FACULTAD DE INFORMÁTICA

Máster Universitario en Nuevas Tecnologías de la Informática
Itinerario de Arquitecturas de Altas Prestaciones y Supercomputación

TRABAJO FIN DE MÁSTER

Optimización de algoritmos paralelos para análisis cinemático de sistemas multicuerpo basado en Ecuaciones de Grupo

Autor:
José Carlos Cano Lorente

Tutores:
Antonio Javier Cuenca Muñoz
Domingo Giménez Cánovas

Cotutor:
Mariano Saura Sánchez
Departamento de Ingeniería Mecánica
Universidad Politécnica de Cartagena

MURCIA, Julio de 2017

Resumen

Se define un sistema multicuerpo como un conjunto de cuerpos conectados entre sí mediante juntas cinemáticas o elementos de fuerza. Cuando uno de esos sistemas mecánicos entra en movimiento, los elementos contiguos tratan de desplazarse unos respecto a otros en la dirección de alguno de los grados de libertad permitidos por el tipo de junta que los une.

Modelar un sistema de este tipo requiere identificar las variables independientes que determinan las posiciones de todos los elementos que lo constituyen y las ecuaciones de restricción que relacionan dichas variables, que dependerán del tipo de formulación elegida.

En una formulación global se seleccionan tantas variables 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.

Una formulación topológica, sin embargo, se basa en la identificación y descomposición del sistema en lazos cerrados independientes, lo que implica un estudio más avanzado de su estructura. Una variante de esta formulación permite dividir el sistema en un conjunto de cadenas cinemáticas simples denominadas Grupos Estructurales, de tal manera que la resolución de la cinemática de cada uno de esos grupos por separado, y en el orden adecuado, resuelve el sistema completo. Además, cuando no hay dependencia entre grupos, éstos se pueden calcular de manera independiente sin afectar al resultado, lo que permite su resolución simultánea en sistemas de cómputo paralelo.

Un ejemplo de este tipo de sistemas multicuerpo es la denominada Plataforma de Stewart, un robot manipulador en el que se controlan los seis grados de libertad de una placa móvil mediante actuadores independientes. La cinemática de los actuadores se puede resolver empleando métodos de álgebra lineal, siendo posible realizar los cálculos en orden arbitrario y en paralelo.

Este Trabajo Fin de Máster profundiza en el estudio de las técnicas y algoritmos de computación paralela aplicables al problema del análisis cinemático de sistemas multicuerpo en general, y de sistemas de cinemática paralela, como la Plataforma de Stewart, en particular. Para ello se ha desarrollado el software de simulación necesario para la cinemática de un sistema, permitiendo variar los parámetros del paralelismo, dispersión y librerías de álgebra matricial empleadas, y decidir de forma automática cuáles son las configuraciones de software que maximizan el rendimiento de los cálculos en función del hardware instalado (*autotuning*).

Además, con objeto de generalizar el estudio, se permite variar el tamaño del problema, tanto en el número de grupos estructurales como en la composición de los mismos,

lo que a la postre implica variar el número de variables y tamaño de las matrices que representan las ecuaciones de restricción. De esta manera se han podido generar configuraciones óptimas para unos tamaños de aprendizaje y se han validado posteriormente con tamaños distintos.

El interés práctico de este trabajo es doble. Por un lado, un software optimizado permitiría el control en tiempo real de los sistemas, incluso empleando hardware con prestaciones limitadas. Por otro lado, en el análisis de nuevos sistemas multicuerpo, como parte del denominado *desarrollo virtual del producto*, donde simulaciones en potentes ordenadores durante la fase de diseño permiten anticiparse al comportamiento del sistema, reduciendo la necesidad de crear prototipos físicos y ensayos experimentales, con aplicación en sectores como son la automoción, aeroespacial, naval, robótica, etc. Es aquí donde el empleo de sistemas de cómputo masivamente paralelos y heterogéneos, combinando procesadores de memoria compartida y tarjetas gráficas dedicadas a cómputo general, muestra todo su potencial.

Índice general

Índice de figuras

Índice de algoritmos

Índice de tablas

Capítulo 1

Introducción

Este capítulo comienza ofreciendo al lector una visión general del problema a tratar y de cuáles son las motivaciones y objetivos que han guiado el desarrollo de esta Tesis de Máster.

Se realiza una introducción teórica en el campo del análisis de sistemas multicuerpo [60, 62, 73], y se presentan algunos trabajos previos que ayudarán a entender el interés en avanzar en el estudio de las técnicas de simulación de estos sistemas, con aplicaciones directas en sectores industriales y de entretenimiento.

Se describen algunas conocidas herramientas de simulación de robots existentes en la actualidad, y cómo algunas de ellas han comenzado a mostrar adaptaciones a los modernos sistemas de cómputo paralelos y heterogéneos.

Además se muestra la evolución de las librerías de álgebra matricial, componentes de software indispensables para resolver los cálculos inherentes al modelado de sistemas mecánicos. Esto dará paso a introducir algunos de los conceptos que pertenecen al ámbito de la paralelización y la optimización automática de código (*autotuning*) y que han sido aplicados en este trabajo.

El capítulo continúa describiendo la metodología seguida para el desarrollo del software que ha proporcionado los datos experimentales necesarios para elaborar este trabajo, y concluye explicando la estructura del presente documento con extractos del contenido de cada capítulo.

1.1. Planteamiento

Atrás quedaron los años en los que el avance en el rendimiento de los ordenadores descansaba sobre el simple hecho del incremento de la frecuencia del reloj y el número de transistores en los procesadores. Alcanzado el límite impuesto por la física en la escala de integración de los chips, la siguiente evolución consistió en aumentar el número de unidades de proceso, y no sólo en grandes *mainframes*, sino también en ordenadores

de consumo. Primero fueron los coprocesadores matemáticos. Después Intel© diseñó la tecnología Hyper Threading para simular dos microprocesadores lógicos dentro de uno físico. Esto permitió procesar hilos o subprocesos en paralelo en el procesador, aprovechando mejor sus recursos. Pero, sin embargo, la gran revolución fue la tecnología de doble núcleo, que permitió tener dos núcleos de ejecución en el mismo chip. Inicialmente AMD lanzó la serie Athlon X2 [45] y fue seguido por Intel© con la serie Core [59].

A partir de entonces los ingenieros han estado trabajado en incrementar el número de cores en los procesadores, permitiendo a los programadores hacer uso de todos ellos compartiendo la memoria del ordenador.

Por otro lado, el advenimiento de las tarjetas gráficas, ha puesto en escena la posibilidad de usar sus coprocesadores, fuertemente optimizados para procesamiento gráfico, dominado por operaciones sobre datos en paralelo, en particular de álgebra lineal y operaciones con matrices.

Si añadimos la posibilidad de interconectar varios multicores, compartiendo sus recursos de memoria y computación, entramos en un escenario en el que problemas científicos y de ingeniería, con elevado coste computacional, son posibles de resolver en tiempos razonables. A esta nueva evolución se le suele denominar como modelo masivamente paralelo [46].

Un ejemplo de aplicación en el ámbito de la ingeniería es el modelado de sistemas mecánicos basados en métodos numéricos. En estos modelos se manejan coordenadas, ecuaciones de restricción y cálculo de desplazamientos finitos, posiciones, velocidades y aceleraciones de todos los componentes que constituyen un sistema. En el desarrollo matemático de este tipo de problemas intervienen la geometría y el álgebra lineal.

Como ejemplo, observamos en la figura 1.1 el modelado de un sencillo mecanismo con seis variables $(x_1, y_1, x_2, y_2, x_3, y_3)$, donde se impone la necesidad de definir las 5 ecuaciones de restricción derivadas de la topología del sistema [52].

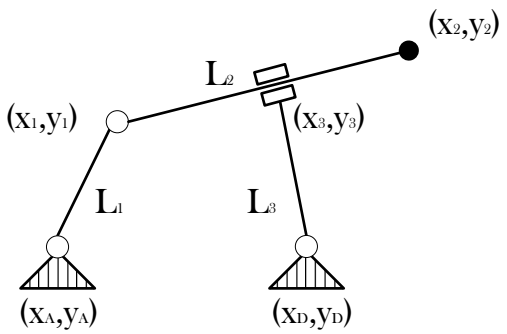


Figura 1.1: Ejemplo de sistema multicuerpo en 2 dimensiones

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

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

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 el aumento de la complejidad del mecanismo y con movimientos en tres dimensiones, como por ejemplo, el de la figura 1.2.

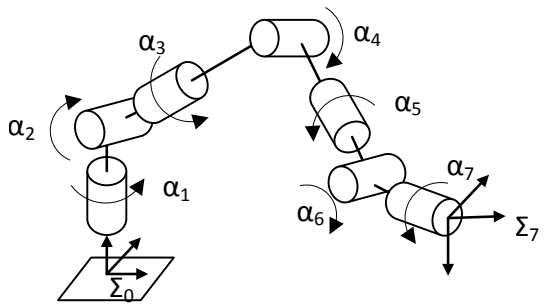


Figura 1.2: Ejemplo de sistema multicuerpo en 3 dimensiones

Aplicando al sistema mecánico a estudio los métodos propios del análisis estructural se obtiene su estructura cinemática: conjunto ordenado de subestructuras de cinemática determinada (calculable), denominados Grupos Estructurales (SG) [71], de forma que el análisis de cada una de estas subestructuras, en el orden en que se han obtenido, permite resolver la cinemática del sistema mecánico completo. Además, si dos o más SG de la estructura cinemática no presentan dependencia entre sí, su cinemática se puede abordar de forma simultánea. En estos casos, con una adecuada configuración y ajustando los parámetros de paralelismo para el tipo de problema y el hardware disponible, se puede aumentar la capacidad de cálculo para llegar a simular mecanismos cada vez más complejos.

1.2. Motivación

El dinamismo que el mercado impone a las empresas a la hora de innovar y desarrollar nuevos productos obliga a éstas a ser muy ágiles en las fases de diseño y prueba de

sus creaciones. A su vez, la reducción de costes de fabricación en la industria hace que el desarrollo de prototipos reales se vea progresivamente sustituido por simulaciones de los productos en potentes ordenadores.

Y lo mismo ocurre con el control en tiempo real de sistemas. La posibilidad de implementar código en cualquier tipo de dispositivo (smartphones, tablets, computadores de placa reducida, etc) hace que todos ellos puedan convertirse en potenciales controladores.

Tanto los grandes ordenadores como los pequeños dispositivos portátiles disponen hoy en día de varios núcleos de proceso que los hace susceptibles de ser empleados en tareas cada vez más complejas.

Colabora a este incremento de rendimiento la existencia de numerosas librerías de software, fuertemente optimizadas para el hardware disponible, haciendo uso de todos los cores disponibles de las CPU, o aprovechando los procesadores y memoria que aportan las actuales GPUs, e incluso explotando ambos recursos.

La combinación de todo lo anterior, y el potencial práctico subyacente, ha motivado el contenido de esta Tesis de Máster.

1.3. Objetivos

Este trabajo tiene como objetivo principal la aplicación de técnicas paralelas al software controlador de un robot, partiendo de una aplicación que, por ser de carácter general, es decir, válida para el análisis cinemático de sistemas multicuerpo con cualquier tipo de estructura cinemática, fue diseñada con un enfoque puramente secuencial. Pretendemos encontrar la configuración óptima de rendimiento (menor tiempo de ejecución) para el uso del software en dos vertientes:

- Como controlador en tiempo real de un sistema en producción.
- Como software simulador de nuevos prototipos mediante procesos iterativos ejecutados en grandes sistemas de cómputo, incluyendo sistemas heterogéneos.

La plataforma robótica concreta que se va a utilizar en este trabajo es un tipo especial de manipulador paralelo [63] denominado Plataforma de Stewart [61, 74]: un sistema multicuerpo que presenta una configuración muy particular, que lo hace especialmente interesante en el campo de la computación paralela.

Se ha partido del trabajo realizado en el Departamento de Ingeniería Mecánica de la Universidad Politécnica de Cartagena [41] en relación con el desarrollo de métodos computacionales para la obtención de la estructura cinemática de sistemas multicuerpo [70], así como de formulaciones cinemáticas [51, 71] y dinámicas [72] basadas en ecuaciones de grupo, como alternativa más eficiente a las formulaciones globales, que son las que con mayor profusión se emplean actualmente por la comunidad científica.

Una de las aplicaciones del método topológico propuesto consiste en la resolución de la cinemática inversa de un robot manipulador paralelo, tipo plataforma de Stewart, para el control de un prototipo construido. Dado que la propuesta utiliza un análisis secuencial de los SG obtenidos, se han planteado los siguientes hitos:

- Análizar la versión secuencial del código, identificando las áreas de más coste computacional, por si fueran susceptibles de paralelismo.
- Desarrollar un software simulador que adopte del código original los algoritmos de resolución de sistemas mecánicos y nos permita incluir controles de tiempos de ejecución y variaciones en el tamaño del problema para poder extenderlo a soluciones generales.
- Aplicar sobre el simulador diferentes técnicas de programación paralelas, estudiando las mejoras en cada caso. Según sea el hardware disponible, comenzar con la adaptación a sistemas multicore con memoria compartida, ampliando al uso de GPUs o multiGPU cuando sea posible. Analizar el rendimiento cambiando la librería de álgebra matricial utilizada, tanto en versiones de matrices densas como en dispersas. Analizar los speed-ups obtenidos respecto a los tiempos de ejecución de la rutina secuencial de partida.
- Crear un proceso de autooptimización con objeto de encontrar las configuraciones del software que ofrezcan mejor rendimiento para diferentes tamaños de problema y configuraciones del sistema de cómputo subyacente. Se entrenará al sistema con un conjunto de tamaños de problema seleccionados y se validará con un conjunto de test. La rutina deberá seleccionar en tiempo de ejecución el valor más adecuado de sus parámetros ajustables para el tamaño y la forma de la matriz que representa el problema a resolver.

1.4. Estado del arte

En esta sección se realiza una breve introducción a las herramientas de software existentes en al ámbito de la simulación de sistemas robóticos, y se exploran también las técnicas de programación paralela actuales. Interesa conocer cuáles de estas técnicas podrían emplearse en nuestro problema para mejorar el rendimiento de los simuladores actuales secuenciales, buscando la explotación de todo el potencial que ofrezca el hardware disponible.

Como vimos en la sección 1.1, los modelos matemáticos que subyacen a los sistemas multicuerpo emplean para su resolución métodos de álgebra lineal. Es por ello que se hace un repaso por las librerías matriciales más conocidas, algunas de las cuales se han utilizado en el desarrollo de este trabajo.

Por último se presenta el concepto de autotuning, como funcionalidad que ofrece el propio software de elegir en tiempo de ejecución los parámetros algorítmicos que maximizan el rendimiento en función del hardware y del tipo o tamaño del problema a abordar.

1.4.1. Entornos de simulación de robots

Existen en el mercado herramientas, tanto comerciales como gratuitas, para simular todo tipo de robots. Algunas de ellas emulan el comportamiento dinámico de equipos industriales existentes en el mercado, pero otras permiten modelar los robots a medida, es decir, añadiendo nuevos sólidos al sistema y uniéndolos mediante juntas de diversa tipología que restringen sus capacidades de movimiento relativo. Con todo ello, el simulador es capaz de calcular los movimientos posibles del mecanismo, incluyendo velocidades y aceleraciones. En los simuladores que permiten especificar el tipo de materiales (masa y rigidez), la simulación también incluye la posibilidad de los correspondientes análisis de comportamiento dinámico y resistente, en función del sistema de fuerzas que actúan entre sus componentes.

Algunos de estos simuladores incluyen entornos gráficos tanto para editar el mecanismo como para representar posteriormente la simulación. Con las herramientas adecuadas de zoom se puede analizar en detalle el movimiento de cada parte del robot y comprobar si existen roces o choques entre algunos elementos.

Otra funcionalidad muy valorada en estos simuladores es la de permitir añadir módulos de código a medida, para incorporar funcionalidades específicas. Podría ser el caso de algún tipo de pieza o de unión entre elementos que no vinieran incluidos en la librería de objetos que ofrece el software estándar. Se emplean para ello API para programación en lenguages como C++, Python, Matlab, Java, LUA.

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

- Gazebo [8]: Cuenta con un completo entorno visual de diseño y simulación en 3D. Ofrece por defecto algunas figuras básicas, como cilindros, esferas y cubos, pero también permite importar archivos creados en otras aplicaciones para incorporarlos al modelo a simular. Es posible definir los límites de los movimientos, fuerzas soportadas, viscosidad y fricción en las uniones.

  La dinámica es compatible con varios de los motores físicos más conocidos, como BULLET [4], DART [7] (Dynamics Animation and Robotics Toolkit), ODE [32] (Open Dynamics Engine) y SIMBODY [39]. Todos ellos ofrecen estructuras de datos y algoritmos para cinemática y dinámica en robótica.

  Además, dispone de una API de programación en C++ para poder incorporar plugins que modifiquen el comportamiento del robot y se ofrece con licencia Apache 2 y, por tanto, libre.

- V-REP [42]: Al igual que en el caso de Gazebo, este sistema permite crear modelos propios o importarlos desde otras herramientas. Dispone de una interfaz integrada para el diseño y la simulación.

La dinámica es compatible, además de con BULLET y ODE, con Newton [31] y dispone de funciones que pemiten enlazar la simulación con otras librerías externas, que se pueden programar en C/C++, Phyton, Java, Matlab y Octave.

Se ofrece con licencia comercial, con una modalidad gratuita con fines educativos.

- Matlab© [24]: Es una herramienta de software matemático que ofrece un entorno de desarrollo integrado con un lenguaje de programación propio, especialmente orientado a problemas de cálculo matricial, por lo que se puede emplear para simulación numérica de sistemas mecánicos como el que nos ocupa. Dispone además de herramientas específicas para diseñar aplicaciones de robótica, encuadradas en la denominada *Robotics System Toolbox* [27].

  Incorpora un ambiente de modelado con diagramas de bloques para la simulación de mecanismos de cuerpos rígidos denominado SimMechanics [28] y un entorno de simulación Simulink® [29] que aporta las herramientas necesarias para estudiar el comportamiento dinámico de los modelos.

  Matlab es software propietario de MathWorks®.

- MapleSim™ [23]: Parte como una evolución del conocido sistema de computación simbólica Maple™ [22] hacia el mundo de los resolvedores numéricos, ofreciendo funcionalidades similares a las herramientas anteriores, incluido 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 genera automáticamente las ecuaciones que describen el sistema completo, y permite su visualización. Se pueden añadir nuevos elementos creando simplemente nuevas ecuaciones que definan su comportamiento.

  MapleSim es software propietario de MapleSoft®.

1.4.2. Paralelismo en entornos de simulación de robots

Las rutinas de simulación de robots se basan en la resolución numérica de los sistemas de ecuaciones que describen los objetos que componen los mecanismos y la interacción entre los mismos. Además, estos cálculos se realizan de manera iterativa para conseguir que los robots alcancen todas las posiciones permitidas y para poder encontrar las posibles singularidades (posiciones peculiares de los ejes que pueden impedir la ejecución de movimientos). Con estos requerimientos de cómputo, se pueden conseguir mejoras en el rendimiento con un aprovechamiento completo de todo el potencial que ofrece el hardware. En el caso de un sistema *multicore*, a través del uso de todos los cores disponibles.

En esta sección se analiza la posibilidad de explotar el paralelismo que ofrecen las herramientas descritas en la sección anterior:

- Gazebo ha introducido recientemente el concepto de "Isla", entendiendo por tal a un grupo de componentes del sistema (cuerpos y uniones entre ellos) que se pueden aislar del resto. En caso de existir, cada "Isla" se puede ejecutar en un thread diferente. Para activar esta funcionalidad es necesario compilar de nuevo parte de la aplicación a partir de una versión que actualmente está en desarrollo.

- En V-REP actualmente es posible ejecutar simultáneamente en varios threads únicamente los procesos que simulan sensores (de distancia, de colisión y de proximidad).

- Matlab© proporciona directivas de programación paralela a través del *Parallel Computing Toolbox* [26]. Esta plataforma funciona poniendo a disposición del usuario copias locales adicionales de la herramienta, a las que se les denomina *workers*, y son éstos los que activan los cores disponibles en el hardware en caso de ser necesario. Además, es posible escalar a computación en clusters ejecutando esa misma aplicación sobre el entorno *Distributed Computing Server* [25]. Algunas funciones de Matlab© se ejecutan automáticamente en paralelo en presencia de estos entornos y algunos módulos adicionales, como los estadísticos, los de optimización y los de visión y procesamiento de imágenes, están diseñados para aprovechar paralelismo.

- En el caso de MapleSim™, MapleSoft® ha introducido en su software los conceptos de *Task Programming* y de *Grid Programming*. En el primer caso se han añadido intrucciones que permiten lanzar tareas para su ejecución simultánea en el ordenador local haciendo uso de los cores disponibles, en un modelo de memoria compartida. En el segundo caso se abre la posibilidad de ejecución en servidores remotos. Este salto al paralelismo es muy reciente, por lo que la propia compañía informa de que no hay certeza de que las funciones básicas de su software se puedan ejecutar en un modo seguro en un entorno multitarea.

Como resultado de este análisis observamos que los entornos de simulación de robots están en un estado incipiente en el empleo del paralelismo como aprovechamiento de todo el poder de cómputo que ofrecen los modernos sistemas multicore.

Una excepción son las herramientas de propósito general como es el caso de Matlab©. Su uso en áreas donde la computación paralela es objeto de investigación (como es el caso del reconocimiento de imágenes), ha provocado una evolución del software para adaptarse a los nuevos requerimiento. Recientemente ha incorporado funciones para el aprovechamiento de las GPUs, unidades de procesamiento de gráficos que contribuyen con sus procesadores y memorias a tareas de cómputo de carácter general. Por contra, el problema de Matlab es que es un lenguaje interpretado, lo que lo hace inviable en aplicaciones de tiempo real.

1.4.3. Técnicas generales de paralelismo

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 paralelas como es el caso de ordenadores multicore o las unidades de procesamiento de gráficos (o *GPU*). Para ello es necesario poder descomponer los problemas en partes independientes cuyas instrucciones puedan ejecutarse de manera simultánea por varios elementos de proceso. Los elementos de procesamiento son diversos e incluyen recursos tales como una computadora con múltiples procesadores, varios ordenadores en red, tarjetas de procesamiento de gráficos, o cualquier combinación de los anteriores.

En línea con todo ello, se han creado modelos de programación paralela cuya clasificación se basa en la arquitectura de la memoria subyacente: compartida o distribuida. OpenMP y MPI son las API más representativas de estos modelos.

OpenMP [35] ofrece una interfaz estándar para la programación multiproceso de memoria compartida [44]. Tiene como base el modelo fork-join, que consiste en dividir tareas pesadas en varios hilos (fork), para recoger posteriormente los resultados de cada uno de ellos en un solo resultado (join).

OpenMP permite especificar el número de threads que se quieren crear. En caso de no indicar un valor concreto, el sistema adopta automáticamente el número de cores disponibles en el sistema. Está permitido especificar un número de threads mayor que el número de unidades de procesamiento disponibles en el hardware. En este caso se puede optar por permitir que el sistema operativo sea el encargado de gestionar el encolado y posterior asignación a un procesador, o bien enlazar explícitamente cada thread OpenMP a una unidad de procesamiento física, método que se conoce como *thread affinity*. Intel© ofrece al programador un interface para gestionar este escenario [19].

MPI [33], por el contrario, es un entorno de programación paralela en el que no se maneja información compartida en memoria, sino que se basa en el paso de mensajes. Aunque en el código se declaren variables, cada proceso crea sus propias copias. Por tanto, estos programas no se pueden comunicar compartiendo zonas de memoria, sino que deben utilizar las rutinas de comunicación que ofrece MPI. Las básicas son:

- *MPI_INIT* para inicializar el entorno MPI.
- *MPI_SEND* para enviar un mensaje a otro proceso.
- *MPI_RECEIVE* para recibir un mensaje de otro proceso.

Este método de programación paralela presenta portabilidad a una gran variedad de sistemas, desde máquinas con memoria compartida hasta una red de estaciones de trabajo. A cada elemento de procesamiento se le asigna un número de identificación único, o rango. Todos los procesadores tienen una copia del mismo programa, pero cada proceso ejecutará distintas sentencias del programa por bifurcaciones introducidas

basadas en el rango del proceso.

Según la rutina de comunicación utilizada, puede ser necesario que el transmisor y el receptor estén *on-line* a la vez (y probablemente un proceso espere respuesta del otro) o, por el contrario, es posible que el emisor envíe la información a un *buffer*, para una entrega posterior, permitiendo que el emisor pueda seguir ejecutando código sin preocuparse de si el receptor ha recibido o no los datos.

De manera simultánea a los avances en el software enfocado a paralelismo, se buscaban también mejoras en el hardware. Pero el aumento de la capacidad de cómputo de los sistemas añadiendo cores a los procesadores presentaba una limitación técnica en términos de eficiencia térmica y límite de integración. La alternativa era la extensión del cálculo a coprocesadores disponibles en otros elementos del hardware, como es el caso de las unidades de procesamiento gráfico. Las GPUs, que fueron inicialmente diseñadas para aplicaciones gráficas, han ido evolucionando en los últimos años para incluir más funcionalidades y posibilidades de programación, lo que las ha convertido en lo que actualmente se denomina como GPUs de propósito general, o GPGPUs. Su gran ancho de banda y el rendimiento que ofrecen en operaciones de coma flotante las ha hecho candidatas a ser usadas en aplicaciones más allá del ámbito del manejo de gráficos, especialmente en áreas dominadas por operaciones sobre datos en paralelo, en particular de álgebra lineal y cálculos con matrices.

Al principio, los programas para GPGPU utilizaban extensiones de conocidas APIs de desarrollo de gráficos (como OpenGL y DirectX), obligando 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. A la vez, los principales fabricantes de estos dipositivos, crearon sus propios entornos, como es el caso de NVIDIA© CUDA™ [6] y OpenCL™ [34].

Debido a las diferencias fundamentales entre las arquitecturas de la GPU y la CPU, no cualquier problema se puede beneficiar de una implementación en una GPU. En concreto, es el acceso a memoria el que plantea las mayores dificultades. Las CPUs están diseñadas para el acceso aleatorio a memoria. Esto favorece la creación de estructuras de datos complejas, con punteros a posiciones arbitrarias en memoria. En cambio, en una GPU el acceso a memoria está mucho más restringido.

Como resultado de la creciente utilización combinada de CPU y GPU en las aplicaciones con alta demanda computacional, especialmente en el ámbito científico, los principales fabricantes comenzaron a crear una nueva generación de microprocesadores:

- Integrando ambos componentes en uno, como ocurrió con los AMD Fusion [1], primera generación de lo que actualmente AMD denomina *Accelerated Processing Units (APU)*.

- Desarrollando chips compuestos de cientos de cores, como es el caso de los Intel MIC [16], que pueden ser integrados en sistemas multicore y pueden hacer uso de los estándares de programación paralela de memoria compartida.

1.4.4. Librerías de álgebra matricial

La resolución de muchos problemas científicos y de ingeniería emplea métodos de álgebra lineal, en particular para el manejo de matrices. Son un ejemplo de ello el tratamiento de señales, la minería de datos, la dinámica de fluidos o, como en nuestro caso, el modelado y simulación de sistemas multicuerpo. Es por ello que se ha dedicado mucho esfuerzo al estudio y optimización de los algoritmos de resolución numérica del álgebra lineal, lo que se ha ido plasmando en el desarrollo de una generación de librerías de software cada vez más optimizadas, contribuyendo con ello a los avances experimentados en el ámbito de la computación científica.

BLAS (Basic Linear Algebra Subprograms) [3] es un conjunto de rutinas que proporcionan los bloques básicos para realizar operaciones sobre vectores y matrices. BLAS en su nivel 1 realiza operaciones del tipo vector-vector, en su nivel 2 realiza operaciones matriz-vector y en el nivel 3 matriz-matriz. Dada su eficiencia y portabilidad, sus algoritmos se usan en el desarrollo de software de álgebra lineal de alta calidad. LAPACK[20] es un ejemplo de ello.

BLACS (Basic Linear Algebra Communication Subprograms) [2] ofrece un interface para resolver problemas de álgebra lineal mediante el paso de mensajes. Se busca con ello facilitar la portabilidad hacia cualquier tipo de plataforma de memoria distribuida. Debido a ello, se usa como capa de comunicación para la librería ScaLAPACK.

PBLAS (Parallel Basic Linear Algebra Subprograms) [36] es un conjunto de rutinas para memoria distribuida similares a las que BLAS proporciona para memoria compartida. De hecho, los niveles son iguales que en BLAS: nivel 1 para operaciones vector-vector, nivel 2 para operaciones matriz-vector y nivel 3 para operaciones matriz-matriz. Ofrece las rutinas básicas que se incluyen en ScaLAPACK.

ScaLAPACK (Scalable Linear Algebre PACKage) [38] es una librería que contiene rutinas de álgebra lineal de alto rendimiento enfocadas a computadores con memoria distribuida. Para minimizar los movimientos de datos, las rutinas se basan en algoritmos que trabajan por bloques.

Como resumen de lo comentado, podemos representar la relación que existe entre las librerías mencionadas mediante el diagrama que se muestra en la figura 1.3.

Se pueden encontrar implementaciones de algunas de estas librerías adaptadas para entornos específicos. Por ejemplo, *MKL* [17] nació en el año 2003 como una evolución de las funciones ofrecidas por BLAS, LAPACK y ScaLAPACK optimizadas para ejecución en procesadores de la familia Intel©.

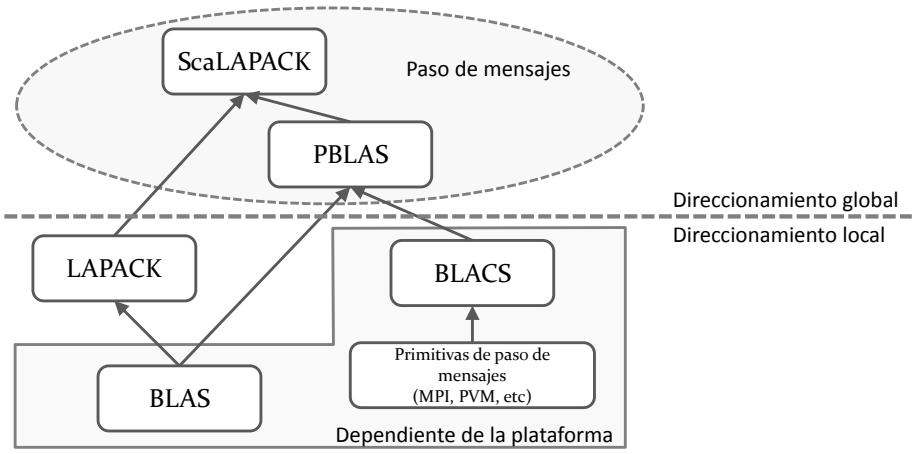


Figura 1.3: Librerías matriciales clásicas

La heterogeneidad del hardware disponible hoy en día para la ejecución de aplicaciones de elevado coste computacional ha llevado a los desarrolladores a invertir esfuerzos en investigación para el desarrollo de nueva librerías, o modificación de las existentes, adaptándose a los nuevos sistemas y explotar al máximo todo su potencial.

Por ejemplo, NVIDIA© ha desarrollado cuBLAS [5], una implementación de BLAS optimizada para trabajar con sus modelos de GPU y que viene incluida en las distribuciones de CUDA™. Esta librería ofrece a los programadores dos tipos de APIs:

- cuBLAS: donde la aplicación debe mover a la memoria de la GPU los datos necesarios para la ejecución de las funciones BLAS, y recoger posteriormente los resultados de vuelta a la memoria de la CPU.
- cuBLAS-XT: donde la aplicación mantiene la información en la memoria de la CPU, y es la librería la que se encarga de descomponer y distribuir el trabajo a una o a varias GPU presentes en el sistema (disponible para un subconjunto de llamadas a BLAS de nivel 3).

Por otro lado, el equipo que desarrolló LAPACK y ScaLAPACK ha diseñado e implementado una nueva generación de librerías de álgebra lineal para arquitecturas híbridas denominada MAGMA (*Matrix Algebra on GPU and Multicore Architectures*) [30], que incorpora los recientes desarrollos en algoritmos híbridos y de planificación, buscando minimizar los costes de sincronización y comunicación que requieren esos algoritmos.

El consorcio de tecnología Khronos Group lanzó OpenCL [34] como marco para estandarizar la escritura de programas que se ejecutan en distintas plataformas conformadas por CPUs, GPUs y otros tipos de procesadores o aceleradores de hardware. AMD, Apple, Intel, Nvidia y otros están apoyando OpenCL. En mayo de 2016 se publicó la versión 2.2 de este estándar. El uso de la API de OpenCL permite a los programadores lanzar Kernels de cómputo escritos en una versión reducida del lenguaje C, para

ser ejecutados tanto en GPU como en CPU.

Vienna Computing Library (ViennaCL) [43] es una biblioteca de computación científica de código abierto, escrita en C++ y que permite usar los recursos disponibles en arquitecturas paralelas por medio de un acceso sencillo y de alto nivel a CUDA, OpenCL y OpenMP. Se centra principalmente en operaciones comunes de álgebra lineal dispersa y densa (niveles BLAS 1, 2 y 3). También proporciona solucionadores iterativos con precondicionadores opcionales para grandes sistemas de ecuaciones.

En algunos casos, el tipo de problema a resolver se puede representar por matrices donde la mayor parte de sus elementos tienen valor cero. Estas matrices dispersas se pueden almacenar empleando técnicas específicas que permiten reducir el tamaño ocupado en memoria. La ventaja es notable en el caso de matrices de gran tamaño donde algoritmos de resolución del álgebra lineal optimizados para este tipo de formato consiguen mejoras en los tiempos de ejecución. Por ejemplo:

- Intel© ofrece junto a la librería MKL un conjunto de funciones englobadas en el paquete *PARDISO (Parallel Direct Sparse Solver)* [18].

- También se pueden encontrar soluciones específicas para matrices disperas, como la ofrecida por *HSL (Harwell Subroutine Library)* [9]. Comenzó a desarrollarse en 1963 para entornos *mainframes* de IBM, y ha ido evolucionando a lo largo del tiempo adaptándose al hardware existente, tanto grandes ordenadores como equipos domésticos, incluyendo ahora el soporte a procesadores multicore. Algunas de las librerías incluidas en este paquete son las siguientes:

  - MA27 [10], especializada en la resolución de sistemas de ecuaciones basados en matrices dispersas simétricas.
  - MA57 [11], versión mejorada de la MA27 para matrices de gran tamaño.
  - HSL_MA86 [12], para resolver sistemas lineales indefinidos sobre matrices simétricas dispersas de gran tamaño. Emplea OpenMP y está diseñada para trabajar en sistemas multicore.
  - HSL_MA87 [13], para resolver sistemas lineales definidos sobre matrices simétricas dispersas de gran tamaño. También emplea OpenMP.
  - HSL_MA97 [14], la más general ya que es capaz de trabajar con sistemas definidos e indefinidos, aplicando diferentes algoritmos de resolución en cada caso. También emplea OpenMP.

  HSL recomienda el uso de HSL_MA97 de manera general, pero matiza que MA57 está más indicada para matrices muy dispersas de pequeño tamaño y HSL_MA86 para trabajar con matrices muy grandes.

  En este trabajo hemos empleado MA27, la única que se ofrece de manera totalmente gratuita.

1.4.5. Autotuning

Con carácter general es de esperar que un código fuente, desarrollado para un sistema de cómputo y para un juego de datos concreto, muy probablemente ofrezca un rendimiento pobre en otros sistemas o con tamaños de problema diferentes.

Como se describe en [68], el autotuning, o Automatic Performance Tuning, es una técnica de naturaleza empírica que busca maximizar el rendimiento de una aplicación software en una gran variedad de arquitecturas de ejecución sin sacrificar la portabilidad o la productividad. Se pretende delegar en los mismos ordenadores la búsqueda de todas las posibilidades existentes, y encontrar automáticamente la configuración que ofrece mayor rapidez de ejecución en función del hardware disponible.

Un proceso de autotuning explora el espacio de parámetros posibles y aplica sobre ellos las optimizaciones conocidas y preimplantadas en el código desarrollado. Esta búsqueda puede realizarse de manera exhaustiva o heurística.

El Grupo de Computación Paralela de la Universidad de Murcia [40] ha publicado recientemente algunos artículos [47, 48, 53, 54, 55] donde se presentan propuestas de aplicación de las técnicas de optimización en el campo de las rutinas de álbegra lineal, buscando un aprovechamiento óptimo de los sistemas homogéneos Multicore y MultiGPU.

Como explica en una entrevista el profesor Thomas Fahringer, de la Universidad de Innsbruch [75], *"La mayoría de los autotuners se limitan a objetivos únicos de optimización tales como comunicación, distribución de trabajo, tiempo de ejecución o consumo de energía. Estas técnicas se refieren a autotuning monoobjetivo. Actualmente, se están actualizando cada vez más tuners para hacer frente a la optimización dual o multiobjetivo para manejar dos o incluso más objetivos de optimización. Para cada objetivo se requiere un conjunto específico de parámetros de ajuste. Por ejemplo, el tiempo de ejecución se puede ajustar utilizando tamaños de mosaico, parámetros de programación, número de hilos o núcleos, distribución de datos y trabajo, entre otros. El consumo de energía puede estar influenciado por ajustes de voltaje y frecuencia, pero también por parámetros de tiempo de ejecución. Los costes de computación en su conjunto están influenciados por parámetros de tiempo de ejecución, tamaño de memoria y almacenamiento de datos asignados. Las transformaciones del programa son opciones de ajuste para los tres objetivos de optimización. Un problema particular es detectar e implementar aquellos parámetros de ajuste que tienen un impacto significativo en un objetivo de optimización dado e ignorar aquellos con efecto despreciable. Además, el espacio de búsqueda puede explotar debido a los grandes rangos de valores de estos parámetros. Aún peor, cambiar el valor de un parámetro puede mejorar cierto objetivo de optimización a costa de otro".*

En cuanto a los algoritmos de búsqueda, el profesor comenta que *"Desafortunadamente, no existe una sola solución que funcione mejor para cada situación. Para el ajuste automático monoobjetivo, la búsqueda aleatoria es adecuada si la diferencia entre*

la alternativa mejor y la peor es pequeña. La búsqueda local, como hill climbing, es una opción para resolver problemas de optimización donde la búsqueda exhaustiva es impracticable. Los métodos de búsqueda local modifican de forma iterativa la configuración de parámetros actual hasta que no se pueda mejorar más. La búsqueda local se puede aplicar para determinar parámetros numéricos, así como secuencias de transformaciones del programa. Sin embargo, los métodos de búsqueda local sufren tres inconvenientes: (1) el óptimo calculado puede depender del punto de partida; (2) la búsqueda local puede quedar atrapada en óptimos locales; (3) pueden ser necesarias numerosas iteraciones".

Adicionalmente, conviene resaltar que las recientes librerías matriciales ya se están desarrollando con vistas a explotar las arquitecturas heterogéneas existentes. Un ejemplo de ello es la Intel Math Kernel Library (Intel©MKL), que ofrece un aprovechamiento dinámico de los cores disponibles en la CPU, o MAGMA, enfocada al aprovechamiento de arquitecturas heterogéneas.

1.5. Metodología

Al inicio de este trabajo tenemos acceso al software desarrollado por el Departamento de Ingeniería Mecánica de la Universidad Politécnica de Cartagena que controla un robot manipulador paralelo de 6 articulaciones. Este software, desarrollado en FORTRAN [15], se basa en estudios de los autores sobre la descomposición de sólidos multicuerpo en componentes independientes [51, 70, 71, 72]. Un primer análisis concluye que la ejecución se realiza de manera secuencial y no aprovecha las posibilidades de cálculo simultáneo de cada articulación, como apuntan los citados estudios.

Por tanto, se aborda la tarea de dotar de paralelismo al citado software mediante la inclusión de técnicas de programación paralela de acuerdo a la siguiente planificación:

1. Desarrollaremos una copia del software original, manteniendo la estructura y las rutinas de cálculo. Será el simulador donde aplicaremos posteriormente los cambios necesarios que permitan gestionar varios tamaños de problema y aplicar técnicas de paralelismo con objeto de realizar comparativas de rendimiento.

2. Buscaremos las secciones del código susceptibles de ejecutarse en paralelo.

3. Comenzaremos aplicando el paradigma de memoria compartida OpenMP [35] con matrices densas. Cambiaremos el tamaño del problema y buscaremos los mejores tiempos de ejecución en función de la combinación de *threads* y cores disponibles.

4. Aplicaremos un segundo nivel de paralelismo sobre la librería Intel©MKL [17], combinando threads para OpenMP con threads para MKL.

5. Ejecutaremos los mismos escenarios sobre matrices dispersas.

6. Repetiremos estos escenarios reemplazando la librería MKL por su versión dispersa MKL PARDISO [18].

7. Haremos lo mismo empleando la librería MA27 [9], que resuelve sistemas de ecuaciones basados en matrices dispersas simétricas.

8. En la siguiente sección abordaremos el uso combinado de CPU y GPU, para lo que usaremos la librería MAGMA [30]. Realizaremos distintas combinaciones de threads OpenMP utilizando una GPU o más, según el hardware subyacente.

1.6. Contenido del documento

En esta sección se describe la estructura que tiene el presente documento, junto con una breve explicación del contenido que alberga cada uno de los capítulos que lo componen:

- En el primer capítulo se realiza un introducción teórica al campo del análisis de los sistemas multicuerpo, donde destaca la importancia de los métodos del álgebra lineal en la resolución de este tipo de problemas. Además se presentan algunas de las herramientas de simulación de robots más conocidas y se estudia el escaso nivel de adaptación que este tipo de software ofrece en la actualidad hacia los modernos sistemas de cómputo multicore. Se describen las técnicas de programación paralelas más comunes y las librerías de álgebra lineal que mejor se adaptan a este tipo de problemas.

- El capítulo 2 realiza una introducción al modelado matemático de sistemas mecánicos, basado en seleccionar el tipo de coordenadas adecuado para representar la geometría y posición de sus elementos y el proceso de análisis estructural que persigue encontrar los bloques mínimos que pueden ser calculados por separado, y el orden en que debe hacerse. Se presenta la Plataforma de Stewart, un tipo especial de sistema multicuerpo que es el objeto de este trabajo de fin de Máster, dado que presenta la característica de contener grupos que pueden calcularse simultáneamente. Por último, se presenta el software desarrollado en el Departamento de Ingeniería Mecánica de la Universidad Politécnica de Cartagena que plasma en un código FORTRAN la base teórica explicada anteriormente.

- El capítulo 3 describe el simulador desarrollado en el marco de esta Tesis el cual, heredando del software original las rutinas de cálculo que resuelven el problema, permite modificar las dimensiones del mismo y aplicar técnicas de paralelismo, pudiendo registrar los tiempos de ejecución para ayudar al análisis de la mejor configuración en cada tipo de problema.

- El capítulo 4 realiza un análisis del rendimiento que ofrece el software de simulación antes de aplicar ninguna técnica de paralelismo al cálculo de los grupos que componen el sistema multicuerpo. Se comparan los resultados obtenidos con cada librería matricial, y que serán la base sobre la que determinar el speed-up de las soluciones que se irán aplicando en los capítulos posteriores. A continuación se aplicará paralelismo a nivel de librerías, en el caso de aquéllas que implementan algoritmos paralelos, como es el caso de MKL y PARDISO, y se analizarán las mejoras respecto a la solución secuencial.

- El capítulo 5 introduce la paralelización usando el sistema de memoria compatida OpenMP. Se estudia la variación en los tiempos de ejecución al aplicar el paralelismo en un sólo nivel (sólo OpenMP), para posteriormente ampliar el estudio a un doble paralelismo (OpenMP $\times$ MKL). Se prueban diversas combinaciones de threads en sistemas compuestos por 12 y 24 cores, variando el tamaño de las matrices que representan la geometría del sistema y modificando el número de elementos que se pueden calcular simultáneamente. Se realiza una primera aproximación a los métodos de *scheduling* a la hora de gestionar la paralelización del bucle que calcula los grupos de la plataforma de Stewart.

- El capítulo 6 aborda el estudio del uso de GPUs en el cálculo de los sistemas de ecuaciones que resuelven el problema cinemático. Se analizan las mejores combinaciones de threads según la cantidad de GPUs disponibles y se presenta una modificación del algoritmo para gestionar los casos en los que el número de grupos paralelos, el de cores y el de GPUs no estén balanceados.

- El capítulo 7 presenta el proceso de autotuning que se ha incorporado a la aplicación, que le permite seleccionar automáticamente la configuración que minimiza los tiempos de ejecución en función del tamaño del problema y del hardware sobre el que se ejecuta el simulador.

- En el último capítulo se enumeran las principales conclusiones obtenidas durante este trabajo, de acuerdo a los objetivos planteados al principio de la Tesis y se expondrán unas líneas futuras de trabajo e investigación en este área.

Capítulo 2

Descripción del problema y entorno de trabajo

En este capítulo se introducen los conceptos relevantes en el campo de los sistemas multicuerpo y algunas de las investigaciones más recientes en este área que han llevado a automatizar el proceso de obtención de los modelos matemáticos que explican su estructura y que sirven como base para el desarrollo del software de simulación. El objetivo que se persigue es estudiar el comportamiento de los sistemas antes de abordar el proceso de fabricación real de los mismos. De esta manera se evitan los costes potenciales que se producirían si los usuarios detectaran fallos debido a la elección errónea de los parámetros que influyen en el diseño.

Describiremos el tipo de mecanismo concreto sobre el que vamos a trabajar, un robot manipulador paralelo denominado Plataforma de Stewart, así como el entorno hardware sobre el que se va a ejecutar la aplicación y las herramientas de software empleadas en el desarrollo de este trabajo.

2.1. Modelado de sistemas mecánicos

La simulación de cualquier sistema multicuerpo comienza con una fase inicial de modelado en la que se deben seleccionar las coordenadas que permitan definir perfectamente la posición de todos los componentes del sistema y establecer el conjunto de ecuaciones que relacionen estas variables entre sí.

Un aspecto que tiene gran relevancia para este análisis, es la elección del tipo de coordenadas a emplear. Se podría optar por usar coordenadas independientes, que son las mínimas necesarias para describir completamente el problema, y cuyo número coincide con el de Grados de Libertad del sistema (su capacidad de movimiento). Por ejemplo, en la figura 2.1, que representa un mecanismo plano de cadena abierta, el valor de los tres ángulos sucesivos determina los tres grados de libertad del sistema y son suficientes para definir el movimiento del robot.

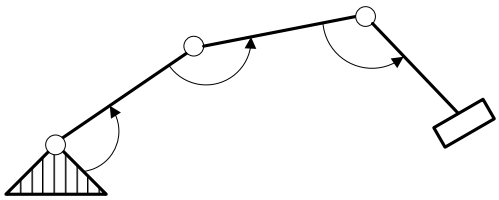


Figura 2.1: Esquema de un mecanismo sencillo de cadena abierta

Ahora bien, si disponemos de un mecanismo clásico de cadena cerrada como el de la figura 2.2, los parámetros elegidos podrían no definir unívocamente la posición del mecanismo, es decir, para un cierto valor del ángulo, caben dos posiciones posibles del mecanismo. Por tanto, este tipo de coordenadas no parece el adecuado para un método de propósito general.

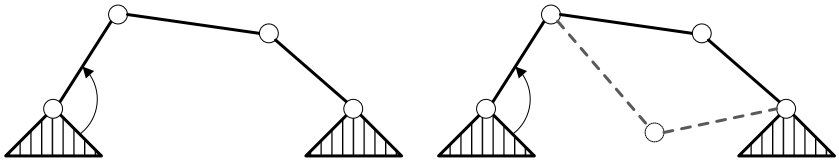


Figura 2.2: Esquema de un mecanismo sencillo de cadena cerrada y soluciones múltiples en coordenadas independientes

Si lo que se quiere es emplear suficientes variables para definir perfectamente la posición de cada elemento del mecanismo, entonces se utilizan las que se denominan coordenadas dependientes.

En este caso ocurre que se tienen más parámetros que grados de libertad del sistema, por lo que, para determinar la solución, se debe encontrar un mecanismo para describir cómo se relacionan entre sí. Para ello se emplean las llamadas *ecuaciones de restricción*, las cuales se encargan de describir en un lenguaje matemático:

- El carácter rígido de los sólidos, por ejemplo el carácter fijo de las distancias o el valor de los ángulos entre los elementos.

- Las condiciones que describen la naturaleza de la unión entre los sólidos, también conocida como *par cinemático*. Uniones serían, por ejemplo, las articulaciones, guías correderas, articulaciones cilíndricas, rótulas esféricas, guías deslizantes, etc. Cada una de ellas, según sea su tipo, puede imponer más de una ecuación de restricción, y tiene relación con el número de grados de libertad del movimiento relativo que restringe dicha unión.

Se cumple que, si designamos por n al número de coordenadas dependientes y g al número de grados de libertad del mecanismo, el número de ecuaciones de restricción necesarias será $r = n - g$.

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

2.1.1. 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 2.3 observamos un solo lazo.

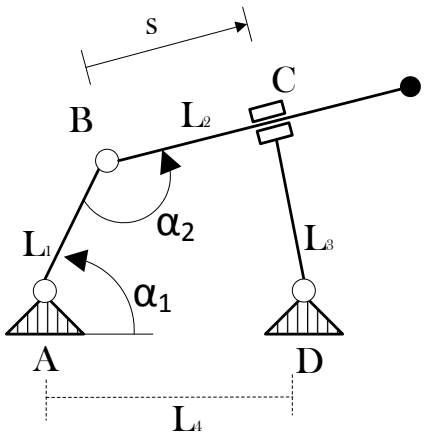


Figura 2.3: Modelado empleando coordenadas relativas

La condición de cierre será:

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

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

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

Como vemos, las formulaciones a que dan lugar estas coordenadas son complicadas y la evaluación de los términos presenta funciones trigonométricas, lo que incrementa el coste computacional.

2.1.2. 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 de un punto cualquiera del elemento (que puede ser el centro geométrico o el de masas), y la orientación del mismo (un ángulo en el caso del plano). 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 2.4 se observa un modelado como el descrito.

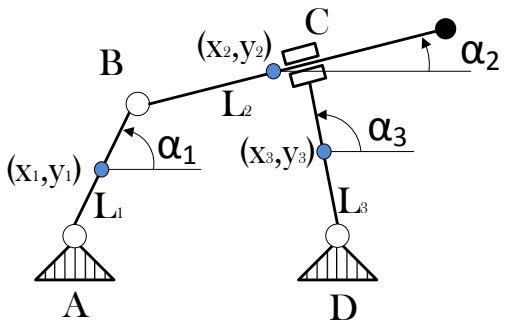


Figura 2.4: Modelado empleando coordenadas de punto de referencia

Como en este caso hemos elegido 9 coordenadas dependientes, y el grado de libertad es 1, hay que imponer 8 ecuaciones de restricción, que se detallan a continuación [52]:

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

Observamos dos ecuaciones de restricción para cada par cinemático, empezando por al par referenciado como A y siguiendo hasta el D.

Este tipo de coordenadas tiene como inconveniente principal el elevado número de coordenadas y la complejidad de las ecuaciones de restricción debido a la presencia de funciones trigonométricas.

2.1.3. 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 en la figura 2.5.

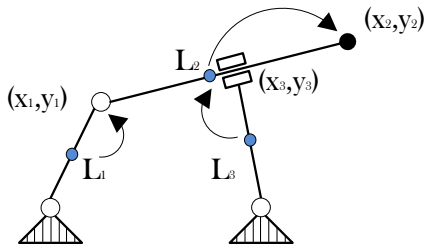


Figura 2.5: Modelado empleando coordenadas naturales

En la introducción al problema del modelado realizado en la sección 1.1 se utilizaron este tipo de coordenadas y se obtuvieron las correspondientes ecuaciones de restricción.

2.1.4. Coordenadas mixtas

Es posible realizar modelados combinando coordenadas naturales con relativas. Este método resulta de gran interés a la hora de definir pares de unión entre elementos que sean del tipo engranaje pues, en este caso, se debe poder especificar la proporcionalidad en los valores de los giros. También se suelen emplear cuando existen elementos extensibles en el sistema, para especificar ecuaciones que reflejen distancias. Por último, otro caso que justificaría esta combinación de tipos de coordenadas se produciría cuando fuera conveniente conocer el valor de un cierto ángulo presente entre elementos del mecanismo.

2.1.5. Tipo de coordenadas usadas en este trabajo

Para el modelado de la plataforma de Stewart objeto del presente trabajo se han seleccionado las coordenadas naturales, ya que presentan la ventaja de obtener ecuaciones de restricción sencillas (sin presencia de funciones trigonométricas) y permiten situar a cada elemento con independencia de los demás, manteniendo un reducido número de coordenadas.

2.2. Análisis estructural

Una manera de abordar el proceso de modelado de sistemas mecánicos consiste en la descomposición del mismo en problemas más pequeños. Para ello, en Teoría de Mecanismos, el análisis estructural se encarga de identificar el subconjunto de sólidos que, unidos entre sí y con movimiento relativo entre ellos, forman un sistema multicuerpo completo.

Con un método grafo-analítico se puede obtener la estructura cinemática de un mecanismo con ayuda de lo que se conoce como *Grafo Estructural*. En este tipo de grafos se puede representar la topología siguiendo la siguiente convención:

- Cada nodo del grafo representa un eslabón, entendiendo por tal a una pieza o grupo de piezas del sistema. Estos nodos se etiquetan con un número que corresponde con el del eslabón.

- Si dos eslabones están unidos de manera que puedan tener movimiento relativo entre ellos, se trazan entre ellos tantas líneas como movimientos permita esa unión.

- De las líneas descritas en el punto anterior, se trazarán con trazo grueso aquéllas que el analista defina como movimientos de entrada, y que se pueden identificar por una variable independiente.

Tomemos como ejemplo un sistema sencillo como el de la figura 2.6, donde se muestran los eslabones numerados del 1 al 4.

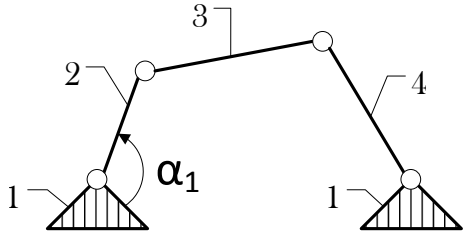


Figura 2.6: Esquema de un cuadrilátero articulado sencillo

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 2.7. Al existir entre ellos un movimiento independiente (el que rige el ángulo α_1), el trazo de la línea es grueso. El resto son uniones simples de una sola línea (sólo es posible un movimiento entre ellos) y de trazo fino.

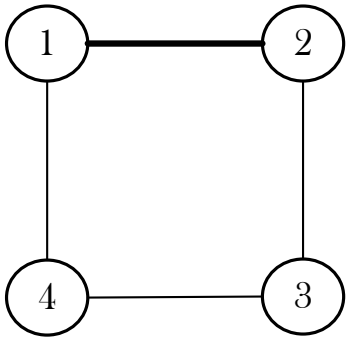


Figura 2.7: Análisis de un cuadrilátero articulado: grafo estructural

La secuencia general del método grafo-analítico [57, 64], concluye con la identificación de los grupos estructurales constituyentes del sistema. En este ejemplo los grupos identificados se muestran en la figura 2.8.

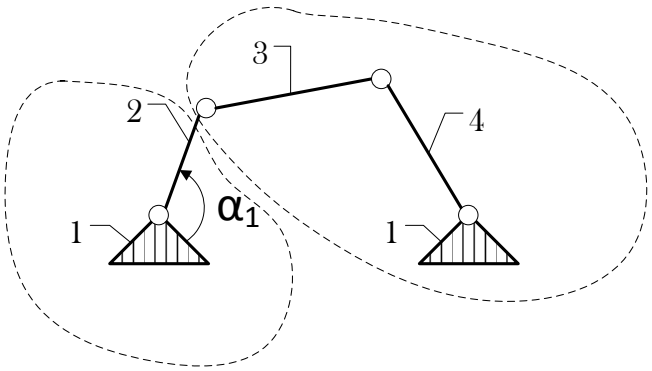


Figura 2.8: Cuadrilátero articulado sencillo: división en grupos estructurales.

Esta información permite elaborar el *Diagrama Estructural* que vemos en la figura 2.9. Cada círculo representa un grupo estructural. El identificado como 0 se añade artificialmente siempre, e identifica a la base inmóvil. El resto se etiquetan con dos dígitos, el número de partes móviles y movimientos de entrada al grupo. Se dibuja una flecha de unión entre dos grupos cuando hay conexión móvil 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.

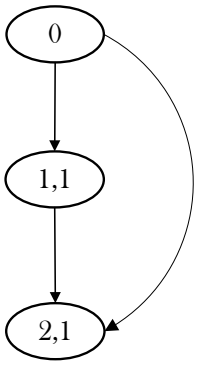


Figura 2.9: Análisis de un cuadrilátero articulado: diagrama estructural

En el caso del modelado de mecanismos complejos, la tarea de identificar los 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, obtiendo su estructura cinemática mediante métodos computacionales tomando como base los métodos grafo-analíticos descritos anteriormente. Uno de los métodos propuestos [69] necesita como entrada la matriz de adyacencia simétrica M. Este tipo de matrices permite representar la topología de un mecanismo con todos sus eslabones.

Un valor de $M(i, j)$ distinto de cero indica que los eslabones i, j forman un par. Como resultado se obtiene una matriz que indica el grupo al que pertenece cada eslabón en el orden en el que se han identificado.

El análisis estructural, en tanto que determina los grupos que componen un sistema multicuerpo, es una parte esencial del modelado y posterior simulación en ordenadores. De hecho, como veremos a continuación, es capaz de detectar la existencia de grupos independientes que, al no tener relación entre ellos, pueden calcularse simultáneamente.

2.3. Plataforma de Stewart

El sistema sobre el que vamos a trabajar es una Plataforma de Stewart: un tipo de robot paralelo que consta de una superficie con el apoyo de seis actuadores independientes, como podemos ver en la figura 2.10. El terminal de la plataforma dispone de seis grados de libertad, por lo que puede ser desplazado en las tres direcciones del espacio, así como girado respecto a esas tres mismas direcciones.

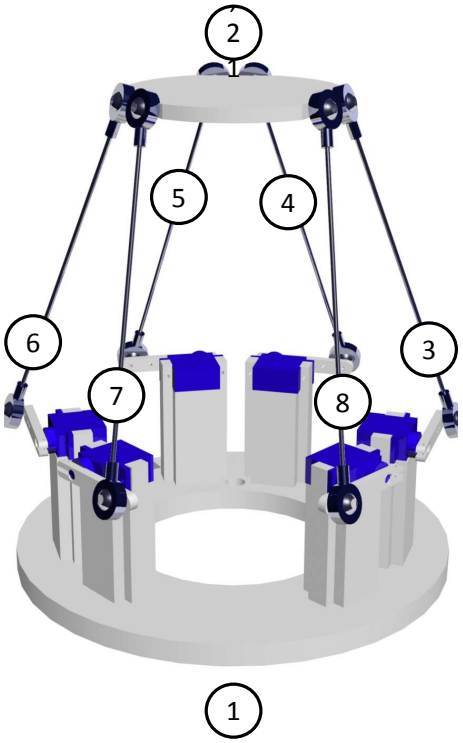


Figura 2.10: Esquema de una Plataforma de Stewart

Existen diversas configuraciones de Plataformas de Stewart, pero el modelo elegido en nuestro caso utiliza actuadores rotativos que actúan sobre manivelas. El movimiento de las manivelas se traslada mediante juntas esféricas a unas barras que se conectan, también con juntas esféricas, directamente con un punto fijo de la plataforma móvil.

Todos ellos actúan de manera simultánea, lo que permite que el mecanismo completo pueda manejar cargas superiores a su propio peso. Además, la rigidez que ofrece esta configuración permite realizar movimientos de gran precisión y alcanzar altas velocidades y aceleraciones en comparación con otros tipos de brazos cuyas articulaciones se disponen en serie, como el mostrado en la figura 1.2.

Esto hace que las aplicaciones de este sistema sean muy diversas, e incluyen simuladores de vuelo, mecanizado, ensamblaje y soldadura de piezas, posicionamiento de precisión de telescopios y asistentes quirúrgicos robóticos.

El análisis estructural (cuya metodología se introdujo en la sección 2.2) revela que este sistema multicuerpo consta de los siguientes grupos estructurales:

- Un bastidor fijo (elemento 1).
- El elemento terminal (elemento 2).
- 6 grupos idénticos (elementos del 3 al 8), formados cada uno a su vez por dos sólidos: una manivela y una barra, y por tres juntas: dos esféricas y una de rotación.

El diagrama estructural asociado a este mecanismo, que observamos en la figura 2.11, muestra estos grupos y sus dependencias, lo que permite determinar el orden en el que se deben resolver cada uno de ellos.

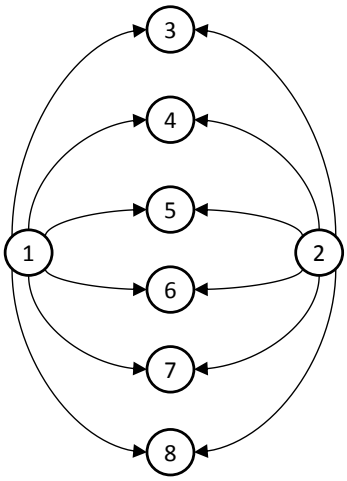


Figura 2.11: Diagrama estructural de una Plataforma de Stewart

El grupo 1 no hay que resolverlo, dado que es inmóvil. La orientación de las flechas nos está indicando que los grupos del 3 al 8 requieren información proveniente de la resolución del grupo 2 y, por tanto, éste será el primero en calcularse. El resto se podrá resolver simultáneamente.

El método de resolución aplicado a cada grupo, introducido en [71], necesita el conjunto de coordenadas del grupo, que denotaremos como q, y las ecuaciones de restricción asociadas Φ. El vector q se divide en un vector de coordenadas independientes, h, obtenido a partir del cálculo del grupo estructural anterior en la secuencia, y un vector de coordenadas dependientes, φ. A partir de aquí, los autores desglosan el proceso en la resolución de los siguientes problemas:

1. **Problema de la posición**: Se busca, dado un valor conocido de las variables independientes del sistema, encontrar el valor de las variables dependientes φ, calculando la matriz Jacobiana Φ_φ [67] de las ecuaciones de restricción respecto al vector de variables dependientes φ, y resolviendo el sistema $\mathbf{\Phi} = 0$ por medio del método iterativo de Newton-Raphson [50, 58]:

$$\varphi_k = \varphi_{k-1} - (\Phi_\varphi)^{-1}_{k-1}(\Phi_q)_{k-1}$$

   donde k identifica al número de iteración. En las iteraciones, el valor de las coordenadas independientes permanece fijo.

2. **Problema de la velocidad**: Una vez obtenido el valor de las variables dependientes φ, éstas se usan para resolver el sistema $\dot{\mathbf{\Phi}} = 0$, derivando la ecuaciones de restricción respecto al tiempo, y resolviéndolas para las coordenadas dependientes. Si denotamos por Φ_h a la matriz Jacobiana respecto de las coordenadas independientes h, entonces, conocido el valor de las velocidades de las coordenadas independientes $\dot{\mathbf{h}}$, podemos escribir:

$$\dot{\varphi} = -(\Phi_\varphi)^{-1}[\Phi_h \dot{h}]$$

3. **Problema de la aceleración**: La aceleración de las variables dependientes se puede resolver derivando los vectores de velocidad de las restricciones respecto al tiempo para calcular $\ddot{\Phi}_q = 0$:

$$\ddot{\varphi} = -(\Phi_\varphi)^{-1}[\Phi_{\ddot{h}} \ddot{h} + \dot{\Phi}_q \dot{q}]$$

El software desarrollado por el grupo de investigación de la UPCT implementa este algoritmo para sistemas multicuerpo planos y espaciales en general, y para la Plataforma de Stewart en particular, y ha sido objeto del ulterior estudio de paralelización abordado en esta Tesis de Máster.

Los autores han observado que las matrices que surgen de una formulación basada en la descomposición de una plataforma de Stewart en grupos estructurales son dispersas, donde más del 70 % de los valores son nulos, estando los valores no nulos concentrados alrededor de la diagonal.

2.4. Software de control

En esta sección se aborda el análisis del código fuente desarrollado en el Departamento de Ingeniería Mecánica de la Universidad Politécnica de Cartagena. Emplea el lenguaje FORTRAN e implementa el algoritmo necesario para resolver los problemas de posición, velocidad y aceleración según quedó descrito en la sección anterior. En el Anexo A se puede consultar el diagrama de flujo que representa la rutina principal que gestiona los cálculos.

La geometría y movilidad de los elementos que constituyen los sistemas multicuerpo se representan mediante matrices, por lo que el cálculo de la posición del robot en cada momento se realiza mediante la resolución de los respectivos sistemas de ecuaciones. En la versión secuencial del software se hace uso de la librería MKL de Intel©, en concreto de las siguientes funciones:

- *DGETRF*, para la descomposición LU de las matrices.
- *DGETRS*, para la resolución del sistema de ecuaciones.
- *DGEMM*, para la multiplicación de matrices.

Aparte de las ya mencionadas, se necesitan las siguientes funciones de soporte para operar sobre las matrices y realizar cálculos adicionales (el coste computacional de éstas y las anteriores se recoge en la tabla 2.1, donde n indica el número de filas o columnas de las matrices y m el número de datos a tratar):

- *reshape*, para construir estructuras con dimensiones específicas a partir de unos datos de entrada, por ejemplo crear matrices a partir de arrays unidimensionales.
- *tranpose*, para la transposición de matrices.
- *norma*, función que implica el cálculo de una raíz cuadrada y un producto escalar y que se emplea para el cálculo del error al resolver el problema de la posición.

Tabla 2.1: Coste computacional de las funciones empleadas en los cálculos de la plataforma de Stewart, donde n indica la dimensión de las matrices (número de filas o columnas) y m el número total de datos a tratar

Función	Coste
DGETRF	$O(n^3)$
DGETRS	$O(n^2)$
DGEMM	$O(n^3)$
reshape	$O(m)$
transpose	$O(m)$
norma	$O(m)$

El código está controlado por un bucle externo que mantiene la ejecución hasta alcanzar un valor preestablecido, que representa el número de iteraciones. Para calcular este valor necesitaremos conocer los siguientes parámetros de entrada:

- `tFin`: El tiempo límite de ejecución.
- `tSalto`: El intervalo de tiempo simulado en cada iteración.

De esta manera estaremos fijando una ejecución del programa durante `tFin`/`tSalto` iteraciones. Cada ejecución realiza todos los cálculos necesarios para determinar la posición, velocidad y aceleración de los componentes (o grupos estructurales) del robot que, en el caso de la Plataforma de Stewart objeto de este estudio, son el Terminal (la plataforma móvil) y los conjuntos que forman cada par Manivela-Barra.

Por otro lado, si nos centramos en el caso particular del robot de la figura 2.10, con seis conjuntos Manivela-Barra formados por un actuador rotativo y una biela, las constantes que indican el tamaño de las matrices (para representar las geometrías) y el número de grupos estructurales serían las siguientes:

- `dim-T`, para el Terminal, con valor fijo de 12.
- `dim-MB`, para los conjuntos Manivela-Barra, con valor fijo 15.
- `dim-MM`, para el centro de masas del Terminal, con valor fijo 3.
- `numGE` número de grupos Manivela-Barra, con valor fijo 6.

Las matrices se cargan al inicio de la ejecución del programa con los valores de los puntos y vectores que definen el mecanismo. Estos valores han sido calculados previamente mediante métodos geométricos.

Con todo ello, el funcionamiento general del programa queda reflejado en el algoritmo 2.1. Como vemos en el bucle `for all` de la línea 4, los pares Manivela-Barra se ejecutan secuencialmente uno tras otro.

Algoritmo 2.1 Esquema de la resolución secuencial de los grupos estructurales en el software de control original

```
1: Cargar en memoria las coordenadas geométricas del modelo
2: for número de iteraciones (tFin · tSalto) do
3:    Resolver la cinemática del Terminal, de dimensión dim-T
4:    for all Manivela-Barra, numGE do
5:       Resolver la cinemática de cada par Manivela-Barra, de dimensión dim-MB
6:    end for
7: end for
```

Cada iteración modifica las coordenadas de todos los elementos que componen el sistema mecánico para representar la ubicación espacial en cada momento. Estos valores servirán de entrada para la siguiente iteración.

Por tanto, las tareas que presentan el mayor coste computacional se encuentran dentro de los bloques que resuelven la cinemática del Terminal y los pares Manivela-Barra. Detallamos a continuación cada uno de ellos.

2.4.1. Solución del Terminal

La resolución de la cinemática del Terminal requiere introducir dos nuevos parámetros que regulan el comportamiento del software de control:

- `tolErr`, representa la tolerancia admitida para el error que se obtiene cuando se resuelve el problema de la posición. Cuando el proceso iterativo de cálculo converge hacia una solución con un error menor del prefijado, el proceso se detiene y la solución se considera válida.
- `numEslab`, indica el número de eslabones presentes en cada grupo estructural. Este valor se emplea en el cálculo de las matrices de transformación y sus derivadas.

En el algoritmo 2.2, que recoge toda la secuencia de cálculos, se observa el bucle donde se produce la resolución iterativa del sistema de ecuaciones en el problema de la posición hasta que se alcanza un error inferior al parámetro `tolErr` (Líneas 2-8).

Algoritmo 2.2 Esquema de la resolución del grupo que representa al Terminal en el software de control original

```
1: { ! Resolver el problema de POSICION}
2: while error >tolErr do
3:    Construir array de dimensión dim-T
4:    Transponer matriz cuadrada de dimensión dim-T
5:    Descomponer LU, matriz cuadrada de dimensión dim-T
6:    Resolver sistema de ecuaciones, dimensión dim-T
7:    Calcular error = norma(dim-T)
8: end while
9: { ! Resolver el problema de VELOCIDAD}
10: Transponer matriz cuadrada de dimensión dim-T
11: Resolver sistema de ecuaciones, dimensión dim-T
12: { ! Resolver el problema de ACELERACION}
13: Transponer matriz cuadrada de dimensión dim-T
14: Resolver sistema de ecuaciones, dimensión dim-T
15: { ! Resolver centro de masas del Terminal}
16: for all número de eslabones, numEslab do
17:    Multiplicar dos matrices cuadradas de dimensión dim-MM
18: end for
19: Construir array de dimensión dim-MM
20: Multiplicar dos matrices cuadradas de dimensión dim-MM
21: Construir array de dimensión dim-MM
22: Multiplicar dos matrices cuadradas de dimensión dim-MM
```

Siendo *steps* el número de iteraciones que finalmente realiza el bucle *while-endwhile*, entonces el coste teórico asociado a este algoritmo es el siguiente:

$O(steps * (3 * dim\text{-}T + dim\text{-}T^2 + dim\text{-}T^3) + 2 * (dim\text{-}T + dim\text{-}T^2) + numEslab * dim\text{-}MM^3 + 2 * (3 + dim\text{-}MM^3))$.

2.4.2. Solución de los pares Manivela-Barra

La plataforma de Stewart que estamos estudiando consta de seis grupos móviles compuestos por una Manivela y un Barra, cada uno identificado por su número de grupo, *ng*. Todos ellos actúan conjuntamente para desplazar el Terminal hasta la posición calculada en la sección anterior. El algoritmo 2.3 contiene la secuencia de cálculos necesarios para resolver el grupo cinemático de cualquiera de los conjuntos Manivela-Barra, siendo éste idéntico para todos ellos.

Algoritmo 2.3 Esquema de la resolución del grupo que representa a un Grupo Manivela-Barra en el software de control original

```
1: { ! Resolver el problema de POSICION}
2: while error >tolErr  do
3:    Construir array de dimensión dim-MB
4:    Transponer matriz cuadrada de dimensión dim-MB
5:    Descomponer LU, matriz cuadrada de dimensión dim-MB
6:    Resolver sistema de ecuaciones, dimensión dim-MB
7:    Calcular error = norma(dim-MB)
8: end while
9: { ! Resolver el problema de VELOCIDAD}
10: Construir array de dimensión dim-MB
11: Resolver sistema de ecuaciones, dimensión dim-MB
12: { ! Resolver el problema de ACELERACION}
13: Construir array de dimensión dim-MB
14: Resolver sistema de ecuaciones, dimensión dim-MB
```

Como vimos en el algoritmo 2.1, que regulaba el funcionamiento general del programa, esta subrutina se ejecuta tantas veces como grupos tengamos, `numGE`. A la entrada del procedimiento se le indica el número de grupo *ng*, lo que le permite cargar desde las variables de trabajo la información específica necesaria para resolver ese componente en particular.

Los cálculos realizados son los mismos que en el caso de la resolución de la cinemática del Terminal, pero cambiando el tamaño de las matrices y excluyendo el cálculo del centro de masas (líneas 16-22 del algoritmo 2.2). También se utiliza el parámetro `tolErr` para controlar la convergencia durante el proceso iterativo de resolver el sistema de ecuaciones en el problema de la posición (línea 2 del algoritmo 2.3), y se le asigna el mismo valor que el empleado para la resolución del Terminal.

Como hicimos en la subsección anterior, denominaremos *steps* al número de iteraciones del bucle *while-endwhile*. Además, como el software original fue diseñado con un enfoque puramente secuencial, el coste de los cálculos de cada grupo Manivela-Barra debe multiplicarse por el número total de grupos, `numGE`. Por tanto el coste teórico asociado al algoritmo es el siguiente:

$$O(numGE * steps * (3 * dim\text{-}T + dim\text{-}T^2 + dim\text{-}T^3 + 2 * (dim\text{-}T + dim\text{-}T^2))$$

2.5. Entorno de trabajo hardware y herramientas software

En esta sección se describe el marco de trabajo empleado durante la realización de esta Tesis de Máster, compuesto por las librerías y compiladores que han sido necesarios para el desarrollo del software, y las plataformas de hardware utilizadas para la realización de los experimentos.

Se han utilizados dos entornos, uno para el desarrollo y pruebas unitarias, y otro para los experimentos reales y análisis de rendimientos. El primero es un ordenador personal *multicore* equipado con todas las herramientas necesarias. El segundo entorno son dos sistemas heterogéneos situados en el laboratorio de Computación Científica y Programación Paralela de la Universidad de Murcia, y que pertenecen al cluster HETEROSOLAR [56]. Describimos a continuación las características de estas plataformas:

- **Entorno de ejecución JUPITER**: Multiprocesador con 32 GB de memoria compartida compuesto de dos hexa-cores (12 cores) Intel Xeon E5-2620 a 2.00GHz. Además dispone de seis GPUs, dos de ellas son NVIDIA Fermi Tesla C2075 con 5375 MBytes de Global Memory y 448 cores. Las otras cuatro se agrupan en dos tarjetas de dos dispositivos NVIDIA GeForce GTX 590 cada una, con 1536 MBytes de Global Memory y 512 CUDA cores.

- **Entorno de ejecución SATURNO**: Multiprocesador con 32 GB de memoria compartida con procesadores Intel Xeon E7530 a 1.87GHz y un total de 24 cores (4 hexa-cores). Dispone de una GPU Tesla K20c (basada en la arquitectura Kepler) con 4800 MBytes de memoria global y 2496 CUDA cores.

- **Entorno de desarrollo**: Portátil DELL equipado con procesador Intel©Core i5 de 4 núcleos, con 8 GB de memoria y una GPU NVIDIA GTX 9600 con 4096 MB GDDR5 de memoria global y 640 CUDA cores.

JUPITER Y SATURNO ejecutan un sistema operativo Linux (kernel 3.13.0-33-generic #58-Ubuntu) y disponen de la versión 7.5 de CUDA [66]. Incorporan el compilador FORTRAN de Intel© en su versión 17.0.1, compilación 20161005, que ofrece la posibilidad de paralelización mediante OpenMP, donde varios *threads* comparten la memoria del proceso que los crea.

El portátil DELL cuenta con un sistema operativo Linux [21] (kernel 4.4.0-78-generic #99-Ubuntu), CUDA 8.0 [65] y la versión 16.0.4, compilación 20160811 del compilador FORTRAN de Intel©.

En cuanto a las librerías de álgebra matricial, si bien se ha tenido en cuenta el formato disperso y simétrico de las matrices que surgen en el modelado de la plataforma de Stewart, también se ha querido trabajar con *solvers* densos para comparar los rendimientos. De esta manera, y siguiendo la metodología descrita en la sección 1.5, se han empleado las siguientes:

- MKL (Intel© Math Kernel Library), incorporada en la suite de programación *Intel Parallel Studio*, e incluye funciones de LAPACK optimizadas para el tratamiento de matrices densas en la familia de procesadores de Intel©.
- MKL PARDISO (Intel© Parallel Direct Sparse Solver Interface), para resolución de sistemas basados en matrices dispersas, también ofrecida en el propio compilador de Intel©.
- MA27, subrutina perteneciente a la HSL (Harwell Subroutine Library) que realiza la resolución de sistemas de ecuaciones dispersas y simétricas.
- MAGMA, indicada para entornos heterogéneos. Emplea funciones de CUDA para el traspaso de información entre la CPU y las GPUs, por lo que es necesario tener instalado previamente el entorno de desarrollo de NVIDIA.

Capítulo 3

Simulador

El objetivo de esta Tesis de Máster es modificar el software de control de la Plataforma de Stewart, creado inicialmente para una ejecución secuencial, aplicándole técnicas de paralelismo y comprobando posteriormente el impacto en el rendimiento (tiempo de ejecución) según el tamaño del problema.

Una vez realizado el análisis de la versión original, se pudieron identificar varios parámetros para controlar la ejecución del software de control del robot paralelo, y que podemos ver recogidos en la tabla 3.1.

parámetro	descripción
`tFin`	tiempo máximo de ejecución del software de control
`tSalto`	incremento de tiempo entre cada iteración
`tolErr`	error admitido en los cálculos del problema de la posición

Tabla 3.1: Parámetros de ejecución del software de control de la Plataforma de Stewart, disponibles en su versión original

Además se observó que el tamaño del problema se especificaba utilizando las constantes mostradas en la tabla 3.2.

constante	descripción
`numGE=6`	número de grupos Manivela-Barra
`dim-T=12`	dimensión de la matriz que representa el terminal
`dim-MB=15`	dimensión de las matrices que representan a los grupos Manivela-Barra
`dim-MM=3`	dimensión de las matrices que representan el centro de masas del Terminal

Tabla 3.2: Constantes que especifican el tamaño del problema de la Plataforma de Stewart, disponibles en su versión original

3.1. Funcionalidades

De cara a crear un entorno de laboratorio orientado a las pruebas decidimos desarrollar un simulador. Este es un nuevo software que contiene las mismas tareas y, por tanto, la misma carga computacional que el original.

El simulador, desarrollado en FORTRAN, incorpora una serie de funcionalidades adicionales que le permiten gestionar la ejecución de un amplio conjunto de pruebas:

- Generar matrices con valores aleatorios para alimentar los cálculos que se ejecutan en el análisis de la cinemática de los sistemas multicuerpo.
- Trabajar con matrices simétricas, tanto densas como dispersas.
- Permitir, mediante parámetros, modificar el tamaño del problema, en concreto, el número de grupos Manivela-Barra y la dimensión de las matrices.
- Incorporar nuevas librerías de álgebra matricial para la resolución de sistemas de ecuaciones y decidir, mediante parámetros, cuales de ellas usar.
- En sus versiones paralelas, admitir tantos parámetros como sean necesarios para introducir cambios en la configuración y de esta manera poder comparar las variaciones en los tiempos de ejecución.
- Incorporar un registro de tiempos de ejecución que pueda almacenarse en fichero para facilitar un análisis posterior.
- Poder especificar el número de muestreos a realizar para calcular tiempos medios y poder descartar valores mínimos y máximos.
- Ser portable a sistemas de cómputo de alto rendimiento, como los disponibles en el clúster HETEROSOLAR de la Universidad de Murcia.

Para especificar una simulación necesitamos poder indicar el tamaño del problema. Para ello introducimos el concepto de *escenario*. Un escenario incluye las dimensiones y tipo de las matrices, el número de grupos estructurales que constituyen el sistema multicuerpo y el número de iteraciones estimadas que, en función del tamaño de las matrices, serán necesarias para resolver los sistemas de ecuaciones. Esta información se almacena en un fichero de texto que debe incluir los campos que se muestran en la tabla 3.3. Se permite incluir en un mismo fichero más de un escenario y realizar varias simulaciones en modo batch.

En el software de control original, la duración de la ejecución la determinaban la variable `tFin` junto al número de iteraciones en el bucle Newton-Raphson (que dependía a su vez del valor de `tSalto` y del tamaño de las matrices). Esta dependencia se podría obviar considerando `tSalto` bajo (del orden de 0.1-0.01 para el incremento en el valor de cualquier coordenada independiente).

campo	descripción
`tfin`	Número de ejecuciones
`tfin2`	Número de iteraciones para resolver el problema de la posición
`numGE`	Número de Grupos Manivela-Barra
`nEQTerminal`	Tamaño de la matriz del Terminal
`nEQManivela`	Tamaño de la matriz de los grupos Manivela-Barra
`nEQMatMul`	Tamaño de la matriz de puntos de interés adicionales
`neslabones`	Número de eslabones del mecanismo
`sparsity`	Dispersión de las matrices (expresada en %)
`symmetric`	Tipo de matrices (0: no simétricas, 1: simétricas)

Tabla 3.3: Contenido de los escenarios que ejecuta el simulador de la Plataforma de Stewart

En el simulador, en cambio, se maneja únicamente el valor de `tfin` para indicar el número de ejecuciones a realizar, ya que lo que interesa en este trabajo es obtener el tiempo empleado por el simulador en realizar `tfin` cálculos. El algoritmo 3.1 refleja el uso de este campo (línea 4) y la inclusión del proceso de lectura de los escenarios (línea 1) al principio de la ejecución.

Algoritmo 3.1 Esquema general del simulador de la Plataforma de Stewart

```
1:  Lectura del fichero de escenarios
2:  for all escenarios encontrados en el archivo de escenarios (numEscenarios) do
3:      Rellenar las matrices con datos aleatorios según escenarios
4:      for número de iteraciones (tfin) do
5:          Resolver la cinemática del Terminal, de dimensión nEQTerminal
6:          for all Manivela-Barra, (numGE) do
7:              Resolver la cinemática del par Manivela-Barra, de dimensión nEQManivela
8:          end for
9:      end for
10:     Almacenar en disco el tiempo de ejecución obtenido
11: end for
```

Por otro lado, dado que el simulador trabaja con matrices alimentadas con datos aleatorios, carece de sentido la constante que controlaba en el software de control original la tolerancia del error en el método iterativo Newton-Raphson, `tolErr`. En su lugar se define un nuevo campo del escenario, `tfin2`, que permite indicar el número de veces que se repetirá la resolución de los sistemas de ecuaciones, como podemos observar en la línea 2 del algoritmo 3.2. Este algoritmo es válido tanto para el cálculo del Terminal como para los grupos Manivelas-Barra, con la diferencia del tamaño de las matrices que se manejan en cada caso, `nEQTerminal` para el Terminal y `nEQManivela` para los grupos Manivela-Barra.

Algoritmo 3.2 Esquema de la resolución del problema de la posición para el Terminal y los grupos Manivelas-Barra en el simulador

1: { ! Resolver el problema de POSICION}
2: **for** número de iteraciones, (`tfin2`) **do**
3: Construir array de dimensión `nEQTerminal` o `nEQManivela`
4: Transponer matriz cuadrada de dimensión `nEQTerminal` o `nEQManivela`
5: Descomponer LU, matriz cuadrada de dimensión `nEQTerminal` o `nEQManivela`
6: Resolver sistema de ecuaciones, dimensión `nEQTerminal` o `nEQManivela`
7: Calcular error = norma(`nEQTerminal` o `nEQManivela`)
8: **end for**

Aparte de los escenarios, que permiten especificar el tamaño del problema, se quieren gestionar las opciones de ejecución del simulador sin necesidad de recompilar el código. Para ello se usarán parámetros, que se podrán proporcionar como argumentos en las llamadas al programa. La primera versión del simulador gestiona los mostrados en la tabla 3.4.

parámetro	descripción
`ArgNumTests`	número de muestreo de tiempos de cada ejecución
`ArgLibrary`	librería de álgebra matricial a emplear en los cálculos

Tabla 3.4: Parámetros de ejecución disponibles en la primera versión del simulador de la Plataforma de Stewart

El parámetro `ArgLibrary` indica al simulador qué librería utilizar en sus cálculos. En la tabla 3.5 se muestran los valores admitidos en la primera versión del simulador.

valor	descripción
1	MKL
2	PARDISO
3	MA27

Tabla 3.5: Valores que admite el parámetro `ArgLibrary`: librerías de álgebra lineal en la primera versión del simulador de la Plataforma de Stewart

Este parámetro puede contener un sólo valor o una lista de ellos. Por ejemplo, si `ArgLibrary`=1, los cálculos se realizarán con MKL. Con `ArgLibrary`=13 se realizarán dos simulaciones, una con MKL (`ArgLibrary`=1) y otra con MA27 (`ArgLibrary`=3).

Por tanto, los parámetros permiten especificar las condiciones de ejecución del simulador, como son el número de muestreos y las librerías empleadas.

3.2. Log de resultados

Los tiempos de ejecución que se obtienen en cada simulación se graban en un fichero de texto plano en formato CSV (comma-separated values), lo que facilitará su importación en otras aplicaciones para su posterior análisis. Cada línea del archivo representa una ejecución, y contiene toda la información que identifica los parámetros y el escenario empleados. La tabla 3.6 muestra la información almacenada de cada ejecución.

campo	descripción
`hora`	hora de finalización de la simulación (hh:mm:ss.ms)
`tfin`	número de veces que se simula cada escenario
`tfin2`	iteraciones en la resolución del sistemas de ecuaciones
`numGE`	número de grupos Manivela-Barra
`nEQTerminal`	tamaño de la matriz del Terminal
`nEQManivela`	tamaño de la matriz de los grupos Manivela-Barra
`nEQMatMul`	tamaño de la matriz de puntos de interés adicionales
`neslabones`	número de eslabones del mecanismo
`librería`	identificador de la librería de álgebra matricial empleada
`dispTerminal`	dispersión de la matriz de Terminal
`dispManivela`	dispersión de la matriz de los grupos Manivela-Barra
`resu-1`	tiempos de ejecución obtenidos
`resu-2`	
...	
`resu-ArgNumTests`	

Tabla 3.6: Contenido del archivo de resultados generado por el simulador

3.3. Generación de matrices aleatorias

Como se mencionó en la sección 2.3, se ha comprobado que las matrices asociadas a las ecuaciones de restricción que se plantean en el análisis de la plataforma de Stewart presentan una estructura simétrica y con una proporción de valores nulos respecto al tamaño total de la matriz superior al 70 %. Además, tienen la particularidad de que los valores no nulos se encuentran concentrados alrededor de la diagonal, por lo que podrían considerarse matrices banda.

Con objeto de comparar el rendimiento de las librerías de álgebra lineal en un entorno lo más cercano posible a los escenarios reales, el simulador ha sido dotado de un generador de matrices con contenido aleatorio. Para ello, se emplean los ya mencionados campos `sparsity` y `symmetric`. El primero es un valor numérico que representa la proporción de valores nulos respecto al tamaño de la matriz (en %) y el segundo puede adoptar cualquiera de los valores mostrados en la tabla 3.7.

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

Tabla 3.7: Estilos de matrices manejados por el simulador

Se muestra a continuación el procedimiento que sigue el simulador para generar los tres tipos de matrices mencionados.

3.3.1. Matrices no simétricas

Si consideramos la generación de una matriz cualquiera no simétrica de tamaño $n \times n$, el proceso es el siguiente:

1. Se rellena la matriz aleatoriamente con valores no nulos.
2. Se eligen al azar $\texttt{sparsity} \cdot \dfrac{n^2}{100}$ elementos de la matriz y se establece su valor a 0.

3.3.2. Matrices banda simétricas

La matriz que se muestra a continuación es una matriz banda, donde $a_{ij} = a_{ji} \quad \forall i, j$, y además se cumple que $a_{ii} \neq 0 \quad \forall i$.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} \end{pmatrix}$$

El proceso seguido por el simulador para crear este tipo de matrices es el siguiente:

1. Rellena la matriz completa con valores 0.
2. Los elementos de la diagonal se reemplazan con valores aleatorios no nulos.
3. Se completan, a ambos lados de la diagonal y respetando la simetría, valores al azar no nulos hasta alcanzar el valor de dispersión indicado en el parámetro `sparsity`.

3.3.3. Matrices simétricas

La siguiente matriz es una matriz simétrica de tamaño $n \times n$, donde $a_{ij} = a_{ji} \quad \forall i, j$.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & a_{17} \\ a_{21} & 0 & a_{23} & a_{24} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & a_{35} & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & a_{45} & a_{46} & 0 \\ 0 & 0 & a_{53} & a_{54} & 0 & 0 & a_{57} \\ 0 & 0 & 0 & a_{64} & 0 & a_{66} & a_{67} \\ a_{71} & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} \end{pmatrix}$$

La creación de una matriz simétrica sigue el siguiente algoritmo:

1. Se rellena toda la matriz con valores aleatorios no nulos.
2. Se eligen $\texttt{sparsity} \cdot \frac{n^2}{100} \cdot \frac{1}{2}$ pares a_{ij}, a_{ji} y se les establece un valor de 0.

3.4. Reutilización de matrices aleatorias

Como se ha comentado anteriormente, el simulador puede ejecutar múltiples escenarios en la misma ejecución, registrando los tiempos empleados en cada caso.

Dado que el objeto es comparar el efecto que ejercen los diferentes parámetros de configuración del software sobre los tiempos de ejecución, la validez de las comparativas se basa en que los juegos de datos (las matrices) sean iguales. Por este motivo, cada vez que el simulador genera una matriz con valores aleatorios, la almacena en disco en formato binario en un archivo que tiene la siguiente nomenclatura:

matrix_{n-filas}_{n-columnas}_{sparsity}.dat

De esta manera, en la siguiente simulación el programa busca en disco una matriz generada previamente con la estructura requerida. Si la encuentra, la incorpora en memoria para usarla en los cálculos. En caso contrario la genera según se ha explicado en la sección anterior, y la graba en disco para que esté disponible en futuras simulaciones.

Capítulo 4

Estudio del rendimiento con librerías matriciales

Como primer objetivo de esta Tesis de Máster analizamos el rendimiento secuencial del software desarrollado por la Universidad Politécnica de Cartagena empleando el simulador en su versión inicial, es decir, explotando el paralelismo interno de las librerías, pero sin estudiar todavía la posibilidad de paralelizar algunas zonas del código.

Ejecutaremos experimentos con diferentes tamaños de problema (escenarios) y registraremos los resultados, los cuales servirán de referencia para calcular el *speed-up* de las siguientes versiones mejoradas del algoritmo.

Los escenarios incluyen tamaños de matrices que varían desde el que representa a una plataforma de Stewart real (con matrices de dimensión 12×12 para el Terminal y 15×15 para las Manivelas) hasta matrices grandes de tamaño 4000×4000, que pudieran representar a sistemas más complejos. Todos ellos presentan en común:

- `tfin` = 10 (las simulaciones realizan todos los cálculos 10 veces).
- `tfin2` = 3 (número de veces que se ejecuta la resolución del sistema de ecuaciones al resolver problema de la posición). Se ha comprobado experimentalmente en el software real que este tipo de problemas converge rápidamente a una solución en un máximo de 3 intentos.
- `numGE` = 6 (plataforma de Stewart estándar, con 6 grupos Manivela-Barra).
- `nEQMatMul` = 3 (tamaño de la matriz empleada para el cálculo de puntos de interés adicionales, como el centro de masas).
- `neslabones` = 3 (número de eslabones del mecanismo).

El simulador utiliza en su versión inicial la librería MKL (Math Kernel Library) de Intel©. Esta librería es capaz de aprovechar en sus cálculos el hardware multicore disponible con tan sólo añadir el parámetro de compilación *-mkl = parallel* y especificar en el código (o mediante la variable de entorno *MKL_NUM_THREADS*) los threads

que utilizarán las rutinas de MKL. Nosotros preferimos fijar el número de threads por código para añadir flexibilidad y la posibilidad de modificarlo durante la ejecución, por lo que añadimos un nuevo parámetro `ArgMKLThreads`. De esta manera, los parámetros manejados por el simulador en esta versión son tres, como vemos en la tabla 4.1.

parámetro	descripción
`ArgNumTests`	número de muestreo de tiempos de cada ejecución
`ArgLibrary`	librería de álgebra matricial a emplear en los cálculos
`ArgMKLThreads`	threads usados por MKL

Tabla 4.1: Parámetros disponibles en la versión del simulador de la Plataforma de Stewart que incluye el control del paralelismo de MKL

La simulación se ejecutará en la plataforma JUPITER (12 cores), que forma parte del cluster HETEROSOLAR de la Universidad de Murcia.

4.1. MKL con matrices densas

Para indicar al simulador que se va a trabajar con matrices densas, es necesario especificar en los escenarios los siguientes campos y valores:

- `sparsity` = 30 (se usarán matrices con un factor de dispersión del 30 %, por lo que se pueden considerar matrices densas).
- `symmetric` = 1 (se usarán matrices banda simétricas. Los tamaños vienen especificados en los escenarios).

Por otro lado, se realizará una única toma de tiempos, por lo que asignaremos al parámetro `ArgNumTests` el valor 1.

Dado que nuestro simulador no implementa en esta fase un algoritmo con paralelelismo explícito, todos los cores están disponibles para el paralelismo interno de las funciones MKL invocadas. Lanzaremos varias ejecuciones asignando al parámetro `ArgMKLThreads` los valores 1, 2, 3, 4, 6 y 12 para comprobar la influencia en los tiempos de ejecución del número de cores utilizados.

La simulación, con `tfin` = 10 y `tfin2` = 3 (10 iteraciones, resolviendo 3 veces los sistemas de ecuaciones en cada iteración) presenta los resultados recogidos en la tabla 4.2. Se puede observar la mejora de rendimiento debido a la paralelización de los cálculos matriciales de la librería MKL, siendo ésta especialmente significativa conforme aumenta el tamaño de las matrices. Para tamaños de matrices inferiores a 400×400, la sobrecarga que supone la creación, gestión y destrucción de threads hace que el aumento del número de éstos no suponga mejora en el rendimiento. Pero es a partir de ese tamaño cuando se incrementa el speed-up, llegando a un valor de 5.92x para el tamaño de matrices de 4000×4000 (figura 4.1).

nEQTerminal	nEQManivela	Secuencial	MKL-2	MKL-3	MKL-4	MKL-6	MKL-12
12	15	0.0088	0.0087	0.0090	0.0090	0.0090	**0.0079**
24	36	0.0124	0.0108	0.0080	0.0080	0.0079	**0.0078**
36	54	0.0193	0.0193	0.0146	0.0147	0.0144	**0.0142**
48	72	0.0262	0.0261	0.0265	0.0270	**0.0196**	0.0268
60	90	0.0398	0.0354	**0.0306**	0.0325	0.0314	0.0352
72	108	0.0438	0.0435	0.0408	0.0437	**0.0399**	0.0465
400	400	0.7657	0.5916	0.4848	0.4543	0.4397	**0.4130**
600	600	2.2405	1.5682	1.1540	0.9976	0.7913	**0.7884**
800	800	4.8492	3.2025	2.2902	1.9040	1.6059	**1.2348**
1000	1000	9.0185	5.8735	4.1229	3.3495	2.6848	**2.2646**
1500	1500	29.7641	18.0639	12.6177	9.9480	7.8841	**6.4505**
2000	2000	67.6850	37.8718	26.0821	20.4364	15.4668	**12.7995**
2500	2500	130.8400	71.8499	49.3655	38.4876	30.0402	**24.4705**
3000	3000	219.2437	118.6172	80.9747	62.9875	49.6130	**36.9491**
3500	3500	348.3112	187.8672	132.7195	100.5447	78.8915	**56.0393**
4000	4000	508.5546	269.4840	186.0264	142.8496	110.8931	**85.9765**

Tabla 4.2: Comparación de los tiempos de ejecución con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`), variando el número de hilos asignados a MKL. Matrices simétricas con dispersión del 30 %

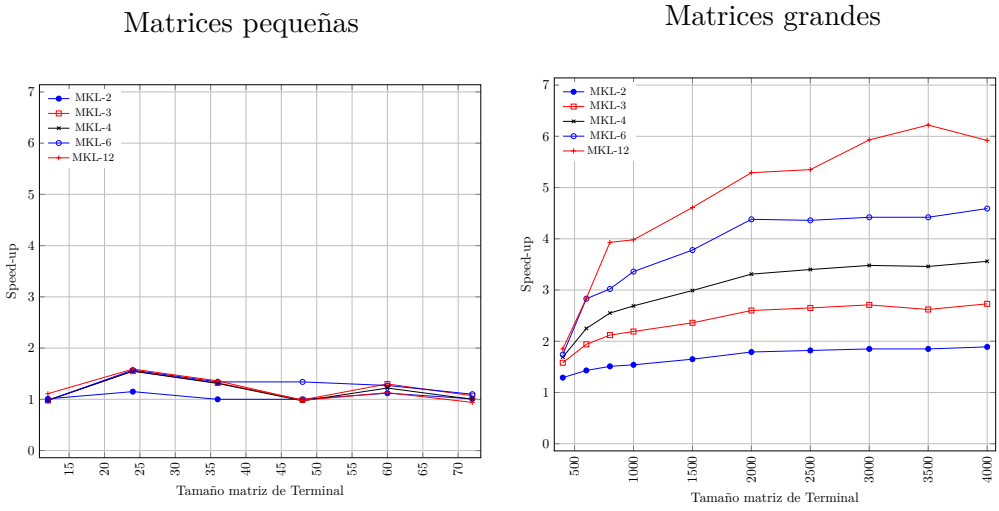


Figura 4.1: Speed-up respecto a MKL secuencial variando el número de hilos asignados a MKL, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Matrices simétricas densas con dispersión del 30 %

Como veremos a lo largo del presente trabajo, el paralelismo de MKL va a prevalecer sobre otro tipo de paralelismo cuando se trabaja con matrices de grandes dimensiones, demostrando un uso muy eficiente de los recursos del hardware.

4.2. MKL con matrices dispersas

Como se ha comentado con anterioridad, las matrices que resultan en el planteamiento de las ecuaciones de restricción son dispersas, con una proporción de valores nulos respecto al tamaño total de la matriz superior al 70 %. Por este motivo, es interesante repetir el experimento anterior, indicando al simulador que genere este tipo de matrices. Queremos comprobar si MKL presenta alguna mejora de rendimiento ligado a la presencia de valores cero en las matrices. Por tanto, los escenarios contendrán, junto al resto de información relativa al tamaño del problema, los siguientes valores:

- `sparsity` = 85 (seleccionamos matrices con un factor de dispersión del 85 %, por encima del 70 %, acorde a este tipo de problema).
- `symmetric` = 1 (se usarán matrices banda simétricas).

La simulación realiza 10 iteraciones y resuelve 3 veces los sistemas de ecuaciones en cada iteración (`tfin` = 10 y `tfin2` = 3).

Los resultados recogidos en la tabla 4.3 muestran la reducción de los tiempos de ejecución debido a la paralelización de los cálculos matriciales de la librería MKL. Los valores obtenidos son muy similares a los mostrados en la tabla 4.2 obtenidos al operar sobre matrices densas. Este resultado es el esperado dado que el solver denso de MKL considera los valores nulos como cualquier otro valor, sin obtener un beneficio adicional en los tiempos de ejecución debido a la dispersión.

En el caso particular de la plataforma de Stewart, donde se trabaja con matrices de tamaños 12×12 para el Terminal y 15×15 para los grupos Manivela-Barra, la ejecución secuencial ofrece mejor rendimiento que la paralela debido a la sobrecarga que supone la gestión de threads, que hace que su uso no sea adecuado en este tamaño de matrices. De hecho, hasta dimensiones de 72×72, el speed-up respecto a la versión secuencial se queda alrededor de 1.3x, pero se incrementa a partir de ese tamaño hasta llegar a un valor de 6.28x en el caso de 4000×4000 (figura 4.2). Intel© indica que la propia librería MKL decide en base al tamaño de las matrices el uso o no en sus cálculos de los cores asignados, e indica que 1000 es el tamaño a partir del cual se explota el multithreading [37], indistintamente de si se trabaja con matrices densas o dispersas.

nEQTerminal	nEQManivela	Secuencial	MKL-2	MKL-3	MKL-4	MKL-6	MKL-12
12	15	**0.0087**	0.0091	0.0090	0.0091	0.0088	0.0091
24	36	0.0130	0.0119	0.0122	0.0081	**0.0080**	0.0082
36	54	0.0206	0.0199	0.0182	0.0150	**0.0149**	0.0149
48	72	0.0270	0.0289	0.0270	0.0266	**0.0235**	0.0273
60	90	0.0387	0.0406	0.0368	0.0311	**0.0281**	0.0351
72	108	0.0497	0.0548	0.0404	0.0415	**0.0388**	0.0497
400	400	0.7518	0.5622	0.4612	0.4514	**0.3366**	0.4312
600	600	2.2202	1.5549	1.1167	0.9565	**0.7731**	0.8965
800	800	4.8385	3.1789	2.2843	1.8757	1.5814	**1.3945**
1000	1000	9.0309	5.8126	4.0693	3.3308	2.8099	**2.2937**
1500	1500	29.4405	17.9960	12.5406	9.7418	7.8384	**7.1244**
2000	2000	67.0978	37.6554	25.8041	20.1326	15.4426	**12.2056**
2500	2500	129.4520	71.2573	48.7372	38.0542	31.8765	**21.6673**
3000	3000	216.9713	117.7045	80.1862	62.4362	48.9069	**35.4511**
3500	3500	344.9432	186.1436	131.1206	99.4298	74.0594	**60.1485**
4000	4000	505.3326	266.4603	183.8802	142.3072	104.3095	**80.5172**

Tabla 4.3: Comparación de los tiempos de ejecución con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`), variando el número de hilos asignados a paralelismo de MKL. Matrices simétricas con dispersión del 85 %

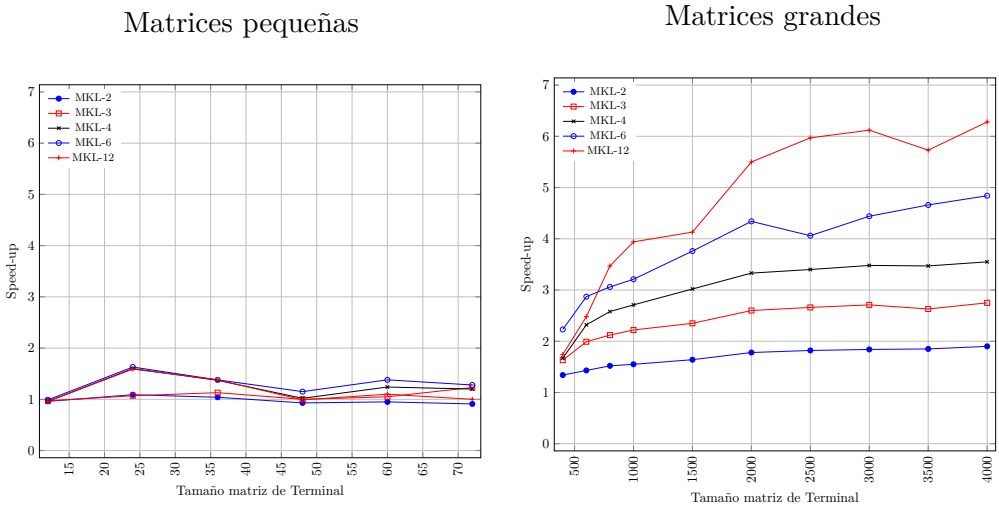


Figura 4.2: Speed-up respecto a MKL secuencial variando el número de hilos asignados a MKL, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Matrices simétricas con dispersión del 85 %

4.3. Librerías especializadas para matrices dispersas: MA27 y PARDISO en ejecución secuencial

Las matrices dispersas, por la particularidad de contener un elevado número de valores nulos, son objeto de un estudio especial. Como fruto de ello, se han elaborado métodos optimizados de almacenamiento en memoria y rutinas de álgebra lineal especializadas, con notables mejoras en los tiempos de ejecución respecto a las librerías diseñadas para matrices densas.

En esta línea, Intel© ofrece la librería *PARDISO (Parallel Direct Sparse Solver)* [18] para trabajar con matrices dispersas simétricas y no simétricas, y *HSL* (*Harwell Subroutine Library*) ha publicado la *MA27* [10], especializada en el manejo de matrices dispersas simétricas.

PARDISO, al igual que su variante MKL para matrices densas, está diseñada para aprovechar los cores disponibles en el hardware. Sin embargo, MA27 no dispone de versión paralela, por lo que su rendimiento sólo se podrá comparar con MKL y PARDISO en ejecuciones secuenciales.

Los escenarios que manejan matrices dispersas incluyen, junto al resto de campos que especifican el tamaño del problema, la siguiente información:

- `sparsity` = 85 (se usarán matrices con un factor de dispersión del 85 %, por encima del 70 %, acorde a este tipo de problema).
- `symmetric` = 1 (se usarán matrices banda simétricas).

En cada simulación realizaremos una única toma de tiempos (`ArgNumTests` = 1).

Una ejecución del simulador con `tfin` = 10 y `tfin2` = 3 obtiene los resultados mostrados en la tabla 4.4, donde se observa que, en ejecuciones secuenciales, la librería MA27 presenta el mejor rendimiento de las tres analizadas hasta tamaños de 1000×1000. En dimensiones mayores, PARDISO es la mejor opción, alcanzando un speed-up de 3.9x en tamaños de 4000×4000 respecto a MKL.

En la figura 4.3 se representan los speed-up respecto a MKL que se consiguen al usar PARDISO y MA27 a la hora de trabajar con matrices dispersas. Se observa el cambio de tendencia que se produce en el punto correspondiente a matrices de tamaño 1500×1500, donde PARDISO muestra su eficiencia. Se puede observar cómo, a pesar de que MKL está enfocada al manejo de matrices densas, cuando trabaja con matrices pequeñas, aunque éstas sean dispersas, sigue presentando mejor rendimiento que PARDISO. Este comportamiento es debido a que PARDISO realiza primero un análisis y una reordenación de la matriz que es necesaria para las etapas posteriores. Sin embargo, en matrices muy pequeñas, ésto supone un coste adicional de tiempo que no se llega a recuperar, a pesar del mejor rendimiento que ofrece esta librería en las fases de factorización y sustitución hacia atrás.

nEQTerminal	nEQManivela	MKL-Secuencial	PARDISO-Secuencial	MA27
12	15	0.0087	0.0329	**0.0022**
24	36	0.0130	0.0496	**0.0057**
36	54	0.0206	0.0720	**0.0104**
48	72	0.0270	0.1055	**0.0185**
60	90	0.0387	0.1199	**0.0272**
72	108	0.0497	0.1483	**0.0357**
400	400	0.7518	1.2535	**0.5167**
600	600	2.2202	2.9647	**1.3795**
800	800	4.8385	5.4637	**2.9086**
1000	1000	9.0309	9.5983	**5.4734**
1500	1500	29.4405	**15.7098**	15.9405
2000	2000	67.0978	**27.4114**	34.5328
2500	2500	129.4520	**43.3851**	65.3583
3000	3000	216.9713	**63.3465**	110.0977
3500	3500	344.9432	**86.3410**	174.2588
4000	4000	505.3326	**128.9157**	257.0304

Tabla 4.4: Comparación de los tiempos de ejecución ofrecidos por MKL, PARDISO y MA27, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`), en ejecución secuencial. Matrices simétricas con dispersión del 85 %

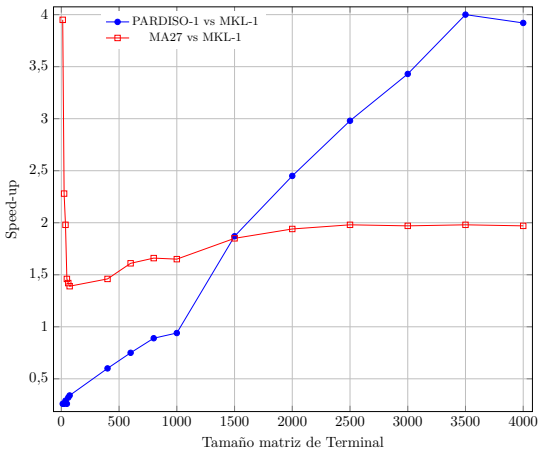


Figura 4.3: Speed-up respecto a MKL al usar PARDISO y MA27 en ejecuciones secuenciales, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Matrices simétricas con dispersión del 85 %

4.4. MKL y PARDISO en ejecución paralela, con matrices dispersas

Tanto MKL como PARDISO son librerías que ofrecen la posibilidad de trabajar en entornos multicore. En la sección 4.2 se analizó la evolución que mostraban los tiempos de ejecución cuando se incrementaban los threads asignados a MKL (tabla 4.3). En esta sección comparamos aquellos tiempos con los obtenidos al utilizar PARDISO asignando el mismo número de cores a ambas librerías, y que se recogen en la tabla 4.5.

Los experimentos se realizan con matrices simétricas con un factor de dispersión del 85 %, ya que son las asociadas a este tipo de problemas. La simulación realiza 10 iteraciones (`tfin` = 10) y resuelve 3 veces los sistemas de ecuaciones en cada iteración (`tfin2` = 3).

`nEQTerminal`	`nEQManivela`	Secuencial	PARD-2	PARD-3	PARD-4	PARD-6	PARD-12
12	15	**0.0397**	0.0502	0.0525	0.0549	0.0592	0.0669
24	36	**0.0482**	0.0722	0.082	0.0825	0.0812	0.0845
36	54	**0.0758**	0.1000	0.1128	0.1123	0.1041	0.0898
48	72	0.1068	**0.1043**	0.1462	0.1469	0.1294	0.1091
60	90	0.1197	0.1221	0.1386	0.1505	0.1361	**0.1136**
72	108	0.1516	0.1312	0.1642	0.1592	0.1611	**0.1302**
400	400	1.2492	1.0093	1.5373	**0.9705**	1.1421	1.2676
600	600	2.9952	2.3027	3.2334	1.9768	**1.9649**	2.4749
800	800	5.6708	4.1138	4.3469	3.5483	**3.3024**	3.6450
1000	1000	9.6694	6.5877	7.3793	5.6505	5.1878	**5.1537**
1500	1500	15.1254	10.6566	10.6437	**9.6423**	9.7869	9.7055
2000	2000	27.4223	19.9028	20.4683	17.0302	20.1799	**17.845**
2500	2500	40.9044	29.9075	29.7599	26.0681	**24.9722**	25.7476
3000	3000	59.9086	40.8671	40.2146	**34.5061**	35.5385	38.6832
3500	3500	83.0922	57.6812	57.4117	**48.4038**	49.3602	53.2995
4000	4000	126.6243	83.1700	82.7058	68.3780	**62.5897**	77.3253

Tabla 4.5: Comparación de los tiempos de ejecución con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`), variando el número de hilos asignados a paralelismo de PARDISO. Matrices simétricas con dispersión del 85 %

En la figura 4.4 se observa que PARDISO, para un mismo número de threads, ofrece menores tiempos de ejecución respecto a la versión MKL cuando se trabaja con matrices dispersas de gran tamaño, siendo este hecho más notable al aumentar la dimensión de las mismas. Sin embargo, hasta matrices de 1500×1500, lo que incluye al problema de la plataforma de Stewart, MKL se muestra más eficiente que PARDISO. Además, gestiona mejor el hecho de ir generando más threads internamente y, por tanto, usando más cores de la CPU. En la misma figura se puede comprobar que, trabajando con 6 threads y matrices grandes, PARDISO es tan sólo ligeramente más rápida, y que asignando 12 threads a ambas librerías éstas llegan a ofrecer unos tiempos de ejecución similares (speed-up de valor 1x).

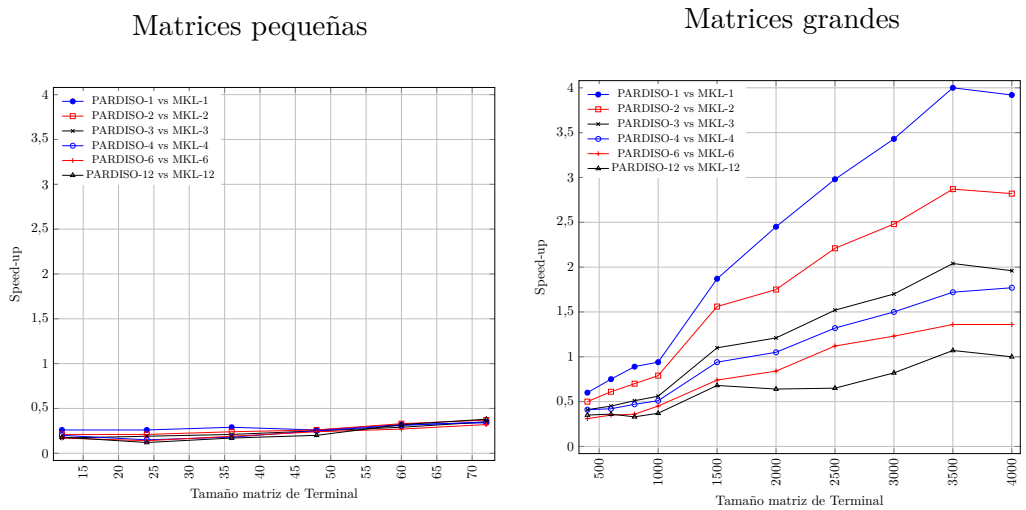


Figura 4.4: Speed-up empleando PARDISO respecto a MKL en modo paralelo, aumentando el número de hilos, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Matrices simétricas con dispersión del 85 %

Del algoritmo que implementa PARDISO para el manejo de matrices dispersas, que incluye etapas de reordenación, factorización y sustitución hacia atrás, la primera de ellas es la que está más optimizada para el paralelismo. Por tanto, el escalado de la librería para arquitecturas multiprocesador de memoria compartida no es tan notable como el del algoritmo para matrices densas que ejecuta MKL.

4.5. Selección de los mejores tiempos de ejecución obtenidos con matrices dispersas con librerías paralelizadas

La tabla 4.6 resume los mejores tiempos de ejecución obtenidos en JUPITER con MA27, MKL y PARDISO, sobre matrices con un factor de dispersión del 85 % y seis grupos estructurales. Las simulaciones realizan 10 iteraciones (`tfin` = 10) y resuelven 3 veces los sistemas de ecuaciones en cada iteración (`tfin2` = 3).

Se observa claramente cómo, en esta plataforma hardware concreta, MA27 realiza los cálculos de manera más rápida hasta tamaños de 72×72 y 108×108 en las matrices que representan al Terminal y a los grupos Manivela-Barra respectivamente. Para tamaños superiores, la posibilidad que ofrecen MKL y PARDISO de generar threads internamente comienza a hacerse patente. MKL, empleando 6 y 12 threads, es la más eficiente hasta tamaños 2500×2500 en ambas matrices. A partir de esa dimensión, es PARDISO, con su algoritmo optimizado para el manejo de matrices dispersas, la opción recomendada. Además, en ambos casos, el número de threads óptimo aumenta conforme lo hace el tamaño del problema a resolver, como era de esperar.

		MA27	MKL		PARDISO	
nEQTerminal	nEQManivela	Secuencial	6 threads	12 threads	4 threads	6 threads
12	15	**0.0022**	0.0088	0.0091	0.0549	0.0592
24	36	**0.0057**	0.0080	0.0082	0.0825	0.0812
36	54	**0.0104**	0.0149	0.0149	0.1123	0.1041
48	72	**0.0185**	0.0235	0.0273	0.1469	0.1294
60	90	**0.0272**	0.0281	0.0351	0.1505	0.1361
72	108	**0.0357**	0.0388	0.0497	0.1592	0.1611
400	400	0.5167	**0.3366**	0.4312	0.9705	1.1421
600	600	1.3795	**0.7731**	0.8965	1.9768	1.9649
800	800	2.9086	1.5814	**1.3945**	3.5483	3.3024
1000	1000	5.4734	2.8099	**2.2937**	5.6505	5.1878
1500	1500	15.9405	7.8384	**7.1244**	9.6423	9.7869
2000	2000	34.5328	15.4426	**12.2056**	17.0302	20.1799
2500	2500	65.3583	31.8765	**21.6673**	26.0681	24.9722
3000	3000	110.0977	48.9069	35.4511	**34.5061**	35.5385
3500	3500	174.2588	74.0594	60.1485	**48.4038**	49.3602
4000	4000	257.0304	104.3095	80.5172	68.3780	**62.5897**

Tabla 4.6: Comparación de los mejores tiempos de ejecución obtenidos con MKL, PARDISO y MA27, con y sin paralelismo interno de las librerías, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Matrices simétricas con dispersión del 85 %

4.6. Influencia del factor de dispersión de las matrices en el rendimiento de las librerías

Como se ha descrito en las secciones precedentes, la dispersión y el tamaño de las matrices son dos factores que influyen en la elección de la mejor librería cuando se quieren resolver problemas de álgebra lineal asociados a un sistema multicuerpo de un determinado tamaño.

En esta sección se realizan experimentos con matrices que presentan cuatro factores de dispersión (10 %, 30 %, 50 % y 85 %) y dimensiones que varían entre 12×12 y 3000×3000. El objetivo de los mismos es determinar qué combinación de tamaños y factores de dispersión pueden marcar la decisión de usar una u otra librería.

Los experimentos se ejecutan en la plataforma SATURNO (24 cores). La elección de una u otra plataforma no afecta a las conclusiones del estudio, ya que lo que se pretende es realizar una comparativa entre las tres librerías, MKL, PARDISO y MA27 cuando se ejecutan sobre un mismo hardware.

En la primera parte del estudio se prueban las librerías sin activar su paralelismo interno, es decir, en ejecución secuencial. En la segunda se asignan 24 threads a MKL y PARDISO, lo que coincide con el número de cores de la CPU. En ambos casos la simulación realiza 10 iteraciones (`tfin` = 10) y resuelve 3 veces los sistemas de ecuaciones en cada iteración (`tfin2` = 3).

4.6.1. Ejecución secuencial

La tabla 4.7 recoge los tiempos de ejecución conseguidos por cada librería en modo secuencial. A la vista de estos resultados podemos extraer la siguientes conclusiones:

- Con matrices muy pequeñas (tamaños apenas de 12×12), MA27 ofrece los mejores tiempos de ejecución sin importar cuál es el factor de dispersión, lo que indica que esta librería implementa un algoritmo muy eficiente para trabajar con matrices de tamaño reducido.
- Para factores de dispersión de hasta el 50 %, el solver denso de MKL ofrece el mejor rendimiento. A partir de ese valor se empieza a observar la conveniencia de emplear un solver disperso.
- Incluso con un factor del 50 %, PARDISO ya mejora los tiempos de MKL a partir de tamaños de matrices 3000×3000.
- Para factores de dispersión del 85 % siempre conviene emplear un solver disperso, MA27 hasta matrices de 1000×1000 y PARDISO a partir de este tamaño.

	10 % Dispersión			30 % Dispersión		
	MKL	Pard	MA27	MKL	Pard	MA27
12	0.0116	0.0610	**0.0060**	0.0135	0.0528	**0.0058**
36	**0.0247**	0.2430	0.0493	**0.0226**	0.2213	0.0476
60	**0.0512**	0.4691	0.1577	**0.0525**	0.3519	0.1462
400	**1.7438**	7.1319	9.0992	**1.7389**	7.7002	7.4483
1000	**21.4157**	55.6105	121.4862	**21.4917**	51.087	99.6011
2000	**158.3345**	262.5957	1221.5517	**157.3394**	202.4178	761.5917
3000	**503.5494**	826.5607	7191.0396	**509.8446**	548.4324	3159.8646
	50 % Dispersión			85 % Dispersión		
	MKL	Pard	MA27	MKL	Pard	MA27
12	0.0133	0.0540	**0.0048**	0.0135	0.0403	**0.0025**
36	**0.0242**	0.1887	0.0378	0.0226	0.0952	**0.0129**
60	**0.0504**	0.2746	0.1095	0.0517	0.1551	**0.0379**
400	**1.7306**	4.2830	4.8173	1.7255	2.0140	**0.9170**
1000	**21.254**	36.3638	59.1881	21.1857	15.1801	**9.2040**
2000	**158.9335**	169.5406	448.0367	153.175	**46.6182**	51.9908
3000	507.5378	**442.8351**	1487.6549	498.8115	**111.4345**	161.2812

Tabla 4.7: Tiempos de ejecución obtenidos empleando MKL, PARDISO y MA27 en modo secuencial, para diversos tamaños de matrices con dispersión entre 10 % y 85 % y 6 grupos estructurales Manivela-Barra

Las figuras 4.5 y 4.6 representan gráficamente la evolución de los tiempos de ejecución (representados en escala logarítmica) que se obtienen sobre matrices pequeñas y grandes, respectivamente.

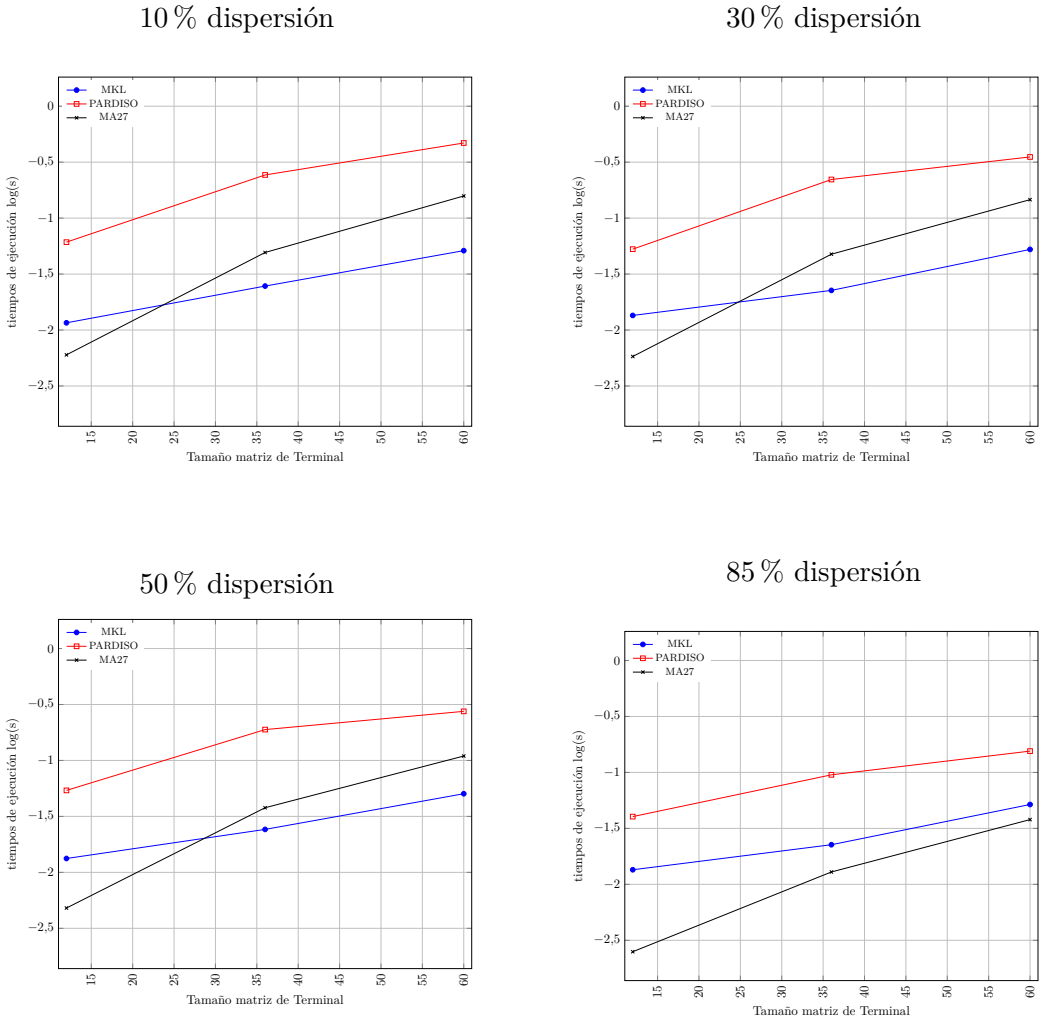


Figura 4.5: Tiempos de ejecución en escala logarítmica obtenidos empleando MKL, PARDISO y MA27 en modo secuencial, para diversos tamaños de matrices pequeñas con dispersión entre 10 % y 85 % y 6 grupos estructurales Manivela-Barra

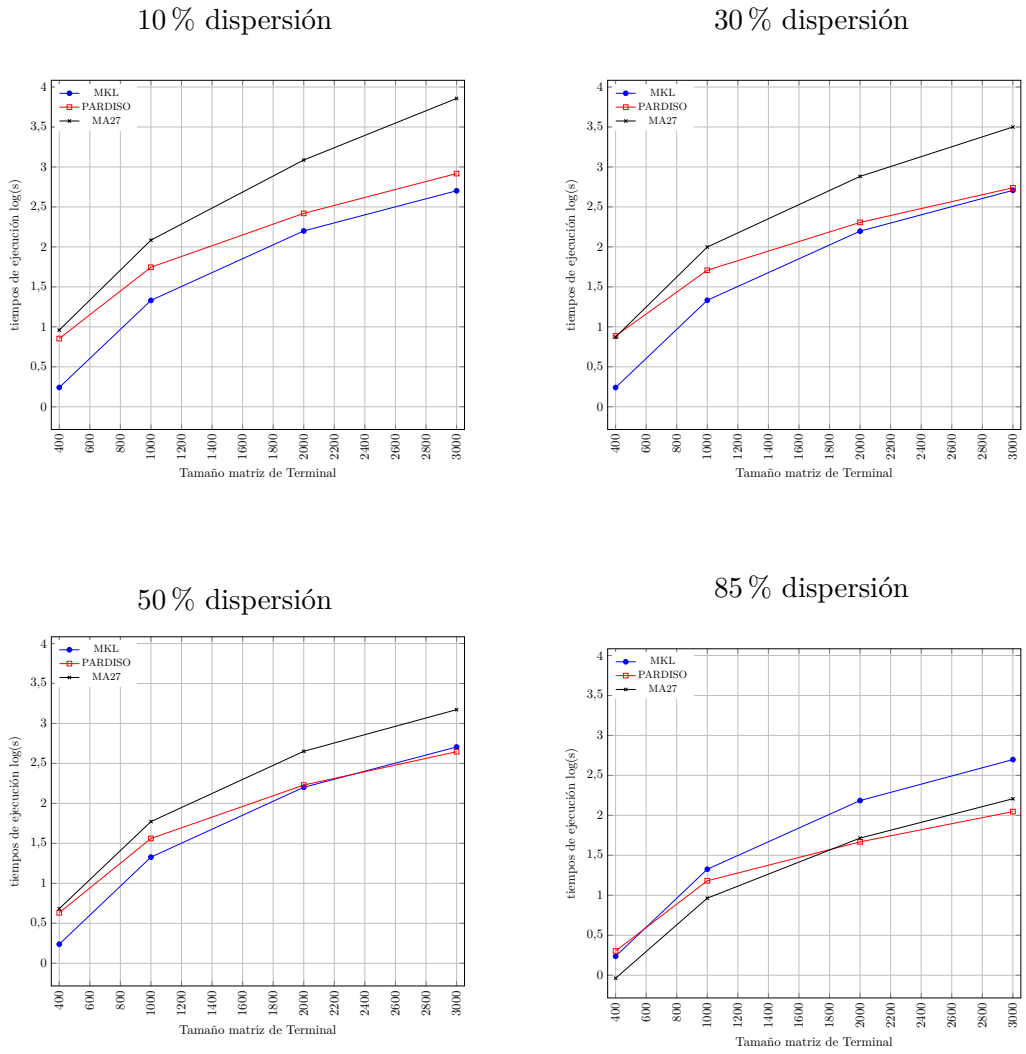


Figura 4.6: Tiempos de ejecución en escala logarítmica obtenidos empleando MKL, PARDISO y MA27 en modo secuencial, para diversos tamaños de matrices grandes con dispersión entre 10 % y 85 % y 6 grupos estructurales Manivela-Barra

4.6.2. Ejecución con librerías paralelizadas

A diferencia de MA27, MKL y PARDISO implementan algoritmos paralelos que son capaces de aprovechar el hardware multicore sobre el que se ejecutan, mejorando los tiempos de ejecución logrados en un sistema monocore. En este apartado se han repetido los experimentos realizados en el anterior, pero asignando 24 threads a MKL y PARDISO, coincidiendo con el número de cores de la CPU. La comparativa, que se muestra en la tabla 4.8, permite obtener las siguientes conclusiones:

- Se confirma que para matrices muy pequeñas (tamaños apenas de 12×12) MA27 ofrece los mejores tiempos de ejecución, a pesar de que no implementa un algoritmo paralelo. Este resultado se produce con independencia del factor de dispersión.
- Para factores de dispersión por debajo del 85 %, MKL presenta el mejor rendimiento, demostrando la eficacia de esta librería en entornos multicore.
- Para factores de dispersión del 85 %, MA27 es recomendable hasta matrices de 60×60. Observamos cómo, a diferencia de la ejecución secuencial donde MKL no obtenía los mejores tiempos en este factor de dispersión, al introducir el paralelismo interno, MKL es la librería más rápida en los tamaños hasta 2000×2000. A partir de entonces, se recomienda el uso de PARDISO.

	10 % Dispersión			30 % Dispersión		
	MKL	Pard	MA27	MKL	Pard	MA27
12	0.0135	0.5104	**0.0062**	0.0127	0.1308	**0.0056**
36	0.0540	0.3770	**0.0514**	**0.0472**	0.3649	0.0731
60	**0.0664**	0.5727	0.1569	**0.0707**	0.4781	0.2058
400	**0.8596**	5.9140	8.9641	**0.6269**	6.0670	6.9733
1000	**3.9570**	32.6937	122.9129	**3.7812**	29.2824	98.9719
2000	**22.9806**	144.6940	1241.6898	**23.2978**	104.3142	752.5696
3000	**61.2246**	325.4159	7521.6602	**62.5042**	218.6050	3198.8582
	50 % Dispersión			85 % Dispersión		
	MKL	Pard	MA27	MKL	Pard	MA27
12	0.0128	0.1432	**0.0075**	0.0138	0.1178	**0.0025**
36	**0.0331**	0.3194	0.0590	0.0498	0.1538	**0.0126**
60	**0.0646**	0.3738	0.1155	0.0540	0.2259	**0.0363**
400	**0.8582**	3.2105	4.8423	**0.8044**	2.1430	0.9119
1000	**4.4219**	18.8996	57.6605	**4.5575**	10.0356	8.1473
2000	**25.4048**	76.8835	443.9571	**22.6304**	28.1212	51.9470
3000	**57.4970**	174.9244	1467.3649	71.4087	**64.8255**	161.4604

Tabla 4.8: Tiempos de ejecución obtenidos empleando MKL, PARDISO y MA27 en modo paralelo con 24 threads, para diversos tamaños de matrices con dispersión entre 10 % y 85 % y 6 grupos estructurales Manivela-Barra

Este comportamiento se observa gráficamente en la figuras 4.7, donde se representan los tiempos de ejecución (en escala logarítmica) cuando se trabaja con matrices pequeñas. La figura 4.8 ofrece la misma información al trabajar con matrices grandes.

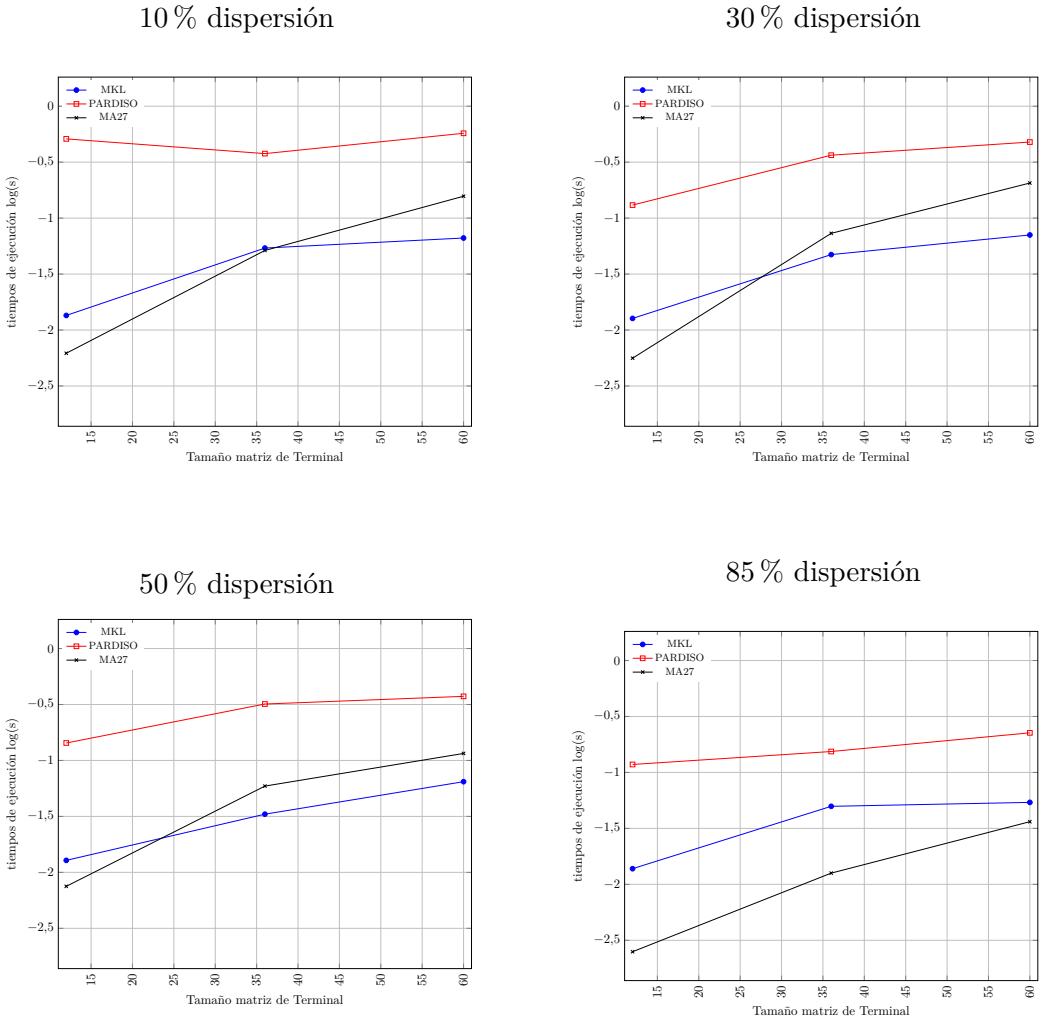


Figura 4.7: Tiempos de ejecución en escala logarítmica obtenidos empleando MKL, PARDISO y MA27 en modo paralelo con 24 threads, para diversos tamaños de matrices pequeñas con dispersión entre 10 % y 85 % y 6 grupos estructurales Manivela-Barra

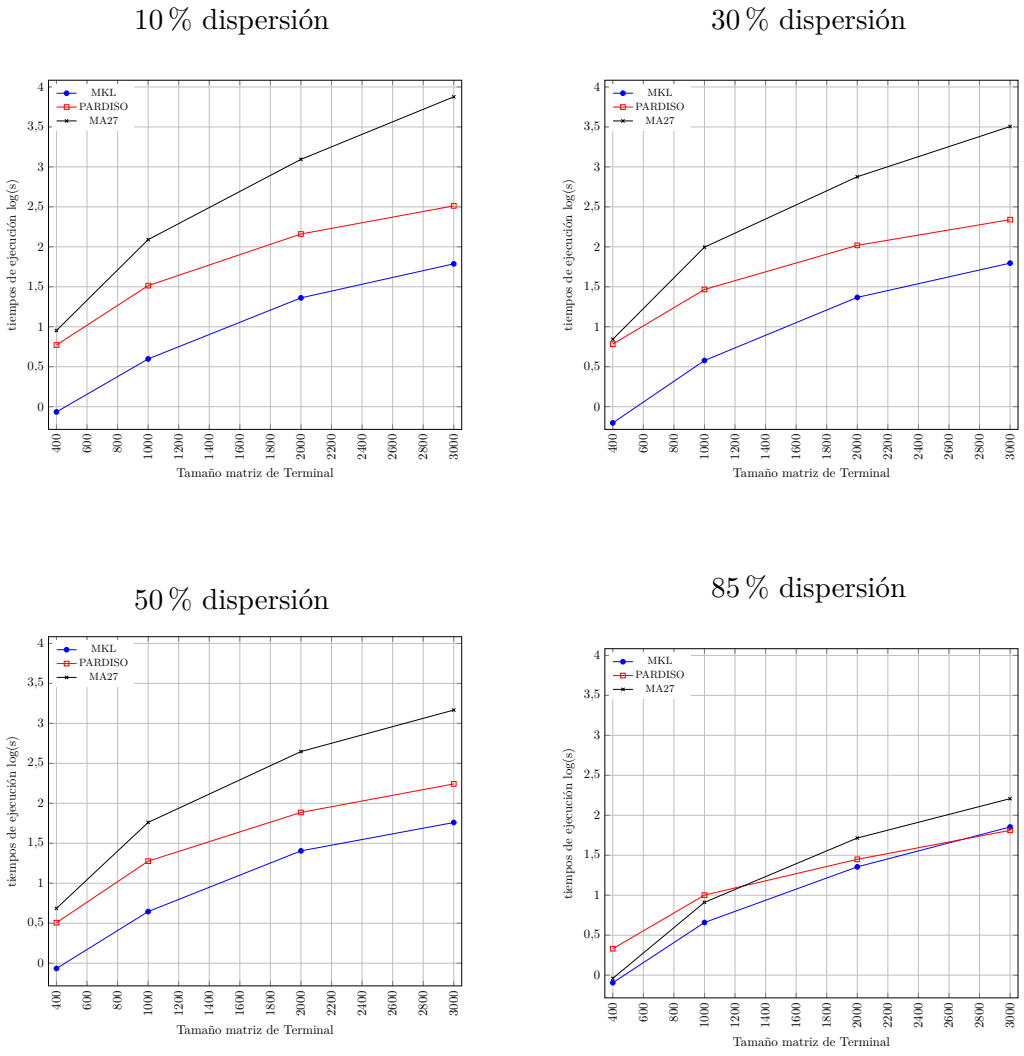


Figura 4.8: Tiempos de ejecución en escala logarítmica obtenidos empleando MKL, PARDISO y MA27 en modo paralelo asignando 24 threads, para diversos tamaños de matrices grandes con dispersión entre 10 % y 85 % y 6 grupos estructurales Manivela-Barra

Los resultados obtenidos en esta sección nos han permitido determinar qué librería obtiene el mejor tiempo de ejecución en función del tamaño de la matriz del Terminal y su factor de dispersión. La tabla 4.9 muestra esta información, distinguiendo los escenarios en los que la librería utiliza o no el paralelismo en su rutina. Las conclusiones que se derivan de estos experimentos son válidas en la plataforma SATURNO, donde se han llevado a cabo las ejecuciones, asignando 24 threads en el caso de activar el paralelismo interno de las librerías.

	En ejecución secuencial				Con paralelismo interno			
	Dispersión				Dispersión			
`nEQTerminal`	10 %	30 %	50 %	85 %	10 %	30 %	50 %	85 %
12	MA27	MA27	MA27	MA27	MA27	MA27	MA27	MA27
36	MKL	MKL	MKL	MA27	MA27	MKL	MKL	MA27
60	MKL	MKL	MKL	MA27	MKL	MKL	MKL	MA27
400	MKL	MKL	MKL	MA27	MKL	MKL	MKL	MKL
1000	MKL	MKL	MKL	MA27	MKL	MKL	MKL	MKL
2000	MKL	MKL	MKL	PARD	MKL	MKL	MKL	MKL
3000	MKL	MKL	PARD	PARD	MKL	MKL	MKL	PARD

Tabla 4.9: Selección de la librería más eficiente en función del tamaño y del factor de dispersión de la matriz del Terminal en la plataforma SATURNO, en ejecuciones secuenciales y con paralelismo interno asignando 24 threads

En el capítulo 7 se explica cómo un proceso de autotuning incluye una primera fase de búsqueda de las configuraciones óptimas del software de simulación para un determinado conjunto de escenarios. La información obtenida es utilizada por el propio software para ajustar los parámetros de ejecución en función del tamaño y naturaleza del problema a tratar. La librería a emplear en los cálculos es uno de esos parámetros ajustables en el caso del simulador de la plataforma de Stewart.

4.7. Conclusiones

Se puede concluir de manera genérica que, dado un problema a resolver en una plataforma dada, si queremos optimizar el tiempo de ejecución habría que decidir tanto la librería a utilizar (en este caso MKL, PARDISO o MA27), como el número de threads (cores de CPU) en base a las dimensiones del problema y dispersión de las matrices.

En programas diseñados con un enfoque puramente secuencial, sin paralelismo explícito, sólo podemos obtener mejoras de rendimiento usando librerías de álgebra matricial que implementen algoritmos paralelos en sus cálculos.

Utilizando el criterio del factor de dispersión de las matrices, los experimentos realizados en SATURNO con MKL, PARDISO y MA27 nos permitían concluir que, en esa plataforma concreta:

- Con factores de dispersión del 10 y el 30 %, MA27 es la librería con menores tiempo de ejecución cuando las matrices corresponden a las del problema de la plataforma de Stewart, con dimensiones de 12×12 y 15×15 para la matrices del Terminal y de los grupos Manivela-Barra, respectivamente. En todos los demás casos, MKL es la más rápida.

- Con factores de dispersión del 50 %, MA27 es también la librería más rápida hasta matrices de 12×12 y 15×15. MKL lo es en el resto de tamaños, excepto en ejecución secuencial, donde PARDISO la supera cuando los tamaños son muy grandes (3000×3000).

- Con factores de dispersión del 85 %, MA27 es la librería más rápida hasta matrices de 1000×1000 en ejecución secuencial y hasta 60×60 en ejecución paralela. A partir de esas dimensiones es preferible PARDISO en ejecución secuencial y MKL en ejecución paralela.

Ahora bien, dado que el problema de la plataforma de Stewart modelada en base a grupos estructurales obtiene matrices con un factor de dispersión superior al 70 %, nuestros experimentos se han realizado principalmente con matrices dispersas en un 85 %. El análisis de estos resultados nos permite extraer las siguientes conclusiones:

- En ejecuciones secuenciales de las librerías, o en entornos monocore, MA27 es la más rápida cuando se trabaja con matrices dispersas de tamaños hasta 1000×1000. Con dimensiones mayores, se recomienda PARDISO por sus menores tiempos de ejecución.

- En ejecuciones paralelas con matrices dispersas, MKL se muestra muy eficiente, pero es superado por PARDISO cuando la dimensión de las matrices es superior a 1500×1500.

Capítulo 5

Explotación del paralelismo en sistemas multicore

Como se analizó en la sección 2.2, el análisis estructural puede descomponer los sistemas mecánicos multicuerpo en unidades menores denominadas Grupos Estructurales. En ese caso es posible calcular cada grupo por separado siguiendo el orden en que se obtuvieron durante el análisis. En este trabajo interesan los mecanismos que contienen grupos que, siendo independientes entre sí, están en la misma etapa en el orden de ejecución, lo que permite calcularlos de manera simultánea.

La plataforma de Stewart presenta esta característica: los grupos Manivela-Barra son independientes entre ellos y pueden calcularse a la vez. Interesa, por tanto, abordar la tarea del análisis del código fuente desarrollado para el control del robot e intentar aplicar técnicas de programación paralela donde sea posible. De los entornos de paralelismo disponibles, comenzaremos con OpenMP [35], un estándar de programación en memoria compartida muy difundido y que nos va a permitir aprovechar todos los procesadores de la CPU.

Este paradigma se basa en que varios threads (hilos de ejecución del programa) pueden realizar la misma tarea a la vez, pero sobre un conjunto de datos distinto ubicado en la memoria del ordenador, que es compartida por todos ellos. Un primer análisis del código fuente permite identificar que la información relativa al modelo geométrico del sistema (datos sobre la geometría y orientación del modelo en el espacio) se almacenan en estructuras de memoria indexadas por su número de grupo estructural. Este diseño va a permitir abordar la tarea de asignar un thread al cálculo de cada grupo, pues cada uno podrá acceder a la zona de memoria que contiene la información que le es necesaria. Idealmente, si disponemos de tantos cores como grupos, la mejora de rendimiento debería estar cercana a ese valor, generando tantos threads como cores. Si, además, disponemos de un número mayor de cores, podremos utilizar paralelismo multinivel y asignar los cores restantes a la paralelización de MKL, como hicimos en el capítulo anterior.

OpenMP permite especificar el número de hilos que se pueden crear en paralelo.

Está permitido indicar un número de threads mayor que el número de cores disponibles. En este caso, se puede delegar en el sistema operativo la tarea de gestionar el encolado y posterior asignación a un procesador, o bien, como se citó en la sección 1.4.3, enlazar explícitamente cada thread OpenMP a una unidad de procesamiento física empleando el método conocido como *thread affinity*.

Por otro lado, el paralelismo anidado, donde cada thread puede dar lugar a su vez a generar un subconjunto nuevo de threads, se introdujo en OpenMP en la versión 2.5. Por ejemplo, vemos en la figura 5.1 un caso en que cada uno de los threads OpenMP lanza a su vez threads para MKL. El paralelismo anidado puede utilizarse estableciendo la variable de entorno *OMP_NESTED* como *TRUE* o llamando a la función *OMP_SET_NESTED()* y pasar el parámetro *TRUE* para habilitarlo. Además es posible especificar el nivel de paralelismo, es decir, el número máximo de regiones paralelas anidadas activas especificándolo en el parámetro de la llamada a la función *OMP_SET_MAX_ACTIVE_LEVELS*.

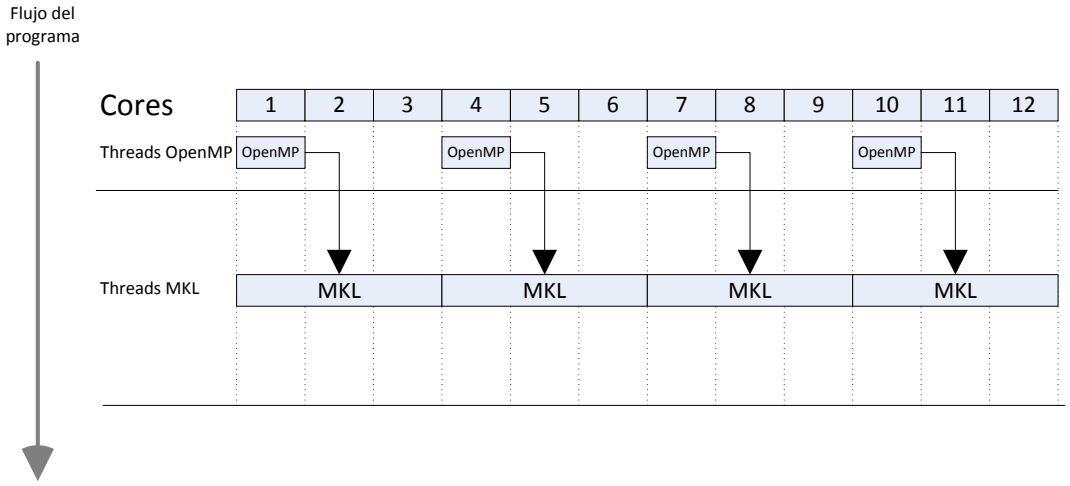


Figura 5.1: Paralelismo anidado OpenMP - MKL

Con objeto de probar el efecto de la variación en el número de threads respecto al número de grupos Manivela-Barra que se quieren resolver, se han añadido dos nuevos parámetros al simulador:

- `ArgThreads` para forzar el número de threads OpenMP.
- `ArgNestedLevel` para indicar el nivel de paralelismo.

La tabla 5.1 muestra la lista completa de parámetros que regulan el funcionamiento del simulador en su versión paralela con OpenMP.

parámetro	descripción
`ArgNumTests`	número de muestreo de tiempos de cada ejecución
`ArgLibrary`	librería de álgebra matricial a emplear en los cálculos
`ArgThreads`	número de threads disponible para paralelismo OpenMP
`ArgMKLThreads`	número de threads usados por MKL
`ArgNestedLevel`	nivel de paralelismo anidado

Tabla 5.1: Parámetros disponibles en el simulador de la plataforma de Stewart, incluyendo `ArgThreads` y `ArgNestedLevel` para el control del paralelismo OpenMP

Las líneas 7 y 12 del algoritmo 5.1 reflejan los cambios introducidos a la hora de crear una región paralela. Cuando la ejecución alcance la línea 7, se crearán *omp_num_threads* hilos, y la instrucción de la línea 8 hará que las iteraciones del bucle *for* se distribuyan entre todos ellos.

Algoritmo 5.1 Pseudocódigo del bucle principal en la versión OpenMP del simulador de la Plataforma de Stewart

```
1:  Lectura de los parámetros de ejecución
2:  for all escenarios encontrados en el archivo de escenarios (numEscenarios) do
3:    Rellenar las matrices con datos aleatorios según parámetros
4:    for número de iteraciones (tfin) do
5:      Resolver la cinemática del Terminal, de dimensión dim-T
6:      omp_num_threads ← ArgThreads
7:      !$OMP PARALLEL PRIVATE idGrupo
8:      !$OMP DO
9:      for all Manivela-Barra (numGE) do
10:       Resolver la cinemática de cada par Manivela-Barra, de dimensión dim-MB
11:     end for
12:     !$OMP END PARALLEL
13:   end for
14:   Guardar en disco el resultado (tiempo de ejecución)
15: end for
```

5.1. Paralelismo OpenMP con librerías en ejecución secuencial

Las primeras pruebas realizadas en este capítulo se enfocan en analizar la variación de rendimiento a medida que se incrementa el número de threads OpenMP generados para cada proceso, manteniendo 1 thread para los cálculos realizados por las librerías MKL, PARDISO y MA27 (es decir, forzando la ejecución secuencial en los algoritmos internos de las librerías).

Los experimentos se realizan con matrices simétricas dispersas con un factor del 85 %, ya que son las asociadas a este tipo de problemas. Recordamos que en la formulación de la plataforma de Stewart basada en grupos estructurales se obtienen matrices con un factor de dispersión siempre superior al 70 %. La simulación realiza 10 iteraciones y resuelve 3 veces los sistemas de ecuaciones en cada iteración (`tfin` = 10 y `tfin2` = 3).

La tabla 5.2 recoge los resultados obtenidos empleando MKL, donde se observa que la configuración que ofrece mejores resultados en la plataforma JUPITER con 12 cores corresponde a la asignación de 6 cores de la CPU al paralelismo OpenMP, generando por tanto 6 threads, lo que coincide con el número de grupos estructurales paralelizables. Podemos comprobar, observando las dos últimas columnas, que la asignación de más de 6 threads en este tamaño de problema no supone, por lo general, mejora en el rendimiento debido a que el bucle *for* (línea 9) del algoritmo 5.1 se limita a trabajar con el número de grupos Manivela-Barra, en este caso 6.

nEQ		Threads OpenMP - MKL secuencial					
Terminal	Manivela	Secuencial	2	3	4	6	12
12	15	0.0087	0.007	0.0075	**0.0066**	0.0073	0.0164
24	36	0.0130	0.0072	0.0052	0.0054	**0.0034**	0.0034
36	54	0.0206	0.0110	0.0097	0.0096	**0.006**	0.0061
48	72	0.0270	0.0198	0.0142	0.0142	**0.0088**	0.0088
60	90	0.0387	0.0313	0.0223	0.0224	0.0135	**0.0120**
72	108	0.0497	0.0431	0.0306	0.0215	**0.0153**	0.0185
400	400	0.7518	0.5109	0.4300	0.3479	**0.3097**	0.3590
600	600	2.2202	1.3027	1.7661	1.0417	**0.6846**	1.0190
800	800	4.8385	2.8444	3.2837	2.1897	**1.7082**	1.7581
1000	1000	9.0309	5.2699	4.0416	4.0405	**2.9625**	4.4596
1500	1500	29.4405	17.0992	13.1206	13.0631	**9.3136**	10.5032
2000	2000	67.0978	39.1350	29.8572	29.7066	**20.9699**	21.8229
2500	2500	129.452	74.6336	56.6841	56.9212	**39.3705**	39.5670
3000	3000	216.9713	125.2382	95.1161	95.3053	**65.6761**	67.8301
3500	3500	344.9432	198.1971	150.6014	150.0411	**103.9906**	112.5781
4000	4000	505.3326	290.3554	220.1784	220.4681	**151.7942**	158.0165

Tabla 5.2: Tiempos de ejecución obtenidos al incrementar el número de threads OpenMP, manteniendo `MKL` secuencial, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Matrices simétricas con dispersión del 85 %

Repitiendo los mismos experimentos con PARDISO y MA27 se obtienen los resultados reflejados en las tablas 5.3 y 5.4. En ellas se observa un comportamiento similar, siendo 6 el número de threads que ofrece menores tiempos de ejecución. Nótese que los rendimientos ofrecidos por estas librerías son mejores que los que se obtienen con MKL, que es lo esperado dado que las matrices empleadas en las pruebas son dispersas.

nEQ		Threads OpenMP - PARDISO Secuencial					
Terminal	Manivela	Secuencial	2	3	4	6	12
12	15	0.0329	0.0204	0.0153	0.0159	**0.0140**	0.0143
24	36	0.0496	0.0303	0.0233	0.0228	**0.0156**	0.0226
36	54	0.0720	0.0453	0.0354	0.0319	**0.0222**	0.0308
48	72	0.1055	0.0625	0.0466	0.0449	**0.0289**	0.0428
60	90	0.1199	0.0730	0.0558	0.0530	**0.0356**	0.0508
72	108	0.1483	0.0875	0.0644	0.0536	**0.0487**	0.0608
400	400	1.2535	0.7907	0.5820	0.5983	**0.4057**	0.5440
600	600	2.9647	1.6960	1.3339	1.3173	**0.9672**	1.1810
800	800	5.4637	3.1815	2.4151	2.4248	**1.7436**	1.7879
1000	1000	9.5983	5.5631	4.2122	4.2899	**2.9947**	3.9712
1500	1500	15.7098	9.0957	6.9240	6.9835	**4.9225**	5.2097
2000	2000	27.4114	16.2677	12.4095	12.4316	**8.7775**	9.2513
2500	2500	43.3851	25.5709	19.5446	19.7325	**13.9872**	15.4933
3000	3000	63.3465	37.9289	29.0187	29.1886	**20.3311**	21.5418
3500	3500	86.3410	51.2597	39.3974	39.0868	**27.6636**	27.9503
4000	4000	128.9157	73.4586	56.8918	56.9194	**40.1353**	42.1692

Tabla 5.3: Tiempos de ejecución obtenidos al incrementar el número de threads OpenMP, manteniendo PARDISO secuencial, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Matrices simétricas con dispersión del 85 %

nEQ		Threads OpenMP - MA27 Secuencial					
Terminal	Manivela	Secuencial	2	3	4	6	12
12	15	0.0022	0.0013	0.0011	0.0011	**0.0009**	0.0010
24	36	0.0057	0.0032	0.0024	0.0024	**0.0016**	0.0025
36	54	0.0104	0.0059	0.0041	**0.0041**	0.0043	0.0042
48	72	0.0185	0.0101	0.0072	0.0074	**0.0044**	0.0072
60	90	0.0272	0.0144	0.0106	0.0106	**0.0089**	0.0104
72	108	0.0357	0.0192	0.0139	0.0118	**0.0111**	0.0138
400	400	0.5167	0.2960	0.2229	0.2371	**0.1559**	0.2251
600	600	1.3795	0.7988	0.6062	0.6084	**0.4321**	0.5425
800	800	2.9086	1.6533	1.2685	1.2635	**0.9069**	0.9290
1000	1000	5.4734	3.1208	2.3557	2.3662	**1.6476**	1.9637
1500	1500	15.9405	9.0904	6.8839	6.8914	**4.7648**	5.0448
2000	2000	34.5328	19.744	14.8578	15.0483	**10.3387**	11.7121
2500	2500	65.3583	37.2259	28.3047	28.2100	**19.4689**	21.4548
3000	3000	110.0977	62.8878	47.8902	47.7553	**32.8752**	35.1146
3500	3500	174.2588	99.2299	75.4788	75.6315	**52.0639**	52.2403
4000	4000	257.0304	146.8244	110.9816	110.6690	**76.5881**	76.2213

Tabla 5.4: Tiempos de ejecución obtenidos al incrementar el número de threads OpenMP, manteniendo `MA27` secuencial, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Matrices simétricas con dispersión del 85 %

La figura 5.2 muestra el speed-up respecto a la ejecución secuencial que se obtiene usando las tres librerías en su mejor configuración (asignando 6 threads OpenMP). Todos son valores muy similares dado que la reducción en los tiempos de ejecución es producida exclusivamente por el paralelismo OpenMP. En el caso de MA27 no podremos obtener en ningún caso mejores rendimientos de los aquí mostrados dado que, como se comentó anteriormente, esta librería no cuenta con implementación paralela como ofrecen MKL y PARDISO, y que intentaremos explotar en la siguiente sección.

Comprobamos que el speed-up no llegar al valor teórico de 6x (que corresponde a la resolución simultánea de los 6 grupos Manivela-Barra usando los 6 threads). El motivo es que, como vimos en el capitulo 3, la simulación de la plataforma de Stewart se realiza en dos fases, en primer lugar el Terminal y posteriormente los grupos Manivela-Barra. Como solo es paralelizable el segundo bloque, y la carga computacional de ambos es la misma, el speed-up teórico es 3.5, como hemos podido comprobar experimentalmente.

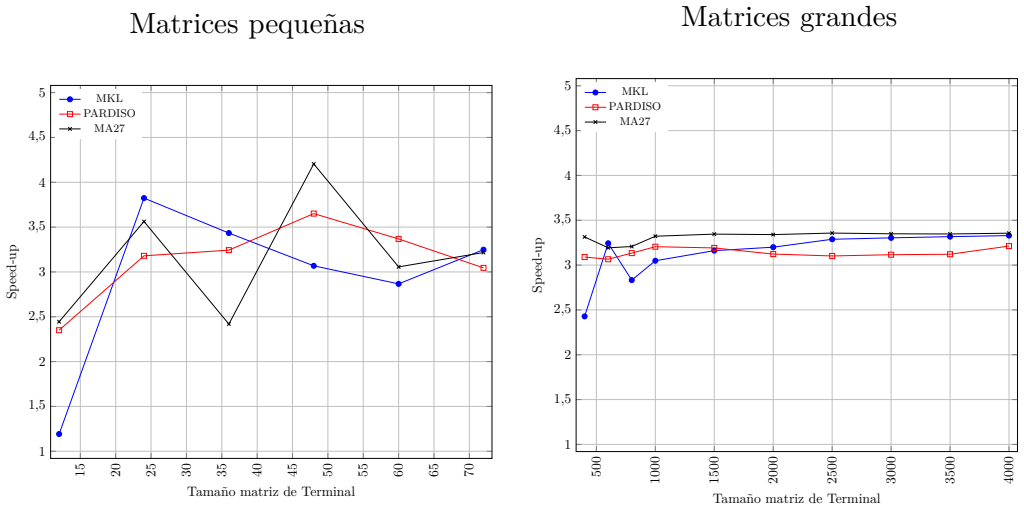


Figura 5.2: Speed-up respecto a la ejecución secuencial cuando se crean 6 threads OpenMP con MKL, PARDISO y MA27 sin paralelismo interno, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Matrices simétricas con dispersión del 85 %

5.2. Paralelismo en dos niveles

Como vimos en el capítulo anterior, MKL y PARDISO pueden, con implementaciones multihilo que lo permitan, usar varios cores simultáneamente para sus cálculos. Por tanto, cuando estas librerías son llamadas desde una región paralela OpenMP, intentarán generar nuevos threads dentro de los anteriores. Para que ésto sea posible, es necesario haber especificado un nivel de anidamiento de valor 2 o superior (donde el primer nivel será para OpenMP y el segundo para el paralelismo interno de las librerías).

Los experimentos de esta sección se han realizado de manera que los threads generados durante la ejecución no excedan el número de cores físicos de la CPU. La tabla 5.5 muestra las combinaciones `ArgThreads` × `ArgMKLThreads` que quedan por debajo de ese límite en una plataforma de 12 cores, como es el caso de JUPITER, perteneciente al cluster HETEROSOLAR.

threads generados	th. OpenMP × th. MKL					
2	1×2	2×1				
3	1×3	3×1				
4	1×4	2×2	4×1			
6	1×6	2×3	3×2	6×1		
12	1×12	2×6	3×4	4×3	6×2	12×1

Tabla 5.5: Combinaciones de threads OpenMP× MKL en un sistema de 12 cores

En esta sección se comparan las ejecuciones de MKL y PARDISO sobre matrices dispersas, variando el número de threads asignados a cada nivel, buscando la combinación que minimiza los tiempos de ejecución. Como referencia usaremos los resultados obtenidos en la ejecución secuencial pura y en el mejor escenario OpenMP, que hemos visto que en el caso de 6 grupos estructurales es la asignación de 6 threads. Se compararán también los resultados con los obtenidos al emplear la librería MA27 en modo secuencial, ya que ésta no dispone de una versión paralela.

Como en los experimentos anteriores, usaremos matrices simétricas dispersas con un factor del 85 %, ya que son las asociadas a este tipo de problemas. La simulación realiza 10 iteraciones y resuelve 3 veces los sistemas de ecuaciones en cada iteración (`tfin` = 10 y `tfin2` = 3).

5.2.1. Experimentos en JUPITER: 12 cores

En este caso se han probado las combinaciones de threads OpenMP que varían desde 1 hasta 12, manteniendo secuenciales las librerías, y también las combinaciones de threads que igualan los 12 cores, es decir, 1×12, 2×6, 3×4, 4×3 y 6×2.

Observando la tabla 5.6 vemos que la mejor combinación de paralelismo anidado OpenMP × MKL es aquélla que alcanza el aprovechamiento de todos los cores de la CPU en la proporción 3×4 en el caso de matrices hasta 400×400, y 2×6 para las matrices mayores. Esta última observación está en concordancia con el comportamiento observado en las secciones 4.1 y 4.2, donde se pudo comprobar, al trabajar con matrices de gran tamaño, que MKL ofrece un algoritmo muy optimizado para explotar el paralelismo. La figura 5.3 representa gráficamente los speed-up logrados con cada configuración.

nEQ		MKL	th. OMP × th. MKL			
Terminal	Manivela	Secuencial	6 × 1	6 × 2	2 × 6	3 × 4
12	15	0.0087	0.0073	0.0082	0.0077	**0.0064**
36	54	0.0206	0.0060	0.0062	0.0133	**0.0059**
60	90	0.0387	0.0135	0.0153	0.0244	**0.0122**
400	400	0.7518	0.3097	0.2518	0.2862	**0.2361**
1000	1000	9.0309	2.9625	2.0108	**1.8331**	2.0624
2000	2000	67.0978	20.9699	13.8333	**10.2912**	11.4366
3000	3000	216.9713	65.6761	43.0783	**34.3508**	41.6309
4000	4000	505.3326	151.7942	90.8851	**68.6536**	76.7721

Tabla 5.6: Tiempos de ejecución obtenidos al combinar threads OpenMP y threads MKL, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores. Matrices simétricas con dispersión del 85 %

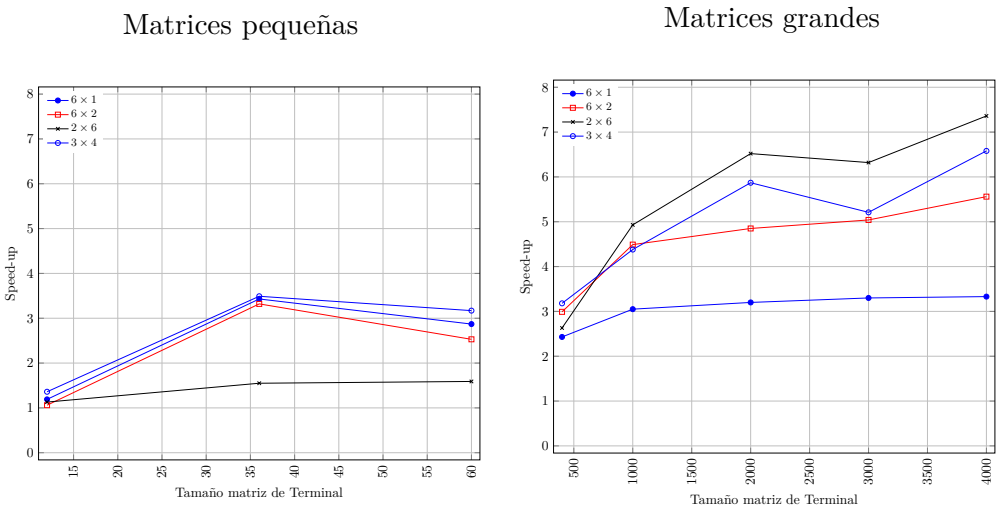


Figura 5.3: Speed-up respecto a la ejecución secuencial usando varias combinaciones de threads OpenMP × MKL, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Hardware con 12 cores. Matrices simétricas con dispersión del 85 %

Los mismos experimentos realizados con PARDISO arrojan los resultados recogidos en la tabla 5.7. Confirmando las conclusiones extraídas en la sección 4.4 sobre su versión paralela, el beneficio de asignar threads a los procesos internos de esta librería no es tan notable como en el caso de MKL. Los resultado muestran que es preferible reservar más threads a OpenMP que a la propia librería. Para matrices de tamaños hasta 400×400, es prácticamente indistinto asignar o no threads a PARDISO y, a partir de 1000×1000, la mejor opción es la de 6 threads OpenMP $\times$ 2 threads PARDISO. La figura 5.4 representa los speed-up conseguidos con las combinaciones comentadas.

nEQ		PARDISO	th. OMP $\times$ th. PARDISO			
Terminal	Manivela	Secuencial	6×1	6×2	2×6	3×4
12	15	0.0329	**0.0140**	0.0234	0.0493	0.0348
36	54	0.0720	**0.0222**	0.0288	0.0747	0.0406
60	90	0.1199	**0.0356**	0.0430	0.0866	0.0566
400	400	1.2535	**0.4057**	0.4945	0.8013	0.5528
1000	1000	9.5983	2.9947	**2.4988**	3.8037	3.4885
2000	2000	27.4114	8.7775	**7.7590**	11.779	9.8063
3000	3000	63.3465	20.3311	**16.732**	26.0609	20.5956
4000	4000	128.9157	40.1353	**31.6888**	48.5558	39.9993

Tabla 5.7: Tiempos de ejecución obtenidos al combinar threads OpenMP y threads PARDISO, varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores. Matrices simétricas con dispersión del 85 %

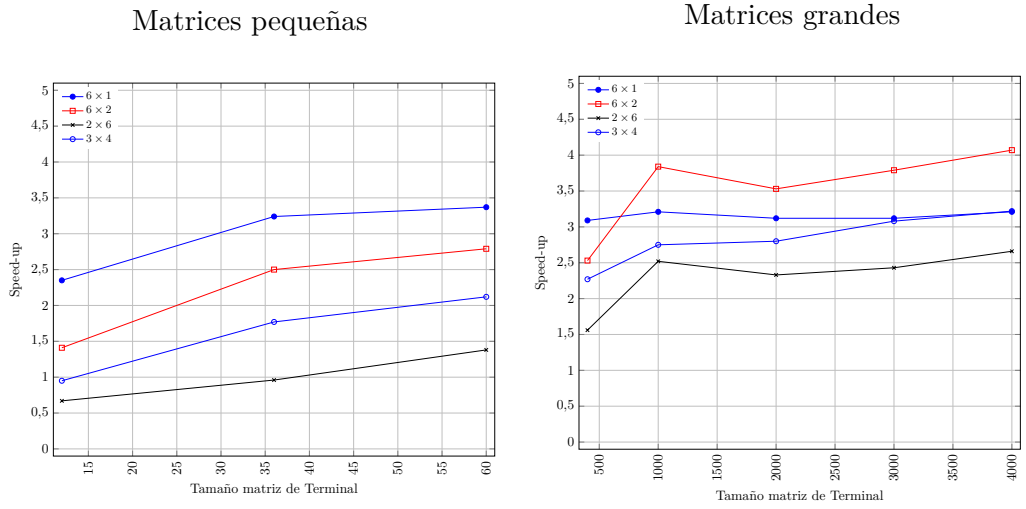


Figura 5.4: Speed-up respecto a la ejecución secuencial usando varias combinaciones de threads OpenMP $\times$ PARDISO, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Hardware con 12 cores. Matrices simétricas con dispersión del 85 %

5.2.2. Experimentos en SATURNO: 24 cores

En esta subsección determinaremos si las conclusiones extraídas tras las ejecuciones en JUPITER son igualmente válidas cuando los experimentos se realizan en una plataforma distinta, como es el caso de SATURNO con 24 cores. Para ello se han probado las combinaciones de threads OpenMP que varían desde 1 hasta 24, manteniendo secuenciales las librerías, y también las combinaciones que igualan los 24 cores, es decir, 1×24, 2×12, 3×8, 4×6, 6×4, 8×3 y 12×2. Los mejores resultados usando MKL se muestran en la tabla 5.8 y confirman que, con matrices grandes, el paralelismo más ventajoso es el de la propia librería, frente al de OpenMP (2 threads OpenMP $\times$ 12 threads MKL en este caso). Los speed-ups se pueden consultar en la figura 5.5.

nEQ		MKL	th. OMP $\times$ th. MKL			
Terminal	Manivela	Secuencial	2×12	8×1	6×4	12×1
12	15	0.0135	**0.011**	0.0124	0.0137	0.0148
36	54	0.0226	0.0288	**0.0072**	0.017	0.0073
60	90	0.0517	0.047	**0.0109**	0.0192	0.0109
400	400	1.7255	0.5717	0.6183	**0.3533**	0.5165
1000	1000	21.1857	**3.3315**	6.8866	3.3407	6.9969
2000	2000	153.175	**15.0134**	52.7357	19.9843	50.6002
3000	3000	498.8115	**42.9624**	160.4282	59.2876	162.4827
4000	4000	1172.057	**104.4767**	360.5952	123.1736	362.2536

Tabla 5.8: Tiempos de ejecución obtenidos al combinar threads OpenMP y threads MKL, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 24 cores. Matrices simétricas con dispersión del 85 %

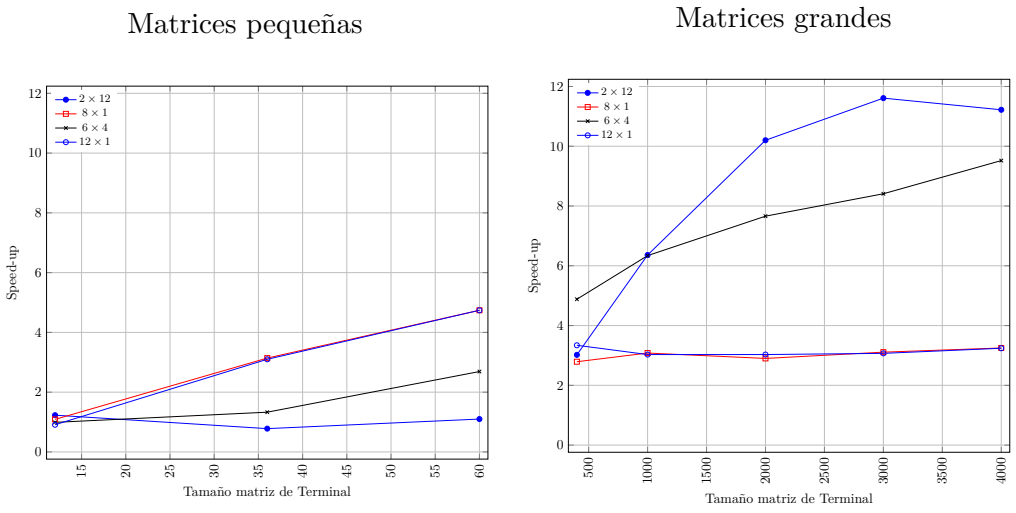


Figura 5.5: Speed-up respecto a la ejecución secuencial usando varias combinaciones de threads OpenMP $\times$ MKL, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Hardware con 24 cores. Matrices simétricas con dispersión del 85 %

De la misma manera, al realizar los experimentos con PARDISO sobre 24 cores, se confirma que en esta librería el número de threads asignados a sus procesos internos no es tan influyente como en el caso de MKL. La tabla 5.9 recoge las mejores combinaciones de threads de todas las probadas y muestra que la combinación óptima para matrices grandes y 6 grupos estructurales paralelizables es de 6 threads OpenMP × 4 threads PARDISO. Para matrices pequeñas el paralelismo OpenMP se muestra como el más efectivo, con un claro 12 threads OpenMP × 1 thread PARDISO como opción preferida. Los speed-ups se pueden consultar en la figura 5.6

nEQ		PARDISO	th. OMP × th. PARDISO			
Terminal	Manivela	Secuencial	2 × 12	8 × 1	6 × 4	12 × 1
12	15	0.0403	0.1414	0.0142	0.0571	**0.014**
36	54	0.0952	0.1364	0.0263	0.0475	**0.0258**
60	90	0.1551	0.1715	0.0441	0.0682	**0.043**
400	400	2.014	1.5007	0.6265	**0.6976**	0.7263
1000	1000	15.1801	6.1079	4.6126	**3.5935**	4.4603
2000	2000	46.6182	18.3725	14.7963	**10.7895**	15.121
3000	3000	111.4345	40.7615	36.863	**24.8652**	37.3051
4000	4000	229.8111	80.9422	75.6751	**52.3393**	76.6807

Tabla 5.9: Tiempos de ejecución obtenidos al combinar threads OpenMP y threads PARDISO, varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 24 cores. Matrices simétricas con dispersión del 85 %

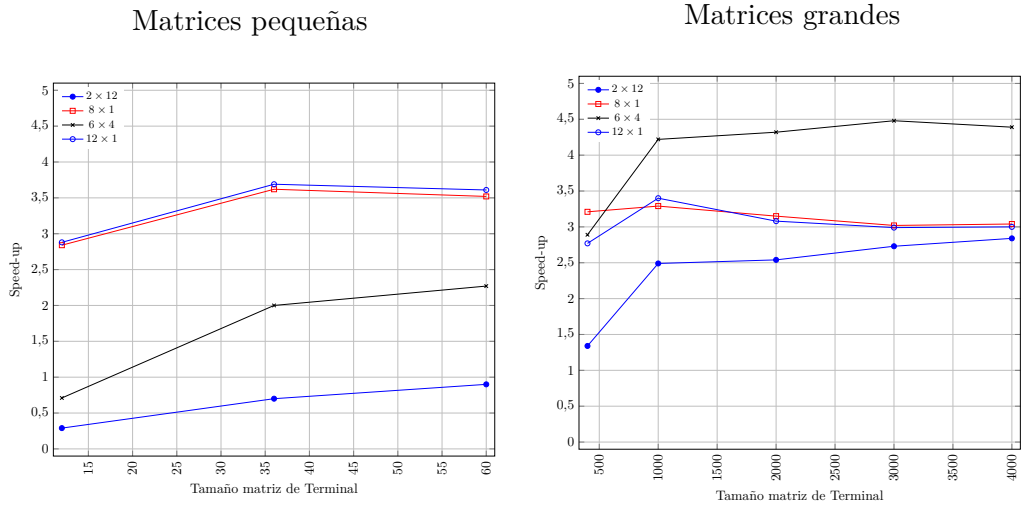


Figura 5.6: Speed-up respecto a la ejecución secuencial usando varias combinaciones de threads OpenMP × PARDISO, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Hardware con 24 cores. Matrices simétricas con dispersión del 85 %

5.3. Selección de los mejores tiempos con paralelismo en dos niveles

En las secciones precedentes se han realizado experimentos para encontrar las mejores configuraciones de threads OpenMP trabajando con cada una de las librerías de álgebra matricial empleadas en el simulador de la plataforma de Stewart, MKL, PARDISO y MA27. Y además, en el caso de MKL y PARDISO, cuyas rutinas permiten paralelismo en los cálculos, se han combinado los threads OpenMP con los asignados a las librerías, buscando mejoras de rendimiento.

En esta sección seleccionamos, de entre todos los experimentos realizados, los que han obtenido los mejores tiempos de ejecución en las plataformas JUPITER y SATURNO usando cualquiera de esas librerías. La tabla 5.10 recoge esta información al trabajar con matrices con un factor de dispersión del 85 % y seis grupos estructurales. Las simulaciones realizan 10 iteraciones (`tfin` = 10) y resuelven 3 veces los sistemas de ecuaciones en cada iteración (`tfin2` = 3).

Se observa que MA27 realiza los cálculos de manera más rápida hasta tamaños de 1000×1000 en las matrices que representan tanto al Terminal como a los grupos Manivela-Barra. Estos tiempos se consiguen en las configuraciones de 12 Threads OpemMP en JUPITER y 24 en SATURNO, dado que MA27 no admite paralelismo en su rutina. Para tamaños superiores, la posibilidad que ofrece PARDISO frente a MA27 de generar threads internamente, combinado con un algoritmo optimizado para el manejo de matrices dispersas, hace que ésta sea la librería más rápida. En una CPU con 12 cores la mejor combinación es 6×2, y en 24 cores la 6×4. En esta ocasión, debido a la dispersión de las matrices, MKL con su solver denso no se ha mostrado tan eficaz y no aparece en esta tabla resumen.

nEQ		JUPITER: 12 cores		SATURNO: 24 cores	
Terminal	Manivela	MA27 12 × 1	PARD 6 × 2	MA27 24 × 1	PARD 6 × 4
12	15	**0.0009**	0.0234	**0.0030**	0.0571
36	54	**0.0043**	0.0288	**0.0060**	0.0475
60	90	**0.0089**	0.043	**0.0132**	0.0682
400	400	**0.1559**	0.4945	**0.2728**	0.6976
1000	1000	**1.6476**	2.4988	**2.8928**	3.5935
2000	2000	10.3387	**7.759**	17.5241	**10.7895**
3000	3000	32.8752	**16.732**	54.6981	**24.8652**
4000	4000	76.5881	**31.6888**	123.6253	**52.3393**

Tabla 5.10: Comparación de los mejores tiempos de ejecución obtenidos con MKL, PARDISO y MA27, con paralismo en dos niveles, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Matrices simétricas con dispersión del 85 %

5.4. Influencia del aumento del número de grupos

Hasta el momento, en este trabajo se han realizado experimentos con 6 grupos estructurales paralelizables (los grupos Manivela-Barra que corresponden a la plataforma de Stewart). En esta sección vamos a generalizar el estudio a sistemas multicuerpo formados por más componentes analizando el efecto que ejerce sobre el coste computacional el incremento a 12, 16, 21 y 26 grupos.

Se han realizado experimentos con las tres librerías (MKL, PARDISO y MA27) con matrices pequeñas de tamaños hasta 60×60 y 90×90, para los grupos que representan el terminal y las manivelas respectivamente, y con matrices grandes, para tamaños a partir de 2000×2000 en ambos casos. La tabla 5.11 recoge los resultados obtenidos en JUPITER usando MKL.

	nEQ		th. OMP × th. MKL					
numGE	Term	Maniv	2 × 6	3 × 4	4 × 3	6 × 1	6 × 2	12 × 1
12	12	15	0.0094	0.0079	0.0078	0.0089	0.0088	**0.0078**
	36	54	0.0237	0.0176	0.0124	0.0099	0.0101	**0.0064**
	60	90	0.0441	0.0386	0.0226	0.0224	0.0236	**0.0138**
	2000	2000	21.4488	**18.0997**	18.8335	33.2223	21.4792	28.3742
	3000	3000	**57.0915**	61.7006	60.7328	98.8415	67.9004	84.5629
	4000	4000	**131.9875**	140.3336	145.4673	228.7952	161.4078	194.2892
16	12	15	0.0037	0.0029	0.0026	0.0016	0.0022	**0.0016**
	36	54	0.0170	0.0127	0.0149	0.0068	0.0076	**0.0052**
	60	90	0.0508	0.0375	0.0335	0.0150	0.0196	**0.0114**
	2000	2000	28.8608	24.9983	**24.6512**	41.6372	31.4104	34.6850
	3000	3000	77.3555	76.6553	**74.9888**	130.8196	82.6664	106.8146
	4000	4000	**174.6793**	176.9465	175.3777	303.7148	195.1640	262.1968
21	12	15	0.0099	0.0037	0.0029	0.0021	0.0021	**0.0019**
	36	54	0.0351	0.0147	0.0130	0.0085	0.0149	**0.0079**
	60	90	0.0714	0.0436	0.0369	0.0191	0.0252	**0.0174**
	2000	2000	33.7653	**27.4524**	35.1464	51.3259	34.7575	37.9592
	3000	3000	108.3810	**97.2876**	101.7590	164.1671	106.8036	122.9380
	4000	4000	215.0848	**214.949**	238.1862	378.7186	239.7968	260.5157
26	12	15	0.0069	0.0044	0.0036	0.0026	0.0028	**0.0024**
	36	54	0.0286	0.0355	0.0149	0.0104	0.0110	**0.0079**
	60	90	0.0841	0.0630	0.0435	0.0234	0.0288	**0.0183**
	2000	2000	42.5543	**36.5608**	37.0554	61.7006	40.1462	47.8761
	3000	3000	**112.7396**	119.1925	121.3996	199.7951	133.6703	151.2720
	4000	4000	**258.6276**	272.7795	286.2665	452.7833	302.4339	365.4496

Tabla 5.11: Tiempos de ejecución obtenidos al combinar threads OpenMP y threads MKL, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 12, 16, 21 y 26 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores. Matrices simétricas con dispersión del 85 %

Como se puede observar, con el aumento en el número de grupos paralelizables, se confirma que las matrices grandes son manejadas con más rapidez por el paralelismo de la propia librería, en las combinaciones que hacen que todos los cores estén ocupados

durante todas las iteraciones. Por ejemplo, en el caso de 16 grupos y 2×6, cada uno de los dos threads OpenMP calculará exactamente $\frac{16}{2} = 8$ grupos asignando 6 threads a MKL. En el caso 4×3, cada thread calculará $\frac{16}{4} = 4$ grupos con 3 threads usados por MKL. Cuando hay 21 grupos, en la combinación 3×4, un thread OpenMP calcula $\frac{21}{3} = 7$ grupos dejando 4 threads a MKL.

Sin embargo, los problemas representados por matrices pequeñas se calculan más rápidamente aumentando el paralelismo OpenMP (12×1 cuando el número de grupos es alto, frente a los 3×4 cuando eran 6 los grupos).

Las ejecuciones con la librería PARDISO, cuyos resultados se muestran en la tabla 5.12, muestran las mismas características que con 6 grupos estructurales, con una clara mejora de rendimiento asignando threads a OpenMP en detrimento de la asignación al paralelismo interno de la librería. En un sistema con 12 cores y un número superior de grupos estructurales, no es de extrañar que la mejor combinación de threads OpenMP y threads PARDISO sea de 12×1.

nEQ			th. OMP × th. PARDISO					
numGE	Term	Maniv	2×6	3×4	4×3	6×1	6×2	12×1
12	12	15	0.0842	0.0596	0.0500	**0.0209**	0.0419	0.0231
	36	54	0.1368	0.0791	0.0736	**0.0349**	0.0495	0.0353
	60	90	0.1512	0.1263	0.0718	0.0562	0.0796	**0.0473**
	2000	2000	21.1950	16.8913	15.1170	13.7537	12.0941	**10.6448**
	3000	3000	46.8865	35.4424	32.7081	31.7584	26.6806	**24.4330**
	4000	4000	87.7763	68.4975	65.6939	62.1044	52.4975	**49.6728**
16	12	15	0.0877	0.0624	0.0450	0.0188	0.0388	**0.0171**
	36	54	0.1312	0.0943	0.0838	0.0385	0.0496	**0.0305**
	60	90	0.1798	0.1386	0.1185	0.0639	0.0736	**0.0501**
	2000	2000	25.6899	20.8820	18.7244	17.8819	15.7097	**14.642**
	3000	3000	59.9358	46.0634	43.2889	42.5419	35.3201	**35.1489**
	4000	4000	112.6297	91.2112	80.0001	84.3465	**68.0332**	69.3354
21	12	15	0.1215	0.0836	0.0600	0.0228	0.0436	**0.0219**
	36	54	0.1730	0.1122	0.1031	0.0507	0.0687	**0.0431**
	60	90	0.2376	0.153	0.1277	0.0807	0.0893	**0.0681**
	2000	2000	32.0843	25.6475	21.7379	22.3684	20.0900	**15.8987**
	3000	3000	73.8862	57.3660	52.4714	53.4410	44.7195	**38.4195**
	4000	4000	146.2502	108.6922	104.1776	102.8016	84.3864	**74.0894**
26	12	15	0.1666	0.1092	0.0778	0.0280	0.0528	**0.0208**
	36	54	0.2095	0.1772	0.1218	0.0604	0.0711	**0.0444**
	60	90	0.2999	0.2146	0.1577	0.0981	0.0995	**0.0695**
	2000	2000	41.6516	32.2497	28.7495	26.6863	22.1178	**20.4074**
	3000	3000	93.2959	72.7546	65.8528	63.4180	52.0657	**49.4086**
	4000	4000	173.6851	134.9991	127.7811	123.3044	99.4763	**96.9931**

Tabla 5.12: Tiempos de ejecución obtenidos al combinar threads OpenMP y threads PARDISO, varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 12, 16, 21 y 26 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores. Matrices simétricas con dispersión del 85 %

5.5. Influencia de la estrategia de reparto de las tareas paralelas

Esta sección profundiza en la búsqueda de las mejores estrategias de asignación entre las iteraciones del bucle *for* que calcula los grupos estructurales y los threads asignados, especialmente en el caso en que el número de grupos estructurales paralelizables exceda al de cores de la CPU. Se consideran tanto los casos en los que ambos están balanceados (cuando es posible una asignación que no deja cores libres) como en los que no lo estén.

Para poder indicar al simulador qué estrategia emplear se ha añadido un nuevo parámetro `ArgLoop`, con lo que la lista completa de opciones que se admiten se representan en la tabla 5.13.

parámetro	descripción
`ArgNumTests`	número de muestreo de tiempos de cada ejecución
`ArgLibrary`	librería de álgebra matricial a emplear en los cálculos
`ArgThreads`	número de threads disponible para paralelismo OpenMP
`ArgMKLThreads`	número de threads usados por MKL
`ArgNestedLevel`	nivel de paralismo anidado
`ArgLoop`	modo de distribución de las tareas en el bucle for paralelizado

Tabla 5.13: Parámetros disponibles en la versión OpenMP del simulador de la Plataforma de Stewart incluyendo la gestión del scheduling

OpenMP permite especificar el modo en que se dividen las iteraciones de un bucle que deben se manejadas por diferentes cores de la CPU, e incluye la posibilidad de que esta asignación sea fija durante toda la ejecución del programa, o variable. Por ello, el nuevo parámetro `ArgLoop` admitirá tres valores, como se muestra en la tabla 5.14.

valor	descripción
`0`	Default
`1`	Dynamic
`2`	Static

Tabla 5.14: Valores que admite el parámetro `ArgLoop`: tipos de estrategia de *scheduling* manejados por el simulador de la Plataforma de Stewart

Hasta el momento, los experimentos que se han realizado en este trabajo con paralelismo OpenMP no indicaban un tipo específico de distribución de las iteraciones, lo que corresponde con un valor del argumento `ArgLoop`=0. En estos casos se realiza una asignación estática, distribuyendo las iteraciones entre los threads en cantidades iguales, pudiendo quedar algún thread con una asignación diferente que complete el total de iteraciones si la división no es exacta. Ahora pretendemos modificar el simulador

para que implemente dos nuevas estrategias que hagan uso de los tipos de asignación estática y dinámica ofrecidas por OpenMP. Además, aprovechando que es posible especificar el número de iteraciones contiguas que se asignan al mismo thread (*chunk size*), desarrollamos una lógica a la hora de calcular ese tamaño en función de que el número de cores sea menor o mayor que el número de grupos paralelos:

- `ArgLoop`=0, *Default* No se indica explícitamente el modo en que las iteraciones se reparten entre los threads.
- `ArgLoop`=1, *Schedule Dynamic size 1* Se divide el total de iteraciones del bucle *for* (número de grupos estructurales) en bloques de tamaño 1 y se van asignando a los threads conforme éstos acaban con la tarea anterior
- `ArgLoop`=2, *Schedule Static size n*, La asignación del trabajo se realiza al principio del bucle y no se modifica. En este caso, el cálculo de n contempla las dos situaciones posibles, como refleja el algoritmo 5.2:
  - Si hay más grupos que threads disponibles para OpenMP, se calcula n dividiendo el número de grupos entre el número de hilos (línea 4). El total de las iteraciones se realizará asignando un bloque de ese tamaño a cada thread (línea 6). El resto que pudiera quedar para completar el 100 % de las iteraciones se realiza en una etapa final, asignándole todos los cores disponibles, con objeto de acelerar su ejecución (líneas 11 y 12).
  - Si hay más cores disponibles que grupos a paralelizar se asigna estáticamente cada iteración a un thread (línea 21 a 24).

Por ejemplo, si tenemos un sistema multicuerpo con 26 grupos paralelizables y disponemos de 12 cores en la CPU con una asignación de 3 threads para OpenMP y 4 threads para MKL, entonces, al existir más grupos que cores, el proceso cuando `ArgLoop`=2 sería el siguiente:

- Se calcularía el tamaño del bloque como $\frac{26\ grupos}{3\ threadsOMP} = 8.6 \rightarrow 8$.
- El resto será: $26\ grupos - (3\ threadsOMP \cdot 8) = 26 - 24 = 2$.
- Se ejecuta el bucle *for* desde 1 hasta $3 \cdot 8 = 24$, con el tamaño de bloque calculado al inicio, 8 en este caso.
- Las 2 iteraciones restantes se ejecutan forzando el número de threads a 2 y, por cada thread OpenMP, se generarán tantos threads MKL que permitan tener ocupados todos los cores (en el caso de JUPITER, $\frac{12\ cores}{2\ threadsOMP} = 6$).

Los experimentos se han realizado en JUPITER con un número de grupos Manivela-Barra superior al números de cores, en concreto 21 y 26, y matrices de tamaños 4000×4000, pues es con matrices grandes donde hemos observado mejor la influencia que ejercen sobre el rendimiento las diversas combinaciones de threads OpenMP y MKL.

Algoritmo 5.2 Pseudocódigo del algoritmo de asignación estática de tareas y threads en el simulador (parámetro `ArgLoop`=2), variando el tamaño del bloque en función del número de grupos estructurales

```
1: omp_num_threads ← ArgThreads
2: mkl_num_threads ← ArgMKLThreads
3: if omp_num_threads <Númerodegrupos then
4:    tamañobloque ← INT(Númerodegrupos/omp_num_threads)
5:    resto ← Númerodegrupos − (omp_num_threads ∗ tamañobloque)
6:    !$OMP DO SCHEDULE(STATIC,tamañobloque)
7:    for k = 1 to omp_num_threads ∗ tamañobloque do
8:       ejecutar simulación
9:    end for
10:   if resto >0 then
11:      omp_num_threads ← resto
12:      mkl_num_threads ← INT((ArgThreads ∗ ArgMKLThreads)/resto)
13:      !$OMP DO SCHEDULE(STATIC,1)
14:      for k = omp_num_threads ∗ tamañobloque to Númerodegrupos do
15:         ejecutar simulación
16:      end for
17:   end if
18:   omp_num_threads ← ArgThreads
19:   mkl_num_threads ← ArgMKLThreads
20: else
21:   !$OMP DO SCHEDULE(STATIC,1)
22:   for k = 1 to Númerodegrupos do
23:      ejecutar simulación
24:   end for
25: end if
```

En esta plataforma, con 12 cores en su CPU, se han empleado las combinaciones de threads OpenMP × MKL que aprovechan el 100 % del hardware de cómputo (1×12, 2×6, 3×4, 4×3, 6×2 y 12×1).

La tabla 5.15 muestra los tiempos de ejecución obtenidos al aplicar las tres configuraciones de scheduling descritas anteriormente, sobre un sistema multicuerpo compuesto por 21 grupos estructurales. En este caso, se observa que la opción de asignación dinámica ofrece mejores rendimientos que la estándar en todas las combinaciones probadas, excepto en los casos de 1×12 y 2×6.

Sin embargo, la asignación estática es mejor que la estándar en el caso 4×3 (4 threads para OMP y 3 threads para MKL). El motivo se encuentra en la relación entre el número de grupos y threads que se produce en este escenario ya que, al dividir los 21 grupos estructurales entre los 4 hilos OpenMP, se obtiene un resto de 1 grupo estructural. En nuestro algoritmo este elemento restante se calcula en último lugar, pero

asignándole todos los cores a MKL, por lo que, al trabajar con matrices grandes, se obtienen muy buenos resultados debido a la optimización de MKL. A pesar de ello, incluso en este caso la asignación dinámica ofrece mejor rendimiento.

	numGE = 21					
th.OMP × th.MKL	1 × 12	2 × 6	3 × 4	4 × 3	6 × 2	12 × 1
ArgLoop0 (Std)	238.0891	215.0848	214.9490	238.1862	239.7968	260.5157
ArgLoop1 (Dyn)	253.2456	215.3644	214.0002	219.5746	231.8144	247.5217
ArgLoop2 (Static)	286.2289	217.1764	224.5785	230.2971	247.3624	302.9380
%ArgLoop1/ArgLoop0	6.37	0.13	**-0.44**	**-7.81**	**-3.33**	**-4.99**
%ArgLoop2/ArgLoop0	20.22	0.97	4.48	**-3.31**	3.16	16.28

Tabla 5.15: Tiempos de ejecución obtenidos y % de reducción respecto al método estándar al modificar el scheduling en el bucle paralelizado, para matrices de tamaño 4000×4000 y 21 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores. Matrices simétricas con dispersión del 85 %

Si repetimos los experimentos subiendo hasta 26 grupos estructurales en la misma CPU con 12 cores, vemos que los casos en los que se produce una reducción del tiempo de ejecución al modificar el scheduling por defecto se da en las combinaciones de threads OpenMP × MKL 3×4, 4×3, 6×2 y 12×1. Todas ellas tienen un resto de 2 al dividir el número de grupos entre los threads OpenMP. Tanto la asignación dinámica como la estática asignando 2 threads a OpenMP y 6 a threads MKL en las iteraciones restantes mejoran los resultados. Al estar trabajando sobre matrices grandes, los threads MKL vuelven a mostrar su potencial respecto al paralelismo OpenMP. La tabla 5.16 recoge los tiempos de ejecución de este experimento.

	numGE = 26					
th.OMP × th.MKL	1 × 12	2 × 6	3 × 4	4 × 3	6 × 2	12 × 1
ArgLoop0 (Std)	294.4234	258.6276	272.7795	286.2665	302.4339	365.4496
ArgLoop1 (Dyn)	293.1446	263.5504	256.6108	257.1548	265.5515	324.5064
ArgLoop2 (Static)	309.6642	259.5620	262.9499	268.7569	289.7602	329.5234
%ArgLoop1/ArgLoop0	-0.43	1.90	**-5.93**	**-10.17**	**-12.20**	**-11.20**
%ArgLoop2/ArgLoop0	5.18	0.36	**-3.60**	**-6.12**	**-4.19**	**-9.83**

Tabla 5.16: Tiempos de ejecución obtenidos y % de reducción respecto al método estándar al modificar el scheduling en el bucle paralelizado, para matrices de tamaño 4000×4000 y 26 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores. Matrices simétricas con dispersión del 85 %

Podemos concluir en este apartado que las variaciones que se obtienen en los tiempos de ejecución debido a la elección de un tipo de scheduling en los bucles paralelizados, requiere un análisis muy detallado, especialmente en el caso en que no estén balanceados el número de grupos estructurales (iteraciones del bucle) y los threads disponibles.

El scheduling dinámico, indicado para cuando las cargas de trabajo de cada iteración son diferentes, no es realmente de interés en el caso de la plataforma de Stewart, ya que los cálculos realizados para cada grupo Manivela-Barra tienen el mismo coste computacional y tan sólo podría ser diferente cuando la convergencia hacia la solución de los sistemas de ecuaciones necesitara más iteraciones en alguno de los grupos estructurales.

El scheduling estático modificado por nosotros en este trabajo puede ofrecer buenos resultados cuando las tareas que quedan por calcular como resto del bucle principal se puedan beneficiar en gran medida de poder disponer de todos los cores (como puede ser el caso de las matrices muy grandes, donde el paralelismo MKL está muy optimizado). Sin embargo, como hemos visto, el paralelismo de PARDISO no ofrece tan buenos resultados, por lo que la valoración de este tipo de scheduling usando esta librería podría ser diferente.

5.6. Conclusiones

El uso de paralelismo OpenMP en plataformas multicore presenta claras ventajas en la resolución simultánea de los grupos estructurales paralelizables de la plataforma de Stewart.

En el caso de usar los cores de la CPU únicamente para OpenMP, sin que las librerías exploten su paralelismo interno, podemos concluir que la mejor opción es la de crear tantos threads como grupos a resolver. Por tanto, en el caso de la plataforma de Stewart con 6 elementos Manivela-Barra, la asignación de 6 threads es la más ventajosa.

El uso del paralelismo de dos niveles es una estrategia donde es necesario seleccionar tanto el número de threads OpenMP como el de la librería utilizada. Una combinación adecuada, en función del tamaño del problema, permite reducir los tiempos de ejecución.

Los experimentos realizados en JUPITER, explotando todos los cores de la CPU, muestran que MKL se comporta mejor cuando se asignan más threads a su algoritmo interno, como es el caso de las combinaciones OMP$\times$MKL de 2×6 con matrices grandes y 3×4 con matrices pequeñas. En SATURNO se observa el mismo comportamiento, especialmente notable en matrices grandes, donde usando los 24 cores, la mejor combinación es 2×12.

Sin embargo, el programa que emplea PARDISO para los cálculos, muestra mejor rendimiento cuando se asignan más threads a OpenMP que a la librería, por ejemplo, 6×2 en JUPITER y 6×4 en SATURNO.

Podemos obtener ligeras mejoras de rendimiento adicionales introduciendo variaciones en el modo de asignar las iteraciones del bucle paralelo a los threads, lo que se conoce en OpenMP como scheduling. Se ha experimentado con escenarios donde el número de grupos y el de cores no está balanceado. En este caso, con matrices gran-

des y usando MKL, se reducen los tiempos de ejecución si los grupos Manivela-Barra excedentes se calculan al final, asignando todos los cores a la librería.

Capítulo 6

Explotación del paralelismo con GPU

Este capítulo analiza el uso de GPUs *(Graphics Processor Units)* para resolver los cálculos que realiza el simulador de la plataforma de Stewart. El objetivo es aprovechar todo el potencial de computación que ofrecen las actuales arquitecturas híbridas formadas por una CPU multicore junto a una o varias GPUs. Se ha seleccionado la librería MAGMA [30] *(Matrix Algebra on GPU and Multicore Architectures)*, que incluye algoritmos que combinan el uso de las CPUs y GPUs. Se realizarán las adaptaciones necesarias en el código fuente para realizar las llamadas a las funciones que ofrece esta librería para la descomposición LU y la posterior resolución de los sistemas de ecuaciones.

El uso de las GPUs no es excluyente y puede combinarse y añadir funcionalidad a los modelos de paralelismo en la CPU con OpenMP y MKL que hemos implementado en el capítulo anterior ya que, en función del número de GPUs disponibles, es posible que interese combinar todos ellos. Este capítulo realizará experimentos de acuerdo a varias estrategias, buscando la mejor combinación de estrategias de reparto de tareas que minimice los tiempos de ejecución.

6.1. Librería MAGMA

MAGMA ofrece al programador dos tipos de interfaces para la mayoría de las funciones que implementa:

- *Interface CPU*: La matriz de datos se encuentra inicialmente en la memoria de la CPU. Por tanto, este interface lo primero que realiza es una reserva de memoria en la GPU y le transfiere toda la matriz. A continuación realiza los cálculos usando un algoritmo híbrido que combina CPU y GPU. Finalmente devuelve el resultado a la memoria de la CPU.
- *Interface GPU*: Usa la información de la matriz que ya está almacenada en la memoria de la GPU, realiza los cálculos combinando CPU y GPU, y almacena el resultado en la memoria de la GPU.

Como se ha descrito a lo largo de esta Tesis, los cálculos que realiza el simulador de sistemas multicuerpo, como es la plataforma de Stewart, implican la resolución de varios sistemas de ecuaciones, incluida una factorización LU previa de las matrices asociadas a esos sistemas.

MAGMA tiene implementada la descomposición LU en ambos interfaces, sin embargo, la resolución del sistema de ecuaciones únicamente la tiene desarrollada con el interface GPU. Por tanto, debemos distinguir dos posibles escenarios, según el interface empleado:

- En el interface CPU, la descomposición LU la realiza MAGMA, empleando recursos de la CPU y GPU. Sin embargo, la resolución del sistema de ecuaciones no está implementada en este interface. En su lugar se emplea la función DGETRS de MKL.
- En el interface GPU, la descomposición LU y la resolución del sistema de ecuaciones se realiza con MAGMA, empleando en ambos casos el algoritmo híbrido que explota recursos de la CPU y GPU.

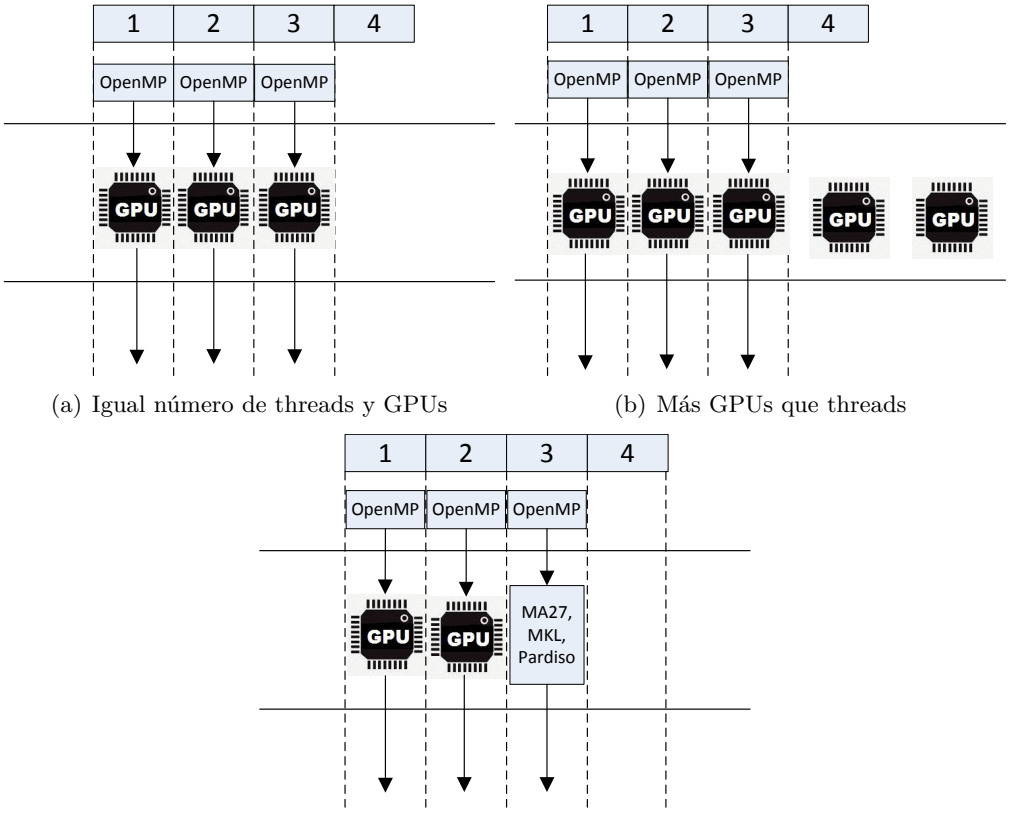
6.2. Paralelismo híbrido CPU+GPU

En el capítulo anterior sobre paralelismo en sistemas multicore, se comprobó experimentalmente que los mejores rendimientos en una CPU se consiguen con la acción combinada del paralelismo OpenMP con el implementado en el interior de las rutinas de las librerías MKL y PARDISO (recordemos que MA27 no implementa algoritmos paralelos). La proporción más ventajosa de threads asignados a uno u otro dependía del tamaño de las matrices que representan las ecuaciones de restricción del modelo mecánico a simular, su factor de dispersión y la librería empleada.

En un entorno multicore, con threads OpenMP donde cada hilo ejecuta los cálculos de un grupo estructural Manivela-Barra, la incorporación a los cálculos de GPUs manejadas por MAGMA genera tres posibles escenarios:

- Disponemos de tantas GPUs como threads generados. En este caso, cada thread OpenMP puede seleccionar una GPU que será la utilizada por MAGMA en sus operaciones, en combinación con la CPU.
- Disponemos de más GPUs que threads, en cuyo caso, en un escenario simple, cada thread puede seleccionar una GPU dejando el resto de GPUs sin utilizar. En un algoritmo mejorado, la elección de una u otra GPU se podría basar en el rendimiento que ofrezca cada una, seleccionando en primer lugar las más potentes.
- No hay suficientes GPUs para los threads que manejan los grupos estructurales. En esta situación, se podría optar entre reducir el número de threads OpenMP para que sea igual al número de GPUs, o dividir el trabajo de manera que unos threads usen GPU y otros utilicen para los cálculos cualquiera de las otras librerías de álgebra lineal disponibles en el simulador (MKL, PARDISO o MA27).

Como ejemplo, podemos observar la figura 6.1, que representa tres escenarios en los que se crean tres threads OpenMP en plataformas hardware con diferente número de GPUs instaladas.



(a) Igual número de threads y GPUs

(b) Más GPUs que threads

(c) Más threads que GPUs

Figura 6.1: Escenarios posibles al incorporar GPUs y la librería MAGMA al simulador de la Plataforma de Stewart

En (a) disponemos de un hardware con tres GPUs donde cada thread selecciona una de ellas para que MAGMA la use, en combinación con la CPU, en sus cálculos. Sin embargo, en (b), dos de las cinco GPUs disponibles se quedan sin utilizar. Por el contrario, en (c), al haber sólo dos GPUs, los cálculos asignados al tercer thread los realiza otra librería (por ejemplo MA27). En este escenario podríamos enfrentarnos a problemas de balanceo si hubiera grandes diferencias entre los tiempos de resolución obtenidos al emplear GPUs y los ofrecidos por las librerías que se ejecutan únicamente CPU. Esto obligaría a seleccionar un método adecuado de *sheduling*.

El simulador de la plataforma de Stewart maneja los dos interfaces de MAGMA, CPU y GPU. Para indicarle cuál debe utilizar se usan los valores 4, 5, 6 y 7 del parámetro `ArgLibrary`. La lista completa de valores para este parámetro se puede consultar en la tabla 6.1.

parámetro	descripción
1	MKL
2	PARDISO
3	MA27
4	MAGMA con interface CPU (número de threads <GPUs)
5	MAGMA con interface GPU (número de threads <GPUs)
6	MAGMA con interface CPU (sin limitación de threads)
7	MAGMA con interface GPU (sin limitación de threads)

Tabla 6.1: Valores que admite el parámetro `ArgLibrary`: librerías de álgebra lineal manejadas por el simulador de la Plataforma de Stewart incluyendo el uso de GPUs con MAGMA

En los casos en que `ArgLibrary` tome el valor 4 o 5, el simulador comprueba que el número de threads OpenMP sea menor o igual que el número de GPUs disponibles. En caso de que sea mayor, reduce el valor de threads y lo iguala al número de GPUs (línea 6 del algoritmo 6.1), ampliando el número de threads asignados al paralelismo de MKL (línea 8).

Algoritmo 6.1 Pseudocódigo del algoritmo para limitar el número de threads al número de GPUs en el simulador (`ArgLibrary`=4 o `ArgLibrary`=5)

```
1:  num_gpus ← magma_num_devices()
2:  omp_num_threads ← ArgThreads
3:  mkl_num_threads ← ArgMKLThreads
4:  if ArgLibrary = 4 or ArgLibrary = 5 then
5:     { ! Limitar el número de threads al de GPUs}
6:     omp_num_threads ← min(num_gpus, ArgThreads)
7:     { ! Asignar cores libres a MKL}
8:     mkl_num_threads ←
9:        max(1, int((ArgThreads * ArgMKLThreads)/omp_num_threads))
10: end if
```

El algoritmo 6.2, en cambio, representa el proceso cuando `ArgLibrary` toma el valor 6 o 7. En este caso se fijan los threads OpenMP según indica el valor del argumento `ArgThreads` (línea2). Dentro de la sección paralela marcada por las líneas 6 y 22, cuando se inicia un thread con un identificador, *TID*, que no supera al número de GPUs (línea 8), ese hilo selecciona la GPU que tiene el mismo identificador e indica a MAGMA que realice con ella los cálculos, combinando recursos de la CPU y la GPU recién seleccionada. En caso contrario (línea 14), los cálculos de ese hilo usarán MKL o PARDISO: PARDISO si el factor de dispersión es mayor del 50 % (línea 15) o MKL en caso contrario.

Algoritmo 6.2 Pseudocódigo del algoritmo para desviar a MKL o PARDISO los cálculos en los threads que no dispongan de una GPU libre en el simulador (`ArgLibrary`=6 o `ArgLibrary`=7)

```
1: num_gpus ← magma_num_devices()
2: omp_num_threads ← ArgThreads
3: mkl_num_threads ← ArgMKLThreads
4: !$OMP PARALLEL PRIVATE idGrupo
5: TID ← omp_get_thread_num()
6: !$OMP DO
7: for all Manivela-Barra, (numGE) do
8:    if TID <num_gpus  then
9:       if ArgLibrary = 6 then
10:         Resolver par Manivela-Barra con MAGMA-CPU
11:      else if ArgLibrary = 7 then
12:         Resolver par Manivela-Barra con MAGMA-GPU
13:      end if
14:   else
15:      if sparsity >50 then
16:         Resolver par Manivela-Barra con PARDISO
17:      else
18:         Resolver par Manivela-Barra con MKL
19:      end if
20:   end if
21: end for
22: !$OMP END PARALLEL
```

Los experimentos de este capítulo se han realizado exclusivamente en JUPITER, ya que este sistema incluye 6 GPUs. De esta manera se han podido probar las distintas combinaciones que se pueden presentar entre el número de threads y el de GPUs.

Los escenarios representan 6 grupos estructurales y matrices simétricas con un factor de dispersión del 85 % (recordamos que en la formulación de la plataforma de Stewart basada en grupos estructurales se obtienen matrices con una factor de dispersión siempre superior al 70 %). Cada simulación realiza 10 iteraciones (`tfin` = 10) y resuelve 3 veces los sistemas de ecuaciones del problema de la posición en cada iteración (`tfin2` = 3), lo que está en concordancia con el número de etapas máximo que necesita el método iterativo Newton-Raphson para converger a una solución en un caso real de control de un robot manipulador.

Aunque el simulador tiene la posibilidad de realizar varias tomas de tiempos en cada ejecución, hemos optado por tomar sólo una, con objeto de reducir el tiempo de simulación (`ArgNumTests` = 1).

6.2.1. Interface CPU

Recordemos que en este interface de MAGMA la descomposición LU de las matrices se calcula en la GPU (en un algoritmo híbrido que combina CPU y GPU). Sin embargo, la resolución del sistema de ecuaciones no está implementada, por lo que emplearemos la función DGETRS de la librería MKL para esta tarea, que se ejecuta en la CPU.

En esta sección se realizan los experimentos que comparan los rendimientos que ofrece la librería MAGMA respecto a MKL y PARDISO utilizando este interface. Probaremos el caso en que el número de threads OpenMP no excede al de GPUs disponibles (parámetro `ArgLibrary`=4 del simulador).

La tabla 6.2 contiene los resultados obtenidos al utilizar MAGMA en la asignación de 6×2, que es la que ofrece menores tiempos de ejecución. Se compara con los tiempos obtenidos con una ejecución MKL secuencial, y con las mejores configuraciones de paralelismo en dos niveles que se obtuvieron en el capítulo anterior con MKL en ejecuciones sobre CPU (que fueron las combinaciones 2×6 y 3×4).

			th. OMP × th. MKL					
nEQ		Secuencial	2 × 6		3 × 4		6 × 2	
Term	Maniv	MKL	MKL	MAGMA	MKL	MAGMA	MKL	MAGMA
12	15	0.0087	0.0077	0.0073	**0.0064**	0.0064	0.0082	0.0064
36	54	0.0206	0.0133	0.0073	**0.0059**	0.0066	0.0062	0.0067
60	90	0.0387	0.0244	0.0326	**0.0122**	0.0269	0.0122	0.0122
400	400	0.7518	0.2862	3.7761	**0.2361**	3.1431	0.2518	1.4334
1000	1000	9.0309	**1.8331**	5.7709	2.0624	5.1961	2.0108	4.2319
2000	2000	67.0978	**10.2912**	17.3762	11.4366	15.4911	13.8333	13.0225
3000	3000	216.9713	34.3508	37.7619	41.6309	31.2775	43.0783	**27.0364**
4000	4000	505.3326	68.6536	62.7292	76.7721	56.0274	90.8851	**48.5804**

Tabla 6.2: Comparación de tiempos de ejecución entre MAGMA con interface CPU y MKL, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores y 6 GPUs. Matrices simétricas con dispersión del 85 %

La configuración 2×6 usa, por tanto, 2 GPUs, y la 3×4 usa 3 GPUs. Como podemos observar, no es extraño que la mejor opción al utilizar MAGMA sea emplear 6 threads OpenMP porque, de esta manera, cada thread reserva un GPU, utilizándose por tanto, las 6 disponibles. La mejoría de rendimiento es notable a partir de un tamaño de matrices de 2000×2000. En el caso de 4000×4000 se consigue una reducción del 46 %, pasando de un tiempo de ejecución de 90.8 segundos a 48.5 segundos. La figura 6.2 representan los speed-up de las combinaciones de theads que ofrecen mejor rendimientos respecto a la ejecución sin ningún tipo de paralelismo con MKL.

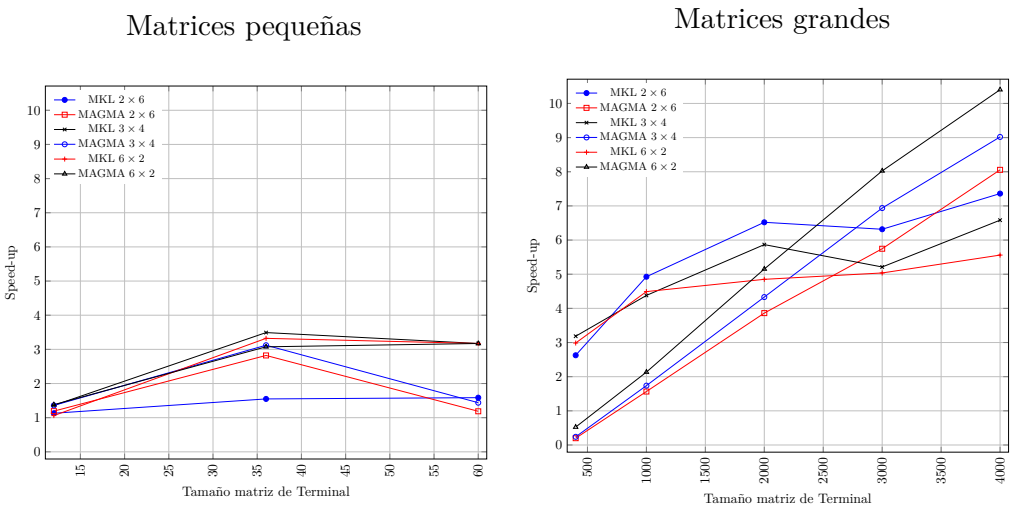


Figura 6.2: Speed-up usando MAGMA con interface CPU y las mejores combinaciones de hilos OpenMP y MKL respecto a una ejecución secuencial de MKL, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Matrices simétricas con dispersión del 85 %

Ahora bien, si comparamos los resultados anteriores conseguidos con MAGMA CPU con los que obtuvimos al utilizar PARDISO, se comprueba que, conforme aumenta la dimensión de las matrices, los beneficios de emplear una librería especializada en matrices dispersas como PARDISO superan al empleo de GPUs. En este caso MAGMA no es la mejor opción, como se puede observar en la tabla de resultados 6.3 y en el gráfico de speed-ups (figura 6.3).

			th. OMP × th. PARDISO			
nEQ		Secuencial	6 × 1		6 × 2	
Terminal	Manivela	PARDISO	PARDISO	MAGMA	PARDISO	MAGMA
12	15	0.0329	0.0140	0.0064	0.0234	**0.0063**
36	54	0.0720	0.0222	0.0073	0.0288	**0.0067**
60	90	0.1199	0.0356	**0.0082**	0.0430	0.0122
400	400	1.2535	**0.4057**	1.6138	0.4945	1.4334
1000	1000	9.5983	2.9947	4.6276	**2.4988**	4.2319
2000	2000	27.4114	8.7775	14.554	**7.7590**	13.0225
3000	3000	63.3465	20.3311	29.6831	**16.7320**	27.0364
4000	4000	128.9157	40.1353	54.5644	**31.6888**	48.5804

Tabla 6.3: Comparación de tiempos de ejecución entre MAGMA con interface CPU y PARDISO, varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores y 6 GPUs. Matrices simétricas con dispersión del 85 %

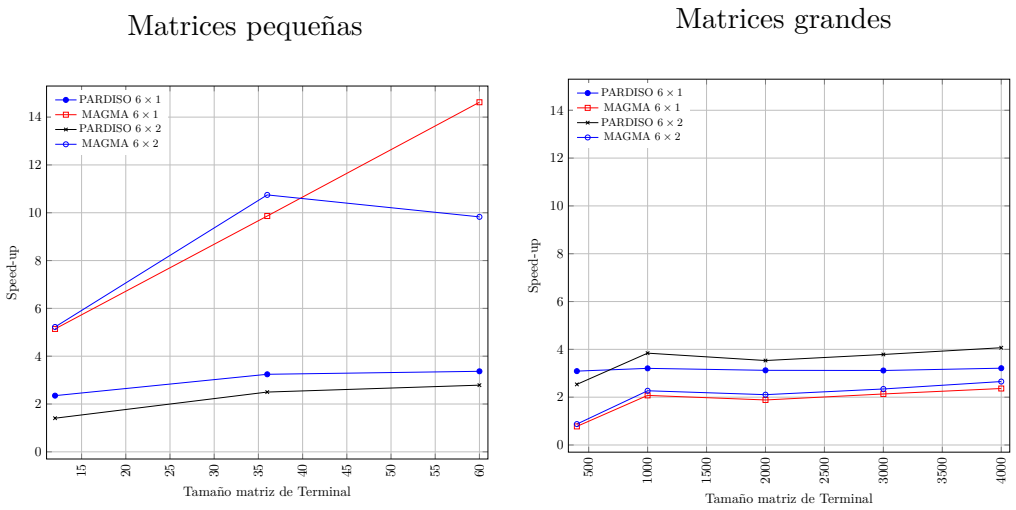


Figura 6.3: Speed-up usando MAGMA con interface CPU y las mejores combinaciones de hilos OpenMP y PARDISO respecto a una ejecución secuencial de PARDISO, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Matrices simétricas con dispersión del 85 %

6.2.2. Interface GPU

Como vimos anteriormente, MAGMA tiene implementadas en su interface GPU tanto la descomposición LU como la resolución del sistema de ecuaciones, y en ambos casos la librería emplea un algoritmo híbrido que explota recursos de la CPU y GPU.

En esta subsección se realizan experimentos que comparan los rendimientos que ofrece la librería MAGMA, usando este interface, respecto a los tiempos de ejecución que se obtuvieron con MKL y PARDISO. Para asegurar que cada thread OpenMP pueda disponer de una GPU para usar con MAGMA, usaremos el parámetro `ArgLibrary`=5 del simulador, que automáticamente ajusta el número de threads si éste es mayor que el número de GPUs instaladas.

La tabla 6.4 muestra los mejores resultados obtenidos al utilizar MAGMA con el interface GPU (combinación 6×2). Se compara con los tiempos obtenidos con una ejecución secuencial de MKL, y las mejores combinaciones de threads OpenMP×MKL que se obtuvieron en el capítulo anterior realizando todos los cálculos con MKL sobre CPU (2×6 y 3×4).

La mejor opción al utilizar GPUs es lanzar 6 threads OpenMP porque, de esta manera, cada thread reserva una GPU para MAGMA, aprovechando por tanto las 6 GPUs disponibles. A partir de tamaños de 1000×1000 se consiguen notables reducciones en el tiempo de ejecución (usando el interface CPU, las mejorías se notaban a partir de 2000×2000). En el caso de 4000×4000 se consigue una reducción del 69 %, pasando de 90.8 a 28.2 segundos. La figura 6.4 muestra los speed-up respecto a MKL secuencial.

nEQ		Secuencial	th. OMP × th. MKL					
			2 × 6		3 × 4		6 × 2	
Term	Maniv	MKL	MKL	MAGMA	MKL	MAGMA	MKL	MAGMA
12	15	0.0087	0.0077	1.2258	**0.0064**	2.0028	0.0082	4.1789
36	54	0.0206	0.0133	0.0999	**0.0059**	0.1131	0.0062	0.1242
60	90	0.0387	0.0244	0.2779	**0.0122**	0.2387	0.0122	0.2362
400	400	0.7518	0.2862	1.2254	0.2361	1.1266	**0.2518**	1.2620
1000	1000	9.0309	**1.8331**	2.7287	2.0624	2.6829	2.0108	2.2188
2000	2000	67.0978	10.2912	9.2946	11.4366	7.8085	13.8333	**5.9685**
3000	3000	216.9713	34.3508	23.7600	41.6309	18.9474	43.0783	**14.0435**
4000	4000	505.3326	68.6536	48.3116	76.7721	38.1400	90.8851	**28.2109**

Tabla 6.4: Comparación de tiempos de ejecución entre MAGMA con interface GPU y MKL, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores y 6 GPUs. Matrices simétricas con dispersión del 85 %

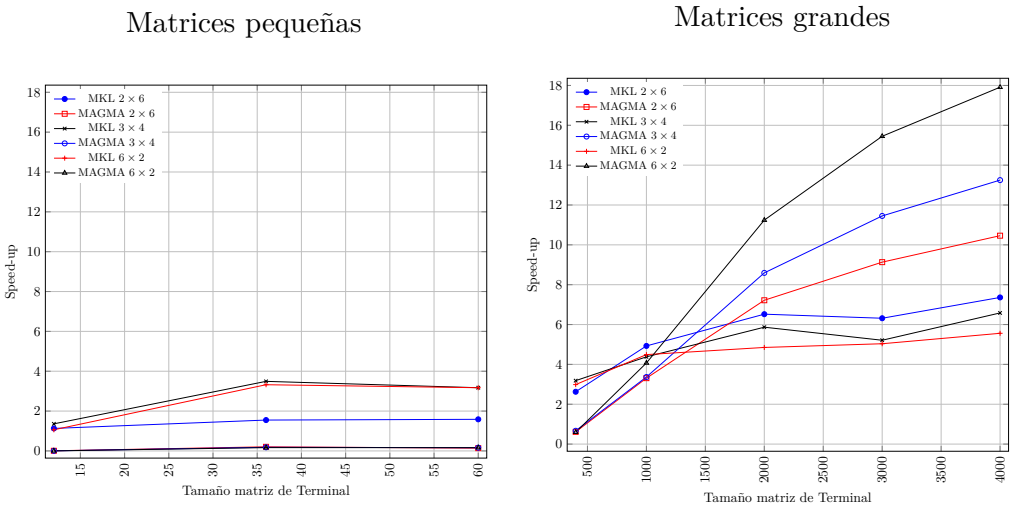


Figura 6.4: Speed-up usando MAGMA con interface GPU y las mejores combinaciones de hilos OpenMP y MKL respecto a una ejecución secuencial de MKL, para diversos tamaños de matrices y 6 grupos estructurales Manivela-Barra. Matrices simétricas con dispersión del 85 %

Ahora bien, a diferencia de lo que ocurría al usar el inteface CPU, donde MAGMA no conseguía mejorar a PARDISO en los tamaños de matrices de los experimentos, el interface GPU mejora los resultados obtenidos por PARDISO cuando las matrices alcanzan tamaños de 1000×1000, como podemos observar en la tabla 6.5.

			th. OMP × th. PARDISO			
nEQ		Secuencial	6×1		6×2	
Terminal	Manivela	PARDISO	PARDISO	MAGMA	PARDISO	MAGMA
12	15	0.0329	**0.0140**	4.0673	0.0234	4.1789
36	54	0.0720	**0.0222**	0.0990	0.0288	0.1242
60	90	0.1199	**0.0356**	0.2490	0.0430	0.2362
400	400	1.2535	**0.4057**	1.4786	0.4945	1.2620
1000	1000	9.5983	2.9947	2.3467	2.4988	**2.2188**
2000	2000	27.4114	8.7775	5.9775	7.7590	**5.9685**
3000	3000	63.3465	20.3311	14.2522	16.7320	**14.0435**
4000	4000	128.9157	40.1353	28.8172	31.6888	**28.2109**

Tabla 6.5: Comparación de tiempos de ejecución entre MAGMA con interface GPU y PARDISO, varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores y 6 GPUs. Matrices simétricas con dispersión del 85 %

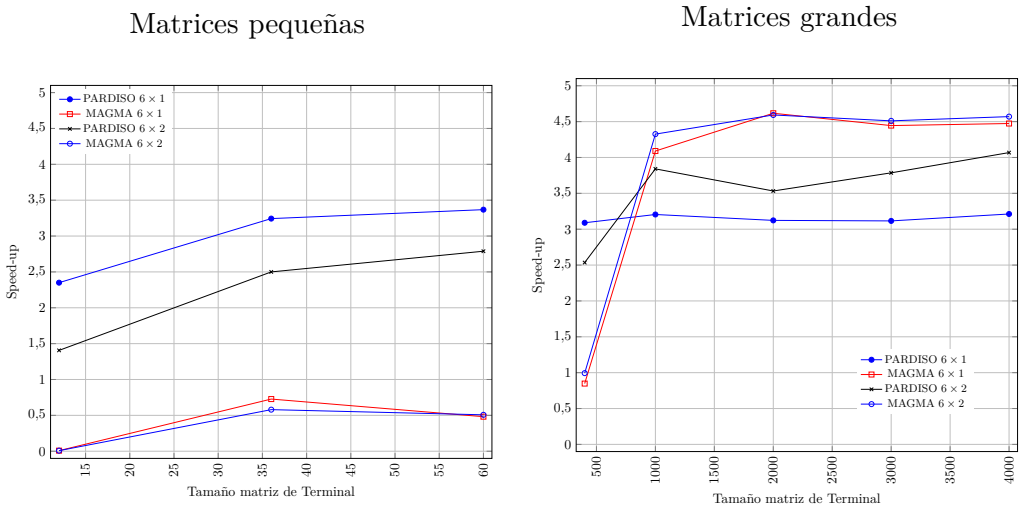


Figura 6.5: Speed-up usando MAGMA GPU y las mejores combinaciones de threads OpenMP × PARDISO respecto a PARDISO secuencial. 6 grupos estructurales Manivela-Barra. Matrices simétricas con dispersión del 85 %

6.3. Selección de los mejores tiempos con paralelismo híbrido CPU + GPU

En los capítulos anteriores se realizaron experimentos para encontrar las mejores configuraciones de threads OpenMP combinados con el paralelismo interno de las rutinas en las librerías MKL, PARDISO y MA27. En este capítulo se han realizado experimentos empleando la librería MAGMA, que implementa algoritmos híbridos, capaces de aprovechar de manera combinada la potencia de cómputo de la CPU y las GPU. Dado que MAGMA puede ser invocada desde un thread OpenMP y es capaz de seleccionar una GPU concreta para realizar los cálculos, se han probado las configuraciones de threads OpenMP que, junto a MAGMA, ofrecían los mejores tiempos de ejecución.

En esta sección seleccionamos, de entre todos los experimentos realizados hasta el momento en esta Tesis, los que han obtenido los mejores tiempos de ejecución en la plataforma JUPITER. La tabla 6.6 recoge esta información cuando se trabaja con matrices con un factor de dispersión del 85 % y seis grupos estructurales. Las simulaciones realizan 10 iteraciones (`tfin` = 10) y resuelven 3 veces los sistemas de ecuaciones en cada iteración (`tfin2` = 3).

Se observa que MA27 realiza los cálculos de manera más rápida hasta tamaños de 1000×1000 en las matrices que representan tanto al Terminal como a los grupos Manivela-Barra. Estos tiempos se consiguen en la configuración de 12 Threads OpenMP, dado que MA27 no admite paralelismo en su rutina. Para tamaños superiores, el potencial que ofrece MAGMA a la hora de usar la CPU y GPU de manera combinada, hace que esta opción sea la más rápida, superando incluso a PARDISO, librería especializada en matrices dispersas. En un hardware con 12 cores y 6 GPUs, la mejor combinación es 6×2 ya que utiliza todas las GPUs. En esta ocasión, debido a la dispersión de las matrices, MKL con su solver denso no se ha mostrado eficaz y no aparece en esta tabla resumen.

		JUPITER: 12 cores + 6 GPUs		
nEQ		MA27 12 × 1	PARD 6 × 2	MAGMA 6×2
Terminal	Manivela	CPU	CPU	CPU + GPU
12	15	**0.0009**	0.0234	4.1789
36	54	**0.0043**	0.0288	0.1242
60	90	**0.0089**	0.0430	0.2362
400	400	**0.1559**	0.4945	1.2620
1000	1000	**1.6476**	2.4988	2.2188
2000	2000	10.3387	7.7590	**5.9685**
3000	3000	32.8752	16.7320	**14.0435**
4000	4000	76.5881	31.6888	**28.2109**

Tabla 6.6: Comparación de los mejores tiempos de ejecución obtenidos con MKL, PARDISO, MA27 y MAGMA, con paralismo en dos niveles y GPU, con varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6 grupos estructurales Manivela-Barra (`NumGE`). Matrices simétricas con dispersión del 85 %

6.4. Influencia del aumento del número de grupos

Los experimentos realizados en la sección anterior simulaban 6 grupos estructurales paralelizables (los grupos Manivela-Barra en el caso de la plataforma de Stewart). Con objeto de generalizar el estudio a sistemas de mayor complejidad, analizamos en esta sección el efecto que ejerce sobre el coste computacional el incremento en el número de grupos. Dado que el sistema sobre el que realizaremos las simulaciones es JUPITER, con 6 GPUs, experimentaremos con tamaños que nos permitan estudiar el comportamiento del simulador en función de si hay proporcionalidad o no entre el número de GPUs y el de grupos. A tal efecto seleccionamos escenarios con 12, 16, 21 y 26 elementos Manivela/Barra.

6.4.1. Interface CPU

En este interface, MAGMA ofrece un algoritmo híbrido que combina CPU y GPU para el cálculo de la descomposición LU de las matrices. Sin embargo, la resolución posterior del sistema de ecuaciones no está implementada, y es necesario recurrir a la función DGETRS de la librería MKL, que se ejecuta exclusivamente en la CPU.

Establecemos en el simulador el parámetro `ArgLibrary`=4 para que el número de threads OpenMP no exceda al de GPUs disponibles, de manera que cada hilo pueda reservarse una GPU para el uso con MAGMA.

La tabla 6.7 compara los mejores resultados obtenidos en la sección 5.4 empleando MKL, y MAGMA con el interface CPU. De esta comparativa podemos concluir que las matrices pequeñas (tamaños inferiores a 2000×2000) se calculan más rápidamente con MKL. Sin embargo, con tamaños mayores, MAGMA (al combinar el uso de CPU y GPU) obtiene mejores resultados, siendo las combinaciones 6×2 y 12×1 las óptimas, dado que ambas explotan las 6 GPUs de JUPITER. Recordamos que con `ArgLibrary`=4, los 12 threads se limitan a 6, para que sea igual al número de GPUs, asignando los 6 threads restantes al paralelismo MKL que, como hemos comentado, se emplea para la resolución del sistemas de ecuaciones (12×1 pasa a 6×2).

La tabla 6.8 recoge los resultados anteriores obtenidos con el interface CPU de MAGMA, y los compara con los mejores tiempos que conseguimos empleando PARDISO, y que se pueden consultar en la sección 5.4. De esta comparativa podemos concluir que las matrices pequeñas (tamaños inferiores a 2000×2000) se calculan más rápidamente con MAGMA pero que, para matrices grandes, PARDISO aprovecha mejor el diseño del algoritmo optimizado para matrices dispersas, y obtiene tiempos de ejecución notablemente menores.

			th. OMP × th. MKL					
	nEQ		2 × 6		6 × 2		12 × 1	
numGE	Term	Maniv	MKL	MAGMA	MKL	MAGMA	MKL	MAGMA
12	12	15	0.0094	0.0032	0.0088	**0.0014**	0.0116	0.0022
	36	54	0.0237	0.0128	0.0101	0.0097	**0.0064**	0.0075
	60	90	0.0441	0.0471	0.0236	0.0205	**0.0138**	0.0200
	2000	2000	21.4488	30.7929	21.4792	**21.2633**	28.3742	21.5894
	3000	3000	57.0915	65.6145	67.9004	**44.6455**	84.5629	44.7031
	4000	4000	131.9875	112.4891	161.4078	81.2630	194.2892	**81.1481**
16	12	15	0.0037	0.0040	**0.0022**	0.0028	0.0026	0.0028
	36	54	0.0170	0.0221	0.0076	0.0086	**0.0052**	0.0087
	60	90	0.0508	0.0535	0.0196	0.0234	**0.0114**	0.0288
	2000	2000	28.8608	40.0627	31.4104	**27.9063**	34.6850	28.0445
	3000	3000	77.3555	83.2919	82.6664	57.8916	106.8146	**57.7109**
	4000	4000	174.6793	144.1233	195.1640	106.3209	262.1968	**105.9838**
21	12	15	0.0099	0.0053	0.0021	0.0023	**0.0019**	0.0024
	36	54	0.0351	0.0222	0.0149	0.0109	**0.0079**	0.0149
	60	90	0.0714	0.0936	0.0252	0.0372	**0.0174**	0.0384
	2000	2000	33.7653	52.6491	34.7575	**35.2789**	37.9592	35.1641
	3000	3000	108.381	109.541	106.8036	**73.7687**	122.9380	74.0750
	4000	4000	215.0848	189.0611	239.7968	**134.9198**	260.5157	135.5642
26	12	15	0.0069	0.0063	0.0028	**0.0027**	0.0024	0.0027
	36	54	0.0286	0.0258	**0.0110**	0.0131	0.0079	0.0132
	60	90	0.0841	0.1020	0.0288	0.0472	**0.0183**	0.0357
	2000	2000	42.5543	62.1918	**40.1462**	42.3773	47.8761	42.6545
	3000	3000	112.7396	130.6194	133.6703	**89.1731**	151.2720	89.6608
	4000	4000	258.6276	227.0927	302.4339	**163.3435**	365.4496	164.0246

Tabla 6.7: Comparación de tiempos de ejecución entre MAGMA con interface CPU y MKL, con 12, 16, 21 y 26 grupos estructurales Manivela-Barra (`NumGE`) y varios tamaños de matrices (`nEQTerminal` y `nEQManivela`). Hardware con 12 cores y 6 GPUs. Matrices simétricas con dispersión del 85 %

			th. OMP × th. PARDISO					
	nEQ		6 × 1		6 × 2		12 × 1	
numGE	Term	Maniv	PARDISO	MAGMA	PARDISO	MAGMA	PARDISO	MAGMA
12	12	15	0.0209	0.0077	0.0419	**0.0014**	0.0231	0.0022
	36	54	0.0349	0.0083	0.0495	0.0097	0.0353	**0.0075**
	60	90	0.0562	0.0214	0.0796	0.0205	0.0473	**0.0200**
	2000	2000	13.7537	23.7762	12.0941	21.2633	**10.6448**	21.5894
	3000	3000	31.7584	49.1267	26.6806	44.6455	**24.4330**	44.7031
	4000	4000	62.1044	89.4840	52.4975	81.2630	**49.6728**	81.1481
16	12	15	0.0188	**0.0018**	0.0388	0.0028	0.0171	0.0028
	36	54	0.0385	**0.0087**	0.0496	0.0086	0.0305	0.0087
	60	90	0.0639	**0.0182**	0.0736	0.0234	0.0501	0.0288
	2000	2000	17.8819	30.7280	15.7097	27.9063	**14.642**	28.0445
	3000	3000	42.5419	64.2767	35.3201	57.8916	**35.1489**	57.7109
	4000	4000	84.3465	117.2140	68.0332	106.3209	**69.3354**	105.9838
21	12	15	0.0228	**0.0023**	0.0436	0.0023	0.0219	0.0024
	36	54	0.0507	**0.0130**	0.0687	0.01090	0.0431	0.0149
	60	90	0.0807	**0.0302**	0.0893	0.0372	0.0681	0.0384
	2000	2000	22.3684	39.0018	20.0900	35.2789	**15.8987**	35.1641
	3000	3000	53.4410	81.3996	44.7195	73.7687	**38.4195**	74.0750
	4000	4000	102.8016	149.0153	84.3864	134.9198	**74.0894**	135.5642
26	12	15	0.0280	0.0029	0.0528	**0.0027**	0.0208	0.0027
	36	54	0.0604	0.0132	0.0711	**0.0131**	0.0444	0.0132
	60	90	0.0981	**0.0288**	0.0995	0.0472	0.0695	0.0357
	2000	2000	26.6863	46.6777	22.1178	42.3773	**20.4074**	42.6545
	3000	3000	63.4180	97.8483	52.0657	89.1731	**49.4086**	89.6608
	4000	4000	123.3044	179.7404	99.4763	163.3435	**96.9931**	164.0246

Tabla 6.8: Comparación de tiempos de ejecución entre MAGMA con interface CPU y PARDISO, con 12, 16, 21 y 26 grupos estructurales Manivela-Barra (`NumGE`) y varios tamaños de matrices (`nEQTerminal` y `nEQManivela`). Hardware con 12 cores y 6 GPUs. Matrices simétricas con dispersión del 85 %

6.4.2. Interface GPU

En su interface GPU, MAGMA implementa algoritmos híbridos, que combinan recursos de CPU y GPU para el cálculo de la descomposición LU de matrices y la resolución de los sistemas de ecuaciones.

Para los experimentos de esta subsección establecemos en el simulador el parámetro `ArgLibrary`=5 para que el número de threads OpenMP no excede al de GPUs disponibles, de manera que cada hilo pueda reservarse una GPU para el uso con MAGMA.

La tabla 6.9 recoge los tiempos de ejecución obtenidos con MAGMA empleando el interface GPU, y los compara con los mejores resultados que se consiguieron empleando MKL en un sistema multicore, según experimentos realizados en la sección 5.4.

			th. OMP × th. MKL					
	nEQ		2 × 6		6 × 2		12 × 1	
numGE	Term	Maniv	MKL	MAGMA	MKL	MAGMA	MKL	MAGMA
12	12	15	0.0094	0.4120	**0.0088**	0.4584	0.0116	0.4497
	36	54	0.0237	0.1938	0.0101	0.2017	**0.0064**	0.1789
	60	90	0.0441	0.4975	0.0236	0.4764	**0.0138**	0.4648
	2000	2000	21.4488	16.1443	21.4792	**9.5609**	28.3742	9.6285
	3000	3000	57.0915	41.7481	67.9004	**21.9079**	84.5629	22.1048
	4000	4000	131.9875	84.6859	161.4078	**44.4854**	194.2892	44.7893
16	12	15	0.0037	0.5093	**0.0022**	0.6000	0.0026	0.6013
	36	54	0.0170	0.2574	0.0076	0.2416	**0.0052**	0.2294
	60	90	0.0508	0.6385	0.0196	0.6168	**0.0114**	0.5874
	2000	2000	28.8608	20.6025	31.4104	**12.5071**	34.6850	12.5501
	3000	3000	77.3555	53.8899	82.6664	**28.8774**	106.8146	28.9161
	4000	4000	174.6793	109.0967	195.1640	**57.8293**	262.1968	58.1561
21	12	15	0.0099	0.6665	0.0021	0.7937	**0.0019**	0.8661
	36	54	0.0351	0.2977	0.0149	0.3345	**0.0079**	0.2751
	60	90	0.0714	0.8593	0.0252	0.7619	**0.0174**	0.7633
	2000	2000	33.7653	27.5496	34.7575	15.6777	37.9592	**15.576**
	3000	3000	108.381	71.7697	106.8036	**36.0209**	122.938	36.3343
	4000	4000	215.0848	145.2595	239.7968	**72.7266**	260.5157	73.3141
26	12	15	0.0069	0.8286	0.0028	0.9924	**0.0024**	1.0467
	36	54	0.0286	0.4159	0.0110	0.3546	**0.0079**	0.3377
	60	90	0.0841	1.0217	0.0288	1.1158	**0.0183**	0.9065
	2000	2000	42.5543	32.4614	40.1462	18.7319	47.8761	**18.5907**
	3000	3000	112.7396	84.3769	133.6703	42.7814	151.2720	**42.3202**
	4000	4000	258.6276	170.1170	302.4339	88.0312	365.4496	**86.8240**

Tabla 6.9: Comparación de tiempos de ejecución entre MAGMA con interface GPU y MKL, con 12, 16, 21 y 26 grupos estructurales Manivela-Barra (`NumGE`) y varios tamaños de matrices (`nEQTerminal` y `nEQManivela`). Hardware con 12 cores y 6 GPUs. Matrices simétricas con dispersión del 85 %

Se puede observar cómo las matrices pequeñas (con tamaños inferiores a 2000×2000) se calculan más rápidamente con MKL. Para tamaños mayores, MAGMA (combinando en sus rutinas el uso de CPU y GPU) obtiene los mejores resultados, siendo las combi-

naciones 6×2 y 12×1 las óptimas dado que ambas explotan las 6 GPUs de JUPITER. Recordamos que con el argumento `ArgLibrary`=5, el simulador cambia el número de threads OpenMP, pasando de los 12 indicados por el parámetro `ArgThreads` a 6, que corresponde con el número de GPUs, asignando los 6 threads restantes al paralelismo de la librería MKL (12×1 pasa a 6×2) que, aunque en este interface no realizan ningún cálculo, se mantiene por unificar el procedimiento empleado con el interface CPU.

La misma comparativa, esta vez con PARDISO en lugar de MKL, se puede consultar en la tabla 6.10. Se muestran los mejores resultados obtenidos empleando PARDISO y el interface GPU de MAGMA. Se observa que las matrices pequeñas (tamaños inferiores a 2000×2000) se calculan más rápidamente con PARDISO, pero MAGMA aprovecha mejor el uso combinado de CPU y GPU, y obtiene tiempos de ejecución notablemente menores cuando las matrices son grandes.

			th. OMP × th. PARDISO					
	nEQ		6 × 1		6 × 2		12 × 1	
numGE	Term	Maniv	PARDISO	MAGMA	PARDISO	MAGMA	PARDISO	MAGMA
12	12	15	**0.0209**	4.2053	0.0419	0.4584	0.0231	0.4497
	36	54	**0.0349**	0.1847	0.0495	0.2017	0.0353	0.1789
	60	90	0.0562	0.4453	0.0796	0.4764	**0.0473**	0.4648
	2000	2000	13.7537	9.9121	12.0941	**9.5609**	10.6448	9.6285
	3000	3000	31.7584	**21.8548**	26.6806	21.9079	24.433	22.1048
	4000	4000	62.1044	44.9134	52.4975	**44.4854**	49.6728	44.7893
16	12	15	0.0188	0.6381	0.0388	0.6000	**0.0171**	0.6013
	36	54	0.0385	0.2349	0.0496	0.2416	**0.0305**	0.2294
	60	90	0.0639	0.7461	0.0736	0.6168	**0.0501**	0.5874
	2000	2000	17.8819	12.6319	15.7097	**12.5071**	14.6420	12.5501
	3000	3000	42.5419	29.0103	35.3201	**28.8774**	35.1489	28.9161
	4000	4000	84.3465	59.3696	68.0332	**57.8293**	69.3354	58.1561
21	12	15	0.0228	0.7607	0.0436	0.7937	**0.0219**	0.8661
	36	54	0.0507	0.2631	0.0687	0.3345	**0.0431**	0.2751
	60	90	0.0807	0.7709	0.0893	0.7619	**0.0681**	0.7633
	2000	2000	22.3684	15.9098	20.09	15.6777	15.8987	**15.576**
	3000	3000	53.4410	36.3547	44.7195	**36.0209**	38.4195	36.3343
	4000	4000	102.8016	74.2192	84.3864	**72.7266**	74.0894	73.3141
26	12	15	**0.028**	1.1026	0.0528	0.9924	0.0208	1.0467
	36	54	0.0604	0.3243	0.0711	0.3546	**0.0444**	0.3377
	60	90	0.0981	1.1193	0.0995	1.1158	**0.0695**	0.9065
	2000	2000	26.6863	18.9614	22.1178	18.7319	20.4074	**18.5907**
	3000	3000	63.418	43.5602	52.0657	42.7814	49.4086	**42.3202**
	4000	4000	123.3044	88.5541	99.4763	88.0312	96.9931	**86.8240**

Tabla 6.10: Comparación de tiempos de ejecución entre MAGMA con interface GPU y PARDISO, con 12, 16, 21 y 26 grupos estructurales Manivela-Barra (`NumGE`) y varios tamaños de matrices (`nEQTerminal` y `nEQManivela`). Hardware con 12 cores y 6 GPUs. Matrices simétricas con dispersión del 85 %

6.5. Obtención de mejoras adicionales en el rendimiento mediante reparto de tareas

En las secciones anteriores, donde se realizaban experimentos con los interfaces CPU y GPU de la librería MAGMA, ideamos un artificio que limitaba el número de threads OpenMP al número de GPUs disponibles. De esta manera, cada hilo seleccionaba una GPU con la que realizar los cálculos por medio de MAGMA.

En este apartado probaremos potenciales mejoras en el rendimiento de la aplicación a través de una modificación que elimina la restricción anterior. El método simplemente consiste en que los thread OpenMP que no puedan reservar una GPU para los cálculos con MAGMA, usarán MKL o PARDISO para las operaciones de descomposición LU y resolución de los sistemas de ecuaciones. La elección de una otra librería dependerá del del factor de dispersión de las matrices (como reflejaba el algoritmo 6.2).

Experimentando en JUPITER, con 6 GPUs instaladas, esta situación se presenta cuando el parámetro `ArgThreads` que fija los threads OpenMP toma el valor 7 o superior. En esta ocasión probaremos con 12 threads (en la configuración 12×1), con matrices con 85 % de dispersión y utilizando los dos interfaces de MAGMA. La tabla 6.11 muestra los tiempos de ejecución logrados en los cuatro escenarios que se explican a continuación:

- (1) Se emplea el interface CPU, cambiando el valor de threads OpenMP de 12 a 6, para que sea igual al número de GPUs y cada uno pueda seleccionar una GPU para MAGMA. Los 6 cores restantes se asignarán a MKL. En este interface de MAGMA la descomposición LU de las matrices se calcula en la GPU (en un algoritmo híbrido que combina CPU y GPU). Sin embargo, la resolución del sistema de ecuaciones no está implementada, por lo que emplearemos la función DGETRS de la librería MKL para esta tarea, que se ejecuta en la CPU.
- (2) También usa el interface CPU, pero el valor de threads OpenMP se mantiene en 12, 6 de ellos usan MAGMA seleccionando una GPU cada uno. Los otros 6 threads usan PARDISO.
- (3) Se usa el interface GPU de MAGMA, cambiando el valor de threads OpenMP de 12 a 6, para que sea igual al número de GPUs. Los 6 cores restantes los usará MKL. En este interface MAGMA tiene implementadas tanto la descomposición LU como la resolución del sistema de ecuaciones, y en ambos casos la librería emplea un algoritmo híbrido que emplea recursos de la CPU y GPU.
- (4) Usando el interface CPU, pero en esta ocasión el valor de threads OpenMP se mantiene en 12, 6 de ellos usan MAGMA seleccionando una GPU cada uno. Los otros 6 threads usan PARDISO.

			Interface CPU		Interface GPU	
	nEQ		(1)	(2)	(3)	(4)
numGE	Term	Maniv	6 × 2	6 GPU + 6 PARDISO	6 × 2	6 GPU + 6 PARDISO
12	12	15	**0.0022**	0.0197	**0.4497**	4.0361
	36	54	**0.0075**	0.0201	0.1789	**0.1065**
	60	90	**0.0200**	0.0293	0.4648	**0.2285**
	2000	2000	21.5894	**14.3433**	9.6285	**8.7889**
	3000	3000	44.7031	**31.2024**	22.1048	**20.0108**
	4000	4000	81.1481	**60.5648**	44.7893	**42.3444**
16	12	15	**0.0028**	0.0046	0.6013	**0.3325**
	36	54	**0.0087**	0.0165	0.2294	**0.1407**
	60	90	**0.0288**	0.0320	0.5874	**0.3657**
	2000	2000	28.0445	**21.5429**	12.5501	**9.9266**
	3000	3000	57.7109	**46.7311**	28.9161	**23.8957**
	4000	4000	105.9838	**87.0157**	58.1561	**49.3556**
21	12	15	**0.0024**	0.0077	0.8661	**0.3816**
	36	54	**0.0149**	0.0193	0.2751	**0.1592**
	60	90	0.0384	**0.037**	0.7633	**0.4347**
	2000	2000	35.1641	**23.9077**	15.5760	**13.1037**
	3000	3000	74.0750	**50.2077**	36.3343	**32.5612**
	4000	4000	135.5642	**97.5703**	73.3141	**70.3011**
26	12	15	**0.0027**	0.0112	1.0467	**0.4498**
	36	54	**0.0132**	0.0322	0.3377	**0.2150**
	60	90	**0.0357**	0.0522	0.9065	**0.5125**
	2000	2000	42.6545	**29.5611**	18.5907	**14.8183**
	3000	3000	89.6608	**66.0383**	42.3202	**37.2712**
	4000	4000	164.0246	**124.8517**	86.8240	**76.0025**

Tabla 6.11: Tiempos de ejecución obtenidos usando los interfaces CPU y GPU de MAGMA, comparando entre limitar a 6 threads o mantener 12 (de los cuales sólo 6 usarán las GPU) para varios tamaños de matrices (`nEQTerminal` y `nEQManivela`) y 6, 12, 16, 21 y 26 grupos estructurales Manivela-Barra (`NumGE`). Hardware con 12 cores y 6 GPUs. Matrices simétricas con dispersión del 85 %

Como podemos observar, con matrices grandes (tamaños a partir de 2000×2000), y con independencia del interface de MAGMA empleado, se obtienen mejores tiempos de ejecución cuando se asignan 12 threads a OpenMP. Como hemos comentado, en este caso 6 threads van a utilizar MAGMA, asignándole una de las GPUs. Los otros 6 threads ejecutarán PARDISO para los cálculos.

Sin embargo, para tamaños pequeños, donde las GPU presentan menos ventajas debido a la sobrecarga que supone la transferencia de información entre la CPU y la GPU, podemos distinguir dos casos en función del tipo de interface de MAGMA empleado:

- Con el interface CPU, donde la descomposición LU se realiza con MAGMA (en la CPU y la GPU) y la resolución del sistema de ecuaciones en la CPU con MKL, la asignación de 2 threads al paralelismo interno de MKL sigue siendo la mejor configuración.
- Con el interface GPU, donde todo el trabajo se realiza con MAGMA (repartiendo trabajo entre la CPU y la GPU), es preferible ejecutar el mayor número de grupos en paralelo, compartiendo los trabajos entre las GPU y PARDISO, donde los threads con Id desde el 0 al 5 usarán GPU, y los que tiene Id del 6 al 11 usarán PARDISO.

6.6. Conclusiones

En las plataformas hardware donde hay instaladas GPUs *(Graphics Processor Units)* y los problemas a resolver requieren de métodos de álgebra lineal, puede resultar interesante utilizar librerías matriciales que exploten la capacidad de procesamiento de carácter general que ofrecen estas unidades gráficas, por ejemplo MAGMA.

Las GPUs necesitan tener la información almacenada en su memoria interna, por lo que los programas que las utilizan deben contemplar la tarea de transferencia de información entre la CPU y la GPU. Por este motivo, con carácter general, podemos concluir que su uso no ofrece ventajas cuando las matrices son de pequeñas dimensiones (tamaños menores de 1000×1000).

De los dos interfaces que ofrece MAGMA, interface CPU e interface GPU, el segundo ofrece todas las funciones de cálculo que se necesitan para resolver la plataforma de Stewart (descomposición LU y resolución de sistemas de ecuaciones). Los experimentos realizados con este interface han mejorado los tiempos logrados por MKL y PARDISO en sus mejores combinaciones de threads cuando las matrices son mayores de 1000×1000, pero no lo han hecho con matrices pequeñas.

Sin embargo, usando el interface CPU de MAGMA, donde sólo la descomposición LU se realiza en las GPU, sólo se obtienen mejoras respecto a MKL cuando las matrices son mayores de 3000×3000, pero nunca mejoran a PARDISO, dada su optimización para el trabajo con las matrices dispersas que corresponden a este tipo de problemas.

Por todo ello, el empleo de GPUs no es de interés en el tipo de problema de la plataforma de Stewart, con matrices dispersas de tamaños 12×12 para representar al Terminal y 15×15 para cada uno de los grupos Manivela-Barra.

Capítulo 7

Autotuning

A lo largo de los capítulos precedentes se ha podido comprobar que el valor de los parámetros de paralelismo que minimizan los tiempos de ejecución en el simulador están estrechamente relacionados con el tamaño del problema, representado por la dimensión de las matrices asociadas al problema cinemático, con el número de grupos estructurales paralelizables del mecanismo, y con el hardware sobre el que ejecuta la simulación.

Podemos concluir que una aplicación compilada con una determinada configuración de paralelismo y librería matricial, seleccionadas en base a un determinado hardware o tamaño de problema, probablemente penalizará el rendimiento de la aplicación cuando el entorno o el problema sean distintos.

Por este motivo en este capítulo se estudia un procedimiento que permita a la aplicación adaptar su configuración al entorno en el que se ejecute, y a los datos a manejar, proceso que se conoce como *autotuning*.

Las maneras de abordar esta tarea son diversas. Por ejemplo, si nos centramos en el aprendizaje del sistema:

1. Se podrían realizar en primer lugar ejecuciones sobre un conjunto de instalación, variando los parámetros hasta encontrar los valores óptimos de éstos.
2. O bien, en la primera ejecución real para un determinado tamaño de problema, se podrían realizar un conjunto de pequeños experimentos aplicando cambios en los parámetros hasta encontrar los que mejor se comportan en ese caso concreto.

Por otro lado, unos vez conocidos los valores óptimos de los parámetros, existen dos maneras de almacenarlos para su uso posterior:

1. Se pueden incluir en el mismo código, por lo que habría que recompilar la aplicación.
2. O bien se pueden almacenar en un archivo de manera que, en una ejecución real, se buscaría en ese archivo la configuración óptima para el escenario con un tamaño más parecido al del problema actual.

En esta Tesis hemos optado por experimentar con un conjunto de escenarios, que llamaremos de aprendizaje, para los que se buscarán las mejores configuraciones de parámetros. Los valores óptimos se almacenarán en un fichero que podrá ser consultado en futuras ejecuciones con problemas reales.

7.1. Modificaciones en el simulador

Recordamos de los capítulos anteriores que el simulador acepta como argumentos los parámetros de paralelismo recogidos en la tabla 7.1, que se aplican a su configuración interna antes de proceder a ejecutar la simulación de los escenarios.

parámetro	descripción
`ArgNumTests`	número de muestreo de tiempos de cada ejecución
`ArgLibrary`	librería de álgebra matricial a emplear en los cálculos
`ArgThreads`	número de threads disponible para paralelismo OpenMP
`ArgMKLThreads`	número de threads usados por MKL
`ArgNestedLevel`	nivel de paralismo anidado
`ArgLoop`	modo de distribución de las tareas en el bucle for paralelizado

Tabla 7.1: Lista definitiva de parámetros disponibles en el simulador de la Plataforma de Stewart

Con objeto de no tener que proporcionar de inicio estos parámetros, y que sea el software el que automáticamente seleccione los más adecuados, se han realizado dos modificaciones:

- Permitir un sólo argumento, con valor t, que indicará al simulador que debe ejecutar un proceso de búsqueda de parámetros óptimos para un conjunto de aprendizaje, probando con unos valores que se informan en un archivo específico para tal fin. Para ello se ha desarrollado en el marco de esta Tesis una librería en ANSI C que se encarga de preparar todas las combinaciones de parámetros y realizar llamadas a la función de FORTRAN que realiza las simulaciones.
- Permitir la ejecución sin argumentos, lo que indicará al simulador que debe buscar, en su base de datos de aprendizaje, la configuración de parámetros que corresponda al escenario (tamaño del problema) más cercano al que se pretende ejecutar.

7.2. Ejecución del simulador con el argumento t: búsqueda de parámetros de paralelismo óptimos

Cuando el simulador recibe el valor t como único argumento, se prepara para realizar un proceso de múltiples ejecuciones en busca de las configuraciones de parámetros óptimas para los escenarios propuestos. Para ello, necesita:

- Un archivo de escenarios, con la misma estructura descrita en la sección 3.1. Este fichero puede contener más de un escenario, en cuyo caso el simulador los procesará todos, buscando los mejores parámetros para cada uno de ellos.
- Uno o más archivos de autotuning que contengan las combinaciones de parámetros que se quieren probar. Este fichero debe contener la siguiente información:
  - Un campo de texto con una descripción
  - Un grupo de parámetros que contiene la lista de todos ellos. Cada uno debe estar especificado por su nombre, tipo de dato y los valores con los que se va a experimentar. Los tipos admitidos son i para números enteros y f para reales. Los valores se pueden especificar como una lista de uno o más números, o bien como un rango (especificado por tres valores: desde, hasta y salto). La tabla 7.2 muestra un ejemplo de la información almacenada en un archivo de autotuning.

	Script Name	First		
	parameterName	parameterValueRange	parameterValue	parameterFormat
parameterList	OMP	{"1","3","1"}		i
	MKL		{"1","3"}	i
	LOOP		{"0"}	i
	LIBRARY		{"1","2","3"}	i
	NESTED	{"1","2","1"}		i

Tabla 7.2: Contenido de los ficheros de autotuning usados por el simulador en la búsqueda de parámetros óptimos

Como hemos comentado, la gestión de este proceso de búsqueda de los mejores parámetros es realizada por la nueva librería desarrollada en C. El proceso es el siguiente:

- La librería lee el archivo de parámetros y, para aquéllos que especifiquen los valores en formato de rango, los divide en sus valores individuales. Por ejemplo, en el caso del parámetro OMP que figura en la tabla 7.2 con la cadena {"1,","3","1"}, se obtendrán los valores discretos que van desde 1 hasta 3, con saltos de 1 unidad. Por tanto, la librería genera {1}, {2} y {3}.
- Con los valores de los parámetros ya discretizados se generan todas las posibles combinaciones entre ellos.
- Para cada escenario (que representa el tamaño del problema), la librería realizará tantas llamadas al simulador FORTRAN como combinaciones de parámetros se han obtenido en el paso anterior, tomando los tiempos de ejecución de cada ejecución.
- Se ordenan todos los resultados y se almacena en el archivo autotuning_data_base.csv la información relativa a los datos del escenario y a los parámetros que consiguen el menor tiempo de ejecución, como muestra la tabla 7.3.

El fichero generado puede ser detectado en ejecuciones posteriores, y la información contenida en él será utilizada para determinar los parámetros paralelos óptimos.

	campo	descripción
Escenarios	`tfin`	Número de ejecuciones
	`tfin2`	Número de iteraciones para resolver la posición
	`numGE`	Número de Grupos Manivela-Barra
	`nEQTerminal`	Tamaño de la matriz del Terminal
	`nEQManivela`	Tamaño de la matriz de los grupos Manivela-Barra
	`nEQMatMul`	Tamaño de la matriz de puntos de interés adicionales
	`neslabones`	Número de eslabones del mecanismo
	`sparsity`	Dispersión de las matrices (expresada en %)
	`symmetric`	Tipo de matrices (0: no simétricas, 1: simétricas)
Parámetros	`OMP`	Threads asignados a OpenMP
	`MKL`	Threads asignados a MKL
	`LOOP`	Tipo de scheduling
	`LIBRARY`	Identificador de librería de álgebra matricial
	`NESTED`	Nivel de paralelismo anidado

Tabla 7.3: Contenido del fichero creado para su uso durante el autotuning: parámetros de paralelismo óptimos para cada tamaño de problema en el simulador de la Plataforma de Stewart

Para distinguirlos, los nombres de los parámetros usados el autotuning son diferentes que los que usa el simulador. Por ejemplo, `OMP` corresponde a `ArgThreads`, `MKL` a `ArgMKLThreads`, `LOOP` a `ArgLoop`, `LIBRARY` a `ArgLibrary` y `NESTED` a `ArgNestedLevel`.

En la figura 7.1 se representa un sencillo ejemplo donde tres parámetros pueden adoptar varios valores. En el caso del parámetro `OMP` se trata de un rango de valores entre 1 y 3. El parámetro `MKL` toma los valores 1 y 3, y `LOOP` tiene el 0 como único valor. Las combinaciones de los valores individuales de cada uno dan lugar a 6 juegos de parámetros.

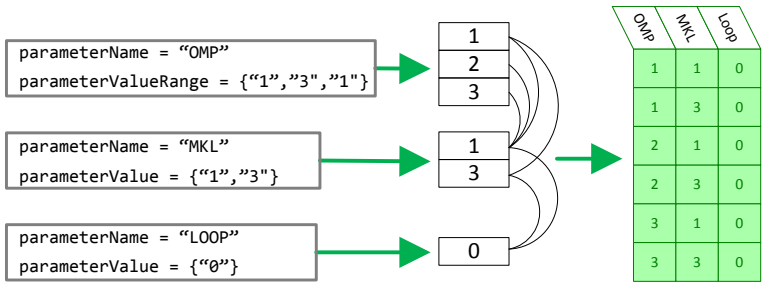


Figura 7.1: Fichero de autotuning usado en el cálculo de parámetros óptimos: descomposición en valores discretos y combinación de valores

7.3. Ejecución del simulador sin argumentos: autotuning

Cuando el simulador no recibe ningún argumento de entrada, debe elegir los parámetros del paralelismo que mejor se adapten al tamaño de problema a simular, especificado en el archivo de escenarios.

Este proceso de autotuning se representa gráficamente en la figura 7.2. En el gráfico se muestran tres escenarios, cada uno con sus tamaños de matrices, número de eslabones y factor de dispersión, como indica el eje X. El eje Y refleja los valores de cada componente, en cada escenario. En la tabla superior derecha se muestran los parámetros de paralelismo óptimos en cada caso.

El primer objetivo del simulador es encontrar los parámetros paralelos que mejor se adaptan al escenario actual, representado en la figura por la línea de trazo grueso. Para ello, el proceso será buscar, entre los escenarios almacenados en el fichero autotuning_data_base.csv, el más parecido al escenario actual.

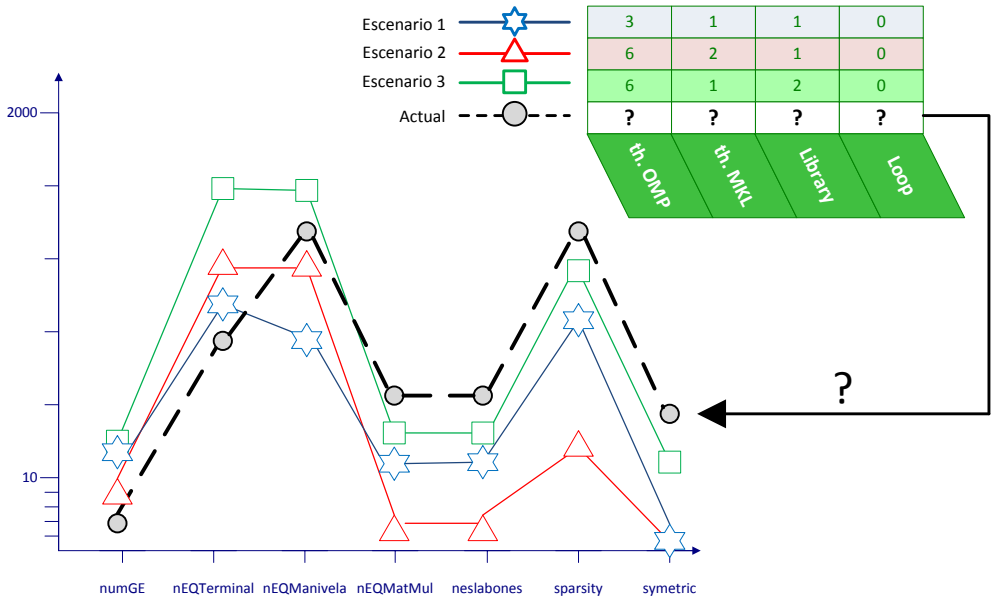


Figura 7.2: Concepto de autotuning en el simulador de la plataforma de Stewart

El proceso consiste en sumar los cuadrados de las distancias relativas entre el escenario actual y los almacenados en el fichero de autotuning, para cada componente (`numGE`, `nEQTerminal`, `nEQManivela`, `nEQMatMul`, `neslabones`, `sparsity` y `symmetric`). El escenario que ofrezca el menor valor de distancia será adoptado como el más parecido al problema en curso, y sus parámetros paralelos se aplicarán para resolver el escenario actual.

La figura 7.3 muestra un ejemplo del cálculo de distancias mínimas para el caso de tener tres escenarios en la base de datos, utilizando únicamente los tamaños de la matriz del terminal (`nEQTerminal`) y de los grupos Manivela-Barra (`nEQManivela`).

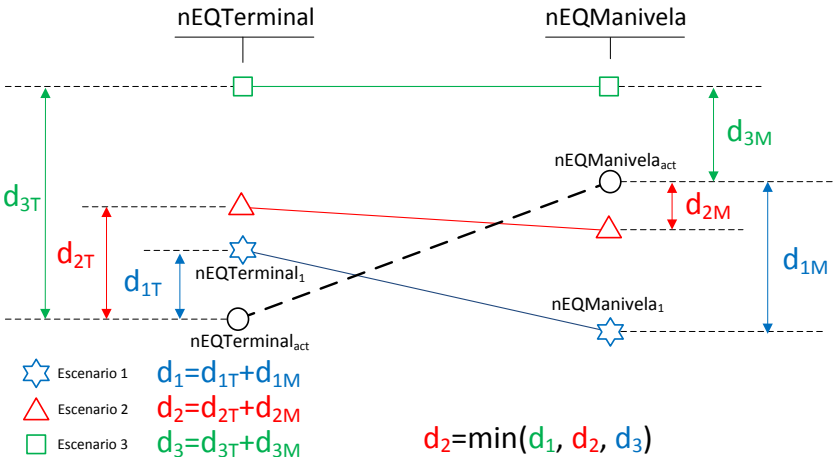


Figura 7.3: Autotuning en el simulador de la plataforma de Stewart: búsqueda de distancias relativas mínimas

En este ejemplo, d_{1T} y d_{1M} representan las distancias relativas entre el tamaño de las matrices del terminal y grupos Manivela-Barra del escenario actual y el escenario número 1. Por tanto,

$$d_{1T} = \left(\frac{\texttt{nEQTerminal}_1 - \texttt{nEQTerminal}_{act}}{\texttt{nEQTerminal}_1} \right)^2$$

$$d_{1M} = \left(\frac{\texttt{nEQManivela}_1 - \texttt{nEQManivela}_{act}}{\texttt{nEQManivela}_1} \right)^2$$

$$d_1 = d_{1T} + d_{1M}$$

Del mismo modo se calcularían d_2 y d_3, las distancias respecto a los escenarios 2 y 3 respectivamente. El escenario seleccionado será el que ofrezca la distancia mínima $min(d_1, d_2, d_3)$ y los parámetros del paralelismo que fueron óptimos para ese escenario se aplicarán ahora al escenario actual. Tras ello se iniciará la simulación.

7.4. Prueba del método de autotuning

Con objeto de comprobar el correcto funcionamiento de este método, hemos realizado un test en JUPITER consistente en:

- Seleccionar un juego de escenarios de aprendizaje. Este consiste en varios tamaños de problema, representados por las dimensiones de las matrices y el número de grupos estructurales. La tabla 7.4 muestra los valores seleccionados para esta prueba.

- Definir los parámetros con los que se va a experimentar. Para este ejercicio en concreto hemos limitado el ejercicio a paralelismo en entornos multicore, con las tres librerías conocidas, MKL, PARDISO y MA27. En base al conocimiento adquirido en las ejecuciones en JUPITER durante la presente Tesis, se han seleccionado combinaciones de threads OpenMP y MKL que se conocen eficientes, y se han distribuido en 4 ficheros de autotuning, que se pueden consultar en las tablas 7.5, 7.6, 7.7 y 7.8.

tfin	tfin2	numGE	nEQTerminal	nEQManivela	nEQMatMul	neslabones	sparsity	symmetric
10	3	6	12	18	3	3	85	1
10	3	6	36	54	3	3	85	1
10	3	6	60	90	3	3	85	1
10	3	6	400	400	3	3	85	1
10	3	6	1000	1000	3	3	85	1
10	3	6	2000	2000	3	3	85	1
10	3	6	3000	3000	3	3	85	1
10	3	12	12	18	3	3	85	1
10	3	12	36	54	3	3	85	1
10	3	12	60	90	3	3	85	1
10	3	12	400	400	3	3	85	1
10	3	12	1000	1000	3	3	85	1
10	3	12	2000	2000	3	3	85	1
10	3	12	3000	3000	3	3	85	1

Tabla 7.4: Juegos de escenarios de aprendizaje para el proceso de autotuning en el simulador de la Plataforma de Stewart

	Name	fichero#1		
	parameterName	parameterValueRange	parameterValue	parameterFormat
parameterList	OMP	{"1","11","1"}		i
	MKL		{"1"}	i
	LOOP		{"0"}	i
	LIBRARY		{"1","2","3"}	i
	NESTED		{"2"}	i

Tabla 7.5: Fichero #1 de aprendizaje para el proceso de autotuning en el simulador de la Plataforma de Stewart

	Name	fichero#2		
	parameterName	parameterValueRange	parameterValue	parameterFormat
parameterList	OMP		{"6"}	i
	MKL	{"1","2","1"}		i
	LOOP		{"0"}	i
	LIBRARY		{"1","2","3"}	i
	NESTED		{"2"}	i

Tabla 7.6: Fichero #2 de aprendizaje para el proceso de autotuning en el simulador de la Plataforma de Stewart

	Name	fichero#3		
	parameterName	parameterValueRange	parameterValue	parameterFormat
parameterList	OMP	{"2","3","1"}		i
	MKL		{"4","6"}	i
	LOOP		{"0"}	i
	LIBRARY		{"1","2","3"}	i
	NESTED		{"2"}	i

Tabla 7.7: Fichero #3 de aprendizaje para el proceso de autotuning en el simulador de la Plataforma de Stewart

	Name	fichero#4		
	parameterName	parameterValueRange	parameterValue	parameterFormat
parameterList	OMP		{"12"}	i
	MKL	{"1","3","1"}		i
	LOOP		{"0"}	i
	LIBRARY		{"1","2","3"}	i
	NESTED		{"2"}	i

Tabla 7.8: Fichero #4 de aprendizaje para el proceso de autotuning en el simulador de la Plataforma de Stewart

Una vez especificados los escenarios y los parámetros, se lanza la ejecución del simulador proporcionando el argumento t. Con ello se inicia un proceso de generación de todas las combinaciones de parámetros de los cuatro ficheros, y la posterior experimentación con todos los escenarios. Los menores tiempos de ejecución se registran en el fichero autotuning_data_base.csv, relacionando de esta manera el tamaño del problema (escenario) con la configuración óptima del simulador (parámetros). La tabla 7.9 muestra el contenido de este fichero tras la ejecución en JUPITER.

Escenarios									Parámetros óptimos				
tfin	tfin2	numGE	nEQTerminal	nEQManivela	nEQMatMul	neslabones	sparsity	symmetric	OMP	MKL	LOOP	LIBRARY	NESTED
10	3	6	12	18	3	3	85	1	6	2	0	3	2
10	3	6	36	54	3	3	85	1	8	1	0	3	2
10	3	6	60	90	3	3	85	1	7	1	0	3	2
10	3	6	400	400	3	3	85	1	6	1	0	3	2
10	3	6	1000	1000	3	3	85	1	6	2	0	3	2
10	3	12	12	18	3	3	85	1	8	1	0	3	2
10	3	12	36	54	3	3	85	1	12	1	0	3	2
10	3	12	60	90	3	3	85	1	12	1	0	3	2
10	3	12	400	400	3	3	85	1	12	3	0	3	2
10	3	12	1000	1000	3	3	85	1	12	2	0	3	2

Tabla 7.9: Contenido del fichero autotuning_data_base.csv, con los parámetros paralelos óptimos para diversos tamaños de problema en JUPITER, obtenidos por el simulador en la fase de aprendizaje

Una vez creada la base de datos, el simulador ya pueda ajustar automáticamente sus parámetros para la siguiente ejecución, cuando se le presente un nuevo tamaño de problema. Para comprobar la eficacia del método de autotuning, elegimos dos escenarios que representan sistemas multicuerpo ligeramente diferentes a aquéllos con los que hemos generado la base de datos. La tabla 7.10 muestra estos escenarios.

tfin	tfin2	numGE	nEQTerminal	nEQManivela	nEQMatMul	neslabones	sparsity	symmetric
10	3	6	300	450	3	3	85	1
10	3	16	300	450	3	3	85	1

Tabla 7.10: Escenarios de prueba para ejecutar con autotuning en el simulador

Al ejecutar el simulador sin argumentos, el software busca un problema similar en la base de datos. Para ello usa el método de las distancias relativas mínimas explicado anteriormente en este capítulo.

En la tabla 7.11 observamos el cálculo de las distancias entre el primero de los dos problemas de prueba y el resto de escenarios almacenados en la base de datos, de los cuales conocemos los parámetros paralelos óptimos. El cálculo se realiza exclusivamente con los componentes `numGE`, `nEQTerminal` y `nEQManivela`, ya que el resto (`tfin`, `tfin2`, `nEQMatMul`, `neslabones`, `sparsity` y `symmetric`) son iguales en todos los casos, por lo que su distancia tiene un valor de 0 y no afecta al cálculo.

	Base de datos			Actual			Distancias relativas			
#	numGE	nEQTerminal	nEQManivela	numGE	nEQTerminal	nEQManivela	numGE	nEQTerminal	nEQManivela	Total
1	6	12	18	6	300	450	0.00	576.00	576.00	1152.00
2	6	36	54				0.00	53.78	53.78	107.56
3	6	60	90				0.00	16.00	16.00	32.00
4	6	400	400				0.00	0.06	0.02	**0.08**
5	6	1000	1000				0.00	0.49	0.30	0.79
6	12	12	18				0.25	576.00	576.00	1152.25
7	12	36	54				0.25	53.78	53.78	107.81
8	12	60	90				0.25	16.00	16.00	32.25
9	12	400	400				0.25	0.06	0.02	0.33
10	12	1000	1000				0.25	0.49	0.30	1.04

Tabla 7.11: Cálculo de las distancias relativas entre todos los escenarios almacenados en la base de datos y el primer problema de test, para los componentes `numGE`, `nEQTerminal` y `nEQManivela`

Observamos que en este primer escenario de prueba, la distancia mínima se produce respecto al escenario de la base de datos identificado por el número 4, con una distancia de 0.08. Los parámetros de paralelismo asociados al escenario 4, que se pueden consultar en la tabla 7.9, son 6 threads OpenMP y 1 thread MKL, y son los que el simulador va a usar para el nuevo escenario.

Los mismo cálculos realizados sobre el segundo escenario de prueba nos indican que la distancia mínima se da respecto al escenario 9, como podemos comprobar en la tabla 7.12, con un valor de 0.19, y que corresponde a unos parámetros paralelos de 12 threads OpenMP y 3 threads MKL.

	Base de datos			Actual			Distancias relativas			
#	numGE	nEQTerminal	nEQManivela	numGE	nEQTerminal	nEQManivela	numGE	nEQTerminal	nEQManivela	Total
1	6	12	18	16	300	450	2.78	576.00	576.00	1154.78
2	6	36	54				2.78	53.78	53.78	110.33
3	6	60	90				2.78	16.00	16.00	34.78
4	6	400	400				2.78	0.06	0.02	2.86
5	6	1000	1000				2.78	0.49	0.30	3.57
6	12	12	18				0.11	576.00	576.00	1152.11
7	12	36	54				0.11	53.78	53.78	107.67
8	12	60	90				0.11	16.00	16.00	32.11
9	12	400	400				0.11	0.06	0.02	**0.19**
10	12	1000	1000				0.11	0.49	0.30	0.90

Tabla 7.12: Cálculo de las distancias relativas entre todos los escenarios almacenados en la base de datos y el segundo problema de test, para los componentes `numGE`, `nEQTerminal` y `nEQManivela`

La tabla 7.13 realiza una comparación en los dos escenarios de prueba, indicando los parámetros de threads OpenMP y threads MKL que sugiere el proceso de autotuning, frente a los óptimos, obtenidos al lanzar el simulador con el parámetro t.

			Autotuning		Óptimos	
numGE	nEQTerminal	nEQManivela	th. OpenMP	th. MKL	th. OpenMP	th. MKL
6	300	450	6	1	6	1
16	300	450	12	3	12	2

Tabla 7.13: Fiabilidad del proceso de autotuning por cálculo de distancias mínimas relativas. Comparación entre parámetros recomendados y parámetros óptimos

Se puede comprobar que, en el primer caso, donde el número de grupos estructurales coincide con el que hay en los escenarios de aprendizaje, el método se muestra muy eficaz, con una coincidencia exacta entre los parámetros paralelos sugeridos y los óptimos. El segundo escenario representa un sistema mecánico con 16 grupos estructurales. Los escenarios más parecidos en la base de datos contienen 12 grupos, que son los que ha seleccionado el proceso de autotuning. En esta ocasión, el método ha acertado con el número de threads OpenMP, y se ha desviado ligeramente en los threads MKL.

7.5. Conclusiones

Un proceso de autotuning permite el ajuste automático de la configuración paralela óptima para un determinado tipo de hardware y frente a un tipo de problema concreto. En este capítulo, se ha mostrado cómo esta técnica puede conseguir reducir notablemente los tiempos de ejecución.

En general, el proceso abarca dos fases:

- Una primera, de aprendizaje, donde se recaban datos de ejecución para un conjunto de escenarios decididos a priori. Esta información alimenta una base de datos con los resultados del aprendizaje.
- Una segunda, de ejecución, donde dado un problema concreto a resolver (con un escenario concreto), se decide qué valores utilizar para los parámetros de ejecución configurables teniendo en cuenta toda la información obtenida y almacenada en la fase de aprendizaje.

De esta manera, el tiempo de cálculo empleado durante la fase de aprendizaje no es estéril, ya que es recuperado en las siguientes ejecuciones, por medio de ahorros futuros en los tiempos de ejecución.

La mayor o menor eficacia de este método depende de poder encontrar entre los resultados del aprendizaje el escenario correcto, el más equiparable al problema a resolver. Para ello, la elección del conjunto de aprendizaje es clave, y esto supone decidir el número de escenarios (tamaños de problema) que debe abarcar. Obviamente, cuanto mayor sea ese número, más tiempo se deberá invertir en el aprendizaje, pero más fácil será encontrar un escenario similar al problema concreto. Un número reducido de escenarios de aprendizaje, por contra, se podrá evaluar sin invertir mucho tiempo, pero quizás sea difícil luego encontrar el mejor ajuste a nuestro tamaño de problema.

Además de la correcta elección del conjunto de escenarios de aprendizaje, otro factor relevante es el algoritmo de búsqueda en el conjunto de resultados. En esta Tesis hemos desarrollado un método basado en distancias mínimas, comparando las variables que representan los tamaños del problema (dimensión de las matrices y número de grupos estructurales), considerando que todas ejercen la misma influencia. Este método simple funciona correctamente cuando hay cierta proporcionalidad entre los tamaños de los escenarios de aprendizaje y los reales, pero habría que estudiar métodos más avanzados que aportaran generalidad.

Capítulo 8

Conclusiones y trabajos futuros

En este capítulo se van a comentar las principales conclusiones derivadas del estudio realizado a lo largo de los capítulos precedentes de este documento. Fruto de ello, se han detectado también futuras vías de trabajo que permitirían avanzar en la aplicación de paralelismo a problemas en el ámbito de la simulación de sistemas mecánicos.

8.1. Conclusiones

Durante los capítulos precedentes se ha trabajado en implementar técnicas paralelas sobre el simulador de sistemas multicuerpo desarrollado partiendo del software controlador de una plataforma de Stewart. Se ha analizado el efecto que ejercen sobre los tiempos de ejecución las diversas configuraciones de threads en plataformas multicore empleando OpenMP. También se ha experimentado con varias librerías de álgebra matricial, algunas de las cuales son capaces de emplear paralelismo en las rutinas que resuelven los sistemas de ecuaciones que se plantean en la simulación de sistemas mecánicos. Finalmente se han extendido los experimentos a plataformas hardware que disponen de GPUs, combinando los cálculos entre la CPU y las GPUs.

Las sucesivas modificaciones en el código fuente, encaminadas a manejar las nuevas configuraciones paralelas, han llevado emparejadas la creación de parámetros para activar o desactivar las diferentes opciones. Estos parámetros se han almacenado en archivos, lo que ha permitido poder realizar múltiples simulaciones por lotes, y obtener una extensa base de experimentos y resultados.

Podemos concluir que:

- De manera genérica se puede afirmar que, en los casos en los que el modelado de sistemas mecánicos puede identificar grupos estructurales cuyos cálculos puedan realizarse de manera simultánea, la aplicación de paralelismo al software de simulación ofrece mejoras de rendimiento respecto al enfoque secuencial.
- En sistemas multicore donde no se aplica paralelismo explícito, el uso de librerías que incorporan algoritmos paralelos en sus rutinas presenta notables ventajas,

pero se ha comprobado que en los tamaños de problema de la plataforma de Stewart donde las matrices son de naturaleza dispersa, una librería especializada, como es MA27, es notablemente más eficaz, aún sin contar con implementación paralela,

- Con tamaños de problema medianos o grandes, librerías comerciales, como MKL para matrices densas o PARDISO para matrices dispersas, consiguen reducir aún más los tiempos de ejecución aprovechando su paralelismo interno.

- Se ha comprobado que el factor de dispersión (porcentaje de valores nulos respecto al total de elementos de la matriz) que recomienda el uso de librerías con solver dispersos, se encuentra notablemente por encima del 50 %. En el caso de la plataforma de Stewart, las matrices que surgen de una formulación basada en grupos estructurales presentan un factor de dispersión superior al 70 %.

- El uso de paralelismo OpenMP, donde es posible asignar threads para la resolución de cada grupo estructural, representa un importante avance en la mejora de las prestaciones, especialmente notable si se dispone de suficientes cores en la CPU para que sea eficiente la creación de tantos threads como grupos.

- En el caso de quedar recursos libres, es posible implementar una estrategia de paralelismo en dos niveles, asignando nuevos threads a las rutinas internas de las librerías desde los hilos de OpenMP. Se ha observado que MKL presenta un mejor rendimiento con un balance mixto entre threads OpenMP y threads MKL, pero en el caso de PARDISO el rendimiento es mejor aumentando los de OpenMP.

- El uso combinado del poder de cómputo de la CPU y las GPUs (*Graphics Processor Units*) que se realiza en librerías como MAGMA ha mostrado también potencial de conseguir mejoras en los tiempos de ejecución, pero sólo cuando las matrices son de grandes dimensiones, por encima de 2000×2000, lo cual no es aplicable en el problema de la plataforma de Stewart, pero podría serlo en la simulación de sistemas mecánicos más complejos.

- Gracias a los experimentos realizados en este trabajo, se ha observado que la tarea de seleccionar los parámetros de paralelismo adecuados está muy relacionada con el hardware y las librerías empleadas para los cálculos. A su vez, la elección de una u otra librería está estrechamente relacionada con el tamaño del problema. Por tanto, la puesta en marcha de un proceso de autotuning para la elección de parámetros, tomando como base el conocimiento obtenido a través de experimentos preliminares, se ha mostrado muy eficaz.

8.2. Trabajos futuros

El desarrollo del presente trabajo plantea nuevos caminos para avanzar en la búsqueda de las mejores técnicas paralelas aplicables al software de simulación y control de robots paralelos. Podemos mencionar los siguientes:

- El estudio del balanceo entre los threads y el número de grupos estructurales que pueden ser calculados simultáneamente, el cual presenta retos que en este trabajo han quedado introducidos en un fase muy preliminar en las secciones dedicadas al estudio de la influencia del aumento del número de grupos. El scheduling, con una adecuada elección de threads asignados a OpenMP y a las rutinas internas de las librerías, puede ser determinante en la obtención de reducciones adicionales en los tiempos de ejecución.

- El estudio de otras librerías que implementen solvers dispersos, más adecuados que los densos para el tipo de matrices que se manejan en este tipo de problemas. En concreto, sería de interés el estudio de las funciones para matrices dispersas sobre GPU que ofrece MAGMA.

- El salto a un esquema de paso de mensajes, para repartir los cálculos entre todos los recursos de un cluster, incluyendo quizás GPUs. Con ello se aumentaría todavía más el poder de computación para la simulación de sistemas multicuerpo complejos.

- Profundizar en la aplicación de técnicas paralelas en pequeños dispositivos, como son Raspberry y ODROID, algunos de los cuales ya disponen de CPU con 8 cores. Su reducido tamaño los hace merecedores de estudio como posibles unidades de control en tiempo real, donde se requiere, además, un consumo de energía reducido [49]. Y una red de este tipo de equipos se podrían usar en la simulación de sistemas mecánicos más complejos, empleando paso de mensajes.

- Perfeccionar el mecanismo de búsqueda de parámetros paralelos óptimos. En concreto, avanzar en métodos de búsqueda alternativos al exhaustivo.

- Añadir nuevos métodos de búsqueda en la base de datos de escenarios óptimos. El método de distancias relativas mínimas empleado en este trabajo es sólo una manera sencilla de comenzar. Una versión más elaborada permitiría asignar pesos a cada componente para, por ejemplo, dar más importancia a la proximidad en el número de grupos paralelos, o al factor de dispersión, que al tamaño mismo de las matrices. Se podrían implementar las nuevas estrategias en el simulador, y lanzar ejecuciones para comprobar la eficacia de cada una de ellas.

Bibliografía

[1] AMD fusion.
http://www.amd.com/en-us/press-releases/Pages/amd-fusion-apu-era-2011jan04.aspx.

[2] BLACS.
http://www.netlib.org/blacs/.

[3] BLAS (basic linear algebra subprograms).
http://www.netlib.org/blas/.

[4] BULLET: Real Time Physics Simulation.
http://bulletphysics.org/wordpress/.

[5] cuBLAS.
https://developer.nvidia.com/cublas.

[6] CUDA Parallel Computing Platform.
http://www.nvidia.com/object/cuda_home_new.html.

[7] DART: Dynamic Animation and Robotic Toolkit.
https://dartsim.github.io/.

[8] GAZEBO: Rotot simulation made easy.
http://gazebosim.org/.

[9] HSL (2013). A collection of Fortran codes for large scale scientific computation.
http://www.hsl.rl.ac.uk/.

[10] HSL_ARCHIVE: MA27.
http://www.hsl.rl.ac.uk/archive/specs/ma27.pdf.

[11] HSL_ARCHIVE: MA57, Sparse symmetric system: multifrontal method.
http://www.hsl.rl.ac.uk/catalogue/ma57.html.

[12] HSL_ARCHIVE: MA86, Sparse symmetric indefinite system using OpenMP.
http://www.hsl.rl.ac.uk/catalogue/hsl_ma86.html.

[13] HSL_ARCHIVE: MA87, Sparse symmetric positive-definite system using OpenMP.
http://www.hsl.rl.ac.uk/catalogue/hsl_ma87.html.

[14] HSL_ARCHIVE: MA97, Sparse symmetric using OpenMP.
http://www.hsl.rl.ac.uk/catalogue/hsl_ma97.html.

[15] Intel© Fortran Compiler.
https://software.intel.com/en-us/fortran-compilers.

[16] Intel© Many Integrated Core Architecture.
http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[17] Intel© Math Kernel Library (intel© MKL).
https://software.intel.com/en-us/intel-mkl.

[18] Intel© MKL PARDISO.
https://software.intel.com/en-us/node/470282.

[19] Intel© Thread Affinity Iinterface.
https://software.intel.com/en-us/node/522691.

[20] Lapack: Linear algebra package.
http://www.netlib.org/lapack/.

[21] Linux.
http://www.linuxfoundation.org/.

[22] Maple: Essential Tool for Mahtematics.
https://www.maplesoft.com/products/maple/.

[23] Maplesim: Advanced System-level Modeling.
http://www.maplesoft.com/products/maplesim/.

[24] MATLAB.
https://es.mathworks.com/products/matlab.html.

[25] MATLAB Distributed Computing Server.
https://es.mathworks.com/products/distriben.html.

[26] MATLAB Parallel Computing Toolbox.
https://es.mathworks.com/products/parallel-computing.html.

[27] MATLAB Robotics System Toolbox.
https://es.mathworks.com/products/robotics.html.

[28] MATLAB Simspace Multibody.
https://es.mathworks.com/products/simmechanics.html.

[29] MATLAB Simulink.
https://es.mathworks.com/products/simulink.html.

[30] Matrix Algebra on GPU and Multicore Architectures.
http://icl.cs.utk.edu/magma/.

[31] Newton Dynamics.
http://newtondynamics.com/forum/newton.php.

[32] ODE: Open Dynamics Engine.
http://www.ode.org/.

[33] Open MPI: Open Source High Performance Computing.
http://www.open-mpi.org/.

[34] The open standard for parallel programming of heterogeneous systems.
https://www.khronos.org/opencl/.

[35] The OpenMP© API specification for parallel programming.
http://openmp.org/wp//.

[36] Parallel Basic Linear Algebra Subprograms (PBLAS).
http://www.netlib.org/scalapack/pblas_qref.html.

[37] Parallelism in the intel© Math Kernel Library.
https://software.intel.com/en-us/articles/parallelism-in-the-intel-math-kernel-library.

[38] ScaLAPACK - Scalable Linear Algebra PACKage.
http://www.netlib.org/scalapack/.

[39] Simbody: Multibody Physics API.
https://simtk.org/projects/simbody/.

[40] Universidad de Murcia. Grupo de Computación Científica y Programación Paralela.
http://luna.inf.um.es/grupo_investigacion/.

[41] Universidad Politécnica de Cartatena. Dpto. de Ingeniería Mecánica.
http://dimec.upct.es/.

[42] V-REP: virtual robot experimentation platform.
http://www.coppeliarobotics.com/.

[43] Vienna Computing Library.
http://viennacl.sourceforge.net/viennacl-about.html.

[44] A. Almeida, Giménez D., Mantas J.M., and A.M. Vidal. *Introducción a la Programación Paralela*. Paraninfo, 1st edition, 2008.

[45] AMD. AMD Athlon™64 × 2 technical specifications.
http://support.amd.com/TechDocs/33425.pdf, 2007.

[46] 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.

[47] G. Bernabé, J. Cuenca, L.P. García, and D. Giménez. Tuning basic linear albegra routines for hybrid CPU+GPU platforms. In *ICCS, Procedia Computer Science*, volume 29, pages 30–39, 2014.

[48] G. Bernabé, J. Cuenca, L.P. García, and D. Giménez. Auto-tuning techniques for linear algebra routines on hybrid platforms. *Journal of Computational Science*, 2015.

[49] G. Bernabé, J.C. Cano, J. Cuenca, A. Flores, D. Giménez, M. Saura, and P. Segado. Exploiting Hybrid Parallelism in the Kinematic Analysis of Multibody Systems based on Group Equations. In *The International Conference on Computational Science*, June 12-14, 2017.

[50] F. Cajori. Historical Note on the Newton-Raphson Method of Approximation. *The American Mathematical Monthly*, 18:29–32, 1911.

[51] 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.

[52] J. Cuadrado. Análisis de Mecanismos por Ordenador. Capítulo 1. http://lim.ii.udc.es/docencia/phd-meccomp/capitulo1.pdf, 2017.

[53] J. Cuenca, L.P. García, and D. Giménez. On optimization techniques for the matrix multiplication on hybrid CPU+GPU platforms. *Annals of MMulticore and GPU Programming*, 1:1–8, 2014.

[54] J. Cuenca, L.P. García, D. Giménez, and F.J. Herrera. Empiricial Modeling: an auto-tuning method for linear algebra routines on CPU plus Multi.GPU Platforms. In *CMMSE*, 2016.

[55] J. Cuenca, L.P. García, D. Giménez, and F.J. Herrera. Guided installation of basic linear algebra routines in clusters with Manycore components. In *SIAM PPSC - MS Auto-Tuning for the post Moore's era*, 2016.

[56] Grupo de Computación Científica y Programación Paralela. Guía rápida del clúster heterosolar. `http://luna.inf.um.es/grupo_investigacion.html`, 2014.

[57] S. Durango, J. Correa, and O.E. Ruiz. Graph-based structural analysis of planar mechanisms. *Meccanica*, 52:441–455, 2017.

[58] A. Galantai. The theory of Newton's method. *Journal of Computational and Applied Mathematics*, 124:25–44, 2000.

[59] Intel©. Legacy Intel© Core™ 2 Processor. https://ark.intel.com/products/family/79667/Legacy-Intel-Core2-Processor, 2017.

[60] J. Jalon. *Kinematic and dynamic simulation of multibody systems*. Springer Verlag, NY, 1993.

[61] D. Lazard and J.P. Merlet. The (true) Stewart platform has 12 configurations. In *IEEE International Conference on Robotics and Automation*, page 2160, 1994.

[62] K. Magnus. *Dynamics of multibody systems*. Springer Verlag, 1978.

[63] J.P. Merlet. *Parallel Robots*. Springer, 2nd edition, 2008.

[64] Y.A. Semenov M.Z. Zolovsky, A.N. Evgrafov and A.V. Solusch. *Advanced theory of mechanism and machines*. Springer, 2000.

[65] NVIDIA. CUDA 8 Features Revealed. https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed.

[66] NVIDIA. CUDA Toolkit v.7.5. http://developer.download.nvidia.com/compute/cuda/7.5.

[67] A. Ollero. *Robótica: manipuladores y robots móviles*. Marcombo, 2001.

[68] Berkeley Lab Computational Research. Auto-tuning. https://crd.lbl.gov/departments/computer-science/PAR/research/autotuning/.

[69] 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.

[70] 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*, 2016.

[71] 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*, June 29 - July 2, 2015.

[72] 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.

[73] A. A. Shabana. *Dynamics of multibody systems*. John Wiley & Sons, 2nd edition, 1998.

[74] D. Stewart. A platform with Six Degrees of Freedom. In *Institution of Mechanical Engineers UK*, volume 180, pages 371–386, 1965.

[75] W. Tichy. Auto-tuning parallel software: an interview with Thomas Fahringer: the multicore transformation. June 2014(5), 2014.

Anexo A

Diagramas de flujo: resolución de la plataforma de Stewart en el software de control original

La figura A.1 representa el bucle principal del software de control de un sistema multicuerpo, como es la plataforma de Stewart, donde se pueden identificar los bloques que contienen las rutinas que resuelven la cinemática del terminal y, posteriormente, los grupos Manivela-Barra en un bucle.

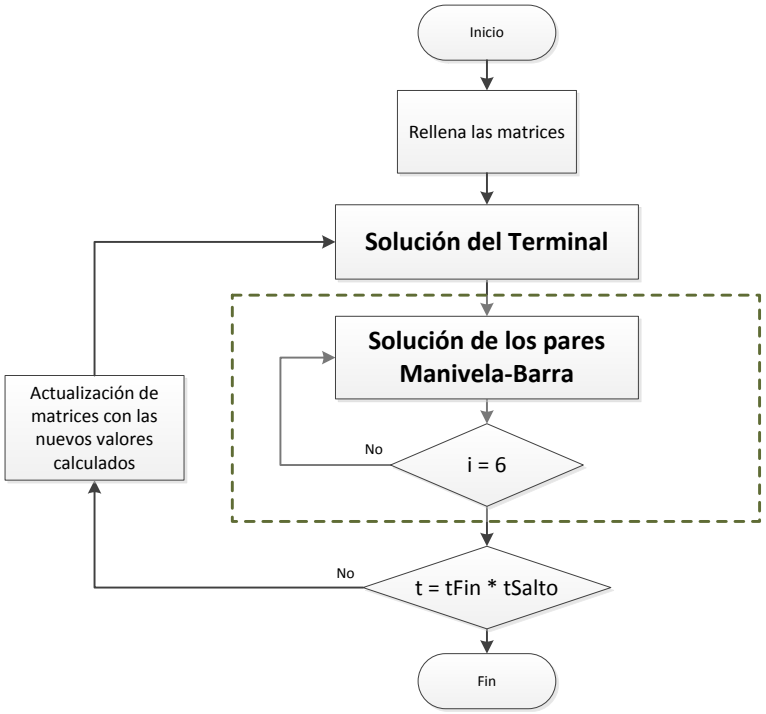


Figura A.1: Diagrama del bucle principal en el software de control original

La figura A.2 muestra la secuencia de cálculos involucrados en la resolución de la cinemática del terminal móvil de la plataforma de Stewart. Se distinguen las funciones de MKL que realizan la descomposición LU de las matrices, DGETRF, y la resolución de los sistemas de ecuaciones, DGETRS.

Se puede observar que el problema del cálculo de la posición supone la resolución de un sistema de ecuaciones por un método iterativo (Newton-Raphson en este caso) controlado por el valor de tolErr, que define el error aceptado en el proceso de convergencia.

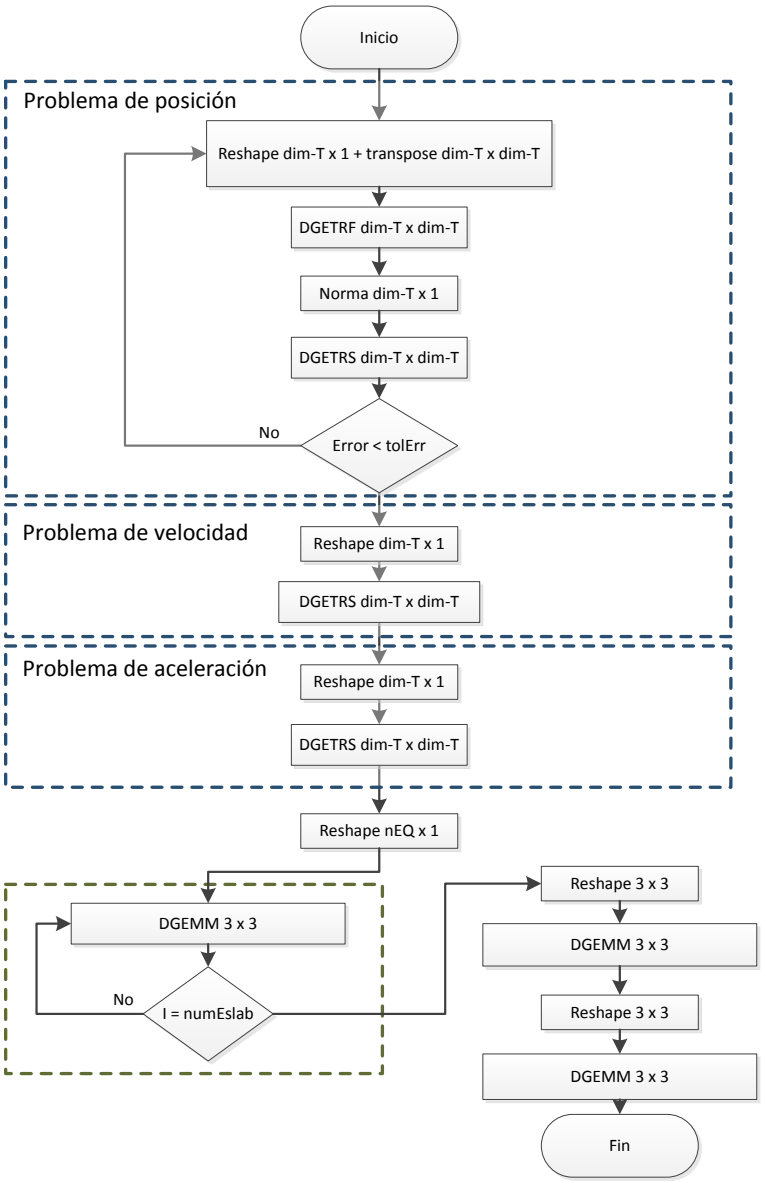


Figura A.2: Diagrama de resolución del grupo Terminal del programa en el software de control original

La figura A.3 muestra la secuencia de cálculos involucrados en la resolución de la cinemática de un grupo compuesto por una manivela, una barra y tres juntas esféricas en la plataforma de Stewart. Se pueden observar las funciones de MKL que realizan la descomposición LU de las matrices, DGETRF, y la resolución de los sistemas de ecuaciones, DGETRS.

Al igual que en el caso del terminal, el problema del cálculo de la posición supone la resolución de un sistema de ecuaciones por el método de Newton-Raphson, donde el valor de tolErr fija el error admitido en la convergencia hacia una solución.

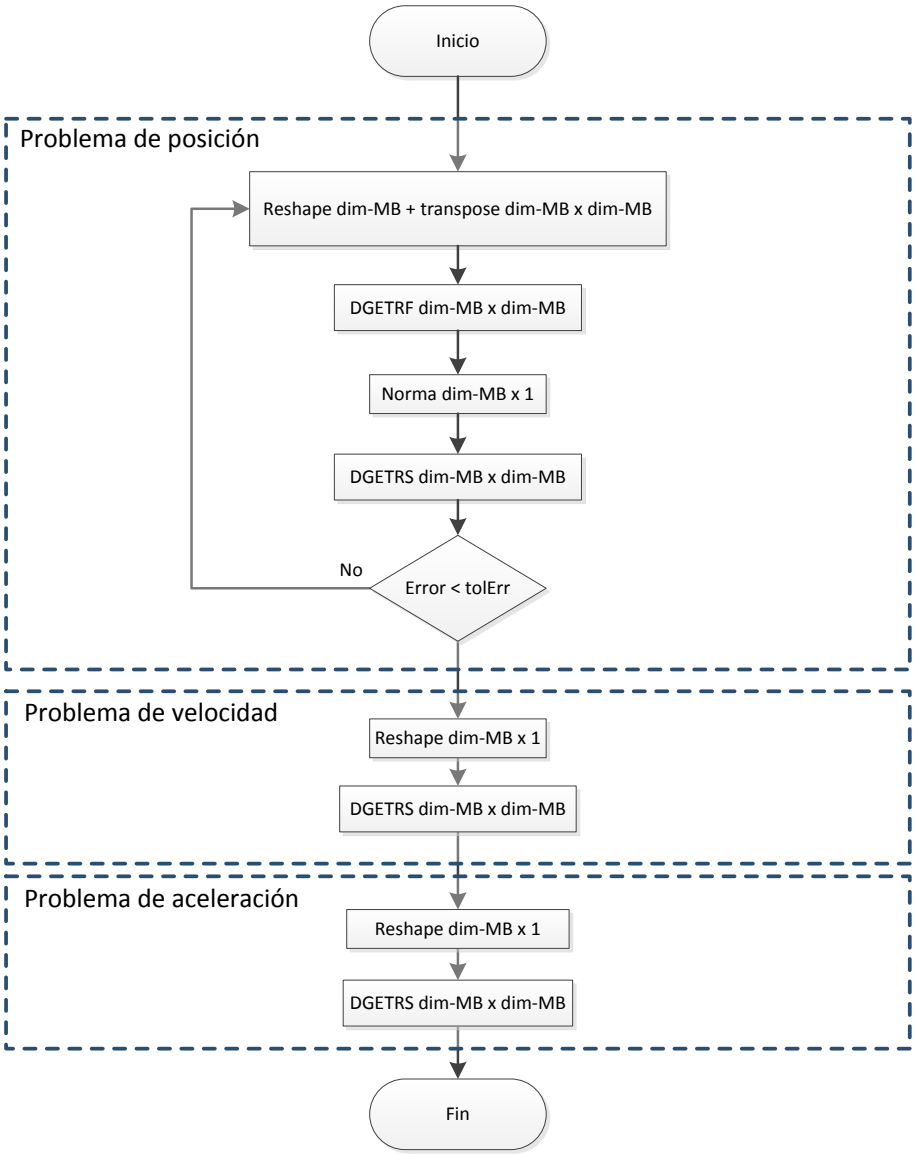


Figura A.3: Diagrama de resolución de los grupos Manivela-Barra en el software de control original

Anexo B

Diagramas de flujo: resolución de la plataforma de Stewart en el simulador

La figura B.1 representa el bucle principal del simulador de un sistema multicuerpo, donde se pueden identificar los bloques que contienen las rutinas que resuelven la cinemática del terminal y, posteriormente, los grupos Manivela-Barra en un bucle. La diferencia con el software de control original radica en el bucle externo adicional que repite las ejecuciones para todos los escenarios a simular, y en la posterior grabación de los tiempos de ejecución obtenidos.

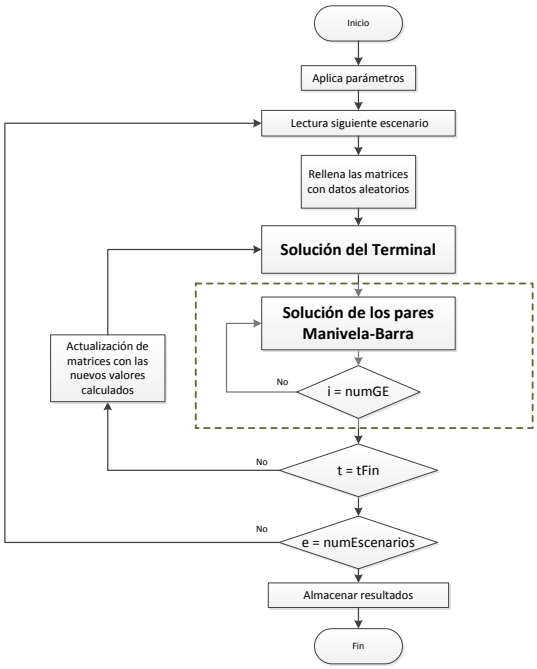


Figura B.1: Diagrama del bucle principal en el simulador

La figura B.2 muestra la secuencia de cálculos involucrados en la resolución de la cinemática del terminal móvil de la plataforma de Stewart cuando se usa la librería MKL. Se distinguen las funciones que realizan la descomposición LU de las matrices, DGETRF, y la resolución de los sistemas de ecuaciones, DGETRS.

Se puede observar que el problema del cálculo de la posición supone la resolución iterativa de un sistema de ecuaciones (el número de veces se especifica en los escenarios).

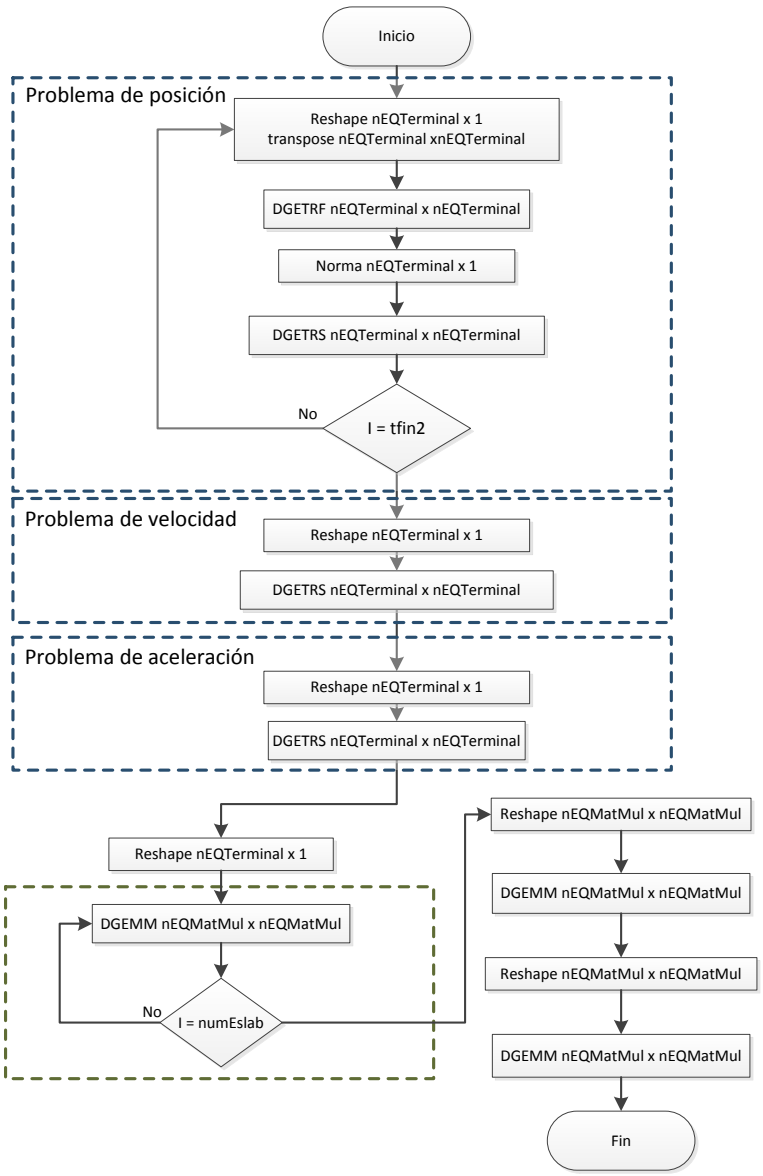


Figura B.2: Subrutina del cálculo cinemático del Terminal en el simulador. Configuración guiada por parámetros y tamaño especificado en escenarios

La figura B.3 muestra la secuencia de cálculos involucrados en la resolución de la cinemática de un grupo compuesto por una manivela, una barra y tres juntas esféricas en el simulador de la plataforma de Stewart. Se pueden observar las funciones de MKL que realizan la descomposición LU de las matrices, DGETRF, y la resolución de los sistemas de ecuaciones, DGETRS.

Al igual que en el caso del terminal, el problema del cálculo de la posición supone la resolución de un sistema de ecuaciones tantas veces como se especifique en los escenarios.

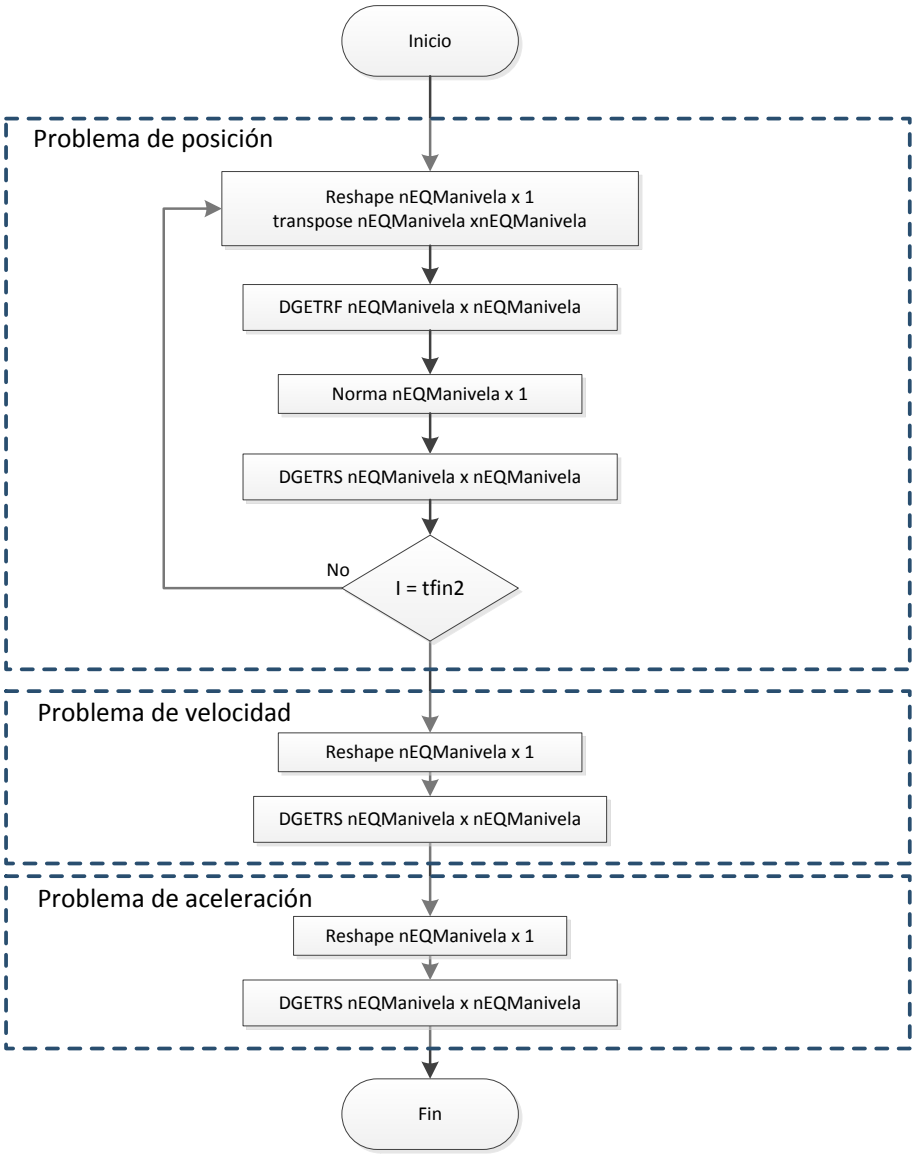


Figura B.3: Subrutina del cálculo cinemático de un grupo Manivela-Barra en el simulador. Configuración guiada por parámetros y tamaño especificado en escenarios

Anexo C

Ejemplo de ejecución en JUPITER

En este anexo se muestra la salida estándar que realiza el simulador de la plataforma de Stewart durante una ejecución. Indica el número de iteraciones y la librería matricial empleada, el tamaño del problema (representado por las dimensiones de las matrices y el número de grupos estructurales), los parámetros de paralelismo relevantes y el tipo de dispersión de las matrices.

Al terminar la simulación de cada escenario se muestra el tiempo efectivo empleado en los cálculos.

```
Tiempo transcurrido carga de datos......    0.0551
================================================
Ejecutando escenario ..................... 1/ 17
Ejecutando iteracion ..................... 1/  1
Usando libreria ......................... 1
Iteraciones bucle principal ............    10
Iteraciones bucle secundario ...........     3
# de grupos estructurales ..............     8
# ecuaciones matriz de terminal ........    12
# ecuaciones matriz de Maniv/Barra .....    18
# ecuaciones matriz de multiplicacion ..     3
# eslabones ............................     3
# OMP Threads ..........................     1
# MKL Threads ..........................     1
# Nested Level .........................     2
# GPU ..................................     6
Sparsity ..............................    85
Symmetric..............................     1
Library................................     1
Tiempo transcurrido de cálculos ........    0.0087
------------------------------------------------
```

A continuación se muestra el resultado de la función magmaf_print_environment(), ofrecida por la librería de álgebra matricial MAGMA, que muestra las GPUs detectadas en el hardware, y que pueden ser utilizadas por las funciones que combinan los cálculos en CPU y GPUs.

```
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 1.
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
Wed May 10 04:48:14 2017
```

Anexo D

Ejemplo de autotuning: búsqueda de parámetros óptimos

En este anexo se muestra un ejemplo del resultado de la ejecución del simulador de la plataforma de Stewart cuando se proporciona como entrada el argumento único t.

Como se describió en el capítulo 7, este parámetro indica al software que debe leer los ficheros de autotuning presentes en el directorio de la aplicación, que contienen las diferentes combinaciones de parámetros paralelos que se van a probar. Se experimentará con todas las combinaciones sobre los escenarios objeto de estudio, y se ordenarán los resultados para obtener el que consigue realizar los cálculos en el menor tiempo de ejecución.

En este ejemplo se crearon cuatro ficheros de parámetros. El contenido de cada uno de ellos se puede consultar en las tablas 7.5, 7.6, 7.7 y 7.8.

A continuación se muestra la salida por consola que realiza el simulador, que contiene los tiempos de ejecución ordenados de menor a mayor obtenidos con los parámetros indicados en cada uno de los cuatro ficheros de autotuning. Al final del proceso, se indica cuál de ellos ha ofrecido el mejor resultado y con qué combinación de parámetros.

fichero1 #	NumGE	nEQ-T	nEQ-M	nEQ-MM	Eslab	Spars	symet	(s)	omp	mkl	loop	library	gpu
23	8	12	18	3	3	85	1	0.000712	8	1	0	3	6
20	8	12	18	3	3	85	1	0.000741	7	1	0	3	6
26	8	12	18	3	3	85	1	0.000742	9	1	0	3	6
17	8	12	18	3	3	85	1	0.000795	6	1	0	3	6
27	8	12	18	3	3	85	1	0.000852	10	1	0	1	6
21	8	12	18	3	3	85	1	0.000854	8	1	0	1	6
18	8	12	18	3	3	85	1	0.000870	7	1	0	1	6
32	8	12	18	3	3	85	1	0.000915	11	1	0	3	6
29	8	12	18	3	3	85	1	0.000941	10	1	0	3	6
11	8	12	18	3	3	85	1	0.001046	4	1	0	3	6
8	8	12	18	3	3	85	1	0.001078	3	1	0	3	6
30	8	12	18	3	3	85	1	0.001081	11	1	0	1	6
14	8	12	18	3	3	85	1	0.001113	5	1	0	3	6
15	8	12	18	3	3	85	1	0.001374	6	1	0	1	6
9	8	12	18	3	3	85	1	0.001395	4	1	0	1	6
12	8	12	18	3	3	85	1	0.001424	5	1	0	1	6
5	8	12	18	3	3	85	1	0.002215	2	1	0	3	6
6	8	12	18	3	3	85	1	0.002402	3	1	0	1	6
3	8	12	18	3	3	85	1	0.003502	2	1	0	1	6
2	8	12	18	3	3	85	1	0.003770	1	1	0	3	6
24	8	12	18	3	3	85	1	0.004243	9	1	0	1	6
25	8	12	18	3	3	85	1	0.008344	9	1	0	2	6
0	8	12	18	3	3	85	1	0.008965	1	1	0	1	6
19	8	12	18	3	3	85	1	0.009318	7	1	0	2	6
22	8	12	18	3	3	85	1	0.009338	8	1	0	2	6
16	8	12	18	3	3	85	1	0.009463	6	1	0	2	6
28	8	12	18	3	3	85	1	0.011311	10	1	0	2	6
31	8	12	18	3	3	85	1	0.011388	11	1	0	2	6
13	8	12	18	3	3	85	1	0.012182	5	1	0	2	6
10	8	12	18	3	3	85	1	0.012382	4	1	0	2	6
7	8	12	18	3	3	85	1	0.014897	3	1	0	2	6
4	8	12	18	3	3	85	1	0.020990	2	1	0	2	6
1	8	12	18	3	3	85	1	0.032998	1	1	0	2	6

fichero4 #	NumGE	nEQ-T	nEQ-M	nEQ-MM	Eslab	Spars	symet	(s)	omp	mkl	loop	library	gpu
35	8	12	18	3	3	85	1	0.000903	12	1	0	3	6
38	8	12	18	3	3	85	1	0.000943	12	2	0	3	6
36	8	12	18	3	3	85	1	0.001030	12	2	0	1	6
39	8	12	18	3	3	85	1	0.001074	12	3	0	1	6
33	8	12	18	3	3	85	1	0.001080	12	1	0	1	6
41	8	12	18	3	3	85	1	0.010177	12	3	0	3	6
34	8	12	18	3	3	85	1	0.011419	12	1	0	2	6
37	8	12	18	3	3	85	1	0.028440	12	2	0	2	6
40	8	12	18	3	3	85	1	0.035822	12	3	0	2	6

fichero3 #	NumGE	nEQ-T	nEQ-M	nEQ-MM	Eslab	Spars	symet	(s)	omp	mkl	loop	library	gpu
53	8	12	18	3	3	85	1	0.001346	3	6	0	3	6
50	8	12	18	3	3	85	1	0.001365	3	4	0	3	6
51	8	12	18	3	3	85	1	0.001705	3	6	0	1	6
48	8	12	18	3	3	85	1	0.001741	3	4	0	1	6
47	8	12	18	3	3	85	1	0.001795	2	6	0	3	6
44	8	12	18	3	3	85	1	0.001815	2	4	0	3	6
45	8	12	18	3	3	85	1	0.002410	2	6	0	1	6
42	8	12	18	3	3	85	1	0.002427	2	4	0	1	6
49	8	12	18	3	3	85	1	0.020603	3	4	0	2	6
46	8	12	18	3	3	85	1	0.032305	2	6	0	2	6
52	8	12	18	3	3	85	1	0.035501	3	6	0	2	6
43	8	12	18	3	3	85	1	0.062878	2	4	0	2	6

fichero2 #	NumGE	nEQ-T	nEQ-M	nEQ-MM	Eslab	Spars	symet	(s)	omp	mkl	loop	library	gpu
59	8	12	18	3	3	85	1	0.000680	6	2	0	3	6
56	8	12	18	3	3	85	1	0.001122	6	1	0	3	6
54	8	12	18	3	3	85	1	0.001154	6	1	0	1	6
57	8	12	18	3	3	85	1	0.001155	6	2	0	1	6
55	8	12	18	3	3	85	1	0.011302	6	1	0	2	6
58	8	12	18	3	3	85	1	0.016848	6	2	0	2	6

Best scenario is script2.cfg iteration 59 calculated in 0.000680 s