:28	1	80	3000	3000	1	11	1	0	SOLVESYS-MKL	0.469360	3000	1	0
SATURNO	02/11/2020 18:29:46	1	80	3000	3000	1	12	1	0	SOLVESYS-MKL	0.432015	3000	1	0
SATURNO	02/11/2020 18:30:05	1	80	3000	3000	1	13	1	0	SOLVESYS-MKL	0.501330	3000	1	0
SATURNO	02/11/2020 18:30:24	1	80	3000	3000	1	14	1	0	SOLVESYS-MKL	0.320967	3000	1	0
SATURNO	02/11/2020 18:30:43	1	80	3000	3000	1	15	1	0	SOLVESYS-MKL	0.264942	3000	1	0
SATURNO	02/11/2020 18:31:01	1	80	3000	3000	1	16	1	0	SOLVESYS-MKL	0.315757	3000	1	0
SATURNO	02/11/2020 18:31:20	1	80	3000	3000	1	17	1	0	SOLVESYS-MKL	0.342421	3000	1	0
SATURNO	02/11/2020 18:31:39	1	80	3000	3000	1	18	1	0	SOLVESYS-MKL	0.329592	3000	1	0
SATURNO	02/11/2020 18:31:57	1	80	3000	3000	1	19	1	0	SOLVESYS-MKL	0.323244	3000	1	0
SATURNO	02/11/2020 18:32:15	1	80	3000	3000	1	20	1	0	SOLVESYS-MKL	0.338135	3000	1	0
SATURNO	02/11/2020 18:32:34	1	80	3000	3000	1	21	1	0	SOLVESYS-MKL	0.304957	3000	1	0
SATURNO	02/11/2020 18:32:52	1	80	3000	3000	1	22	1	0	SOLVESYS-MKL	0.274855	3000	1	0
SATURNO	02/11/2020 18:33:10	1	80	3000	3000	1	23	1	0	SOLVESYS-MKL	0.323349	3000	1	0
SATURNO	02/11/2020 18:33:30	1	80	3000	3000	1	24	1	0	SOLVESYS-MKL	0.329238	3000	1	0

Figura 6.13: Consulta de información de entrenamiento mediante el visor incorporado en la aplicación. Se muestran, filtrados, los tiempos de ejecución obtenidos en SATURNO por la librería 1 (MKL) al manipular matrices de tamaño 3000×3000 , simétricas y con un factor de dispersión del 80% y para cada valor de threads asignados al paralelismo interno.

El simulador muestra en la consola información de la ruta seleccionada para cada escenario y la librería asignada a cada uno de los grupos que componen el modelo. La figura 6.14 reoge el informe generado por el simulador en la resolución de la plataforma de Stewart de acuerdo al escenario MBS-STEWART8-80 con matrices de 3000×3000 . Se puede observar el tiempo obtenido (31.436998 segundos) y la ruta que ha guiado la ejecución. En un primer paso (Time Step 1) se resuelve el grupo Inicio, en una segunda etapa lo hace SG_T y a continuación se resuelven de manera simultánea los grupos desde el SG_MB_1 hasta el SG_MB_6 asignando cuatro threads al paralelismo de la librería PARDISO, explotando de esta manera en su totalidad los 24 cores físicos de SATURNO.

Se puede validar el resultado obtenido consultando la información de la tabla 6.25, donde se muestran los mejores tiempos de ejecución obtenidos al simular la plataforma Stewart con las librerías MKL y PARDISO mediante las rutas que incluyen paralelismo de grupos. La combinación 3×8 resuelve el modelo en dos etapas de tres manivelas cada una. La 2×12 incluye tres etapas, donde cada una calcula dos manivelas, y la 4×6 resuelve dos manivelas en una etapa y cuatro en la siguiente. La configuración obtenida en el modo autooptimizado, que determina una resolución simultánea de los seis grupos usando PARDISO, obtiene un tiempo de ejecución de 31.436998 segundos, muy cercano al óptimo de 30.14942 segundos que se consigue al resolver primero dos manivelas y a continuación las otras cuatro (4×6).

Experimentos						Autooptimizada
3×8		2×12		4×6		6×4
MKL	PARD	MKL	PARD	MKL	PARD	PARD
48.29932	34.66773	38.08665	43.07153	35.43077	30.14942	31.436998

Tabla 6.25: Mejores combinaciones de ejecución de la plataforma de Stewart en SATURNO para matrices de 3000×3000 y 80 % de dispersión, frente a la obtenida en una ejecución autooptimizada.

Cabe mencionar que es posible que el proceso de autooptimización determine que un mismo modelo deba resolver diferentes grupos con librerías distintas, por ejemplo podría sugerir que el grupo SG_T, que ejecuta la rutina SG_KINEM_6C, lo haga con MA27 y que el resto de grupos (del SG_MB_1 al SG_MB_6) resuelvan las rutinas SG_KINEM_REC con MKL.

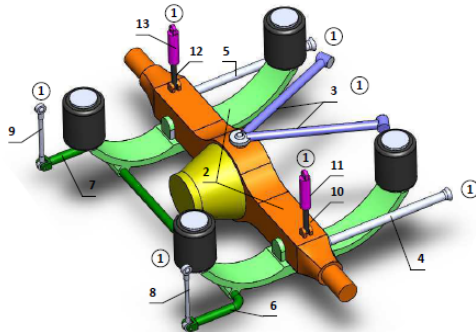
																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														</
--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

Figura 6.14: Información incluida en el LOG de ejecución del simulador en modo autooptimizado en el que podemos encontrar, además del tiempo de ejecución obtenido, la ruta seleccionada y las rutinas asignadas a cada grupo. Escenario con matrices de tamaño 3000×3000 , simétricas y con un factor de dispersión del 80%, en la plataforma SATURNO.

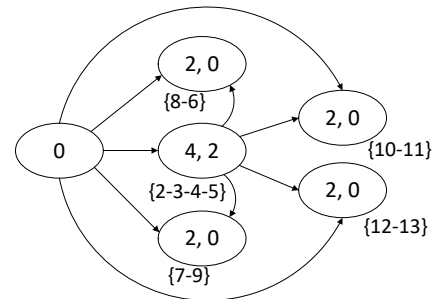
6.1.2 Modelo de suspensión de un camión

El segundo modelo presentado en este capítulo de experimentos corresponde al sistema de suspensión de ejes de un camión. Se trata de un sistema multicuerpo escalable, ya que se puede incluir un número creciente de ejes en el modelo. Cada uno de estos ejes está compuesto por trece cuerpos y diferentes tipos de articulaciones cinemáticas. La figura 6.15 muestra los cuerpos que forman cada uno de los ejes (izquierda) y su esquema estructural (derecha).

La estructura cinemática de este mecanismo se compone de tres tipos de grupos, por lo que será necesario crear en el simulador tres tipos de rutinas que se asignarán a los grupos en función de su tipo. La rutina `SG_KINEM_REC` ya se creó para la plataforma de Stewart, así como la `SG_KINEM_6C`, que se puede reutilizar aquí para crear la `SG_KINEM_2C3E1R`. Por tanto será necesario crear una tercera (`SG_KINEM_REP`).



(a) Elementos que forman el mecanismo correspondiente a un eje de un camión.



(b) Diagrama estructural que muestra los grupos estructurales identificados y las dependencias entre los mismos.

Figura 6.15: Sistema mecánico que representa el sistema de suspensión de un camión.

Una vez creadas las rutinas, se pueden introducir en el simulador todos los grupos que forman el modelo de la suspensión, al que hemos llamado `MBS-TRUCK` (figura 6.16). El nombre `_SG_2C3E1R` hace referencia a un grupo estructural con dos juntas cardan, tres esféricas y una de rotación.

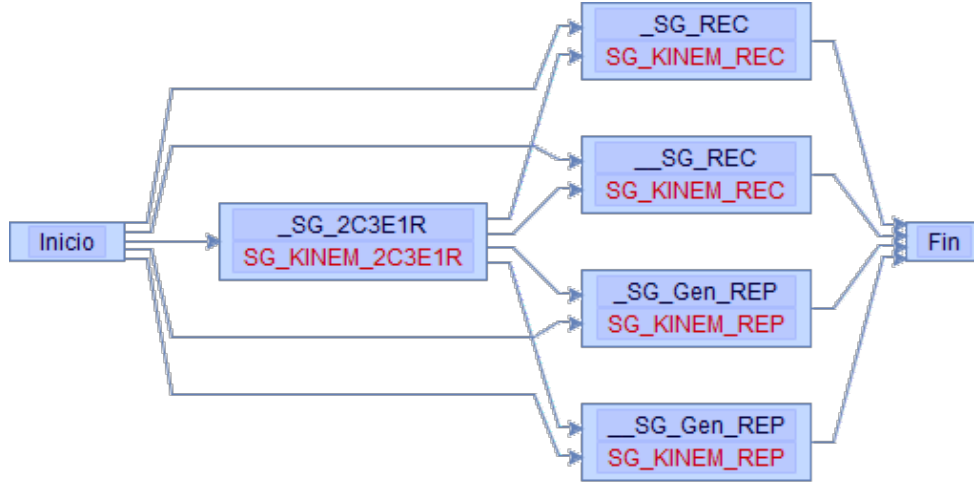


Figura 6.16: Representación en el simulador del modelo MBS-TRUCK de resolución de un mecanismo de suspensión de un camión.

A diferencia del modelo de la plataforma de Stewart estudiado en las secciones precedentes, en el modelo de la suspensión los grupos que son susceptibles de calcularse de manera simultánea contienen rutinas diferentes:

- Los grupos $\{_SG_REC, ___SG_REC\}$ calculan la SG_KINEM_REC .
- Los grupos $\{_SG_Gen_REP, ___SG_Gen_REP\}$ usan la SG_KINEM_REP .

Por tanto, la resolución simultánea de los grupos $\{_SG_REC, ___SG_REC\}$ va a ofrecer unos tiempos de ejecución diferentes a la resolución simultánea de los grupos $\{_SG_REC, _SG_Gen_REP\}$.

La herramienta incorporada en el simulador para el análisis del grafo de resolución de un modelo permite elaborar el grafo representado en la figura 6.17, donde se recogen todas las posibilidades de ordenación y agrupación de cálculos que permiten resolver el modelo MBS-TRUCK. Al igual que en la plataforma de Stewart, en el modelo de la suspensión del camión estudiaremos, además de la rama que ejecuta en secuencia todos los grupos, alguna rama que introduzca paralelismo de grupos.

También se analizará la extensión de la suspensión de un eje a la simulación de un modelo de mayor complejidad como el que representa el sistema de suspensión de un camión de tres ejes.

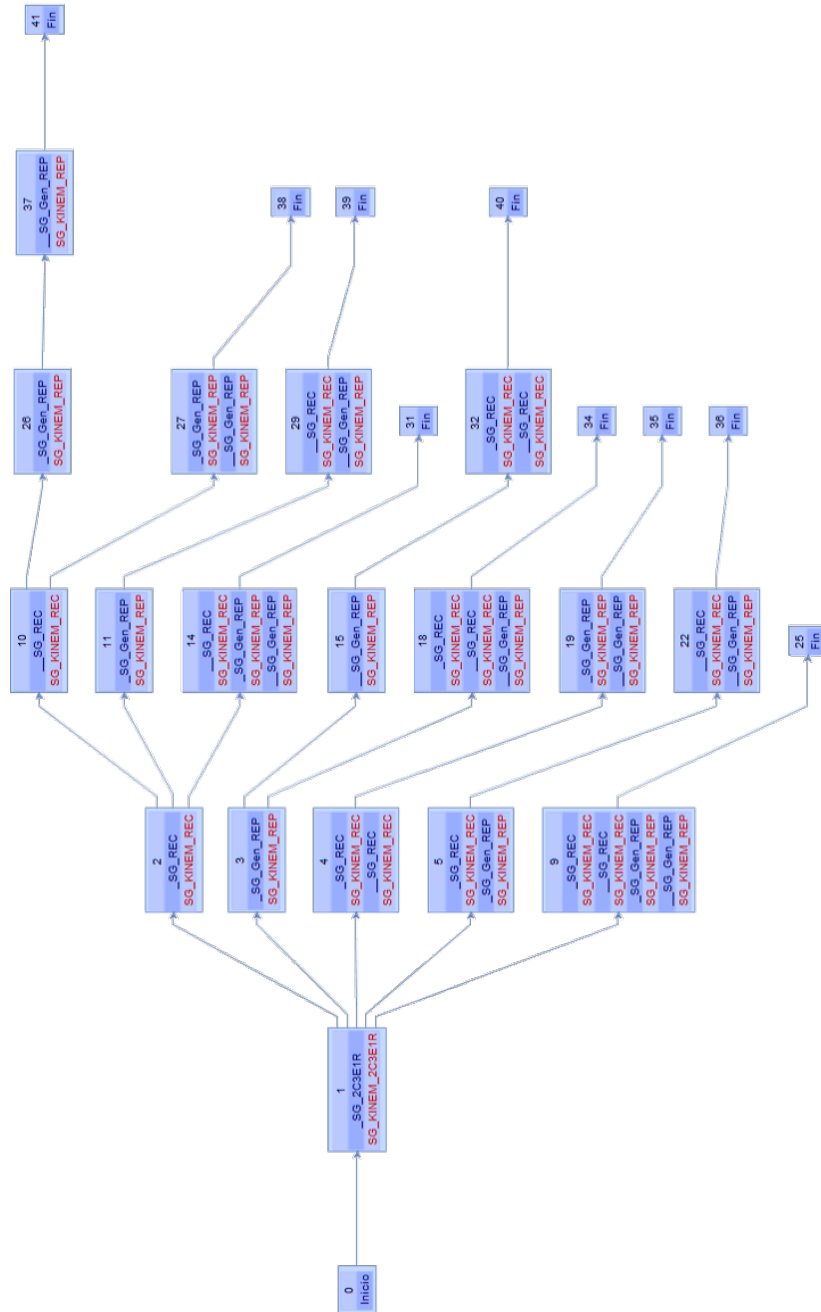


Figura 6.17: Árbol de rutas que representa las alternativas de ordenación y agrupación de cálculos para resolver un sistema mecánico de suspensión de un eje de un camión como el representado en el modelo de la figura 6.16.

6.1.2.1 Suspensión de un camión: resolución secuencial

En este apartado se incluyen experimentos que no aplican paralelismo de grupos. Se trata de obtener información sobre el rendimiento de las librerías al variar las asignaciones de threads a sus rutinas internas. La figura 6.18 muestra la ruta que corresponde a una resolución en secuencia de todos los grupos estructurales del modelo del sistema de suspensión de un camión.

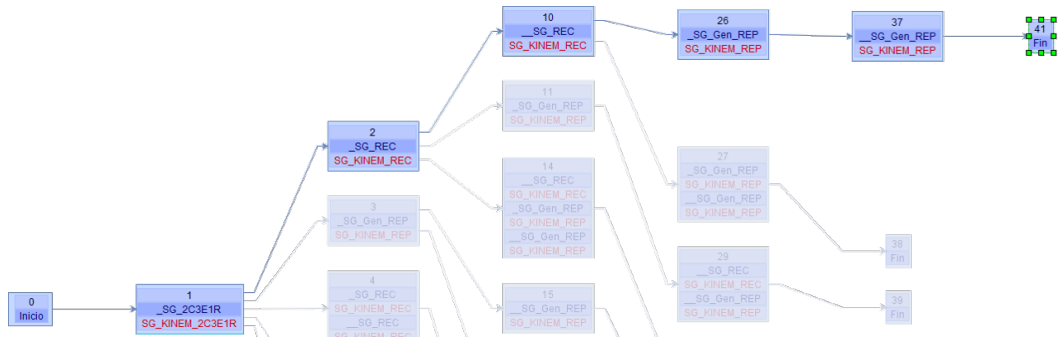


Figura 6.18: Ruta que resuelve en secuencia todos los grupos en los que se descompone el modelo de la suspensión de un eje de un camión como el representado en la figura 6.16.

De cara a los experimentos, los escenarios que representan los datos necesarios para resolver los grupos estructurales serán similares a los definidos en la sección 6.1.1.1 para la simulación de la plataforma de Stewart. Partiendo de matrices de tamaño 37×37 , llegaremos hasta tamaños de 3000×3000 para representar grupos más complejos con cada vez mayor número de coordenadas. Al obtenerse el modelo mediante una formulación basada en ecuaciones de grupo, las matrices obtenidas son simétricas con un factor de dispersión del 80% y los valores no nulos situados alrededor de la diagonal. Con todo ello podemos crear un conjunto de escenarios que nombraremos desde MBS-TRUCK1-80 hasta MBS-TRUCK7-80, cuyo detalle se puede consultar en la tabla 6.26.

A continuación seguimos las instrucciones recogidas en la sección A.8.17 para seleccionar en el simulador la ruta mostrada en la figura 6.18, que permite obtener los tiempos de resolución mediante la ejecución en secuencia de los grupos integrantes del modelo.

Escenario	Tipología	% Disp.	nEQ
MBS-TRUCK1-80	Banda	80	37
MBS-TRUCK2-80	Banda	80	60
MBS-TRUCK3-80	Banda	80	120
MBS-TRUCK4-80	Banda	80	360
MBS-TRUCK5-80	Banda	80	1000
MBS-TRUCK6-80	Banda	80	2000
MBS-TRUCK7-80	Banda	80	3000

Tabla 6.26: Escenarios definidos para la simulación de la suspensión de un eje de un camión que describen los tamaños de las matrices nEQ, su tipología y factor de dispersión.

En este experimento usaremos las librerías MKL y PARDISO. Para el número de threads tenemos en cuenta el número de cores físicos, 24 en el caso de la plataforma SATURNO del cluster *Heterosolar*. El script creado contiene la información mostrada en la tabla 6.27. La configuración del simulador para una ejecución en modo múltiple con una ruta preseleccionada se muestra en la tabla 6.28.

Parámetro algorítmico	Valores
Threads del primer nivel (OpenMP)	{1}
Thread del segundo nivel (MKL)	{1,2,3,4,8,16,24}
Número de GPUs	{0}
Librería	{1 : <i>MKL</i> , 2 : <i>PARDISO</i> }

Tabla 6.27: Script creado para el experimento de simulación de la suspensión de un camión. Se reserva un thread al primer nivel de paralelismo OpenMP para una ejecución en secuencia de los grupos. Los threads asignados a las librerías MKL y PARDISO se limitan al número de cores físicos de SATURNO (24).

Parámetro	Valores
Modelo a simular	{ MBS-TRUCK }
Modo de ejecución	Múltiple
Ruta	Preseleccionada

Tabla 6.28: Configuración del simulador para la ejecución del modelo de la suspensión de un camión en modo múltiple y con una ruta preseleccionada.

La simulación obtiene los resultados mostrados en las tablas 6.29 y 6.30, donde se observan las mejoras de rendimiento al paralelizar los cálculos matriciales de las librerías MKL y PARDISO respectivamente.

nEQ	MKL_sq	MKL_th2	MKL_th3	MKL_th4	MKL_th8	MKL_th16	MKL_th24
37	0.05891	0.06130	0.06021	0.05963	0.04973	0.08738	0.06907
60	0.08139	0.08459	0.07112	0.08652	0.06713	0.07123	0.06858
120	0.11569	0.10540	0.08769	0.08719	0.08732	0.12637	0.09976
360	0.74432	0.52438	0.42205	0.42033	0.34809	0.35174	0.43335
1000	12.16384	7.65464	6.00778	5.12701	3.70253	3.06832	3.20775
2000	84.39074	49.59199	36.00251	28.36838	20.62638	18.56056	18.10030
3000	263.40308	149.52103	106.81606	86.59923	52.73259	51.17312	49.33507

Tabla 6.29: Comparación de los tiempos de ejecución del modelo de la suspensión (MBS-TRUCK) obtenidos en la plataforma SATURNO con varios tamaños de matrices (nEQ) y variando el número de hilos asignados a MKL. Matrices simétricas con dispersión del 80%.

nEQ	PARD_sq	PARD_th2	PARD_th3	PARD_th4	PARD_th8	PARD_th16	PARD_th24
37	0.10518	0.09204	0.09451	0.09214	0.09932	0.10330	0.11242
60	0.14430	0.13272	0.15599	0.14020	0.15992	0.14654	0.17051
120	0.19471	0.15493	0.16133	0.16742	0.16930	0.19899	0.22135
360	1.09878	0.86491	0.87407	0.85483	0.80201	0.79954	0.93228
1000	8.06686	5.92144	6.14033	5.30444	4.66469	4.73366	4.56816
2000	40.34643	28.23735	28.44071	23.60055	21.53056	21.03959	22.99616
3000	98.10715	75.53687	73.53182	60.32187	53.17042	51.20848	56.99657

Tabla 6.30: Comparación de los tiempos de ejecución del modelo de la suspensión (MBS-TRUCK) obtenidos en la SATURNO con varios tamaños de matrices (nEQ) y variando el número de hilos asignados a PARDISO. Matrices simétricas con dispersión del 80%.

A partir de la información de dichas tablas podemos seleccionar los mejores resultados de cada una de las librerías y elaborar una nueva tabla (6.31) que muestra las mejores asignaciones de threads al paralelismo interno de MKL y PARDISO. En el caso de MA27, al no ofrecer una versión paralela, los resultados mostrados corresponden a una ejecución secuencial. Podemos comprobar que MKL es la librería que ofrece los mejores rendimientos en la plataforma SATURNO cuando se resuelven en secuencia todos los grupos estructurales que componen el modelo MBS-TRUCK. Y esto ocurre en todos los tamaños de matrices ensayados.

nEQ	MA27	MKL				PARDISO		
	seq	th4	th8	th16	th24	th2	th16	th24
37	0.06426	0.05963	0.04973	0.08738	0.06907	0.09204	0.10330	0.11242
60	0.08294	0.08652	0.06713	0.07123	0.06858	0.13272	0.14654	0.17051
120	0.11007	0.08719	0.08732	0.12637	0.09976	0.15493	0.19899	0.22135
360	0.59348	0.42033	0.34809	0.35174	0.43335	0.86491	0.79954	0.93228
1000	8.13748	5.12701	3.70253	3.06832	3.20775	5.92144	4.73366	4.56816
2000	48.51549	28.36838	20.62638	18.56056	18.10030	28.23735	21.03959	22.99616
3000	144.41975	86.59923	52.73259	51.17312	49.33507	75.53687	51.20848	56.99657

Tabla 6.31: Comparación de los mejores tiempos de ejecución del modelo de la suspensión (MBS-TRUCK) obtenidos en la plataforma SATURNO con varios tamaños de matrices (nEQ) variando la asignación de threads a las librerías MKL y PARDISO frente a MA27 en modo secuencial. Matrices simétricas con dispersión del 80%.

6.1.2.2 Suspensión de un camión: resolución paralela

Esta sección aborda la simulación del modelo de la suspensión del camión introduciendo paralelismo explícito en la solución de grupos. La figura 6.19 muestra dos posibles rutas que proponen la resolución simultánea de grupos:

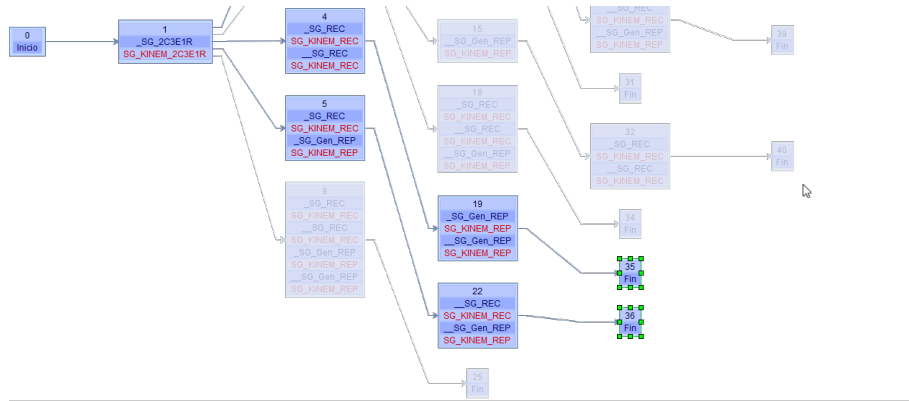


Figura 6.19: Las rutas resaltadas introducen paralelismo en la resolución de la suspensión de un camión como la representada en el modelo de la figura 6.16.

- La primera ruta comienza calculando el grupo `_SG_2C3E1R`. A continuación resuelve en paralelo `{_SG_REC, __SG_REC}` y, finalmente, los grupos `{_SG_Gen_REP, __SG_Gen_REP}` de manera simultánea.
- La segunda ruta calcula `_SG_2C3E1R` y continúa resolviendo en paralelo los grupos `{_SG_REC y _SG_Gen_REP}`. Por último trata simultáneamente los grupos `{__SG_REC, __SG_Gen_REP}`.

Nuestro análisis pretende mostrar la utilidad del software para simular varias rutas preseleccionadas por el usuario de la aplicación, y con ello encontrar la que ofrece mejor rendimiento. En este caso observamos que la primera ruta seleccionada resuelve en paralelo grupos que ejecutan la misma rutina, por lo que el coste computacional es similar a igualdad de escenarios. Sin embargo, la segunda ruta ejecuta de manera simultánea grupos que contienen rutinas diferentes.

Del mismo modo que en la secciones anteriores, indicaremos al simulador las rutas seleccionadas para la simulación. También comprobaremos que el script contenga paralelismo en dos niveles. Dado que en las rutas elegidas se resuelven dos grupos estructurales de manera simultánea, debemos añadir al menos el valor que indique dos threads al primer nivel de paralelismo OpenMP. También indicaremos en dicho script el uso de MKL y PARDISO, como se muestra en la tabla 6.32.

Parámetro algorítmico	Valores
Thread del primer nivel (OpenMP)	{2}
Thread del segundo nivel (MKL)	{1, 2, 3, 4, 8, 16, 24}
Número de GPUs	{0}
Librería	{1 : <i>MKL</i> , 2 : <i>PARDISO</i> }

Tabla 6.32: Script creado para la simulación del modelo de la suspensión empleando las librerías MKL y PARDISO. Se asignan dos threads al primer nivel de paralelismo para permitir la resolución simultánea de dos grupos. El número de threads que se pone a disposición del paralelismo implícito de las librerías se limita al número de cores físicos de SATURNO (24).

El simulador realiza los cálculos en cada una de las rutas aplicando todas las combinaciones de parámetros algorítmicos que se derivan de la información almacenada en el script. Una consulta en la base de datos permite elaborar la tabla 6.33 que muestra, para cada dimensión de las matrices, los tiempos obtenidos al ejecutar en paralelo dos grupos estructurales conforme aumentamos los threads asignados a las librerías hasta alcanzar el máximo de los 24 cores disponibles en SATURNO (combinaciones 2×1 , 2×2 , 2×3 , 2×4 , 2×8 y 2×12). Se incluyen dos columnas para mostrar los mejores tiempos obtenidos en la sección anterior en una ejecución en secuencia de todos los grupos, y las asignaciones de threads correspondientes en cada caso.

_SG_2C3E1R { _SG_REC, __SG_REC } { _SG_Gen_REC, __SG_Gen_REC }								
nEQ_MB	MKL		th. OMP × th. MKL					
			2 × 1	2 × 2	2 × 3	2 × 4	2 × 8	2 × 12
37	1 × 8	0.04973	0.03578	0.03219	0.03043	0.03099	0.03203	0.03833
60	1 × 8	0.06713	0.04331	0.04061	0.04067	0.03991	0.07803	0.12226
120	1 × 4	0.08719	0.10462	0.05796	0.05956	0.05535	0.06388	0.11576
360	1 × 8	0.34809	0.43377	0.37073	0.25038	0.24390	0.21624	0.34104
1000	1 × 16	3.06832	6.65778	4.06421	3.49995	2.76865	3.06329	2.58186
2000	1 × 24	18.10030	46.45289	32.70827	19.34889	15.80933	13.27327	11.64035
3000	1 × 24	49.33507	136.36582	80.03931	58.31936	47.11536	33.43908	31.74696
nEQ_MB	PARDISO		th. OMP × th. PARDISO					
			2 × 1	2 × 2	2 × 3	2 × 4	2 × 8	2 × 12
37	1 × 2	0.09204	0.06633	0.06858	0.07082	0.06797	0.07863	0.11447
60	1 × 2	0.13272	0.07773	0.09088	0.08900	0.09298	0.10754	0.11796
120	1 × 2	0.15493	0.14095	0.11490	0.11742	0.11908	0.13404	0.14295
360	1 × 6	0.79954	0.60551	0.47227	0.45961	0.46350	0.46859	0.74193
1000	1 × 24	4.56816	5.25509	3.84593	4.08118	3.40019	3.57016	4.00334
2000	1 × 16	21.03959	26.69744	19.53241	18.69812	17.59701	15.63244	18.43505
3000	1 × 16	51.20848	56.66381	40.71513	41.45576	33.26496	30.96204	36.18046
_SG_2C3E1R { _SG_REC, _SG_Gen_REC } { __SG_REC, __SG_Gen_REC }								
nEQ_MB	MKL		th. OMP × th. MKL					
			2 × 1	2 × 2	2 × 3	2 × 4	2 × 8	2 × 12
37	1 × 8	0.04973	0.03477	0.03329	0.03087	0.03102	0.03234	0.03923
60	1 × 8	0.06713	0.04729	0.04521	0.04519	0.04557	0.05499	0.05234
120	1 × 4	0.08719	0.09702	0.06574	0.06837	0.06852	0.07647	0.08516
360	1 × 8	0.34809	0.58851	0.44067	0.35714	0.35258	0.29864	0.31047
1000	1 × 16	3.06832	10.78664	6.43909	5.83582	4.82973	4.02271	4.05059
2000	1 × 24	18.10030	66.63464	41.59162	31.82747	26.66708	19.40707	18.44834
3000	1 × 24	49.33507	219.85583	131.32784	92.80238	85.23057	50.86643	48.17318
nEQ_MB	PARDISO		th. OMP × th. PARDISO					
			2 × 1	2 × 2	2 × 3	2 × 4	2 × 8	2 × 12
37	1 × 2	0.09204	0.07154	0.07161	0.07342	0.07596	0.08710	0.10277
60	1 × 2	0.13272	0.09133	0.10861	0.11226	0.11200	0.12593	0.13328
120	1 × 2	0.15493	0.15047	0.14723	0.15083	0.15272	0.16734	0.19426
360	1 × 16	0.79954	0.84933	0.68717	0.68020	0.67075	0.66874	0.71371
1000	1 × 24	4.56816	8.12228	6.07445	5.63400	5.21969	5.11202	5.50033
2000	1 × 16	21.03959	39.21499	29.18973	28.28186	25.18586	23.46454	25.81158
3000	1 × 16	51.20848	88.28458	62.84572	66.42331	55.23624	52.94094	51.36046

Tabla 6.33: Tiempos de ejecución obtenidos al combinar threads OpenMP y threads asignados a las librerías MKL y PARDISO, con varios tamaños de matrices (nEQ) para resolver el modelo MBS-TRUCK mediante dos rutas con paralelismo de grupos. Hardware SATURNO con 24 cores físicos. Matrices simétricas con dispersión del 80%.

Como se observó en experimentos anteriores en este capítulo, el aumento de los threads asignados al paralelismo implícito de las librerías supone reducciones en los tiempos, en especial conforme aumenta el tamaño de las matrices. Por otro lado se observa que, en general, los tiempos obtenidos en las ejecuciones que siguen rutas que resuelven simultáneamente grupos con la misma rutina son inferiores a los tiempos obtenidos al combinar grupos con rutinas diferentes. Esto es debido a que, en ejecuciones paralelas, el grupo que finaliza sus cálculos en primer lugar debe esperar a la conclusión del grupo más lento. Cuando los grupos que tienen mayor coste computacional están presentes en varias etapas de la ruta de ejecución la penalización se repite en todas ellas.

Otra posibilidad de resolución del modelo de la suspensión consiste en aplicar un mayor nivel de paralelismo. La ruta mostrada en la figura 6.20 supone el cálculo simultáneo de los cuatro grupos que contienen las rutinas SG_KINEM_REC y SG_KINEM_REP.

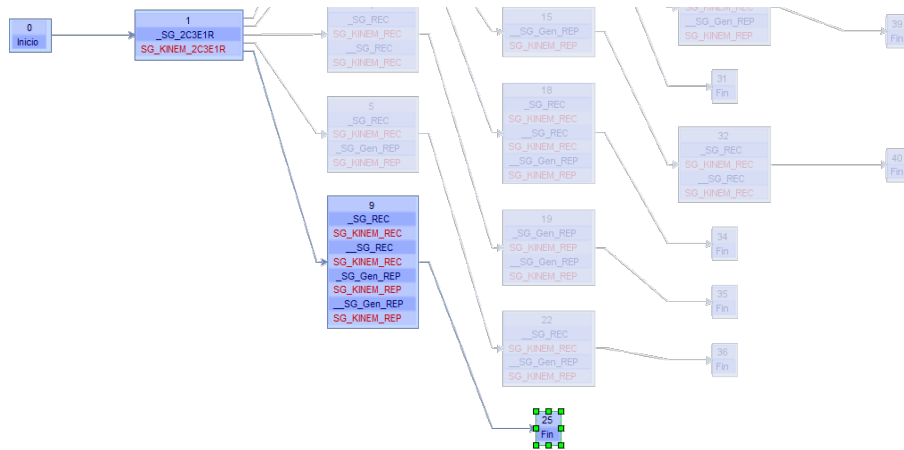


Figura 6.20: Ruta de resolución de la suspensión de un camión representada en el modelo de la figura 6.16 mediante el cálculo simultáneo de cuatro grupos.

Los tiempos de ejecución obtenidos siguiendo esta ruta se recogen en la tabla 6.34. Una comparación con los mostrados en la tabla 6.33 indica que en este modelo la resolución simultánea de dos grupos ofrece mejores resultados que paralelizar cuatro grupos cuando se manejan matrices de tamaños 360×360 y superiores. Este hecho se debe a que con un paralelismo de grupos menor es posible asignar más threads a los cálculos de MKL, lo que pone de manifiesto el gran rendimiento paralelo de esta librería.

nEQ	4×1	4×2	4×3	4×6
37	0.08391	0.04248	0.03042	0.04242
60	0.05286	0.04068	0.03374	0.03438
120	0.10308	0.05578	0.04721	0.05163
360	0.37355	0.26474	0.23452	0.24067
1000	5.77922	3.82559	3.01013	2.84275
2000	38.31067	24.11854	21.28847	12.53698
3000	121.90986	65.81306	49.52898	34.99459

Tabla 6.34: Tiempos de ejecución obtenidos al simular el modelo MBS-TRUCK con MKL mediante el cálculo simultáneo de cuatro grupos estructurales. Hardware SATURNO con 24 cores, con varios tamaños de matrices con dispersión del 80 %.

6.1.2.3 Suspensión de un camión: escalado a un modelo multieje

En esta sección se mostrará la utilización del software para resolver mecanismos complejos contruidos a partir de modelos creados previamente en el simulador, reutilizando la información de optimización de los mismos para establecer la mejor configuración de ejecución para el modelo compuesto.

Para ilustrar este caso usaremos el modelo de la suspensión estudiado en la secciones anteriores para construir un modelo que simule una cabeza tractora de tres ejes. Dado que todos los ejes se pueden calcular de manera independiente se puede construir un modelo con tres grupos como el mostrado en la figura 6.21(a).

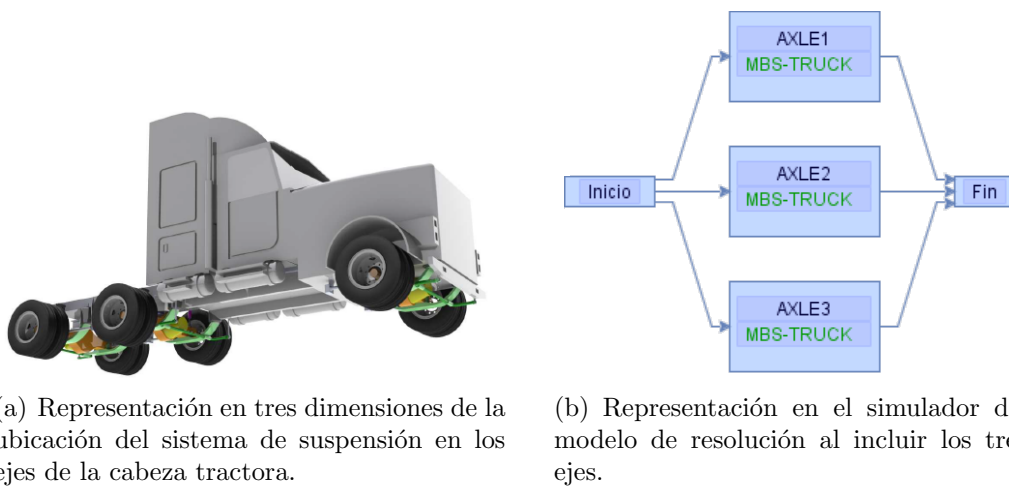


Figura 6.21: Sistema de suspensión en una cabeza tractora de tres ejes.

El simulador permite insertar modelos previamente creados dentro de un grupo. En este caso la resolución de dicho grupo implica la realización de todos los cálculos que componen el modelo importado. La figura 6.21(b) representa el nuevo modelo de tres ejes donde cada grupo resuelve el modelo insertado MBS-TRUCK.

La solución del sistema que engloba los tres ejes se puede abordar de varias formas. La figura 6.22 muestra resaltada la ruta que calcula en secuencia cada eje por separado. Sin embargo, la mostrada en la figura 6.23 supone la resolución simultánea de los tres ejes.

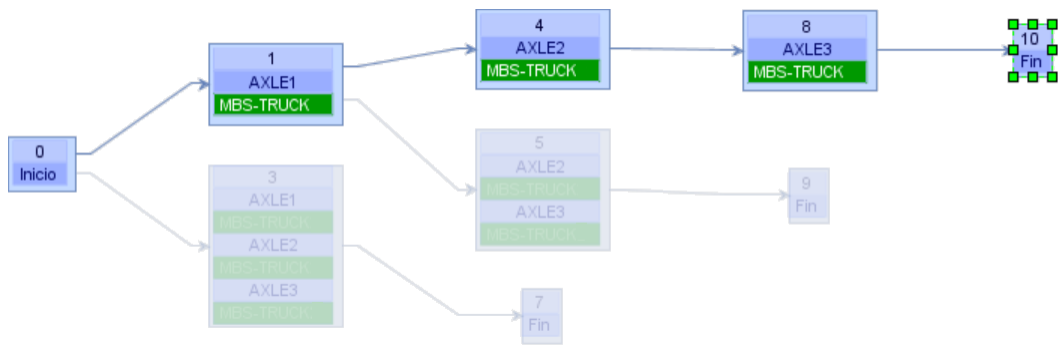


Figura 6.22: Ruta para la resolución en secuencia de los tres ejes de un sistema de suspensión multieje.

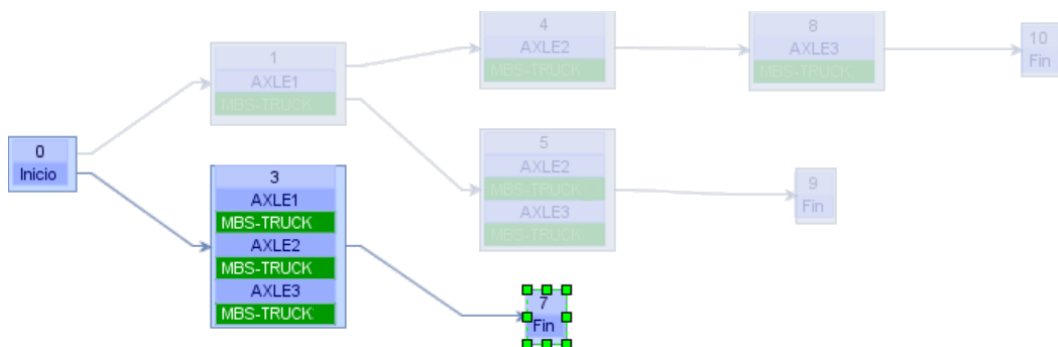


Figura 6.23: Ruta para la resolución simultánea de los tres ejes de un sistema de suspensión multieje.

Ahora bien, como se describió en la sección 6.1.2, el modelo de suspensión de un eje puede resolverse a su vez siguiendo alguna de las opciones de agrupación de cálculos reflejadas en su árbol de rutas (figura 6.17). Por tanto, antes de simular el sistema multieje será necesario indicar al software la ruta que debe seguir para resolver individualmente cada uno de los ejes. Esta asignación se hace en el modelo del multieje y no modifica las rutas que puedan estar seleccionadas en el modelo individual. La correcta elección de una u otra ruta va a depender de factores tales como las librerías de cómputo y los recursos del hardware. Para ayudarnos en dicha selección usaremos la información obtenida en las secciones anteriores, donde se realizaron simulaciones del modelo de un solo eje. Analizaremos por separado dos posibles aproximaciones a la solución del modelo compuesto:

- Cálculo de los tres ejes en secuencia: En este caso todos los recursos de un sistema multicore se pueden asignar a la resolución de cada eje. De acuerdo a las conclusiones obtenidas en la sección anterior (6.1.2.2), la resolución en paralelo de los grupos que forman el modelo de un eje es más rápida que la ejecución secuencial. Por tanto, para resolver el modelo multieje completo interesa introducir algún tipo de paralelismo de grupos al resolver los ejes que lo componen. La tabla 6.35 muestra los mejores tiempos que se habían obtenido con MKL en la plataforma SATURNO con 24 cores y diversas opciones de paralelismo.

nEQ	4×3	2×8	2×12
37	0.03042	0.03203	0.03833
60	0.03374	0.07803	0.12226
120	0.04721	0.06388	0.11576
360	0.23452	0.21624	0.34104
1000	3.01013	3.06329	2.58186
2000	21.28847	13.27327	11.64035
3000	49.52898	33.43908	31.74696

Tabla 6.35: Tiempos de ejecución del modelo MBS-TRUCK, suspensión de un eje, obtenidos con MKL en la plataforma SATURNO usando hasta 24 cores con varios tamaños de matrices y diferentes combinaciones de paralelismo. Matrices simétricas con dispersión del 80%.

Las columnas 2×8 y 2×12 se han elaborado con datos de la tabla 6.33 y muestran los tiempos de ejecución con resoluciones en paralelo de los grupos que contienen la rutina `_SG_KINEM_REC` seguidos de los que contienen la rutina `_SG_KINEM_REP`. La columna 4×3 es la mejor asignación de threads extraída de la tabla 6.34, donde se muestran diferentes opciones para el cálculo de los cuatro grupos en paralelo.

Se observa que en un hardware con 24 cores la resolución más eficiente con matrices de tamaños hasta 120×120 se obtiene calculando simultáneamente los cuatro grupos, asignando tres cores a MKL. La figura 6.24 muestra el simulador con el modelo de tres ejes cuando se ha seleccionado en el modelo importado la ruta que resuelve un eje de acuerdo a esta estrategia.

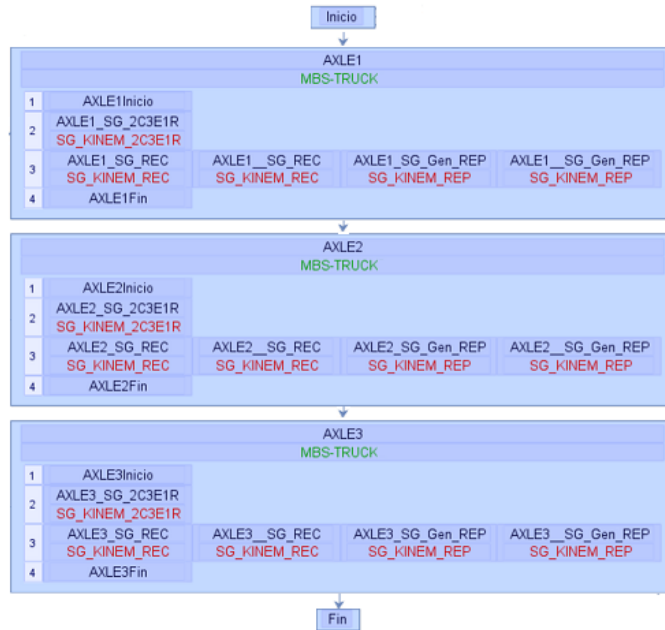


Figura 6.24: Ruta de solución en secuencia del modelo de tres ejes, resolviendo en paralelo los cuatro componentes paralelizables de cada eje.

Para tamaños de matrices mayores la mejor opción consiste en resolver en paralelo los grupos que contienen la misma rutina, asignando a MKL ocho cores en el caso de 360×360 y doce cores en el resto. Una vez seleccionada dicha ruta en cualquiera de los ejes del modelo multieje, el simulador actualiza el grafo como refleja la figura 6.25.

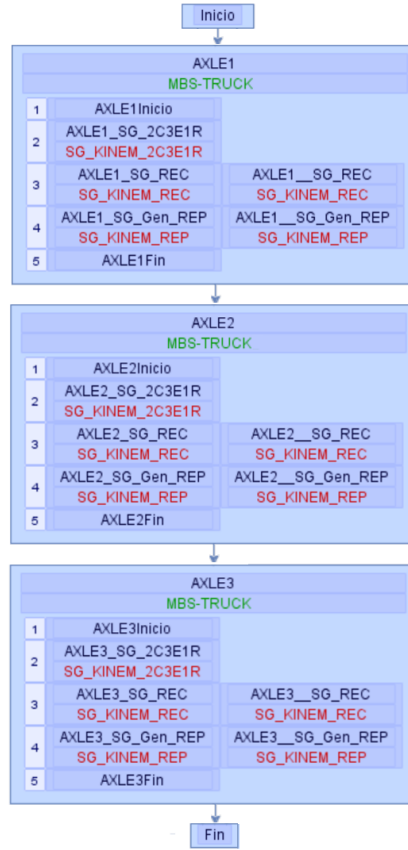


Figura 6.25: Ruta de solución en secuencia del modelo de tres ejes, resolviendo en paralelo bloques de dos grupos dentro de cada eje.

- Cálculo de los tres ejes de manera simultánea: En este caso los 24 cores de SATURNO se deben repartir, siendo una posible asignación la reserva de ocho cores a la resolución de cada eje. Para encontrar la mejor ruta que resuelva un solo eje de acuerdo a la nueva restricción de recursos consultamos las combinaciones de ocho o menos threads reflejadas en la tabla 6.36. Las columnas 2×3 y 2×4 son las mejores combinaciones mostradas en la tabla 6.33 cuando se resuelven dos grupos en paralelo. Las columnas 4×1 y 4×2 contienen los tiempos de ejecución recogidos en la tabla 6.34 obtenidos en la resolución simultánea de cuatro grupos.

Observamos que en todos los tamaños de matrices el mejor rendimiento se obtiene resolviendo simultáneamente dos grupos, asignando tres threads a MKL (2×3) en el caso de 37×37 , y cuatro (2×4) en el resto de tamaños.

nEQ	4 × 1	4 × 2	2 × 3	2 × 4
37	0.08391	0.04248	0.03043	0.03099
60	0.05286	0.04068	0.04067	0.03991
120	0.10308	0.05578	0.05956	0.05535
360	0.37355	0.26474	0.25038	0.24390
1000	5.77922	3.82559	3.49995	2.76865
2000	38.31067	24.11854	19.34889	15.80933
3000	121.90986	65.81306	58.31936	47.11536

Tabla 6.36: Tiempos de ejecución del modelo MBS-TRUCK, suspensión de un eje, con MKL en SATURNO para varios tamaños de matrices (nEQ) y combinaciones de paralelismo hasta ocho cores. Matrices simétricas con dispersión del 80%.

En esta aproximación a la resolución del modelo multieje, donde los tres ejes se resuelven simultáneamente, los tiempos mostrados son los requeridos para resolver el modelo completo. Por ejemplo, en el caso de matrices de 3000×3000 , la ejecución asignando 2×4 threads a cada eje consumiría 47.115364 segundos. Sin embargo, con una ejecución en secuencia de los ejes, el tiempo de resolución sería 3 veces el tiempo recogido en la tabla 6.35, es decir, $3 \cdot 31.74696 = 95.24089$ segundos para una asignación de 2×12 threads a cada eje. Por tanto la resolución paralela de los tres ejes mejora en 48.12552 segundos a la secuencial. La figura 6.26 muestra la representación en el simulador del modelo con esta estrategia de ejes calculados de manera simultánea, y con paralelismo de dos grupos.

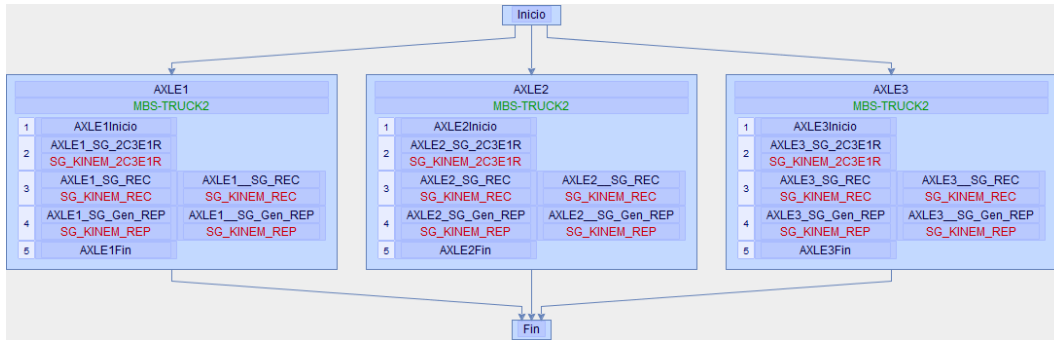


Figura 6.26: Ruta de solución en paralelo del modelo de tres ejes, resolviendo en paralelo grupos de dos componentes de cada eje.

En casos donde el reparto de recursos no sea tan claro como el presentado (24 cores entre 3 ejes), se debería realizar un conjunto adicional de diferentes pruebas.

En esta sección se ha mostrado el uso del simulador como herramienta de optimización de modelos complejos (por ejemplo una suspensión de tres ejes) en los que la selección de la ruta y los parámetros algorítmicos de los modelos simples que los componen (suspensión de un eje) se deciden en función de las mejores configuraciones obtenidas en la resolución experimental de esos modelos simples. Esta optimización es complementaria a la que realiza el simulador de manera automática en su modo de autooptimización, en la que los datos usados para la selección de la mejor configuración son los tiempos de entrenamiento de las funciones disponibles en el simulador.

6.2 Aplicación del simulador a la optimización de rutinas de álgebra lineal

En la disciplina del álgebra lineal existen rutinas en las que el enfoque estándar de resolución puede no ser el óptimo en términos de tiempo de computación. Un ejemplo es la multiplicación de matrices, que presenta un interés especial por su uso en la resolución de problemas científicos y de ingeniería, y por estar incluida en el cálculo de numerosas rutinas de álgebra lineal. Además del algoritmo de resolución tradicional basado en tres bucles, se han desarrollado otros como son la multiplicación por bloques, o el algoritmo de Strassen. En estos casos el simulador puede ayudar a encontrar los mejores parámetros algorítmicos, librerías y asignación de operaciones a unidades de cómputo. El resultado obtenido puede ayudar al desarrollo óptimo de rutinas de nivel jerárquico superior que hagan uso de las rutinas básicas.

6.2.1 Multiplicación de matrices por bloques

La multiplicación por bloques se muestra especialmente eficiente al manipular matrices con un gran número de filas y columnas. En este caso puede ser interesante descomponer el problema en otros que utilizan matrices de menor tamaño. Las matrices originales se descomponen por filas o por columnas trazando imaginariamente líneas rectas entre los elementos que las componen (figura 6.27).

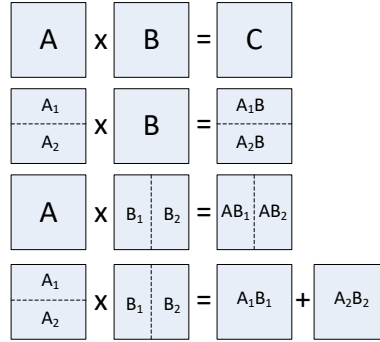


Figura 6.27: Varias aproximaciones a la multiplicación de matrices por bloques.

Las operaciones posteriores sobre matrices ya divididas se producen considerando los bloques como elementos de una matriz, según la fórmula $C_{ij} = \sum_{k=1}^r A_{ik}B_{kj}$. Para que un producto por bloques $AB = C$ pueda realizarse, las matrices A y B deben estar descompuestas en bloques de forma conforme, es decir, el número de bloques columna de la matriz A debe ser igual que el de bloques fila de la matriz B , y dichos bloques han de ser de tamaño adecuado para que se puedan multiplicar por elementos como sigue:

$$AB = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right] \cdot \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right] = \left[\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{array} \right]$$

Para comprobar mediante el uso del simulador las mejoras obtenidas en los tiempos de ejecución que se consiguen con este algoritmo de multiplicación de matrices creamos inicialmente un modelo básico que resuelve una multiplicación de dos matrices con sus tamaños originales. El tiempo de resolución de dicho modelo con varios tamaños de matrices nos servirá de base con la que comparar posteriores ejecuciones donde aplicaremos divisiones por bloques.

Para su implementación en el simulador basta con crear una rutina de usuario que incluya una multiplicación. Si llamamos a dicha rutina RMM, entonces un modelo para resolver una multiplicación podría ser el mostrado en la figura 6.28.

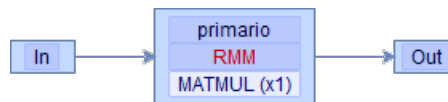


Figura 6.28: Representación en el simulador del modelo MATMUL de multiplicación de matrices $AB = C$.

Al igual que se hizo en los experimentos con sistemas multicuerpo, crearemos un conjunto de escenarios para especificar los tamaños de las matrices y su factor de dispersión. En esta sección dedicada a la multiplicación, trabajaremos con matrices no simétricas, con factores de dispersión del 30%, y tamaños dentro del conjunto $\{100 \times 100, 500 \times 500, 1000 \times 1000, 2000 \times 2000, 3000 \times 3000\}$. Estos escenarios se muestran detallados en la tabla 6.37.

Escenario	% Disp.	nROWS	nCOLS
MATMUL_SCENARIO_MULT0100	30	100	100
MATMUL_SCENARIO_MULT0500	30	500	500
MATMUL_SCENARIO_MULT1000	30	1000	1000
MATMUL_SCENARIO_MULT2000	30	2000	2000
MATMUL_SCENARIO_MULT3000	30	3000	3000

Tabla 6.37: Escenarios creados para la resolución de una multiplicación de matrices. Los escenarios definen la tipología de las matrices, dispersión y tamaños. nROWS y nCOLS indican el número de filas y columnas, respectivamente.

6.2.1.1 Experimentos en sistemas con CPU multicore

En este apartado se realizan ejecuciones usando la multiplicación de matrices implementada en MKL, tanto en JUPITER (que dispone de 12 cores físicos), como en SATURNO (con 24 cores físicos). En la tabla 6.38 se pueden consultar los resultados de las ejecuciones obtenidas en JUPITER con diferentes asignaciones de threads.

nROWS	MKL_th1	MKL_th2	MKL_th4	MKL_th8	MKL_th10	MKL_th12
100	0.00030	0.00045	0.00039	0.00028	0.00029	0.00029
500	0.01780	0.01248	0.00519	0.00363	0.00417	0.00496
1000	0.13394	0.06613	0.03401	0.02176	0.02452	0.03260
2000	0.84583	0.44830	0.24137	0.14324	0.11863	0.16818
3000	2.82709	1.46063	0.77464	0.42583	0.41011	0.30510

Tabla 6.38: Comparación de los tiempos de ejecución del modelo de la multiplicación de matrices (MATMUL) obtenidos en JUPITER con varios tamaños de matrices (nROWS×nCOLS) y variando el número de hilos asignados a MKL.

Los mismos experimentos, pero ejecutados en la plataforma SATURNO, permiten obtener los datos mostrados en la tabla 6.39

nROWS	MKL_th1	MKL_th2	MKL_th4	MKL_th8	MKL_th10	MKL_th24
100	0.00080	0.00096	0.00043	0.00069	0.00076	0.00252
500	0.03816	0.02386	0.01099	0.01003	0.00875	0.01221
1000	0.28872	0.14988	0.09032	0.04769	0.04291	0.06681
2000	2.27098	1.15297	0.60477	0.38777	0.37256	0.24645
3000	7.64038	3.88234	1.96197	1.16516	0.82309	0.75068

Tabla 6.39: Comparación de los tiempos de ejecución del modelo de la multiplicación de matrices (MATMUL) obtenidos en SATURNO con varios tamaños de matrices (nROWS×nCOLS) y variando el número de hilos asignados a MKL.

En ambas plataformas se observa que la mejora de rendimiento debido a la paralelización de los cálculos matriciales de la librería MKL es más significativa conforme aumenta el tamaño de las matrices. Es especialmente notable para matrices grandes (2000×2000 y 3000×3000), donde el speed-up llega a valores de 9.27x para un tamaño de matrices de 3000×3000 en JUPITER y de 10.18x en SATURNO (figura 6.29). En todo caso el speed-up está lejos del óptimo, lo que nos lleva a estudiar a continuación el uso de un paralelismo de dos niveles gracias a la resolución de la multiplicación de matrices por bloques.

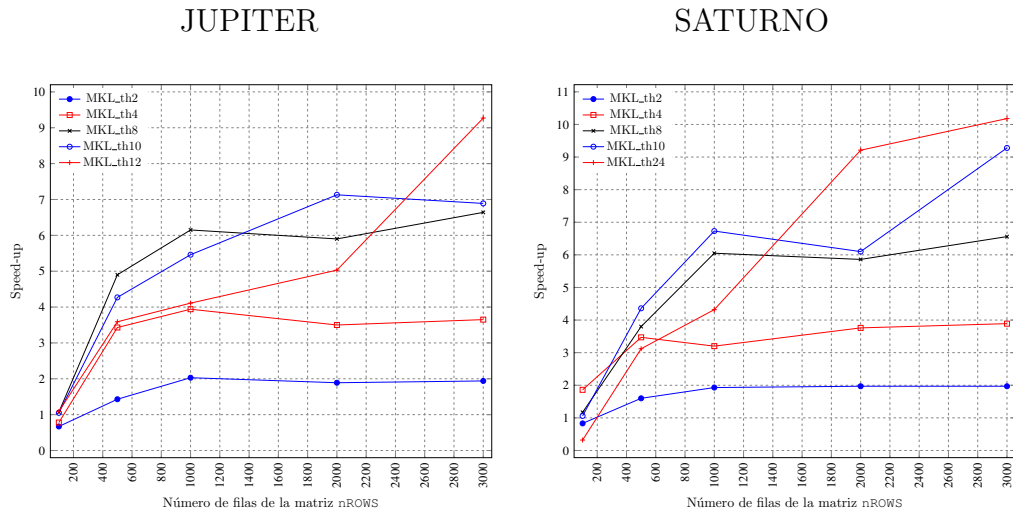


Figura 6.29: Speed-up respecto a MKL secuencial variando el número de hilos asignados a MKL, para diversos tamaños de matrices.

Los siguientes experimentos en esta sección están encaminados a mostrar el uso del simulador para el estudio del rendimiento del algoritmo de la multiplicación de matrices por bloques. Para ello, la matriz B que forma parte del producto $AB = C$ se va a dividir por columnas en un determinado número de bloques. Interesa conocer cuál es el tamaño óptimo de dichos bloques, por lo que será necesario repetir las ejecuciones empleando varias estrategias. Comenzaremos creando el modelo MATMULT_COLS_50 mostrado en la figura 6.30.

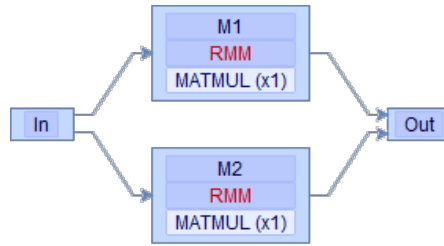


Figura 6.30: Representación en el simulador del modelo MATMULT_COLS_50 de multiplicación de matrices $AB = C$, dividiendo por columnas la matriz B en dos bloques de igual tamaño.

Este modelo permite resolver las dos multiplicaciones necesarias tras dividir por columnas la matriz B en dos bloques de igual tamaño. Como vimos en las secciones anteriores, dado un modelo, el simulador puede elaborar las rutas que permiten resolverlo. En la figura 6.31 se observan las que se originan en el caso de dos grupos.

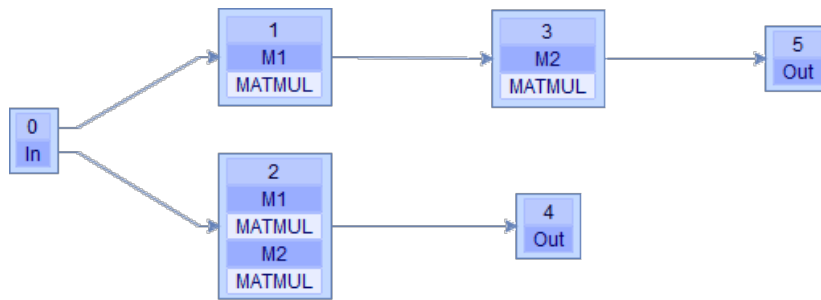


Figura 6.31: Árbol de rutas de ordenación y agrupación de cálculos para resolver una multiplicación de matrices por bloques mediante el modelo de la figura 6.30.

A continuación creamos dos nuevos modelos para resolver la multiplicación cuando la matriz original B se divide por columnas en tres y en cinco bloques,

con tamaños 33 y 20% del original respectivamente. La figura 6.32 muestra el modelo MATMULT_COLS_35 y la figura 6.33 el MATMULT_COLS_20.

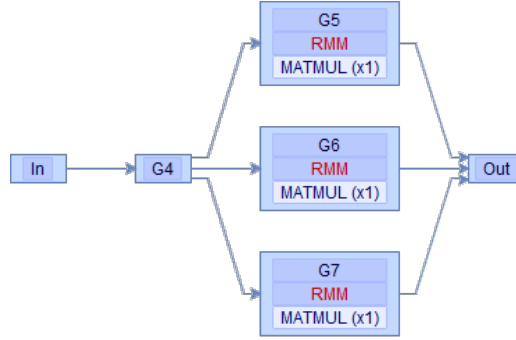


Figura 6.32: Representación del modelo MATMULT_COLS_35 de multiplicación de matrices $AB = C$, dividiendo por columnas la matriz B en tres bloques.

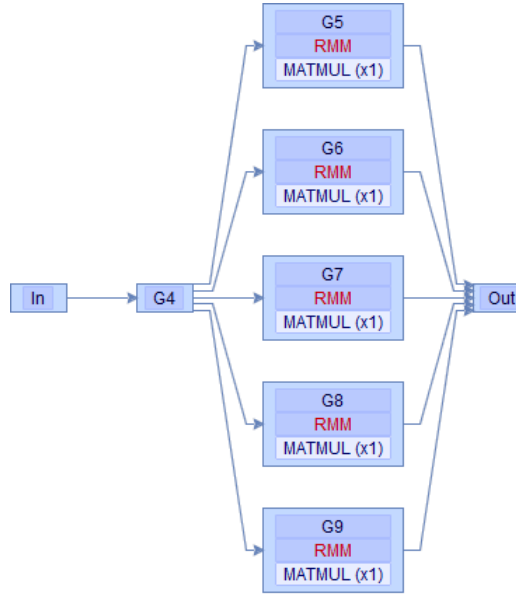


Figura 6.33: Representación del modelo MATMULT_COLS_20 de multiplicación de matrices $AB = C$, dividiendo por columnas la matriz B en cinco bloques.

Los nuevos modelos pueden resolverse siguiendo tantas estrategias como indiquen sus árboles de rutas. Para los experimentos siguientes vamos a considerar aquellas ejecuciones que resuelven de manera simultánea todos los grupos que componen los modelos, y compararemos los rendimientos con la versión que realiza la multiplicación sin descomposición por bloques. Dichas rutas se muestran en las figuras 6.34, 6.35 y 6.36.

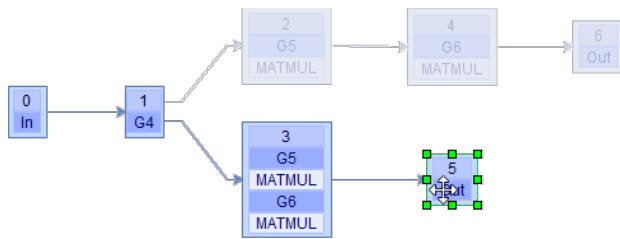


Figura 6.34: Selección de la ruta que aplica paralelismo de grupos para resolver una multiplicación de matrices por bloques mediante el modelo de la figura 6.30.

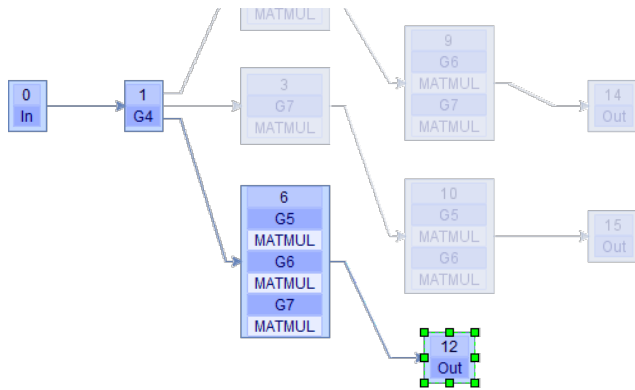


Figura 6.35: Selección de la ruta que aplica paralelismo de grupos para resolver una multiplicación de matrices por bloques mediante el modelo de la figura 6.32.

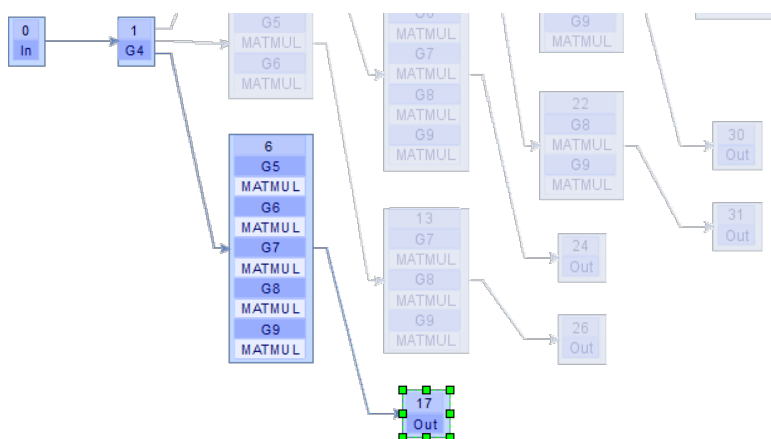


Figura 6.36: Selección de la ruta que aplica paralelismo de grupos para resolver una multiplicación de matrices por bloques mediante el modelo de la figura 6.33.

Cada grupo en los citados modelos manipula un bloque de la matriz original. Si observamos la figura 6.37 comprobamos que el número de columnas de B se divide por dos en cada escenario. Así, la multiplicación con una matriz de tamaño 1000×1000 en el modelo inicial, se resuelve mediante dos multiplicaciones con matrices de 1000×500 en el nuevo modelo.

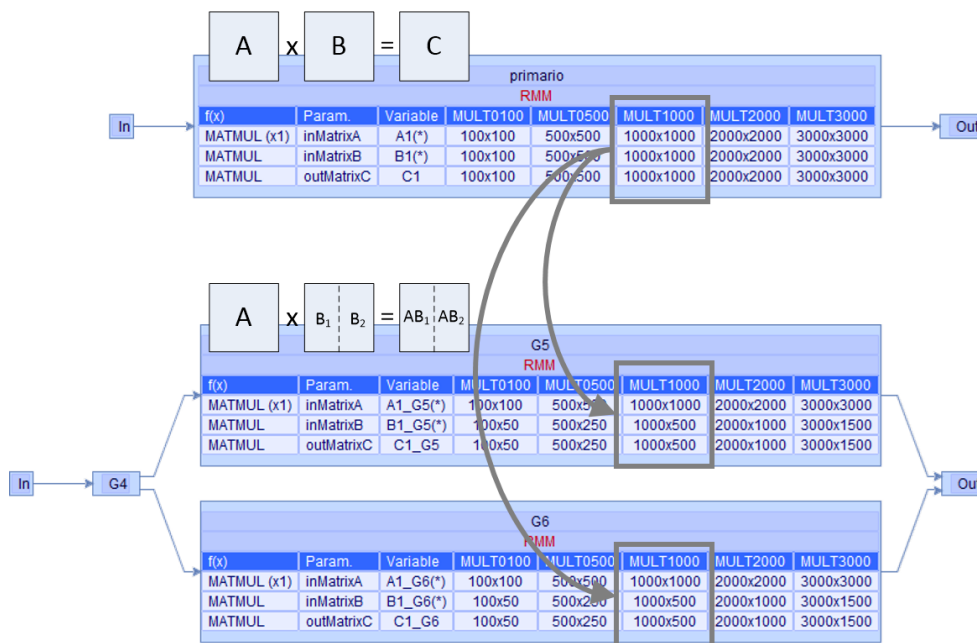


Figura 6.37: Modelos MATMULT (arriba) y MATMULT_COLS_50 (abajo) para la multiplicación de matrices $AB = C$ en el simulador. Una multiplicación de 1000×1000 se reemplaza por dos multiplicaciones 1000×500 .

Para facilitar la tarea al usuario, se permite indicar al simulador que resuelva varios modelos en una sola ejecución. En este experimento concreto basta con seleccionar los tres modelos creados en la pantalla de configuración del editor de modelos. La tabla 6.40 muestra de manera esquemática la información más relevante del archivo de configuración generado.

Parámetro	Valores
Modelos a simular	{ MATMULT_COLS_50, MATMULT_COLS_35, MATMULT_COLS_20 }
Modo de ejecución	Múltiple
Ruta	Preseleccionada

Tabla 6.40: Configuración aplicable a la ejecución de tres modelos de multiplicación de matrices por bloques, con una ruta preseleccionada.

Con esta configuración el simulador realiza múltiples ejecuciones, tantas como combinaciones de modelos, escenarios y parámetros algorítmicos. Tras la ejecución de cada modelo se generan archivos de registro para cada escenario, donde se pueden consultar los tiempos de ejecución obtenidos para cada combinación de parámetros algorítmicos. Además, estos tiempos se almacenan en la base de datos. La tabla 6.41 muestra las mejores combinaciones de threads de primer y segundo nivel de paralelismo, OpenMP y MKL respectivamente, obtenidos a partir de las simulaciones en la plataforma SATURNO. Se observa que con matrices pequeñas la multiplicación directa es la más eficiente empleando MKL. Sin embargo, aumentando el tamaño de las matrices se obtiene una ventaja notable al dividir la matriz original a la vez que se introduce paralelismo para resolver simultáneamente los grupos que manejan matrices de tamaños menores que el de la matriz original. Por ejemplo, para matrices de dimensiones 3000×3000 la mejor estrategia es la aplicada en el modelo MATMULT_COLS_20, que divide por bloques la matriz B en tamaños que representan el 20% del tamaño original, generando por tanto cinco matrices que se pueden multiplicar a la vez asignando cinco threads al paralelismo OpenMP y cuatro al paralelismo de MKL.

nROWS	MATMUL		COLS50	COLS35	COLS20
	1×4	1×20	2×12	3×8	5×4
100	0.00043	0.00412	0.00065	0.00058	0.00072
500	0.01099	0.00643	0.01770	0.01010	0.00702
1000	0.12732	0.06158	0.03780	0.04037	0.03883
2000	0.60477	0.27485	0.27300	0.24414	0.28912
3000	1.96197	0.78933	0.74721	0.68834	0.57356

Tabla 6.41: Comparación de los tiempos de ejecución obtenidos en SATURNO al simular los modelos de la multiplicación de matrices tradicional (MATMUL) y por bloques (COLS50, COLS35, COLS20), para varios tamaños de matrices (nROWS).

6.2.1.2 Experimentos en sistemas multiGPU

En este apartado se incluyen experimentos que muestran el uso de varias GPUs en la resolución de la multiplicación de matrices por bloques. El simulador utiliza la librería MAGMA para asignar tareas de cómputo a las GPUs disponibles (seis en el caso de la plataforma hardware JUPITER).

La función `magma_print_environment()` permite recopilar los detalles del hardware instalado que puede ser explotado mediante la librería MAGMA. El listado 6.1 muestra la información obtenida al ejecutarse en JUPITER, donde se observan las seis GPUs (devices 0 al 5), sus modelos y características.

Listado 6.1 Resultado del comando `magma_print_environment()` de la librería MAGMA para recopilar información del hardware.

```
MAGMA 2.2.0  compiled for CUDA capability >= 2.0,
32-bit magma_int_t,
64-bit pointer.
CUDA runtime 7050, driver 7050.

OpenMP threads 12.
MKL 2017.0.1,
MKL threads 12.

device 0: GeForce GTX 590, 1215.0 MHz clock, 1535.7 MiB memory,
capability 2.0
device 1: Tesla C2075, 1147.0 MHz clock, 5375.4 MiB memory,
capability 2.0
device 2: GeForce GTX 590, 1215.0 MHz clock, 1535.7 MiB memory,
capability 2.0
device 3: GeForce GTX 590, 1215.0 MHz clock, 1535.7 MiB memory,
capability 2.0
device 4: GeForce GTX 590, 1215.0 MHz clock, 1535.7 MiB memory,
capability 2.0
device 5: Tesla C2075, 1147.0 MHz clock, 5375.4 MiB memory,
capability 2.0

Fri Jan 15 18:47:51 2021
```

Para verificar el uso de varias GPUs en el simulador usaremos el modelo `MATMULT_COLS_35` que resuelve una multiplicación de matrices $AB = C$ dividiendo por columnas la matriz B en tres bloques (figura 6.32). Al disponer de seis GPUs, es posible resolver simultáneamente los tres grupos que componen dicho modelo asignando las multiplicaciones incluidas en cada rutina a una GPU diferente. Para esta ordenación de los cálculos el simulador se guiará por la ruta paralela mostrada en la figura 6.35.

La tabla 6.42 muestra el script correspondiente a los experimentos que se van a realizar. Se asignan tres threads al paralelismo OpenMP para permitir la resolución simultánea de los grupos del modelo. Para el paralelismo MKL se asignan hasta cuatro threads para no exceder el número de 12 cores físicos de JUPITER. Para MAGMA se indica el uso de tres GPUs.

Parámetro algorítmico	Valores
Thread del primer nivel (OpenMP)	{3}
Thread del segundo nivel (MKL)	{1,2,3,4}
Número de GPUs	{3}
Librería	{1 : <i>MKL</i> , 7 : <i>MAGMA</i> }

Tabla 6.42: Script creado para la simulación del modelo de la multiplicación de matrices por bloques empleando las librerías MKL y MAGMA. Se asignan tres threads al paralelismo OpenMP para la resolución simultánea de los grupos, tres GPUs para MAGMA y hasta cuatro threads para MKL. Plataforma JUPITER con 12 cores físicos.

La tabla 6.43 muestra los tiempos de ejecución obtenidos, donde se puede observar el mejor rendimiento ofrecido por las GPUs cuando el tamaño de las matrices crece. Con matrices de 500 filas el mejor tiempo se obtiene con MKL mediante la asignación de tres threads.

nROWS	MKL				MAGMA
	3 × 1	3 × 2	3 × 3	3 × 4	3 GPUs
500	0.01019	0.00635	0.00350	0.00403	0.01055
1000	0.05926	0.02977	0.03592	0.02950	0.01796
2000	0.36205	0.16690	0.12079	0.13858	0.07952
3000	1.09032	0.86692	0.72478	0.58676	0.22992

Tabla 6.43: Comparación de los tiempos de ejecución del modelo de la multiplicación de matrices por bloques (MATMULT_COLS_35) obtenidos en JUPITER con varios tamaños de matrices (nROWS×nCOLS), variando el número de hilos asignados a MKL y con tres GPUs explotadas con MAGMA.

En la versión actual del simulador, en una ejecución que requiera el uso de n GPUs de manera simultánea, el software utiliza las n primeras que aparecen en la lista obtenida con la función `magma_print_environment()`. En la simulación mostrada en esta sección, en la que se requieren 3 GPUs, se seleccionan dos GeForce GTX 590 (devices 0 y 2) y una Tesla C2075 (device 1).

Usaremos el simulador para demostrar la influencia del tipo de GPU en el rendimiento obtenido al resolver los grupos. Usaremos la información de tiempos de ejecución que el simulador registra en la resolución de cada grupo. Para ilustrarlo, creamos la tabla 6.44 que muestra algunos de los tiempos de ejecución recogidos en la tabla 6.43, a los que se ha añadido el correspondiente desglose por grupos.

nROWS	MKL				MAGMA
	3×1	3×2	3×3	3×4	3 GPU's
2000	0.36205	0.16690	0.12079	0.13858	0.07952
G5	0.36173	0.16156	0.11757	0.09029	0.07550
G6	0.35431	0.16166	0.12054	0.11709	0.04293
G7	0.32139	0.13978	0.10287	0.08498	0.07027
3000	1.09032	0.86692	0.72478	0.58676	0.22992
G5	1.08003	0.86657	0.71487	0.29032	0.20032
G6	1.07855	0.53282	0.72442	0.57703	0.13511
G7	0.88808	0.79364	0.32048	0.32993	0.22352

Tabla 6.44: Desglose por grupos de los tiempos de ejecución del modelo de la multiplicación de matrices por bloques (MATMULT_COLS_35) en JUPITER con varios tamaños de matrices (nROWS×nCOLS), variando el número de hilos asignados a MKL y con tres GPUs explotadas con MAGMA.

Si observamos el conjunto de datos referido al escenario de matrices de 3000 filas, el tiempo de ejecución del modelo completo con MAGMA es de 0.22992 segundos. Este tiempo se obtiene con el cálculo en paralelo de los grupos *G5*, *G6* y *G7*, cuyos tiempos han sido de 0.20032, 0.13511 y 0.22352 segundos respectivamente. Dado que los grupos tratan matrices de igual tamaño, sería de esperar unos tiempos de resolución similares. Sin embargo observamos que dos de los valores están en torno a 0.2 segundos, y un valor alrededor de 0.1 segundos, lo que se puede explicar por el hecho de que en la lista de GPUs usadas en el experimento encontramos dos tecnologías diferentes, una Tesla C2075 y dos GeForce GTX 590.

Podemos concluir que, cuando se dispone de un conjunto heterogéneo de GPUs, se debe realizar un análisis detallado que asegure la asignación más ventajosa de cálculos a los nodos de cómputo, decisión que viene determinada por el tipo de los datos que se manejan y de las prestaciones del hardware, como indicamos en una de las líneas de trabajos futuros.

6.2.2 Multiplicación de matrices: algoritmo de Strassen

Frente a la resolución tradicional basada en tres bucles, con un coste computacional de orden $\theta(n^3)$, la multiplicación de Strassen, que data de 1969 [281], consigue un coste computacional de orden $\theta(n^{2.8074})$. Se conocen otros algorit-

mos incluso más eficientes para la multiplicación de matrices [166, 183], e incluso adaptaciones del algoritmo tradicional a los nuevos sistemas computacionales. La inclusión en esta sección de experimentos del algoritmo de Strassen en concreto viene motivada por su idoneidad para ser representado como un modelo compuesto por grupos de operaciones que pueden ser resueltos en paralelo y ser asignados a las diferentes unidades de computación disponibles.

El algoritmo de Strassen establece que, dadas dos matrices cuadradas A y B que consideramos por simplificación de dimensiones $A, B \in 2^n \times 2^n$, se puede realizar la multiplicación $C = AB$ dividiéndolas en bloques de igual tamaño $A_{ij}, B_{ij} \in 2^{n-1} \times 2^{n-1}$:

$$A = \left[\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array} \right], B = \left[\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array} \right], C = \left[\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array} \right]$$

Con este planteamiento se necesitan ocho multiplicaciones y cuatro sumas para obtener la multiplicación de las matrices originales:

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

El algoritmo de Strassen define un nuevo conjunto de matrices M_k :

$$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$

$$M_2 = (A_{2,1} + A_{2,2})B_{1,1}$$

$$M_3 = A_{1,1}(B_{1,2} - B_{2,2})$$

$$M_4 = A_{2,2}(B_{2,1} + B_{1,1})$$

$$M_5 = (A_{1,1} + A_{1,2})B_{2,2}$$

$$M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$

$$M_7 = (A_{1,2} + A_{1,1})(B_{1,1} + B_{1,2})$$

Estas matrices M_k se usan a continuación para obtener las $C_{i,j}$ finales:

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$

$$C_{1,2} = M_3 + M_5$$

$$C_{2,1} = M_2 + M_4$$

$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

Las matrices M_k se pueden calcular de manera independiente y, por tanto, en paralelo.

El grafo asociado a los cálculos del algoritmo de Strassen representado en la figura 6.38 muestra las matrices a calcular y sus dependencias, lo que permite determinar el orden en el que se deben resolver cada una de ellas.

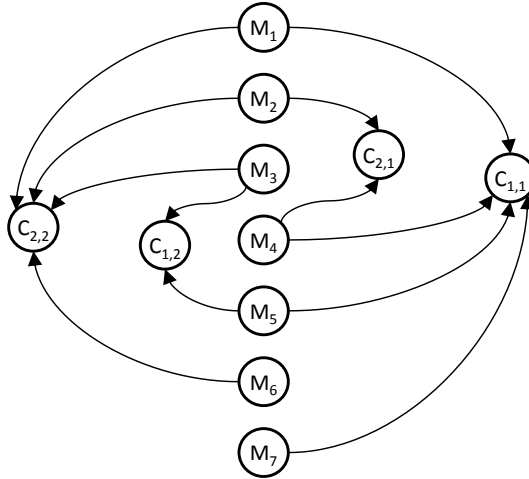


Figura 6.38: Grafo de los cálculos en el algoritmo de Strassen.

Para representar dicho grafo en el simulador se necesitan tres rutinas:

- RDivide: Contiene las operaciones suma y resta de matrices necesarias para formar los operandos de las siete multiplicaciones de matrices que permiten obtener M_k , $k = 1 \dots 7$.
- RMM: Contiene una multiplicación de matrices.
- RCombine: Ejecuta las operaciones suma y resta para obtener las $C_{i,j}$ finales.

Estas rutinas permiten crear en el simulador todos los grupos que forman el modelo del algoritmo de Strassen, al que hemos nombrado como STRASSEN, y que se encuentra representado en la figura 6.39.

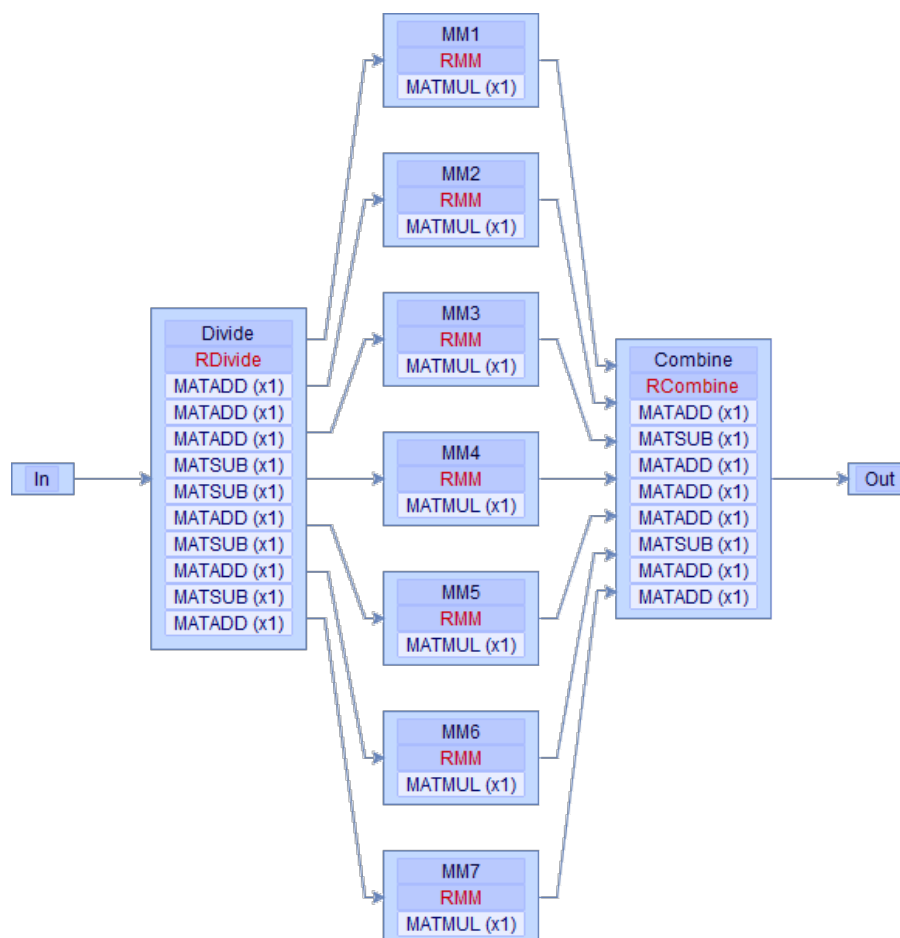


Figura 6.39: Representación gráfica en el simulador del modelo STRASSEN de multiplicación de matrices sin recursión mediante el algoritmo de Strassen. Además del nombre de las rutinas, se muestran las funciones que las componen.

En la figura 6.40 se observan todas las posibilidades para la resolución del algoritmo de Strassen sin recursión, es decir, sin que ninguna de las multiplicaciones de matrices presentes en el algoritmo se resuelvan a su vez mediante Strassen.

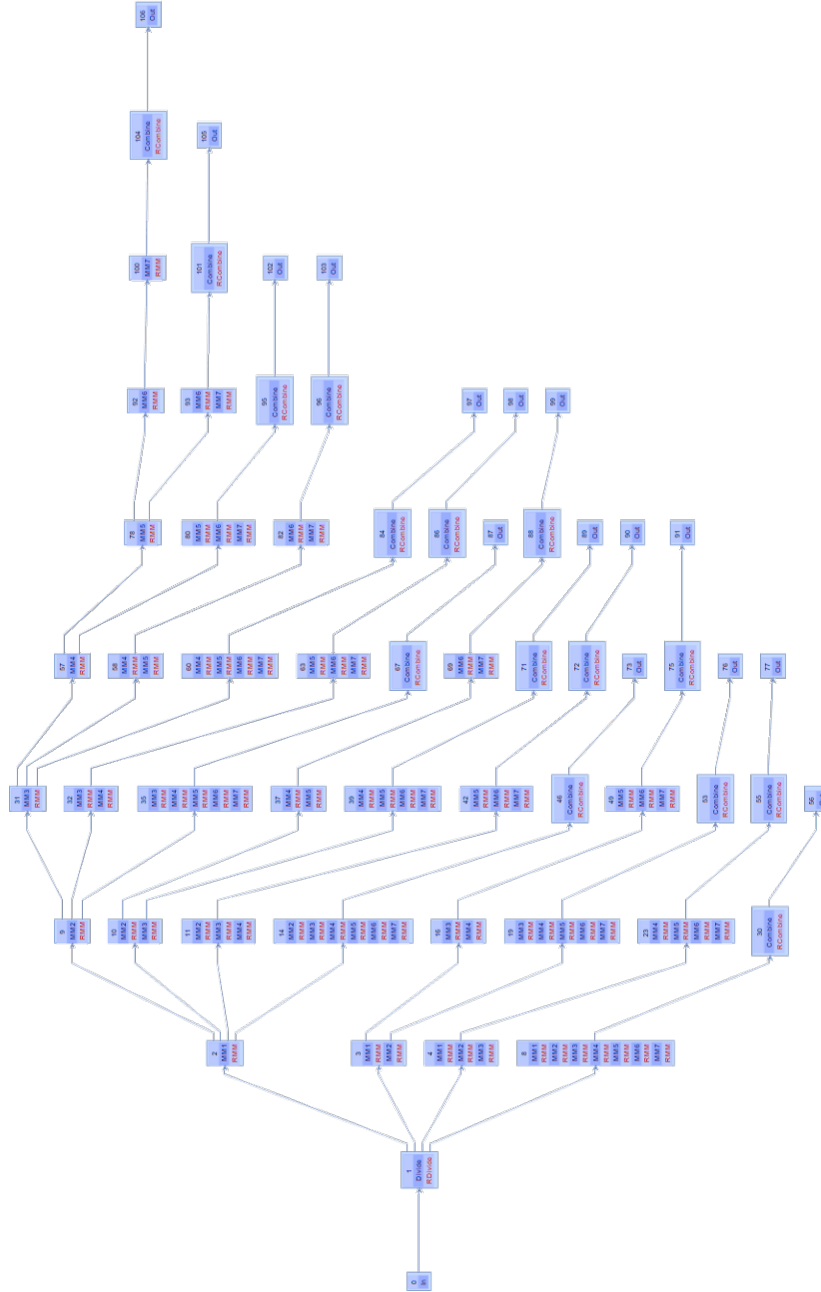


Figura 6.40: Árbol de rutas que representa las alternativas de ordenación y agrupación de cálculos para resolver una multiplicación de matrices mediante el algoritmo de Strassen como el representado en el modelo de la figura 6.39.

6.2.2.1 Experimentos en sistema con CPU monocore

Con objeto de comprobar la eficiencia del algoritmo de Strassen planteamos un conjunto inicial de experimentos usando el simulador con el modelo MATMUL representado en la figura 6.28, que fue desarrollado en la sección 6.2.1 para el estudio de la multiplicación por bloques. Posteriormente repetiremos ejecuciones con el modelo STRASSEN. Para la primera ejecución indicaremos al simulador que use un solo thread, simulando un sistema hardware monocore, lo que nos permitirá observar la eficiencia real del algoritmo sin influencia de las mejoras que pueda aportar el paralelismo. La ejecución en secuencia de todos los grupos que forman el modelo de Strassen corresponde en el simulador a la ruta mostrada en la figura 6.41. Para configurar el modelo de modo que utilice únicamente dicha ruta necesitamos acceder al visor de rutas, tal y como se explica en la sección A.8.17 del anexo, y seleccionarla explícitamente.

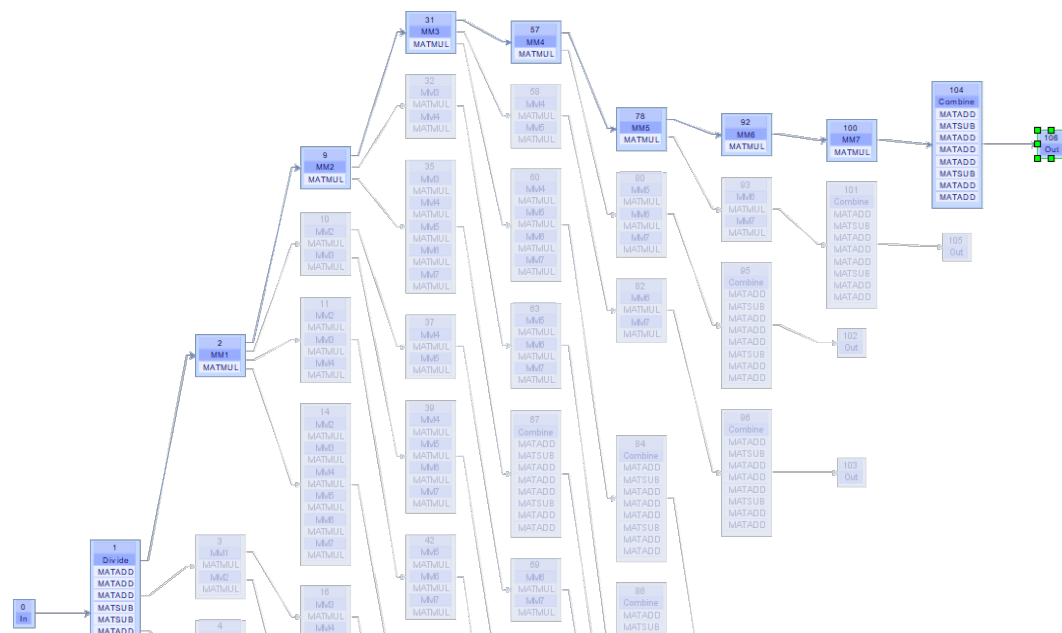


Figura 6.41: Representación gráfica en el simulador de la ruta de ejecución del modelo STRASSEN de multiplicación de matrices mediante el algoritmo de Strassen, aplicable a sistemas con CPU monocore, donde todos los grupos se ejecutan en secuencia.

En esta sección usaremos escenarios con matrices grandes, con tamaños de 6000×6000 , 8000×8000 y 12000×12000 en el modelo de la multiplicación de matrices tradicional. Por tanto, según la definición de Strassen, las multiplicaciones manejadas en dicho algoritmo serán de tamaños 3000×3000 , 4000×4000 y 6000×6000 respectivamente. Dado que queremos realizar las simulaciones con unos parámetros algorítmicos concretos (MKL y un thread simulando un sistema monocore), podemos usar el modo simple descrito en la sección 5.6.2.2. La tabla 6.45 muestra los valores de la configuración del simulador, indicando el modo de simulación simple y los valores de los parámetros a emplear. En este modo el simulador realizará una única ejecución por cada escenario, empleando en todos ellos los mismos valores de los parámetros algorítmicos.

Parámetro	Valores
Modelos a simular	{ STRASSEN }
Modo de ejecución	Simple
Ruta	Preseleccionada
Número de threads de primer nivel (OpenMP)	{ 1 }
Número de threads de segundo nivel (MKL)	{ 1 }
Número de GPUs	{ 0 }
Librería	{ 1 : <i>MKL</i> }

Tabla 6.45: Configuración aplicable a la ejecución del modelo de la multiplicación mediante Strassen. En este experimento la ruta preseleccionada ejecuta en secuencia los grupos, asignando un único core a la librería MKL.

La tabla 6.46 muestra los resultados comparados obtenidos en ambas plataformas (JUPITER y SATURNO).

nROWS \times nCOLS	JUPITER		SATURNO	
	MATMUL	STRASSEN	MATMUL	STRASSEN
6000×6000	22.56067	21.20005	54.36038	50.13080
8000×8000	53.34177	47.80797	130.18680	114.81141
12000×12000	179.10999	160.26036	428.79211	379.44492

Tabla 6.46: Tiempos de ejecución (en segundos) obtenidos con el simulador en una multiplicación sin bloques (modelo MATMUL) y mediante el algoritmo de Strassen (modelo STRASSEN). Matrices cuadradas con dispersión del 30% en las plataformas JUPITER y SATURNO, empleando un único core y MKL.

Se observa la reducción de los tiempos de ejecución obtenida al realizar la multiplicación de matrices usando el algoritmo de Strassen frente al método convencional. En matrices de tamaño 6000×6000 la mejora se sitúa en el 8% en SATURNO y en el 6% en el caso de JUPITER. Para matrices de mayor tamaño, se mejora el rendimiento en un 12% en SATURNO y un 10% en JUPITER, muy cerca del 12.5% teórico.

Repetiremos ahora los experimentos usando la librería MAGMA para aprovechar las GPUs instaladas en las plataformas JUPITER y SATURNO. Para ello volvemos a modificar la configuración del simulador de acuerdo a los valores mostrados en la tabla 6.47.

Parámetro	Valores
Modelos a simular	{ MATMUL, STRASSEN }
Modo de ejecución	Simple
Ruta	Preseleccionada
Número de threads de primer nivel (OpenMP)	{ 1 }
Número de GPUs	{ 1 }
Librería	{ 7 : MAGMA }

Tabla 6.47: Configuración aplicable a la ejecución de los modelos de la multiplicación sin bloques y Strassen. Se preselecciona la ruta secuencial, sin paralelismo de grupos, y usando una GPU a través de la librería MAGMA.

Los resultados de los experimentos están recogidos en la tabla 6.48 y representan los tiempos obtenidos empleando una GeForce GTX 590 en la plataforma JUPITER, y una Tesla K20c en SATURNO.

nROWS×nCOLS	JUPITER		SATURNO	
	MATMUL	STRASSEN	MATMUL	STRASSEN
6000 × 6000	1.82795	3.36921	1.48523	4.14287
8000 × 8000	4.10491	5.56739	3.35258	6.47299
12000 × 12000	12.82721	16.04158	6.74356	16.88393

Tabla 6.48: Tiempos de ejecución (en segundos) obtenidos con el simulador en una multiplicación sin bloques (modelo MATMUL) y mediante el algoritmo de Strassen (modelo STRASSEN). Plataformas JUPITER y SATURNO empleando una GPU mediante la librería MAGMA.

Además, el modelo STRASSEN_R1 también se ejecutará de acuerdo a una ruta secuencial, pues en todo momento estamos considerando un sistema moncore. Dicha ruta se muestra en la figura 6.43.

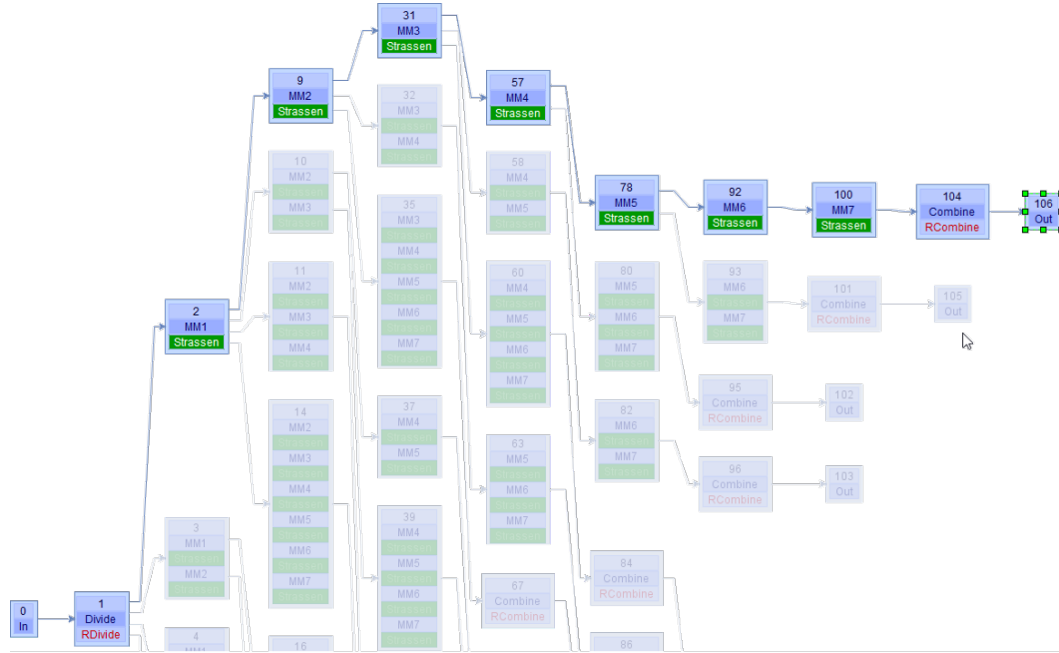


Figura 6.43: Representación gráfica en el simulador de la ruta de ejecución del modelo STRASSEN_R1 de multiplicación de matrices mediante el algoritmo de Strassen con un nivel de recursión, aplicable a sistemas moncore.

La tabla 6.49 muestra que añadir un nivel de recursión en el algoritmo de Strassen no mejora los rendimientos para los tamaños de matrices usados en los experimentos. Esto se debe a que la ventaja obtenida en las multiplicaciones usando matrices cuyos tamaños son la mitad del original no consigue compensar la sobrecarga que supone añadir las operaciones de suma y resta adicionales que completan el algoritmo de Strassen que resuelve cada grupo. Ahora bien, con matrices de tamaño 12000×12000 se empieza a observar en JUPITER una mejora en los tiempos de ejecución. En este caso, y debido a la recursión, las matrices con las que se trabaja son de 3000×3000 , que es un tamaño con el que hemos observado en los experimentos una mejora de rendimiento que está entre el 6 y el 8% según la plataforma.

nROWS	JUPITER			SATURNO		
	MATMUL	STRASSEN		MATMUL	STRASSEN	
		R0	R1		R0	R1
6000	22.56067	21.20005	22.179228	54.36038	50.13080	58.838894
8000	53.34177	47.80797	48.730221	130.18680	114.81141	122.493195
12000	179.10999	160.26036	157.767395	428.79211	379.44492	410.243134

Tabla 6.49: Tiempos de ejecución (en segundos) obtenidos con el simulador en una multiplicación tradicional (modelo MATMUL) y mediante el algoritmo de Strassen sin recursión (R0) y con un nivel de recursión (R1). Matrices cuadradas con dispersión del 30% en las plataformas hardware JUPITER y SATURNO empleando un único core.

6.2.2.2 Experimentos en sistema con CPU multicore

Debido a que MKL es una librería *multi-threading*, es posible explotar su paralelismo implícito y obtener mejores tiempos de ejecución, como hemos visto en los resultados obtenidos en los experimentos en secciones anteriores. Con la introducción de este tipo de paralelismo será difícil conseguir que el algoritmo de Strassen pueda ofrecer ventajas frente a este tipo de librerías tan optimizadas. Para realizar experimentos que nos permitan obtener conclusiones, configuramos de nuevo el simulador conforme se muestra en la tabla 6.50, forzando un nuevo juego de simulaciones con asignaciones de dos y tres threads en las llamadas a la rutina DGEMM de multiplicación de matrices implementada en la librería MKL.

Parámetro	Valores
Modelos a simular	{ STRASSEN }
Modo de ejecución	Simple
Ruta	Preseleccionada
Número de threads de primer nivel (OpenMP)	{ 1 }
Número de threads de segundo nivel (MKL)	{ 2,3 }
Número de GPUs	{ 0 }
Librería	{ 1 : MKL }

Tabla 6.50: Configuración aplicable a la ejecución del modelo de la multiplicación mediante Strassen. En este experimento la ruta preseleccionada es la secuencial, sin paralelismo de grupos, empleando dos y tres threads en las llamadas a la multiplicación de matrices DGEMM de la librería MKL.

La tabla 6.51 muestra los resultados de los experimentos con los nuevos parámetros algorítmicos aplicados a la ejecución de los modelos MATMUL y STRASSEN en la plataforma SATURNO. Observamos que las reducciones de los tiempos de ejecución que se obtenían con el algoritmo de Strassen frente a una multiplicación convencional en sistemas moncore, que se vieron en la sección anterior, son menores conforme aumentamos el número de threads de dos a tres. Asignar tres threads el algoritmo de Strassen ya no ofrece ventajas, salvo para tamaños grandes.

nROWS×nCOLS	2 threads			3 threads		
	MATMUL	STRASSEN	%	MATMUL	STRASSEN	%
6000 × 6000	29.24644	26.59713	9.1	19.22396	19.84433	-3.2
8000 × 8000	70.83429	66.17148	6.6	44.32339	45.53305	-2.7
12000 × 12000	224.74800	205.56395	8.5	147.06793	143.12148	2.7

Tabla 6.51: Tiempos de ejecución (en segundos) obtenidos con el simulador para una multiplicación tradicional sin bloques (modelo MATMUL) y mediante el algoritmo de Strassen (modelo STRASSEN). Plataforma hardware SATURNO, asignando dos y tres threads.

De igual modo ocurre en JUPITER (tabla 6.52), donde los resultados muestran un porcentaje de mejora que se reduce conforme se produce un incremento en el número de threads, que en el caso de matrices 12000×12000 pasa del 9.2% con dos threads a 5.8% con tres threads, cuando en sistemas con un solo thread la ventaja del algoritmo de Strassen estaba en el 10.5%.

nROWS×nCOLS	2 threads			3 threads		
	MATMUL	STRASSEN	%	MATMUL	STRASSEN	%
6000 × 6000	11.31462	10.67342	5.7	7.69943	7.40049	3.9
8000 × 8000	26.73003	24.68509	7.7	18.07492	17.12385	5.3
12000 × 12000	89.84947	81.57596	9.2	70.71327	66.61833	5.8

Tabla 6.52: Tiempos de ejecución (en segundos) obtenidos con el simulador en una multiplicación tradicional sin bloques (modelo MATMUL) y mediante el algoritmo de Strassen (modelo STRASSEN). Plataforma hardware JUPITER, asignando dos y tres threads.

En cualquier caso es importante resaltar que no es un objetivo de este trabajo conseguir una implementación de Strassen que mejore a MKL, sino mostrar cómo el simulador puede ser una herramienta de gran utilidad en el análisis del comportamiento de las rutinas ante diferentes escenarios de ejecución.

6.3 Conclusiones

En este capítulo se ha mostrado la utilidad del simulador a través de su aplicación a problemas reales dentro del área del análisis de sistemas multicuerpo y del álgebra matricial. Hemos visto cómo se representan los modelos, entendidos estos como un conjunto de cálculos y sus dependencias, y cómo es posible realizar ejecuciones siguiendo una o varias estrategias de agrupación de cálculos denominadas rutas. Hemos comprobado que, a la vista de los resultados, un usuario puede determinar la agrupación de cálculos que consigue obtener los menores tiempos de ejecución. Esta funcionalidad es especialmente útil en modelos sencillos, con pocas rutas, donde se quieren comparar rutas que el usuario puede conocer de antemano que pueden ser las más eficientes. Para modelos complejos, la herramienta de autooptimización resulta de mayor utilidad, dado que analiza el árbol de rutas completo en busca de la mejor elección que aproveche los recursos del hardware disponible.

En los ejemplos mostrados dentro del área de los sistemas multicuerpo hemos visto cómo la información obtenida permite a un experto en mecanismos conocer cuál es la forma más adecuada de plantear su algoritmo de resolución, ya que el simulador aporta indicaciones de en qué momento debe incluir paralelismo y qué librería se comporta mejor.

En el área del álgebra matricial se ha mostrado que con el simulador se puede realizar un estudio del comportamiento que van a tener diferentes rutinas equivalentes, como la multiplicación MKL y la multiplicación Strassen, para resolver un problema en diferentes plataformas, lo que permite no solamente escoger los mejores parámetros algorítmicos de cada una de ellas en cada situación, sino también poder elegir qué rutina/algoritmo es la más apropiada en cada caso (polialgoritmo).

La información obtenida a partir de las simulaciones puede servir para construir una tabla de decisión que puede ser enlazada en rutinas que incluyan una opción de autooptimización para determinar su mejor configuración. Hemos visto cómo, en general, en un sistema multicore donde se pueden emplear dos librerías paralelas se debería conformar, para una rutina que resuelve un determinado número de grupos, un esquema de posibles decisiones como el mostrado en la tabla 6.53.

RUTINA: G1, G2, G3				
Tamaño	100		500	
Dispersión (%)	30	50	30	50
Ruta	G1-G2-G3	G1-G2-G3	G1-G2+G3	G1+G2+G3
Threads OpenMP	1	1	2	3
Librería	MKL	MKL	PARDISO	MKL
Threads librería	8	8	4	2

Tabla 6.53: Tabla que recoge las mejores configuraciones de ejecución de una rutina que resuelve tres grupos para varios tamaños de matrices y factores de dispersión, obtenidas por medio de simulaciones usando las librerías MKL y PARDISO.

Este supuesto teórico hace referencia a una rutina que se resuelve mediante tres grupos, G1, G2 y G3. La tabla permite determinar, para cada tamaño de matrices y factor de dispersión, la mejor opción de ordenación de los cálculos (ruta), el número de threads OpenMP, la mejor librería y los threads asignados al paralelismo implícito. La cantidad de parámetros que se deben determinar dependerá del tipo de rutina y del hardware. Por ejemplo, en el caso de rutinas anidadas el nivel de recursión es otro factor que debe determinarse, y en el caso de plataformas con GPUs, habría que contemplar el número de GPUs asignadas a la resolución de cada grupo.

Capítulo 7

Conclusiones, trabajo futuro y contribuciones

Este capítulo recoge las conclusiones obtenidas de acuerdo a los objetivos planteados en esta tesis y desvela líneas de trabajo futuras que profundizan en esta misma línea de investigación. También se detallan las publicaciones generadas durante la elaboración de este trabajo.

7.1 Conclusiones

La heterogeneidad de las plataformas hardware actuales que combinan CPU multicores con otros elementos de computación paralela, como son las GPUs, ha permitido abordar la resolución de numerosos problemas científicos complejos. Si bien en muchos casos ya existían los modelos matemáticos correspondientes a dichos problemas, su resolución numérica no era abordable en un tiempo aceptable debido a la limitada capacidad de cómputo que ofrecían los ordenadores de generaciones anteriores. Sin embargo, las plataformas modernas admiten la aplicación de técnicas de paralelismo para resolver este tipo de problemas mediante el reparto de los cálculos entre varias unidades de cómputo, con la consiguiente reducción de los tiempos de ejecución. En concreto, esta tesis planteaba en su inicio el estudio y optimización de algoritmos paralelos para la resolución de sis-

temas multicuerpo que se estudian en ingeniería mecánica. Con carácter general, se planteaba proporcionar una herramienta de software que ayudara a expertos en mecanismos, no necesariamente conocedores de las técnicas de paralelismo, a encontrar el código óptimo que resolviera problemas planteados mediante un modelo matemático representativo de sistemas multicuerpo reales.

En esta línea, se planteaba como primer objetivo la necesidad de analizar la posibilidad real de optimizar la computación de las simulaciones de dichos sistemas sobre diversas plataformas computacionales, con un enfoque especial en la explotación eficiente del paralelismo. Para ello se ha colaborado con el Departamento de Ingeniería Mecánica de la Universidad Politécnica de Cartagena, que está trabajando en una metodología para el modelado automático de sistemas mediante un análisis estructural basado en ecuaciones de grupo. Esta técnica permite determinar el conjunto de grupos estructurales que, calculados en un determinado orden, resuelven el modelo en su totalidad. El planteamiento de un problema con este tipo de enfoque nos permitió pensar en una solución general en la que, dado un grafo que represente grupos de cálculos en sus nodos, y líneas dirigidas para indicar las relaciones de dependencia, se pudieran encontrar las diferentes opciones de reordenar y agrupar los cálculos respetando la dependencia temporal entre los grupos. El software desarrollado en esta tesis incorpora una herramienta visual para la captura del modelo a simular y obtiene, también de manera gráfica, el árbol de rutas de resolución. Dicho árbol constituye la base sobre la que se construye la funcionalidad del simulador, dado que la elaboración del mismo determina qué grupos de cálculos se pueden resolver de manera simultánea y, por tanto, en paralelo.

El segundo objetivo planteado en esta investigación consistía en la introducción efectiva de técnicas de autooptimización que permitieran aumentar la eficiencia en la ejecución. La aplicación de técnicas de paralelismo por sí mismo no tiene por qué suponer la solución óptima. Esta depende en gran medida del tipo de hardware sobre el que se realizan las ejecuciones, y también de las librerías de álgebra lineal empleadas en los cálculos. La selección de los parámetros algorítmicos, así como de las técnicas de paralelismo aplicadas, será más o menos acertada en función del correcto balance entre todos ellos. El simulador desarrollado incluye una funcionalidad de autooptimización capaz de encontrar la ruta más rápida

de entre las disponibles, proponiendo en cada nodo la librería que ofrece mejor rendimiento teórico y los parámetros algorítmicos requeridos para ello. La ruta óptima se considera teórica en cuanto que toma como base los tiempos de ejecución de las librerías en simulaciones aisladas variando los parámetros algorítmicos dentro de un conjunto de entrenamiento preestablecido.

El tercer objetivo, resultado de las investigaciones desarrolladas en los objetivos previos, ha sido el desarrollo de una herramienta que facilite la simulación de ejecuciones en las plataformas computacionales que se indiquen. Esta herramienta permite analizar las técnicas de optimización y autooptimización propuestas en los objetivos anteriores. En la sección de experimentos se muestran ejemplos de este tipo de ejecución, con casos concretos de reducción de los tiempos de ejecución frente a otras alternativas no optimizadas. La ventaja para un usuario no experto es evidente en cuanto que le sugiere tanto la manera de reordenar el código, como los parámetros de paralelismo a emplear. Como se ha comentado anteriormente, estos parámetros dependen del hardware y de la librería. En cuanto al hardware, el simulador contempla el número de threads en CPU, y el número de GPUs disponibles. En cuanto a las librerías, se puede especificar el número de threads asignados a los códigos que implementan. Un proceso de autooptimización permite determinar qué parte de los cálculos se asigna a una CPU y qué parte se resolverá en una GPU.

Finalmente, comprobada la eficacia del simulador en el ámbito de los sistemas multicuerpo, se ha abordado la aplicación de las ideas anteriores, incluido el uso del propio simulador, a rutinas de álgebra lineal con una descomposición similar a la realizada en simulaciones de multicuerpos. Se han presentado en este sentido resultados de las metodologías aplicadas sobre la rutina de multiplicación de matrices, tanto en su resolución por bloques, como por medio de Strassen.

Así pues, se ha demostrado la utilidad del simulador como herramienta de apoyo a científicos con interés en elaborar códigos optimizados para resolver determinados problemas. La tarea de obtener los modelos matemáticos correspondientes a cada problema reside en el propio usuario, como experto en el campo de investigación correspondiente. A partir de un modelo, el software proporciona al usuario una herramienta visual para representar su algoritmo y proponerle la reordenación del código y los cálculos susceptibles de paralelización.

Además de la aplicación práctica a problemas reales, se deriva una utilidad de carácter meramente educativo, dado que es posible crear modelos no necesariamente reales que pueden mostrar a un usuario o estudiante de manera interactiva las ventajas que supone la aplicación de determinadas técnicas de paralelismo, así como la de seleccionar una librería adecuada según el tipo de cálculo a realizar en una determinada plataforma de hardware.

7.2 Trabajo futuro

La investigación realizada en esta tesis ha conducido al desarrollo de una herramienta de software que incluye las funcionalidades de acuerdo a los objetivos fijados, pero a partir de la cual se pretende avanzar introduciendo mejoras y nuevas funcionalidades, tanto en su interfaz gráfico como en el proceso de simulación. A continuación detallamos un conjunto de tareas que ya han sido identificadas y marcadas como objetivos para ser incluidas en futuras versiones del simulador:

- **Gestión de familias de GPUs:** Una limitación del software actual consiste en que, a pesar de que puede ejecutar cálculos en más de una GPU, la selección de las mismas se realiza a partir de la lista de dispositivos libres, sin considerar la heterogeneidad de las prestaciones y memoria de las diferentes GPUs instaladas. Para salvar dicha limitación, se propone mejorar el software mediante la aplicación de una heurística concreta de reparto de trabajo. Por ejemplo, se podría asignar la unidad más potente en el caso de requerir una sola GPU. Y en el caso de que sean necesarias más de una GPU, un proceso las podría seleccionar de manera que los cálculos de mayor carga computacional o mayores requerimientos de memoria se asignen primero a las GPUs más potentes, realizando las siguientes asignaciones en orden descendente de prestaciones y tamaños de las matrices. Se trataría por tanto de extender la actual asignación de cálculos a una GPU basado en el número de dispositivo por una asignación mejorada a partir de una lista de registros que contenga los dispositivos y sus características técnicas más relevantes, como el número de cores, velocidad de procesador y cantidad de memoria.

- Manejo de paralelismo en otro tipo de coprocesadores: La versión actual del software gestiona paralelismo mediante el uso de los cores de la CPU y mediante ejecuciones en GPUs, pero no contempla el uso de otros dispositivos paralelos. Un ejemplo serían los MIC (*Many Integrated Cores*) de Intel. En este caso particular, debido a su arquitectura basada en los procesadores tradicionales para propósito general, sus núcleos se pueden programar utilizando código fuente C, C++ y FORTRAN estándar, lo que permite una integración directa en el simulador. Se estudiará, en general, la inclusión de nuevos tipos de hardware, modificando a la vez en el simulador el proceso de selección de parámetros algorítmicos y de asignación de cálculos para contemplar los nuevos tipo de unidades de cómputo.
- Mejora en la búsqueda de parámetros algorítmicos: El proceso actual se basa en una búsqueda exhaustiva en la base de datos de entrenamiento, limitando el espacio de búsqueda mediante el descarte de antemano de aquellas combinaciones de threads y número de GPUs que puedan exceder las capacidades del hardware. Se propone la inclusión de técnicas alternativas que, mediante la aplicación de heurísticas, permitan reducir los tiempos de búsqueda en el espacio de opciones.
- Incorporación de técnicas de modelado teórico de los tiempos de ejecución: De esta manera se podría reducir el tiempo de experimentación con las rutinas básicas, y también guiar la generación del árbol de posibles agrupaciones y asignaciones de las tareas a realizar para un determinado modelo, posibilitando así el análisis de modelos de mayor tamaño. También es posible combinar el modelado teórico con técnicas experimentales, utilizándose el modelado en una primera fase usando fórmulas de rendimiento para determinar las configuraciones más prometedoras, para las que se llevarían a cabo los experimentos.
- Ajuste dinámico de los parámetros algorítmicos: Se trata de comprobar, en cada etapa en la que se divide la simulación, los recursos disponibles del hardware en ese momento y decidir si se debe continuar por la misma, o saltar a otra ruta que permita completar el resto de cálculos pendientes. Se podrían obtener de esta manera mejoras de rendimiento en sistemas donde nuestro problema se ejecuta compartiendo recursos con otras aplicaciones.

- Rutinas de cálculo personalizadas: Se permitirá a un usuario la posibilidad de añadir sus propias rutinas, o bien enlazar librerías de terceros para resolver determinados cálculos. De esta manera se elimina la restricción actual por la que se puede usar un conjunto limitado de librerías y rutinas incorporadas en el simulador. Por ejemplo, en el caso de la multiplicación de matrices, actualmente se puede resolver mediante la función DGEMM de MKL o de MAGMA, que son las que están instaladas. Las librerías disponibles en la versión actual del simulador se han seleccionado de manera que ofrezcan diferentes prestaciones según el tamaño de las matrices y sus factores dispersión, y con ello validar el modo en el que el software es capaz de seleccionar una u otra librería en un proceso de autooptimización. Se trabajará en dos vías:
 - La inclusión de un editor donde capturar un código fuente. Esta opción requiere un compilador de C o FORTRAN en el hardware donde se ejecuta el simulador. El intercambio de datos entre el simulador y el nuevo código introducido por el usuario se realizará siguiendo una API estándar que se definirá con este fin.
 - Permitir el enlace en el simulador de un código ya compilado. Como en la opción anterior, se debe establecer un interfaz para el intercambio de información (básicamente el formato de los argumentos de entrada y salida de las rutinas externas que se vayan a incorporar).
- Generación de un código optimizado: La información generada por el simulador consiste en un archivo de registro que contiene el resultado de la simulación y el informe de autooptimización en su caso. Adicionalmente los tiempos de ejecución obtenidos se almacenan en una base de datos. En futuras versiones se propone la generación de un código fuente de manera que el usuario pueda copiarlo e incluirlo directamente en su aplicación.
- Herramientas de exportación: Se incorporará la funcionalidad de exportar un subconjunto de la información almacenada en la base de datos a través de archivos de texto plano o con formato CSV (*comma-separated values*). Este tipo de formatos permite que la información sea manipulada fácilmente por otras herramientas de análisis de datos y creación de gráficos disponibles en el mercado.

- Consulta de datos históricos: En el proceso de autooptimización, cuando no se encuentre ningún dato de entrenamiento para alguna función en concreto, se propone recurrir a la base de datos de tiempos reales de la función que se hayan obtenido durante la simulación de algún modelo real. Esta acción tendría preferencia sobre la solución actual de realizar una interpolación en base a los datos de entrenamiento existentes.
- Especialización de los scripts: Se podrán asignar diferentes scripts a los grupos, y no solo a nivel de modelo. El objetivo es permitir que un usuario pueda comprobar el rendimiento de la simulación resolviendo grupos con diferentes librerías indicando específicamente los valores de los parámetros algorítmicos a nivel de cada grupo. La versión actual del simulador contempla esta opción en una ejecución autooptimizado, pero no en los modos simple y múltiple, donde los parámetros algorítmicos aplican por igual a todos los grupos.
- Mejoras en el proceso de creación del árbol de rutas: En modelos complejos, con un elevado número de grupos y dependencias, el árbol de rutas puede llegar a contener una gran cantidad de ramas y requerir tiempos de cómputo elevados para su creación. Se investigarán dos alternativas de mejora del procedimiento actual: por un lado una estrategia de añadir en primer lugar los nodos más prometedores, realizando la poda de las ramas que se puedan descartar a priori; en segundo lugar, ofrecer al usuario la posibilidad de seleccionar criterios para la generación que permitan filtrar y limitar el número de rutas a generar.
- Extensión a paralelización MPI: El objetivo de esta tarea es incluir en el simulador la opción de adaptar los algoritmos para su ejecución en entornos de memoria distribuida, como clusters, NUMA y redes de sistemas heterogéneos. En este escenario se recurre a un paradigma de programación paralela basada en paso de mensajes, en la que los algoritmos deben incluir códigos de comunicación y sincronización. Surgen de ellos la necesidad de nuevos parámetros algorítmicos, derivados del estándar MPI, como el número de procesos. Además, las especificaciones del hardware que deben ser conocidas por el simulador deben incluir ahora las características de las redes de comunicaciones, con el ancho de banda y la latencia como factores críticos.

- Integración del simulador con otras herramientas: Se va a trabajar en la creación de un interfaz del simulador con otras herramientas que incluyan procesos de optimización basados en tiempos experimentales. Con ello se pretende reusar la información generada por el simulador al optimizar un modelo o rutina, para optimizar a su vez rutinas de nivel superior que la puedan utilizar.
- Integración del simulador con un módulo de análisis estructural computacional de sistemas multicuerpo: El paralelismo híbrido desarrollado en esta tesis para la simulación cinemática de sistemas multicuerpo parte del análisis de su estructura cinemática y de la definición, para cada grupo obtenido, de las rutinas específicas del análisis cinemático. Para analistas que actualmente tengan desarrolladas sus propias rutinas de análisis cinemático o dinámico de sistemas multicuerpo, pero no saquen provecho de su estructura cinemática, les será de gran ayuda disponer de un módulo de análisis estructural que obtenga, de forma automática, la estructura cinemática del sistema a analizar.
- Estandarización de la creación de rutinas de análisis y la definición de modelos: Dado que el simulador desarrollado se basa en un análisis modular de sistemas multicuerpo y, en general, de problemas de álgebra lineal, se deberán crear estándares en la definición de rutinas y de modelos de grupos estructurales de topologías determinadas que permitan su intercambio entre desarrolladores de diferentes grupos de investigación, de ámbito nacional o internacional. De esta forma, cualquier investigador podrá disponer de una librería de rutinas y de modelos de grupos estructurales que facilitarían la creación de nuevos sistemas multicuerpo a analizar. De igual forma, cualquier analista en esta “comunidad multibody” podrá refrescar modelos de sistemas mecánicos ya analizados cambiando una o varias rutinas, o incluso grupos estructurales completos, por versiones optimizadas ofrecidas por otros desarrolladores.
- Convertir esta herramienta de simulación en herramienta de análisis: En su estado actual, la herramienta desarrollada simula la resolución de problemas de álgebra lineal ejecutando las funciones básicas que se indican en las correspondientes rutinas, agrupadas en los diferentes grupos que definen una

determinada estructura cinemática. Será muy interesante evolucionar hacia una herramienta donde las diferentes rutinas lean, desde ficheros creados por los analistas para cada grupo estructural, las matrices necesarias para el análisis que se pretende llevar a cabo: cinemático, dinámico, síntesis óptima, etc, de forma que se obtengan resultados numéricos que permitan estudiar el comportamiento del sistema mecánico analizado. Un paso adicional en esta evolución será la integración de esta herramienta de análisis con software de terceros (OpenSceneGraph, Blender) para la representación tridimensional del comportamiento del sistema mecánico.

- Aplicación a la optimización de rutinas de análisis: Aparte de los avances mencionados, deseables para la mejora de esta herramienta de simulación, también sería importante aprovechar todo su potencial actual para profundizar en la optimización de las rutinas actuales de análisis y síntesis en MBS. A modo de ejemplo, se propone desglosar cada grupo en subgrupos con rutinas específicas para análisis de posición, velocidad y aceleración. También se podrá estudiar qué agrupaciones de funciones se pueden paralelizar en formulaciones cinemáticas y dinámicas globales para optimizar las rutinas de análisis, concepto fundamental por el que se ha desarrollado esta herramienta.

7.3 Difusión

En esta sección se incluyen, por orden cronológico, las publicaciones y trabajos para congresos generados durante la elaboración del presente trabajo y que han justificado el interés en la línea de investigación seguida en esta tesis doctoral:

- P. Segado, J. C. Cano, J. Cuenca, D. Giménez, M. Saura, P. Martínez. Eficiencia de la paralelización multihilo en el análisis cinemático de sistemas multicuerpo basado en Ecuaciones de Grupo. XXI Congreso Nacional de Ingeniería Mecánica, Elche, España, (2016).

Este estudio surge en las etapas iniciales de la colaboración mantenida con la Universidad Politécnica de Cartagena. Es elaborado por su departamento

de Ingeniería Mecánica y describe la influencia de la dimensión de los grupos estructurales en el tiempo de resolución de sistemas multicuerpo y las posibles mejoras surgidas con la implementación de técnicas de paralelización multihilo.

- Gregorio Bernabé, José-Carlos Cano, Javier Cuenca, Antonio Flores, Domingo Giménez, Mariano Saura-Sánchez and Pablo Segado-Cabezas. Exploiting Hybrid Parallelism in the Kinematic Analysis of Multibody Systems Based on Group Equations. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

Describe la investigación llevada a cabo durante el trabajo de fin de máster de título *Optimización de algoritmos paralelos para análisis cinemático de sistemas multicuerpo basado en Ecuaciones de Grupo*, dentro de la especialidad de Arquitectura de Altas Prestaciones y Supercomputación del Máster de Nuevas Tecnologías en Informática de la Universidad de Murcia. En él se analiza un código en FORTRAN desarrollado por el Departamento de Ingeniería Mecánica de la Universidad Politécnica de Cartagena para la simulación de una plataforma robótica de Stewart. Se trabajó en las modificaciones necesarias para añadir paralelismo en la resolución de ciertos grupos estructurales y, mediante experimentos en ordenadores personales con CPU multinúcleo, en Raspberry Pi y en las plataformas del cluster *Heterosolar*, se mostraban las mejoras obtenidas variando el tipo de librería y configuraciones paralelas. Este trabajo constituye el germen de la presente tesis doctoral.

- José-Carlos Cano, Javier Cuenca, Domingo Giménez, Mariano Saura-Sánchez and Pablo Segado-Cabezas. A Parallel Simulator for the Kinematic Analysis of Multibody Systems Based on Group Equations. Proceedings of the 18th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE, July 9–14, 2018.

Este trabajo representa la generalización de la investigación a cualquier tipo de sistemas multicuerpo, e introduce los primeros bocetos de la aplicación de escritorio que formará parte del futuro simulador. Se presenta la captura de cualquier tipo de mecanismo mediante la adición de grupos y sus dependencias.

- José-Carlos Cano, Javier Cuenca, Domingo Giménez, Mariano Saura-Sánchez, and Pablo Segado-Cabezoz. A parallel simulator for multibody systems based on group equations. *The Journal of Supercomputing*, 75:1368–1381, 2018. Esta revista se encuentra en 2019 en el Q1 (96/420) de la clasificación de Computer Science del Journal Citation Report.

Este artículo presenta el simulador de la cinemática de MBS y describe su uso para determinar las configuraciones satisfactorias de los parámetros de paralelismo (número de subprocesos OpenMP y de GPU, así como la biblioteca de álgebra lineal) para la obtención del menor tiempo de simulación. Se presenta también la primera versión de la herramienta de autooptimización para ayudar a los usuarios no expertos en la selección de la configuración.

- Jesús Cámara, José-Carlos Cano, Javier Cuenca, Domingo Giménez, Mariano Saura-Sánchez: Adapting a multibody system simulator to auto-tuning linear algebra routines (póster). *SIAM Conference on Parallel Processing for Scientific Computing (PP20)*, Seattle, Washington, U.S., 12-15 Febrero, 2020.

En este póster se presentan las primeras ideas de una de las líneas de trabajo futuro planteadas, consistente en la integración de una metodología de autooptimización jerárquica con el simulador en el ámbito de los sistemas multicuerpo.

Anexo A

Simulador: Manual de uso

Este anexo está dirigido a usuarios nuevos de PARCSIM e incluye una guía práctica de formación con indicaciones paso a paso del uso del software, desde la captura del problema a estudiar hasta el proceso de simulación y posterior análisis de resultados.

A.1 Conceptos básicos

Esta sección describe brevemente los conceptos y términos relacionados con el uso de la aplicación PARCSIM, y que han sido descritos en detalle en la sección 5.2.

A.1.1 Funciones

En PARCSIM se denominan *Funciones* las operaciones y procedimientos de álgebra lineal incorporados en el simulador que pueden ser usadas por un usuario para construir algoritmos de resolución de ciertos problemas numéricos. Entre ellas se pueden encontrar operaciones algebraicas básicas y transformaciones de matrices, como una suma o una traspuesta, o funciones de mayor nivel, normalmente importadas de librerías externas, como por ejemplo la multiplicación de matrices o una descomposición LU. Las funciones se identifican en el simulador

mediante una clave (como por ejemplo MATADD, MATTRN, MATMUL y DGETRF, en el caso de las mencionadas).

A.1.2 Rutinas de usuario

En PARCSIM una *Rutina* representa una secuencia de funciones, creada por un usuario para resolver un determinado problema o una parte de él. Durante la simulación, una rutina ejecuta en secuencia todas las funciones que la componen. Como ejemplo, se puede crear una rutina con el nombre ADDMUL para representar un algoritmo que resuelve una suma y una multiplicación de matrices, en cuyo caso incluiría a las funciones MATADD y MATMUL.

A.1.3 Modelos

Un *Modelo* en PARCSIM es la representación en forma de grafo acíclico del algoritmo de resolución de un problema científico. En este tipo de grafo los nodos representan en el simulador los bloques de instrucciones que ejecutan determinadas secciones del algoritmo, y las líneas dirigidas entre los nodos indican el orden en que se deben resolver.

A.1.4 Grupos

Los *Grupos* en el simulador son los diferentes subsistemas o módulos en los que se divide un algoritmo. El usuario define los cálculos que se deben ejecutar para resolver cada uno de dichos grupos, lo cual se realiza por medio de la asignación de una rutina de usuario. Por ejemplo, un problema que suma y multiplica dos conjuntos de matrices se podría representar por medio de dos grupos que contengan la rutina ADDMUL, cada una de ellos operando sobre uno de los conjuntos.

A.1.5 Variables

Las *Variables* se usan para identificar a las matrices que se usarán como argumentos o parámetros de las funciones que el usuario utiliza en PARCSIM para simular un determinado modelo. Las variables no hacen referencia al tamaño o formato de dichas matrices.

A.1.6 Escenarios

El término *Escenario* hace referencia a la naturaleza del problema a resolver. Un escenario, por tanto, asigna un número de filas y columnas, factor de dispersión y tipo a las variables descritas en la sección anterior.

A.1.7 Scripts

Se denomina *Script* al conjunto de parámetros algorítmicos que el software aplica durante la simulación de un modelo, en concreto el número de threads, la cantidad de GPUs y las librerías de cómputo utilizadas. Los valores de los parámetros se pueden especificar como listas o como rangos.

A.1.8 Rutas

En PARCSIM una *Ruta* especifica una determinada manera de ordenar y agrupar los cálculos, y por tanto los grupos, que resuelven un modelo. Para que una ruta sea válida debe respetar las dependencias que refleja el grafo del modelo. El conjunto de todas las rutas válidas se representa en PARCSIM en forma de árbol, donde cada rama corresponde a una ruta.

A.2 Componentes del software

En la sección 5.8 se describieron los dos componentes que forman el software PARCSIM:

- PARCSIM-MB o Model Builder: Es el interfaz gráfico que facilita la tarea de creación en el software de los modelos que representan problemas numéricos que el usuario quiere simular. También permite especificar la configuración de los parámetros que gestionan la ejecución del software.
- PARCSIM-RUN o Runtime: Es el componente encargado de la simulación. Necesita para su ejecución acceder a los ficheros de texto generados por PARCSIM-MB para almacenar la información sobre los modelos, los escenarios y los parámetros algorítmicos.

A.3 Requisitos del sistema

El componente PARCSIM-MB está desarrollado en JAVA y se distribuye como un archivo empaquetado `.jar`, que incluye el software base y las librerías necesarias. Requiere el JRE (Java Runtime Environment) 8 o superior, y puede ser ejecutado tanto en Windows© como en Linux. Se ha probado su funcionamiento en Windows© 7 y 10, y también en Linux Ubuntu 14.04.

El componente PARCSIM-RUN, encargado de realizar las simulaciones, puede ejecutarse en Windows© y Linux. Se requiere de un programa de instalación para cada uno de dichos sistemas operativos. Este software se encuentra disponible en la dirección http://luna.inf.um.es/grupo_investigacion/software, donde también se puede encontrar la guía de instalación.

A.4 Interfaz gráfico de PARCSIM-MB: vista general

El componente PARCSIM-MB proporciona al usuario una herramienta visual para crear y simular modelos. La figura A.1 muestra un esquema general de su

interfaz gráfico, en el que se pueden distinguir los siguiente elementos principales:

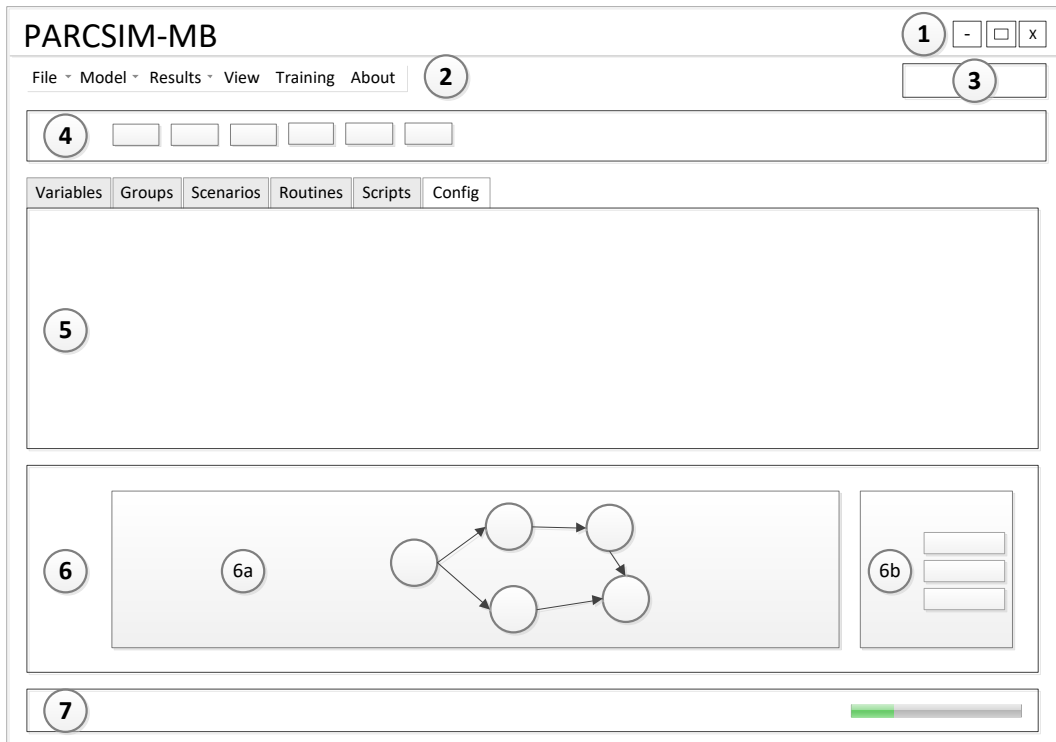


Figura A.1: Vista esquemática del interfaz gráfico que ofrece PARCSIM-MB (Model Builder).

- ① Los botones de ventana, que permiten maximizar y minimizar el área de trabajo.
- ② Una barra de menús con los accesos a las funcionalidades del software. Está organizada en submenús, como se verá en detalle en la sección A.5.
- ③ Un cuadro de texto que muestra el nombre del modelo que se está editando. Al inicio de la aplicación, o cuando no hay ningún modelo activo, se muestra el mensaje `No model selected`.
- ④ Una barra de herramientas que contiene una selección de los accesos directos a las utilidades usadas más habitualmente por un usuario.
- ⑤ El área principal de trabajo, donde se captura la información de los modelos, escenarios y scripts, y donde se establece la configuración general del simulador. El área de trabajo se describe en detalle en la sección A.6.

- ⑥ Una ventana de visualización del grafo correspondiente al modelo que se está editando. En esta parte del interfaz se pueden distinguir dos áreas:
 - 6a) El grafo en sí, que se actualiza automáticamente cuando el usuario crea nuevos grupos o actualiza las dependencias en el modelo.
 - 6a) Una barra de herramientas que contiene accesos directos a las utilidades que permiten modificar la visualización del grafo, como por ejemplo girarlo o cambiar su tamaño. Las funcionalidades incluidas en estos accesos directos se pueden consultar en detalle en la sección A.7.
- ⑦ Una barra de estado que muestra información del hardware y del sistema operativo sobre el que se está ejecutando PARCSIM-MB. Incluye información sobre la memoria consumida por el software.

A.5 Barra de menús

Este componente de PARCSIM-MB distribuye en submenús los enlaces a las funcionalidades del software conforme muestra la figura A.2. A continuación se describen en detalle cada uno de dichos submenús y los elementos que los componen.

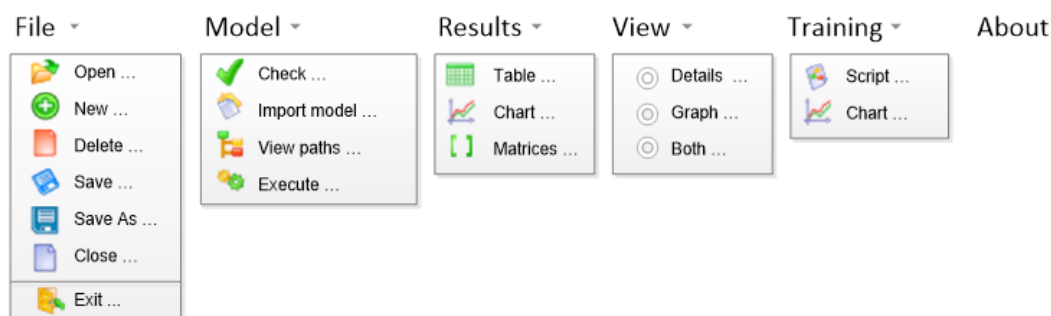









Figura A.2: Barra de menú del interfaz gráfico de PARCSIM-MB. La imagen muestra los submenús desplegados para mostrar los enlaces a las funcionalidades del software.





A.5.1 Menú *File*

Contiene utilidades que permiten trabajar con los ficheros que almacenan los modelos de usuario. Incluye la creación, borrado y actualización de los mismos, así como un enlace para el cierre de la aplicación:

-  Open ... : Permite la selección de un modelo existente. Para ello el software despliega una ventana del sistema operativo para buscar archivos con extensión “.mdl” en el directorio de la aplicación. Una vez seleccionado un fichero, el software lee su contenido y lo incorpora al interfaz gráfico.
-  New ... : Se usa para crear un nuevo modelo. En el caso de que haya un modelo abierto, este se cierra y se prepara el sistema para comenzar la edición con todas las estructuras de datos vacías.
-  Delete ... : Contiene la acción de eliminar un modelo, que se selecciona por medio de una ventana del sistema operativo que muestra los archivos con extensión “.mdl” que se encuentran en el directorio de la aplicación. Tras una confirmación por parte del usuario, se realiza el borrado definitivo de todos los archivos relacionados con dicho modelo. Esta operación no se puede deshacer.
-  Save ... : Esta opción guarda en disco el modelo que se está editando.
-  Save As ... : Permite guardar con un nombre diferente el modelo que se está editando. El modelo original permanece inalterado.
-  Close... : Cierra el modelo que se está editando.
-  Exit... : Esta opción cierra la aplicación. Se solicita confirmación al usuario en el caso de que haya un modelo abierto con modificaciones pendientes de guardar.


A.5.2 Menú *Model*



Este submenú incluye acciones que tienen efecto sobre el modelo activo:

-  Check... : Realiza una comprobación de la consistencia del modelo que se muestra en el editor. Se verifica que se cumplen los siguientes requisitos, que definen un modelo bien formado:
 - Todos los grupos tienen un sucesor.
 - Solo hay un grupo que representa el final del grafo.
 - Los parámetros de todas las funciones usadas en el algoritmo tienen asignados una variable.
 - Los tamaños de las matrices que corresponden con las variables usadas cumplen los criterios de número de filas y columnas en función de las restricciones que impone la operación matricial a realizar.
-  Import model... : Esta opción permite combinar dos modelos. El software muestra una ventana proporcionada por el sistema operativo para la búsqueda de archivos con extensión “.mdl” en el directorio de la aplicación. Una vez seleccionado un archivo, el software añade al modelo actual todos los componentes incluidos en el archivo elegido.
-  View paths... : Genera el árbol de las rutas que permiten la resolución de un modelo y lo representa gráficamente.
-  Execute... : Esta utilidad muestra el interfaz de ejecución de modelos. Contiene una ventana con la información de estado que PARCSIM envía a la consola del sistema operativo durante las simulaciones.

A.5.3 Menú *Results*

Este submenú recoge las utilidades que permiten consultar la información generada por el simulador, tanto de los tiempos de ejecución obtenidos en la resolución de modelos, grupos y funciones, como del contenido de las matrices en cada etapa de la simulación. El usuario dispone de tres modos de consulta:

-  Table ... : Esta opción muestra los resultados en formato de tabla, con una cabecera que incluye un filtro por cada columna. Es posible activar varios filtros simultáneamente.

-  Chart ... : Muestra la información mediante gráficos de líneas que facilitan la comparación visual de rendimientos. El usuario selecciona el hardware, el modelo, el tipo y la dispersión de las matrices empleadas. La gráfica obtenida muestra en el eje *X* los tamaños de las matrices para las que se encuentran resultados y en el eje *Y* los tiempos de ejecución. Cada serie corresponde a una combinación de librería, número de threads y cantidad de GPUs.
-  Matrices ... : Contiene una utilidad para visualizar el contenido de las matrices usadas por el simulador en cualquier etapa de la ejecución.



A.5.4 Menú *View*

En PARCSIM es posible modificar la distribución de la información en el interfaz gráfico para aprovechar de manera adecuada el área de visualización en función de la tarea que el usuario está realizando. Se puede seleccionar entre tres opciones:

- Details: Este modo usa toda la ventana de la aplicación para mostrar los detalles del modelo, los grupos que lo forman, los escenarios, los scripts y los parámetros de configuración del simulador.
- Graph: En esta opción se muestra únicamente el grafo correspondiente al modelo que se va a simular.
- Both: Divide la pantalla horizontalmente en dos paneles que muestran a la vez los detalles del modelo y el grafo asociado. Es posible cambiar el tamaño de las dos áreas desplazando la línea de división usando el ratón.

A.5.5 Menú *Training*

Este submenú contiene las acciones relacionadas con el entrenamiento del simulador que, como vimos en la sección 5.6.2.1, se encuentra disponible a través de un modo de simulación específico en PARCSIM:

-  Script ... : Esta opción permite acceder al editor donde se especifican los escenarios y los scripts de entrenamiento. El simulador utiliza esta información para la obtención de los tiempos de ejecución de las funciones usando los tamaños de matrices y parámetros algorítmicos especificados en este apartado.
-  Chart ... : Contiene un enlace a una herramienta de generación de gráficos de líneas que muestran el rendimiento de las funciones obtenido en el modo de entrenamiento del simulador.

A.6 Área de trabajo

El área de trabajo es la sección del interfaz gráfico donde el usuario crea los modelos, los escenarios y los scripts, y donde se pueden ajustar los parámetros que configuran el simulador. Está organizada mediante pestañas que facilitan la agrupación y el acceso a los diferentes elementos que se pueden modificar. A continuación se describen las diferentes vistas que se activan al seleccionar cada una de dichas pestañas, y la información contenida en todas ellas.

A.6.1 Área de trabajo: *Variables*

En esta vista se editan las variables necesarias para especificar un modelo. Estas variables representan a las matrices que se van a usar como argumentos de las funciones que se ejecutarán durante la simulación como parte del algoritmo de resolución. La figura A.3 muestra los datos que se capturan en esta vista.

Figura A.3: Vista de mantenimiento de las variables de tamaño y las variables del modelo en el interfaz gráfico PARCSIM-MB.

En (1) se capturan las variables del modelo (*Model variables*), que representan los datos (matrices) usados en el algoritmo de cálculo. El usuario puede crear tantas variables como necesite añadiendo su nombre a la lista. Las columnas *Cols* y *Rows* muestran el tamaño de las matrices mediante fórmulas que hacen uso de las variables de tamaño que se crean en (2). Estas variables (*Size variables*) sirven para representar el tamaño del problema, y por tanto tendrán valores distintos para cada escenario.

Por ejemplo, en un sistema multicuerpo que consta de dos elementos de distinto tipo, el proceso de modelado matemático obtendrá un número diferente de coordenadas para representar cada uno de ellos. Se podrán crear entonces las variables `nRowsElemento1` y `nColElemento1` para representar el número de filas y columnas del primer elemento, respectivamente. De igual manera se crearán `nRowsElemento2` y `nColElemento2` para el segundo. Por otro lado, se crearán las variables del modelo, que serán los argumentos de todas las funciones que se deben ejecutar para resolver cada elemento, por ejemplo $\{\text{Matriz}_{11}, \text{Matriz}_{12}, \dots, \text{Matriz}_{1m}\}$ para la rutina que resuelve el primero, y $\{\text{Matriz}_{21}, \text{Matriz}_{22}, \dots, \text{Matriz}_{2n}\}$ para el segundo. Para especificar los tamaños de estas $m+n$ variables se puede hacer mediante referencias a las variables de tamaño `nRowsElemento1`, `nColElemento1`, `nRowsElemento2` y `nColElemento2`, que tendrán valores únicos en cada escenario.

Es posible asignar variables de tamaño a un grupo de variables de modelo de forma masiva. Para ello, en (3) se seleccionan las variables de modelo que se van a modificar, en (4) se introducen los valores que se quiere aplicar y se pulsa en el botón **Apply**. En la sección A.8.15 se muestra un ejemplo de uso de las variables.

A.6.2 Área de trabajo: Groups

Esta vista muestra la información relacionada con la funcionalidad de añadir, eliminar y modificar grupos. Como referencia, el número de grupos que forman un modelo aparece indicado entre paréntesis en el texto de la etiqueta Groups. Si observamos el área etiquetada con ① en la figura A.4, un control desplegable sirve para seleccionar uno de los grupos ya creados en el modelo actual. Para crear un nuevo grupo se debe clicar sobre **New Group**. En ② se capturan el nombre del grupo y el tipo de cálculo que contiene. Como se describió en 5.2.4, un grupo puede ejecutar una rutina que puede contener en su definición funciones y rutinas anidadas u otro modelo (el conjunto de rutinas/funciones que lo definen). En caso de que el grupo resuelva un modelo completo se deberá indicar la ruta en el campo Branch. El borrado de un grupo se realiza seleccionando **Delete Group**.

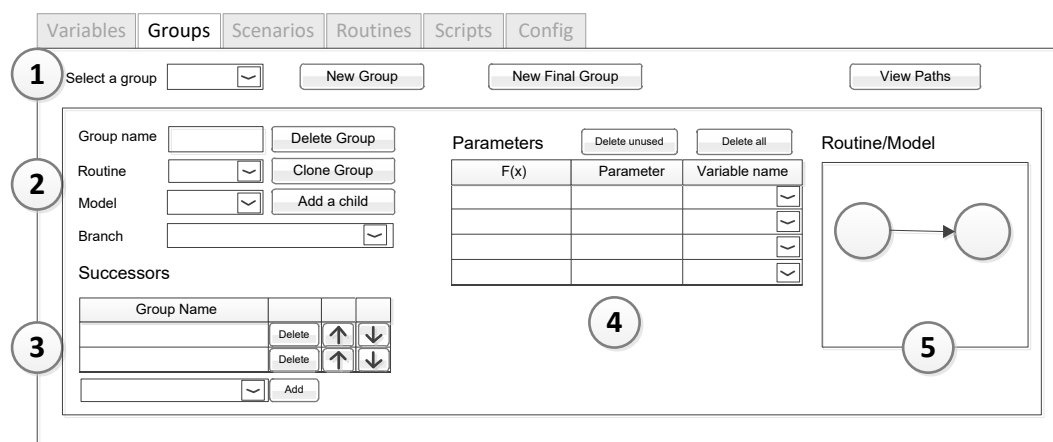


Figura A.4: Vista del mantenimiento de grupos que conforman un modelo en el interfaz gráfico PARCSIM-MB.

En esta vista encontramos también una serie de funciones que facilitan al usuario la tarea de creación de grupos:

- **Clone Group**: Permite crear un número especificado de grupos como copias del grupo seleccionado. El usuario indica la cantidad n de grupos a crear y selecciona si quiere que se hereden también los predecesores y sucesores. Los nuevos grupos se crearán inicialmente con nombres compuestos por la letra G seguidas de un número entero secuencial (posteriormente el usuario puede

cambiar estos nombres asignados por defecto). El proceso de clonado, sin embargo, no actualiza de forma automática todas sus variables asociadas.

- **Add a child**: Crea un nuevo grupo como hijo del grupo seleccionado y añade automáticamente la unión entre ambos. El sistema propone como nombre del nuevo grupo la letra G seguida de un número entero secuencial, y solicita al usuario que modifique o confirme dicho nombre mediante una ventana de diálogo.
- **New final group**: Esta utilidad crea un nodo que actuará como cierre del algoritmo, por lo que todos los grupos que en este momento no tengan sucesor añaden al nuevo grupo como tal. El sistema propone como nombre del nuevo grupo la letra G seguida de un número entero secuencial, y espera a que el usuario modifique o confirme dicho nombre. El grupo recién creado, al ser el final del algoritmo, no tendrá sucesores.

En la zona etiquetada con ③ se indican los sucesores, es decir, el grupo o grupos que se deben ejecutar a continuación del grupo actual, reflejando de este modo la dependencia entre grupos. Los grupos sucesores deben haber sido creados previamente, ya que la selección de los mismos se realiza por medio de un desplegable que muestra los grupos disponibles. Una vez seleccionado, el nuevo sucesor se añade clicando sobre el botón **Add**. La dependencia se puede eliminar con **Delete**.

En ④ se indican los argumentos que usarán las funciones que se van a ejecutar en cada grupo. Se seleccionan mediante un desplegable que muestra todas las variables que se han creado en el correspondiente editor, como se describió en la sección A.6.1. El sistema solicita tantos como requieran las funciones incluidas en la rutina o modelo del grupo activo. Se añaden los botones **Deleted unused** y **Delete all** que permiten borrar los que no se vayan a usar, o todos ellos.

Cuando el grupo ejecuta una rutina, el área marcada como ⑤ muestra un grafo que representa la secuencia de funciones y el despliegue en funciones de las rutinas anidadas que contiene dicha rutina. En caso de ejecutar un modelo, se muestra un grafo de la ruta seleccionada.

A.6.3 Área de trabajo: *Scenarios*

El proceso de crear un escenario permite especificar los tamaños, tipos y factores de dispersión de las matrices que actúan como argumentos de las funciones. Como se observa en la figura A.5, en el área marcada como ① se puede seleccionar un escenario mediante un control desplegable, o bien crear un nuevo escenario pulsando sobre **New Scenario** y tecleando un nombre que lo identifique. En esta misma pantalla las áreas ② y ③ recogen las variables de tamaño y las variables del modelo respectivamente, todas ellas creadas previamente como se describió en la sección A.6.1. Una vez que se introduce un valor para una variable de tamaño en ②, automáticamente se actualiza el número de filas y columnas de las variables del modelo mostradas en ③. El tipo y el factor de dispersión de las matrices se modifica mediante los desplegables incluidos en este interfaz que, en la versión actual del software, afectan a todas las variables por igual. En ④ se puede indicar al simulador que genere la matriz correspondiente a una variable usando valores aleatorios con el tamaño y tipo indicados. Otra opción es seleccionar en ⑤ un fichero existente en disco con los valores de la matriz.

Figura A.5: Vista del mantenimiento de escenarios incluido en el interfaz gráfico PARCSIM-MB.

En la zona etiquetada como ⑥ se puede introducir un valor n que indica el número de iteraciones necesarias para completar la simulación. Por ejemplo, en un modelo que resuelve la posición de la manivela de un cuadrilátero en un determinado instante, la repetición de los cálculos 36001 veces permite obtener el tiempo de ejecución total correspondiente a la simulación de dicho sistema cuando gira de 0 a 360° con incrementos de 0.01° . El valor por defecto es 1.

Durante la simulación de un modelo los valores que contienen las matrices pueden generarse de manera aleatoria al inicio de la ejecución, o se pueden leer de un archivo almacenado en disco clicando en **Open**, o bien introduciendo valores específicos mediante un editor de matrices al que se accede seleccionando **Create**.

A.6.4 Área de trabajo: *Routines*

Las rutinas en PARCSIM son secuencias de cálculos ejecutados en un orden determinado. El usuario puede definir todas las rutinas que necesite haciendo clic sobre **New routine** e introduciendo un nombre identificativo. Para editar una rutina, se puede seleccionar en el desplegable mostrado en el área etiquetada como ① en la figura A.6.

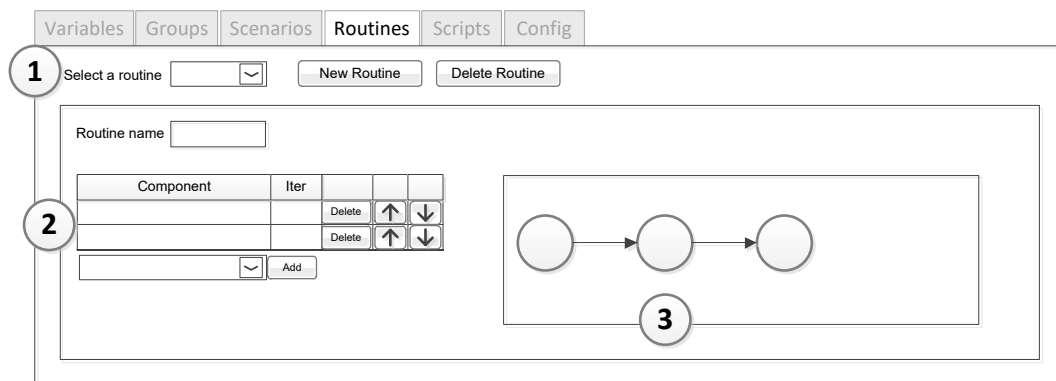


Figura A.6: Vista del mantenimiento de rutinas en el interfaz gráfico PARCSIM-MB.

En ② se añaden los cálculos que contiene dicha rutina (el editor les denomina *Components*). El desplegable muestra las funciones incorporadas en el simulador y las rutinas previamente creadas. Una vez seleccionada una función o una rutina, se añaden mediante el botón **Add**. Es posible añadir la misma función o rutina más de una vez. Para cambiar el orden en el que se ejecutan los componentes se usan los botones **↑** y **↓**. En el campo *Iter* se indica cuántas iteraciones/veces hay que repetir un determinado componente (función o rutina anidada). Esta operación permite simular procesos iterativos como el de Newton-Raphson. El área etiquetada con ③ muestra un grafo actualizado con las funciones que forman la rutina y su ordenación.

A.6.5 Área de trabajo: *Scripts*

Los scripts son los grupos de parámetros algorítmicos usados durante la simulación. Incluyen el número de threads de primer y segundo nivel, el número de GPUs y el tipo de librería a emplear. El simulador ejecutará el modelo para cada combinación de parámetros algorítmicos. El primer nivel de paralelismo se usa para la paralelización explícita (solución simultánea de grupos) y el segundo nivel para la implícita (para ejecución de cálculos en librerías que lo permiten). Se podría utilizar el paralelismo explícito para dividir, además de la ejecución de grupos en paralelo, la resolución de un problema en diferentes tramos. Por ejemplo, para resolver la cinemática de un cuadrilátero articulado, cuando la manivela gira desde 0° hasta 360° en tramos entre $0-90^\circ$, $90-180^\circ$, $180-270^\circ$ y $270-360^\circ$.

El usuario puede crear cualquier número de scripts por medio del correspondiente botón **New Script** del interfaz gráfico mostrado en la figura A.7. El sistema propone por defecto un nuevo nombre compuesto por la letra S seguida de un número secuencial que se muestra al usuario en una ventana de diálogo, donde puede ser modificado.

Para que el simulador use un script es necesario marcar la casilla ☐Active.

1 Script

Script name ☒ Active

	OMP1	OMP2	Library	GPU
From				
To				
Step				
	<input type="button" value="Delete"/>	<input type="button" value="Delete"/>	<input type="button" value="Delete"/>	<input type="button" value="Delete"/>

Values list

		<input type="button" value="Delete"/>
		<input type="button" value="Delete"/>
		<input type="button" value="Delete"/>

Condition

Constant name	Constant value

Figura A.7: Vista del mantenimiento de scripts en el interfaz gráfico PARCSIM-MB.

La información que incluye cada script define el conjunto de valores que pueden adoptar los parámetros algorítmicos. Estos se pueden introducir en dos formatos:

- Como rangos, tal como se observa en ②, donde se especifica el valor inicial, el final y el salto.
- Como una lista de valores, añadiendo líneas a la tabla mostrada en ③.

Ambos modos son excluyentes, de manera que valores introducidos como un rango impiden especificar valores individuales, y viceversa. Para eliminar valores o rangos se puede clicar sobre los botones Delete.

La introducción de valores de los parámetros algorítmicos en formato de rango ofrece comodidad para un usuario, pero puede generar un número elevado de combinaciones. Para reducir dicho número, es posible introducir en ④ una fórmula que represente una expresión booleana que se debe satisfacer para que el simulador utilice dichos valores. Un ejemplo sería el caso en el que los threads de primer y segundo nivel combinados excedieran el número de cores del hardware. En una plataforma con 12 cores, donde los threads OMP1 se han especificado como un rango de 1 a 6, y los threads OMP2 como otro rango de 1 a 6, si queremos limitar las combinaciones a aquellas que no excedan los 12 cores físicos, la fórmula a introducir sería: $OMP1 \cdot OMP2 < 13$. El número de cores se podría almacenar en una constante creada en el área ⑤. Por ejemplo, si el número de cores fuera una constante MAXTHREADS, con valor 12, la fórmula anterior se podría sustituir por $OMP1 \cdot OMP2 < (MAXTHREADS + 1)$.

A.6.6 Área de trabajo: *Config*

Esta vista agrupa la información relativa a la configuración general del simulador. En las opciones generales recogidas en la zona marcada como ① en la figura A.8, encontramos el campo de texto `Hardware`, que permite al usuario identificar la plataforma donde se lanza la simulación. Este valor se guardará en la base de datos junto a los tiempos de ejecución, de manera que se podrán comparar rendimientos de simulaciones asociados a plataformas hardware diferentes.

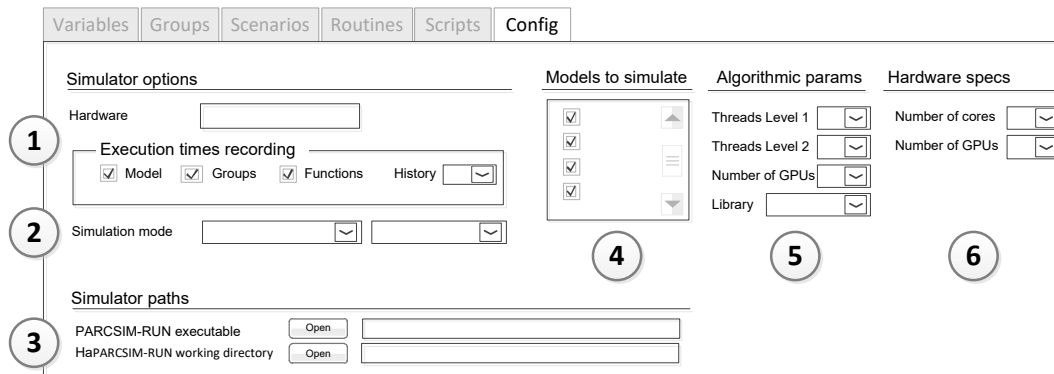


Figura A.8: Vista del editor de las opciones de configuración del simulador.

En esta pantalla también se especifica qué información de tiempos de ejecución se almacena en la base de datos:

- ☐ Model: Marcando esta casilla se activa la grabación en la base de datos de los tiempos de resolución de los modelos completos.
- ☐ Groups: Marcando esta casilla se almacenan los tiempos obtenidos al ejecutar cada grupo del modelo.
- ☐ Functions: Marcando esta casilla se graban los datos de ejecución de cada función en cada grupo.
- History: Es el número máximo de registros que se almacenarán en la base de datos para una determinada clave formada por el modelo, grupo, función, parámetros algorítmicos y escenario. Si en el proceso de grabación se excede este número, se borran de la base de datos los más antiguos para cada clave.

En la zona etiquetada como ② encontramos Simulation Mode, donde se puede elegir entre los cuatro modos de simulación que se describieron en la sección 5.6.2:

- Modo de entrenamiento: En este modo se realizan ejecuciones individuales de todas las funciones disponibles en el simulador, o una selección de ellas. Los cálculos se repiten para todos los escenarios y scripts de entrenamiento y los resultados se almacenan en la base de datos de entrenamiento, que será consultada en el proceso de autooptimización.

- **Modo Simple:** Se simula un modelo (o una lista de modelos) tantas veces como escenarios se hayan definido, aplicando en cada caso los parámetros algorítmicos fijos que se capturan en la zona marcada con (5):
 - Threads del primer nivel de paralelismo.
 - Threads del segundo nivel de paralelismo.
 - Número de GPUs instaladas en el sistema.
 - Librería, seleccionable mediante un desplegable que muestra la lista de todos los paquetes implementados en el simulador.
- **Modo Múltiple:** En este modo se simula un modelo o lista de modelos, con cada uno de sus escenarios y para cada combinación de parámetros algorítmicos obtenidos a partir de los datos definidos en los scripts. De esta manera, a diferencia del modo simple, donde se realiza una ejecución con un conjunto único de parámetros, en el múltiple los parámetros generan varias ejecuciones.
- **Modo autooptimizado:** Es un modo del simulador que decide automáticamente los mejores parámetros algorítmicos para cada escenario, así como la mejor ruta de cálculo basándose en los tiempos de ejecución de las funciones obtenidos en el modo de entrenamiento, y buscando el máximo aprovechamiento del hardware descrito en (6):
 - Número de cores.
 - Número de GPUs disponibles.

Es posible realizar el entrenamiento en un sistema y la simulación en otro.

En la zona etiquetada con (3) se especifican los directorios donde se encuentran el simulador y los ficheros necesarios para su ejecución:







- **Executable:** Especifica el directorio en el que se ha instalado el archivo ejecutable del simulador. Pulsando Open se despliega una ventana del sistema operativo que permite navegar por la estructura de carpetas.

- **Working Directory:** Especifica el directorio en el que se encuentran los archivos que necesita el simulador durante la ejecución (los que contienen la estructura de los diferentes modelos creados por el usuario, así como las rutinas y las funciones). Puede ser una carpeta diferente a la que contiene el ejecutable.

En ④ se especifican los modelos que se simularán en la próxima ejecución del software, y que se seleccionan activando las casillas ☐ junto a cada modelo.

A.7 Visor del grafo de modelos

La zona inferior del interfaz del editor incluye un visor donde se muestra el grafo del modelo, que es actualizado dinámicamente conforme el usuario modifica los grupos o los escenarios. Para cada grupo se visualizan su nombre y la rutina que resuelve. El visor incluye una barra de botones con funcionalidades que modifican y permiten ajustar el aspecto del grafo:

-  **Print** : Envía a la impresora una imagen del grafo mostrado.
-  **Auto layout** : Es el modo por defecto durante la edición de grupos. El software distribuye automáticamente los grupos en la pantalla de acuerdo a las relaciones de dependencia.
-  **Rotate** : Permite alternar entre las orientaciones horizontal y vertical.
-  **Zoom to fit** : Ajusta el tamaño de manera que todos los grupos sean visibles.
-  **Zoom in** : Aumenta el tamaño de los grupos y el texto.
-  **Zoom out** : Reduce el tamaño de los grupos y el texto.
- La casilla ☐ **Show params** se usa para hacer que el grafo muestre en cada nodo las funciones, todos sus parámetros y los tamaños de las matrices para cada escenario asociado al modelo activo.
- Activando la casilla ☐ **Show functions** el grafo muestra en cada nodo, además del nombre de la rutina, las funciones que la componen.

Haciendo clic sobre cualquier grupo mostrado en el grafo se seleccionará dicho grupo en el editor de grupos, permitiendo de esta manera un acceso rápido a su edición.

A.8 Lección paso a paso

Esta sección está planteada como una guía práctica para el uso del simulador PARCSCIM. En ella se describen todos los pasos que se siguen desde la construcción de un modelo hasta su simulación. Por último, se realiza una introducción a la herramienta de análisis de los tiempos de ejecución obtenidos en simulaciones empleando diferentes tamaños de problema, parámetros de paralelismo y librería de cómputo.

A.8.1 Preparación: elección de un problema

A lo largo de este tutorial trabajaremos en la resolución de un problema sencillo, sin interés científico, pero que ayudará al usuario a conocer y familiarizarse con los diferentes conceptos manejados por el simulador. El problema elegido pertenece al ámbito del álgebra matricial y trabaja con tres matrices cuadradas $\{A1, A2, A3\}$ y dos matrices columna $\{B1, B2\}$, como se ha representado en la figura A.9. También se van a usar variables para contener los resultados intermedios de las operaciones, en concreto las matrices cuadradas $\{C, T\}$ y las columna $\{X1, X2, D, R\}$.

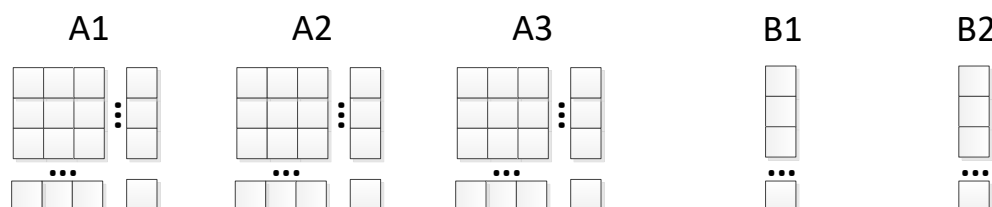


Figura A.9: Representación de la información matricial manejada por el simulador para la resolución del problema numérico de este tutorial.

La solución de nuestro problema comienza sumando las dos primeras matrices $A1$ y $A2$. El resultado, junto a $B1$, constituye un sistemas de ecuaciones que debe ser resuelto. Por otro lado, $A2$ y la segunda matriz columna $B2$ forman otro sistema de ecuaciones. Se suman $X1$ y $X2$, soluciones de los dos sistemas descritos anteriormente, y se obtiene el término independiente de un nuevo sistema de ecuaciones que usará la traspuesta de la tercera matriz $A3$. En la figura A.10(a) se representa el algoritmo que resuelve este problema realizando los cálculos en secuencia.

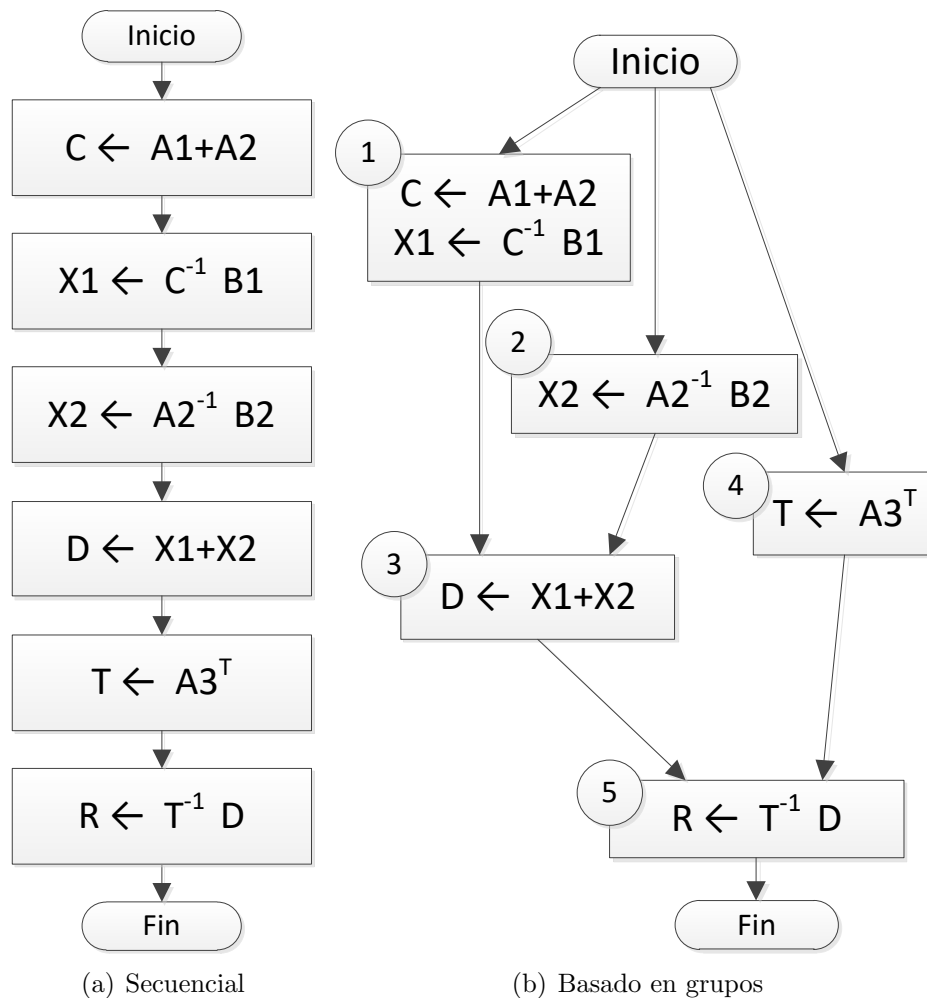


Figura A.10: Algoritmo de resolución del problema numérico de ejemplo usado en este tutorial.

Un análisis en detalle de la figura nos permite identificar operaciones que no comparten información y que, por tanto, se pueden agrupar y calcular de manera independiente. En A.10(b) observamos una variante del algoritmo donde se han segmentado los cálculos en cinco grupos, numerados del 1 al 5, unidos mediante líneas dirigidas para representar las relaciones de dependencia. Con carácter general, en el simulador los grupos pueden contener una sola operación o más de una, como vemos que ocurre con el grupo 1, que incluye una suma de matrices y una resolución de un sistema de ecuaciones. Esta forma de representación en forma de grafo nos ayuda a identificar las posibilidades de ejecución simultánea de grupos, como se observa con los grupos 1, 2 y 4.

El primer paso para simular en PARCSIM la resolución de este problema es crear el modelo que represente el algoritmo mostrado en la figura A.10(b).

A.8.2 Creación del modelo

Al iniciar el software, el editor PARCSIM-MB no contiene ningún modelo activo. La barra de herramientas ofrece información relativa al estado actual, como se observa en las tres áreas resaltadas en la figura A.11, donde el área de trabajo únicamente permite el acceso a la configuración del simulador.

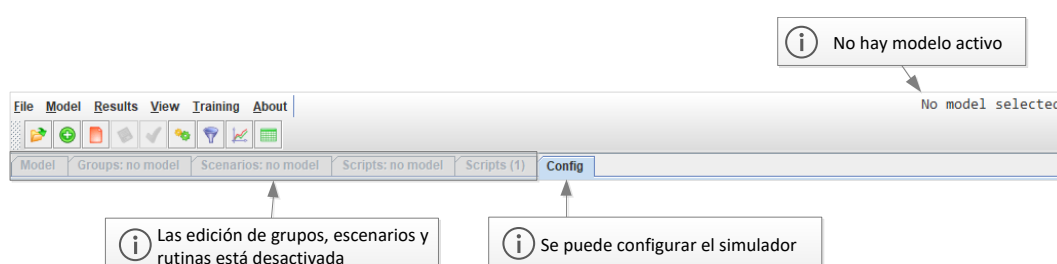


Figura A.11: Barra de estado de la aplicación PARCSIM-MB cuando no hay ningún modelo activo. Solo es posible acceder a la configuración general del simulador.

La figura A.12 muestra las opciones del menú que se encuentran activas en este estado, sin ningún modelo activo.

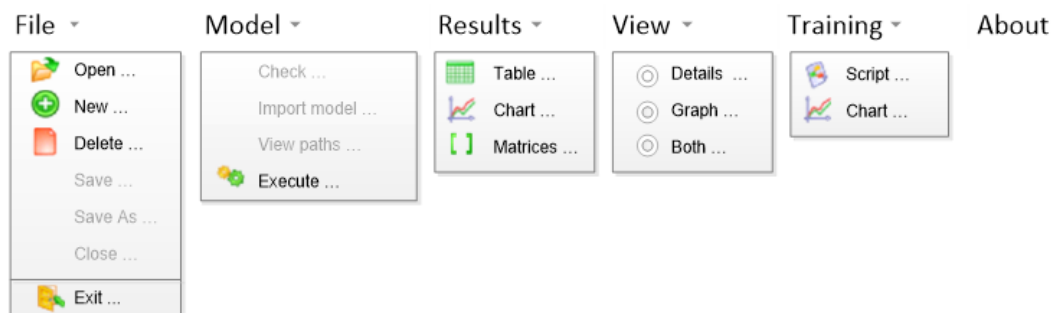


Figura A.12: Barra de menú de la aplicación PARCSIM-MB. La imagen muestra los submenús desplegados para mostrar los enlaces a las funcionalidades del software que se encuentran habilitadas cuando no hay ningún modelo activo.

Para crear un modelo, hacemos click sobre `File`→ `New ...`. El editor nos mostrará un cuadro de diálogo donde se solicita un nombre para el nuevo modelo. Por defecto se propone un identificador compuesto por la letra M (de modelo) seguida de un número entero que será el siguiente valor secuencial en la lista de modelos ya creados. El usuario puede cambiar este nombre y poner otro de su elección. El software comprueba que este identificador no haya sido usado anteriormente y prepara las estructuras vacías para comenzar la edición. Para esta lección elegimos `Tutorial` como nombre para nuestro modelo, y pulsaremos en `Aceptar` como se refleja en la figura A.13.

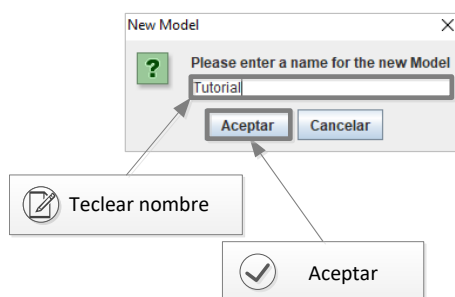


Figura A.13: Edición del nombre de un nuevo modelo durante su creación.

Tras ello, los enlaces a las zonas del área de trabajo para la creación de variables, grupos, escenarios, rutinas y scripts quedan habilitados, y las etiquetas nos indican que el modelo aún está vacío y que por tanto contiene 0 grupos, 0 escenarios y 0 scripts (figura A.14).



Figura A.14: Interfaz gráfico PARCSIM-MB: Información del número de elementos al iniciar la creación de un nuevo modelo.

A.8.3 Creación del primer grupo del modelo

Si observamos el algoritmo representado en la figura A.10(b), el primer grupo que hay que añadir es el etiquetado como *Inicio*, que marca el comienzo del algoritmo. Para crearlo debemos primero hacer visible la pantalla de gestión de grupos haciendo clic en la etiqueta *Groups*, como muestra la figura A.15.

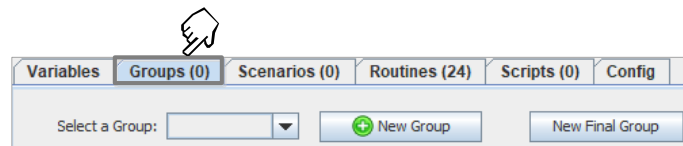


Figura A.15: Interfaz gráfico PARCSIM-MB: Selección de la vista para la creación de un nuevo grupo.

El número entre paréntesis junto al texto *Groups* indica el número de grupos que contiene el modelo en cada momento.



Hacemos clic sobre el botón . El editor nos mostrará un cuadro de diálogo para solicitarnos un nombre para el nuevo grupo. Por defecto el software propone un identificador formado por la letra *G* (de Grupo) seguida de un número entero que corresponde con el siguiente en el orden de creación de los grupos para este modelo. El usuario puede cambiar este nombre y poner otro de su elección (el software comprobará que no haya sido usado previamente). Para este grupo escribimos *Inicio* y pulsamos  como se refleja en la figura A.16.



Figura A.16: Introducción del nombre de un grupo durante su creación.

Una vez añadido el nuevo grupo, el software realiza las modificaciones en el editor mostradas en la figura A.17 para reflejar el nuevo elemento:

- ① Automáticamente la ventana del grafo se actualiza para incluir el nuevo grupo Inicio.
- ② Se actualiza el número de grupos creados (1 en este momento) y dicho valor se muestra junto a la etiqueta “Groups”.

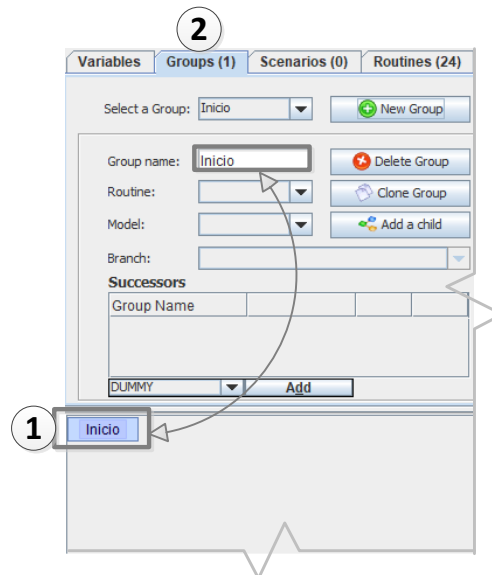



Figura A.17: Interfaz gráfico que refleja el nuevo grupo Inicio.

En caso de tener que cambiar el nombre del grupo, basta con editarlo en la caja de texto etiquetada como “Group Name”. El grafo se actualizará automáticamente al acabar la modificación para mostrar el nuevo nombre.

En nuestro algoritmo el grupo Inicio no realiza ningún cálculo, por lo que no es necesario realizar ninguna acción adicional en este momento. Volveremos a él después para asignarle los grupos que le siguen en secuencia en el algoritmo. Pero antes es necesario seguir creando más grupos.

A.8.4 Creación del segundo grupo

El algoritmo representado en la figura A.10(b) muestra que, tras el grupo Inicio, se deben ejecutar los grupos con las etiquetas 1, 2 y 4. En esta sección vamos a describir el proceso de crear el primero de ellos.

Como en la sección anterior, hacemos clic sobre el botón  y en el cuadro de diálogo emergente cambiamos el nombre propuesto por el de Grupo1, como se muestra en la figura A.18.

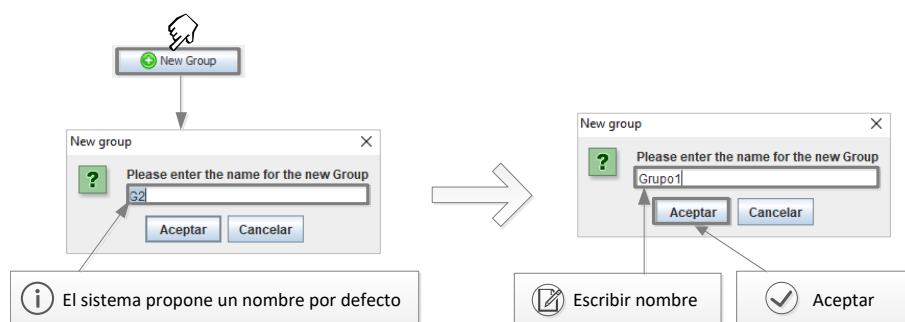


Figura A.18: Proceso de creación del grupo etiquetado como Grupo1.

A.8.5 Creación de una rutina

A diferencia del grupo Inicio, el Grupo1 realiza dos operaciones, una suma de matrices y una resolución de un sistema de ecuaciones. En PARSCIM, la manera de informar al simulador de los cálculos que debe realizar un grupo es asociarle una rutina.

Para crear una rutina debemos tener acceso a la vista de gestión de rutinas haciendo clic sobre la etiqueta “Routines”, como se muestra en la figura A.19. El número entre paréntesis junto al texto Routines nos indica el número de rutinas creadas hasta este momento. Las rutinas no están asociadas a un modelo concreto. Por tanto, éstas pueden ser reutilizadas en la resolución de otros problemas que se introduzcan en el simulador posteriormente.

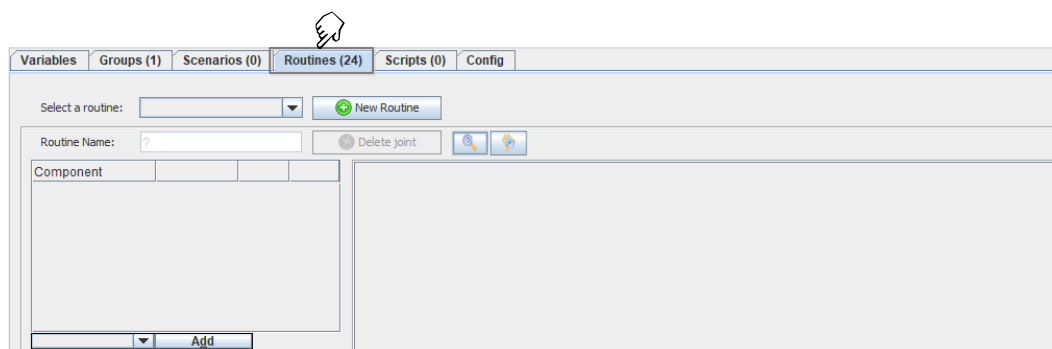



Figura A.19: Activación de la vista de gestión de rutinas.

Haremos clic sobre el botón , como se muestra en la figura A.20. De manera similar a la creación de grupos, el cuadro de diálogo que aparece nos propone un nombre para la nueva rutina, que será inicialmente la letra R (de rutina) seguida de un número entero secuencial. También aquí es posible modificar el identificador propuesto. En este caso le asignaremos el nombre `ADD_SYS`, para recordarnos que esta rutina se encarga de realizar una suma (`ADD`) y una resolución de un sistema de ecuaciones (`SYS`). Si fuera necesario es posible cambiar el nombre de la rutina editándolo en la caja de texto etiquetada como “Routine Name”.

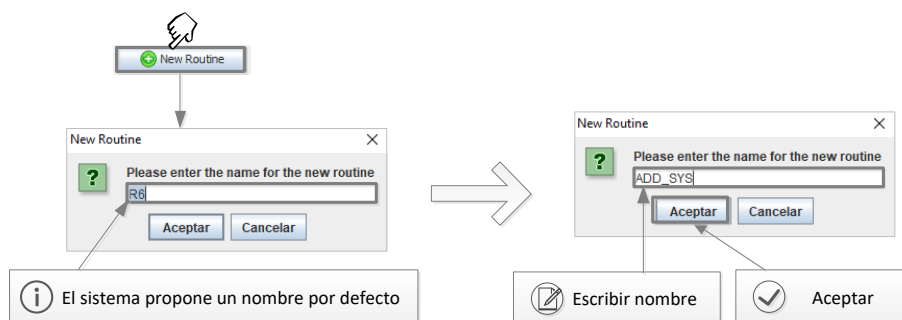


Figura A.20: Proceso de creación de la rutina `ADD_SYS`.

Para incluir las funciones que se ejecutan en esta rutina seguiremos los pasos que se detallan a continuación, y que están reflejados en la figura A.21. En primer lugar añadimos la función suma de matrices, que en el simulador tiene el nombre `MADADD`:

- ① Se hace clic sobre el desplegable, donde se muestran todas las funciones y rutinas disponibles.

- ② Se hace clic sobre la función MADADD, que quedará seleccionada para ser añadida a la rutina.
- ③ Se hace clic sobre **Add** para realizar la asignación de la función a la rutina.
- ④ Se introduce el número de iteraciones/veces que hay que repetir dicha función. El valor por defecto es 1.

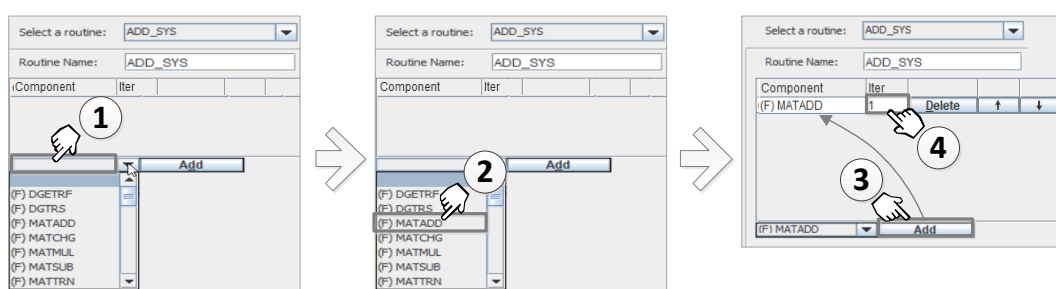


Figura A.21: Proceso de asignación de la función MADADD a la rutina ADD_SYS.

Este proceso puede repetirse tantas veces como sea necesario. Junto al nombre de la función se muestran los botones **Delete**, **↑** y **↓**, que permiten eliminar la función de esta rutina y modificar el orden en el que se ejecuta.

Del mismo modo que hemos añadido la función MADADD, añadimos la función SOLVESYS. Una vez realizado, la rutina contendrá las dos funciones requeridas, como muestra la figura A.22. Se observa a la derecha de la imagen el gráfico de las funciones y la relación entre ellas.

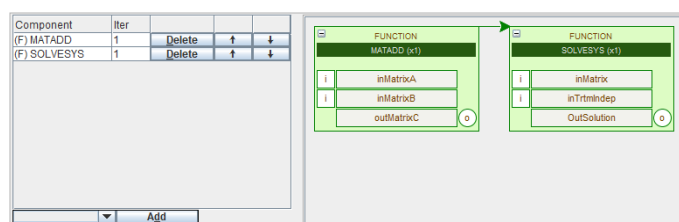


Figura A.22: Interfaz gráfico actualizado para mostrar la rutina ADD_SYS que incluye a las funciones MATADD y SOLVESYS.

A.8.6 Asignar la nueva rutina ADD_SYS al Grupo1

Para asignar la rutina que acabamos de crear al grupo etiquetado como Grupo1 volvemos a la ventana de edición de grupos, y seguimos los pasos mostrados en la figura A.23:

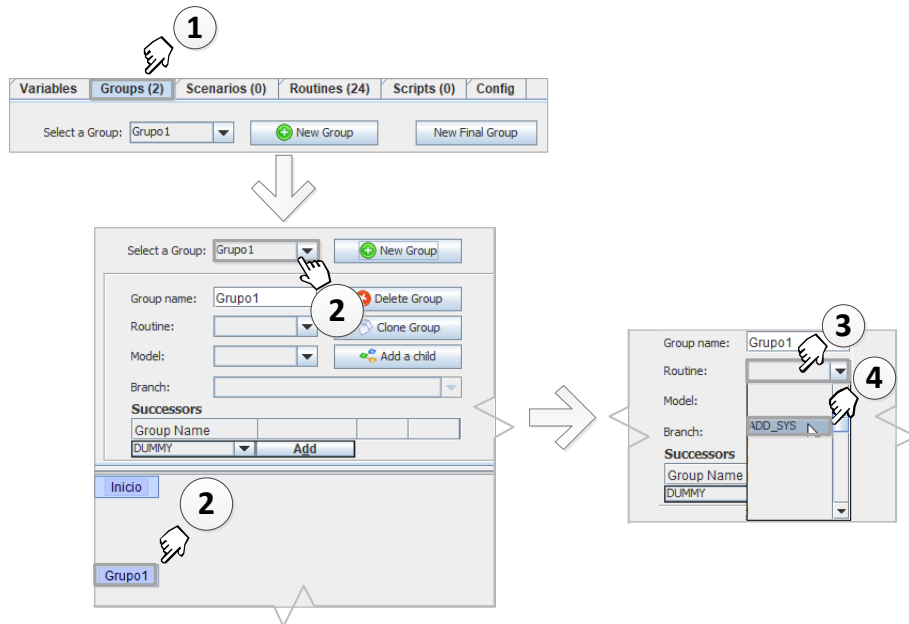


Figura A.23: Asignación de la rutina ADD_SYS al grupo Grupo1.

- ① Hacemos visible la pantalla de gestión de grupos pinchando en la etiqueta “Groups”.
- ② Seleccionamos el grupo al que le vamos asociar una rutina. Para ello podemos hacer clic en el desplegable de grupos, o directamente sobre el nodo representado en la ventana del grafo.
- ③ Se hace clic sobre el desplegable que muestra las rutinas disponibles.
- ④ Se hace clic sobre la rutina que queremos asociar al Grupo1 (en este caso ADD_SYS).

Una vez finalizada la asignación, el grafo se actualiza para mostrar en el grupo la asignación de la rutina, como se puede observar en la figura A.24.

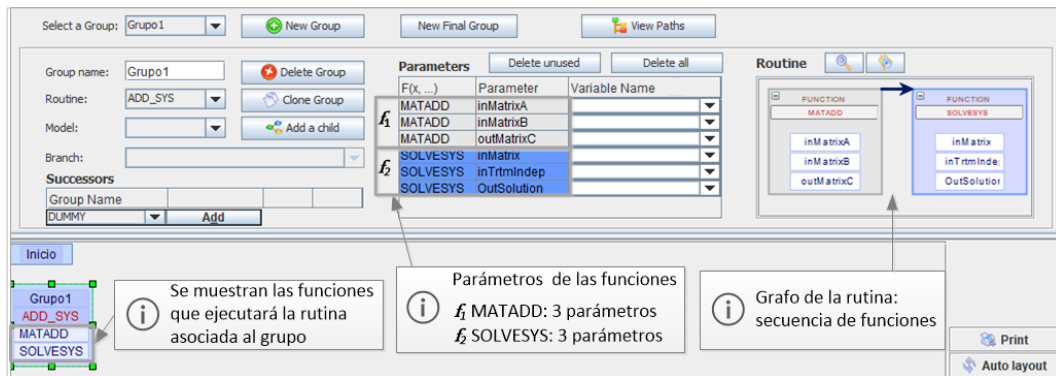


Figura A.24: Interfaz gráfica donde se muestra un grupo con una rutina asignada, detalle de las funciones asociadas y el grafo que muestra la secuencia de cálculos.

A.8.7 Relación entre grupos: asignación de dependencias

Una vez creados los dos primeros grupos ya podemos indicarle al simulador que el Grupo1 se debe ejecutar después del grupo Inicio. Para crear esta dependencia es necesario seguir los pasos que se detallan a continuación, y que se pueden consultar también en la figura A.25:

- ① Nos aseguramos de que tenemos visible la pantalla de gestión de grupos seleccionando la etiqueta “Groups”.
- ② Activamos el grupo al que le vamos a asignar un sucesor, en nuestro caso será Inicio. Para ello hacemos clic sobre el nodo representado en la ventana del grafo.
- ③ Desplegamos el control que muestra los grupos ya creados en este modelo, y hacemos clic sobre el Grupo1 (el que será un sucesor).
- ④ Mediante el botón **Add** realizamos la asignación del sucesor. El grafo del modelo se actualiza para reflejar la dependencia entre ambos grupos mediante una línea dirigida.

En el caso de que un grupo tenga más de un sucesor, se repetiría este proceso tantas veces como sea necesario. Para eliminar una dependencia se puede pulsar sobre el botón **Delete** que aparece junto al identificador del sucesor. La eliminación de un sucesor no borra el grupo, únicamente la relación de dependencia.

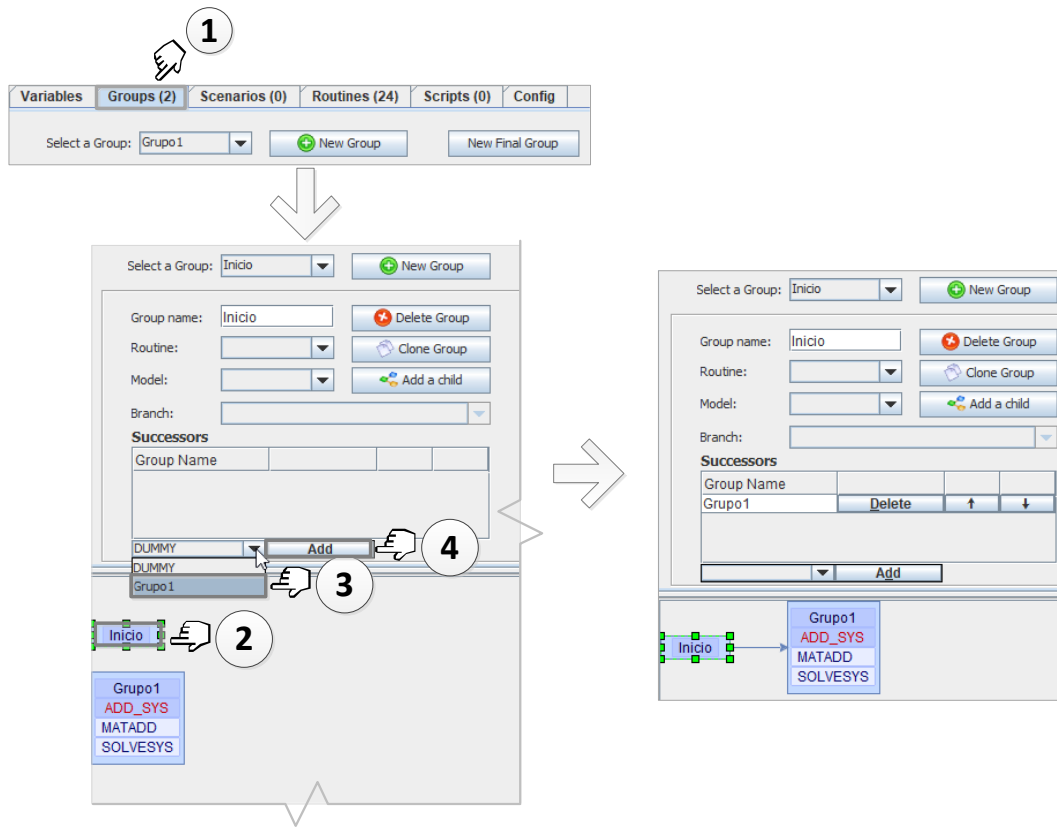



Figura A.25: Creación de la dependencia de Grupo1 respecto a Inicio.

A.8.8 Guardar los cambios

A lo largo del proceso de edición de un modelo es posible guardar el modelo en su configuración actual. Para ello, podemos acceder al menú y seleccionamos **File** → **Save ...**. También se puede hacer clic sobre el icono  de la barra de herramientas. El software mostrará un mensaje informando del resultado de la grabación. En la figura A.26 se nos indica que el modelo se ha guardado, pero con algunos errores.

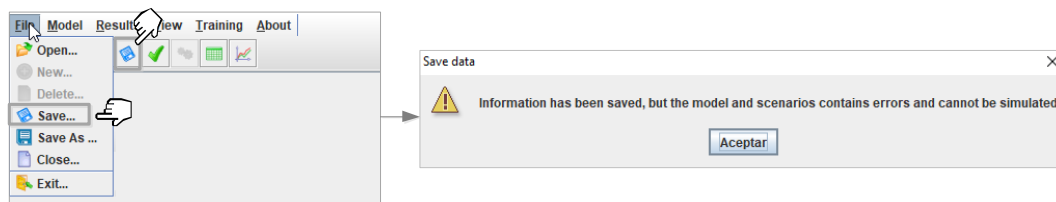




Figura A.26: Proceso de grabación de un modelo y mensaje informativo.

Para conocer el motivo del error podemos hacer clic sobre el icono  de la barra de herramientas o navegar al menú `Model` →  `Check...`. Con ello obtenemos un mensaje del software con una descripción del motivo del error. Como vemos en la figura A.27, PARCSIM informa de que no existe en el modelo un grupo “Final”, un grupo de cierre del algoritmo. Como veremos después, para indicar que un determinado grupo es el último del algoritmo, se le asigna un sucesor ficticio, un grupo denominado DUMMY, que es creado automáticamente por el software.

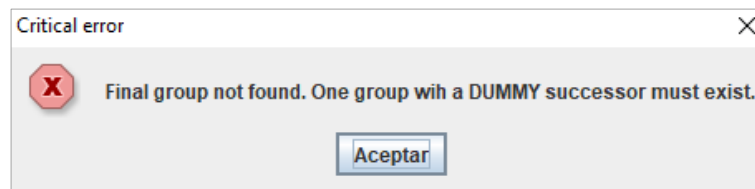




Figura A.27: Mensaje de error obtenido al comprobar la consistencia de un modelo no completado.


A.8.9 Creación del resto de grupos

Volviendo al algoritmo representado en la figura A.10(b), el grupo `Inicio` tiene otros dos sucesores aparte del grupo `Grupo1`, los grupos 2 y 4. Para crear estos nuevos grupos se puede proceder uno a uno, como hemos visto en la sección A.8.4, o se puede utilizar alguna de las dos herramientas que proporciona el simulador para la creación automatizada de grupos:

-  `Clone group`: Permite crear un número n de grupos como copias del grupo seleccionado. El usuario indica el número de copias a crear e indica si quiere que los nuevos grupos hereden los predecesores y/o los sucesores. Los nuevos grupos tendrán la misma rutina que el original y se crearán con nombres compuestos por la letra G seguida de un número entero secuencial. Posteriormente el usuario puede cambiar los nombres y modificar la rutina asociada.

-  **Add a child**: Esta utilidad permite crear un nuevo grupo como hijo del grupo seleccionado, generando automáticamente la dependencia entre ambos. El sistema propone como nombre del nuevo grupo la letra G seguida de un número entero secuencial y espera que el usuario modifique o confirme dicho nombre mediante una ventana de diálogo.

En esta ocasión vamos a clonar el Grupo1, creando dos nuevos grupos que se encuentran al mismo nivel (que tienen el mismo predecesor). Para ello debemos proceder como indica la figura A.28:

- ① Seleccionar el grupo que queremos clonar, por ejemplo haciendo clic sobre el Grupo1 en la ventana del grafo.
- ② Hacer clic sobre el botón  **Clone group**.
- ③ Indicar el número de copias a realizar (en este ejemplo queremos crear dos) y desmarcar la opción de copiar los sucesores, manteniendo marcada la de predecesores (únicamente queremos que dependan del mismo grupo).

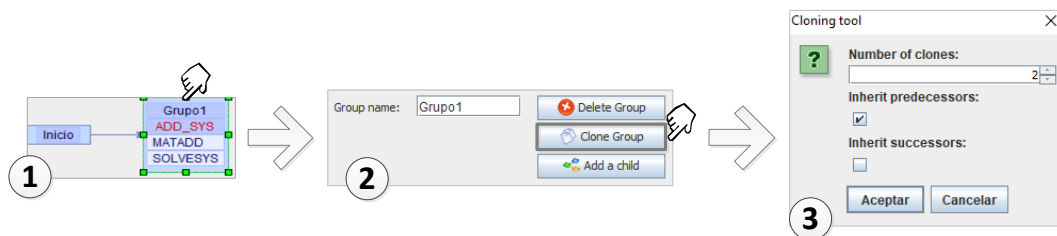


Figura A.28: Creación de grupos mediante el procedimiento de clonado.

Este proceso habrá creado dos nuevos grupos G3 y G4. Una vez creados, podemos modificar lo que sea necesario. Por ejemplo, el recién creado G3 lo renombraremos como Grupo2. Además, como el proceso de clonado crea los grupos con la misma rutina que el grupo original, en el caso del Grupo2 esta deberá ser modificada. Como en este momento aún no hemos creado su rutina, simplemente le quitaremos la incorrecta. Posteriormente crearemos la suya y se la asignaremos. Para realizar estos cambios seguimos los siguientes pasos que muestra la figura A.29:

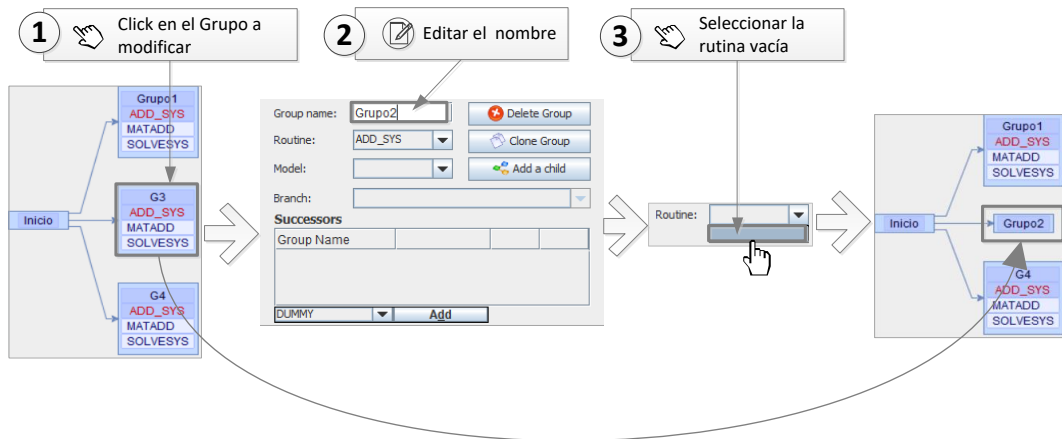


Figura A.29: Cambio del nombre y borrado de la rutina de un grupo.

- ① Seleccionamos el grupo que queremos editar (en este momento el G3) haciendo clic sobre él en la ventana del grafo.
- ② Hacemos clic sobre el nombre del grupo y escribimos el nuevo nombre, Grupo2.
- ③ Seleccionamos el desplegable de las rutinas y elegimos la opción que está vacía. De este modo quitamos la que le fue asignada en el proceso de clonado. Observamos que el grafo refleja los cambios que se acaban de realizar.

Para avanzar con la construcción de nuestro modelo necesitamos crear el grupo sucesor del Grupo1 que aparece con la etiqueta 3 en el algoritmo mostrado en la figura A.10(b). Para ello vamos a usar la herramienta que nos permite añadir un hijo. Su uso se explica a continuación, y también se representa en la figura A.30:

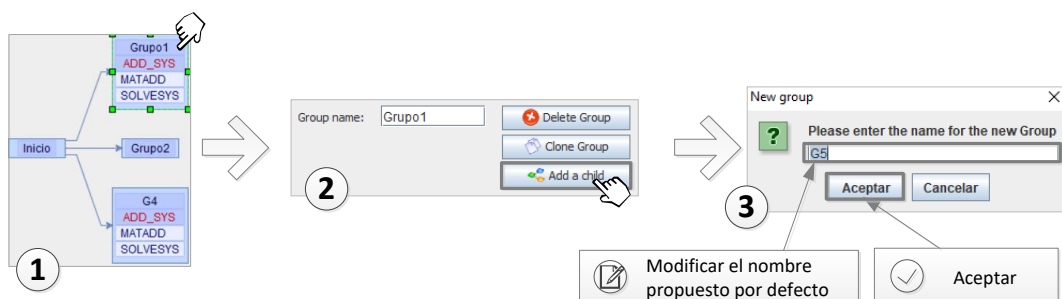
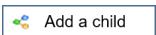


Figura A.30: Herramienta para añadir un hijo al grupo Grupo1.

- ① Se selecciona el grupo al que queremos añadir un sucesor. Una manera sencilla para ello es hacer clic sobre el grupo Grupo1 en la ventana del grafo.
- ② Se hace clic sobre el botón .
- ③ Como en otros procesos de creación, el software ofrece un nombre por defecto. Podemos cambiarlo o confirmarlo. En esta ocasión introduciremos Grupo3.

Una vez creado el Grupo3, como sucesor del Grupo1, ya estamos en disposición de poder añadirlo también como sucesor del Grupo2. Procedemos de un modo similar a como se describió en la sección A.8.7. El G4 se renombra también, en este caso como Grupo4, y se eliminan las rutinas que se han clonado. Como resultado, el grafo del modelo quedará como el que muestra la figura A.31.

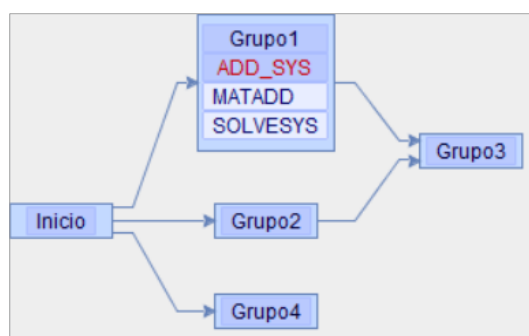


Figura A.31: Grafo que muestra un modelo aún incompleto del algoritmo mostrado en la figura A.10(b).

A.8.10 Construcción del resto del grafo

Con las herramientas incluidas en PARCSIM para crear grupos descritas en las secciones precedentes es posible crear el resto de grupos necesarios para completar el algoritmo descrito en la figura A.10(b) y obtener un grafo como el mostrado en la figura A.32.

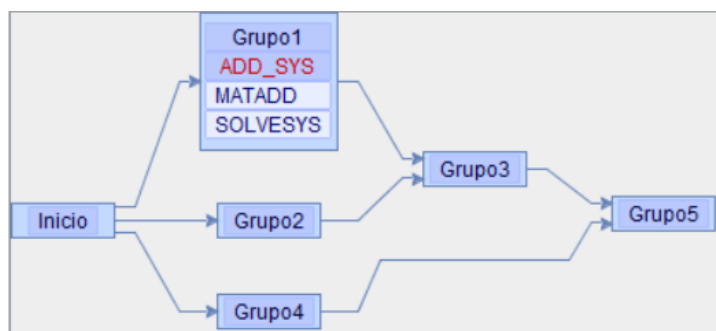


Figura A.32: Grafo con todos los grupos que componen el modelo del ejemplo de la figura A.10(b), a falta de la creación de un grupo final.

Queda pendiente añadir el grupo que completa el algoritmo, que será un grupo final, sin sucesores. El modo de indicar que un grupo no tiene sucesores es asignarle un sucesor ficticio denominado DUMMY, un grupo que el simulador tiene implementado por defecto. Sin embargo, una opción más directa es utilizar la herramienta para crear grupos finales, como se muestra en la figura A.33:

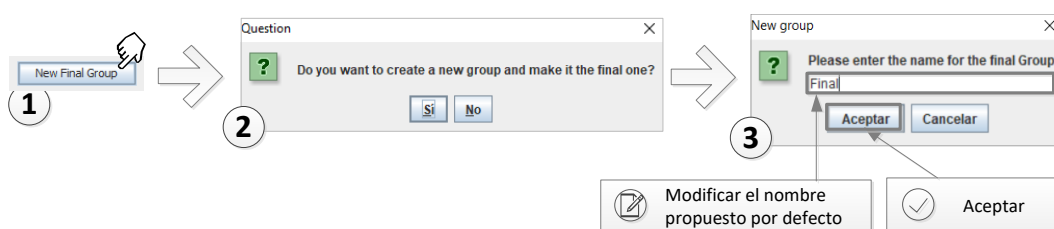


Figura A.33: Procedimiento para crear un grupo final.

- ① Haciendo clic en **New Final Group** se crea un grupo final y se hace que todos los grupos que en ese momento no tienen sucesor añadan al nuevo grupo como tal.
- ② El sistema pide al usuario la confirmación antes de iniciar la creación del nuevo grupo.
- ③ El sistema propone como nombre la letra G seguida de un número entero secuencial, y espera que el usuario modifique o confirme dicho nombre. En nuestro modelo este grupo será **Final**.

La figura A.34 muestra el grafo de nuestro modelo con todos los grupos requeridos en nuestro algoritmo.

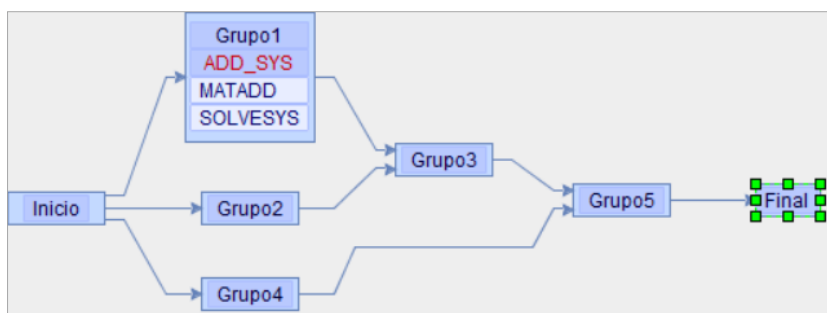


Figura A.34: Grafo completo con todos los grupos que componen el modelo del algoritmo de la figura A.10(b).

A.8.11 Creación del resto de rutinas

Para poder completar la creación del modelo que representa nuestro algoritmo debemos crear las rutinas que se deben asignar a los grupos que están aún incompletos:

- Los grupos Grupo2 y Grupo5, que resuelven un sistemas de ecuaciones: Crearemos una única rutina a la que llamaremos LINSYS y que asignaremos a ambos grupos.
- El grupo Grupo4, que realiza una transposición de una matriz: Crearemos una rutina que llamaremos TRANS.
- El grupo Grupo3, que resuelve una suma de matrices: Crearemos una rutina a la que le daremos el nombre ADD.

Una vez creadas todas las rutinas siguiendo los mismos pasos que en la sección A.8.5, podemos asignarlas a los grupos correspondiente como hicimos en la sección A.8.6. Una vez completado este ejercicio, nuestro modelo tendrá el aspecto mostrado en la figura A.35.

A.8.12 Comprobación del modelo

Al verificar el modelo seleccionando Model → o haciendo clic sobre el icono de la barra de herramientas, observamos que ya no se obtienen errores

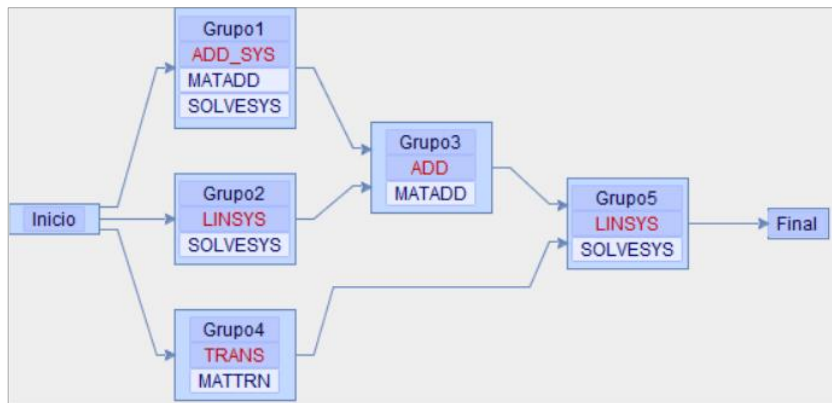


Figura A.35: Grafo con todos los grupos y las rutinas que componen el modelo del ejemplo.

referentes a la estructura del modelo. Esto es debido a que está bien construido, pues todos los grupos tienen un sucesor y existe un grupo que marca el fin del algoritmo. Sin embargo obtenemos nuevos avisos. En el ejemplo de la figura A.36 el software nos recuerda que la rutina `ADD_SYS` asociada al grupo `Grupo1` no tienen definidos todos los valores de sus parámetros (los argumentos de las funciones que lo componen).

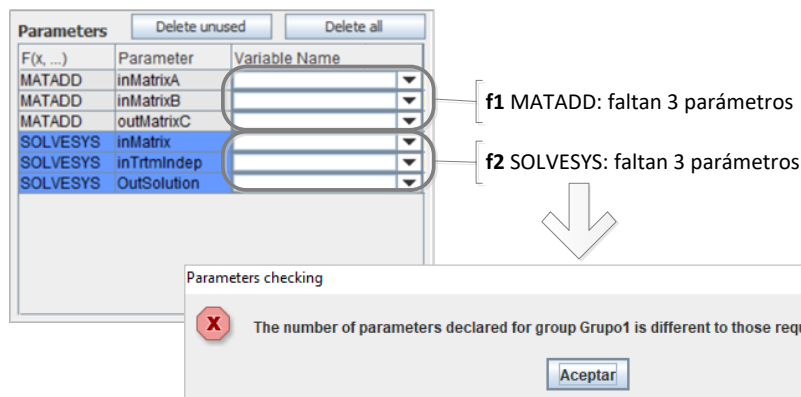


Figura A.36: Error obtenido al comprobar los argumentos de las funciones que se ejecutan en los grupos.

Es sencillo comprobar visualmente la falta de parámetros consultando el grafo integrado en el software. Como se muestra en la figura A.37, al marcar la casilla etiquetada como ☐ Show params. se muestran, además de las funciones que se ejecutan en cada grupo, los argumentos que necesitan dichas funciones. Si observamos el Grupo1, la función `MATADD` requiere de `inMatrixA`, `inMatrixB` y

outMatrixC. Y comprobamos que la columna “Variable” no contiene valores. Es necesario indicar qué variables serán los parámetros de todas las funciones pero, para ello, debemos haber creado previamente dichas variables.

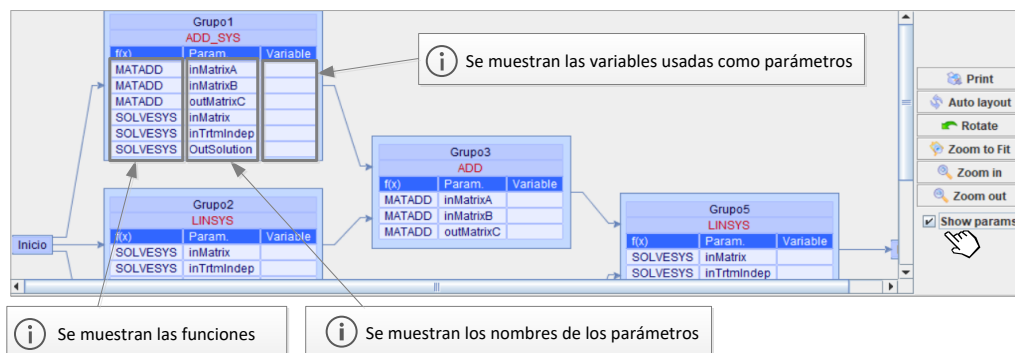


Figura A.37: Visor del grafo ampliado que permite visualizar los parámetros de las funciones.

A.8.13 Crear las variables del modelo

Como vimos en la sección A.8.6, cuando se asigna una rutina a un grupo se están asignando en realidad todas las funciones que componen la rutina. Es necesario conocer en el momento de la simulación los argumentos de todas ellas. Por ejemplo, en la rutina ADD_SYS que contiene dos funciones se necesitarán seis argumentos, tres para la función MATADD y tres para la SOLVESYS.

Según la descripción de nuestro problema numérico, se necesitan cinco matrices cuadradas $\{A1, A2, A3, C, T\}$ y seis matrices columna $\{B1, B2, X1, X2, D, R\}$ para almacenar la información manejada por el modelo (sección A.8.1).

La creación de las variables se realiza a través de un control gráfico de tipo lista, que muestra las variables ya creadas y una línea en blanco adicional para añadir nuevas entradas. El procedimiento a seguir se muestra en la figura A.38:

- ① Hacemos visible la pantalla de gestión de variables seleccionando la etiqueta “Variables”.
- ② Hacemos clic sobre la línea en blanco para activar el modo de edición.

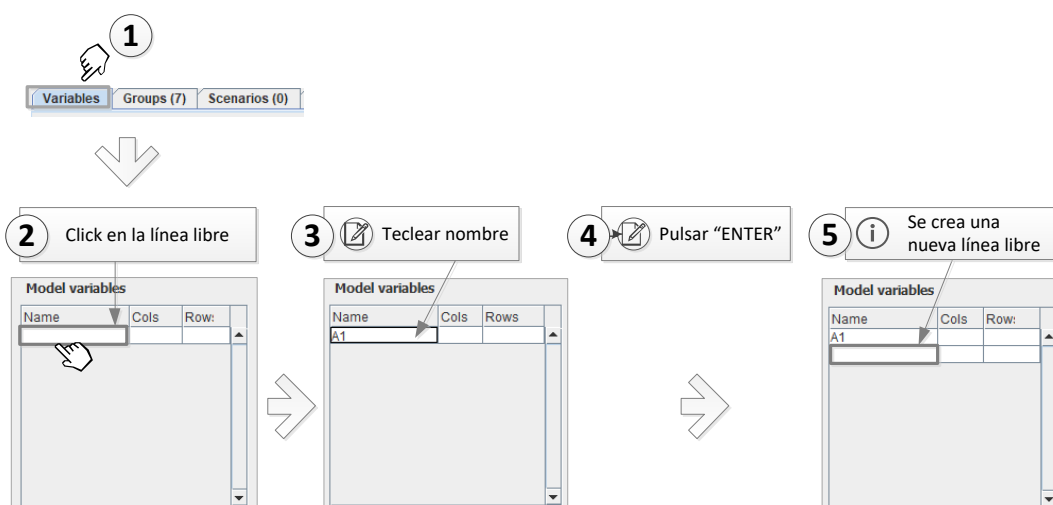


Figura A.38: Procedimiento para la creación de variables.

- ③ Escribimos el nombre de la nueva variable, en este caso comenzaremos con el identificador *A1*.
- ④ Pulsaremos “ENTER” para terminar la edición y añadir la nueva variable a la lista.
- ⑤ Automáticamente se crea una nueva línea en blanco que permite continuar añadiendo variables.

Repetimos este proceso para crear el resto de variables que representan a las matrices $\{A2, A3, C, T\}$ y a los vectores $\{B1, B2, X1, X2, D, R\}$. Observamos cómo el sistema añade las variables a la lista en orden alfabético.

Para editar una variable ya creada, se puede hacer doble-clic sobre su identificador para activar el modo de edición.

A.8.14 Crear las variables de tamaño

Antes de iniciar la simulación de un modelo, el software debe conocer también el tamaño, tipo y valores de las matrices representadas por las variables creadas en la sección anterior. Para simplificar la tarea de asignar un número de filas y columnas a las matrices, usaremos un nuevo conjunto de variables que denominaremos *variables de tamaño*, cuyos valores se especificarán en los escenarios.

Para ilustrar el uso de las variables de tamaño volvemos a considerar el algoritmo mostrado en la figura A.10(b). En el grupo 1 vemos que la matriz C es el resultado de la suma de las matrices cuadradas $A1$ y $B2$. Por tanto, $A1$, $B2$ y C debe tener el mismo número de filas y columnas. Si necesitamos crear n escenarios y x_i representa el valor de las filas y columnas para dichas matrices en un escenario i , entonces habría que realizar 6 asignaciones individuales y, por tanto, $6n$ asignaciones en el total de los escenarios en estudio:

$$\begin{aligned}Filas(A1)_i &= x_i \quad \forall i \\Filas(A2)_i &= x_i \quad \forall i \\Filas(C)_i &= x_i \quad \forall i \\Columnas(A1)_i &= x_i \quad \forall i \\Columnas(A2)_i &= x_i \quad \forall i \\Columnas(C)_i &= x_i \quad \forall i\end{aligned}$$

Sin embargo, podríamos crear una variable t y realizar las siguientes asignaciones:

$$\begin{aligned}Filas(A1) &= t \\Filas(A2) &= t \\Filas(C) &= t \\Columnas(A1) &= t \\Columnas(A2) &= t \\Columnas(C) &= t\end{aligned}$$

De esta manera, bastaría con asignar un valor a t en cada escenario, y automáticamente se estarían modificando el número de las filas y/o columnas de las matrices cuyo valor dependiera de esa variable t . En PARCSIM, t sería una *variable de tamaño*.

Si analizamos los cálculos que se realizan usando las matrices en el modelo del ejemplo, como hemos hecho con $A1$, $B2$ y C , observamos que podemos especificar los tamaños con tan solo dos variables de tamaño. Podemos llamar a estas variables `sizeMat` y `sizeOne`. En la tabla A.1 se muestran las variables del modelo y

las variables de tamaño que sirven para establecer su número de filas y columnas. Se muestran dos posibles escenarios (Escenario-1 y Escenario-2) donde, variando únicamente el valor de una variable de tamaño, se establece el formato de todas las matrices.

Variable del modelo	Filas	Columnas	Escenario-1	Escenario-2
			sizeMat= 20 sizeOne= 1	sizeMat= 100 sizeOne= 1
A1	sizeMat	sizeMat	20×20	100×100
A2	sizeMat	sizeMat	20×20	100×100
A3	sizeMat	sizeMat	20×20	100×100
B1	sizeMat	sizeOne	20×1	100×1
B2	sizeMat	sizeOne	20×1	100×1
C	sizeMat	sizeMat	20×20	100×100
D	sizeMat	sizeOne	20×1	100×1
R	sizeMat	sizeOne	20×1	100×1
T	sizeMat	sizeMat	20×20	100×100
X1	sizeMat	sizeOne	20×1	100×1
X2	sizeMat	sizeOne	20×1	100×1

Tabla A.1: Variables del modelo para el algoritmo de la figura A.10(b). Dos variables de tamaño (sizeMat y sizeOne) son suficientes para establecer las dimensiones de las matrices. Se muestran dos escenarios con diferentes valores de sizeMat.

La creación de las variables de tamaño se realiza a través de un control gráfico de tipo lista que muestra las variables de tamaño ya creadas y una línea en blanco adicional que permite añadir nuevas entradas. El procedimiento a seguir se muestra en la figura A.39:

- ① Hacemos visible la pantalla de gestión de variables haciendo clic en la etiqueta “Variables”.
- ② Seleccionamos la línea en blanco para iniciar el modo de edición.
- ③ Escribimos el nombre de la nueva variable, en este caso comenzaremos con la identificada como sizeMat.
- ④ Pulsamos “ENTER” para terminar la edición y añadir esta variable a la lista.

- ⑤ Automáticamente se crea una nueva línea en blanco que nos permite continuar añadiendo variables.

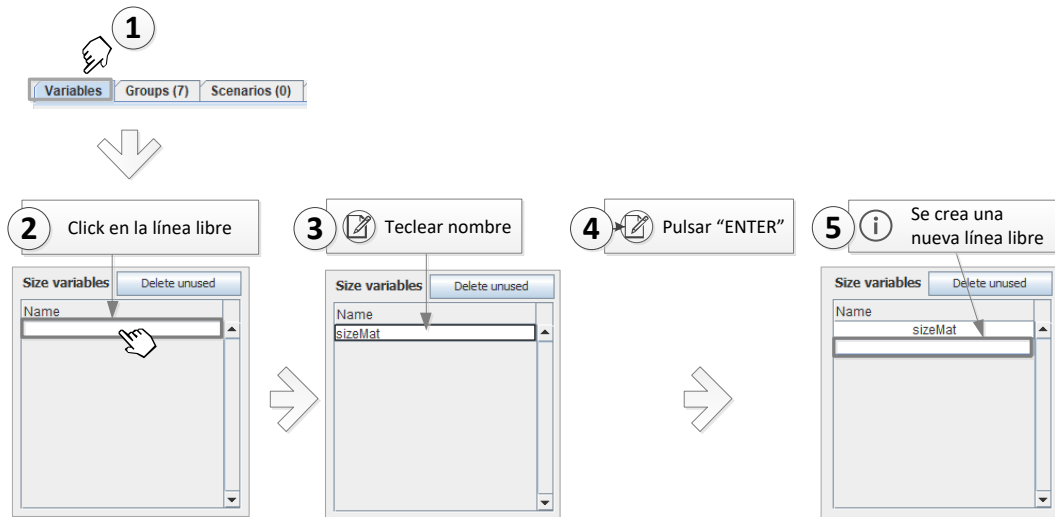


Figura A.39: Proceso de creación de variable de tamaño.

Repetimos este proceso para crear la variable `sizeOne`. Para editar una variable ya creada, se puede hacer doble-click sobre su identificador para activar el modo de edición.

A.8.15 Asociar variables de tamaño a variables de modelo

En este momento ya hemos definido las variables de nuestro modelo, y también las variables que nos van a ayudar a definir los tamaños de las matrices para cada escenario, como muestra la figura A.40.

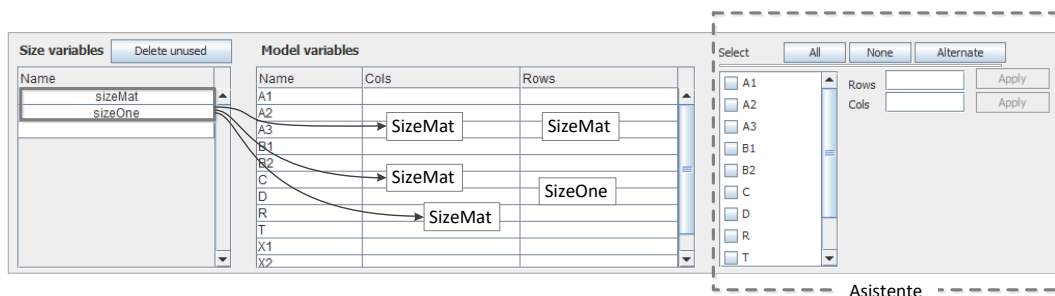


Figura A.40: Vista de las variables de tamaño y de modelo creadas.

La siguiente tarea consiste en utilizar el asistente para indicar, para cada variable de modelo, qué variable de tamaño especifica su número de filas y de columnas. El procedimiento a seguir se muestra en la figura A.41:

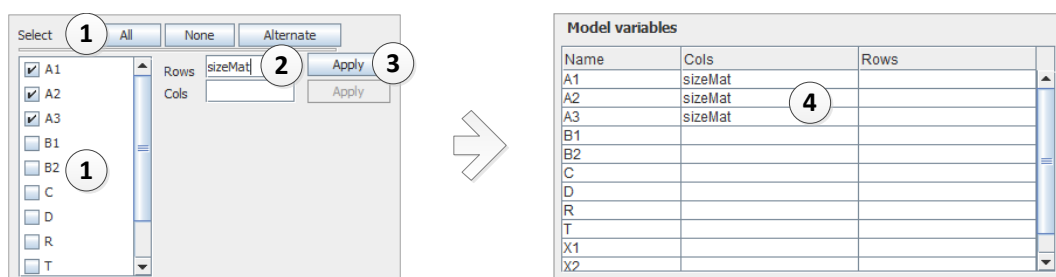


Figura A.41: Asistente para la asignación de variables de tamaño a variables de modelo.

- ① Seleccionamos la o las variables de modelo que vamos a modificar. Para ello se pueden marcar las casillas ☐ junto al nombre de las variables, o usar cualquiera de los siguientes botones:
 - **All**: Para seleccionar todas las variables del modelo.
 - **None**: Para deseleccionar todas las variables del modelo.
 - **Alternate**: Para invertir el estado de selección de las variables.
- ② Escribimos el nombre de la variable de tamaño en el campo de texto “Rows” o “Cols”, según queramos definir el número de filas o de columnas respectivamente. Cuando el texto tecleado corresponda con una variable de tamaño existente, el botón **Apply** se habilitará. En este caso tecleamos la variable de tamaño `sizeMat`.
- ③ Hacemos clic sobre **Apply**.
- ④ La asignación queda reflejada en el editor.

Si repetimos el proceso anterior para el resto de las variables de modelo obtenemos el resultado mostrado en la figura A.42.

Model variables		
Name	Cols	Rows
A1	sizeMat	sizeMat
A2	sizeMat	sizeMat
A3	sizeMat	sizeMat
B1	sizeMat	sizeOne
B2	sizeMat	sizeOne
C	sizeMat	sizeMat
D	sizeMat	sizeOne
R	sizeMat	sizeOne
T	sizeMat	sizeMat
X1	sizeMat	sizeOne
X2	sizeMat	sizeOne

Figura A.42: Lista completa de las variables de modelo con la especificación de su tamaño.

A.8.16 Asignación de variables de modelo a argumentos

Una vez creadas todas nuestras variables, ya podemos asignarlas como argumentos a las funciones que se van a ejecutar en nuestro modelo. La asignación en PARCSIM-MB se realiza mediante controles desplegables que muestran la lista de variables de modelo disponibles (figura A.43).

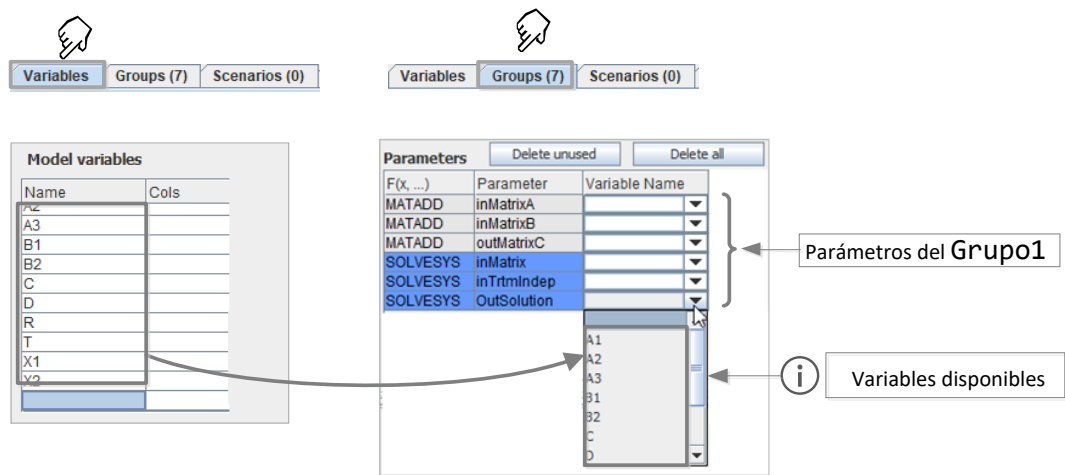


Figura A.43: Argumentos de las funciones que se ejecutan en un grupo y sus controles desplegables con las variables del modelo disponibles.

Se observa que las variables creadas en la vista de “Variables” se hacen visibles en los seis desplegables que corresponden a los argumentos de las funciones que se ejecutan en el Grupo1. El proceso de asignación del primer argumento (la variable A1) al Grupo1 se representa en la figura A.44:

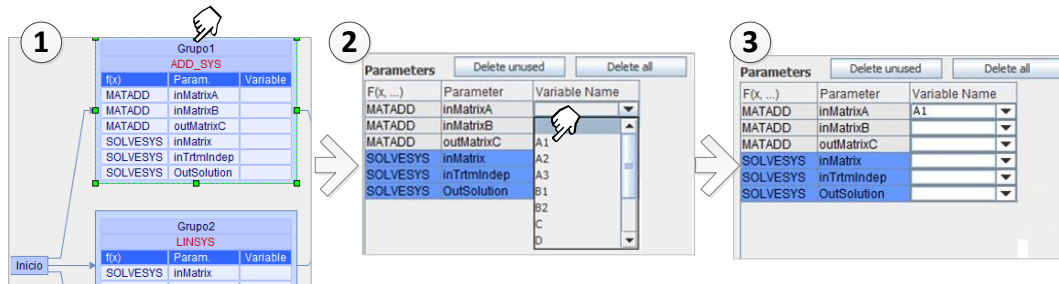


Figura A.44: Proceso de asignación del primer argumento de la función MATADD (la variable A1) al Grupo1.

- ① Se selecciona el grupo para cuyas funciones queremos especificar los parámetros. Una manera sencilla para ello es hacer clic sobre el grupo Grupo1 en la ventana del grafo.
- ② Se hace clic sobre el desplegable del parámetro correspondiente y se selecciona la variable A1.
- ③ El cuadro de texto se actualiza para mostrar la variable seleccionada.

Repetiendo el proceso hasta completar todos los parámetros del grupo Grupo1, con la casilla etiquetada como ☐ Show params. marcada, obtenemos el resultado que se muestra en la figura A.45.

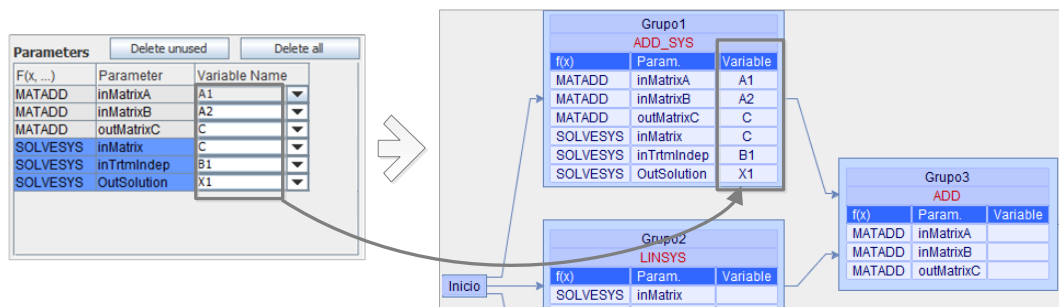




Figura A.45: Grupo1 con todos sus parámetros y variables asignadas.

Si verificamos de nuevo el modelo seleccionando `Model` →  `Check...`, o haciendo clic sobre el icono  de la barra de herramientas, observamos que los errores relativos a la falta de parámetros en el Grupo1 se han subsanado, y el nuevo mensaje hace referencia al Grupo2, como observamos en la figura A.46.

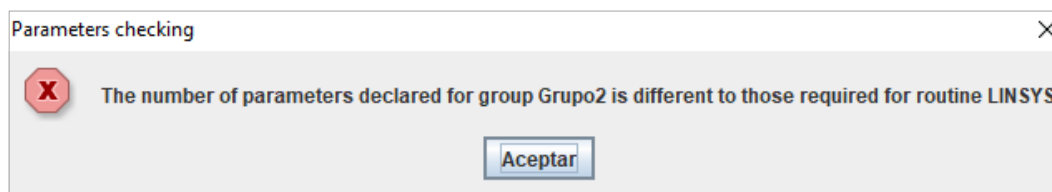


Figura A.46: Mensaje de error por falta de parámetros en la rutina que ejecuta el Grupo2.

Una vez capturados los parámetros en todos los grupos que forman el modelo, cuando verificamos el modelo obtendremos un mensaje de ausencia de errores, como el que se muestra la figura A.47.

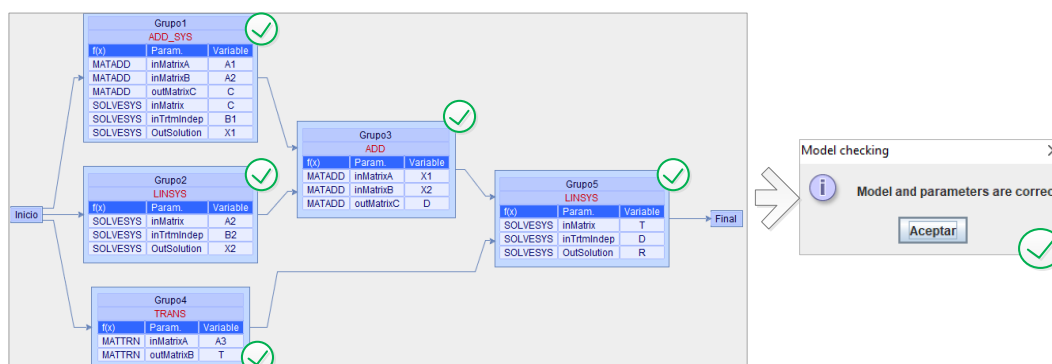


Figura A.47: Modelo completo donde todas las funciones tienen asignados sus argumentos.

A.8.17 Rutas

Cuando un modelo está completo, PARCSIM es capaz de analizar su grafo y calcular las diferentes alternativas de ordenación de los cálculos, respetando las relaciones de dependencia entre los grupos.

Como se describió en la sección A.1.8, cada una de las ordenaciones posibles se denomina ruta. En la figura A.48 se muestran las rutas identificadas en el modelo del ejemplo. Se trata de un visor gráfico accesible seleccionando **Model** →

 **View paths...** o haciendo clic sobre el botón  **View paths** en la vista de grupos.

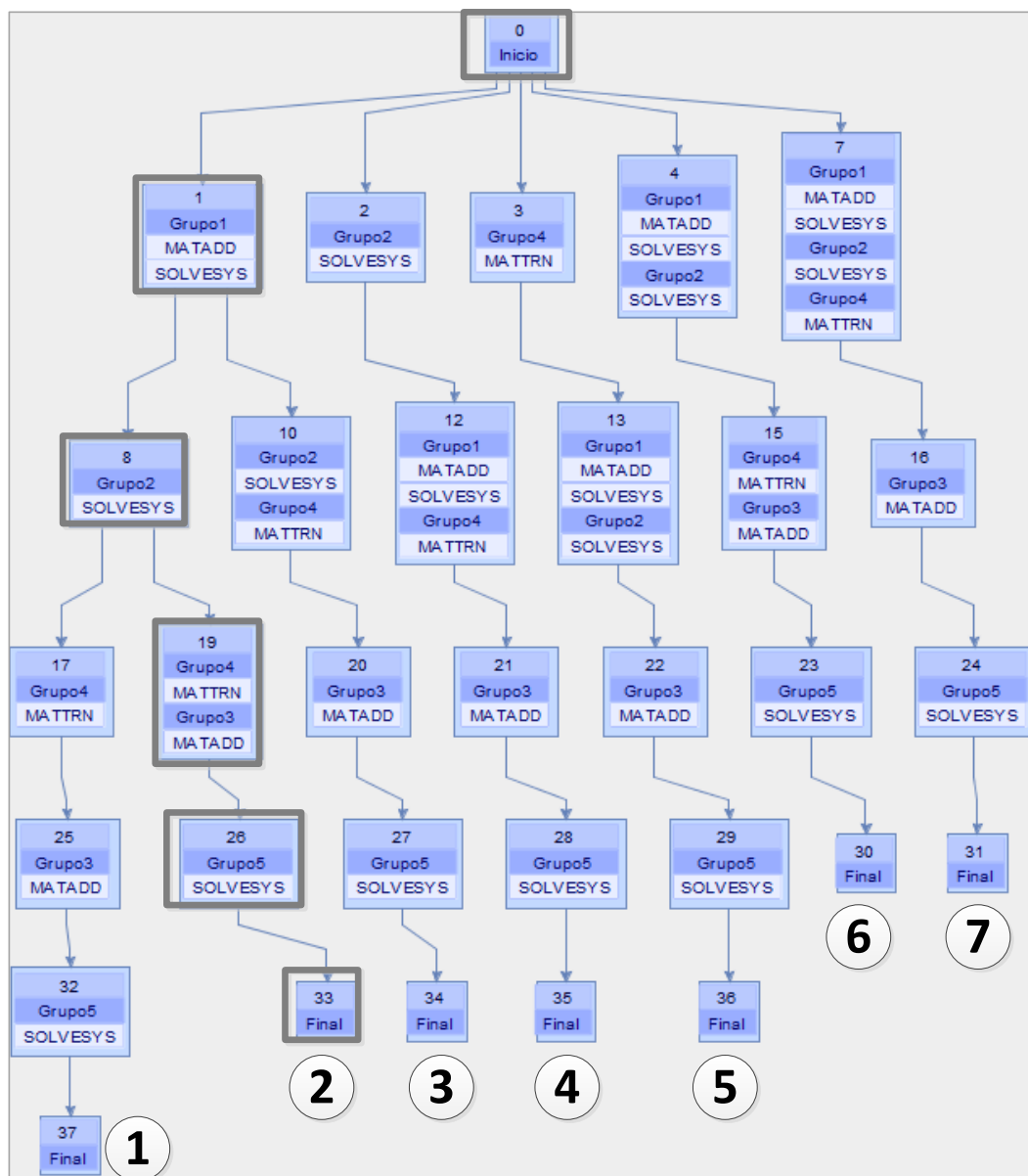
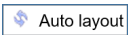








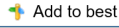


Figura A.48: Rutas de resolución para el modelo de la figura A.10(b). Se ha resaltado la ruta 2, que incluye el cálculo simultáneo de los grupos Grupo3 y Grupo4.

Se pueden identificar siete secuencias de ejecución (rutas) válidas, etiquetas del ① al ⑦ en la imagen. Todas las rutas proponen en algún momento la ejecución simultánea de más de un grupo, excepto la ①. Resaltamos en la figura los cálculos que se siguen en la ruta ②, donde podemos encontrar una etapa en la que los grupos Grupo3 y Grupo4 se resuelven en paralelo. Si el hardware donde se realiza la simulación del modelo dispone de más de una unidad de cómputo, las rutas que permitan resolución simultánea de grupos podrían requerir menores tiempos de ejecución que otras con ejecuciones secuenciales. Uno de los objetivos de la simulación es identificar la mejor de todas las alternativas, en función de los parámetros algorítmicos fijados.

El visor incluye una barra de botones con funcionalidades que modifican y permiten ajustar el aspecto del grafo y seleccionar una o varias rutas para las próximas simulaciones:

-  : El software distribuye automáticamente los nodos del árbol en la pantalla.
-  : Ajusta el tamaño del árbol de manera que todos los nodos sean visibles.
-  : Desplaza el grafo hasta que el nodo inicial queda centrado en pantalla.
-  : Permite alternar entre las orientaciones horizontal y vertical.
-  : Aumenta la separación entre los diferentes niveles del árbol.
-  : Reduce la separación entre los diferentes niveles del árbol.
-  : Aumenta el tamaño de los nodos y el texto con la descripción de los grupos y rutinas que los componen.
-  : Reduce el tamaño de los nodos y el texto con la descripción de los grupos y rutinas que los componen.
-  : Resalta frente al resto la ruta o rutas del árbol que se hayan marcado previamente como las más eficientes.

-  : Si hay una ruta seleccionada en el grafo (se ha hecho clic sobre cualquier de los nodos que la integran) esta opción añade dicha ruta al conjunto de las seleccionadas. En caso contrario, se eliminará cualquier selección anterior. Durante el proceso de simulación se usan las rutas seleccionadas para realizar ejecuciones de todas ellas. Si no hay ninguna ruta preseleccionada, el software estimará la mejor alternativa en base a la información de entrenamiento de la que se disponga en la base de datos.

Haciendo clic sobre cualquier nodo en el gráfico, se resaltará la rama completa de la que ese nodo forme parte.

A.8.18 Creación de un escenario

En este momento ya se ha definido completamente el modelo y PARCSIM tiene la información necesaria acerca de los cálculos que hay que llevar a cabo para resolver un determinado problema numérico. También se han definido el conjunto de variables que usará nuestro algoritmo que, como vimos en la sección A.1.5, tendrán un formato matricial. Sin embargo aún no se ha dicho nada del formato de dichas variables.

Como vimos en 5.2.6, los escenarios definen la naturaleza del problema a resolver, y serán el conjunto de valores que pueden tomar las tres características que definen las matrices: dimensiones, tipo y factor de dispersión. Para poder crear escenarios debemos hacer visible la vista correspondiente haciendo clic en la etiqueta “Escenarios”, como se muestra en la figura A.49.

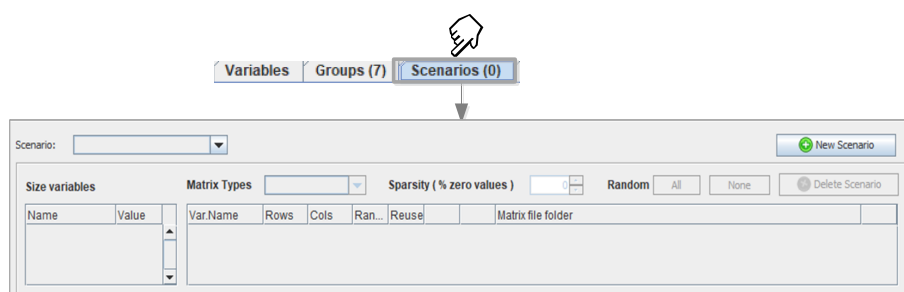




Figura A.49: Activación de la vista de gestión de escenarios.

El número entre paréntesis en la etiqueta indica los escenarios que hay definidos para este modelo en este momento.

Comenzamos creando un escenario que denominaremos Escenario-1. Para ello hacemos clic sobre el botón . El editor nos mostrará un cuadro de diálogo como el de la figura A.50 para solicitarnos un identificador. Tecleamos Escenario-1 y pulsamos .

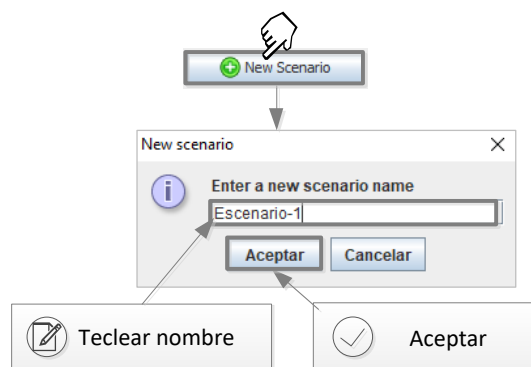


Figura A.50: Solicitud de un identificador para un nuevo escenario durante su fase de creación.

Una vez creado el nuevo escenario, el editor se actualiza y muestra la vista de edición donde se pueden capturar los valores que lo componen, como se puede observar en la figura A.51:

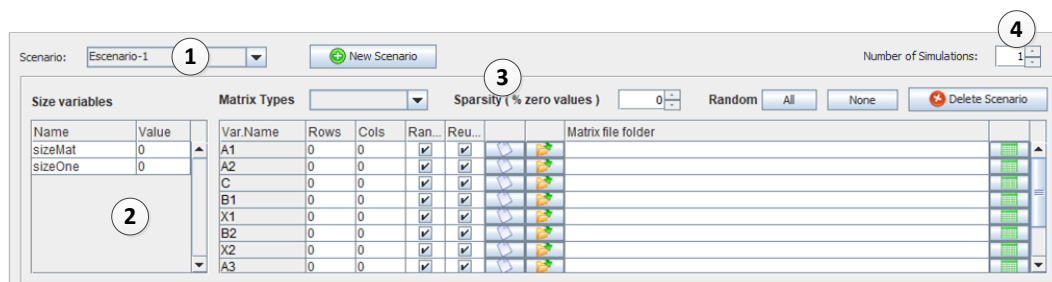


Figura A.51: Vista de mantenimiento de un nuevo escenario. Se observan las dos variables de tamaño que definimos para especificar el tamaño de las variables del modelo.

- ① Se añade el identificador del nuevo escenario al desplegable que permitirá posteriormente seleccionar un escenario a editar.
- ② Se observan las dos variables de tamaño que añadimos en la sección A.8.14 de este tutorial, `sizeMat` y `sizeOne`.
- ③ Inicialmente, el tipo de matriz y dispersión están pendientes de asignar.
- ④ Se especifica el número de veces que se repetirá la simulación del modelo.

La figura A.52 muestra el grafo cuando marcamos la casilla ☐ Show params .

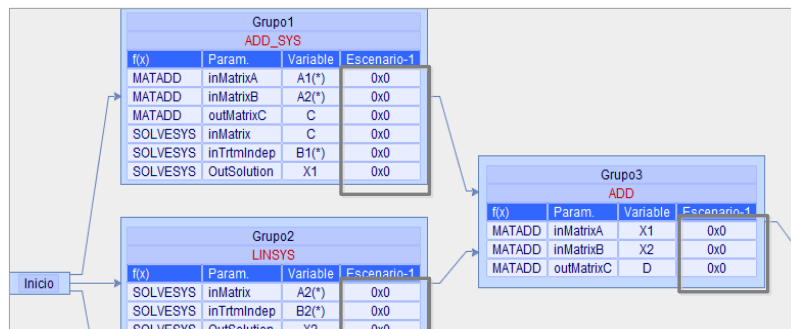


Figura A.52: Vista del grafo del modelo que muestra el escenario recién creado.

Como el escenario aún no tiene valores asignados, se muestran tamaños de matrices 0×0 . Para especificar los valores de un escenario se procede como se describe en la figura A.53:

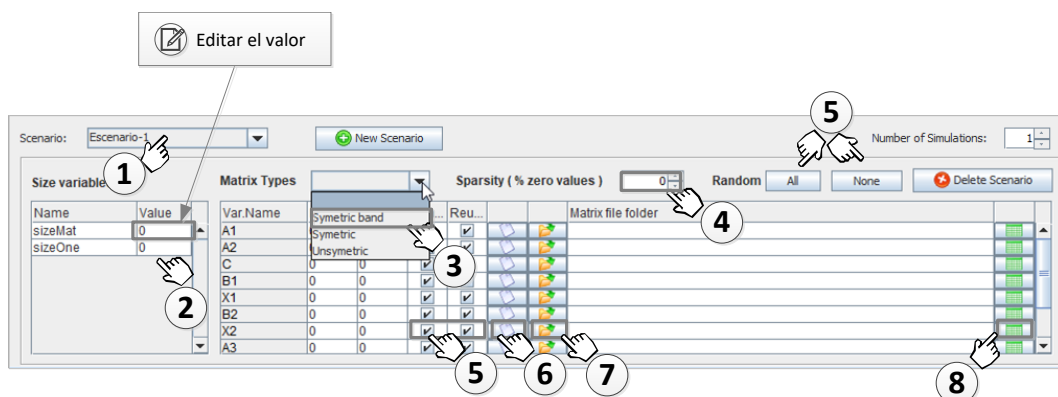


Figura A.53: Edición de los valores de un escenario.



- ① Se selecciona el escenario que vamos a editar. Para ello se hace clic en el desplegable que muestra los identificadores de los escenarios creados para el modelo activo.
- ② Se introduce el valor que toman las variables de tamaño en este escenario. Esta acción modifica inmediatamente los valores de filas y columnas de las variables del modelo.
- ③ Se especifica el tipo de matriz. Para ello se hace clic sobre el desplegable y se selecciona el tipo adecuado. En la versión actual del simulador este valor se aplica a todas las variables de este escenario.
- ④ Se selecciona el factor de dispersión (es decir, el porcentaje de valores cero sobre el número total de elementos de la matriz). Se usan los botones correspondientes para incrementar y reducir los valores. En la versión actual del simulador este valor se aplica a todas las variables de este escenario.
- ⑤ Se especifica si las matrices se rellenarán con valores aleatorios al iniciar la simulación, como se explicó en la sección 5.2.6. Para ello se puede marcar individualmente la casilla ☐ correspondiente a una o varias variables del modelo, o usar los botones **All** y **None**.
- ⑥ Si la casilla ⑤ no está seleccionada, se puede elegir la captura manual de los valores iniciales de una matriz. Esta opción abrirá un editor en forma de tabla.
- ⑦ Alternativamente a la captura manual de los valores de una matriz, éstos se pueden recuperar de un fichero binario (*unformatted*) de FORTRAN que debe haberse generado en ejecuciones anteriores. Esta opción muestra una ventana del sistema operativo para selección de ficheros.
- ⑧ En el caso de que se haya seleccionado un fichero en ⑦, esta opción permite mostrar su contenido en una nueva ventana que muestra los datos de la matriz.

A.8.19 Validación de escenarios

Las funciones implementadas en el simulador imponen restricciones al formato que deben adoptar sus argumentos. Tomemos como ejemplo la función `SOLVESYS` encargada de la resolución de un sistema de ecuaciones. Esta función requiere para su ejecución:

- `inMatrix`: La matriz de coeficientes del sistema a resolver.
- `inTrtmindep`: Una matriz columna que representa el término independiente.
- `outSolution`: Una matriz columna que recibe la solución del sistema de ecuaciones.

Si partimos de un sistema cuadrado, donde `inMatrix` tiene dimensión $n \times n$, entonces el término independiente `inTrtmindep` deberá tener dimensiones $n \times 1$, lo mismo que el parámetro que contiene la solución del sistema, `outSolution`.

Como se describió en la sección 5.2.1, PARCSIM almacena reglas que especifican relaciones entre los tamaños de los parámetros y, por tanto, puede verificar automáticamente los escenarios introducidos por el usuario para comprobar la coherencia en los mismos. Si consideramos un escenario incorrecto como el mostrado en la figura A.54, al verificar el modelo seleccionando `Model` →  `Check...` o haciendo clic sobre el icono  de la barra de herramientas, obtenemos un mensaje donde se nos describe el primer error encontrado. En este caso se trata de que el término independiente debe tener dimensiones 20×1 .

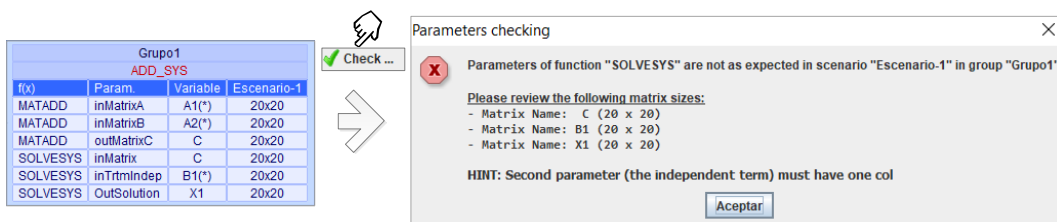


Figura A.54: Escenario incorrecto debido a los tamaños de las matrices.

Si modificamos el escenario para que las variables que representen vectores tengan un formato 20×1 entonces el error desaparece, como podemos comprobar en la figura A.55.

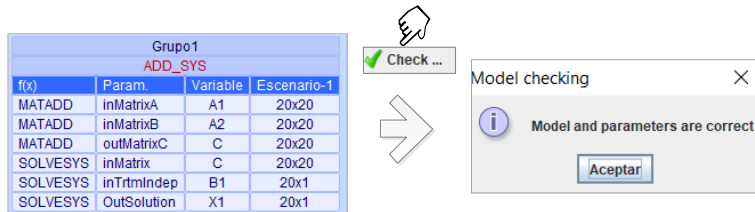


Figura A.55: Mensaje de escenario correcto.

A.8.20 Creación de un script

En la sección 5.2.7 se definieron los scripts como conjuntos de los parámetros algorítmicos utilizados por el simulador: el número de threads, la cantidad de GPUs y la librería de cómputo a utilizar. Para poder gestionar los scripts se debe activar la vista correspondiente haciendo clic en la etiqueta “Scripts”, como se muestra en la figura A.56.

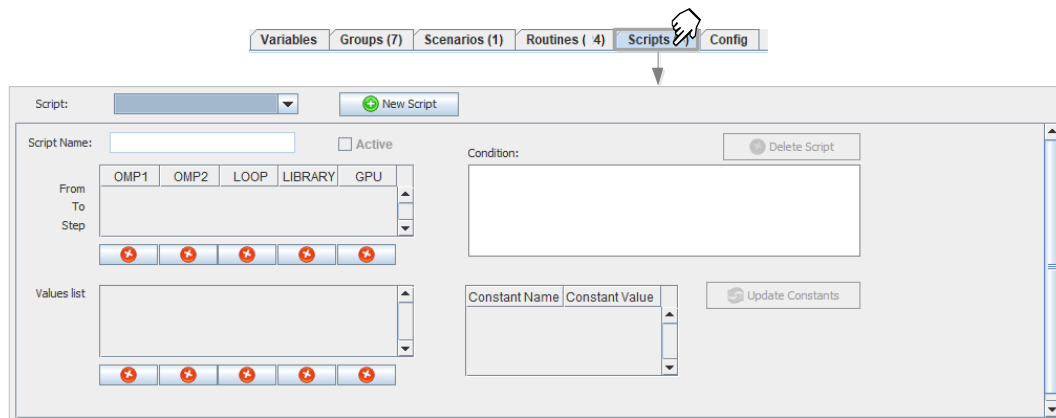


Figura A.56: Activación de la vista de gestión de scripts.

Para crear un script hacemos clic sobre el botón . El editor nos mostrará un cuadro de diálogo solicitando un nombre para el nuevo script. Por defecto el software propone un identificador formado por la letra S (de Script) seguida de un número entero, que corresponde con el siguiente en secuencia en el orden de creación de los scripts. El usuario puede cambiar este nombre y poner

otro de su elección (el software comprobará que no haya sido usado anteriormente en el mismo modelo). En este ejemplo escribimos `ScriptTutorial1` y pulsamos **Aceptar**, como vemos en la figura A.57.

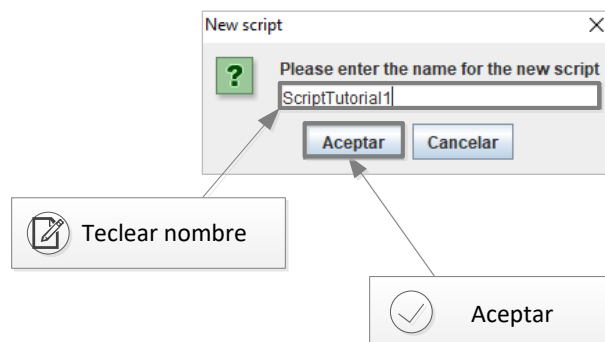


Figura A.57: Captura del nombre de un nuevo script durante su proceso de creación.

Una vez creado el script, PARCSIM-MB modifica el interface y prepara los campos que permiten su edición, como se puede observar en la figura A.58:

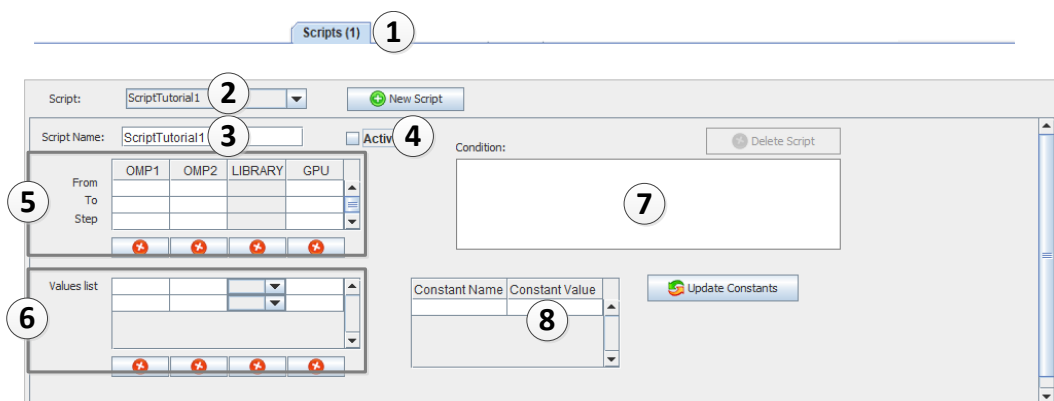


Figura A.58: Editor preparado para introducir datos de un nuevo script.

- ① Se actualiza el número de scripts creados (1 en este momento) y se muestra en la etiqueta “Scripts”.
- ② El desplegable de selección de scripts incorpora el elemento recién creado.

- ③ En caso de necesitar cambiar el nombre del script, basta con editarlo en la caja de texto etiquetada como “Script Name”.
- ④ La casilla de verificación ☐ Active permite indicar al simulador que utilice este script para las ejecuciones.
- ⑤ Se editan los valores de cada parámetro algorítmico. En esta ventana se permite indicar rangos de valores. Se solicita un valor inicial, un final y un intervalo.
- ⑥ En esta ventana se pueden introducir valores individuales de los parámetros.
- ⑦ Se puedan crear fórmulas. La fórmula expresará una condición entre los valores de los parámetros que, de cumplirse, hará que esa combinación sea usada por el simulador.
- ⑧ Es posible crear constantes, mediante pares de nombre y valor. Las constantes creadas se pueden usar en las fórmulas descritas en el punto anterior.

Ahora asignaremos algunos valores a los parámetros OMP1 y OMP2. El primero se refiere al número de threads OpenMP y el segundo al de threads reservados a las librerías con paralelismo implícito. La figura A.59 muestra algunos valores introducidos en el editor. El primero representa al rango de valores $\{1, \dots, 4\}$. El segundo son los valores individuales 1 y 3. Se observa que la captura de rangos y de valores individuales son excluyentes, por lo que cuando se comienza a introducir valores en una de las opciones la otra queda deshabilitada.

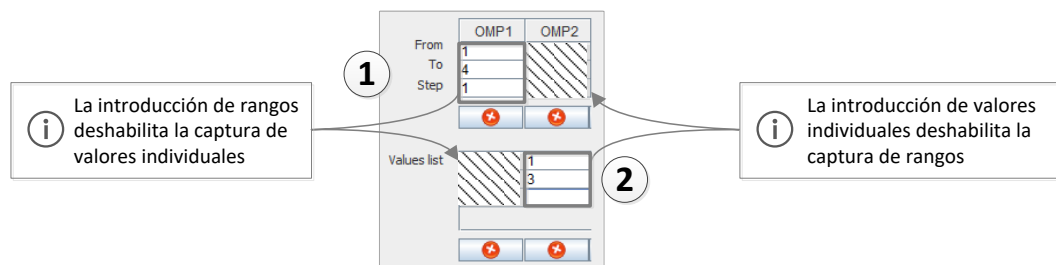


Figura A.59: Captura de rangos de valores e individuales de parámetros algorítmicos durante la edición de un script.

En la versión actual del simulador no se consideran niveles adicionales de paralelismo que admiten algunas librerías, como es el caso de LAPACK.

Cuando el simulador trate nuestro modelo, realizará tantas ejecuciones como combinaciones de parámetros algorítmicos se puedan generar en base a los scripts. Si consideramos únicamente los parámetros OMP1 y OMP2 que acabamos de introducir, obtenemos los ocho pares de valores que se muestran en la tabla A.2.

Ejecución	Parámetro	
	OMP 1	OMP 2
1	1	1
2	1	3
3	2	1
4	2	3
5	3	1
6	3	3
7	4	1
8	4	3

Tabla A.2: Combinaciones de valores obtenidos por el simulador a partir de la información introducida en el editor de PARCSIM en los parámetros OMP1 $\{1, \dots, 4\}$ y OMP2 $\{1, 3\}$, threads OpenMP y threads asignados al paralelismo de las librerías, respectivamente.

PARCSIM permite acotar las combinaciones para que cumplan una determinada condición. Por ejemplo, en el caso de disponer de un máximo de 4 cores, los pares de threads $\text{OMP1} \times \text{OMP2}$ $\{2, 3\}$, $\{3, 3\}$ y $\{4, 3\}$ crearían un número de threads superior a ese valor de 4. Para validar las combinaciones de parámetros podemos crear una formula que exprese una condición. En este caso particular sería:

$$\text{OMP1} \cdot \text{OMP2} < 5$$

Una alternativa es crear una constante que contenga el número de cores de la CPU y usarla en la ecuación:

$$\text{OMP1} \cdot \text{OMP2} < \text{maxthreads} + 1$$

La creación de constantes se realiza a través de un control gráfico de tipo lista, que muestra las ya creadas. En este control aparece una línea en blanco que permite añadir nuevas entradas. El procedimiento a seguir se muestra en la figura A.60:

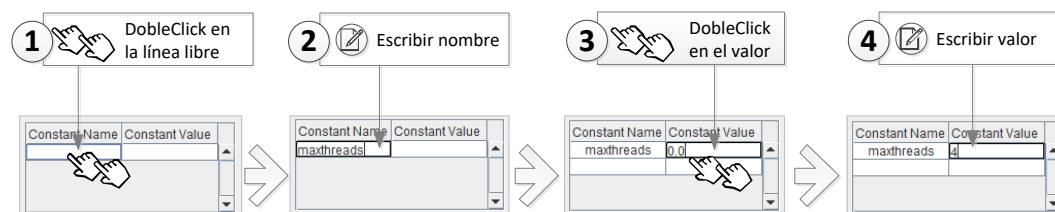


Figura A.60: Proceso para la creación de constantes durante la edición de los scripts.

- ① Hacemos doble clic sobre la línea en blanco para iniciar el modo de edición.
- ② Escribimos el nombre de la nueva constante. Tecleamos `maxthreads` y pulsamos “ENTER” para terminar la edición y añadir esta constante a la lista.
- ③ Hacemos doble clic sobre el campo del valor para iniciar el modo de edición.
- ④ Escribimos el nuevo valor y pulsamos “ENTER” en el teclado.

Una vez creada una constante, la podemos usar en nuestra fórmula. La figura A.61 muestra una ecuación que relaciona dicha constante con el número total de threads activos, obtenidos al combinar los de primer nivel (según indica el parámetro `OMP1`) y los de segundo nivel de paralelismo (parámetro `OMP2`). También se puede usar el parámetro del número de GPUs para crear reglas en referencia al número de GPUs instaladas.

Recordamos que es necesario marcar la casilla de verificación ☐ Active para indicar al simulador que use este script. No obstante, si al guardar la información del script el software detecta un error, se informará del mismo y se desactivará esta casilla hasta que el usuario corrija los errores.

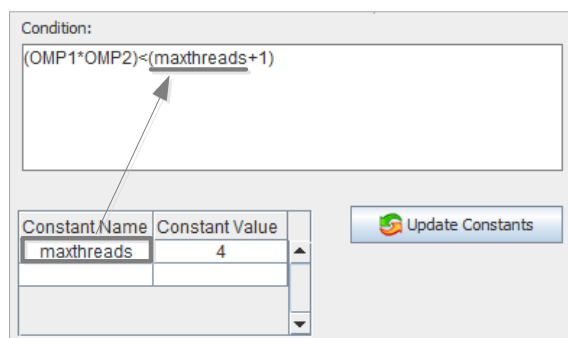


Figura A.61: Validación de los valores de los scripts mediante fórmulas. El número de threads combinados a partir de los del primer nivel (valor de OMP1) y del segundo nivel (valor de OMP2) se limitan al valor 4 de la constante `maxthreads`.

A.8.21 Preparación del simulador: Ubicación de los ficheros de configuración

Como se describió en la sección A.2, el módulo encargado de las simulaciones se denomina PARCSIM-RUN. Se trata de una aplicación de consola que necesita acceder a los archivos de configuración que describen el modelo, los parámetros algorítmicos y el modo de simulación. Estos archivos, creados usando el componente PARCSIM-MB, deben estar presentes en el directorio donde se ejecuta el simulador. Para evitar tener que copiar manualmente los ficheros cada vez que se modifiquen, lo más conveniente es informar al editor de la ubicación del simulador para que la información se actualice automáticamente. Para notificar dichos directorios se accede a la pantalla de configuración haciendo clic en la etiqueta “Config” (figura A.62).

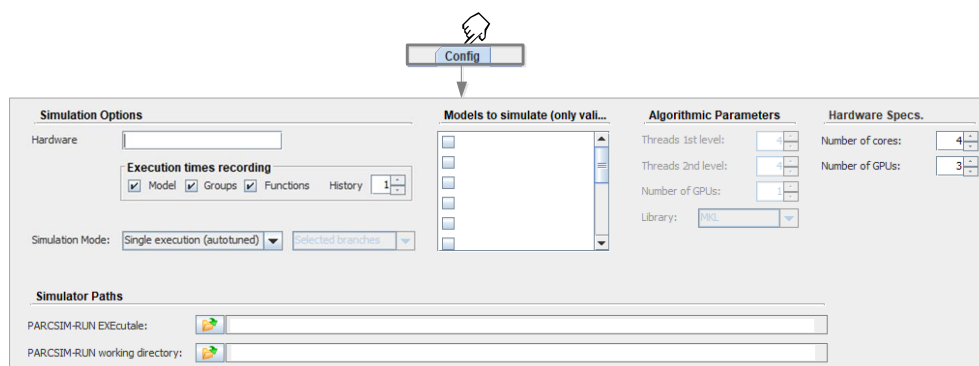


Figura A.62: Activación de la vista de configuración del simulador.

Se pueden especificar dos directorios:

- `PARSIM-RUN EXEcutable`: Ruta donde se encuentra el archivo ejecutable de `PARSIM-RUN`. Es necesario seleccionarlo para poder lanzar la simulación desde el editor `PARCSIM-MB`, como veremos en la siguiente sección.
- `PARSIM-RUN working directory`: Directorio donde se encuentran los archivos de configuración. Por defecto es la misma ruta donde esté el ejecutable del simulador, pero se puede cambiar.

La figura A.63 muestra como ejemplo el proceso de selección del directorio de instalación del simulador:

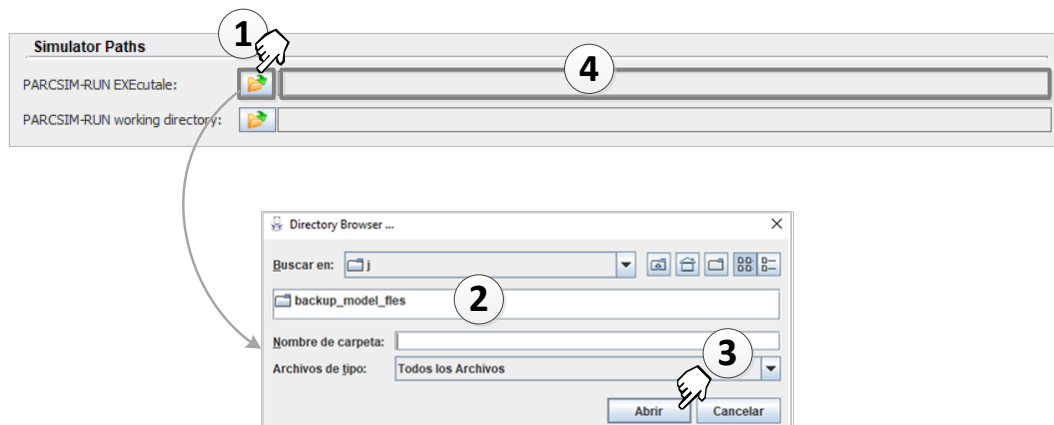


Figura A.63: Selección del directorio de ubicación del ejecutable del simulador.

- ① Hacemos clic sobre el icono de la carpeta para abrir la ventana de selección de directorios.
- ② Utilizamos los comandos disponibles para elegir el directorio.
- ③ Hacemos clic en **Abrir** para confirmar la selección.
- ④ La ruta seleccionada queda reflejada en el campo de texto correspondiente.

A.8.22 Preparación del simulador: Control de la simulación

Como paso previo adicional antes de lanzar una simulación, existen tres parámetros que se deben establecer y que son accesibles en la vista de configuración, como vemos en la figura A.64:

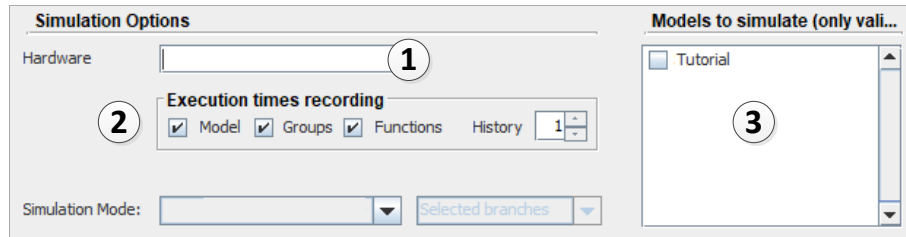


Figura A.64: Selección de los parámetros que determinan la ejecución del simulador.

- ① **Hardware:** Este campo es opcional, y es un texto libre que sirve para identificar la plataforma hardware donde se van a realizar las simulaciones. Esta información se almacena en la base de datos junto a los tiempos de ejecución. Se podrá usar para filtrar los resultados y comparar tiempos de ejecución entre distintos tipos de hardware.
- ② **Execution times recording:** Sirve para gestionar el modo en el que el simulador almacena en la base de datos los tiempos de ejecución obtenidos. Marcando ☐ **Model** se activa la grabación de registros en la base de datos de los tiempos de resolución de modelos completos. Si se marca ☐ **Groups** también se guarda el detalle por grupos. La casilla ☐ **Functions** permite indicar si se almacenan los datos de ejecución de cada función en cada grupo. History es el número máximo de registros que se almacenarán en la base de datos para una determinada combinación de modelo, función, parámetros algorítmicos y escenario.
- ③ **Model to simulate:** Se indica al simulador el modelo o modelos que se deben ejecutar. Para seleccionarlos hay que marcar la casilla de verificación junto al nombre de los modelos elegidos.

A.8.23 Preparación del simulador: Modos de simulación

A continuación se describen las configuraciones específicas para cada uno de los cuatro modos de simulación disponibles en PARCSIM, y que se describieron en la sección 5.6.2.

A.8.23.1 Modo de entrenamiento

En este modo se ejecutan de manera individual todas las funciones que están definidas en el simulador, o una selección de ellas. Los cálculos se repiten para todos los escenarios y scripts de entrenamiento, y los resultados quedan almacenados en la base de datos de autooptimización. El modo se selecciona haciendo clic sobre el desplegable etiquetado como “Simulation mode” y seleccionando la opción Training mode. El conjunto de scripts y escenarios de entrenamiento se capturan en una vista especial a la que se accede haciendo clic en Training → Scripts and scenarios ... La figura A.65 muestra la información gestionada en esta vista:

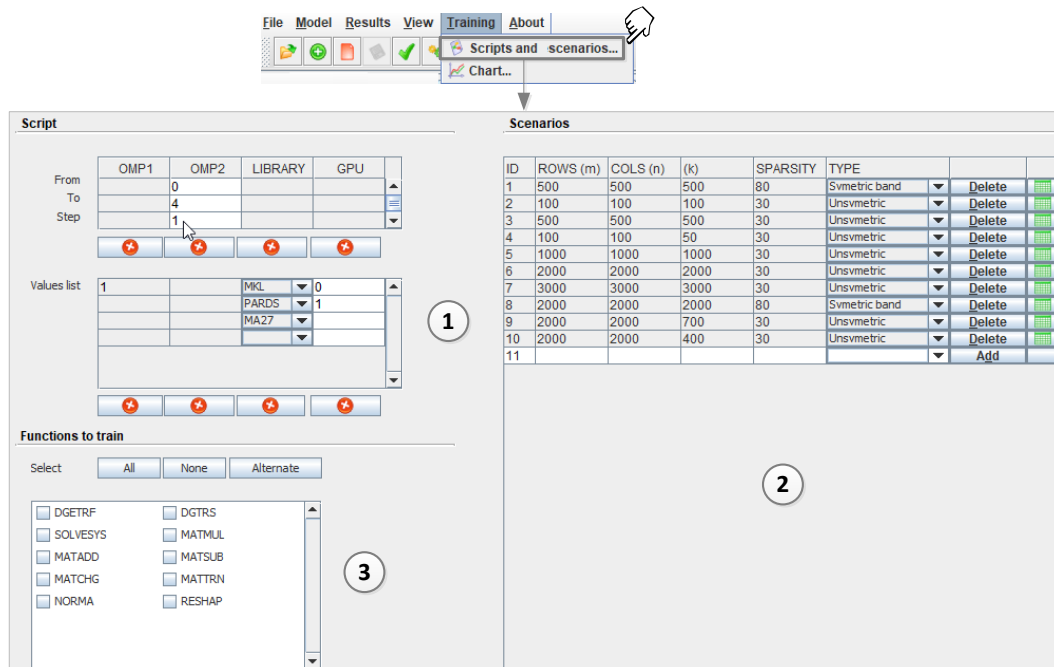


Figura A.65: Ventana de captura del conjunto de entrenamiento en el simulador (scripts y escenarios).

- ① Scripts: representan el conjunto de parámetros algorítmicos de entrenamiento. Su captura se realiza de la misma manera que los scripts aplicables a la simulación de modelos descrita en la sección A.8.20.
- ② Scenarios: representan el tamaño, tipos y dispersión de las matrices que se usarán para ejecutar las funciones.
- ③ Functions: es posible seleccionar para su entrenamiento todas o un subconjunto de las funciones disponibles.

El interfaz que permite la captura de los escenarios de entrenamiento se muestra en la figura A.66.

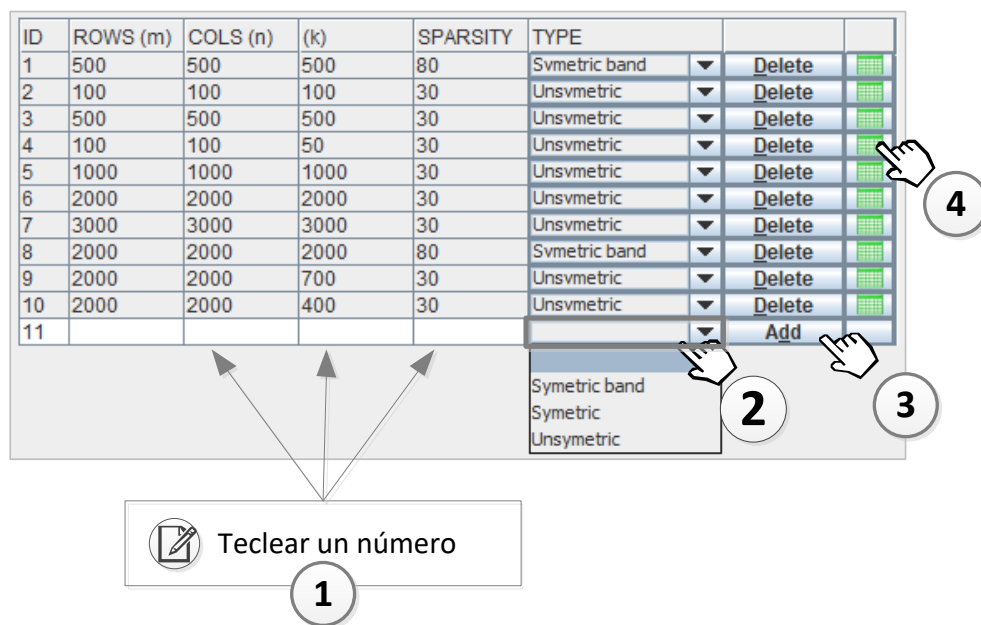


Figura A.66: Proceso de captura de un nuevo escenario de entrenamiento.

El proceso se realiza editando el contenido de la tabla mostrada. Se añade una línea por cada nuevo escenario, completando la información requerida como se describe a continuación:

- ① Se introducen el número de filas, columnas y factor de dispersión de las matrices.

- ② Se selecciona el tipo de matriz haciendo clic sobre el valor correspondiente en un desplegable.
- ③ Se pulsa en **Add** para agregar los datos recién introducidos al conjunto de escenarios de entrenamiento. Un escenario se puede eliminar pulsando sobre **Delete**.
- ④ Haciendo clic sobre este icono se mostrarán los valores de una matriz almacenados en un archivo binario dentro de la carpeta de ejecución del simulador, y que corresponda a las dimensiones, tipología y factor de dispersión del escenario correspondiente a la línea del icono.

La figura A.67 muestra en ejemplo de una matriz de dimensiones 100×100 , no simétrica y con factor de dispersión 30% generada por el simulador en ejecuciones previas con valores aleatorios.

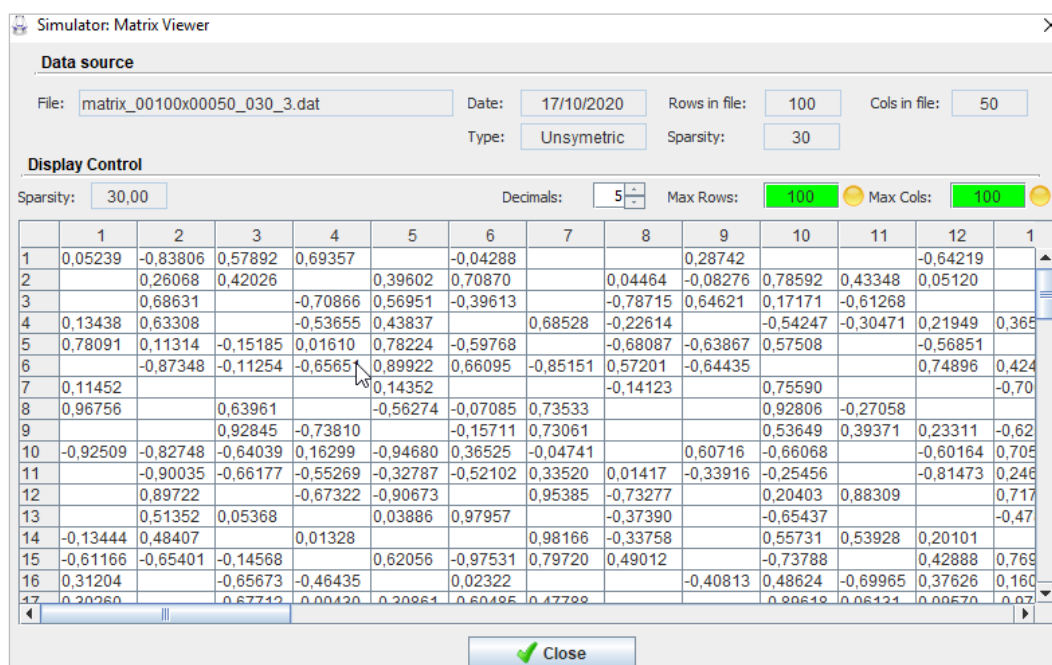


Figura A.67: Visualización del contenido de una matriz en el simulador.

A.8.23.2 Modo simple

En este modo el software usa unos valores específicos de los parámetros algorítmicos, que es necesario capturar en la pantalla de configuración como muestra la figura A.68:

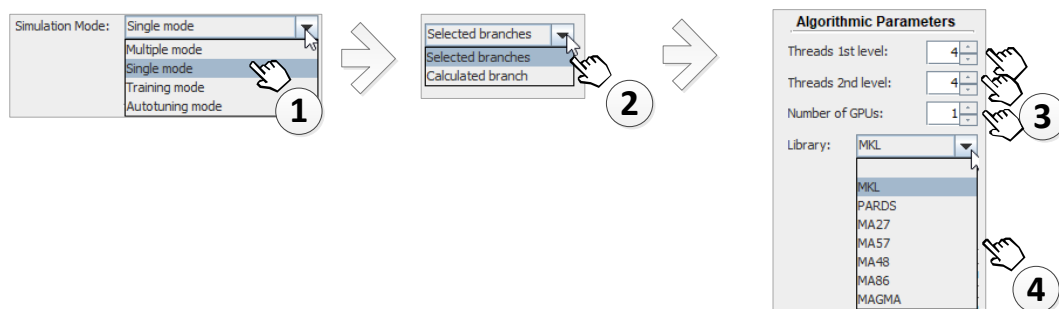


Figura A.68: Captura de la configuración y los parámetros algorítmicos aplicables al modo de simulación simple.

- ① Se selecciona el modo simple en el desplegable.
- ② Se selecciona la opción `Selected branches` para indicar al simulador que ejecute un modelo siguiendo unas rutas elegidas por el usuario (ver sección A.8.17), o la opción `Calculated branch` para que el simulador calcule la ruta más favorable en función de los parámetros algorítmicos especificados.
- ③ Se indican los threads del primer y segundo nivel de paralelismo y el número de GPUs, haciendo clic sobre los controles numéricos correspondientes.
- ④ Se selecciona la librería mediante un desplegable que muestra la lista de todos los paquetes incluidos en el simulador.

A.8.23.3 Modo múltiple

En este modo el simulador accede a los ficheros de scripts creados por el usuario para obtener de ellos los valores de los parámetros algorítmicos, y ejecuta los cálculos con todas las combinaciones que genera a partir de ellos. En la tabla A.2

vimos un ejemplo de pares obtenidos con los valores $\{1, \dots, 4\}$ del parámetro threads OMP1 y los valores $\{1, 3\}$ del parámetro threads OMP2.

El modo múltiple se activa seleccionando Multiple mode en el desplegable etiquetado como “Simulation mode”. Al igual que ocurre en el modo simple, se dispone de otro desplegable que permite elegir entre la opción Selected branches para forzar al simulador a ejecutar el modelo siguiendo unas rutas elegidas por el usuario (ver sección A.8.17), o la opción Calculated branch para que el simulador seleccione la ruta más favorable para la combinación de parámetros algorítmicos usados en cada ejecución.

A.8.23.4 Modo autooptimizado

En este modo el usuario especifica las unidades de cómputo que componen el hardware donde se ejecutará la simulación. El simulador decide los mejores parámetros algorítmicos y la mejor ruta de cálculo basándose en los resultados almacenados en la base de datos de entrenamiento, buscando el mejor aprovechamiento de los recursos para cada escenario. La figura A.69 muestra el procedimiento de captura de la configuración que se aplica a este modo:

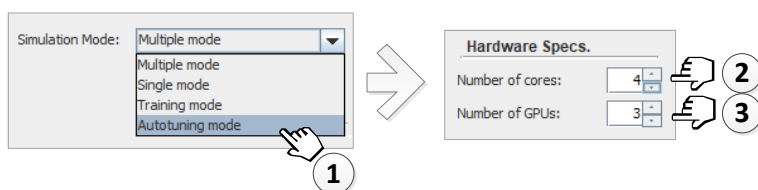



Figura A.69: Captura de la configuración y las especificaciones del hardware necesarios para el modo de simulación autooptimizado.

- ① Se selecciona el modo en el desplegable.
- ② Se indica el número de cores instalados en el hardware haciendo clic sobre el control numérico. El simulador considerará este número como el máximo número de threads que podrá crear, combinados los del primer y segundo nivel de paralelismo.

- ③ Se selecciona el número de GPUs disponibles haciendo clic sobre el control numérico. Si la librería de cálculo tiene implementada la funcionalidad de usar estos coprocesadores, el simulador podrá decidir asignarles algunas tareas.

A.8.24 Guardar la configuración del simulador

Como ocurre con los modelos, scripts y escenarios asociados, la información relativa a la configuración del simulador se almacena en un archivo que se genera al guardar los cambios. Para ello, podemos acceder al menú y seleccionamos **File** → **Save ...**. También se puede clicar sobre el icono  de la barra de herramientas. Si el sistema no nos muestra ningún error, entonces todo está preparado para la ejecución de una simulación.

A.8.25 Simulación

En este apartado vamos a ejecutar el simulador PARCSIM-RUN desde el mismo editor PARCSIM-MB. La figura A.70 muestra cómo acceder a la vista de ejecución y la información relevante que contiene:

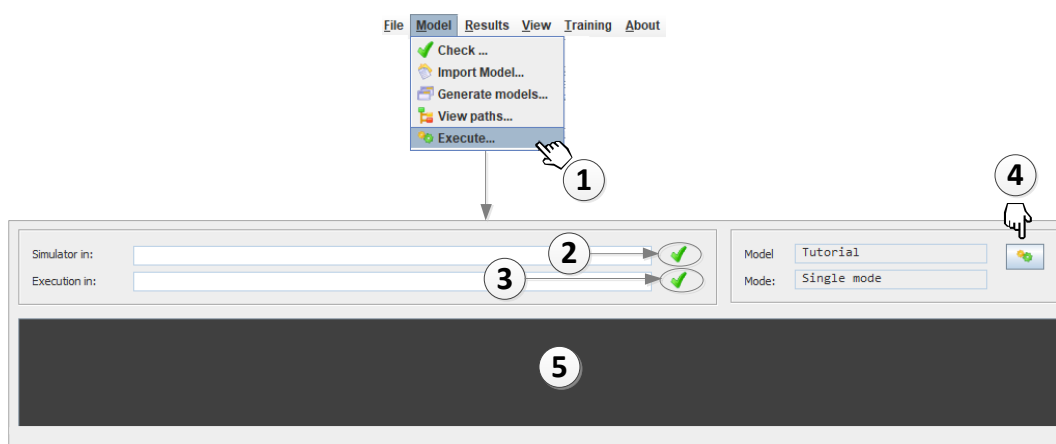









Figura A.70: Ventana de simulación.

- ① Abrimos la ventana de simulación. Para ello, podemos acceder al menú y seleccionar `Model` →  `Execute...`. También se puede hacer clic en el icono  de la barra de herramientas.
- ② El sistema comprueba que el ejecutable del simulador se encuentra en el directorio indicado en la configuración, mostrando  si no hay errores, o  en caso contrario.
- ③ El sistema comprueba que existe el directorio que contiene los archivos de configuración, mostrando  si no hay errores, o  en caso contrario.
- ④ Cuando las validaciones de los directorios son positivas, el botón , que permite iniciar la simulación, queda habilitado.
- ⑤ Haciendo clic sobre dicho botón comienza la ejecución, y el visor muestra los mensajes que el software genera para informar del estado de la simulación.

A.8.25.1 Primera ejecución: Modo múltiple con ruta seleccionada

Para probar el simulador, vamos a realizar una ejecución en el modo múltiple sobre una ruta seleccionada por el usuario. Recordamos que en este modo de simulación el software realiza múltiples ejecuciones de cada modelo, una para cada escenario y para cada combinación de parámetros algorítmicos obtenidas a partir de los scripts activos. Por tanto, antes de lanzar la ejecución vamos a comprobar que toda esta información está disponible:

- Para ello volvemos a la pantalla de configuración y nos aseguramos de haber introducido la información correcta (figura A.71):

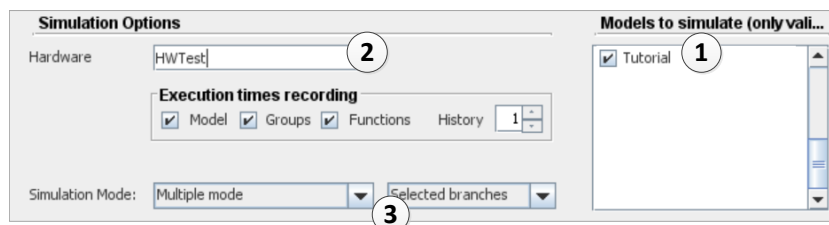


Figura A.71: Datos de configuración para una simulación en `Multiple mode`.

- ① Hemos seleccionado el modelo a simular, en nuestro caso será “Tutorial”.
 - ② Hemos tecleado una identificación del hardware. Este campo es opcional. En caso de no estar vacío, este dato se graba en la base de datos junto a los tiempos de ejecución, lo que permitirá comparar ejecuciones en varias plataformas. En este ejemplo usamos “HWTest”.
 - ③ Hemos seleccionado el modo de simulación correcto.
- Para comprobar los escenarios abrimos de nuevo el modelo (si no lo estuviera) y navegamos a la vista correspondiente (figura A.72):

Var.Name	Rows	Cols	Ran...	Reuse	Matrix file folder
A1	20	20	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
A2	20	20	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
C	20	20	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
B1	20	1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
X1	20	4	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
B2	20		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
X2	20		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
A3	20	20	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
T	20	20	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
D	20	1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
R	20	1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

Figura A.72: Datos de escenario para una simulación en Multiple mode.

- ① El tipo de matriz es el correcto.
 - ② Hemos introducido un factor de dispersión.
 - ③ Los tamaños de las matrices están completos.
 - ④ Hemos seleccionado que el simulador cree las matrices con valores aleatorios, de acuerdo al formato y factor de dispersión indicados.
- También comprobaremos el script, navegando a la vista correspondiente (figura A.73), para comprobar que:
 - ① El script está marcado como activo.
 - ② Los valores o el rango de parámetros algorítmicos están completos.

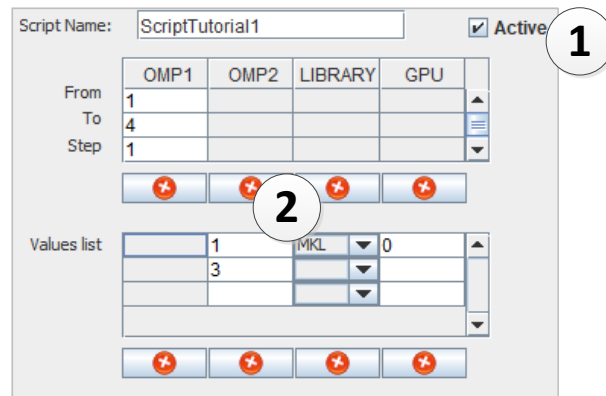


Figura A.73: Datos del script para una simulación en Multiple mode.

- Por último, y dado que queremos simular el modelo sobre una ruta determinada seleccionada por nosotros, debemos ir de nuevo al modelo para asegurarnos de que dicha ruta está preseleccionada. Para ello seguimos los pasos descritos en la sección A.8.17, como muestra la figura A.74.

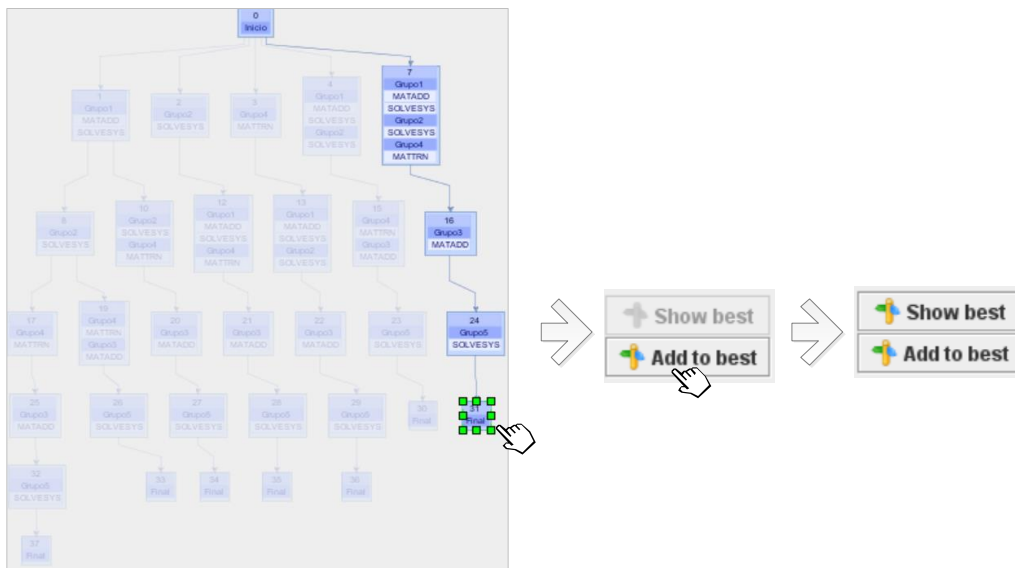

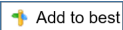
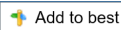


Figura A.74: Selección de una ruta para una simulación en Multiple mode.

Pulsando sobre el botón  veremos, resaltadas sobre el resto, la (o las) rutas que están seleccionadas. Si dicho botón no está habilitado, significa que no hay ninguna ruta seleccionada .

En el caso de que se muestren otras rutas diferentes a la que queremos simular, basta con hacer clic sobre cualquier zona de la pantalla que no contenga ningún nodo y pulsar sobre . Al no detectar ninguna selección el sistema entenderá que queremos eliminar cualquier selección previa, y pedirá una confirmación mediante un mensaje. A continuación, podemos marcar la ruta que hemos elegido para la simulación y hacemos clic de nuevo sobre .

Una vez que hemos establecido la configuración adecuada en el simulador, podemos iniciar la ejecución como se describió en la sección A.8.25. La figura A.75 muestra el resultado tras la ejecución.

1

2

3

SCRIPT_Scri

#

nuclei

Branch

steps

m_time

m_space

m_rows

m_cols

(s)

ompth1

ompth2

library

gpu

groups sequence

Information

Simulation has finished

Aceptar

6

1

10

20

20

0.015158

4

1

1

0

1-2+3+4-5-6-7

3

1

10

20

20

0.015402

1

3

1

0

1-2+3+4-5-6-7

2

1

10

20

20

0.015679

2

3

1

0

1-2+3+4-5-6-7

7

1

10

20

20

0.017052

2

1

1

0

1-2+3+4-5-6-7

4

1

10

20

20

0.017482

4

3

1

0

1-2+3+4-5-6-7

5

1

10

20

20

0.018114

3

1

1

0

1-2+3+4-5-6-7

0

1

10

20

20

0.018364

3

3

1

0

1-2+3+4-5-6-7

0.020707

1

1

1

0

1-2+3+4-5-6-7

Model

scenario

Branch

steps

(s)

ompth1

ompth2

library

gpu

groups sequence

Tutorial

Escenario-1

1

5

0.015158

4

1

1

0

(Inicio) (Grupo1+Grupo2+Grupo4) (Grupo3) (Grupo5) (Final)

4

Figura A.75: Resultados de la primera simulación en Multiple mode.

- ① Un mensaje que nos informa de posibles errores ocurridos durante la simulación. En esta ocasión la ejecución ha terminado de manera correcta.
- ② La ventana de la consola nos muestra los mensajes generados, y que contiene los tiempos de ejecución ordenados de menor a mayor obtenidos en todas las combinaciones de parámetros generadas por el script activo.
- ③ También se pueden identificar los valores de los parámetros algorítmicos aplicados en cada ejecución.
- ④ Si no se han producido errores, la última sección del informe muestra un resumen con la mejor selección de parámetros algorítmicos.

A.8.25.2 Consulta de resultados

El simulador PARCSIM-RUN almacena en la base de datos los tiempos de ejecución obtenidos con cualquiera de los modos descritos. Esta información se puede consultar en el editor PARCSIM-MB mediante una utilidad que encontramos en la ventana de consultas. En la figura A.76 se muestra el procedimiento para activar y usar esta funcionalidad:

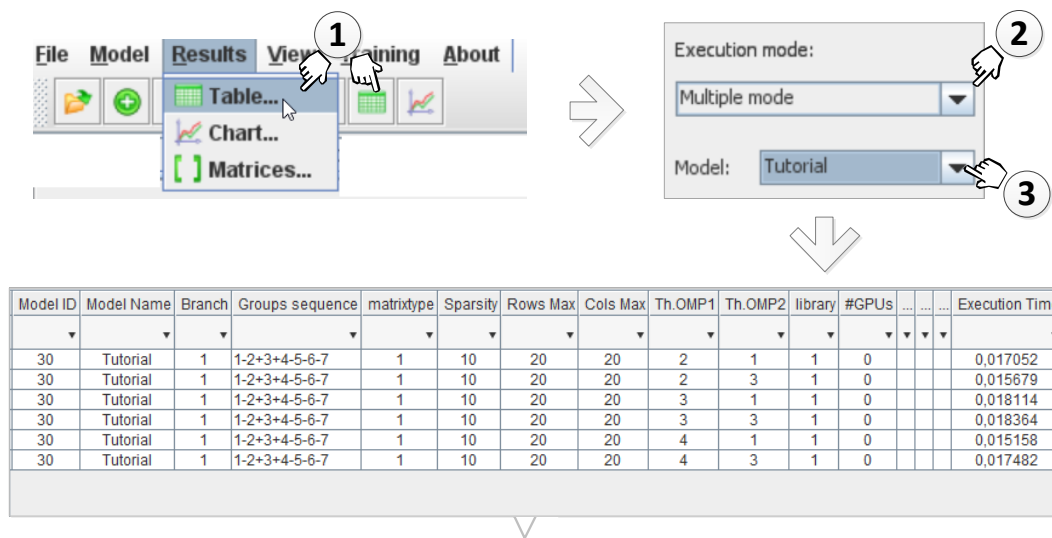


Figura A.76: Consulta de resultados.

- ① Accedemos al menú y seleccionamos Results→ Table También se puede hacer clic sobre el icono de la barra de herramientas.
- ② Seleccionamos el modo de simulación del que queremos consultar los resultados.
- ③ Seleccionamos el modelo.

Como vemos en la figura A.77, el software muestra una tabla con los registros que satisfacen los filtros aplicados. Esta información se organiza en cuatro grandes bloques:

Model Name	Branch	Groups sequence	Th.OMP1	Th.OMP2	library	#GPUs	matrixtype	Sparsity	Rows Max	Cols Max	Group	Step	Function	Execution Time
▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	5 ▼
Tutorial	1	1-2+3+4-5-6-7	1	1	1	0	1	10	20	20				0,030317
Tutorial	1	1-2+3+4-5-6-7	1	1	1	0	1	10	20	20	Grupo1	2		0,002463
Tutorial	1	1-2+3+4-5-6-7	1	1	1	0	1	10	20	20	Grupo1	2	MATADD-MKL	0,000104
Tutorial	1	1-2+3+4-5-6-7	1	1	1	0	1	10	20	20	Grupo1	2	SOLVESYS-MKL	0,002347
Tutorial	1	1-2+3+4-5-6-7	1	1	1	0	1	10	20	20	Grupo2	2		0,000067

1 Modelo y ruta

2 Escenario

3 Parámetros
algorítmicos

4 Tiempo de ejecución
desglosado por
grupos y funciones

Figura A.77: Información obtenida en la consulta de resultados.

- ① Información sobre la ruta usada por el simulador, y que muestra todos los grupos del modelo con la siguiente codificación:
 - Signo + que separa los códigos de los grupos que se han calculado en paralelo, en una misma etapa de tiempo.
 - Signo - separa etapas de tiempo.
- ② Datos sobre el escenario (tamaño y formato de las matrices).
- ③ Información sobre los parámetros algorítmicos. Al tratarse de una simulación en modo múltiple, se usan las combinaciones extraídas de los scripts.
- ④ Los tiempos de ejecución a tres niveles: el modelo completo, los grupos y las funciones que componen cada grupo.
- ⑤ Una línea para filtros permite al usuario consultar valores concretos.

Observamos en la tabla que la columna `Groups sequence` muestra la misma ruta en todos los registros porque hemos ejecutado el modelo con una ruta preseleccionada. En caso de permitir al simulador la selección automática de la ruta, entonces la columna `Groups sequence` puede contener valores diferentes en función de los parámetros algorítmicos aplicados en cada ejecución.

Se pueden aplicar filtros sobre los valores de una o varias columnas de la tabla. La figura A.78 muestra el modo de activarlos:

- ① Se comienza pulsando sobre el icono  en la columna por la que se quiere filtrar.

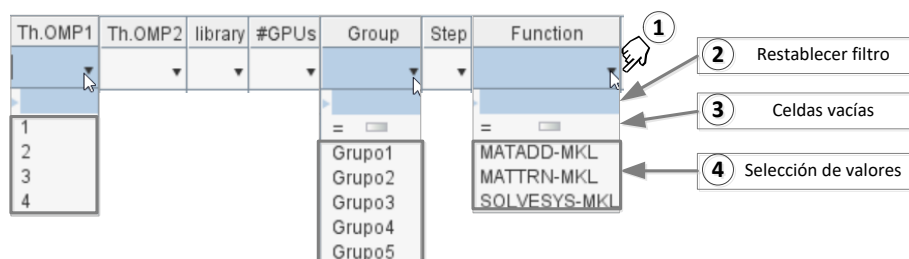




Figura A.78: Uso de filtros en la consulta de resultados.

- ② En caso de que hubiera una selección anterior, esta opción permite restablecerla.
- ③ Esta opción se muestra únicamente si existen celdas vacías en la columna seleccionada, con objeto de poder filtrar por ellas.
- ④ Aquí vemos la lista de todos los valores de dicha columna, y nos permite mostrar únicamente los registros que contienen un cierto valor.

A.8.25.3 Consulta gráfica de tiempos de ejecución

El software ofrece un modo de representar mediante gráficos de líneas los tiempos de ejecución obtenidos al simular los modelos para cada conjunto de parámetros algorítmicos. Esta utilidad se encuentra accesible en el submenú de consulta de resultados, como podemos ver en la figura A.79:

- ① Seleccionamos Results →  Chart ..., o hacemos clic sobre el icono  de la barra de herramientas.
- ② Seleccionamos los valores de los filtros y el modelo o modelos que queremos mostrar.
- ③ Obtenemos una gráfica con los valores medios, para cada tamaño de las matrices. Dado que hemos simulado solo un escenario, obtenemos solo un conjunto de puntos, que representan los tiempos de ejecución para el Escenario-1, con tamaño de matrices de 20×20 .
- ④ Se muestra una serie para cada conjunto de parámetros algorítmicos, como indica la leyenda.

- ⑤ Se incluye una tabla que muestra, además del valor medio mostrado en la gráfica, los valores mínimos y máximos.
- ⑥ Haciendo clic sobre los tiempos de ejecución mostrados en una serie, se obtiene una imagen de la ruta seguida para conseguir dicho tiempo, como se muestra en ⑦.

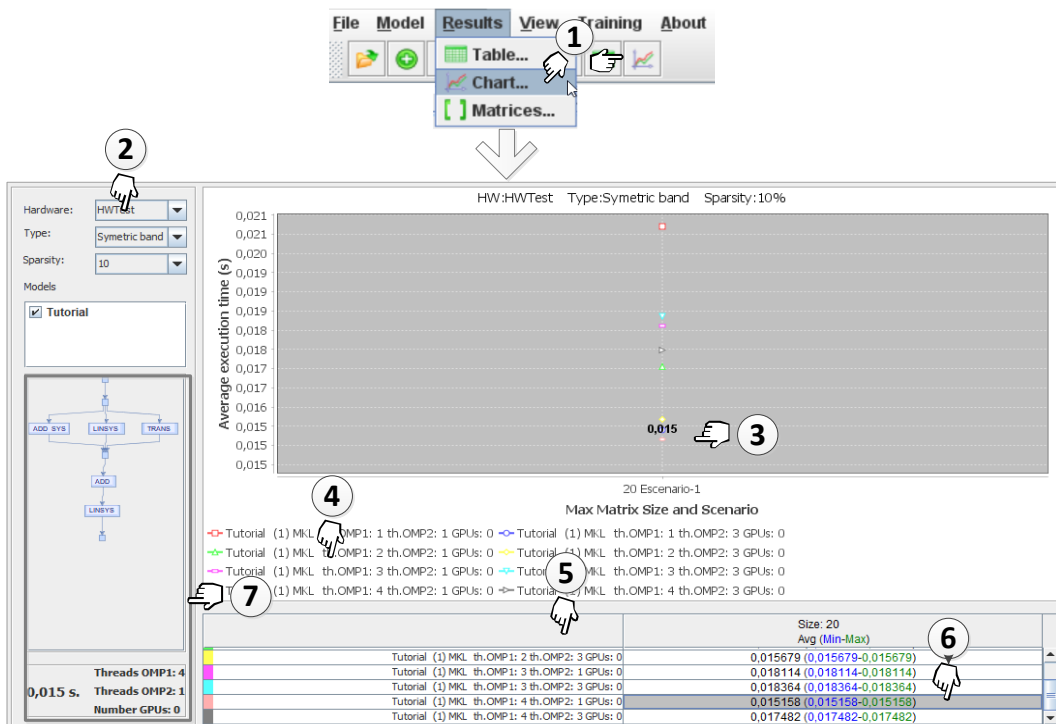


Figura A.79: Consulta gráfica de los tiempos de ejecución en función de los parámetros algorítmicos. Se muestran datos obtenidos en la primera simulación en modo múltiple.

A.8.25.4 Segunda ejecución: Modo múltiple con ruta calculada

En la sección A.8.25.1 realizamos la simulación del modelo Tutorial en el modo de ejecución múltiple. En dicha ejecución el simulador realizaba los cálculos que marcaba la ruta que previamente habíamos seleccionado.

A continuación vamos a repetir la ejecución pero delegando en el simulador la tarea de seleccionar la mejor ruta, que deberá calcular en cada iteración justo

antes de proceder a realizar los cálculos. Este proceso de estimación, que fue descrito en la sección 5.5.4, necesita disponer de los tiempos de entrenamiento de las funciones implementadas en el simulador. La obtención de dichos tiempos se realiza ejecutando el simulador en el modo de entrenamiento.

Por tanto, modificamos la configuración del simulador y la ajustamos con los valores que muestra la figura A.80. Tras ello lanzamos la ejecución.

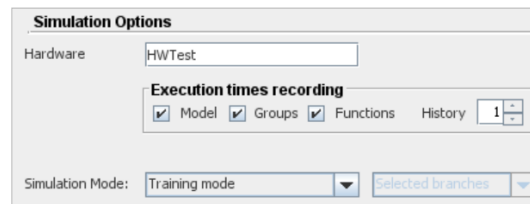


Figura A.80: Configuración del simulador para el modo de entrenamiento.

Podemos comprobar la información obtenida realizando una nueva consulta a la base datos aplicando un filtro para el modo de entrenamiento, con lo que obtendremos una vista de los resultados similar al que muestra la figura A.81.

hardware	Simulation start	matritype	Sparsity	Rows Max	Cols Max	Th.OMP1	Th.OMP2	library	#GPUs	Function	Execution Time
HWTest	19/07/2020 11:15:58	1	10	10	10	1	0	3	0	SOLVESYS-MA27	0,000151
HWTest	19/07/2020 11:15:58	1	10	10	10	1	0	3	0	MATMUL-MA27	0,002406
HWTest	19/07/2020 11:15:58	1	10	10	10	1	0	3	0	MATADD-MA27	0,000049
HWTest	19/07/2020 11:15:58	1	10	10	10	1	0	3	0	MATSUB-MA27	0,000048
HWTest	19/07/2020 11:15:58	1	10	10	10	1	0	3	0	MATCHG-MA27	0,000035
HWTest	19/07/2020 11:15:58	1	10	10	10	1	0	3	0	MATTRN-MA27	0,000041
HWTest	19/07/2020 11:15:58	1	10	10	1	1	0	3	0	NORMA-MA27	0,000031

Figura A.81: Consulta de resultados de entrenamiento.

Con la información de entrenamiento disponible, podemos configurar el simulador para una ejecución en el modo múltiple con selección automática de rutas (figura A.82).

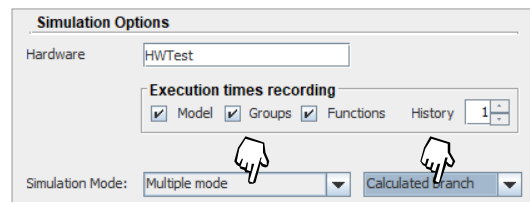


Figura A.82: Configuración del simulador para la ejecución en Multiple mode con selección automática de rutas.

En la figura A.83, donde se observan los resultados de la simulación, comprobamos que el proceso de selección ha encontrado tres rutas distintas, que están en consonancia con los parámetros algorítmicos de cada iteración. Por ejemplo, la ruta etiquetada como ③ supone una ejecución secuencial de todos los grupos porque el parámetro que indica el primer nivel de paralelismo (ompth1) tiene un valor de 1, lo que fuerza a ejecutar como máximo un grupo en cada etapa del algoritmo.

m_rows	m_cols	(s)	ompth1	ompth2	library	gpu	groups sequence
20	20	0.015166	3	1	1	0	1-2+3-4+5-6-7
20	20	0.015774	4	1	1	0	1-2+3-4+5-6-7
20	20	0.017110	2	3	1	0	1-2+3-4+5-6-7
20	20	0.017210	2	1	1	0	1-2+3-4+5-6-7
20	20	0.0174	3	1	1	0	1-2+3+4-5-6-7
20	20	0.017	1	1	1	0	1-2-3-4-5-6-7
20	20	0.0196	4	1	1	0	1-2+3+4-5-6-7
20	20	0.027217	1	1	1	0	1-2-3-4-5-6-7

(s)	ompth1	ompth2	library	gpu	groups sequence
0.015166	3	1	1	0	(Inicio) (Grupo1+Grupo2) (Grupo4+Grupo3) (Grupo5) (Final)

Figura A.83: Resultados obtenidos en una simulación múltiple con selección automática de rutas. El software ha detectado tres posibles rutas en relación con los parámetros algorítmicos aplicados.

De entre todas las obtenidas, la ruta que supone un menor tiempo de ejecución es la que ejecuta en paralelo los grupos {Grupo1, Grupo2} y los grupos {Grupo3, Grupo4} en la etapa siguiente, con asignación de 3 threads al primer nivel de paralelismo usando la librería con identificador 1 (que corresponde a MKL, como indica la tabla 5.1).

A.8.25.5 Visor de rutas interactivo

La ejecución anterior ha usado un procedimiento de búsqueda de la ruta con el menor tiempo de ejecución estimado. Dicho proceso puede ser visualizado mediante una herramienta incorporada a PARCSIM-MB a la que se accede desde el visor de rutas descrito en la sección A.8.17 de este manual.

Mostraremos el uso de dicha herramienta sobre el modelo Tutorial creado a lo largo de las secciones precedentes de este documento. Como vemos en la figura A.84, al marcar la casilla ☐ Calculate best, se muestran los campos que recogen las especificaciones del hardware ① y que permiten seleccionar un escenario ②. Cada vez que se modifica cualquiera de estos datos, el gráfico se actualiza para mostrar la ruta más rápida usando dicha información.

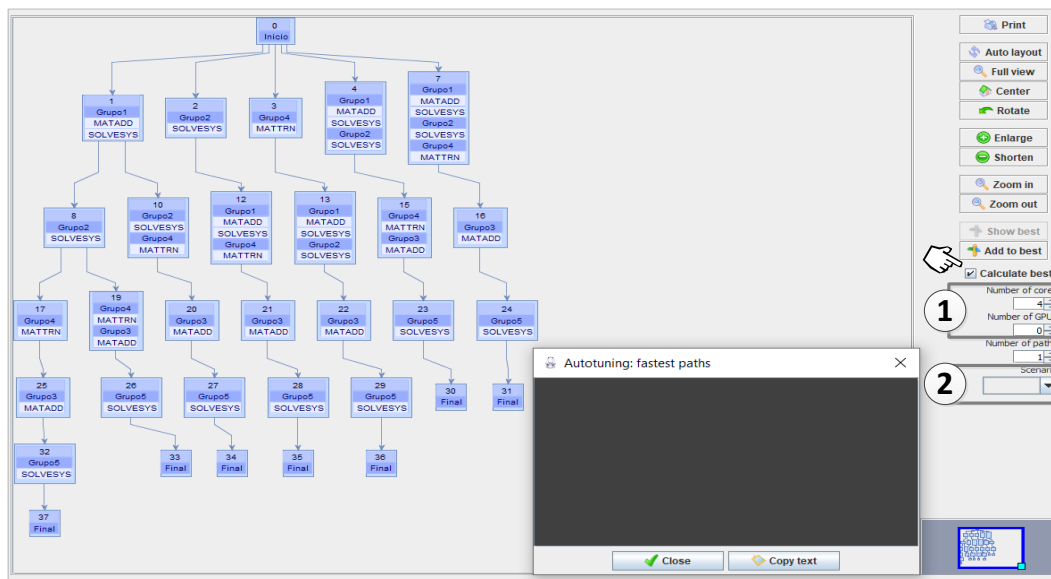


Figura A.84: Acceso a la funcionalidad interactiva de cálculo de la ruta con menor tiempo de ejecución estimado en función de unas determinadas especificaciones de hardware y tamaños de problema.

Una vez que el usuario ha introducido la información, el grafo se actualiza y resalta la ruta elegida. En cada nodo se muestran los cálculos a realizar, el número de threads asignados al segundo nivel de paralelismo y la librería a utilizar, como vemos en la figura A.85.

Es posible especificar la cantidad de rutas a obtener. El software las resalta en el grafo, y muestra en el visor de texto el detalle de cada una de ellas, ordenadas de menor a mayor tiempo de ejecución, y la diferencia de cada una respecto a la precedente. La figura A.86 muestra un ejemplo con el cálculo de tres rutas.

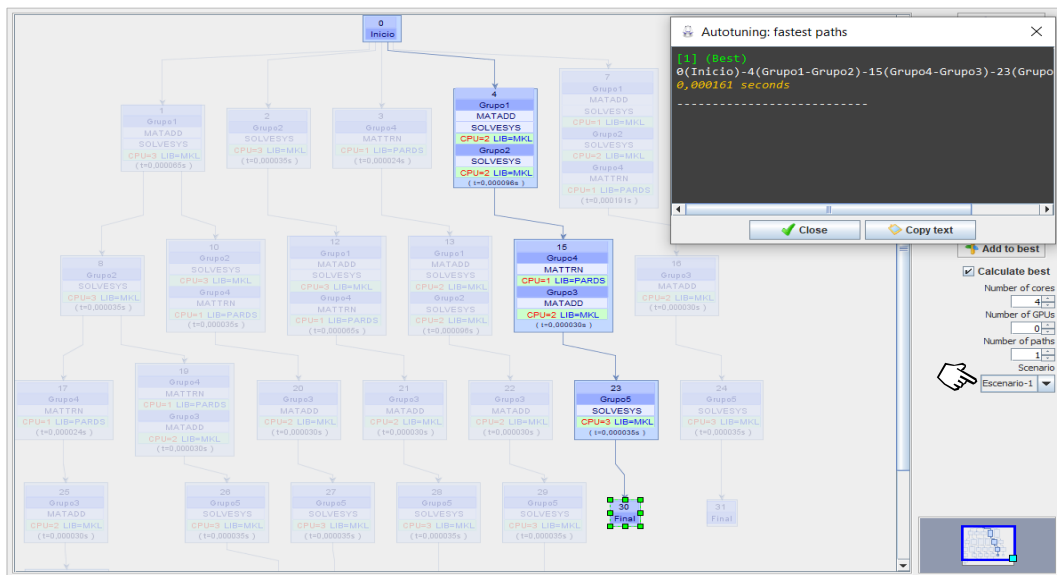


Figura A.85: Obtención de la mejor ruta mediante la funcionalidad interactiva incorporada en el software.

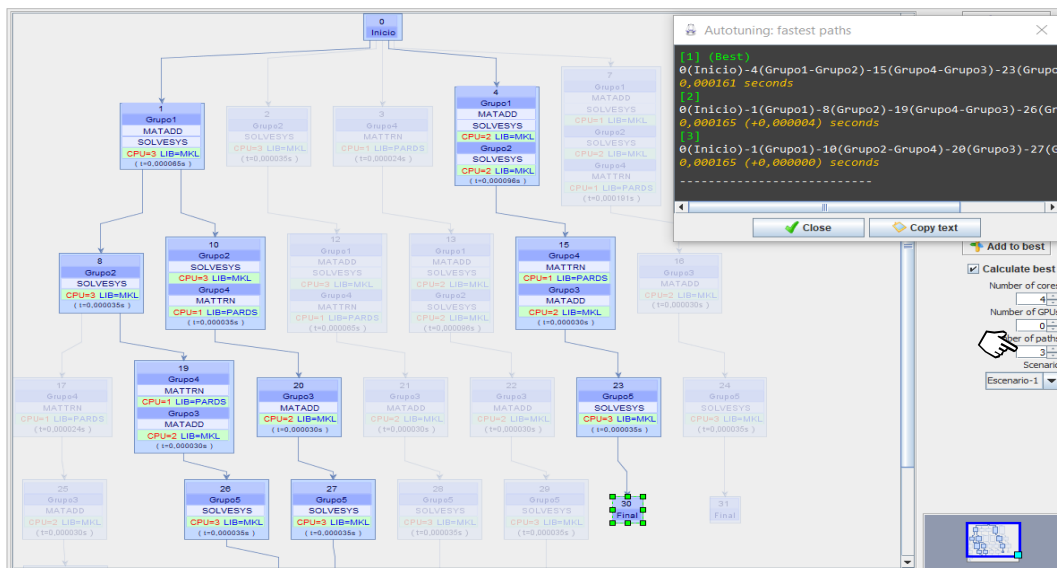


Figura A.86: Obtención de las tres rutas teóricamente más eficientes. El cuadro de texto ofrecido en el visor de rutas muestra los tiempos obtenidos con cada ruta, ordenados de menor a mayor, y las diferencias entre ellas.

A.8.26 Funciones avanzadas

En los siguientes apartados se describen un grupo de funcionalidades adicionales ofrecidas por el simulador.

A.8.26.1 Enlace de argumentos en rutinas

En las rutinas es posible indicar el modo en el que los valores de los argumentos de salida de unas funciones pueden asignarse a argumentos de entrada de otras funciones que se ejecutan posteriormente en la misma rutina. Para mostrarlo partimos de una sencilla rutina creada a modo de ejemplo, que se muestra en la figura A.87.

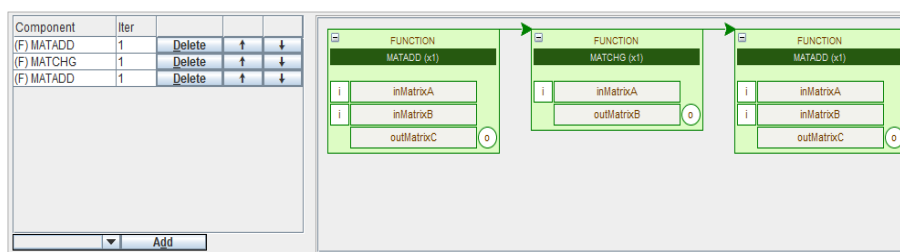


Figura A.87: Rutina de ejemplo para el proceso de enlace de argumentos.

Si queremos indicar que el argumento de salida `outMatrixC` de la primera función `MATADD` se usa como argumento de entrada `inMatrixA` de la función `MATCHG`, el procedimiento se realiza como se muestra en la figura A.88:

- ① Se selecciona con el ratón el icono ① del argumento de origen.
- ② Se suelta sobre el icono ② del argumento de destino.

Del mismo modo que hemos creado el enlace anterior, podemos crear otro más para obtener un resultado como el de la figura A.89.

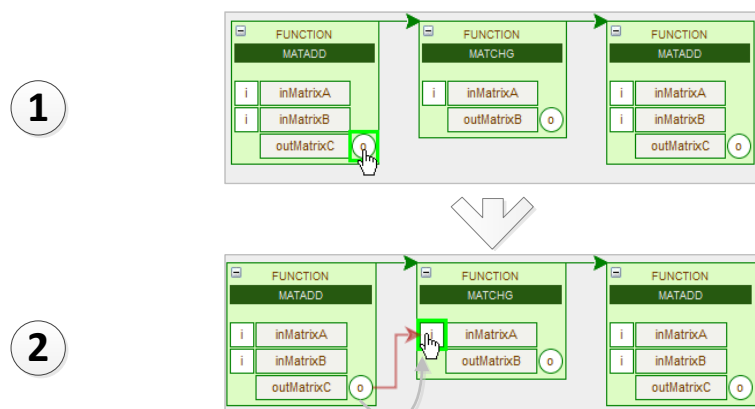


Figura A.88: Enlace de argumentos mediante el procedimiento de arrastrar desde el origen y soltar en el destino en el grafo de una rutina.

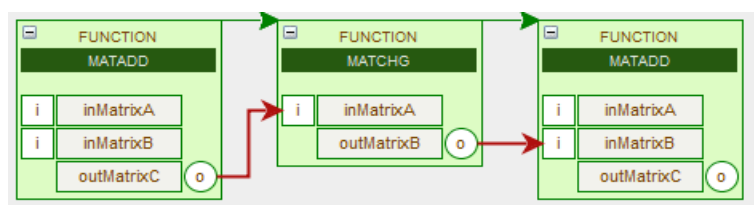


Figura A.89: Rutina de ejemplo que muestra dos enlaces entre parámetros.

A.8.26.2 Ejemplo de anidamiento de rutinas

Como se describió en 5.2.2, es posible incluir rutinas que se ejecutan como parte de otras rutinas, y esto es posible hasta dos niveles de anidamiento. Con ello, el usuario puede definir pequeñas secuencias de cálculos y reutilizarlas en sus algoritmos.

En este apartado mostramos un ejemplo de una rutina anidada que hace uso de la que hemos creado en la sección anterior para mostrar el modo de establecer enlaces entre los parámetros. En el grafo de la figura A.90 se encuentran representados todos los elementos que componen esta nueva rutina: las funciones MATADD, MATTRN y SOLVESYS, y la rutina Demo que se usa en tres ocasiones. Las funciones se representan mediante bloques bordeados con una línea continua. Las rutinas lo hacen con líneas discontinuas. El enlace entre los parámetros se indica mediante líneas dirigidas de color verde cuando corresponden a la rutina anidada, y en color rojo si son enlaces creados para la rutina en curso.

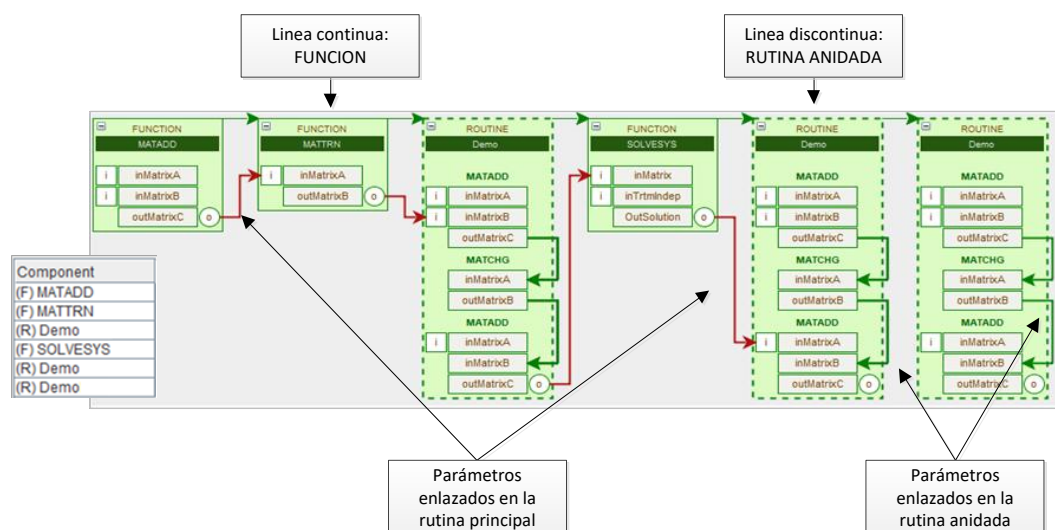


Figura A.90: Ejemplo de rutina anidada con enlace entre argumentos propios y heredados de la rutina anidada.

A.8.26.3 Importación de modelos

Es habitual en el ámbito científico la reutilización de algoritmos que resuelvan determinados problemas numéricos. En PARCSIM, durante el proceso de creación de un modelo, es posible importar los grupos y sus interdependencias desde otro modelo que ha sido creado con anterioridad.

Para ello, se hace clic sobre `Model` → `Import model...`. El software nos muestra una ventana del sistema operativo para buscar archivos con extensión “.mdl” en el directorio de la aplicación, como muestra la figura A.91.

Una vez que se ha seleccionado el fichero que contiene el modelo a importar, el software lee su contenido y lo añade al interfaz gráfico. Los grupos del modelo original se renombran añadiéndoles al inicio el carácter `_`. Las variables de modelo se añaden a las actuales renombrándolas de igual manera que se hace con los grupos. La figura A.92 muestra el grafo donde se observan (parte inferior) los grupos que han sido importados. El usuario debe modificar las dependencias para colocar los nuevos grupos en el lugar adecuado del grafo.

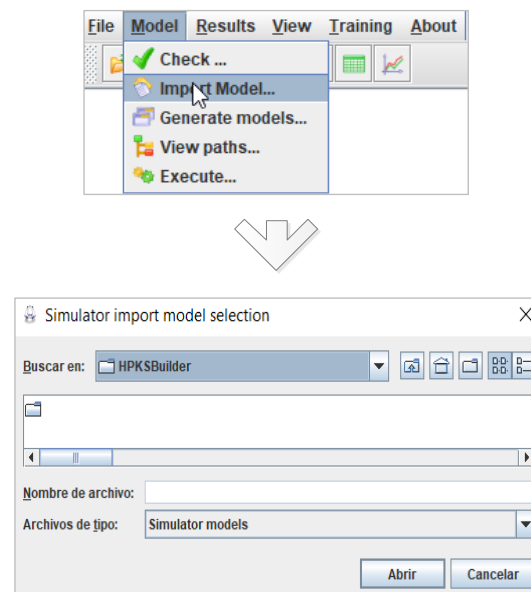


Figura A.91: Selección de archivos durante la importación de modelos.

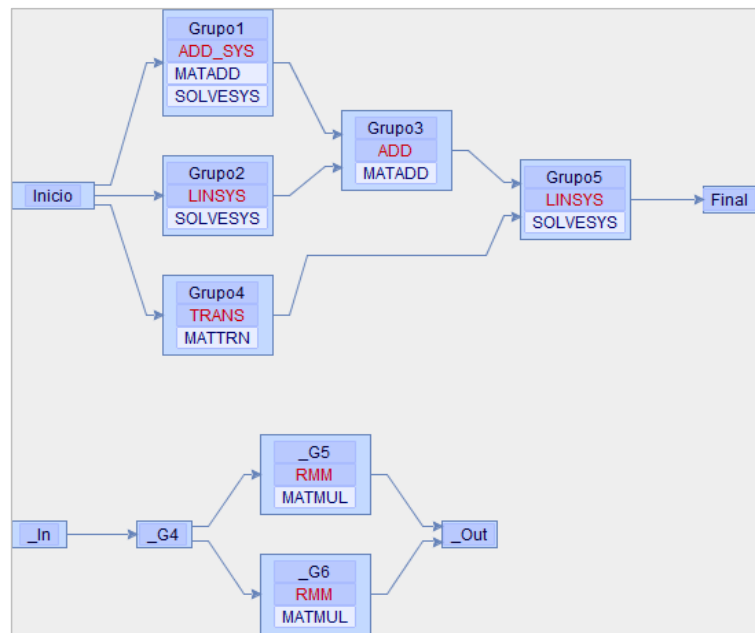


Figura A.92: Grupos importados desde otro modelo e insertados en el modelo activo.

A.8.26.4 Inserción de modelos dentro de grupos

En este manual hemos creado el modelo `Tutorial` para representar el algoritmo que resuelve un problema numérico sencillo mediante su descomposición en subproblemas representados mediante grupos que resuelven una determinada rutina. Ahora bien, según la complejidad del problema a modelar, puede resultar necesario que un grupo tenga que resolver un modelo completo.

PARCSIM contempla esta posibilidad. Podemos imaginar un nuevo problema igual al planteado en la figura A.10(a) salvo que ejecute, en una última etapa, una multiplicación de matrices. Podemos entonces usar el editor para crear un nuevo `Grupo6` y colocarlo en secuencia a continuación del `Grupo5`. Le asignaremos una rutina de multiplicación de matrices disponible, `RMM`. El modelo queda como el mostrado en la figura A.93.

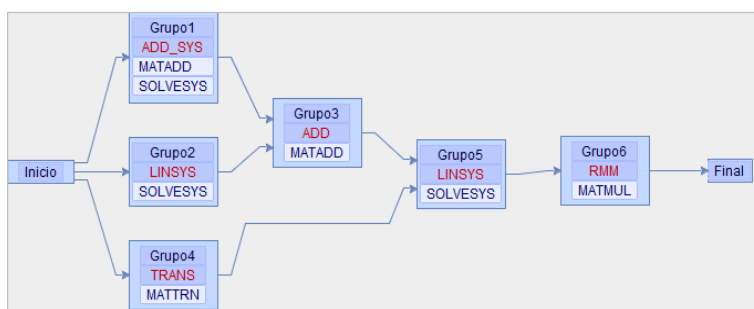


Figura A.93: Modificación del modelo `Tutorial` que incluye un nuevo `Grupo6` que resuelve una multiplicación de matrices.

Si quisiéramos resolver la multiplicación de matrices mediante una descomposición en bloques, podríamos pensar en usar algún modelo creado con tal fin. Se podría importar tal modelo, pero esta acción añadiría los grupos, como hemos visto en la sección anterior. Otra alternativa es incluir uno de los modelos como sustitución de la rutina `RMM`.

El procedimiento, reflejado en la figura A.94, es el siguiente:

- ① Seleccionamos el `Grupo6` haciendo clic sobre él en el grafo.
- ② Hacemos clic sobre el desplegable junto al campo de selección de modelo.

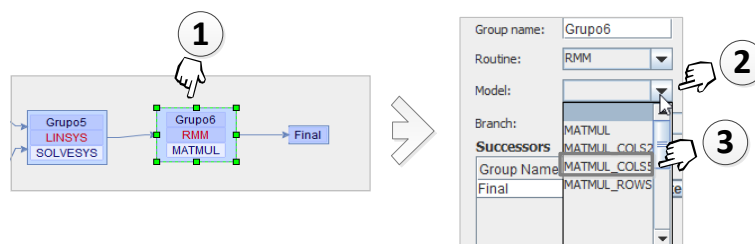


Figura A.94: Selección de un modelo para ser ejecutado en el Grupo6.

- ③ Seleccionamos en la lista el nombre del modelo que queremos insertar. en este caso será MATMUL_COLS50_G1, un modelo que resuelve una multiplicación de matrices por bloques resultantes al dividir por columnas la matriz original en dos bloques de igual tamaño.

El resultado de esta acción se muestra en la figura A.95, donde conviene destacar:

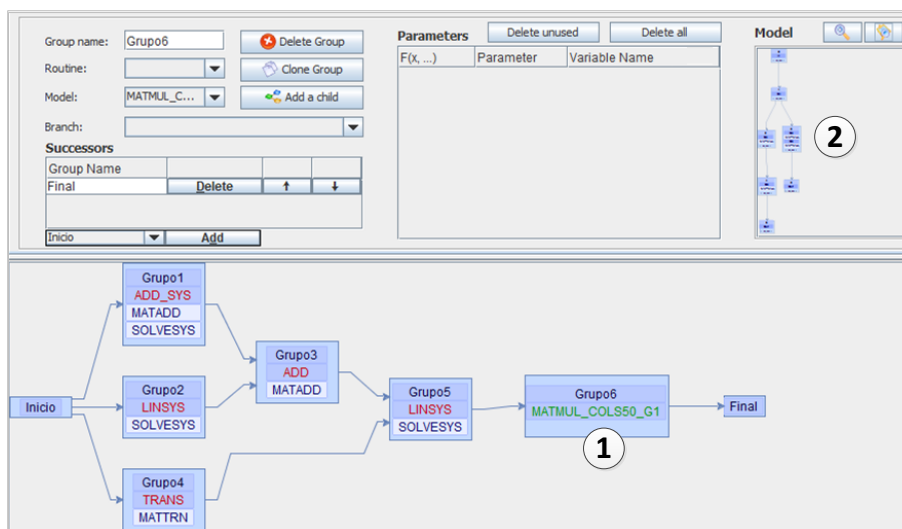


Figura A.95: Interfaz que muestra el modelo MATMUL_COLS50_G1 para ser ejecutado en el Grupo6.

- ① El grafo muestra que el modelo MATMUL_COLS50_G1 está integrado en el Grupo6.
- ② El visor gráfico, que habitualmente representa las funciones de una rutina, ahora muestra todas las rutas del modelo importado, MATMUL_COLS50_G1.

Para que el simulador sepa exactamente cómo ejecutar el modelo insertado en el Grupo6, PARCSIM necesita conocer una ruta. Esta se puede seleccionar, bien mediante el desplegable que se muestra junto a la etiqueta “Branch”, o bien directamente sobre el grafo que muestra el árbol de rutas del modelo MATMUL_COLS50_G1. La figura A.96 muestra ambas opciones.

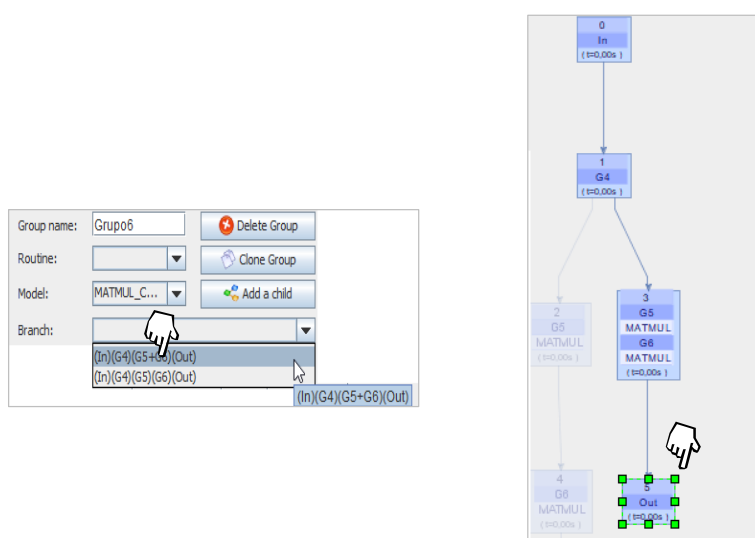


Figura A.96: Selección de la rama del modelo MATMUL_COLS50_G1 que se va a ejecutar en el Grupo6.

La selección de una rama modifica de nuevo el grafo para reflejar esta información (figura A.97).

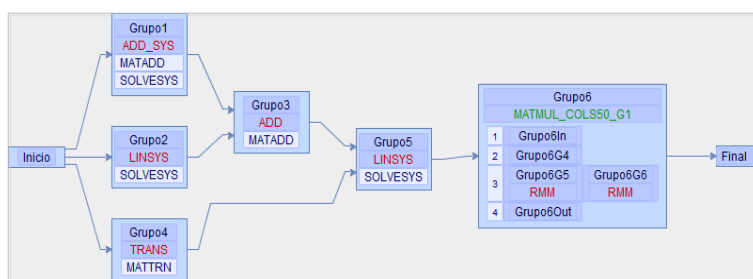


Figura A.97: Grafo del modelo Tutorial con el Grupo6 ejecutando una rama del modelo MATMUL_COLS50_G1.

A diferencia del proceso de importación descrito en la sección anterior, donde se creaban tantos grupos nuevos como aportaba el modelo importado, la inclusión de un modelo en un grupo hace que dicho modelo sea tratado de manera indivisible, como si fuera una rutina. En la figura A.98 se representa el nuevo árbol de rutas con nodos en cuyo interior se resuelve el modelo MATMUL_COLS50_G1.

No se permite más de un nivel de recursividad en el uso de modelos insertados. Por este motivo el software no muestra en la lista de modelos a anidar los que ya tienen un anidamiento en alguno de sus grupos.

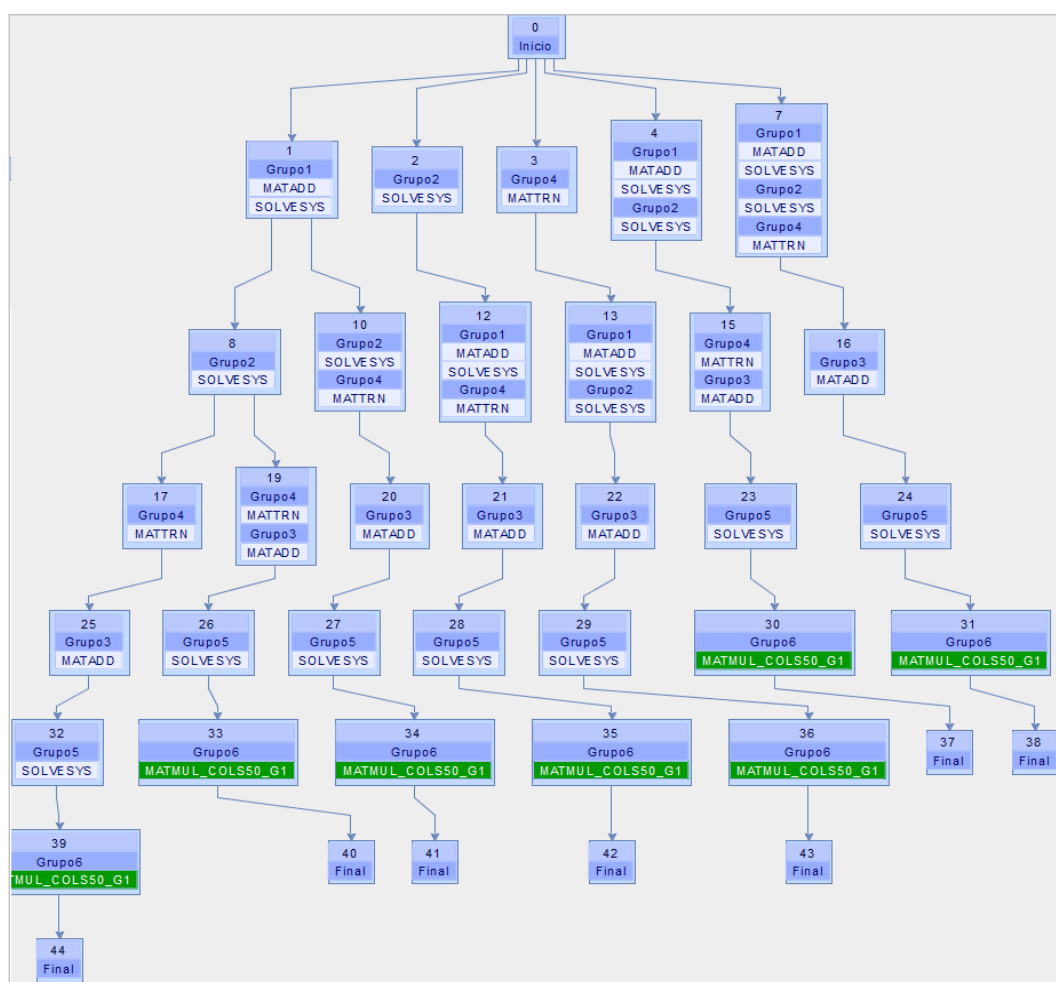


Figura A.98: Árbol de rutas del modelo Tutorial con el Grupo6 ejecutando el modelo MATMUL_COLS50_G1.

A.9 Preguntas frecuentes

¿Cómo debo copiar los modelos que he creado cuando quiero usarlos en otro ordenador?

Los modelos creados en PARCSIM se almacenan en varios archivos de texto. Todos ellos comienzan con el nombre del modelo y con una extensión que indica el contenido:

- `.mdl`: Habrá un único archivo con esta extensión. Contiene la definición del modelo, con los grupos y sus dependencias, así como las rutinas que ejecutan y las variables.
- `.ser`: Es un único archivo que existe si el árbol de rutas ha sido creado. Almacena en formato binario todas la ramas del árbol.
- `.sce`: Uno o más archivos, que corresponden a los escenarios creados para el modelo.

Por ejemplo, para un modelo de nombre `ModeloAnexo` podremos tener:

- Un archivo `ModeloAnexo.mdl`.
- Un archivo `ModeloAnexo.ser` si se ha generado el árbol. En caso contrario este archivo no existirá.
- Uno o más archivos `ModeloAnexo_SCENARIO_*.sce` (* se sustituye por el nombre del escenario en cada caso).

Además de los archivos anteriores, también se debe copiar `routines.rou`, ya que contiene la definición de las rutinas.

¿Por qué obtengo un mensaje durante la generación del árbol de rutas indicando que se ha alcanzado el número máximo de nodos o el límite de tiempo?

Dependiendo del número de grupos y de sus dependencias, el proceso que crea el árbol de rutas puede tomar varios minutos, o estar formado por un gran número de nodos. Por este motivo se han creado dos variables que permiten controlar dicho proceso:

- `maxTimeToGenerateTrees`: Especifica el tiempo límite (en segundos). Alcanzado ese valor el proceso se detiene e informa al usuario mediante un mensaje.
- `maxNumNodesinTrees`: Especifica el número de nodos máximo permitidos en un árbol.

Estos valores se puede modificar editando el fichero `treeslimits.cfg` que se puede encontrar en el directorio de la aplicación. Una vez realizado el cambio se debe cerrar la aplicación y volver a abrirla para que los nuevos valores tengan efecto.

¿Por qué obtengo un mensaje avisando de que el proceso para elegir la mejor ruta no tiene información suficiente en la base de datos?

En un proceso de simulación autooptimizado el software estima la ruta que resuelve el modelo de la manera más rápida. Para ello consulta una base de datos de entrenamiento donde se almacenan los tiempos de ejecución de las funciones básicas que componen los algoritmos. Se deben encontrar tiempos correspondientes a los tipos y tamaños de las matrices del modelo que se quiere simular. En caso de no encontrarlos, se buscan aquellos más parecidos. El software indica que no hay información suficiente en caso de no encontrar datos para alguna función.

A.10 Software de terceros

El simulador consta de dos componentes:

- El componente PARCSIM-MB (Model Builder): Ofrece un asistente gráfico para la gestión del simulador, desarrollado en JAVA. Para añadir todas las funcionalidades de este módulo, se han utilizado las siguientes librerías:
 - `jgraph`: Es una librería Open Source para JAVA que permite crear diagramas como los que utiliza el simulador para mostrar los modelos y el árbol de rutas. Web: <https://github.com/jgraph/jgraphx>

- SQLite database engine: Es una librería escrita en C que proporciona un sencillo gestor de bases de datos SQL. El código fuente está disponible y es gratuito. Web: <https://www.sqlite.org>
 - TableFilter: Son un conjunto de componentes JAVA gratuitos que permiten crear filtros para gestionar la información mostrada en tablas. Web: <http://coderazzi.net/tablefilter>
 - Mxparser: Es una librería gratuita que implementa un *parser* de expresiones matemáticas. Web: <http://mathparser.org>
 - JFreeChart: Es una librería gratuita que permite crear gráficos en aplicaciones JAVA, incluyendo los más habituales de barras y de líneas. Web: <http://www.jfree.org/jfreechart>
- El componente PARCSIM-RUN: Es el núcleo de ejecución del simulador y está desarrollado en C y FORTRAN. Los cálculos matriciales se delegan en las siguientes librerías:
- MKL (Math Kernel Library): Librería desarrollada por Intel© optimizada para su familia de microprocesadores.
 - PARDISO (Parallel Direct Sparse Solver): Forma parte de la familia MKL de Intel© y está optimizado para ofrecer un rendimiento mejorado en la resolución de grandes sistemas de ecuaciones lineales dispersos. Con el simulador se distribuyen los paquetes de distribución libre que ofrece el fabricante para MKL y PARDISO.
 - MAGMA (*Matrix Algebra on GPU and Multicore Architectures*): Biblioteca de álgebra lineal densa similar a LAPACK para arquitecturas heterogéneas / híbridas. Se puede usar gratuitamente bajo el copyright de la Universidad de Tennessee. Se usa en el simulador para explotar las GPUs instaladas en el hardware.
 - HSL (*Harwell Subroutine Library*): Librería desarrollada en el Laboratorio STFC Rutherford Appleton. Entre sus rutinas más conocidas se encuentran las relacionadas con la resolución de sistemas de ecuaciones lineales dispersos, algunas de las cuales se han incorporado al simulador con propósitos académicos y de investigación. Para uso comercial es necesario solicitar licencias al fabricante.

A.11 Datos de contacto

José Carlos Cano Lorente
jcarlos.canol@um.es

Javier Cuenca Muñoz
Department of Engineering and Technology of Computers
Computer Science Faculty
University of Murcia
Campus de Espinardo, 30100, Murcia, Spain
jcuenca@um.es

Domingo Giménez Cánovas
retired from the Department of Computing and Systems
Computer Science Faculty
University of Murcia
Campus de Espinardo, 30100, Murcia, Spain
domingo@um.es

Mariano Saura Sánchez
Department of Mechanical Engineering, Materials and Manufacturing
Technical University of Cartagena
Doctor Fleming, s/n, 30202 Cartagena, Spain
msaura.sanchez@upct.es

Scientific Computing and Parallel Programming Group
University of Murcia
<http://www.um.es/pcgum>

Bibliografía

- [1] ABB. RobotStudio: Programming Tool for Robotics. <https://new.abb.com/products/robotics/robotstudio>. Online; accedido 17-04-2020.
- [2] ABB Robotics. RAPID Instructions, Functions and Data types. https://library.e.abb.com/public/688894b98123f87bc1257cc50044e809/Technical%20reference%20manual_RAPID_3HAC16581-1_revJ_en.pdf/. Online; accedido 26-04-2020.
- [3] Adaptive Computing. TORQUE Resource Manager. <https://adaptivecomputing.com/cherry-services/torque-resource-manager/>. Online; accedido 17-05-2020.
- [4] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better - a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*, volume 2. Morgan Kaufmann, 2010.
- [5] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):012037, 08 2009.
- [6] Shameem Akhter and Jason Roberts. *Multi-Core Programming: Increasing Performance through Software Multi-threading*. Intel Press, 2006.
- [7] Francisco Almeida, Domingo Giménez, José Miguel Mantas, and Antonio Vidal. *Introducción a la Programación Paralela*. Paraninfo, 2008.

- [8] Pedro Alonso, Ravi Reddy, and Alexey Lastovetsky. Experimental Study of Six Different Implementations of Parallel Matrix Multiplication on Heterogeneous Computational Clusters of Multicore Processors. In *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 263–270, 2010.
- [9] P. Alpatov, G. Baker, H. Edwards, John Gunnels, G. Morrow, James Overfelt, and Robert van de Geijn. PLAPACK Parallel Linear Algebra Package Design Overview. In *Conference: Supercomputing, ACM/IEEE 1997*, pages 29– 29, 12 1997.
- [10] Alejandro Álvarez-Melcón, Domingo Giménez, Fernando-Daniel Quesada-Pereira, and Tomás Ramírez. Hybrid-Parallel Algorithms for 2D Green’s Functions. In *Procedia Computer Science*, volume 18, pages 541–550, 2013.
- [11] AMD. AMD Fusion. <http://developer.amd.com/wordpress/media/2012/10/apu101.pdf>. Online; accedido 18-04-2020.
- [12] AMD. AMD Athlon™64 × 2 Technical Specifications. <http://support.amd.com/TechDocs/33425.pdf>, 2007. Online; accedido 18-04-2020.
- [13] Nakul Manchanda; Karan Anand. Non-Uniform Memory Access (NUMA). <https://web.archive.org/web/20131228092942/http://www.cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf>. Online; accedido 02-05-2020.
- [14] E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users’ Guide (Third Edition)*. Society for Industrial and Applied Mathematics, 1999.
- [15] K.S. Anderson and S. Duan. A hybrid parallelizable low-order algorithm for dynamics of multi-rigid-body systems: Part I, chain systems. *Mathematical and Computer Modelling*, 30(9):193 – 215, 1999.
- [16] Ansys. Ansys High Performance Computing. <https://www.ansys.com/products/platform/ansys-high-performance-computing>. Online; accedido 09-05-2020.

- [17] Ansys. Ansys: Rapid 3D Design Exploration. <https://www.ansys.com/products/3d-design>. Online; accedido 09-05-2020.
- [18] AnyCode Co. Marilou Robotics Studio. <http://www.anycode.com/index.php>. Online; accedido 17-04-2020.
- [19] Apache Software Foundation. Apache License, Version 2.0. <https://www.apache.org/licenses/LICENSE-2.0/>. Online; accedido 26-04-2020.
- [20] Juan Aparicio, Jose-Juan López-Espín, Raúl Martínez-Moreno, and Jesús T. Pastor. Benchmarking in Data Envelopment Analysis: An Approach Based on Genetic Algorithms and Parallel Programming. *Advances in Operations Research*, 2014:9, 02 2014.
- [21] K. Asanovic, R. Bodik, B. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, and S. W. Williams. The landscape of parallel computing research: A view from Berkeley. Technical report, UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
- [22] M. Asch, Terry Moore, Rosa M. Badia, Micah Beck, Peter H. Beckman, T. Bidot, François Bodin, Franck Cappello, Alok N. Choudhary, Bronis R. de Supinski, Ewa Deelman, Jack J. Dongarra, Anshu Dubey, Geoffrey C. Fox, H. Fu, Sergi Girona, William Gropp, Michael A. Heroux, Yutaka Ishikawa, Katarzyna Keahey, David E. Keyes, Bill Kramer, J.-F. Lavignon, Y. Lu, Satoshi Matsuoka, Bernd Mohr, Daniel A. Reed, S. Requena, Joel H. Saltz, Thomas C. Schulthess, Rick L. Stevens, D. Martin Swamy, Alexander S. Szalay, William M. Tang, G. Varoquaux, Jean-Pierre Vilotte, Robert W. Wisniewski, Z. Xu, and Igor Zacharov. Big data and extreme-scale computing. *Int. J. High Perform. Comput. Appl.*, 32(4):435–479, 2018.
- [23] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A Survey on Compiler Autotuning using Machine Learning. *ACM Computing Surveys*, 51:96:1–96:42, 09 2018.
- [24] MORSE authors. HLA-based multi-node simulation. https://www.openrobots.org/morse/doc/tutorials/hla_tutorial.html. Online; accedido 03-05-2020.

- [25] Top 500 Authors. Top 500 Supercomputer Site. <https://www.top500.org/>. Online; accedido 01-04-2020.
- [26] Jacques Bahi, Sylvain Contassot-Vivier, and Raphael Couturier. *Parallel Iterative Algorithms: From Sequential to Grid Computing*. Chapman & Hall, 2007.
- [27] David Bailey, Robert Lucas, and Samuel Williams. *Performance Tuning of Scientific Applications*. Chapman & Hall/CRC computational science, 11 2010.
- [28] Gregory Baker, John Gunnels, Greg Morrow, Béatrice Rivière, and Robert van de Geijn. PLAPACK: High Performance through High-Level Abstraction. In *ICPP '98*, pages 414–422, 01 1998.
- [29] Prasanna Balaprakash, Jack J. Dongarra, Todd Gamblin, Mary W. Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard W. Vuduc. Autotuning in high-performance computing applications. *Proc. IEEE*, 106(11):2068–2083, 2018.
- [30] Marcos Barreto, Murilo Boratto, Pedro Alonso, and Domingo Giménez. Automatic routine tuning to represent landform attributes on multicore and multi-GPU systems. *The Journal of Supercomputing*, 70(2):733–745, 03 2014.
- [31] Tal Ben-Nun and Torsten Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *ACM Comput. Surv.*, 52(4), August 2019.
- [32] Gregorio Bernabé, Raúl Hernández, and Manuel E. Acacio. Parallel implementations of the 3D fast wavelet transform on a raspberry pi 2 cluster. *J. Supercomput.*, 74(4):1765–1778, 2018.
- [33] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: A portable high-performance ANSI C coding methodology. *Proceedings of the International Conference on Supercomputing*, pages 253–260, 07 1997.

-
- [34] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, 2004.
- [35] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User’s Guide*. Society for Industrial and Applied Mathematics, 1997.
- [36] BLAS Legacy Website. Basic Linear Algebra Communication Subprograms. <http://www.netlib.org/blacs/>. Online; accedido 19-04-2020.
- [37] BLAS Legacy Website. BLAS (Basic Linear Algebra Subprograms). <http://www.netlib.org/blas/>. Online; accedido 19-04-2020.
- [38] BLAS Legacy Website. LAPACK: Linear Algebra PACKage. <http://www.netlib.org/lapack/>. Online; accedido 19-04-2020.
- [39] BLAS Legacy Website. LINPACK Fortran Subroutines. <https://www.netlib.org/linpack/>. Online; accedido 19-04-2020.
- [40] BLAS Legacy Website. PBLAS: Parallel Basic Linear Algebra Subprograms. http://www.netlib.org/scalapack/pblas_qref.html. Online; accedido 19-04-2020.
- [41] Blender Foundation. Blender: Free and Open Source 3D Creation Suite. <https://www.blender.org/>. Online; accedido 13-04-2020.
- [42] Clifford S. Bonaventura. *A Modular Approach to the Dynamics of Complex Robot Systems*. PhD thesis, The Pennsylvania State University, 2002.
- [43] Murilo Boratto, Pedro Alonso, Domingo Giménez, and Alexey Lastovetsky. Automatic tuning to performance modelling of matrix polynomials on multicore and multi-GPU systems. *The Journal of Supercomputing*, 73(1):227–239, 01 2017.
- [44] Eric A. Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1994.

- [45] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
- [46] Bullet Robotic Team. Bullet Real-Time Physics Simulation. <http://bulletphysics.org/wordpress>. Online; accedido 13-04-2020.
- [47] Phillips C. and R. Harbor. *Feedback Control Systems. Third Edition*. Prentice Hall, New Jersey, 1996.
- [48] Jesús Cámara, Javier Cuenca, and Domingo Giménez. Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters. *The Journal of Supercomputing*, 76:9922–9941, 12 2020.
- [49] Jesús Cámara Moreno. *Técnicas de Optimización de Rutinas Paralelas de Álgebra Lineal en Sistemas Heterogéneos*. PhD thesis, Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, 2020.
- [50] José-Carlos Cano. Optimización de algoritmos paralelos para análisis cinemático de sistemas multicuerpo basado en ecuaciones de grupo. Master’s thesis, Universidad de Murcia, 2017.
- [51] Adrián Castelló, Rafael Mayo, Judit Planas, and Enrique S. Quintana-Ortí. Exploiting Task-Parallelism on GPU Clusters via OmpSs and rCUDA Virtualization. In *IEEE Trustcom/BigDataSE/ISPA*, volume 3, pages 160–165, 2015.
- [52] A. I. Celdrán, D. Dopico, and M. Saura. Análisis computacional de la estructura cinemática de sistemas multicuerpo espaciales. In *XXI Congreso Nacional de Ingeniería Mecánica*, 2016.
- [53] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kauffman, 2001.

- [54] Guillaume Chapuis, Mathilde Le Boudic-Jamin, Rumen Andonov, Hristo Djidjev, and Dominique Lavenier. Parallel seed-based approach to multiple protein structure similarities detection. *Sci. Program.*, 2015:279715:1–279715:12, 2015.
- [55] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, pages 111–122, 2005.
- [56] Chun Chen. *Model-Guided Empirical Optimization for Memory Hierarchy*. PhD thesis, University of Southern California, 2007.
- [57] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and R. Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. Technical Report CS-95-292, Computer Science Dept. University of Tennessee, 05 1995.
- [58] I. . Chung and J. K. Hollingsworth. A case study using automatic performance tuning for large-scale scientific programs. In *2006 15th IEEE International Conference on High Performance Distributed Computing*, pages 45–56, 2006.
- [59] CM Labs. Newton Dynamics, a cross-platform life-like physics simulation library. <https://www.cm-labs.com/vortex-studio/>. Online; accedido 17-04-2020.
- [60] Computational Mathematics Group at the STFC Rutherford Appleton Laboratory. HSL (2013). A collection of Fortran codes for large scale scientific computation. <http://www.hsl.rl.ac.uk/>. Online; accedido 16-05-2020.
- [61] Computational Mathematics Group at the STFC Rutherford Appleton Laboratory. MA27: Solve sparse symmetric system. <http://www.hsl.rl.ac.uk/archive/specs/ma27.pdf>. Online; accedido 16-05-2020.
- [62] Computational Mathematics Group at the STFC Rutherford Appleton Laboratory. MA48: Sparse unsymmetric system: driver for conventional direct

- method. <http://www.hsl.rl.ac.uk/catalogue/ma48.html>. Online; accedido 16-05-2020.
- [63] Computational Mathematics Group at the STFC Rutherford Appleton Laboratory. MA86 Sparse symmetric indefinite system using OpenMP. http://www.hsl.rl.ac.uk/catalogue/hsl_ma86.html. Online; accedido 16-05-2020.
- [64] Keith Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 08 2002.
- [65] Coppelia Robotics. CoppeliaSim Remote API. <https://www.coppeliarobotics.com/helpFiles/en/remoteApiOverview.htm>. Online; accedido 20-04-2020.
- [66] Coppelia Robotics. V-REP, Virtual Robot Experimentation Platform. <https://www.coppeliarobotics.com/>. Online; accedido 17-04-2020.
- [67] J. H. Crichtley and K. S. Anderson. A Parallel Logarithmic Order Algorithm for General Multibody System Dynamics. *Multibody System Dynamics*, 12:75–93, 08 2004.
- [68] J. Cuadrado. Análisis de Mecanismos por Ordenador. Capítulo 1. <http://lim.ii.udc.es/docencia/phd-meccomp/capitulo1.pdf>. Online; accedido 01-05-2020.
- [69] J. Cuadrado, J. Cardenal, P. Morer, and E. Bayo. Intelligent Simulation of Multibody Dynamics: Space-State and Descriptor Methods in Sequential and Parallel Computing Environments. *Multibody System Dynamics*, 4:55–73, 02 2000.
- [70] J. Cuenca, L. P. García, and D. Gimenez. On optimization techniques for the matrix multiplication on hybrid CPU+GPU platforms. *Annals of MMulticore and GPU Programming*, 1:1–8, 2014.
- [71] Javier Cuenca, José-Matías Cutillas-Lozano, Domingo Giménez, Alberto Pérez-Bernabeu, and José J. López-Espín. Exploiting heterogeneous para-

- lism on hybrid metaheuristics for vector autoregression models. *Electronics*, 9(11), 2020.
- [72] Javier Cuenca, Luis García, Domingo Giménez, and Jack Dongarra. Processes Distribution of Homogeneous Parallel Linear Algebra Routines on Heterogeneous Clusters. In *IEEE International Conference on Cluster Computing*, pages 1–10, 2005.
- [73] Javier Cuenca, Luis-Pedro García, and Domingo Giménez. A proposal for autotuning linear algebra routines on multicore platforms. *Procedia Computer Science*, 1:515–523, 05 2010.
- [74] Javier Cuenca, Luis-Pedro García, and Domingo Giménez. Improving Linear Algebra Computation on NUMA Platforms through Auto-tuned Nested Parallelism. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 66–73, 2012.
- [75] Javier Cuenca, Luis-Pedro García, Domingo Giménez, José González, and Antonio Vidal. Empirical Modelling of Parallel Linear Algebra Routines. In *5th International Conference on Parallel Processing and Applied Mathematics*, volume 3019, pages 169–174, 2003.
- [76] Javier Cuenca, Luis-Pedro García, Domingo Giménez, and Francisco-José Herrera. Guided installation of basic linear algebra routines in a cluster with manycore components. *Concurrency and Computation: Practice and Experience*, 29(15), 2017.
- [77] Javier Cuenca, Luis-Pedro García, Domingo Giménez, and Manuel Quesada-Martínez. Analysis of the Influence of the Compiler on Multi-core Performance. In *18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP*, pages 170–174, 2010.
- [78] Javier Cuenca, Domingo Giménez, and José González. Modeling the behaviour of linear algebra algorithms with message-passing. In *9th Euromicro Workshop on Parallel and Distributed Processing*, pages 282–289, 2001.
- [79] Javier Cuenca, Domingo Giménez, and José González. Architecture of an Automatically Tuned Linear Algebra Library. *Parallel Computing*, 30(2):187–210, 02 2004.

- [80] Javier Cuenca, Domingo Giménez, José González, Jack Dongarra, and Kenneth Roche. Automatic Optimisation of Parallel Linear Algebra Routines in Systems with Variable Load. In *11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 409–416, 2003.
- [81] José-Matías Cutillas-Lozano and Domingo Giménez. Optimizing a parameterized message-passing metaheuristic scheme on a heterogeneous cluster. *Soft Computing*, 21(19):5557–5572, 09 2016.
- [82] Cyberbotics. EASY-ROB – Smart Software for Smart Robotics. <https://easy-rob.com/en/start-2/>. Online; accedido 17-04-2020.
- [83] Cyberbotics. Webots OPEN SOURCE ROBOT SIMULATOR. <http://www.cyberbotics.com/>. Online; accedido 17-04-2020.
- [84] Dassault Systemes. ABAQUS UNIFIED FEA. <https://www.3ds.com/products-services/simulia/products/abaqus/>. Online; accedido 24-01-2021.
- [85] Dassault Systemes. SOLIDWORKS 2020: Empowering Design Innovation. <https://www.solidworks.com/>. Online; accedido 09-05-2020.
- [86] Dassault Systemes. SOLIDWORKS Simulation Solutions. <https://www.solidworks.com/category/simulation-solutions>. Online; accedido 09-05-2020.
- [87] NVIDIA Developers. *NVIDIA CUDA C Programming Guide 8.0*. NVIDIA Corporation, 2017.
- [88] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience*, 28(17):4385–4404, 12 2016.
- [89] Murilo do Carmo Boratto. *Modelos Paralelos para la Resolución de Problemas de Ingeniería Agrícola*. PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2015.
- [90] Jack Dongarra. The Future of the BLAS. In *PPSC*, 1999.

- [91] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers, 2003.
- [92] Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov. HPC Programming on Intel Many-Integrated-Core Hardware with MAGMA Port to Xeon Phi. In *10th International Conference on Parallel Processing and Applied Mathematics, PPAM*, pages 571–581, 2013.
- [93] Jack Dongarra and Piotr Luszczek. *ScaLAPACK*, pages 1773–1775. Springer US, Boston, MA, 2011.
- [94] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 03 1988.
- [95] Jack J. Dongarra and R. Clint Whaley. A User’s Guide to the BLACS v1.0. Technical Report CS-95-292, Dept. of Computer Sciences, University of Tennessee, Knoxville, TN 37996, 05 1997.
- [96] John Drake, Philip Jones, Mariana Vertenstein, James White, and Patrick Worley. *Software Design for Petascale Climate Science*, pages 125–146. Chapman and Hall/CRC, 01 2008.
- [97] José Duato, Antonio J Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *International Conference on High Performance Computing and Simulation, HPCS*, pages 224–231, 2010.
- [98] Nicolas Dupin and El-Ghazali Talbi. Parallel matheuristics for the discrete unit commitment problem with min-stop ramping constraints. *Int. Trans. Oper. Res.*, 27(1):219–244, 2020.
- [99] EM Photonics and NVIDIA. CULA GPU-accelerated Linear Algebra Libraries. <http://www.culatools.com/>. Online; accedido 19-04-2020.
- [100] Energid. Actin SDK. <https://www.energid.com/actin/simulation-capabilities>. Online; accedido 17-04-2020.

- [101] Stéphane Eranian. The perfmon2 interface specification. In *7th Linux Symposium*, 2005.
- [102] Mokhtar Essaid, Lhassane Idoumghar, Julien Lepagnot, and Mathieu Brévilliers. GPU parallelization strategies for metaheuristics: a survey. *International Journal of Parallel, Emergent and Distributed Systems*, pages 1–26, 01 2018.
- [103] Constantinos Evangelinos, Robert Walkup, Vipin Sachdeva, Kirk E. Jordan, Hormozd Gahvari, I-Hsin Chung, Michael P. Perrone, Ligang Lu, Lurng-Kuo Liu, and Karen A. Magerlein. Determination of performance characteristics of scientific applications on IBM blue gene/q. *IBM J. Res. Dev.*, 57(1/2):9, 2013.
- [104] F. Cugnon, A. Cardona, A. Selvi, C. Paleczny, M. Pucheta. Synthesis and optimization of flexible mechanisms. In Carlo L. Bottasso, editor, *Multi-body Dynamics: Computational Methods and Applications*, pages 81–93, ECCOMAS, 2009. Springer.
- [105] Jianbin Fang, Ana Lucia Varbanescu, Baldomero Imbernón, José-María Cecilia, and Horacio Emilio Pérez Sánchez. Parallel Computation of Non-Bonded Interactions in Drug Discovery: Nvidia GPUs vs. Intel Xeon Phi. In *2nd International Conference on Bioinformatics and Biomedical Engineering*, pages 579–588, 2014.
- [106] Massimiliano Fatica. Accelerating Linpack with CUDA on heterogenous clusters. In *2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 46–51, 2009.
- [107] R. Featherstone and D. Orin. Robot dynamics: equations and algorithms. In *2000 IEEE Int. Conf. on Robotics and Automation*, volume 1, pages 826–834, 2000.
- [108] Roy Featherstone. A Divide-and-Conquer Articulated-Body Algorithm for Parallel $O(\log(n))$ Calculation of Rigid-Body Dynamics. Part 1: Basic Algorithm. *The International Journal of Robotics Research*, 18(9):867–875, 1999.

-
- [109] Len Freeman and Chris Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.
- [110] Marc Freese, Surya Singh, Fumio Ozaki, and Nobuto Matsuhira. Virtual Robot Experimentation Platform V-REP: A Versatile 3D Robot Simulator. In *Proceedings of the Second International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, page 51–62. Springer-Verlag, 2010.
- [111] Matteo Frigo. A fast fourier transform compiler. *SIGPLAN Not.*, 34(5):169–180, 05 1999.
- [112] Matteo Frigo and Steven Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, 1998.
- [113] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.
- [114] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson, Christopher K. I. Williams, and Michael F. P. O’Boyle. MILEPOST GCC: Machine Learning Enabled Self-tuning Compiler. *Int. J. Parallel Program.*, 39(3):296–327, 2011.
- [115] Kyle Gallivan, William Jalby, Allen Malony, and Harry Wijshoff. Performance Prediction for Parallel Numerical Algorithms. *International Journal of High Speed Computing*, 1(3):31–62, 03 1991.
- [116] Luis-Pedro García, Javier Cuenca, and Domingo Giménez. Auto-Optimization of Linear Algebra Parallel Routines: The Cholesky Factorization. In *Current & Future Issues of High-End Computing, Proceedings of the International Conference on Parallel Computing*, pages 229–236, 2005.
- [117] Luis-Pedro García, Javier Cuenca, and Domingo Giménez. Including Improvement of the Execution Time in a Software Architecture of Libraries

- With Self-Optimisation. In *2nd International Conference on Software and Data Technologies, Proceedings*, pages 156–161, 2007.
- [118] Luis-Pedro García, Javier Cuenca, and Domingo Giménez. Using Experimental Data to Improve the Performance Modelling of Parallel Linear Algebra Routines. In *7th International Conference on Parallel Processing and Applied Mathematics*, page 1150–1159, 2007.
- [119] Luis-Pedro García, Javier Cuenca, Francisco-José Herrera, and Domingo Giménez. On Guided Installation of Basic Linear Algebra Routines in Nodes with Manycore Components. In *7th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 114–122, 2016.
- [120] Javier García de Jalón and Eduardo Bayo. *Kinematic and Dynamic Simulation of Multibody Systems: The Real Time Challenge*. Springer-Verlag, New York (USA), 1994.
- [121] Gazebo Team. Gazebo Parallel Physics. <http://gazebosim.org/tutorials?tut=parallel&cat=physics>. Online; accedido 20-04-2020.
- [122] Gazebo Team. Gazebo: Rotot Simulation Made Easy. <http://gazebosim.org/>. Online; accedido 13-04-2020.
- [123] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 User’s Guide and Reference Manual. Technical Report ORNL/TM-12187, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, 11 1995.
- [124] Georgia Tech and Carnegie Mellon University. Dynamic Animation and Robotics Toolkit. <https://dartsim.github.io/>. Online; accedido 13-04-2020.
- [125] Domingo Giménez. Revisiting Strassen’s Matrix Multiplication for Multi-core Systems. *Annals of Multicore and GPU Programming*, 4(1):1–8, 2017.
- [126] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34:261–317, 06 2006.

- [127] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (Fourth Edition)*. The John Hopkins University Press, 2013.
- [128] Óscar Gómez, José-Juan López-Espín, and Antonio Peñalver Benavent. Parallel algorithm for prediction of variables in simultaneous equation models. In *17th International Conference on High Performance Computing & Simulation, HPCS 2019, Dublin, Ireland, July 15-19, 2019*, pages 1032–1035. IEEE, 2019.
- [129] Martín González, Jose-Juan López-Espín, Juan Aparicio, and Domingo Giménez. A parallel application of matheuristics in data envelopment analysis. In Fernando de la Prieta, Sigeru Omatu, and Antonio Fernández-Caballero, editors, *Distributed Computing and Artificial Intelligence, 15th International Conference, DCAI 2018, Toledo, Spain, 20-22 June 2018*, volume 800 of *Advances in Intelligent Systems and Computing*, pages 172–179. Springer, 2018.
- [130] F. González, A. Luaces, D. Dopico, and M. Gonzalez. Parallel Linear Equation Solvers and OpenMP in the Context of Multibody System Dynamics. In *Proceedings of the ASME 2009 International Design Engineering Technical Conferences*, volume 4, pages 61–71, 08 2009.
- [131] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (Second Edition)*. Addison-Wesley, 2003.
- [132] GWT-TUD GmbH. Vampir - Performance Optimization. <https://vampir.eu/>. Online; accedido 23-05-2020.
- [133] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–11, 2009.
- [134] Jianyu Huang, Tyler Smith, Greg Henry, and Robert van de Geijn. Strassen’s Algorithm Reloaded. In *SC’16 - International Conference for High Performance Computing*, pages 690–701, 2016.
- [135] Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn. Implementing Strassen’s Algorithm with CUTLASS on NVIDIA Volta GPUs. *ArXiv*, abs/1808.07984, 08 2018.

- [136] K. A. Huck and A. D. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 41–41, 2005.
- [137] Cameron Hughes and Tracey Hughes. *Professional Multicore Programming: Design and Implementation for C++ Developers*. Wiley Publishing, Inc., 2008.
- [138] Sascha Hunold and Thomas Rauber. Automatic tuning of pdgemm towards optimal performance. In *Euro-Par 2005 Parallel Processing*, pages 837–846, 2005.
- [139] I-Hsin Chung and J. K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 30, 2004.
- [140] I-Hsin Chung and J. K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 30–30, 2004.
- [141] IBM. The Most Powerful Computers on the Planet. <https://www.ibm.com/thought-leadership/summit-supercomputer/>. Online; accedido 22-05-2020.
- [142] ICL Team. Chameleon: A Dense Linear Algebra Software for Heterogeneous Architectures. <https://gitlab.inria.fr/solverstack/chameleon>. Online; accedido 20-03-2020.
- [143] ICL team. Performance Application Programming Interface. <https://icl.utk.edu/papi/>. Online; accedido 23-05-2020.
- [144] Baldomero Imbernón, Antonio Llanes, José-Matías Cutillas-Lozano, and Domingo Giménez. HYPERDOCK: improving virtual screening through parallel hyperheuristics. *Int. J. High Perform. Comput. Appl.*, 34(1), 2020.
- [145] Baldomero Imbernón, Javier Prades, Domingo Giménez, José Cecilia, and Federico Silla. Enhancing large-scale docking simulation on heterogeneous systems: An MPI vs rCUDA study. *Future Generation Computer Systems*, 79:26–37, 02 2018.

- [146] Innovative Computing Laboratory (ICL) . Matrix Algebra on GPU and Multicore Architectures. <http://icl.cs.utk.edu/magma/>. Online; accedido 19-04-2020.
- [147] Innovative Computing Laboratory (ICL) . PLASMA: Parallel Linear Algebra Software for Multicore Architectures. <http://icl.cs.utk.edu/plasma/software/>. Online; accedido 19-04-2020.
- [148] Intel Corporation. Intel C++ Compiler. <https://software.intel.com/content/www/us/en/develop/tools/compilers/c-compilers.html>. Online; accedido 15-05-2020.
- [149] Intel Corporation. Intel Fortran Compiler. <https://software.intel.com/content/www/us/en/develop/tools/compilers/fortran-compilers.html>. Online; accedido 15-05-2020.
- [150] Intel Corporation. Intel Math Kernel Library. <https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html>. Online; accedido 19-04-2020.
- [151] Intel Corporation. Intel MKL PARDISO - Parallel Direct Sparse Solver Interface. <https://software.intel.com/en-us/node/470282>. Online; accedido 16-05-2020.
- [152] Intel©. Atom Processors Family. <https://www.intel.com/content/www/us/en/products/processors/atom.html>. Online; accedido 18-04-2020.
- [153] Intel©. Hyper-Threading Technology. <https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html>. Online; accedido 18-04-2020.
- [154] Intel©. Intel©Many Integrated Core Architecture (Intel©MIC Architecture). <https://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>. Online; accedido 18-04-2020.
- [155] Intel©. Intel© Thread Affinity Control. <https://software.intel.com/en-us/articles/openmp-thread-affinity-control>. Online; accedido 18-04-2020.

- [156] Intel©. Intel©Xeon Phi Processors. <https://ark.intel.com/content/www/us/en/ark.html#@PanelLabel75557>. Online; accedido 18-04-2020.
- [157] Intel©. Legacy Intel© Core™ 2 Processor. <https://ark.intel.com/content/www/us/en/ark/products/series/79666/legacy-intel-core-processors.html>, 2017. Online; accedido 18-04-2020.
- [158] International Organization for Standardization. ISO/IEC 1539-1:2018 - Programming languages — FORTRAN. <https://www.iso.org/standard/72320.html>. Online; accedido 15-05-2020.
- [159] International Organization for Standardization. ISO/IEC 9899:2018 - Programming languages — C. <https://www.iso.org/standard/74528.html>. Online; accedido 15-05-2020.
- [160] Muhammad Ali Ismail, S. Mirza, Talat Altaf, Dr. Mirza, and Dr. Altaf. Concurrent Matrix Multiplication on Multi-Core Processors. *International Journal of Computer Science and Security*, 5:208–220, 05 2011.
- [161] Jack Dongarra. List of Freely Available Software for Linear Algebra on the Web. <http://www.netlib.org/utk/people/JackDongarra/la-sw.html>. Online; accedido 20-03-2020.
- [162] Jaeyoung Choi and J. J. Dongarra. Scalable linear algebra software libraries for distributed memory concurrent computers. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 170–177, 1995.
- [163] John W. Eaton. GNU Octave: Scientific Programming Language. <https://www.gnu.org/software/octave/>. Online; accedido 26-04-2020.
- [164] Julio Jerez and Alain Suero. Newton Dynamics, a Cross-platform Life-like Physics Simulation Library. <http://newtondynamics.com/forum/newton.php/>. Online; accedido 17-04-2020.
- [165] Alexey Kalinov and Alexey Lastovetsky. Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers. *Journal of Parallel and Distributed Computing*, 61(4):520–535, 04 2001.

-
- [166] Elaye Karstadt and Oded Schwartz. Matrix Multiplication, a Little Faster. In *29th ACM Symposium on Parallelism in Algorithms and Architectures*, page 101–110, 2017.
- [167] Takahiro Katagiri. Abclibscript: A computer language for automatic performance tuning. In Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda, editors, *Software Automatic Tuning, From Concepts to State-of-the-Art Results*, pages 295–313. Springer, 2010.
- [168] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. Effect of auto-tuning with user’s knowledge for numerical software. In *1st Conference on Computing Frontiers*, pages 12–25, 2004.
- [169] Stephen Keckler, William Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 11 2011.
- [170] Khronos Group. The Industry’s Foundation for High Performance Graphics. <https://www.khronos.org/>. Online; accedido 18-04-2020.
- [171] Khronos Group. The Open Standard for Parallel Programming of Heterogeneous Systems. <https://www.khronos.org/opencl/>. Online; accedido 18-04-2020.
- [172] Ayca Kirimtat, Ondrej Krejcar, Rafael Dolezal, and Ali Selamat. A mini review on parallel processing of brain magnetic resonance imaging. In Ignacio Rojas, Olga Valenzuela, Fernando Rojas, Luis Javier Herrera, and Francisco M. Ortuño Guzmán, editors, *Bioinformatics and Biomedical Engineering - 8th International Work-Conference, IWBBIO 2020, Granada, Spain, May 6-8, 2020, Proceedings*, volume 12108 of *Lecture Notes in Computer Science*, pages 482–493. Springer, 2020.
- [173] David Kirk and Hwu Wen-Mei. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2016.
- [174] Tobias Klein, Peter Mindek, Ludovic Autin, David S. Goodsell, Arthur J. Olson, M. Eduard Gröller, and Ivan Viola. Parallel generation and visualization of bacterial genome structures. *Comput. Graph. Forum*, 38(7):57–68, 2019.

- [175] Tamara G. Kolda, Robert Michael Lewis, and Virginia Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45:385–482, 2003.
- [176] M. Z. Kolovsky, A. N. Evgrafov, Y. A. Semenov, and A. V. Slousch. *Advanced theory of mechanism and machines*. Springer, 2000.
- [177] Branko Kolundzija, Dragan Olcan, Dusan Zoric, and Sladjana Maric. Accelerating WIPL-D numerical EM kernel by using graphics processing units. In *10th International Conference on Telecommunication in Modern Satellite Cable and Broadcasting Services, TELSIKS*, volume 2, pages 413–419, 2011.
- [178] Alice Koniges. *Industrial Strength Parallel Computing*. Morgan Kaufmann Publishers, 2000.
- [179] Erricos Kontoghiorghes. *Parallel Algorithms for Linear Models: Numerical Methods and Estimation Problems*. Kluwer Academic Publishers, 2000.
- [180] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, pages 12—23, 2003.
- [181] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. Autotuning GEMM kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 01 2012.
- [182] Alexey Lastovetsky and Ravi Reddy. A Novel Algorithm of Optimal Matrix Partitioning for Parallel Dense Factorization on Heterogeneous Processors. In *Lecture Notes in Computer Science*, volume 4671, pages 261–275, 2007.
- [183] François Le Gall. Powers of Tensors and Fast Matrix Multiplication. In *International Symposium on Symbolic and Algebraic Computation, ISSAC*, pages 296–303, 2014.
- [184] Legacy Website. ScaLAPACK - Scalable Linear Algebra PACKage. <http://www.netlib.org/scalapack/>. Online; accedido 19-04-2020.

- [185] Linux Foundation. LINUX. <https://www.linuxfoundation.org/>. Online; accedido 15-05-2020.
- [186] Francisco López-Castejón and Domingo Giménez. Auto-optimization on parallel hydrodynamic codes: an example of COHERENS with OpenMP for multicore. In *XVIII International Conference on Water Resources*, 2010.
- [187] LUA Community. The Programming Language LUA. <https://www.lua.org/>. Online; accedido 26-04-2020.
- [188] Mark Lundstrom. Moore’s Law Forever? *Science*, 299:210–211, 2003.
- [189] K. Magnus. *Dynamics of multibody systems*. Springer Verlag, 1978.
- [190] Subhendu Maity, Subbareddy Bonthu, Kaushik Sasmal, and Hari Warrior. Role of parallel computing in numerical weather forecasting models. *International Journal of Computer Applications*, CCSN2012(4):975–8887, 03 2013.
- [191] P. Malczyk, J. Fraczek, F. González, and J. Cuadrado. Index-3 divide-and-conquer algorithm for efficient multibody system dynamics simulations: theory and parallel implementation. *Nonlinear Dynamics*, 95:727–747, 2018.
- [192] MapleSoft. Maple: Essential Tool for Mathematics. <https://www.maplesoft.com/products/maple/>. Online; accedido 17-04-2020.
- [193] MapleSoft. Maplesim: Advanced System-level Modeling. <http://www.maplesoft.com/products/maplesim/>. Online; accedido 17-04-2020.
- [194] MapleSoft. Multithreaded Programming. <https://www.maplesoft.com/support/help/Maple/view.aspx?path=multithreaded>. Online; accedido 20-04-2020.
- [195] MapleSoft. Overview of the Grid Package. <https://www.maplesoft.com/support/help/Maple/view.aspx?path=Grid>. Online; accedido 20-04-2020.
- [196] MathWorks. Language Fundamentals. <https://www.mathworks.com/help/matlab/language-fundamentals.html/>. Online; accedido 26-04-2020.

- [197] MathWorks. MATLAB. <https://es.mathworks.com/products/matlab.html>. Online; accedido 17-04-2020.
- [198] MathWorks. MATLAB Parallel Computing Toolbox. <https://es.mathworks.com/products/parallel-computing.html>. Online; accedido 20-04-2020.
- [199] MathWorks. MATLAB Parallel Server. <https://es.mathworks.com/products/matlab-parallel-server.html>. Online; accedido 20-04-2020.
- [200] MathWorks. MATLAB Robotics System Toolbox. <https://es.mathworks.com/products/robotics.html>. Online; accedido 17-04-2020.
- [201] MathWorks. MATLAB Simspace Multibody. <https://es.mathworks.com/products/simmechanics.html>. Online; accedido 17-04-2020.
- [202] MathWorks. MATLAB Simulink. <https://es.mathworks.com/products/simulink.html>. Online; accedido 17-04-2020.
- [203] Simon McIntosh-Smith, James Price, Richard Sessions, and Amaury Ibarra. High performance in silico virtual drug screening on many-core processors. *The International Journal of High Performance Computing Applications*, 29(2):119–134, 05 2015.
- [204] Everton Mendonça, Marcos Barreto, Vinícius Guimarães, Nelci Santos, Samuel Pita, and Murilo Boratto. Accelerating docking simulation using multicore and GPU systems. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Giuseppe Borruso, Carmelo Maria Torre, Ana Maria A. C. Rocha, David Taniar, Bernady O. Apduhan, Elena N. Stankova, and Alfredo Cuzzocrea, editors, *Computational Science and Its Applications - ICCSA 2017 - 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part I*, volume 10404 of *Lecture Notes in Computer Science*, pages 439–451. Springer, 2017.
- [205] Panagiotis Michailidis and Konstantinos G. Margaritis. Parallel direct methods for solving the system of linear equations with pipelining on a multicore using OpenMP. *J. Computational Applied Mathematics*, 236:326–341, 09 2011.

- [206] Microsoft. CUDA Parallel Computing Platform. <https://developer.nvidia.com/cuda-toolkit>. Online; accedido 18-04-2020.
- [207] Microsoft. DirectX graphics and gaming. <https://docs.microsoft.com/es-es/windows/win32/directx>. Online; accedido 18-04-2020.
- [208] Microsoft Corporate. Windows. <https://www.microsoft.com/eN-US/windows/>. Online; accedido 15-05-2020.
- [209] Jagdish J. Modi. *Parallel Algorithms and Matrix Computation*. Clarendon Press, 1988.
- [210] Gordon Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *Solid-State Circuits Newsletter, IEEE*, 11:33–35, 10 2006.
- [211] Gordon Moore. Progress in digital integrated electronics. *IEEE Solid-State Circuits Society Newsletter*, 11(3):36–37, 2006.
- [212] MORSE authors. MORSE, The Modular OpenRobots Simulation Engine. <https://www.openrobots.org/wiki/morse>. Online; accedido 17-04-2020.
- [213] MPI Forum. Forum for the Message Passing Interface. <https://www.mpi-forum.org/>. Online; accedido 20-03-2020.
- [214] MSC Software. Adams: The Multibody Dynamics Simulation Solution. <https://www.mscsoftware.com/product/adams>. Online; accedido 09-05-2020.
- [215] MSC Software. MSC Nastran: Multidisciplinary Structural Analysis. <https://www.mscsoftware.com/product/msc-nastran>. Online; accedido 10-05-2020.
- [216] Javier Cuenca Muñoz. *Optimización Automática de Software Paralelo de Álgebra Lineal*. PhD thesis, Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, 2004.
- [217] Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda. *Software Automatic Tuning. From Concepts to State-of-the-Art Results*. Springer, 2010.

- [218] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An Improved Magma Gemm For Fermi Graphics Processing Units. *International Journal of High Performance Computing Applications*, 24(4):511–515, 11 2010.
- [219] National Science Foundation. CCSM4.0 Public Release. <http://www.cesm.ucar.edu/models/ccsm4.0/>. Online; accedido 30-05-2020.
- [220] National Science Foundation. Community Earth System Model. <http://ccsm.ucar.edu/>. Online; accedido 30-05-2020.
- [221] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 01 1965.
- [222] Yiinju Nelson, Bhupesh Bansal, Mary Hall, Aiichiro Nakano, and Kristina Lerman. Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. In *Parallel and Distributed Processing Symposium, International*, pages 1–8, 04 2008.
- [223] Bradford Nichols, Dick Buttlar, and Jacqueline Farrel. *Pthreads Programming: A Posix Standard For Better Multiprocessing*. O’Reilly, 1996.
- [224] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 03 2008.
- [225] NVIDIA Corporation. cuBLAS Library User’s Guide v8.0. https://docs.nvidia.com/cuda/archive/8.0/pdf/CUBLAS_Library.pdf. Online; accedido 19-04-2020.
- [226] NVIDIA Corporation. CUDA Parallel computing platform. <https://developer.nvidia.com/about-cuda>. Online; accedido 19-04-2020.
- [227] NVIDIA Corporation. Dense Linear Algebra on GPUs. <https://developer.nvidia.com/cublas>. Online; accedido 19-04-2020.
- [228] NVIDIA Corporation. Using the CUBLASXT API. <https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublasXt-api>. Online; accedido 19-04-2020.
- [229] OGRE Team. OGRE: Open Source 3D Graphics Engine. <https://www.ogre3d.org/>. Online; accedido 13-04-2020.

-
- [230] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In *Lecture Notes in Computer Science, Proceedings of the 7th International Conference on High-Performance Computing for Computational Science*, volume 4395, pages 305–318, 2007.
- [231] Open MPI Project. Open MPI: Open Source High-Performance Computing. <https://www.open-mpi.org/>. Online; accedido 20-03-2020.
- [232] OpenCV Team. OpenCV. <https://opencv.org/>. Online; accedido 26-04-2020.
- [233] OpenMP Architecture Review Board. The OpenMP API Specification for Parallel Programming. <https://www.openmp.org/>. Online; accedido 20-03-2020.
- [234] OpenMP Architecture Review Board. The OpenMP API Specification v5.0. <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>. Online; accedido 20-03-2020.
- [235] Oracle. JAVA software. <https://www.oracle.com/java/>. Online; accedido 26-04-2020.
- [236] James M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, 1989.
- [237] John Owens, Mike Houston, David Luebke, Simon Green, John Stone, and James Phillips. GPU computing. *Proceedings of the IEEE*, 96:879–899, 05 2008.
- [238] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.
- [239] Parallelum. Scientific Computing and Parallel Programming Group. http://luna.inf.um.es/grupo_investigacion/. Online; accedido 20-03-2020.
- [240] Antoine Petit, L. Blackford, Jack Dongarra, Brett Ellis, Graham Fagg, Kenneth Roche, and Sathish Vadhiyar. Numerical Libraries and the Grid. *International Journal of High Performance Computing Applications*, 15(4):359–374, 01 2001.

- [241] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. *Procedia Computer Science*, 18:1312–1321, 12 2013.
- [242] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):21–45, 2004.
- [243] Python Software Foundation. PYTHON. <https://www.python.org/>. Online; accedido 26-04-2020.
- [244] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the International Conference on Supercomputing*, pages 249–258, 01 2006.
- [245] Apan Qasem and Ken Kennedy. Model-guided empirical tuning of loop fusion. *Int. J. High Perform. Syst. Archit.*, 1(3):183–198, 2008.
- [246] Yutong Qin, Jianbiao Lin, and Xiang Huang. Massively parallel ray tracing algorithm using GPU. *CoRR*, abs/1504.03151, 2015.
- [247] G. Quaranta, P. Masarati, and P. Mantegazza. Multibody Analysis of Controlled Aeroelastic Systems on Parallel Computers. *Multibody System Dynamics*, 8:71–102, 08 2002.
- [248] Michael Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, 2004.
- [249] Ari Rasch, Michael Haidl, and Sergei Gorlatch. ATF: A Generic Auto-Tuning Framework. In *2017 IEEE 19th International Conference on High Performance Computing and Communications*, pages 64–71, 12 2017.
- [250] Thomas Rauber and Gudula Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2012.
- [251] Ravi Reddy and Alexey Lastovetsky. HeteroMPI+ScaLAPACK: Towards a ScaLAPACK (dense linear solvers) on heterogeneous networks of computers. In *Lecture Notes in Computer Science, Proceedings of the International*

- Conference on High-Performance Computing*, volume 4297, pages 242–253, 2006.
- [252] Daniel Reed and Jack Dongarra. Exascale Computing and Big Data. *Communications of the ACM*, 58(7):56–68, 07 2015.
- [253] Reinders, J. An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors. https://www.cs.unc.edu/~prins/Courses/633/Readings/An-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_Reinders.pdf. Online; accedido 18-04-2020.
- [254] Rice University - Rice Computer Science. HPCToolkit: integrated suite of tools for measurement and analysis. <http://hpctoolkit.org/>. Online; accedido 23-05-2020.
- [255] RoboDK. ROBODK: Simulate Robot Applications. <https://robodk.com/>. Online; accedido 17-04-2020.
- [256] Russ Smith. ODE: Open Dynamics Engine. <http://www.ode.org/>. Online; accedido 13-04-2020.
- [257] Hocine Saadi, Nadia Nouali-Taboudjemat, Abdellatif Rahmoun, Baldomero Imbernon, Horacio Emilio Pérez Sánchez, and José M. Cecilia. Efficient gpu-based parallelization of solvation calculation for the blind docking problem. *J. Supercomput.*, 76(3):1980–1998, 2020.
- [258] Takao Sakurai, Takahiro Katagiri, Hisayasu Kuroda, Ken Naono, Mitsuyoshi Igai, and Satoshi Ohshima. A Sparse Matrix Library with Automatic Selection of Iterative Solvers and Preconditioners. *Procedia Computer Science*, 18:1332–1341, 12 2013.
- [259] Takao Sakurai, Takahiro Katagiri, Hisayasu Kuroda, Ken Naono, Mitsuyoshi Igai, and Satoshi Ohshima. A sparse matrix library with automatic selection of iterative solvers and preconditioners. In Vassil N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, volume 18 of *Procedia Computer Science*, pages 1332–1341. Elsevier, 2013.

- [260] Santiago Sánchez, Erney Ramírez-Aportela, José Garzón, Pablo Chacón, Antonio S. Montemayor, and Raúl Cabido. FRODRUG: A Virtual Screening GPU Accelerated Approach for Drug Discovery. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 594–600, 2014.
- [261] M. Saura, A. Celdrán, D. Dopico, and J. Cuadrado. Computational structural analysis of planar multibody systems with lower and higher kinematic pairs. *Mechanism and Machine Theory*, 71:79–92, 2014.
- [262] M. Saura, D. Dopico, P. Segado, and P. Martínez. Eficiencia de una formulación cinemática computacional basada en ecuaciones de grupo. In *XXI Congreso Nacional de Ingeniería Mecánica, Elche*, 2016.
- [263] M. Saura, B. Muñoz, D. Dopico, P. Segado, and J. Cuadrado. Multibody Kinematics. A Topological Formulation Based on Structural-Group Coordinates. In *ECCOMAS Thematic Conference on Multibody Dynamics, Barcelona*, June 29 - July 2, 2015.
- [264] Robert R. Schaller. Moore’s law: past, present, and future. *IEEE Spectrum*, 34(6):52–59, 06 1997.
- [265] Ridgway Scott, Terry Clark, and Babak Bagheri. *Scientific Parallel Computing*. Princeton University Press, 2005.
- [266] P. Segado, D. Dopico, and M. Saura. Formulaciones dinámicas en coordenadas naturales para la aproximación basada en ecuaciones de grupo. In *XXI Congreso Nacional de Ingeniería Mecánica*, 2016.
- [267] Jesús Pérez Serrano, E. Sandes, A. Melo, and M. Ujaldón. DNA sequences alignment in multi-GPUs: acceleration and energy payoff. *BMC Bioinformatics*, 19(421), 2018.
- [268] A. A. Shabana. *Dynamics of multibody systems*. John Wiley & Sons, 2nd edition, 1998.
- [269] Ronald Shonkwiler and Lew Lefton. *An Introduction to Parallel and Vector Scientific Computation*. Cambridge University Press, 2006.

- [270] Siemens. Scalable Simulation Software. <https://solidedge.siemens.com/en/solutions/products/simulation/solid-edge-simulation/>. Online; accedido 09-05-2020.
- [271] Siemens. Solid Edge. <https://solidedge.siemens.com/en/>. Online; accedido 09-05-2020.
- [272] Federico Silla, Javier Prades, Sergio Iserte, and Carlos Reaño. Remote GPU Virtualization: Is It Useful? In *2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), Barcelona*, pages 41–48, 2016.
- [273] SimTK team. Simbody: Multibody Physics API. <https://simtk.org/projects/simbody/>. Online; accedido 13-04-2020.
- [274] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI. The Complete Reference (Second Edition)*. The MIT Press, 1998.
- [275] Source Forge. Linux Performance Counters Driver. <https://sourceforge.net/projects/perfctr/>. Online; accedido 23-05-2020.
- [276] Source Forge. Open Source Performance Analysis Environment for Linux. <http://perfsuite.sourceforge.net/>. Online; accedido 23-05-2020.
- [277] Source Forge. perfmon2: The Hardware-based Performance Monitoring Interface for Linux. <http://perfmon2.sourceforge.net/lists.html>. Online; accedido 23-05-2020.
- [278] SQLite Consortium. SQLite. <https://www.sqlite.org/index.html>. Online; accedido 16-05-2020.
- [279] Standard C++ Foundation. Standar C++. <https://isocpp.org/>. Online; accedido 26-04-2020.
- [280] John Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(1-3):66–72, 05 2010.

- [281] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 3(14):354–356, 1969.
- [282] El-Ghazali Talbi. *Parallel Evolutionary Combinatorial Optimization*, pages 1107–1125. Springer Berlin Heidelberg, 2015.
- [283] El-Ghazali Talbi and Geir Hasle. Metaheuristics on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):1–3, 01 2013.
- [284] Technische Universitat Darmstadt. Scalasca: performance optimization of parallel programs. <https://www.scalasca.org/>. Online; accedido 23-05-2020.
- [285] T. N. Theis and H.-S. Philip Wong. The End of Moore’s Law: A New Beginning for Information Technology. *Computing in Science & Engineering*, 19(2):41–50, 03 2017.
- [286] Ruben A. Tikidji-Hamburyan, Vikram K. Narayana, Zeki Bozkus, and Tarek A. El-Ghazawi. Software for brain network simulations: A comparative study. *Frontiers Neuroinformatics*, 11:46, 2017.
- [287] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, 2009.
- [288] University of Oregon. PerfExplorer - User’s Manual. <https://www.cs.uoregon.edu/research/tau/docs/perfexplorer/>. Online; accedido 29-05-2020.
- [289] Robert van de Geijn and Kazushige Goto. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):Article 12, 05 2008.
- [290] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.
- [291] Richard Vuduc, James Demmel, and Jeff Bilmes. Statistical Models for Automatic Performance Tuning. In *International Conference on Computational Science*, volume 18, pages 117–126, 2001.

-
- [292] Richard Vuduc, James Demmel, and Katherine Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics Conference Series*, 16:521–530, 01 2005.
- [293] Feng Wang, Can-Qun Yang, Yun-Fei Du, Juan Chen, Hui-Zhan Yi, and Wei-Xia Xu. Optimizing Linpack Benchmark on GPU-Accelerated Petascale Supercomputer. *Journal of Computer Science and Technology*, 26(5):854–865, 09 2011.
- [294] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Singh, Sayantan Sur, and D.K. Panda. MVAPICH2GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science - Research and Development*, 26(3-4):257–266, 06 2011.
- [295] K. Wehage, R. Wehage, and B. Ravani. Generalized coordinate partitioning for complex mechanisms based on kinematic substructuring. *Mechanism and Machine Theory*, 92:464 – 483, 2015.
- [296] R. A. Wehage and E. J. Haug. Generalized coordinate partitioning for dimension reduction in analysis of constrained dynamic systems. *Journal of Mechanical Design*, 104(1):247–255, 1982.
- [297] R. Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 01 2001.
- [298] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (Second Edition)*. Prentice-Hall, 2005.
- [299] Felix Wolf, Brian Wylie, Erika Ábrahám, Daniel Becker, Wolfgang Frings, Karl Furlinger, Markus Geimer, Marc-André Hermanns, Bernd Mohr, Shirley Moore, Matthias Pfeifer, and Zoltán Szebenyi. Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, pages 157–167, 01 2008.

- [300] Dong Hyuk Woo, Joshua B. Fryman, Allan D. Knies, and Hsien-Hsin S. Lee. Chameleon: Virtualizing idle acceleration cores of a heterogeneous multicore processor for caching and prefetching. *ACM Trans. Archit. Code Optim.*, 7(1), 5 2010.
- [301] World Wide Web Consortium (W3C) . Scalable Vector Graphics (SVG). <https://www.w3.org/Graphics/SVG/>. Online; accedido 13-04-2020.
- [302] Patrick Worley, Arthur Mirin, Anthony Craig, Mark Taylor, John Dennis, and Mariana Vertenstein. Performance of the community earth system model. In *Proceedings of 2011 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, page 54, 11 2011.
- [303] Xingfu Wu, Valerie E. Taylor, Shane Garrick, Dazhi Yu, and Jacques Richard. Performance Analysis, Modeling and Prediction of a Parallel Multiblock Lattice Boltzmann Application Using Prophesy System. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona, Spain*, pages 1–8. IEEE Computer Society, 2006.