FACULTAD DE INFORMÁTICA

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

TRABAJO FIN DE MÁSTER

TÉCNICAS HEURÍSTICAS PARALELAS EN ACOPLAMIENTO DE COMPUESTOS BIOACTIVOS

Autor:
Baldomero Imbernón Tudela

Dirigido por:
Domingo Giménez Cánovas

MURCIA, 1 de Septiembre de 2015

Resumen

El descubrimiento de fármacos es un proceso largo y costoso que involucra varias etapas; entre ellas destaca la identificación de candidatos a fármacos; es decir moléculas potencialmente activas para neutralizar una determinada proteína involucrada en una enfermedad. Esta etapa se fundamenta en la *optimización* del acoplamiento molecular entre un receptor y un ingente número de candidatos a fármacos, para determinar cuál de estos candidatos obtiene una mayor intensidad en el acoplamiento. El acoplamiento molecular entre dos compuestos bioactivos está sujeto a una serie de fenómenos físicos presentes en la naturaleza. Entre estos fenómenos destaca una fuerza atractiva a grandes distancias denominada *Fuerza de Van Der Waals o dispersión*. El potencial de *Lennard-Jones* es un modelo matemático que representa este comportamiento, permitiendo trasladar esta interacción molecular a una simulación en plataformas computacionales de silicio. Sin embargo, el ingente número de operaciones necesario para desarrollar este tipo de simulación sobre compuestos biológicos reales requiere técnicas algorítmicas y plataformas informáticas innovadoras. En este trabajo fin de máster se propone un esquema metaheurístico parametrizado y paralelo para la optimización del potencial de *Lennard-Jones* que determine la interacción molecular entre compuestos bioactivos reales. Las metaheurísticas son técnicas algorítmicas empleadas, generalmente, en la optimización de cualquier tipo de problema, proporcionando soluciones satisfactorias. Algunos ejemplos de metaheurísticas incluyen *búsquedas locales*; que centran su campo de actuación a su entorno de soluciones (vecinos) más cercanos; *búsquedas basadas en poblaciones* muy utilizadas en la simulación de procesos biológicos y entre los que destacan los *algoritmos evolutivos* o las *búsquedas dispersas* por mencionar algunos ejemplos. Los esquemas parametrizados de metaheurísticas definen una serie de funciones básicas (*Inicializar, Fin, Seleccionar, Combinar, Mejorar e Incluir*) a fin de parametrizar el tipo de metaheurística concreta a instanciar en cada ejecución de la aplicación, permitiendo así no sólo la optimización del problema a resolver, sino también del algoritmo empleado para su resolución. La gran mayoría de algoritmos metaheurísticos son, por definición, masivamente paralelos, y por tanto su implementación en plataformas secuenciales compromete tanto la eficiencia como la eficacia de los mismos. En este trabajo se adapta además la instanciación del esquema metaheurístico a plataformas masivamente paralelas y heterogéneas como procesadores de memoria compartida y tarjetas gráficas. Los resultados de los primeros experimentos secuenciales determinan que existen varias funciones del esquema, como Inicializar y Combinar, cuyo coste se ve incrementado de forma considerable si los valores de los parámetros metaheurísticos son elevados. Por ello, se plantea la opción de paralelizar la ejecución en la medida de lo posible, obteniendo un esquema parametrizado paralelo basado en memoria compartida. Este esquema consta de la misma estructura que el anterior aunque se añaden nuevos parámetros de paralelismo para especificar el número de hilos a ejecutar en cada una de las funciones. Esto nos permite diseñar metaheurísticas paralelas para trabajar con los datos de forma mucho más eficiente. La nueva versión paralela ahorra coste computacional si se compara con los tiempos de la versión secuencial. Esto permite incrementar los valores del ciertos parámetros metaheurísticos para incrementar las posibilidades de encontrar mejores lugares de acoplamiento entre ambos compuestos, obteniendo valores

aún mejores que en la versión secuencial. Las técnicas masivamente paralelas en GPU ayudan a realizar estos cálculos poniendo a disposición de la aplicación miles de núcleos capaces de funcionar en paralelo y, además, con la posibilidad de compartir memoria entre ellos y así reducir aún más los accesos a memoria. Aun así, existen compuestos celulares de decenas de miles de átomos para los que el uso de una sola GPU puede ser insuficiente, convirtiéndola en un cuello de botella. Esto hace necesario extender el esquema a multiGPU para dividir la carga computacional y poder abordar este tipo de compuestos con suficientes garantías de rendimiento. Para mejorar el rendimiento y maximizar la paralelización de la aplicación, es fundamental aprovechar al máximo los recursos que nos ofrece la máquina, por ello, se realiza un trabajo previo para ajustar los parámetros de la opción paralela elegida al entorno de ejecución y trabajar con los parámetros que mejor se adapten a la máquina. Trabajar con extensiones de C como CUDA, nos ofrece otra perspectiva del problema, permitiéndonos diseñar el algoritmo de forma que podamos aprovechar todos los núcleos y multiprocesadores de las GPUs de NVIDIA de la forma más eficiente posible. El esquema metaheurístico paralelo híbrido en multicore CPU y multiGPU obtiene el mejor rendimiento, y es la combinación ideal para el problema de acoplamiento molecular.

Índice general

Índice de figuras

Índice de algoritmos

Índice de tablas

Capítulo 1

Introducción

En este capítulo se va a realizar el planteamiento del problema, la motivación que nos lleva a realizar este trabajo fin de máster y los objetivos que se pretenden lograr. Seguidamente se aportan estudios previos sobre el acoplamiento entre compuestos y cómo la investigación en estos potenciales puede ayudar al descubrimiento de nuevos fármacos, además de describir un conjunto de aplicaciones bioinformáticas que utilizan el potencial que se estudia en este trabajo. Además, se van a explicar conceptos tan importantes en este trabajo como heurística y metaheurística, haciendo hincapié en esta última en sus aplicaciones. A continuación se describirá el entorno de trabajo en el que se va a ejecutar la aplicación, además de la metodología que se va a llevar a cabo para su desarrollo.

1.1. Planteamiento

En estos últimos años, el modelo de computación conocido desde la década de los años 50 y que perdura hasta nuestros días ha ido evolucionando de una manera vertiginosa, cambiando físicamente el modelo de computación establecido hasta ahora, pasando de un modelo secuencial a un modelo paralelo, y en una siguiente evolución a un modelo masivamente paralelo [5].
Estos modelos masivamente paralelos, son muy utilizados en una gran cantidad de aplicaciones científicas que requieren una cantidad ingente de recursos del sistema y, además, un gasto muy importante en procesos de cómputo, por lo que su uso puede aportar un ahorro muy importante en este tipo de procesos.
Con el desarrollo de aplicaciones de programación paralela en diversos campos, se han abierto una serie de caminos, todavía muchos en fase incipiente, relacionados con la optimización de funciones y cálculos biomédicos.
Los procesos de acoplamiento molecular, también conocidos como procesos de *docking* [53], se han beneficiado de una manera muy importante con el desarrollo de tecnologías masivamente paralelas, y su función es predecir si una molécula es candidata para enlazar con otra. Este cálculo, con un coste computacional muy elevado a priori, es usado en el descubrimiento de nuevos fármacos. Para decidir este nivel de acoplamiento, se usa una función denominada función *scoring* [8], que agrupa un conjunto de cálculos y potenciales cuyo valor definirá el nivel de acoplamiento de estas moléculas en una

posición y orientación determinada.
La figura 1 representa el cálculo del potencial de Lennard-Jones [56], además de la evolución de su valor. En la parte superior derecha de la figura se muestra la fórmula del cálculo del potencial, que tiene dos variables en función de los átomos partícipes en el cálculo en cada momento. Por un lado está R, que es la distancia euclídea entre un átomo del receptor con otro del ligando, y por otro σ, que depende del tipo de átomos que intervienen en el cálculo en cada momento. La gráfica registra la evolución del valor del potencial, en función de la relación entre R y σ, y, como se observa, su valor se va a elevar enormemente cuando el radio vaya disminuyendo con respecto al valor σ.

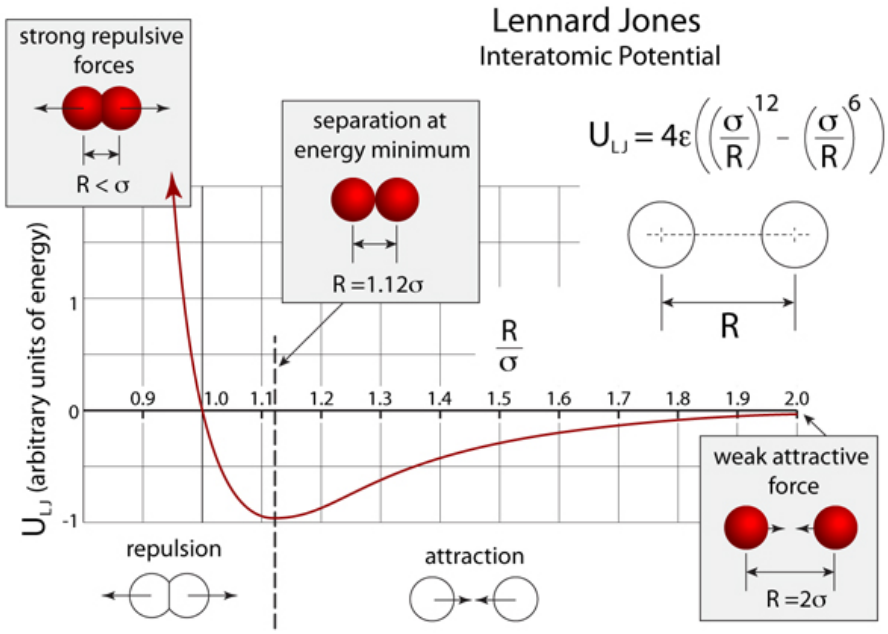

Figura 1.1: Cálculo y representación gráfica del potencial de Lennard-Jones.

El potencial de Lennard-Jones es utilizado para resolver una parte de las Fuerzas de Van der Waals [45]. Este potencial es usado en la mayoría de los procesos de *docking*, además de ser un término fundamental de la función *scoring* a minimizar de la mayoría de este tipo de software.
Minimizar los términos de la función *scoring* lleva importantes costes de cómputo, y desarrollar procesos basados en heurísticas y metaheurísticas [33] que reduzcan estos costes nos puede ayudar a encontrar soluciones mucho más ligeras en cómputo y con una alta precisión a la hora de alinear compuestos.

1.2. Motivación

La evolución de las técnicas de programación y las nuevas características que han ido desarrollando tanto los sistemas multinúcleo como las GPUs (Graphics Processing Unit) [43], han llevado a demostrar que con su uso se produce un incremento muy importante del rendimiento de las aplicaciones o, como define en prensa esta evolución de

la GPU Jack Dongarra, Profesor del Innovative Computing Laboratory de la Universidad de Tennesse, *Las GPUs han evolucionado hasta un punto en que muchas de las aplicaciones industriales actuales se ejecutan en ellas con niveles de rendimiento muy superiores a los que ofrecerían si se ejecutasen en sistemas multinúcleo. Las arquitecturas informáticas del futuro serán sistemas híbridos con GPUs compuestas por núcleos de procesamiento paralelo que trabajarán en colaboración con las CPUs multinúcleo* [1].
El incremento del rendimiento que ofrecen los algoritmos al incorporar la GPU y sistemas multinúcleo en su ejecución, lleva a plantear la opción de aplicar este modelo masivamente paralelo de forma generalizada en procesos de optimización.
El usar gran cantidad de núcleos en paralelo para mejorar nuestras aplicaciones permite dar un salto considerable a la hora de valorar y diseñar algoritmos, dado que el poder aprovecharnos de este enorme potencial de rendimiento para nuestras operaciones de cálculo, primará en gran medida para adaptar nuestro algoritmo a estas necesidades computacionales.
La alta precisión que nos dan los procesos heurísticos y metaheurísticos, combinados con el uso del modelo masivamente paralelo, nos va a permitir obtener soluciones confiables en tiempos razonables.

1.3. Objetivos

En este trabajo fin de máster, se ha considerado como objetivo principal trabajar en la obtención de la interacción óptima entre moléculas trabajando en dos sentidos, por un lado, la optimización máxima de todo el proceso de cálculo para obtener el resultado final, en sistemas multicore, GPU y multiGPU y, por otro lado, aplicando heurísticas y metaheurísticas a la "Selección de movimientos" del proceso de *docking*.
Se ha considerado, además del objetivo principal, varios más concretos que nos van a propiciar evaluar el rendimiento y las posibilidades del algoritmo en diferentes entornos de ejecución, tanto a nivel secuencial, paralelo con sistemas basados en memoria compartida en la CPU y masivamente paralelo usando multicore+GPU y multicore+multiGPU. El alto coste computacional que tiene el problema va a ser analizado y desmenuzado en los tres entornos anteriores aportando resultados concretos y sucesivas mejoras. Estos objetivos serían:

- Optimizar el cálculo del potencial de Lennard-Jonnes, reduciendo al máximo el tiempo de ejecución para obtener los resultados, maximizando su eficiencia en la medida de lo posible. Vamos a partir de una versión secuencial y aplicaremos sobre ella diferentes técnicas paralelas, estudiando la evolución y mejora del rendimiento en sistemas multicore basados en memoria compartida en la CPU y su ejecución en GPU y multiGPU.

- Estudiar las técnicas metaheurística aplicadas, en función de su tiempo de ejecución y de la eficiencia en el acoplamiento molecular. Se desarrollará un esquema metaheurístico parametrizado secuencial, sobre el que se le irán aplicando técnicas paralelas para mejorar su rendimiento, a través de sistemas multicore basados en memoria compartida en la CPU y su ejecución en GPU y multiGPU.

- Obtener una configuración paralela lo más óptima posible, para conseguir el máximo rendimiento de los recursos asignados para su cómputo.

Estos objetivos pretenden llevar a cabo una meta global, que es minimizar el potencial de Lennard-Jones en la interacción de dos moléculas, generando para ello un código paralelo lo más eficiente y óptimo posible.

1.4. Estado del arte

La programación de métodos científicos, y más concretamente su uso en la aceleración del cálculo de interacciones moleculares, ha despertado un gran interés en los investigadores, gracias al avance en el desarrollo de procesos paralelos y masivamente paralelos. En esta sección se van a describir técnicas de interacción molecular y metaheurísticas, que se van a utilizar en este trabajo fin de máster, además de estudios previos sobre trabajos relacionados con el paralelismo en cálculos energéticos.

1.4.1. Interacción molecular

Actualmente, existe multitud de software informático de predicción de interacción molecular, también conocido como procesos o algoritmos de *docking*, y que usa el cálculo de la energía que se produce en este tipo de interacciones para predecir lugares de acoplamiento o unión entre dos compuestos.
En muchos casos el potencial de Lennard-Jones, usado para resolver las fuerzas de dispersión y repulsión o Interacciones de Van der Waals, está presente en muchos de estos algoritmos, pero su coste, en tiempo de ejecución, es elevado, ya que el valor total de esta interacción se obtiene calculando la suma del potencial individual entre cada dos átomos de los compuestos.

1.4.1.1. Potenciales de interacción molecular

En procesos *docking* existen una serie de cálculos usados dentro de estos algoritmos para medir el acoplamiento de dos compuestos. Ya que estos procesos usan una variedad de potenciales para predecir sus resultados, vamos a nombrar y a describir cuatro de ellos, aunque algunos por sí solos han dado nombre a software especializado.

- ***MEP*** [29]. *Molecular Electrostatic Potential* o Potencial electrostático molecular, cuyos resultados arrojan zonas de mejor acoplamiento que otras entre compuestos sin tener que llegar a la estabilidad total del enlace, sino interaccionando con fuerzas eléctricas. Este cálculo electrostático ha sido optimizado en la GPU con una ganancia en cómputo muy importante [19].

- ***GRID*** [34]. En este caso se construye una red alrededor de la molécula, donde podemos calcular en cada uno de los puntos de la red diversos potenciales, tanto el visto anteriormente, conocido como potencial electrostático, como otros, incluido el de Lennard-Jones.

- ***MLP*** [22]. *Molecular Lipophilicity Potential* o Potencial de lipofilicidad molecular es usado en diversos algoritmos de *docking* para calcular la posibilidad de que un compuesto interactúe con otro en función de la presencia de lípidos en su estructura. Este potencial puede ser combinado con otros para mejorar la precisión en el acoplamiento.

- ***CoMFA*** [10]. *Comparative Molecular Field Analysis*. Utiliza en su cálculo interacciones Lennard-Jones y también potenciales electrostáticos. Se basa en el estudio de los campos eléctricos y electrostáticos de los compuestos, aplicando técnicas estadísticas, como la de mínimos cuadrados parciales para medir estas interacciones.

1.4.1.2. Aplicaciones de interacción molecular

Existen diversas herramientas de *docking* que incorporan en sus algoritmos alguno de los cálculos vistos en el apartado anterior, especialmente el referido al cálculo del potencial de Lennard-Jones. Vamos a ver a continuación cuatro de ellas:

- La herramienta DOCK [18] incorpora en sus cálculos de interacción entre compuestos, además del potencial de Lennard-Jones, la energía electrostática calculada a través del potencial de Coulomb, para modelar el acoplamiento molecular en base a su estructura.

- Otra herramienta muy potente en el campo de la química computacional, desarrollada desde finales de los años 70 es *Assisted Model Building with Energy Refinement*, o AMBER [49], que ha ido incorporando herramientas para simulación de moléculas, dinámica molecular y cálculos genéticos, entre otros. Como otras herramientas, el potencial de Lennard-Jones forma parte de sus cálculos, junto a otros cálculos energéticos para sus predicciones, que van desde modelos electrostáticos a transformadas de Fourier. En sus ultimas versiones, como AMBER 14 [4], tiene soporte para multiGPU.

- AutoDock [26] y AutoDock VINA [58] son los programas de *docking* para predicción de interacción entre moléculas más utilizados hoy en día. Permiten de una forma intuitiva especificar todos los parámetros de los compuestos, como su flexibilidad y torsión, entre otros. Utilizan además modelos de rejilla, como los tipo GRID, con términos de cálculo como Lennard-Jones para obtener energías. Estos productos tienen una serie de extensiones para ampliar tanto el tipo de compuesto como los procesos que se realizan sobre ellos.

- GOLD [60] es otra herramienta generada por investigadores de la Universidad de Cambridge, que realiza procesos de *docking* e incluye una potente herramienta de *Virtual Screnning* o cribado virtual [51]. Para calcular las interacciones moleculares que integra este software, utiliza cálculos energéticos como los descritos en apartados anteriores del tipo electrostático o Lennard-Jones, además de contemplar la flexibilidad en el compuesto receptor, donde un mismo compuesto adopta distintas formas en sucesivos instantes de tiempo.

Todas estas herramientas tienen una gran carga de cómputo en sus ejecuciones, y tanto su evolución como su desarrollo avanzan cada día. Grupos de investigación de todo el mundo en campos orientados a la bioinformática y química computacional, demandan servicios de supercomputación constantemente para ejecutar sus simulaciones en estos paquetes de software. Centros como el BSC [15] *Barcelona Supercomputing Center* proporcionan esta gran capacidad de cómputo aquí en España, aunque a pequeña escala existen otras posibilidades, como el cluster HETEROSOLAR del Grupo de Computación Científica y Programación Paralela de la Universidad de Murcia [14].

1.4.2. Heurísticas, metaheurísticas y sus aplicaciones

Las técnicas heurísticas se pueden aplicar a muchos campos de investigación científicos. En Inteligencia Artificial se utilizan en aspectos que aplican o usan conocimiento no formalizado a la hora de resolver tareas, manteniendo un equilibrio fundamental en relación a tres aspectos: rendimiento, calidad de la solución y optimización de recursos. Los procedimientos heurísticos son aquellos que nos proporcionan una solución con un alto grado de confianza de que el resultado es cercano al óptimo [48]. En nuestro caso, se está implementando un procedimiento heurístico para un problema de optimización matemática, minimizando el potencial de Lennard-Jones, por lo que obtener una solución de este tipo con un coste computacional razonable, aprovechando al máximo los recursos, puede ser ideal para este problema.

1.4.2.1. Metaheurísticas

Las técnicas metaheurísticas suponen un nivel de abstracción mayor al de las heurísticas, pues aplican de forma sistemática estrategias para la aplicación de heurísticas [33]. Podemos considerar cuatro tipos de metaheurísticas fundamentales de acuerdo al procedimiento heurístico que utilizan:

- Metaheurísticas de relajación. Las metaheurísticas aplicadas en este caso, consisten en resolver el problema relajando las necesidades del modelo original, con modificaciones al problema, consiguiendo la solución al problema original de una forma más sencilla. Los métodos de relajación lagrangiana [7] o de restricciones subordinadas [44] son ejemplos de este tipo de metaheurísticas.

- Metaheurísticas constructivas. En este caso, las metaheurísticas consiguen la solución realizando análisis y selección de los procesos que la forman, incorporando elementos de forma iterativa a una estructura de solución que inicialmente está vacía. GRASP [21] es una metaheurística representativa de esta categoría.

- Metaheurísticas de búsqueda. Las metaheurísticas de este tipo, recorren el espacio de todas las soluciones de forma iterativa, buscando mejores soluciones a la original. Existen varios tipos de búsquedas: búsqueda global, local, monótona, entre otras. La búsqueda tabú [24] es un claro ejemplo de metaheurísticas de búsqueda.

- Metaheurísticas evolutivas. En este caso se produce una evolución de las soluciones del conjunto de partida, estableciendo la metaheurística estrategias de evo-

lución interactuando soluciones, obteniendo otras. Los algoritmos genéticos [25] pertenecen a este tipo de metaheurísticas.

Existen otro conjunto de metaheurísticas que no encajan en ninguno de los modelos concretos anteriores, como es el caso de las de descomposición [52, 64]. Las metaheurísticas de descomposición resuelven el problema original dividiéndolo en subproblemas muchos más sencillos de abordar, y así construir la solución del problema original. Esta técnica de descomposición, puede ser muy útil a la hora de abordar problemas de paralelización, donde tenemos que identificar, distribuir y equilibrar las partes susceptibles a aplicar técnicas paralelas.

1.4.2.2. Aplicaciones de metaheurísticas

Las metaheurísticas, como hemos comentado, resuelven problemas de optimización en un tiempo razonable con una solución altamente confiable, por lo que se pueden aplicar en diversos ámbitos y disciplinas científicas a problemas donde obtener la solución óptima supone un coste computacional muy elevado.
Diferentes disciplinas, como la Investigación Operativa, Ingeniería o Inteligencia Artificial, han ido incorporando planteamientos metaheurísticos en la solución a sus problemas. Como ejemplo de ello, el aplicar procesos metaheurísticos, como búsqueda Tabú o Temple Simulado [50], pueden ayudar al aprendizaje de redes neuronales en Inteligencia Artificial.
En el diseño de redes reside un gran problema de optimización [32], basado en el ahorro de costes. En este caso, aplicando técnicas metaheurísticas tales como búsquedas Tabú, Algoritmos Genéticos o incluso la técnica GRASP, entre otras, se pueden obtener soluciones exitosas.
La investigación en procesos bioinformáticos también se ha beneficiado de la aplicación de técnicas metaheurísticas en sus campos, como en el estudio del ADN [35], o integrando varias técnicas metaheurísticas en procesos de *docking* [31], añadiendo nuevas funcionalidades a software ya existente como puede ser en este caso Autodock.
La elección de una metaheurística u otra o usar una combinación de ellas, puede ser una opción a tener en cuenta a la hora de abordar problemas de optimización, cambiando de una a otra si en cierto número de iteraciones no obtenemos una solución de calidad. Por ello, el usar metaheurísticas parametrizadas [2] puede ser una elección muy acertada a la hora de abordar la fase de optimización.
Problemas NP-completos, como es el problema TSP [30] *Travelling Salesman Problem* o del viajante de comercio, cuyo objetivo es encontrar el camino más corto, han sido resueltos con una buena solución y un costo computacional aceptable aplicando la técnica metaheurística ACO [17] o *Ant Colony Optimization*, que se basa en el funcionamiento de una colonia de hormigas. Dada la evolución de las GPU, este algoritmo ACO ha sido implementado para GPU [9] obteniendo la resolución de este problema con un grado de paralelismo elevado, además de una solución de alta calidad.
El campo de estudio de los algoritmos evolutivos ha crecido en interés investigador por la aplicación de algoritmos genéticos en varios campos. El *Knapsack problem* o problema de la mochila ha sido resuelto mediante un algoritmo genético pero, en este caso, usando técnicas masivamente paralelas con multiGPU [27] para lograr el resul-

tado. También se han trabajo modelos como *Cellular Genetic Algorithm* o Algoritmo genético celular [61], donde se divide el trabajo entre diferentes GPUs de tipo *desktop* o de escritorio, accesibles económicamente por la mayoría de usuarios y se consigue obtener grandes beneficios computacionales.
Como hemos visto, las metaheurísticas ofrecen soluciones a problemas de optimización con soluciones confiables, y con un coste computacional que puede verse claramente reducido usando estrategias de paralelismo sobre la metaheurística o metaheurísticas que se van a aplicar para la resolución del potencial de Lennard-Jones.

1.5. Metodología

Dadas las características de optimización del problema que nos ocupa, tenemos que establecer una solución inicial, e ir aplicando diferentes técnicas para ir obteniendo soluciones cada vez mejores. Para ello se definen una serie de fases:

1. Generar un número de movimientos y posiciones aleatorias inicial de una de las moléculas con la otra fija, en posiciones previamente determinadas, en un radio establecido a priori. Cada una de estas nuevas posiciones es denominada conformación.

2. Calcular para cada una de estas conformaciones el potencial de Lennard-Jones y tenerlo como una primera solución inicial.

3. Ir aplicando la metaheurística a las soluciones iniciales, un determinado número de veces para ir refinando este acoplamiento, encontrando soluciones más óptimas en función de los valores del potencial de cada una de las conformaciones.

Este algoritmo que se propone, dividido en los tres pasos anteriores, se debe someter a un proceso para dotarlo de un alto grado de paralelismo. Por ello, cada uno de los puntos se va a someter al ciclo de desarrollo denominado APOD [42]:

- Evaluar *(Assess)*. Identificar las zonas del código que son susceptibles de aplicar técnicas de paralelización.

- Paralelizar *(Parallelize)*. Identificadas las zonas de código a ser paralelizadas, se piensa como hacer este proceso, pudiendo tener principalmente tres caminos para ello. Usar librerías ya optimizadas, es el primero de estos caminos si nuestro código se puede adaptar a ellas. Dejar el peso de la paralelización al compilador es la segunda opción, dado que el desarrollador solo se centraría en el código secuencial, colocando directivas. La tercera opción es desarrollar código paralelo uno mismo controlando todo el proceso, desde la carga en memoria hasta la ejecución de instrucciones.

- Optimizar *(Optimize)*. Al completar la fase de paralelización del código, se nos plantea la necesidad de mejorarlo y obtener el máximo rendimiento posible.

- Implementar *(Deploy)*. Terminada la optimización de una de las partes del algoritmo marcadas como paralelizables, se implementa y se ejecuta, volviendo de nuevo a la fase inicial del ciclo de desarrollo, comparándola con la versión original, viendo las posibles mejoras.

Particularmente, en GPU las técnicas que podemos aplicar se basan en tres aspectos clave: Maximizar la ocupación de los diferentes núcleos, Optimizar el uso de la memoria y Gestionar de la forma más eficiente el flujo de datos.
Después de la finalización del desarrollo de esta solución en este trabajo fin de máster, se procederá a su aplicación a problemas reales de estudio de interacciones moleculares donde todavía no se conocen soluciones óptimas.

1.6. Entornos de trabajo

En esta sección se describe el marco de trabajo que se ha utilizado para la realización y evaluación de las pruebas, tanto las plataformas paralelas NVIDIA [41] de propósito general, incluyendo las versiones de software específicas de las herramientas usadas, como las características de los sistemas *multicore* que se han empleado para llevar a cabo los experimentos. A continuación vamos a describir los ordenadores que incorporan todas estas plataformas:

- **Jupiter.** Multiprocesador con 32 GB de memoria compartida, dispone de dos hexa-cores (12 cores) Intel Xeon E5-2620 a 2.00GHz. Además dispone de seis GPUs, dos GPUs son NVIDIA Fermi Tesla C2075 con 5375 MBytes de Global Memory y 448 cores. Las otras cuatro GPUs se agrupan en dos tarjetas, cada una con dos dispositivos NVIDIA GeForce GTX 590 con 1536 MBytes de Global Memory y 512 CUDA cores. Se encuentra situado en el laboratorio de Computación Científica y Programación Paralela de la Universidad de Murcia, y pertenece al cluster HETEROSOLAR [14].

- **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). Se encuentra situado en el laboratorio de Computación Científica y Programación Paralela de la Universidad de Murcia, y pertenece al cluster HETEROSOLAR. Dispone de una GPU Tesla K20c (basada en la arquitectura Kepler) con 4800 MBytes de memoria global y 2496 núcleos de CUDA.
  En cuanto a software, Saturno dispone de CUDA, en su versión 5.5 [39], por lo que las aplicaciones dispondrán de las características que ofrece este software, incluido paralelismo dinámico, Multiprocesador de streaming y permitir a múltiples núcleos de la CPU usar de forma simultánea los núcleos de la arquitectura CUDA en una misma GPU Kepler.

- **Hertz.** Multiprocesador con 8 GB de RAM, y un QuadCore Xeon E31220 de 3.10GHz alojado en la Universidad Católica de Murcia. Además dispone de una tarjeta Tesla K40c (basada como la anterior en arquitectura Kepler) con 12 GB de memoria global y 2880 núcleos de CUDA.

En sus sistemas, dispone de la versión 6.5 [40] de CUDA, con un aumento en rendimiento de los dispositivos NVIDIA y eficiencia energética, además de preparar al programador de CUDA en el modelo de memoria unificada, que habilita a las aplicaciones acceder a la memoria de la GPU y la CPU sin la necesidad de copiar manualmente los datos entre sí.

Ambas plataformas hardware disponen de OpenMP [12], que nos permite dotar de paralelismo a nuestras aplicaciones con el uso de memoria compartida en la CPU. Su funcionamiento radica en la idea de creación de un determinado conjunto de hilos que comparten las variables del proceso padre que los lanza. En el código de nuestra aplicación colocaremos directivas de compilador, además de poder usar una librería de funciones y un conjunto de variables de entorno que puede regir el funcionamiento de la aplicación.

1.7. Estructura del trabajo fin de máster

En esta sección vamos a describir la estructura que va a tener el trabajo fin de máster, junto con una breve explicación del contenido que alberga cada uno de los capítulos que la componen:

- En el primer capítulo se ha realizado un planteamiento inicial del problema de optimización que supone el cálculo del potencial de Lennard-Jones. Además se describen los campos de aplicación que pueden ser beneficiados al realizar este estudio. En uno de sus apartados, se describen las técnicas software más utilizadas actualmente para medir la interacción entre dos compuestos, y que albergan en su interior el cálculo de este potencial junto a otros.

- En el capítulo 2 se describe la representación de los datos en el sistema, tanto los componentes iniciales, receptor y ligando, como la definición de individuo en nuestro problema de optimización. Se describirán estructuras básicas de almacenamiento del *fitness* y vectores fundamentales con los que se va a trabajar. Abordamos también la descripción de esquemas metaheurísticos basados en búsqueda local y poblacional que serán tratados, además de implementados algunos de ellos, en este trabajo fin de máster.

- En el tercer capítulo se va a estudiar el esquema metaheurístico general y su parametrización en su modalidad secuencial. Se describirá cada uno de los procedimientos que forman parte del esquema, explicando primeramente la tarea que realizan en el esquema general y, posteriormente, los parámetros que van asociados a cada uno de ellos en el esquema parametrizado. A continuación, se procede a explicar la adaptación del esquema metaheurístico parametrizado al acoplamiento entre dos compuestos y, por último, se muestra un primer estudio experimental de funcionamiento del esquema metaheurístico aplicado al problema.

- En el capítulo 4 se introduce la paralelización usando un sistema de memoria compartida en CPU. Se redefinirán cada una de las funciones del esquema secuencial,

adaptándolas a un sistema paralelo para reducir el coste computacional. Dada esta evolución, se van a definir un nuevo conjunto de parámetros de paralelismo para indicar el número de hilos con los que queremos que se ejecuten cada una de las funciones y procedimientos internos del sistema. Seguidamente se va a realizar un estudio experimental para observar el nivel óptimo de hilos para cada una de las funciones y determinar el ahorro conseguido con estas nuevas técnicas. Además, se describirá el proceso de obtención automática del valor óptimo de hilos en la máquina donde vamos a ejecutar nuestra aplicación y, así, adaptar a ella los parámetros paralelos. Por último, se extraerán unas conclusiones del estudio experimental y de las técnicas aplicadas.

- En el quinto capítulo se aplicarán técnicas masivamente paralelas en la GPU. Ciertas partes del código, previamente seleccionadas después de estudiar su rendimiento en modo secuencial y paralelo en la CPU, se ejecutarán a la GPU y se estudiará su evolución y coste temporal. Se expondrán algunas de las técnicas de optimización que se han implementado en el código y, seguidamente, se realizará un estudio experimental de rendimiento para extraer el número de hilos óptimo para la ejecución de este caso concreto en plataformas NVIDIA. Como en el capítulo anterior, se describirá el procedimiento para conseguir de forma automática el número óptimo de hilos por bloque en el dispositivo que se va a utilizar para realizar la ejecución actual. Por último, se extraerán una serie de conclusiones del estudio experimental realizado.

- En el capítulo 6 se va a explicar la técnica basada en multiGPU que se va a aplicar a funciones seleccionadas en capítulos anteriores, que pueden verse afectadas por cuellos de botella. Seguidamente se describirán los algoritmos de las diferentes funciones afectadas, explicando además los tipos posibles de computación en multiGPU tratados. También se abordará el procedimiento para la optimización automática del número de hilos por bloque en cada una de las tarjetas que van a intervenir en el cómputo, y el reparto de la carga computacional entre todas ellas. A continuación se realizará un estudio experimental como se ha realizado en capítulos anteriores. Finalizaremos con las conclusiones derivadas de este estudio.

- En el séptimo y último capítulo se van a enumerar las principales conclusiones y resultados obtenidos durante todo este trabajo, que dan cumplimiento a los objetivos propuestos. Además se expondrán unas líneas futuras de trabajo e investigación hacia donde se puede seguir avanzando.

Capítulo 2

Descripción del problema

En este capítulo se va describir la representación de datos del problema, identificando la forma de representación de un individuo en el sistema, para poder seguir su evolución en sus repetidas transformaciones a las que va a ser sometido. También se van a describir algunas de las técnicas metaheurísticas más conocidas, tanto basadas en búsqueda local como en población. Además especificaremos su aplicación al problema de acoplamiento molecular que se estudia en este trabajo.

2.1. Representación de datos

Las características del problema y el trabajar con acoplamiento molecular, nos hace buscar compuestos reales para realizar y estudiar el acoplamiento entre ellos. El conjunto de datos válido para realizar estos cálculos, no es posible extraerlo de una generación aleatoria de puntos en el espacio.
Para conseguir datos reales, descargamos los compuestos de un banco de datos biológico muy usado a nivel global, denominado Protein Data Bank [46] y que incorporamos a nuestra aplicación desde su formato mol2 [57]. Estos ficheros contienen una cantidad de datos superior a los que nuestros cálculos necesitan, y solo serán almacenados los datos necesarios.
Los dos tipos de compuestos considerados son el denominado ***receptor*** y el ***ligando***. En nuestro caso el receptor va a ser el mismo durante todo el cálculo, variando de posición en el espacio el compuesto ligando para conseguir la mejor posición de acoplamiento posible. Para medir la bondad del acoplamiento, y obtener un valor numérico o *fitness*, usaremos el potencial de Lennard-Jones.
Para almacenar los compuestos receptor y ligando usamos dos estructuras, almacenando en tres vectores con datos en simple precisión las tres coordenadas de cada uno de los átomos. Además en cada una de estructuras existe otro vector que almacena el tipo de cada uno de los átomos como una variable entera. El tipo de átomo puede ser Nitrógeno, Oxígeno, Azufre, Carbono, Enlace de Hidrógeno, Iridio, Cloro, Flúor, Bromo, Litio, Sodio, Potasio, Magnesio, Calcio, Zinc, o Desconocido para identificar si no es ninguno de los anteriores. A cada uno de los tipos se les asigna un número y éste se almacena en el vector.
Ambas estructuras no sufren modificaciones a lo largo de todo el proceso de cálculo,

usando estructuras vectoriales alternativas para almacenar todo el abanico de posiciones distintas del ligando que van a derivarse del proceso metaheurístico.

2.1.1. Definición de individuo o conformación

Previamente al comienzo del cálculo, se deben extraer una serie de puntos candidatos para establecer un acoplamiento, donde en su entorno se van a situar copias del ligando original, variando la posición en el espacio y el ángulo. Estos puntos quedan determinados por los átomos de carbono del compuesto receptor, que almacenaremos en un vector.
Una conformación o individuo vendrá determinado por un desplazamiento en torno a uno de estos puntos candidatos y por una cierta rotación con respecto al ligando original. Su almacenamiento quedará definido con:

- Tres vectores de datos en simple precisión, uno para cada coordenada, para almacenar el desplazamiento de cada conformación con respecto al origen de coordenadas en valores absolutos. El tamaño de cada vector se corresponde con el número de individuos que se va a considerar.

- Un vector de estructuras, cada una compuesta por cuatro componentes donde se almacena cada cuaternión [20]. Esta estructura almacena la rotación de una cierta conformación con respecto a la forma original. El tamaño del vector vendrá determinado por el número de conformaciones, una estructura por cada una de ellas.

2.1.2. Estructura de almacenamiento

Para almacenar el *fitness*, se ha construido una estructura con dos vectores. Uno de ellos para guardar el potencial de Lennard-Jones de cada una de las conformaciones, y otro vector con un número que se le asigna a cada una de las conformaciones.
El motivo de asignar un número a cada conformación es disminuir la carga computacional cuando procedemos a ordenar por mejor valor de *fitness*, ya que solo ordenamos el vector que lo almacena, usando un vector de claves junto a esa ordenación.
Este vector de claves se corresponde con el vector que almacena el número de conformación y que deberemos usar para localizar correctamente la conformación en los vectores de desplazamiento y de rotación, ya que estos no se ordenan.
Tanto los vectores que definen a los individuos o conformaciones, junto a los que forman parte del *fitness* forman una misma estructura para simplificar los argumentos de llamadas a las funciones. Además contendrá las coordenadas exactas de los puntos candidatos, junto con un conjunto de valores necesarios para calcular el *fitness* asociados a cada tipo de átomo, denominados parámetros de Van der Waals.
Además de las mencionadas, existe una estructura de datos denominada *Devices* que se utiliza exclusivamente cuando trabajamos con GPUs. En ella se almacenan datos del dispositivo, junto con su carga de trabajo dentro de la aplicación si se eligen ciertas opciones de computación.

2.2. Esquemas metaheurísticos

Las técnicas basadas en la aproximación y en los métodos estocásticos obtienen buenas soluciones en un tiempo razonable, renunciando a la obtención del resultado óptimo. Las técnicas metaheurísticas forman parte de este grupo y, además, aportan un proceso capaz de escapar de óptimos locales y realizar una búsqueda mucho más completa en todo el espacio de búsqueda.
Una de las características más importantes de una metaheurística es la independencia de la misma del problema a tratar, dado que podemos cambiar o modificar la función o funciones objetivo, y la metaheurística a aplicar no tiene que verse afectada.

2.2.1. Metaheurísticas de búsqueda local

Este tipo de técnicas metaheurísticas usan el concepto de *k-vecinos*, donde k es el número de vecinos que se va a generar desde la posición de una conformación, actualizando progresivamente la solución mediante la exploración por vecindad. La vecindad de una conformación es el número de conformaciones alcanzables desde su actual posición en un cierto rango de desplazamiento. La determinación de la vecindad es muy importante en estas metaheurísticas, pudiendo construirla aplicando un operador de movimiento sobre el individuo original.
La idea de funcionamiento es muy intuitiva, ya que se parte del valor de *fitness* del elemento origen, y si en su exploración del vecindario se encuentra un individuo cuyo *fitness* es mejor, se asume la nueva solución y sigue el proceso. La finalización del mismo puede depender de varios factores: Un límite de saltos entre vecinos, estancamiento de la solución un número determinado de iteraciones, o simplemente llegamos a una solución aceptable entre ciertos rangos establecidos a priori. Tres ejemplos de metaheurísticas usadas actualmente basadas en búsquedas locales serían:

- VNS o *Variable Neighborhood Search* (Búsqueda en Vecindario Variable) [36]. Permite variaciones de la estructura de vecindad, con diferentes grados de libertad. Las estructuras de vecindad se definen al comienzo del proceso algorítmico. La búsqueda parte de la zona más alejada, hasta la más próxima de su vecindad, escogiendo a los individuos que ofrezcan una mejor solución que la obtenida hasta ese instante.

- ILS o *Iterated Local Search* (Búsqueda Local Iterada) [54]. En esta técnica se le aplica un cambio a la solución inicial generando otra nueva, que seguidamente se someterá a un proceso heurístico de mejora del que se obtendrá una nueva solución que reemplazará a la original si reúne todos los requisitos.

- TS o *Tabu Search* (búsqueda Tabú) [24], es un método de búsqueda local muy utilizado y referenciado como técnica metaheurística representativa de métodos basados en búsquedas locales. Es una búsqueda basada en el uso de memoria evitando el volver a recorrer el mismo espacio de búsqueda varias veces y, por tanto, se pueden implementar procedimientos capaces de realizar búsquedas en el espacio de soluciones de forma eficaz y eficiente.
  En la lista tabú se registran soluciones que no deben ser elegidas. Estas soluciones

pueden ser los últimos movimientos realizados en el entorno, soluciones obtenidas de forma reciente, entre otras.

2.2.2. Metaheurísticas basadas en población

Las metaheurísticas basadas en población trabajan con un conjunto de individuos que representan un conjunto de soluciones y no una sola solución como sucede en las metaheurísticas de búsqueda. Existen diversas metaheurísticas basadas en este principio, caso de los algoritmos evolutivos, los que se fundamentan en búsquedas dispersas y también los que se basan en colonias de hormigas. Vamos a comentar las principales características de cada uno de ellos.

- Los algoritmos evolutivos [6] se basan en la evolución de los seres vivos, adaptándose al entorno. En este caso, cada individuo de la población representa una solución al problema. Su funcionamiento parte de una base aleatoria a la hora de generar la población. Para ir evolucionando esta población se trabaja con operaciones de selección, combinación o mutación, generando nuevos individuos. La política de reemplazo de estos algoritmos es fundamental a la hora de poder escapar de mínimos locales y explorar otras vías de evolución al elegir no quedarse con los individuos con los que mayor rendimiento se obtiene.

- Los algoritmos genéticos [28] son un claro ejemplo de algoritmo evolutivo, y su objetivo es la progresiva evolución hacia valores óptimos. Consiste en una rutina matemática a la que entran varios progenitores, y genera otro conjunto de individuos que correspondería a la siguiente generación. El comportamiento de un algoritmo genético puede variar según su implementación. Particularmente en nuestro caso, consiste en seleccionar de la población generada los individuos que tengan mejor valor de *fitness*, rechazando el resto de individuos generados.

- La búsqueda dispersa o *Scatter Search* [23] pertenece al grupo de algoritmos evolutivos basados en poblaciones, pero en este caso la selección de un conjunto de soluciones no se realiza de forma aleatoria, sino que se trabaja con un conjunto de buenas soluciones que permita mejorar la búsqueda manteniendo la diversidad.
  A continuación se generan nuevos subconjuntos, y estos se combinarán entre sí. Como en algoritmos a la clase que pertenece, después de aplicar operaciones de combinación, se aplicarán operaciones de mejora entre individuos para obtener valores de adaptación, o en nuestro caso mejores valores de *fitness*. Una diferencia muy importante con los algoritmos genéticos es la opción de mejorar las combinaciones, que aunque se asemeja más a un método relacionado con las búsquedas locales, se usa como una opción muy recomendable y diferenciadora en este tipo de algoritmos.

- En los algoritmos basados en colonias de hormigas [16] la idea se basa en la forma natural que tienen las hormigas de buscar alimento, dado que salen del hormiguero y de forma aleatoria eligen una dirección de exploración. Cuando una hormiga ha encontrado alimento regresa a la colonia dejando un rastro de feromonas, siendo esto en el mundo de la computación simulado por un grafo.

Este rastro probabilístico otorga a las aristas del grafo que pertenecen al camino una probabilidad superior que a las demás. Este rastro va perdiendo fuerza o probabilidad si no se pasa por él en un determinado intervalo, por lo que es una buena forma de evitar caer en óptimos locales, dando la posibilidad de que se sigan otras rutas.

2.3. Conclusiones

Con lo expuesto en este capítulo podemos extraer las siguientes conclusiones:

- Realizar una representación de datos con unas estructuras eficientes y adaptadas al tipo de dato que se quiere almacenar en ella nos va a propiciar hacer una optimización de recursos desde el propio diseño de la aplicación.
- El uso de puntos de referencia para realizar movimientos y rotaciones permite minimizar el coste de cálculo trabajando solo con un punto para posición y rotación, dejando el trabajo de calcular la posición real de la conformación o individuo solo cuando sea necesaria para calcular su *fitness*.
- Trabajar con metaheurísticas de búsqueda local y basadas en población nos ofrece un conjunto de técnicas para explorar y encontrar en el vecindario puntos de acoplamiento que proporcionen mejores valores de *fitness*, y otro conjunto de ellas para explorar posibles cruces y mutaciones para diversificar las búsquedas evitando caer en mínimos locales.

Capítulo 3

Esquemas metaheurísticos parametrizados

En este capítulo vamos a describir el esquema general parametrizado de una metaheurística, y seguidamente se van a desmenuzar cada uno de los parámetros metaheurísticos que usan estas funciones básicas del esquema. A continuación se analizará cómo aplicar este esquema parametrizado al problema del acoplamiento de moléculas, y por último se realizará un estudio experimental de la versión secuencial.

3.1. Esquema metaheurístico general

Diversos autores plantean esquemas algorítmicos unificados [47, 59] para representar metaheurísticas, y coinciden en plantear un conjunto de funciones similares. Este esquema, definido en el algoritmo 3.1, comprende un total de seis funciones:

- *Inicializar.* Donde se genera el primer conjunto de soluciones.
- *Fin.* Especificamos una condición de parada del algoritmo, que puede ser común a las diferentes metaheurísticas.
- *Seleccionar.* Escogemos un conjunto de referencia para trabajar con él en sucesivas funciones.
- *Combinar.* Dado el conjunto de referencia anterior, se realizan un conjunto de combinaciones de acuerdo con la metaheurística elegida.
- *Mejorar.* Intensificamos sobre algunos de los elementos para intentar mejorar el *fitness* obtenido.
- *Incluir.* En este caso, nos quedamos con un conjunto de valores después de los procedimientos de combinación y mejora. Este conjunto no tiene que ser obligatoriamente los de mejor *fitness*, sino que dependerá de los criterios de inclusión de la metaheurística.

Algoritmo 3.1 Esquema general de metaheurísticas

```
1: Inicializar(S)
2: while no Fin(S) do
3:    Ssel = Seleccionar(S)
4:    Scom = Combinar(Ssel)
5:    Scom = Mejorar(Scom)
6:    S = Incluir(Scom)
7: end while
```

En este esquema, los argumentos S, Ssel y Scom son los encargados de almacenar el conjunto de soluciones del problema, junto con los datos necesarios para su correcta interpretación, como son los tres vectores que representan la posición de la conformación en el espacio y al vector que contiene la información de rotación de cada una de ellas. Además es posible cambiar la implementación de las funciones descritas anteriormente y sin cambiar el esquema adaptarse a otra metaheurística que le queramos aplicar.

3.2. Esquema metaheurístico general parametrizado

Esta versatilidad que permite el esquema visto en el apartado anterior, a la hora de adaptarse a una metaheurística u otra cambiando la implementación de sus funciones, nos permite ir un poco más allá e introducir una serie de parámetros adicionales a las funciones [2].
Estos parámetros adicionales convierten el esquema metaheurístico general en un esquema metaheurístico parametrizado, tal y como aparece en el algoritmo 3.2. La elección de estos parámetros que se le van pasando a cada una de las funciones, nos hace tener muchas posibilidades de encontrar una metaheurística que arroje una serie de resultados favorables y que respondan a los requisitos del problema con el que estamos trabajando.

Algoritmo 3.2 Esquema general metaheurístico parametrizado

```
1: Inicializar(S,ParamIni)
2: while no Fin(S) do
3:    Seleccionar(S,Ssel,ParamSel)
4:    Combinar(Ssel,Scom,ParamCom)
5:    Mejorar(Scom,ParamImp)
6:    Incluir(Scom,S,ParamInc)
7: end while
```

Es importante tener en cuenta el coste computacional que la elección de los parámetros va a suponer para encontrar una posible solución.
A continuación se van a ir describiendo para este caso concreto los parámetros metaheurísticos asociados a cada una de las funciones básicas del esquema:

- ***Inicializar***. La función devuelve un primer conjunto de soluciones fruto de los parámetros metaheurísticos elegidos. Primeramente se genera aleatoriamente una

población de tamaño *NEIIni*, se realiza una mejora de un porcentaje de elementos generados inicialmente indicado por *PEMIni*. Esta mejora se realizará tantas iteraciones como marque el parámetro *IEMIni*. Por último el conjunto solución estará formado por *NEFMIni + NEFPIni* elementos, donde el primer parámetro indica el porcentaje de elementos mejores y el segundo el de peores a seleccionar. Al coger no solo los mejores, diversificamos la búsqueda para intentar evitar mínimos locales.
Por tanto trabajamos con un conjunto de cinco parámetros metaheurísticos *ParamIni = {NEIIni, NEFMIni, NEFPIni, PEMIni, IMEIni}*.

- ***Fin***. El resultado de esta función va a determinar si seguimos ejecutando el esquema metaheurístico. En este caso se han establecido dos parámetros para la función de fin. El primero de ellos se refiere al número de pasadas máximas sin mejora, *NIRFin*. El segundo al número máximo de iteraciones, con o sin mejora, denominado *NMIFin*.
  En la función Fin trabajamos por tanto con un conjunto de dos parámetros metaheurísticos, *ParamFin = {NIRFin, NMIFin}*.
  La gestión de señales a través de su captura, hace poder dotar a este esquema la posibilidad de parar el cómputo cuando se lleven consumidos una cantidad concreta de segundos. La función *signal(señal,funcion_captura)* permite capturar las señales y hacer un tratamiento posterior, en nuestro caso, la señal `SIGALRM`. La función de tratamiento de la señal provocará la finalización de la aplicación.

- ***Seleccionar***. Esta función permite seleccionar un conjunto de referencia para trabajar en las siguientes fases. El porcentaje de selección vendrá marcado por los valores de dos parámetros metaheurísticos. Uno de ellos se denomina *NEMSel*, e identifica la cantidad de elementos mejores que queremos seleccionar. El segundo parámetro recibe el nombre de *NEPSel*, cuyo valor medido en tanto por ciento nos va a indicar el número de elementos peores que queremos que el conjunto de referencia contenga. La suma de ambas cantidades deberá representar como máximo el 100 % de elementos del conjunto de referencia.
  El conjunto de parámetros metaheurísticos en este caso es *ParamSel = {NEMSel, NEPSel}*.

- ***Combinar***. Las opciones de combinación vienen dadas por el valor de *NMMCom*, cuyo valor nos va a indicar qué porcentaje de elementos mejores se van a combinar entre ellos. El parámetro *NMPCom* nos define el porcentaje de elementos mejores que se van a combinar con elementos peores y el último parámetro *NPPCom* indica el porcentaje de elementos peores que queremos que se combinen entre ellos.
  En Combinar se establece un conjunto de tres parámetros metaheurísticos, *ParamCom = {NMMCom, NMPCom, NPPCom}*.

- ***Mejorar***. La función mejorar tiene el mismo objetivo que la mejora en la inicialización, que es lograr un mejor *fitness* en posiciones cercanas diversificando la búsqueda, buscando algún vecino con un mejor acoplamiento. Para gestionar el funcionamiento de este paso, se disponen de dos parámetros metaheurísticos.

El primero de ellos, denominado *PEMImp*, define a qué porcentaje de elementos mejores se les va a aplicar esta intensificación. El segundo parámetro, llamado *IMEImp*, señala cuántas iteraciones se van a realizar para intentar encontrar una mejora de *fitness*.
En la función mejorar, el conjunto de parámetros metaheurísticos lo forman dos elementos, *ParamImp = {PEMImp, IMEImp}*.

- ***Incluir***. En esta última función abordamos la inclusión en el conjunto de referencia. Se dispone de un parámetro metaheurístico denominado *NEMInc*, que indicará el porcentaje de elementos mejores que queremos incluir en el conjunto de referencia. Esta configuración del conjunto de referencia, tiene como fin intentar no caer en mínimos locales que distorsionen el resultado, aumentando la región a explorar.
  El conjunto de parámetros metaheurísticos de esta función lo forma un solo elemento, *ParamInc = {NEMInc}*.

Para finalizar este apartado, comentar que tenemos un total de quince parámetros metaheurísticos repartidos entre las seis funciones básicas, con los que podremos seleccionar y adecuar las metaheurísticas a nuestro problema. De esta forma tendremos mayores posibilidades de encontrar una metaheurística satisfactoria para el problema con el que trabajamos.

3.3. Aplicación del esquema metaheurístico parametrizado al acoplamiento de moléculas

En el acoplamiento de dos moléculas, entre un receptor y un ligando, existen puntos en el receptor donde el ligando puede acoplarse. Estos puntos de candidatos al acoplamiento son guardados en el sistema al principio de la aplicación y la forma de almacenamiento se especifica en el capítulo 2.
En este tipo de problemas se trabaja con la posibilidad de aplicar el esquema metaheurístico parametrizado a cada uno de los puntos candidatos de acoplamiento que dispone el receptor. La cantidad de puntos candidatos es diferente de un compuesto a otro.
Al finalizar el algoritmo, nos quedaremos con el individuo que tenga el *fitness* óptimo de todos los individuos de todos los puntos de acoplamiento.
Además de los parámetros metaheurísticos que se pasan a cada una de las seis funciones básicas y que han sido descritos en el apartado anterior, también forma parte de los argumentos la estructura de almacenamiento de las soluciones descrita en el capítulo 2. Referente a las estructuras del receptor y ligando originales, forman parte de los argumentos de las funciones de inicializar, combinar y mejorar.
Para expresar el orden de ejecución de los algoritmos de las funciones básicas, vamos a definir:

- m al número de puntos candidatos o puntos de acoplamiento del receptor.
- r al número de átomos de la proteína o receptor.
- l al número de átomos del ligando.

- o al número de elementos mejores en la función Combinar.
- p al número de elementos peores en la función Combinar.
- q al número de elementos mejores-peores en la función Combinar.
- n al número total de conformaciones o individuos.
- u al número de elementos seleccionados para la fase de mejora en la función de Inicializar.
- v al número de elementos seleccionados para la fase de Mejorar.
- w al número de elementos combinados.

De los parámetros definidos anteriormente existen unos dependientes del problema, como son m, r y l. El resto, denominados parámetros algorítmicos, se definen en función de la metaheurística y van a tener valores diferentes dependiendo de la función en la que son utilizados. El caso de n su valor cambia en la función de Inicializar tomando el valor de $m * NEIIni$. En la función combinar es el total de elementos combinados $w * m$, y en la función Mejorar toma los valores de u y v según *PEMIni* y *PEMImp* respectivamente. Los valores o, p y q dependen de los valores de los parámetros metaheurísticos *NMMCom*, *NPPCom* y *NMPCom*.
Cuando se computa un individuo en un procesador existen una serie de costes asociados a este computo. Se va a definir una constante que denominaremos K_{CPU} que aglutina todos ellos. Esta constante se irá multiplicando por los diferentes elementos que se computan en cada una de las funciones, y así, calcular su coste total. El rendimiento será mejor cuanto más que pequeño sea su valor.
A continuación vamos a describir en pseudocódigo el trabajo que realiza cada una de las funciones básicas del esquema metaheurístico parametrizado aplicado a la interacción molecular.

Inicializar(S,ParamIni)

Como hemos comentado antes, genera un primer conjunto de soluciones, y su esquema se muestra en el algoritmo 3.3.

Algoritmo 3.3 Pseudocódigo función *Inicializar(S,ParamIni)*

```
1: Generar_posiciones_iniciales_aleatorias(S,ParamIni)
2: Calcular_finess(Proteina,Ligando,S)
3: if PEMIni >0 then
4:    Mejorar(S,ParamIni)
5: end if
6: Ordenar_fitness(S)
```

La función denominada Generar_posiciones_iniciales_aleatorias genera una primera distribución de conformaciones en torno a los puntos candidatos en un cierto radio

pasado como parámetro, y su esquema de trabajo queda reflejado en algoritmo 3.4. La función Desplazar_y_rotar_aleatorio(punto_candidato,angulo,radio) va generando cada una de las posiciones en función del punto candidato introducido y un cierto ángulo y desplazamiento con respecto al punto candidato generado de forma aleatoria. El coste del algoritmo es $O(K_{CPU} * n)$.

Algoritmo 3.4 Pseudocódigo función *Generar_posiciones_iniciales_aleatorias (S,ParamIni)*

```
1: for i = 1 to n do
2:    Desplazar_y_rotar_aleatorio(i,ángulo,radio)
3: end for
```

La función Calcular_fitness nos calcula el potencial de Lennard-Jones para cada uno de los individuos o conformaciones generadas con anterioridad, y almacena su valor en el conjunto de soluciones. En el algoritmo 3.5 podemos ver su esquema de cómputo, similar al explicado en el capítulo 1. El coste del algoritmo es $O(K_{CPU} * n * r * l)$.

Algoritmo 3.5 Pseudocódigo función *Calcular_fitness(Proteina,Ligando,S)*

```
1: for k = 1 to n do
2:    for i = 1 to r do
3:       for j = 1 to l do
4:          vdwEnergy = 4 * epsilon * (term12(i, j) − term6(i, j))
5:          vdwTerm+ = vdwEnergy
6:       end for
7:    end for
8:    S_energy[k] = vdwTerm
9:    vdw_term = 0
10: end for
```

Con respecto a la función Mejorar, nos permite realizar iteraciones variando las posiciones originales de los individuos para ver si en su entorno existe alguna que nos proporcione un mejor *fitness*. En el pseudocódigo del algoritmo 3.6, que describe su comportamiento, podemos observar que cada vez que generamos una nueva posición calculamos el *fitness*, y la inclusión del nuevo individuo en el conjunto de referencia dependerá de si este valor mejora el anterior. El coste computacional del algoritmo es $O(K_{CPU} * IMEIni * (u + u * r * l))$. Claramente, el término $u * r * l$, que representa cálculo de *fitness*, llevá el mayor peso.
En el algoritmo 3.6, la variable *Stmp* almacena temporalmente las nuevas posiciones de las conformaciones generadas en su mejora, antes de incluirlas o no en el conjunto de referencia. Para finalizar la fase de inicialización, ordenamos los valores de *fitness* generados alrededor de cada uno de los puntos candidatos.

Seleccionar(S,Ssel,ParamSel)

La variable *Ssel* representa el conjunto de elementos que estamos seleccionando y que pasarán a la siguiente fase del esquema metaheurístico parametrizado.

Algoritmo 3.6 Pseudocódigo función *Mejorar(S,ParamIni)*

```
1: Copia(Stmp,S)
2: K = 0
3: while K < IMEIni do
4:    for i = 1 to u do
5:       Desplazar_ y_rotar_aleatorio(elemento_Stmp,ángulo,desplazamiento)
6:    end for
7:    Calcular_finess(Proteina,Ligando,Stmp)
8:    Incluir_si_mejora(S,Stmp)
9:    K = K + 1
10: end while
```

Según los parámetros metaheurísticos *NEMSel* y *NEPSel* se selecciona un conjunto de referencia de cada uno de los puntos candidatos del receptor. En el algoritmo 3.7 se define el esquema de esta función. El coste del algoritmo es $O(K_{CPU} * m)$.

Algoritmo 3.7 Pseudocódigo función *Seleccionar(S,Ssel,ParamSel)*

```
1: for i = 1 to m do
2:    Extraer_conjunto_referencia(S,Ssel,ParamSel)
3: end for
```

Combinar(Ssel,Scom,ParamCom)

Las variables *Ssel* y *Scom* representan respectivamente los elementos seleccionados en la etapa anterior y los elementos combinados en esta. Para comenzar, obtenemos el número de elementos a combinar según los porcentajes de los parámetros metaheurísticos *NMMCom*, *NMPCom* y *NPPCom*. Las combinaciones se realizan entre dos elementos, y su proceso está representado en el algortimo 3.8. De cada una de estas combinaciones se generan dos elementos, cada uno con un ángulo aleatorio diferente, pero situados en el mismo lugar en el espacio.

Algoritmo 3.8 Pseudocódigo función *Combinar_dos_elementos (Ssel,Scom,elemento_1,elemento_2,ParamCom)*

1: Calcular el punto medio entre ambos elementos. Esa será la posición del nuevo elemento.
2: Obtener un ángulo aleatorio entre ± A, siendo A un valor pasado como parámetro, a partir del ángulo del primer progenitor.
3: Guardar en el vector de combinaciones la posición y el ángulo.

En el algoritmo 3.9 se describe el proceso general de combinación donde, según los porcentajes indicados en los parámetros metaheurísticos, se producen las combinaciones entre las conformaciones. El algoritmo toma como entrada exclusivamente los elementos seleccionados en la etapa anterior y devuelve las posiciones ya combinadas. El coste del algoritmo es $O(K_{CPU} * (m * o + m * p + m * q + n * r * l))$. Los tres primeros términos

de la suma corresponden a los costes derivados de la combinación entre los elementos mejores, elementos peores y, entre mejores y peores seleccionados para la combinación. El término de mayor coste es el último sumando, que corresponde al cálculo del *fitness*.

Algoritmo 3.9 Pseudocódigo función *Combinar(Ssel,Scom,ParamCom)*

```
1: for i = 1 to m do
2:    for j = 1 to o do
3:       Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
4:    end for
5:    for k = 1 to p do
6:       Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
7:    end for
8:    for l = 1 to q do
9:       Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
10:   end for
11: end for
12: Calcular_finess(Proteina,Ligando,Scom)
```

El conjunto de conformaciones que han sido generadas para cada punto candidato se ordenan de menor a mayor, debiendo colocar los índices en función de esta ordenación. Para terminar esta fase, se calcula el *fitness* de todos los elementos combinados.

Mejorar(Scom,ParamImp)

Esta función es similar a la descrita en la función de inicialización, mejorando un porcentaje de individuos de cada uno de los puntos candidatos un número k de ocasiones. Estos valores los dictan dos parámetros metaheurísticos *PEMImp* y *IMEImp*, diferentes a los que son pasados en la fase de inicialización, aunque con significado similar. La variable *Scom* contiene los elementos combinados en la fase anterior. Su coste computacional es $O(K_{CPU} * IMEImp * (v + v * r * l))$

Incluir(S,Scom,ParamInc)

En esta función calculamos los elementos que debemos incluir de cada uno de los puntos candidatos, en función del valor de los dos parámetros metaheurísticos que influyen en este cálculo, *NEMInc* y *NEPInc*. En el algoritmo 3.10 se expresan estos pasos, debiendo realizar esta inclusión de elementos mejores y peores para cada uno de los puntos candidatos al acoplamiento. La variable *Scom* almacena las conformaciones procedentes de la fase de combinación. El coste del algoritmo es $O(K_{CPU} * m)$.
En la tabla 3.1 podemos observar que los principales costes teóricos se localizan en las funciones que integran el algoritmo 3.5, que representa el coste del cálculo del *fitness*. Este cálculo está integrado en las funciones de Inicializar, Combinar y Mejorar. Los algoritmos 3.4, 3.5 y 3.6 corresponden a diferentes etapas de la fase de Inicializar, repitiendo el algoritmo 3.6 en la fase de Mejorar. El algoritmo 3.7 se identifica con la fase de Seleccionar y, los algoritmos 3.9 y 3.10 corresponden a las fases de Combinar e

Algoritmo 3.10 Pseudocódigo función *Incluir(S,Scom,ParamInc)*

```
1: Obtener número de elementos mejores y peores a incluir, según porcentajes de NEMInc y NEPInc, de Scom.
2: for i = 1 to m do
3:    Copiar_mejores(S,Scom,ParamInc)
4:    Copiar_peores(S,Scom,ParamInc)
5: end for
```

Funciones		Algoritmos	Modelo Secuencial
Inicializar	Generar_Posiciones_Iniciales	Algoritmo 3.4	$O(K_{CPU} * n)$
	Cálculo del Fitness	Algoritmo 3.5	$O(K_{CPU} * n * r * l)$
	Mejorar	Algoritmo 3.6	$O(K_{CPU} * IMEIni * (u + u * r * l))$
Seleccionar		Algoritmo 3.7	$O(K_{CPU} * m)$
Combinar		Algoritmo 3.9	$O(K_{CPU} * (m * o + m * p + m * q + w * r * l))$
Mejorar		Algoritmo 3.6	$O(K_{CPU} * IMEImp * (v * (v * r * l))$
Incluir		Algoritmo 3.10	$O(K_{CPU} * m)$

Tabla 3.1: Resumen de lo tiempos teóricos de ejecución de los algoritmos considerados en el modelo secuencial.

Incluir respectivamente.

3.4. Resultados computacionales secuenciales

En esta sección vamos a probar diferentes combinaciones de parámetros metaheurísticos para ver cómo se comportan ante el problema de miminización que nos ocupa. Se evaluará tanto el *fitness* obtenido como el tiempo de ejecución.

3.4.1. Análisis del *fitness*

Se han elegido un total de ocho combinaciones para evaluar, por un lado, la combinación paramétrica para este problema concreto que obtenga mejor *fitness*, y por otro lado estudiar los tiempos de ejecución de cada una de las funciones para determinar dónde se concentran los mayores costes. Con estos datos se propondrán en capítulos posteriores estrategias de paralelización para disminuir este coste. En la tabla 3.2 se muestra por columnas el valor de cada uno de los parámetros metaheurísticos en cada una de las ocho combinaciones consideradas.
Las combinaciones se han configurado de forma que no incluyan ninguna mejora en alguna de ellas, simulando a un Algoritmo Genético como la combinación 1 o variando los porcentajes de los parámetros metaheurísticos de mejora o centrándonos en trabajar solo con un conjunto pequeño de conformaciones cumpliendo alguna de las características, que determina un *scatter search* o búsqueda dispersa, caso de las combinaciones 2 y 5. El resto de ellas son variantes de las anteriores sin ningún patrón determinado; simplemente se busca la diversidad de los elementos para mejorar el *fitness*.
Se han elegido dos compuestos determinados para estas pruebas, el receptor se denomina *2bsm_rec.mol2* y el ligando *2bsm_lig.mol2*, ambos extraídos de una base de

	COMBINACIONES							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
NEIIni	64	64	64	64	128	128	128	128
PEMIni	0	0	20	20	0	0	20	20
IMEIni	0	0	10	10	0	0	10	10
NEFMIni	100	50	100	50	100	50	100	50
NEFPIni	0	50	0	50	0	50	0	50
NEMSel	100	50	50	100	100	50	50	100
NEPSel	0	50	50	0	0	50	50	0
NMMCom	50	25	50	100	50	25	50	100
NPPCom	0	25	25	0	0	25	25	0
NMPCom	0	50	25	0	0	50	25	0
PEMImp	0	25	25	0	0	25	25	0
IMEImp	0	10	10	0	0	10	10	0
NEMInc	100	80	80	100	100	80	80	100
NIRFin	3	3	3	3	3	3	3	3
NMIFin	10	10	10	10	10	10	10	10

Tabla 3.2: Valores de los parámetros metaheurísticos asociados a las combinaciones consideradas.

datos de compuestos conocida [46] situados en la posición óptima de acoplamiento. De esta forma conocemos el valor óptimo del potencial de Lennard-Jones entre estos dos compuestos.
En la tabla 3.3 se muestran los resultados del *fitness* y tiempo de ejecución para cada una de las combinaciones. Estas simulaciones se han ejecutado en el ordenador Hertz. Dado que su aleatoriedad dicta que en cada ejecución este valor va a variar, se ha realizado la ejecución de las ocho combinaciones un total de diez veces y se ha obtenido su media, que es el valor que figura en la tabla. La diferencia entre el resultado de cada uno de los experimentos con respecto a la media es de $\pm$ 30 unidades.

	COMBINACIONES							
	1	2	3	4	5	6	7	8
Fitness	-105.55	-101.41	-102.98	-108.80	-117.01	-106.16	-106.81	-123.47
Tiempo (seg)	4315.1	2932.7	1870.6	15334.4	8775.4	6074.8	8220.7	35319.5

Tabla 3.3: Valores de *fitness* y tiempo de ejecución para cada una de las combinaciones de la tabla 3.2. en el ordenador Hertz.

Con los resultados que se muestra, se determina que la combinación 8 obtiene los mejores valores de *fitness*. Con estos datos, además podemos extraer varias conclusiones:

- Se deben redefinir las combinaciones, modificando los parámetros metaheurísticos para intentar mejorar el *fitness*, dado que el valor óptimo de *fitness* para el acoplamiento de estos dos compuestos es -277.35 y el mejor obtenido es -123.47.

Este acoplamiento óptimo se extrae de computar el receptor y ligando en la mejor posición de ambos ya conocida [46]. De esta manera podemos evaluar el resultado obtenido.

- Dada la aleatoriedad de los resultados, debemos aumentar el número de los parámetros metaheurísticos NIRFin y NMIFin para luego volver a realizar la media de ellos y observar si mejora el resultado.

- El alto coste computacional, hace que debamos buscar técnicas paralelas con las que podamos aumentar ciertos parámetros metaheurísticos sin que suponga una carga computacional imposible de asumir en modo secuencial.

3.4.2. Análisis del coste computacional

Como se observa en la tabla 3.3, el tiempo de ejecución es muy alto cuando combinamos una gran cantidad de elementos e intentamos mejorar en las fases de Inicializar y Mejorar como sucede en la combinación 8.
Vamos a estudiar el coste individual de cada una de las funciones del esquema metaheurístico parametrizado que se ha descrito en secciones anteriores. Los costes se muestran en la tabla 3.4.

	COMBINACIONES							
	1	2	3	4	5	6	7	8
Inicializar	34.524	34.648	99.353	99.248	69.110	69.416	204.141	204.088
Seleccionar	0.0008	0.00085	0.0004	0.0007	0.0009	0.0009	0.0011	0.0008
Combinar	4280.453	2897.069	1446.889	15234.969	8706.195	6003.229	7204.227	35114.76
Mejorar	0	1.00861	324.402	0	0	2.084	812.326	0
Incluir	0.059	0.035	0.018	0.218	0.128	0.086	0.103	0.730

Tabla 3.4: Tiempo de ejecución en segundos de cada una de las funciones, para las distintas combinaciones consideradas.

Se observa claramente que los principales costes se sitúan en las función de Combinar, aunque vamos a profundizar un poco más en todas ellas y ver entre qué procedimientos internos se distribuye este coste. Para ello vamos a ejecutar solo una pasada de estas funciones y medir los tiempos interiores de los diferentes procedimientos que los forman. En la tabla 3.5 se muestra esta distribución temporal para cuatro combinaciones representativas de la tabla 3.2, que incluyen la variedad de operaciones que vamos a analizar.
La mejora dentro de la función Inicializar incluye el cálculo del potencial de Lennard-Jones. La función Mejorar es similar a la que se ejecuta en la fase de inicialización, aunque llamada desde otro punto del proceso y con diferente número de elementos. Para predecir su comportamiento y evolución, medirla en la fase de inicialización es suficiente.

		COMBINACIONES			
		1	4	5	8
Inicializar	*Generar elementos iniciales*	0.0034	0.0034	0.0068	0.0067
	Cálculo Fitness	34.5821	34.5565	69.1203	69.4246
	Mejora Inicializar	0	64.8099	0	135.0212
	Funciones Auxiliares	0.0006	0.00062	0.0011	0.0009
	Total	34.5861	99.3704	69.1282	204.4534
Combinar	*Generar elementos combinados*	0.0347	0.1404	0.1402	0.5630
	Cálculo Fitness	535.7670	2187.8270	2177.0998	8776.6210
	Total	535.8017	2187.9674	2177.2400	8777.1841

Tabla 3.5: Tiempo de ejecución en segundos de los procedimientos internos de las funciones Inicializar y Combinar.

3.5. Conclusiones

Después de analizar los resultados podemos extraer varias conclusiones:

- La primera y más evidente es que el coste más importante de cualquiera de las fases del proceso, sea en la inicialización, combinación o mejora, es el cálculo del *fitness*, que supone más del 98 % del coste total de cada fase.

- Otra conclusión es la evolución y la predicción de su coste computacional en el futuro, dado que el cálculo del *fitness* va en función de cuatro parámetros: puntos candidatos del receptor, número de conformaciones por cada uno de esos puntos, número de átomos del receptor y número de átomos del ligando.
  Para comprobar esta última deducción, vamos a verificarlo en las dos fases. En la inicialización, si observamos las combinaciones 1 y 5, se inicializan el doble de elementos en la combinación 5 que en la combinación 1. Observamos que el tiempo de la combinación 5 es el doble que la combinación 1, por lo que en principio se confirma esta deducción. Para verificarlo definitivamente, vamos a observar la fase de combinar, entre por ejemplo la combinación 1 y la combinación 4. En este caso, tenemos el mismo número de elementos iniciales y seleccionados, pero en la combinación 4 se escogen la totalidad de elementos para combinar y por tanto multiplicamos por 4 el número de elementos combinados con respecto a la combinación 1, de 992 a 4032. Como podemos comprobar en la tabla 3.5, el tiempo de la combinación 4 en esa fase es 4 veces el de la combinación 1, por lo que se ha comprobado en dos fases la evolución lineal del cálculo del *fitness*.

Con todos estos datos, es de vital importancia trabajar y mejorar en la medida de lo posible el cálculo del *fitness* con técnicas paralelas, dado que con estos datos cualquier ahorro de tiempo en su obtención mejorará de forma muy importante el rendimiento de nuestra aplicación.

Capítulo 4

Esquemas paralelos parametrizados en sistemas basados en memoria compartida en CPU

En este capítulo vamos a abordar el tema del paralelismo aplicado a nuestro esquema metaheurístico parametrizado. Para ello se adaptarán los esquemas paramétricos secuenciales a otros paralelos de memoria compartida en la CPU para mejorar el rendimiento de nuestra aplicación. Más concretamente, este cambio se aplicará a todas las funciones, haciendo un especial hincapié en las funciones donde la carga computacional es mayor, y que han sido detectadas en el capítulo anterior. Seguidamente se realizará un estudio de rendimiento para ver la evolución del coste computacional con respecto al esquema secuencial. Antes de finalizar con las conclusiones derivadas del estudio, se describirá el proceso de auto-optimización que se ha incorporado a la aplicación, para encontrar de forma automática el número óptimo de hilos en la maquina donde se va a ejecutar.

4.1. Esquema parametrizado paralelo en memoria compartida en la CPU

Partiendo de la base del algoritmo 3.2, vamos a convertirlo en un esquema parametrizado paralelo en memoria compartida en la CPU, donde de forma independiente se va añadir paralelismo a cada una de las funciones que lo forman. Se indicará a través de nuevos parámetros de paralelismo el número de hilos a usar en cada una de ellas. Este nuevo esquema queda definido en el algoritmo 4.1.
Esta parametrización de los hilos en las funciones, nos va a permitir estudiar el rendimiento variando el número de hilos que le asignamos a cada una de ellas, obteniendo al número óptimo de hilos de una forma experimental.
A la hora de realizar la paralelización de cada una de las funciones, vamos a establecer previamente el número de hilos con los que queremos ejecutar cada una de sus partes.

Algoritmo 4.1 Esquema parametrizado paralelo

```
1: Inicializar(S,ParamIni,Threads1Ini,Threads1Fit,Threads2Fit)
2: while no Fin(S) do
3:    Seleccionar(S, Ssel, ParamSel,Threads1Sel)
4:    Combinar(Ssel, Scom, ParamCom,Threads1Com,Threads2Com,Threads1Fit,
      Threads2Fit)
5:    Mejorar(Scom,ParamImp,Threads1Imp,Threads1Fit,Threads2Fit)
6:    Incluir(Scom,S,ParamInc,Threads1Inc)
7: end while
```

Este esquema de asignación de hilos, en nuestro caso, va tener dos niveles como máximo. El algoritmo de un primer nivel de paralelismo figura en el algoritmo 4.2 y el de dos niveles en el algoritmo 4.3.

Algoritmo 4.2 Paralelismo de un nivel

```
1: omp_set_num_threads (nivel1)
2: #pragma omp parallel for
3: Definición del bucle
4:     Tratamiento de los elementos
```

El esquema del algoritmo 4.2 va a ser usado en funciones como Inicializar, donde va a permitir realizar movimientos y rotaciones o calcular el *fitness* de varias conformaciones al mismo tiempo. Además va a propiciar poder ejecutar ordenaciones y selecciones de conformaciones de varios puntos candidatos del receptor al mismo tiempo. En las funciones Seleccionar, Combinar e Incluir también se va a aplicar a nivel de puntos candidatos de acoplamiento y, además, en la función que calcula el *fitness*, va a permitir trabajar con varias conformaciones a la vez.

Algoritmo 4.3 Paralelismo de dos niveles

```
1: omp_set_nested(1)
2: omp_set_num_threads (threads-nivel1)
3: #pragma omp parallel for
4: Definición del bucle nivel 1
5:     Tratamiento de los elementos nivel 1
6:     omp_set_num_threads (threads-nivel2)
7:     #pragma omp parallel for
8:     Definición del bucle nivel 2
9:       Tratamiento de los elementos nivel 2
```

El esquema de dos niveles se va a utilizar en varias funciones del esquema, presentado especial atención a su correcta configuración en el calculo del *fitness*, dado su alto coste. En el primer nivel se va a paralelizar el cálculo de las conformaciones y en el segundo nivel se va a realizar el cálculo del potencial de Lennard-Jones entre varios átomos de cada conformación a la misma vez.

4.2. Aplicación del esquema metaheurístico paralelo al acoplamiento de moléculas

Dadas las características del acoplamiento entre dos compuestos, y la multitud de puntos donde se puede producir a priori, la paralelización de las funciones metaheurísticas va a enfocarse en realizar en paralelo varios puntos candidatos al acoplamiento o varias conformaciones a la vez, ya que al usar compuestos reales por muy pequeño que sea el receptor, el número de puntos candidatos va a ser más elevado que los hilos que disponga nuestro sistema multicore en memoria compartida en CPU, por lo que en la mayoría de los casos, la paralelización se va a realizar teniendo en cuenta esta premisa.
A la hora de establecer una nomenclatura para referirnos a los hilos en cada una de las funciones del esquema parametrizado paralelo o procedimientos internos que las forman, los vamos a identificar con la palabra *Threads* seguido de un número para indicar en qué nivel se definen esos hilos y una abreviatura de la función. Para verlo mucho más claro vamos a describir cada una de las funciones que componen el esquema.
Para expresar el orden de ejecución de los algoritmos de las funciones básicas o de los procedimientos que las componen, en el capítulo 3 se definieron una serie de variables. A continuación las recordamos para facilitar la lectura:

- m al número de puntos candidatos o puntos de acoplamiento del receptor.
- r al número de átomos de la proteína o receptor.
- l al número de átomos del ligando.
- o al número de elementos mejores en la función Combinar.
- p al número de elementos peores en la función Combinar.
- q al número de elementos mejores-peores en la función Combinar.
- n al número total de conformaciones o individuos.
- u al número de elementos seleccionados para la fase de mejora en la función de Inicializar, según porcentaje especificado en *PEMIni*.
- v al número de elementos seleccionados para la fase de Mejorar, según porcentaje especificado en *PEMImp*.
- w al número de elementos combinados.
- $th1$ al número de hilos del primer nivel.
- $th2$ al número de hilos del segundo nivel.

Las variables de los algoritmos paralelizados conservan la misma función que cuando fueron definidas en su modalidad secuencial. También conservan su misma función los parámetros algorítmicos y los dependientes del problema. Se añaden otros dos parámetros para definir el primer y segundo nivel de paralelismo según la función. Estos parámetros reciben el nombre de parámetros de paralelismo. Al igual que en el modelo

secuencial, se va a definir una constante que denominamos K_{MCPU} que agrupa los costes asociados de computar dentro de un sistema multicore. Cuanto más pequeña sea la constante K_{MCPU} el rendimiento del sistema será mejor.

Inicializar(S,ParamIni,Threads1Ini,Threads1Imp,Threads1Fit, Threads2Fit)

La función Inicializar recibirá cuatro parámetros paralelos *Threads1Ini*, *Threads1Imp*, *Threads1Fit* y *Threads2Fit*. Se parte del esquema del algoritmo 3.3 y la función Generar_posiciones_iniciales_aleatorias quedaría según el esquema del algoritmo 4.4.
Dado que cualquier compuesto real va a tener siempre varios átomos de carbono en su estructura por la propia naturaleza de los seres vivos, vamos a poder usar el esquema de paralelización de dos niveles visto en el algoritmo 4.3 y probar más adelante si el rendimiento con el uso de este esquema es superior, y no solo paralelizamos a un nivel.
El coste del algoritmo es $O(K_{MCPU} * n/th1)$.

Algoritmo 4.4 Pseudocódigo función *Generar_posiciones_iniciales_aleatorias (S,ParamIni,Threads1Ini)*

```
1: omp_set_num_threads(Threads1Ini)
2: #pragma omp parallel for
3: for i = 1 to n do
4:    Desplazar_y_rotar_aleatorio(i,angulo,radio)
5: end for
```

La función Calcular_fitness se define en su modalidad secuencial en el algoritmo 3.5, y su esquema paralelizado se muestra a continuación en el algoritmo 4.5. Su coste computacional es $O(K_{MCPU} * n * r * l/(th1 * th2))$.

Referente a la función Mejorar, descrita en el algoritmo 3.6, su esquema paralelizado figura en el algoritmo 4.6. El parámetro paralelo *Threads1Imp* se utiliza solamente si mejoramos en esta fase de Inicializar.
El coste computacional del algoritmo es $O(K_{MCPU} * IMEIni * (u/th1 + u * r * l/(th1 * th2)))$. Como en funciones anteriores, el mayor coste computacional lo soporta el término $u * r * l/(th1 * th2)$, que se refiere al cálculo del *fitness* de todos los individuos seleccionados para realizar la mejora.

Seleccionar(S,Ssel,ParamSel,Threads1Sel)

La función Seleccionar, definida en el algoritmo 3.7 va a recibir el parámetro paralelo *thread1Sel* definiendo el número de hilos para su paralelización, que va a ser en un solo nivel. El algoritmo paralelizado está definido en el algoritmo 4.7. El coste computacional es $O(K_{MCPU} * m/th1)$.

Algoritmo 4.5 Pseudocódigo función *Calcular_fitness(Proteina,Ligando,S, Threads1Fit,Threads2Fit)*

```
1: omp_set_nested(1)
2: omp_set_num_threads(Threads1Fit1)
3: #pragma omp parallel for
4: for k = 1 to n do
5:    omp_set_num_threads(Threads2Fit2)
6:    #pragma omp parallel for
7:    for i = 1 to r do
8:       for j = 1 to l do
9:          vdwEnergy = 4 * epsilon * (term12(i, j) − term6(i, j))
10:         vdwTerm+ = vdwEnergy
11:      end for
12:   end for
13:   S_energy[k] = vdwTerm
14:   vdw_term = 0
15: end for
```

Algoritmo 4.6 Pseudocódigo función *Mejorar(S,ParamIni,Threads1Imp, Threads1Fit,Threads2Fit)*

```
1: Stmp = S
2: K = 0
3: while K < IMEIni do
4:    omp_set_num_threads(Threads1Imp)
5:    for i = 1 to u do
6:       Desplazar_ y_rotar_aleatorio_mejora(elemento_Stmp,ángulo,
         desplazamiento)
7:    end for
8:    Calcular_finess(Proteina,Ligando,Stmp,Threads1Fit,Threads2Fit)
9:    Incluir_si_mejora(S,Stmp)
10:   K = K + 1
11: end while
```

Algoritmo 4.7 Pseudocódigo función *Seleccionar(S,Ssel,ParamSel,Threads1Sel)*

```
1: omp_set_num_threads(Threads1Sel)
2: #pragma omp parallel for
3: for i = 0 to m do
4:    Extraer_conjunto_referencia(S,Ssel,ParamSel)
5: end for
```

Combinar(Ssel,Scom,ParamCom,Threads1Com,Threads2Com, Threads1Fit,Threads2Fit)

La función Combinar, definida en su modalidad secuencial en el algoritmo 3.9 va a recibir cuatro parámetros paralelos *Thread1Com*, *Thread2Com*, *Threads1Fit* y *Threads2Fit*. Los dos primeros para paralelizar la combinación de elementos con dos niveles, y los otros dos para el cálculo de *fitness*. El esquema paralelizado está descrito en el algoritmo 4.8. Su coste computacional es $O(K_{MCPU} * (m * o/(th1 * th2) + m * p/(th1 * th2) + m * q/(th1 * th2) + w * r * l/(th1 * th2)))$. Los tres primeros términos de la suma representan el coste de combinar los elementos mejores, peores, y mejores con peores con un esquema paralelo de dos niveles. El último sumando representa el coste del cálculo del *fitness* con un esquema de dos niveles, de los individuos seleccionados para su combinación.
También es importante señalar, que en el mismo nivel de paralelismo del mismo algoritmo, puede haber diferente número de hilos. Esta situación se produce porque el número óptimo de hilos puede variar entre los diferentes cálculos que se realizan en el algoritmo.

Algoritmo 4.8 Pseudocódigo función *Combinar(Ssel,Scom,ParamCom, Threads1Com,Threads2Com,Threads1Fit,Threads2Fit)*

```
1: omp_set_nested(1)
2: omp_set_num_threads(Threads1Com)
3: #pragma omp parallel for
4: for i = 1 to m do
5:    omp_set_num_threads(Threads2Com)
6:    #pragma omp parallel for
7:    for j = 1 to o do
8:       Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
9:    end for
10:   #pragma omp parallel for
11:   for k = 1 to p do
12:      Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
13:   end for
14:   #pragma omp parallel for
15:   for l = 1 to r do
16:      Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
17:   end for
18: end for
19: Calcular_finess(Proteina,Ligando,Scom,Threads1Fit,Threads2Fit)
```

Mejorar(Scom,ParamImp,Threads1Imp,Threads1Fit, Threads2Fit)

Esta función, como sucede en la versión secuencial, es similar a la descrita en la función de Inicializar pero cambiando los parámetros metaheurísticos *IMEImp* y *PEMImp* que

representan la intensificación de la mejora y el porcentaje de selección de elementos combinados para realizarla, además del conjunto de datos de entrada. En este caso usamos solamente el valor del parámetro paralelo *Threads1Imp*, que define la paralelización a nivel de conformaciones. Los parámetros paralelos *Threads1Fit* y *Threads2Fit* son para paralelizar a dos niveles en el cálculo del *fitness*. Este cálculo se debe realizar cada vez que variamos la posición o rotación de cada una de las conformaciones, en cada una de las pasadas. Como sucede en la función anterior, el número de hilos por nivel en el algoritmo puede variar, ya que integra diversos cálculos. El coste computacional de esta función es $O(K_{MCPU} * IMEImp * (v/th1 + v * r * l/(th1 * th2)))$.

Incluir(S,Scom,ParamInc,Threads1Inc)

La función Incluir, definida en el algoritmo 3.10 va a recibir el parámetro *Thread1Inc* para realizar una paralelización a un nivel. Su esquema paralelo queda definido en el algoritmo 4.9. El coste del algoritmo es $O(m/th1)$.

Algoritmo 4.9 Pseudocódigo función *Incluir(S,Scom,ParamInc,Threads1Inc)*

```
1: Obtener número de elementos mejores y peores a incluir, según porcentajes de NEMInc y NEPInc, de Scom.
2: omp_set_num_threads(Threads1Inc)
3: #pragma omp parallel for
4: for i = 0 to m do
5:    Copiar_mejores (S,Scom,ParamInc)
6:    Copiar_peores(S,Scom,ParamInc)
7: end for
```

Funciones		Algoritmos	Modelo Paralelo Multicore
Inicializar	Generar_Posiciones_Iniciales	Algoritmo 4.4	$O(K_{MCPU} * n/th1)$
	Cálculo del Fitness	Algoritmo 4.5	$O(K_{MCPU} * n * r * l/(th1 * th2))$
	Mejorar	Algoritmo 4.6	$O(K_{MCPU} * IMEIni * (u/th1 + u * r * l/(th1 * th2)))$
Seleccionar		Algoritmo 4.7	$O(K_{MCPU} * m/th1)$
Combinar		Algoritmo 4.8	$O(K_{MCPU} * (m * o/(th1 * th2) + m * p/(th1 * th2) + m * q/(th1 * th2) + w * r * l/(th1 * th2)))$
Mejorar		Algoritmo 4.6	$O(K_{MCPU} * IMEImp * (v/th1 + v * r * l/(th1 * th2)))$
Incluir		Algoritmo 4.9	$O(K_{MCPU} * m/th1)$

Tabla 4.1: Resumen de los tiempos teóricos de ejecución de los algoritmos considerados en el modelo Multicore con CPU

En la tabla 4.1 se muestran los costes teóricos de las funciones del esquema paralelo parametrizado. Como en el modelo secuencial, las funciones que integran el algoritmo 4.5, que representa el coste del cálculo del *fitness*, tendrán un coste superior. Este cálculo está integrado en las funciones de Inicializar, Combinar y Mejorar. Los algoritmos 4.4, 4.5 y 4.6 corresponden a la fase de Inicializar. En la fase de Mejorar se repite el algoritmo 4.6. El algoritmo 4.7 se identifica con la fase de Seleccionar y, en cuanto al coste de las fases de Combinar e Incluir, este queda reflejado en los algoritmos 4.8 y 4.9.

4.3. Resultados computacionales en sistemas basados en memoria compartida en CPU

En esta sección vamos a probar diferentes combinaciones de parámetros metaheurísticos, ampliando la tabla 3.2 para diversificar más las pruebas e intentar obtener mejores valores de *fitness*. Además de evaluar el ahorro obtenido en coste computacional por el uso de un sistema multicore, se van a variar las combinaciones de hilos sin variar el resto de parámetros de la metaheurística. Seguidamente se va a evaluar el valor final de *fitness*, como en el modelo secuencial, para cada una de las combinaciones de parámetros metaheurísticos. Para finalizar se van a estudiar individualmente el ahorro computacional en cada una de las funciones del esquema metaheurístico parametrizado comparando el valor en modo secuencial y usando un sistema multicore.
Dadas las conclusiones extraídas del modelo secuencial en el capítulo anterior, se van a variar las ocho combinaciones para intentar obtener mejores valores de *fitness* que en la versión secuencial, elevando los parámetros metaheurísticos de cada una de las combinaciones. Los nuevos valores se encuentran en la tabla 4.2. Esta variación se va a centrar en el aumento del número de conformaciones iniciales por punto candidato, mejora de mayor número de elementos, elevar el número de combinaciones o aumentar el número de pasadas.
Quedó demostrado en el estudio secuencial del capítulo anterior, que elevar los valores de este conjunto de parámetros metaheurísticos va a conllevar el aumento del tiempo de ejecución. En este punto influye la paralelización de las partes del código con mayores costes y que fueron detectadas en ese estudio. Este ahorro computacional va a ser analizado comparando los tiempos de ejecución entre la versión secuencial y paralela de las combinaciones de la tabla 3.2 con el número de hilos óptimo para cada función, extraído del estudio experimental realizado con los valores de la tabla 3.2.
Con respecto a los compuestos elegidos para esta fase experimental, se han escogido los mismos compuestos que en la fase secuencial para ver si se produce una mejora del *fitness*. Estos compuestos son *2bsm_rec.mol2* como receptor y *2bsm_lig.mol2* el ligando.

4.3.1. Análisis del coste computacional

En esta sección, primeramente se va a estudiar el ahorro en coste computacional de usar técnicas paralelas con respecto al modelo secuencial tradicional y, seguidamente, se va a estudiar en profundidad la evolución del coste de las funciones Inicializar y Combinar. Este estudio comprenderá la variación del número de hilos, obteniendo el número de hilos óptimo de forma experimental para cada función.

4.3.1.1. Estudio experimental del modelo paralelo

Para realizar este estudio, y teniendo un esquema de dos niveles, se va a proceder primero al aumento del primer nivel, dejando el segundo nivel fijo, y seguidamente aumentaremos y disminuiremos cada nivel a la vez hasta encontrar el valor de cada uno donde el rendimiento es mayor.

	COMBINACIONES							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
NEIIni	64	64	96	96	128	128	256	256
PEMIni	0	20	0	20	0	20	0	20
IMEIni	0	10	0	10	0	10	0	10
NEFMIni	100	50	100	50	100	100	100	50
NEFPIni	0	50	0	50	0	0	0	50
NEMSel	100	50	100	100	100	50	50	100
NEPSel	0	50	0	0	0	50	50	0
NMMCom	50	50	50	100	50	25	50	100
NPPCom	0	25	0	0	0	25	25	0
NMPCom	0	50	0	0	0	50	25	0
PEMImp	10	25	25	10	25	25	25	10
IMEImp	5	10	10	5	10	10	10	5
NEMInc	100	80	80	100	100	80	80	100
NIRFin	5	5	5	5	5	5	5	5
NMIFin	20	20	20	20	20	20	20	20

Tabla 4.2: Valores de los parámetros metaheurísticos asociados a cada combinación.

Utilizaremos para este estudio las dos funciones que acumulan en su interior el mayor coste computacional, Inicializar y Combinar, con los valores de los parámetros metaheurísticos de la tabla 3.2 de las combinaciones 1, 4, 5 y 8. Aunque la función mejorar también es costosa, es llamada en varias combinaciones en la parte de inicialización, por lo que ya es tenida en cuenta.
Este coste temporal, como se dedujo en el estudio secuencial, proviene en su mayor parte del cálculo del potencial de Lennard-Jones entre los dos compuestos.
En las figuras 4.1 y 4.2 se muestra la ejecución en Saturno, donde se aprecia la disminución del tiempo de ejecución de las funciones Inicializar y Combinar conforme va en aumento el número de hilos del primer nivel, a partir de este momento denominado N1, con el segundo nivel, o N2, fijo en 2 hilos en cada una de las funciones.
Con respecto a la variación del nivel dos, vamos a aumentar el número de hilos de este nivel sin que la combinación de niveles sobrepase el total de *cores* del nodo. Con estos cálculos se va a comprobar de forma experimental qué combinación es idónea para este caso concreto de estudio.
En las figuras 4.3 y 4.4 se representa este análisis, mostrando la evolución del tiempo en las fases de Inicializar y Combinar en el nodo Saturno, y observando la disminución paulatina del tiempo de ejecución conforme se va reduciendo el valor del número de hilos del segundo nivel. Cuando todos los hilos se sitúan en el primer nivel se obtiene el mejor tiempo.
En las figuras 4.5 y 4.6 se muestra la ejecución en el nodo Jupiter, de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), ajustado a los cores del nodo. Como sucede en el nodo Saturno, se produce una disminución del tiempo de ejecución conforme se aglutinan los hilos en el primer de paralelismo, obteniendo el mejor resultado cuando concentramos todos los hilos en el primer nivel.

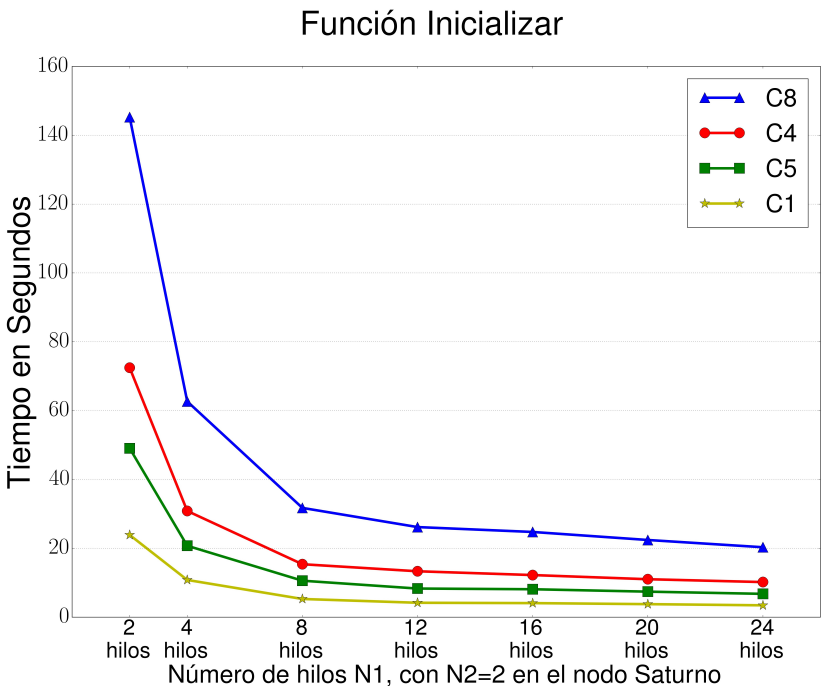

Figura 4.1: Tiempo de ejecución de la función Inicializar en el nodo Saturno, de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), variando el número de hilos del nivel uno, con 2 hilos en el segundo nivel de cada una de las funciones.

El estudio se ha realizado en el nodo Saturno usando los 24 cores con *hyperthreading* y en Jupiter usando sus 12 cores. En la figura 4.3 se determina el tamaño óptimo para los parámetros de paralelismo de la función Inicializar. En este caso, todas las combinaciones que se están sometiendo al estudio, con un valor en el nivel N1 de 48 hilos y en el nivel N2 de 1 hilo se obtiene el mejor rendimiento en tiempo de ejecución. En la figura 4.4 se exponen los resultados de la función Combinar, con un resultado similar al obtenido en la función Inicializar. Con respecto a las funciones Seleccionar e Incluir, ambas tienen un coste que supone menos del 0.3 % del coste total del esquema, aunque se ha constatado que un número de 48 hilos es el óptimo para estas funciones, según los resultados obtenidos en las combinaciones que han formado parte del estudio experimental realizado.

4.3.1.2. Modelo secuencial vs Modelo paralelo en OpenMP

En las tablas 4.3 y 4.4 se comparan la suma de los tiempos obtenidos en cada una de las funciones del esquema metaheurístico cuando se realiza una sola pasada del esquema, con las combinaciones de la tabla 3.2. Las tablas recogen las dos versiones, tanto la secuencial como la paralela con el número de hilos óptimo para cada caso. En la columna denominada *Speed-up* se muestra la ganancia de las veces que la versión paralela es mejor que la secuencial. Este valor es calculado dividiendo el valor de la versión secuencial entre el valor de la versión paralela. Las simulaciones se han realizado

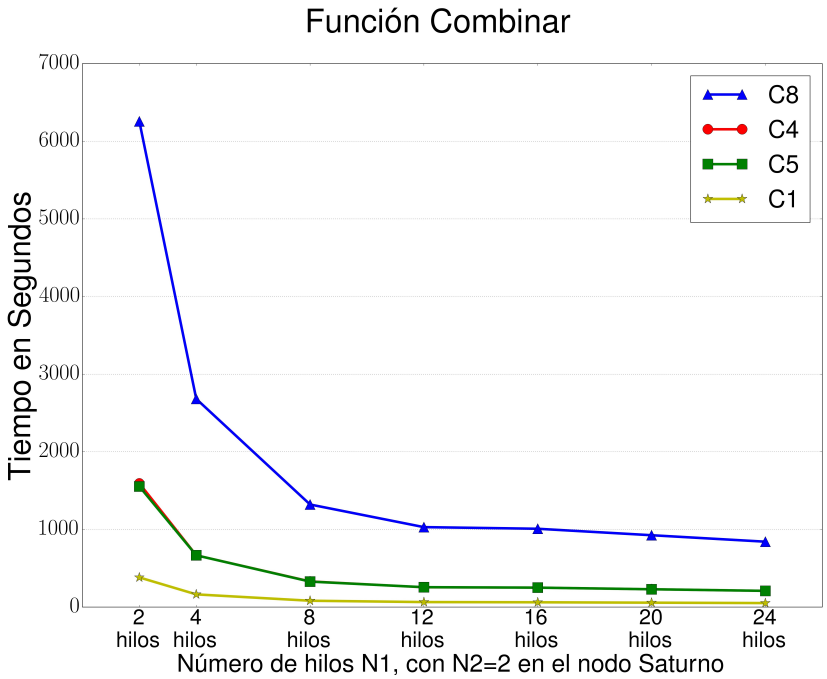

Figura 4.2: Tiempo de ejecución de la función Combinar en el nodo Saturno de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), variando el número de hilos del nivel uno, con 2 hilos en el segundo nivel de cada una de las funciones.

en los nodos Jupiter y Saturno del cluster HETEROSOLAR, con un valor de 48 hilos en el primer nivel de paralelismo en Saturno y 12 hilos en Jupiter.

COMBINACIONES	Versión Secuencial	Versión Paralela	Speed-up
1	754.39	69.93	10.7
2	430.36	40.31	10.6
3	600.07	55.87	10.7
4	3016.08	278.94	10.8
5	2967.37	275.52	10.7
6	1679.67	157.42	10.6
7	2036.41	190.47	10.7
8	11870.49	1101.48	10.7

Tabla 4.3: Comparativa del tiempo de ejecución entre la versión secuencial y versión paralela en OpenMP de las combinaciones de la tabla 3.2. Ejecuciones realizadas en el nodo Jupiter con 12 hilos en el primer nivel. Tiempo medido en segundos.

Analizando estos resultados, no hay ninguna combinación que obtenga una mejora sustancial con respecto a otra. Se obtiene un ahorro muy importante con respecto al método secuencial por la paralelización del cálculo, reflejándose en un mejor rendimiento. El *speed-up* obtenido en todas las combinaciones es muy satisfactorio, variando

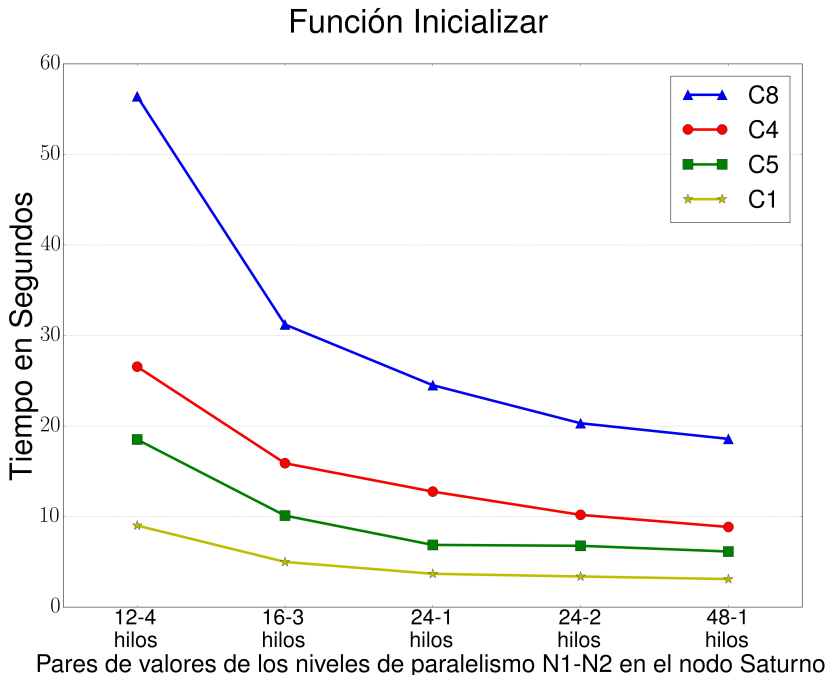

Figura 4.3: Tiempo de ejecución de la función Inicializar en el nodo Saturno, de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), ajustando el número de hilos del nivel uno y dos.

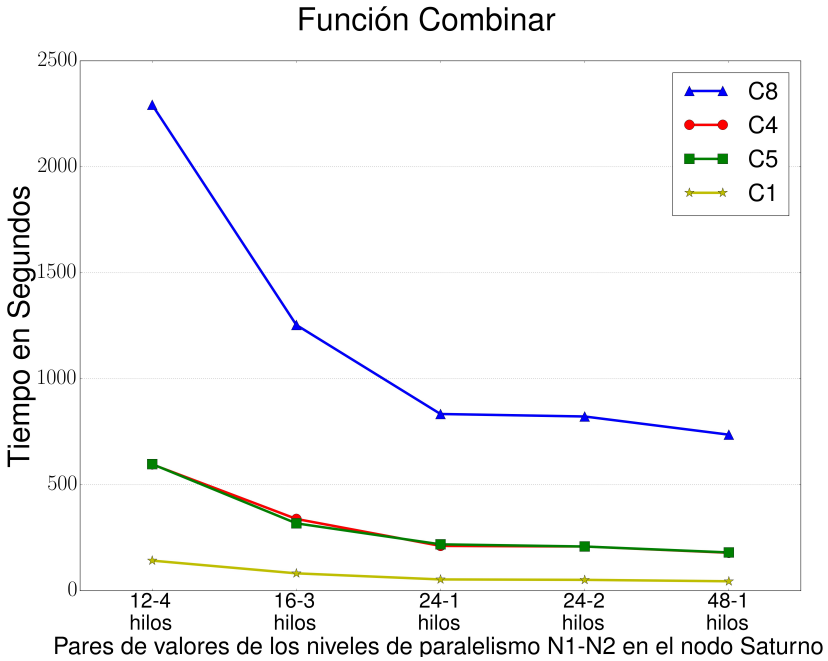

Figura 4.4: Tiempo de ejecución de la función Combinar en el nodo Saturno, de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), ajustando el número de hilos del nivel uno y dos.

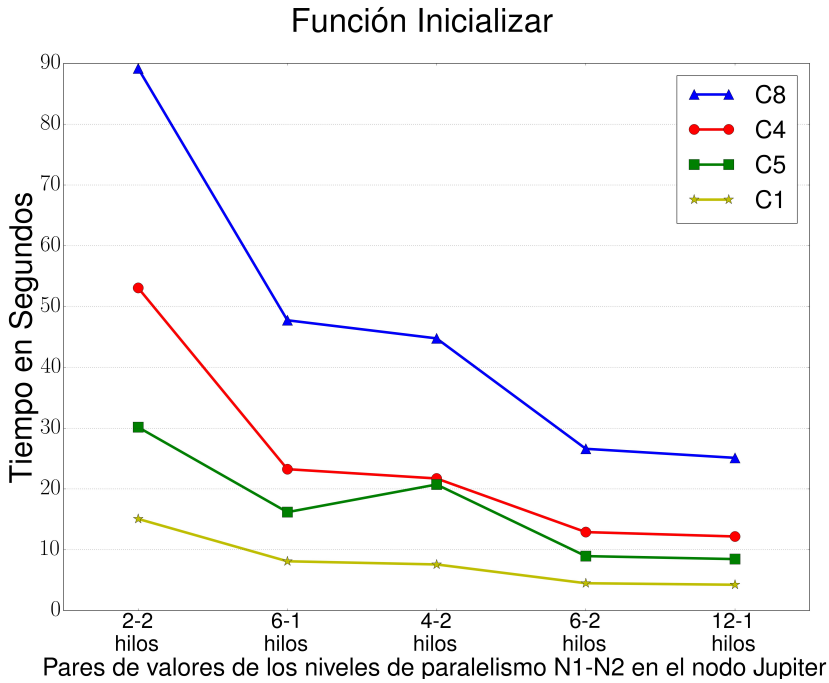

Figura 4.5: Tiempo de ejecución la función Inicializar en el nodo Jupiter, de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), ajustando el número de hilos del nivel uno y dos.

COMBINACIONES	Versión Secuencial	Versión Paralela	Speed-up
1	1035.01	47.86	21.6
2	589.57	28.48	20.7
3	826.76	39.66	20.8
4	4142.36	188.25	22
5	4073.55	187.21	21.7
6	2320.59	107.49	21.6
7	2795.38	132.65	21
8	16272.27	755.46	21.5

Tabla 4.4: Comparativa del tiempo de ejecución entre la versión secuencial y versión paralela en OpenMP de las combinaciones de la tabla 3.2. Ejecuciones realizadas en el nodo Saturno con 48 hilos en el primer nivel. Tiempo medido en segundos.

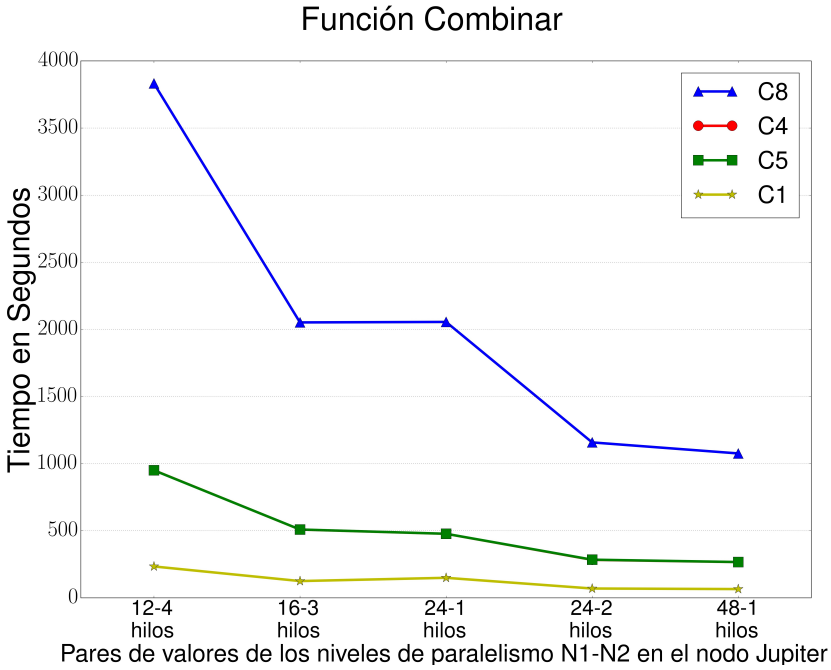

Figura 4.6: Tiempo de ejecución la función Combinar en el nodo Jupiter, de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), ajustando el número de hilos del nivel uno y dos.

los resultados entre 20.7 y 22.

4.3.2. Análisis del *fitness*

En las tablas 4.5 y 4.6, se muestran los resultados del fitness y tiempo de ejecución para cada una de las combinaciones de la tabla 4.2. Como en la fase secuencial, se ha realizado la ejecución de las ocho combinaciones un total de diez veces obteniendo su media, dado que la aleatoriedad de las ejecuciones hace que obtengamos valores distintos de *fitness* y tiempo de ejecución en cada una de ellas. Este valor medio es el que se ha colocado en la tabla. La diferencia entre el resultado de cada uno de los experimentos con respecto a la media es de ±22 unidades en el nodo Saturno y ±30 en el nodo Jupiter.

	COMBINACIONES							
	1	2	3	4	5	6	7	8
Fitness	-107.11	-105.21	-107.12	-187.93	-132.85	-103.26	-113.51	-108.21
Tiempo (seg)	2185.69	2389.3	8904.4	13241.5	15576.2	3854.5	14964.6	27983.4

Tabla 4.5: Valores de *fitness* y tiempo de ejecución de cada una de las combinaciones de la tabla 4.2. en el nodo Saturno

Entre los resultados de *fitness* y tiempos de ejecución obtenidos entre los nodos Saturno

	COMBINACIONES							
	1	2	3	4	5	6	7	8
Fitness	-95.31	-119.35	-137.57	-99.93	-156.74	-211.71	-123.31	-101.14
Tiempo (seg)	2646.13	798.71	3750.12	8597.58	5407.49	4979.27	9686.17	26139.2

Tabla 4.6: Valores de *fitness* y tiempo de ejecución de cada una de las combinaciones de la tabla 4.2. en el nodo Jupiter

y Jupiter reflejados en las tablas 4.4 y 4.5, se observan algunas diferencias significativas entre ciertas combinaciones ejecutadas en ambos nodos y que vamos a comentar:

- Partiendo de la base de que las características del hardware de Jupiter es mayor que en Saturno, se observa una elevada diferencias en el valor de *fitness* de la combinación 4, aunque puede explicarse debido a que se ha realizado una intensificación superior en Saturno con un mayor número de pasadas. Esto explica la obtención de mejores resultados al aumentar el número de pasadas, no estancándose en un mínimo local inferior como sucede en Jupiter.

- En la combinación 6 ocurre lo contrario; la computación y el número de pasadas que se realizan en Jupiter es bastante superior que la realizada en Saturno, aunque con menor número de *cores*.

Con los valores de la tabla 4.2 de las ocho combinaciones, podemos extraer una serie de conclusiones:

- Obtenemos mejores resultados de media en la mayoría de los casos con respecto al modelo secuencial, aumentando los valores de los parámetros metaheurísticos. El mejor resultado se sitúa en -187.93 del nodo Saturno y -211.71 en el nodo Jupiter, más cerca de los -277.35 óptimos. Este acoplamiento óptimo se extrae de computar el receptor y ligando en la mejor posición de ambos ya conocida [46].

- El coste computacional sigue siendo elevado, aunque usemos el número de hilos óptimo obtenido mediante experimentación.

- Usar técnicas masivamente paralelas en la GPU puede ser otra posible solución para obtener mejores valores de *fitness*, pudiendo aumentar los valores de los distintos parámetros metaheurísticos con menor coste computacional.

4.4. Auto-optimización en sistemas Multicore

Cuando realizamos la implementación de una aplicación, no sabemos a priori cual es el entorno en que va a ser ejecutada. Esto significa que dependiendo de los parámetros de paralelismo que se introduzcan de una manera aleatoria y a decisión del usuario final, el rendimiento de la aplicación puede verse seriamente afectado. Por ello, son necesarios una serie de cálculos y pruebas previas para que el rendimiento de la aplicación se adapte al entorno de ejecución previsto.

Este campo de investigación se ha ido diversificando en diferentes disciplinas. En librerías de álgebra lineal como Atlas [62] para sistemas multicore, se realiza este proceso denominado de *auto-tunning* en su instalación, y se generan los ejecutables con la configuración óptima. También si hablamos de metaheurísticas, se han desarrollado estudios para auto-optimizar estas técnicas en sistemas multicore [11], y en multicore y GPU [55].
Existen varias técnicas si se quiere abordar este proceso [63], en una primera fase de instalación se podría proceder a realizar una serie de ejecuciones sobre un conjunto de instalación, explorando todas las posibles variaciones de los parámetros de paralelismo. Seguidamente se incluirían en el código de la aplicación, los valores de los parámetros óptimos de paralelismo y se compilaría el código. En la primera ejecución real también se puede realizar un conjunto de pequeños experimentos y determinar la mejor configuración de parámetros paralelos para nuestro problema concreto, y almacenarlos para futuras ejecuciones.
En nuestro caso particular, si se elige la opción de auto-optimización se ha optado por la segunda opción, realizando el proceso en la primera ejecución real y almacenar el resultado de estos parámetros paralelos. Estos parámetros se ajustarán en función de unas pequeñas ejecuciones previas con un conjunto de entrenamiento. Estas ejecuciones se realizarán primeramente sobre la función de cálculo de *fitness*, que ha sido detectada en el capítulo anterior como la más costosa de forma experimental, aglutinando más del 98 % del cómputo. A continuación se ajustarán los parámetros paralelos del resto de funciones básicas del esquema metaheurístico paralelo.
El proceso genera un fichero de configuración para almacenar el valor óptimo de hilos en cada uno de los niveles de paralelismo de cada una de las funciones. En ejecuciones posteriores se detecta la presencia de ese fichero, se extrae la información, y se aplican los valores a la ejecución actual. Las fases que comprenden este proceso son:

- Generar un conjunto de entrenamiento.
- Establecer el número de hilos al máximo e ir calculando el tiempo de cada función.
- Ir progresivamente disminuyendo el valor del número de hilos y ejecutar con cada valor la función hasta llegar a la unidad, almacenando el mejor resultado.
- Repetir para los diferentes niveles de paralelismo considerados en la función.
- Repetir este proceso con todas las funciones paralelizadas del esquema.
- Con el mejor resultado de cada función se genera un fichero de configuración para ejecuciones posteriores.

En esta fase no nos preocupamos del valor de *fitness*, dado que es una fase única y exclusivamente destinada a obtener la combinación óptima de hilos en cada nivel de paralelismo. El resultado a modo de ejemplo de una de estas ejecuciones se encuentra en el Anexo A.
Como se puede observar en este Anexo, se produce una primera lectura de parámetros de la ejecución. A continuación se inicia la fase de auto-optimización si no se ha realizado antes en esta máquina y el usuario lo indica, comenzando con la generación del

conjunto de entrenamiento.
Seguidamente, se prosigue con los dos niveles de paralelismo del cálculo del *fitness*, fijando el segundo nivel a una unidad y el primero con el máximo número de hilos del nodo. Se va ejecutando el cálculo sobre el conjunto de entrenamiento, disminuyendo progresivamente el valor del número de hilos de uno en uno hasta la unidad, almacenando para cada uno de los valores su tiempo. Para ejecutar las pruebas con dos niveles, se divide el máximo número de hilos del nodo entre dos y se le asigna al primer nivel. El segundo nivel toma el valor de dos y se va ejecutando de nuevo el cálculo sobre el conjunto de entrenamiento incrementando en dos unidades el segundo nivel y dividiendo entre dos el primero hasta llegar a dos unidades. Nos quedamos con la combinación que mejor tiempo ha obtenido.
A continuación se trabaja la optimización en la función de Inicializar, que a parte de finalizar con el cálculo del *fitness*, dispone de un nivel de paralelismo en la generación de elementos iniciales por lo que procedemos a encontrar su valor óptimo de hilos. El proceso consiste en obtener el valor máximo de hilos e ir disminuyendo hasta la unidad para ver que valor es el más idóneo para este caso.
La función Seleccionar tiene un solo nivel de paralelismo y se procede de la misma forma que el caso de la generación de elementos iniciales de la función de Inicializar. Seguidamente se procede a encontrar el valor óptimo para la función Combinar procediendo de la misma manera que en el cálculo del *fitness*. Por último se procede a calcular el valor óptimo de las funciones Mejorar, sin el cálculo del *fitness* optimizado anteriormente, e Incluir, de la misma forma que la función Seleccionar y generación de elementos iniciales. Cuando finalizan todas las fases se muestra el tiempo transcurrido y el valor de los niveles de paralelismo de cada función después de la fase de auto-optimización.
En la tabla 4.7 se muestran los tres conjuntos de parámetros paralelos que se van a evaluar, para ver la importancia de realizar la fase de auto-optimización antes de ejecutar la aplicación en una nueva máquina. En las dos primeras columnas se aportan dos definiciones manuales de parámetros, que un usuario sin conocimiento del funcionamiento de la aplicación puede introducir. En la primera configuración se utiliza la mitad de hilos para el primer nivel de paralelismo porque tiene menos computación, y en la segunda configuración se usan todos los hilos para cada una de las funciones, distribuidos según disponga el usuario cuando se utilizan dos niveles de paralelismo. En la tercera columna se aportan los valores de los parámetros paralelos, obtenidos después de ejecutar la fase de auto-optimización en el nodo Jupiter. Ambas combinaciones manuales suponen que el usuario final conoce el número de cores del nodo que va a ejecutar la aplicación, aunque en realidad no tiene porque saberlo.

En la tabla 4.8 podemos apreciar el SPEED-UP que se obtiene al ejecutar la aplicación con los valores de la tabla 4.7. El tiempo empleado en la fase de auto-optimización en Jupiter es de 137,43 segundos, y cada vez que se realice una ejecución con los parámetros paralelos óptimos, obtenemos una ganancia en ambas combinaciones de parámetros metaheurísticos. Esto significa, que si un usuario final no conoce que parte del cálculo metaheurístico soporta la mayor parte del cómputo, puede obtener un rendimiento bajo en tiempos de ejecución. En este caso, ejecutar la aplicación mas de 4 veces en la misma máquina compensaría el gasto de la fase de auto-optimización.

Parámetros Paralelos	Configuración Manual 1	Configuración Manual 2	Configuración Óptima
Threads1Ini	6	12	11
Threads1Fit	6	6	12
Threads2Fit	2	2	1
Threads1Sel	6	12	8
Threads1Com	6	6	11
Threads2Com	2	2	1
Threads1Imp	6	12	9
Threads1Inc	6	12	8

Tabla 4.7: Combinaciones de parámetros paralelos para sistemas multicore

Combinaciones	Configuración Manual 1	Configuración Manual 2	Configuración Óptima	SPEED-UP
4	319.11	315.06	279.11	1.1
8	1280.76	1252.82	1099.87	1.1

Tabla 4.8: Tiempo de ejecución de las combinaciones 4 y 8 de la tabla 3.2 en el nodo Jupiter con los valores de los parámetros paralelos de la tabla 4.7. SPEED-UP entre la configuración óptima y la configuración manual 1. Tiempo medido en segundos.

4.5. Conclusiones

Después de realizar este estudio sobre la aplicación de paralelismo con OpenMP usando sistemas multicore basados en memoria compartida, llegamos a un conjunto de conclusiones:

- Aplicando paralelismo obtenemos una mejora sustancial en coste computacional, consiguiendo un *speed-up* entre 20.7 y 22 en el mejor de los casos. Este cálculo se muestra en las tablas 4.3 y 4.4.
- Con este ahorro podemos elevar el valor de ciertos parámetros metaheurísticos para intentar mejorar los valores de *fitness* obtenidos en la modalidad secuencial.
- Para obtener los mejores rendimientos usando este esquema paralelo en diferentes máquinas, se debe realizar previamente una serie de pruebas para adaptar el valor de los diferentes hilos en cada una de las fases, para que la mejora sea la más óptima posible. Esto se puede realizar aplicando la opción de auto-optimización descrita en apartados anteriores y que puede llevar a obtener el mejor rendimiento posible con esta implementación.
- En general, el tiempo sigue siendo elevado para cualquier combinación. Probar con técnicas masivamente paralelas en la GPU se convierte en un paso necesario para hacer que esta reducción sea realmente significativa e importante.

Capítulo 5

Técnicas masivamente paralelas aplicadas al esquema metaheurístico híbrido parametrizado en multicore + GPU

En este capítulo vamos a introducir la GPU en las zonas donde se concentran los mayores costes computacionales del esquema metaheurístico parametrizado. Aprovechamos la gran cantidad de núcleos a nuestro alcance en la GPU para explotar aún más el esquema paralelo. Se adaptarán las funciones necesarias para aprovechar los recursos que nos ofrece la GPU de la manera más óptima posible, combinándolas con las técnicas de optimización que han sido utilizadas hasta ahora. A continuación se explicará el proceso de auto-optimización incorporado a la aplicación para encontrar el valor óptimo de hilos por bloque en cada uno de los *kernels* para un cierto dispositivo. Finalizaremos este capítulo con un estudio experimental y las conclusiones que podemos extraer del mismo.

5.1. Esquema parametrizado paralelo híbrido usando sistemas de memoria compartida en CPU y GPU

Cuando utilizamos la GPU, los costes de preparar los datos para su proceso junto con los movimientos de datos en memoria que se deben realizar, hacen que su uso se centre en los procesos que soporten mayor carga computacional, pudiendo utilizar para el resto de funciones un esquema paralelo basado en CPU con memoria compartida.
Por tanto se plantea para este problema una solución híbrida, tomando como partida los estudios anteriores y aplicando técnicas masivamente paralelas en la GPU en las partes donde se haya detectado esta necesidad.
Al igual que sucede con el número de hilos en CPU, donde al variar su número podemos aumentar o reducir el rendimiento, en la GPU pasa una situación similar con el número

de hilos por bloque. Por tanto, se van a añadir una serie de parámetros de paralelismo a la metaheurística para especificar este número y encontrar experimentalmente el número óptimo de hilos por bloque para cada una de las funciones que usen la GPU para sus cálculos.
Para identificar estos parámetros, vamos a establecer una nomenclatura muy similar a usada en el capítulo 4 para nombrar los hilos en cada nivel con la palabra *Threads*, pero añadiendo que nos referimos a bloque, por lo que usaremos *threadsblock* seguido de un sufijo identificando al *kernel*. Por último, se añadirá el sufijo que identifica a cada una de las funciones del esquema si es propio exclusivamente de ella.
Para identificar las llamadas a la GPU para la ejecución de funciones, se van a especificar entre los signos < y > primeramente el número bloques y a continuación el número de hilos por bloque. El esquema de llamada sería *funcion<n_bloques, hilos_bloque>(parámetros)*. En el algoritmo 5.1 se describe la estructura de este nuevo esquema, conservando las mismas funciones las variables con nombre similar a las descritas en el algoritmo 4.1. Al igual que en capítulos anteriores, para facilitar la lectura y comprensión de los algoritmos, recordamos la definición de una serie de variables:

- m al número de puntos candidatos o puntos de acoplamiento del receptor.
- r al número de átomos de la proteína o receptor.
- l al número de átomos del ligando.
- n al número total de conformaciones o individuos.
- o al número de elementos mejores en la función Combinar.
- p al número de elementos peores en la función Combinar.
- q al número de elementos mejores-peores en la función Combinar.

Como en capítulo anterior, las variables de los algoritmos paralelizados conservan la misma función que cuando fueron definidas en su modalidad secuencial. También conservan su misma función los parámetros algorítmicos y del problema que definen el comportamiento de la aplicación.
El trabajo con la GPU requiere un tiempo de activación del dispositivo t_a, un tiempo de transferencia de datos $t_{HostToGpu}$ del *host* hacia el dispositivo de lo que necesitemos para computar, y un tiempo de transferencia $t_{GpuToHost}$ desde el dispositivo al *host* para devolver los resultados. El número de elementos a tratar es similar al modelo secuencial y multicore, aunque lo multiplicamos por una constante que denominaremos C_{GPU} que determina el gasto de computar en GPU. Por tanto el coste total de la función de Inicializar, que denominamos K_{GPU}, será la suma de todos esos costes parciales más el tiempo de computar en la GPU. Es importante señalar que cuanto más pequeña sea la constante K_{GPU} mejor rendimiento obtendremos, al igual que sucede en los modelos secuencial y multicore.
A continuación se va a describir el esquema de las diversas funciones que van a sufrir variación en su procedimientos con el uso de la GPU en sus procedimientos internos. Las funciones Seleccionar e Incluir del esquema general del algoritmo 5.1 no sufren

modificaciones y sus esquemas algorítmicos se mantienen como las descritas en los algoritmos 4.7 y 4.9 respectivamente.
El vector *Devices* descrito en el capítulo 2, almacena las características del dispositivo y, a partir de ahora, se pasará como parámetro a las funciones que hagan uso de la GPU, ya que en esta estructura se almacenan las características del dispositivo.

Algoritmo 5.1 Esquema parametrizado paralelo híbrido multicore y GPU

```
1: Inicializar(S,Devices,ParamIni,Threads1Ini,ThreadsblockMoveIni,
   ThreadsblockRot,ThreadsblockQuat,ThreadsblockRandom,
   ThreadsblockMoveImp,ThreadsblockIncImp,ThreadsblockFit)
2: while no Fin(S) do
3:    Seleccionar(S,Ssel,ParamSel,Threads1Sel)
4:    Combinar(Ssel,Scom,Devices,ParamCom,Threads1Com,Threads2Com,
      ThreadsblockFit)
5:    Mejorar(Scom,Devices,ParamImp,ThreadsblockMoveImp,ThreadsblockRot,
      ThreadsblockIncImp,ThreadsblockFit)
6:    Incluir(Scom,S,ParamInc,Threads1Inc)
7: end while
```

Inicializar(S,Devices,ParamIni,Threads1Ini,ThreadsblockMoveIni, ThreadsblockRot,ThreadsblockQuat,ThreadsblockRandom, ThreadsblockMoveImp,ThreadsblockIncImp,ThreadsblockFit)

La función Inicializar recibe ocho parámetros específicos para su paralelización. *Threads1Ini* indica el número de hilos para usar con CPU para procedimientos internos en los que la GPU no va a estar presente, mientras que *ThreadsblockMoveIni* y *ThreadsblockRot* para generación de posiciones iniciales de las conformaciones, *ThreadsblockQuat* y *ThreadsblockRandom* para Inicializar el conjunto de número aleatorios y el vector de rotaciones de las conformaciones, *ThreadsblockFit* va a ser específicamente utilizado para el cálculo del *fitness*, *ThreadsblockMoveImp* para paralelizar el movimiento de las conformaciones en la fase de mejora y *ThreadsblockIncImp* va a ser utilizado para paralelizar la función de inclusión de los elementos mejorados.
El coste de los *kernels* Init_Rot, Init_Random, Mover y Rotar tiene un coste de $O(n * K_{GPU})$. El cálculo del *fitness* tiene un coste de $O(n * u * l * K_{GPU})$, y la función Mejorar tiene un coste de $O(K_{GPU} * IMEIni * (n + n + n * u * l + n))$.
En este caso concreto de la función de Inicializar el número de conformaciones n tiene un valor de $m * NEIIni$, aunque dependiendo de los valores de los parámetros algorítmicos esta cantidad puede variar en cada una de las funciones del esquema metaheurístico parametrizado paralelo.
El esquema de la función Inicializar está definido en el algoritmo 5.2. Al usar la GPU, cambia totalmente la estructura seguida hasta el momento de los modelos secuencial y paralelo con CPU.
Usamos en las funciones Mover y Rotar dos de los parámetros metaheurísticos específicos para la GPU indicando el número de hilos por bloque que queremos en estos

Algoritmo 5.2 Pseudocódigo función *Inicializar(S,Devices,ParamIni,Threads1Ini, ThreadsblockMoveIni,ThreadsblockRot,ThreadsblockQuat, ThreadsblockRandom,ThreadsblockMoveImp,ThreadsblockIncImp, ThreadsblockFit)*

```
1: Host_To_GPU(S,Stmp)
2: Init_Rot<conformaciones/ThreadsblockQuat,ThreadsblockQuat>
   (Stmp,ParamIni)
3: Init_Random<conformaciones/ThreadsblockRandom,ThreadsblockRandom>
   (Stmp,ParamIni)
4: Mover<conformaciones/ThreadsblockMoveIni,ThreadsblockMoveIni>
   (Stmp,ParamIni)
5: Rotar<conformaciones/ThreadsblockRot,ThreadsblockRot>
   (Stmp,ParamIni)
6: if PEMIni >0 then
7:    Mejorar(Stmp,Devices,ParamIni,ThreadsblockMoveImp,ThreadsblockRot,
      ThreadsblockIncImp,ThreadsblockFit)
8: end if
9: Calcular_fitness<conformaciones/ThreadsblockFit,ThreadsblockFit>
   (Stmp,ParamIni)
10: GPU_To_Host(S,Stmp)
```

procedimientos, mientras que el otro parámetro, *ThreadsblockFit*, se aplica en la función Calcular_fitness cuyo esquema para cada uno de los hilos se muestra en el algoritmo 5.3.

Algoritmo 5.3 Pseudocódigo función *Calcular_fitness <conformaciones/ThreadsblockFit,ThreadsblockFit>(S,ParamIni)*

```
1: pos = posicion_atomo
2: conf = obtener_conformacion_atomo()
3: for i = 1 to r do
4:    vdwEnergy = 4 * epsilon * (term12(i, pos) − term6(i, pos))
5:    vdwTerm+ = vdwEnergy
6: end for
7: Sincronizar_hilos()
8: energia[conf] = Sumar_reduciendo_atomos_conformacion()
```

Este esquema es ejecutado por cada uno de los hilos, y en secciones posteriores se estudiarán mecanismos de optimización sobre el mismo, dado que experimentalmente en los capítulos 3 y 4 se ha determinado que es la función de cálculo más costosa y que está presente en las funciones de Inicializar, Combinar y Mejorar del esquema general metaheurístico parametrizado.
Con respecto a la función mejorar, en la GPU pasa a tener la estructura que se plantea en el algoritmo 5.4, muy diferente a la descrita en el algoritmo 4.6 del capítulo anterior en sistemas multicore.

Algoritmo 5.4 Pseudocódigo función *Mejorar(S,Devices,ParamIni, ThreadsblockMoveImp,ThreadsblockRot,ThreadsblockFit, ThreadsblockIncImp)*

```
1: K = 0
2: Host_To_GPU(S,Stmp)
3: while K < IMEIni do
4:    Mover_mejora<conformaciones/ThreadsblockMoveImp,ThreadsblockMoveImp>
      (Stmp,ParamIni)
5:    Rotar<conformaciones/ThreadsblockRot,ThreadsblockRot>
      (Stmp,ParamIni)
6:    Calcular_fitness<conformaciones/ThreadsblockFit,ThreadsblockFit>
      (Stmp,ParamIni)
7:    Incluir_si_Mejora<conformaciones/ThreadsblockIncImp,ThreadsblockIncImp>
      (Stmp,ParamIni)
8:    K = K + 1
9: end while
10: GPU_To_Host(S,Stmp)
```

Combinar(Ssel,Scom,Devices,ParamCom,Threads1Com,Threads2Com, ThreadsblockFit)

La función no sufre grandes modificaciones con respecto al esquema del algoritmo 4.8, aunque se convierte en una función híbrida en la que parte se realiza usando paralelismo en la CPU y otra parte, la que conlleva el cálculo de Lennard-Jones después de generar las nuevas conformaciones, se realiza en la GPU. Por tanto, dos parámetro de paralelismo indican los hilos de la primera parte de la ejecución en la CPU, paralelizando la combinación de conformaciones en varios puntos candidatos, y otro parámetro se referirá al número de hilos por cada bloque en la ejecución en la GPU. El coste computacional de la función Combinar, según los diferentes costes explicados en la función de inicializar es $O(K_{MCPU} * (m * o/(th1 * th2) + m * p/(th1 * th2) + m * q/(th1 * th2)) + w * u * l * K_{GPU})$El esquema algorítmico se presenta en el algoritmo 5.5.

Mejorar(Scom,Devices,ParamImp,ThreadsblockMoveImp,ThreadsblockRot, ThreadsblockIncImp,ThreadsblockFit)

La función del esquema del algoritmo 5.1, como sucede en la versión paralela en CPU, es similar a la descrita en la fase de Inicializar, usando el mismo número de parámetros metaheurísticos para especificar los hilos por bloque en la GPU que usan sus procedimientos internos; tanto la generación de nuevas posiciones, como el cálculo del potencial de Lennard-Jones. Lo único que cambian son los parámetros metaheurísticos *IMEImp* y *PEMImp*, que son específicos para esta función, y el conjunto de datos de entrada.

Algoritmo 5.5 Pseudocódigo función *Combinar(Ssel,Scom,Devices,ParamCom, Threads1Com,Threads2Com,ThreadsblockFit)*

```
1: omp_set_nested(1)
2: omp_set_num_threads(Threads1Com)
3: #pragma omp parallel for
4: for i = 1 to m do
5:    omp_set_num_threads(Threads2Com)
6:    #pragma omp parallel for
7:    for j = 1 to o do
8:       Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
9:    end for
10:   #pragma omp parallel for
11:   for k = 1 to p do
12:      Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
13:   end for
14:   #pragma omp parallel for
15:   for l = 1 to r do
16:      Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
17:   end for
18: end for
19: Host_To_GPU(Scom,Stmp)
20: Calcular_fitness<conformaciones/ThreadsblockFit,ThreadsblockFit>
    (Stmp,ParamIni)
21: GPU_To_Host(Scom,Stmp)
```

5.2. Técnicas de optimización en GPU para aplicar en la metaheurística parametrizada

El proceso de introducir cálculos en la GPU es una necesidad que se ha ido confirmando a la hora de ir realizando los sucesivos estudios experimentales de los capítulos 3 y 4. Dado que se disponen de GPUs de NVIDIA se va a utilizar CUDA para afrontar este proceso y explotar al máximo el paralelismo. Como en otros lenguajes de programación, se deben tener en cuenta todas las herramientas que dispone el lenguaje para realizar este proceso, y extraer el máximo rendimiento posible. Además, conocer la arquitectura subyacente y saber que trabajamos con plataformas NVIDIA, nos hace colocar valores para nuestros parámetros en rangos en que tradicionalmente se obtienen los mejores resultados.
El tamaño del WARP o unidad mínima de planificación es muy importante actualmente en CUDA, y define el número de hilos que se ejecutan a la vez realmente dentro de un multiprocesador en la GPU. Esta característica es fundamental a la hora de plantear y abordar optimizaciones en CUDA.
El número de hilos por bloque con los que se lanzarán las ejecuciones en la GPU serán 64, 128, 256 y 512, realizando pruebas con cada uno de estos tamaños para así obtener el tamaño óptimo de bloque para cada uno de los procedimientos.

5.2.1. Uso de memoria compartida e instrucciones intrínsecas

En la implementación de este código, más concretamente en el cálculo del potencial de Lennard-Jones, se han incluido dos optimizaciones importantes que ayudan a mejorar su rendimiento. Estas son: Uso de la memoria compartida o *shared memory* y el conjunto de instrucciones intrínsecas ___*shfl* para ayudar a aumentar el rendimiento de las sumas por reducción.

- El uso de la memoria compartida aprovecha la posibilidad de reutilización de datos por parte de los hilos del mismo bloque. En nuestro caso concreto, vamos cargando partes del compuesto de mayor tamaño, realizada por cada hilo.
  Con la parte cargada en la memoria compartida, según la implementación que se ha llevado a cabo, cada hilo del bloque computa la parte del cálculo del potencial correspondiente a la parte cargada. Cuando todos los hilos han finalizado, se procede a cargar otro trozo y proseguir, utilizando para coordinar todos los hilos los mecanismos de sincronización que nos brinda CUDA, tales como *cudaDeviceSynchronize()*.
  En la computación de alto rendimiento, el uso de memoria compartida reduce el uso de accesos a memoria y por tanto aumenta el rendimiento de nuestro código, pero a la hora de hacer uso de este mecanismo de optimización hay que estudiar el porcentaje de reutilización de datos, ya que podría tener un coste que no superaría el beneficio obtenido.

- La posibilidad de usar el conjunto de instrucciones ___*shfl* está disponible en dispositivos con computabilidad 3.X o superior, y su uso puede aportar una mejora del tiempo de ejecución de nuestras aplicaciones, al permitir el intercambio de

valores entre hilos del mismo WARP sin usar la memoria compartida.
El uso de este conjunto de instrucciones que nos ofrece CUDA nos va a permitir reducir y obtener la suma de forma rápida y eficiente a nivel de WARP, dado que nuestro código configura de forma eficiente el conjunto de hilos que se lanzan al mismo tiempo en la GPU, identificando cada conformación con una unidad de planificación o WARP.
Dada la aplicación concreta de estas instrucciones a una computabilidad determinada, se ha realizado un trabajo de adaptación de los *kernels* a la computabilidad del dispositivo en que se está ejecutando, derivando la reducción que se realiza a nivel de WARP hacia *___shfl*, o utilizando instrucciones compatibles con una computabilidad inferior.

5.2.2. Maximizar la ocupación

Maximizar el nivel de paralelismo optimizando al máximo el nivel de ocupación de los multiprocesadores es de vital importancia para el rendimiento, ya que, aunque exista la posibilidad de hacer coincidir un bloque por multiprocesador, no significa que obtengamos el máximo rendimiento, porque pueden existir sincronizaciones entre ellos que los paralicen hasta que otros hagan su trabajo.
En un multiprocesador solo puede haber un número determinado de bloques activos, por lo que debe de haber una cantidad suficiente de bloques por multiprocesador, para que el planificador pueda ejecutar otro cuando haya factores de sincronización que hagan a unos bloques detener su ejecución.
Equilibrar la memoria compartida es otro factor a tener en cuenta a la hora de tener el máximo número de bloques activos al mismo tiempo en el multiprocesador, ya que se divide equitativamente entre todos estos bloques hasta que se agota.

5.3. Resultados computacionales en multicore en CPU y GPU

En esta sección vamos a estudiar y ejecutar en multicore en CPU y GPU las combinaciones de parámetros metaheurísticos de las tablas 3.2 y 4.2. Esto lo realizaremos con los compuestos *2bsm_rec.mol2* y *2bsm_lig.mol2*. Con los resultados que arrojen las ejecuciones de los compuestos *2bsm_rec.mol2* y *2bsm_lig.mol2* vamos a evaluar tanto el ahorro obtenido en coste computacional como el valor de *fitness*. A continuación, con estos mismos compuestos, se va a realizar una comparativa completa con la GPU desde la ejecución en secuencial hasta multicore en CPU. Seguidamente se va a estudiar individualmente el ahorro computacional en cada una de las funciones del esquema metaheurístico parametrizado, buscando los valores óptimos de hilos por bloque tanto en procedimientos que utilizan memoria compartida a nivel de bloque, caso del cálculo del *fitness*, como en los que no se usa.
A continuación se va a proceder a realizar el estudio experimental en la GPU con los compuestos *PROT_rec.mol2* y *ligando_lig.mol2*, cuyo tamaño molecular del receptor es muy elevado. Este estudio consistirá en ejecutar las combinaciones de parámetros metaheurísticos 1, 2, 5 y 6 de la tabla 3.2. Las ejecuciones se realizarán en la Tesla K20c

del nodo Saturno y la Tesla K40c del ordenador Hertz. Para finalizar se expondrán las conclusiones derivadas de los estudios experimentales realizados.

5.3.1. Análisis del coste computacional y cálculo de *fitness* de los compuestos *2bsm_rec.mol2* y *2bsm_lig.mol2*

En esta sección, primeramente se va a estudiar el ahorro en coste computacional de usar técnicas masivamente paralelas en GPU con respecto al modelo paralelo multicore en CPU y, seguidamente, se va a estudiar en profundidad la evolución del coste de las funciones Inicializar y Combinar, variando el número de hilos por bloque de sus procedimientos internos y, así, obteniendo el número de hilos óptimo de forma experimental. A continuación se compararán los valores de *fitness* que se derivan de las ejecuciones de la tabla 3.2 y, que compararemos con otras versiones de cómputo anteriores.

5.3.1.1. Estudio experimental del modelo paralelo en OpenMP y en GPU

Para realizar este estudio, se va a variar los parámetros que nos indican el número de hilos por bloque asignados a cada uno de los procedimientos internos que se ejecutan en la GPU, de las diferentes funciones del esquema metaheurístico, de forma que podamos establecer los valores que supongan el mayor ahorro de tiempo.
La función Inicializar tiene dos parámetros relacionados con el uso de la GPU. El primero de estos dos valores asigna el número de hilos por bloque en la generación de elementos y el segundo al cálculo del *fitness*. Referente a la función Combinar, se usa un parámetro para especificar el número de hilos en la CPU para realizar la combinación de los elementos, y otro para especificar el número de hilos por bloque en la GPU para el cálculo del *fitness* de las conformaciones combinadas. Por tanto en esta última función nos centraremos en el segundo parámetro. Como en el capítulo anterior, usaremos combinaciones de los parámetros metaheurísticos de la tabla 3.2, concretamente las combinaciones 1, 4, 5 y 8.

Como podemos apreciar en las figuras 5.1 el número de hilos por bloque de la función que genera las posiciones de las diferentes conformaciones no supone un ahorro o disminución del tiempo de ejecución en la función Inicializar.

Cuando pasamos a evaluar el número de hilos por bloque de la función que calcula el *fitness*, en nuestro caso el potencial de Lennard-Jones, tenemos que realizar un estudio más personalizado, dado que usamos memoria compartida para su cálculo. Por ello debemos ajustar su reserva para maximizar el número de bloques activos por multiprocesador y, a la vez, conseguir el mejor rendimiento posible. Su distribución se realiza dividiendo el tamaño total de la memoria compartida en cada multiprocesador entre la cantidad usada por cada bloque, obteniendo el total de bloques que permanecerán activos por cada uno de los multiprocesadores.
Como se observa en las figuras 5.2 y 5.3, el tamaño óptimo de bloque es de 128 hilos, aunque tamaños de 256 o 512 hilos por bloque también obtienen resultados muy próximos al óptimo.

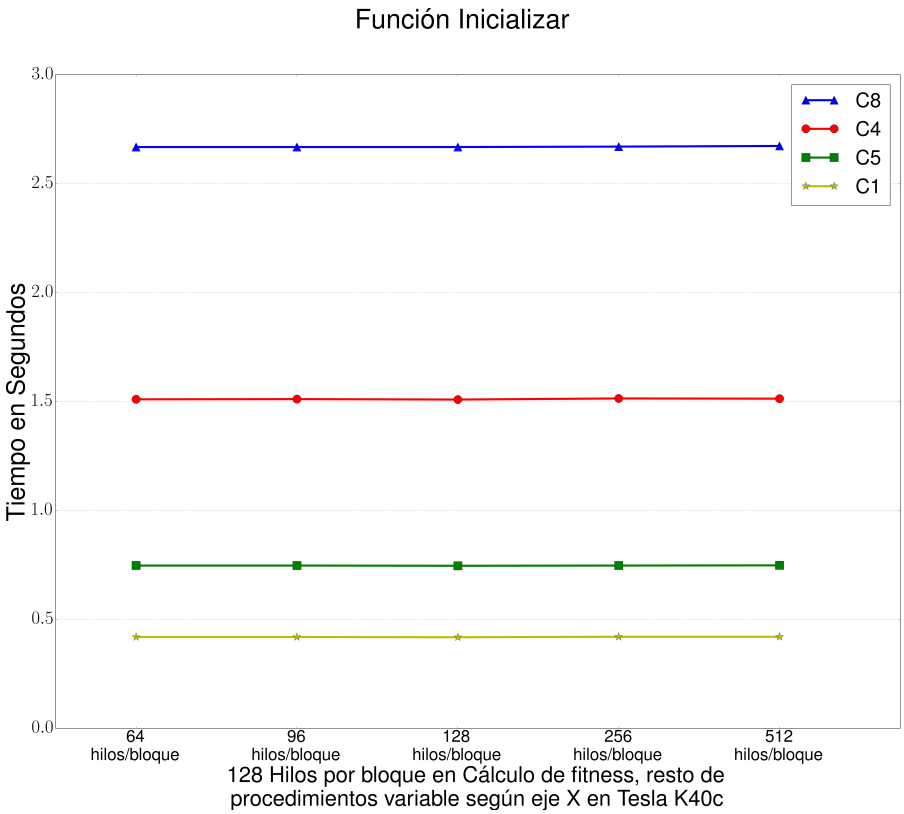

Figura 5.1: Tiempo de ejecución de la función Inicializar en el ordenador Hertz, con GPU Tesla K40c de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), con número de hilos por bloque en el cálculo de fitness fijo en 128 hilos por bloque, en el resto de procedimientos variable según eje X.

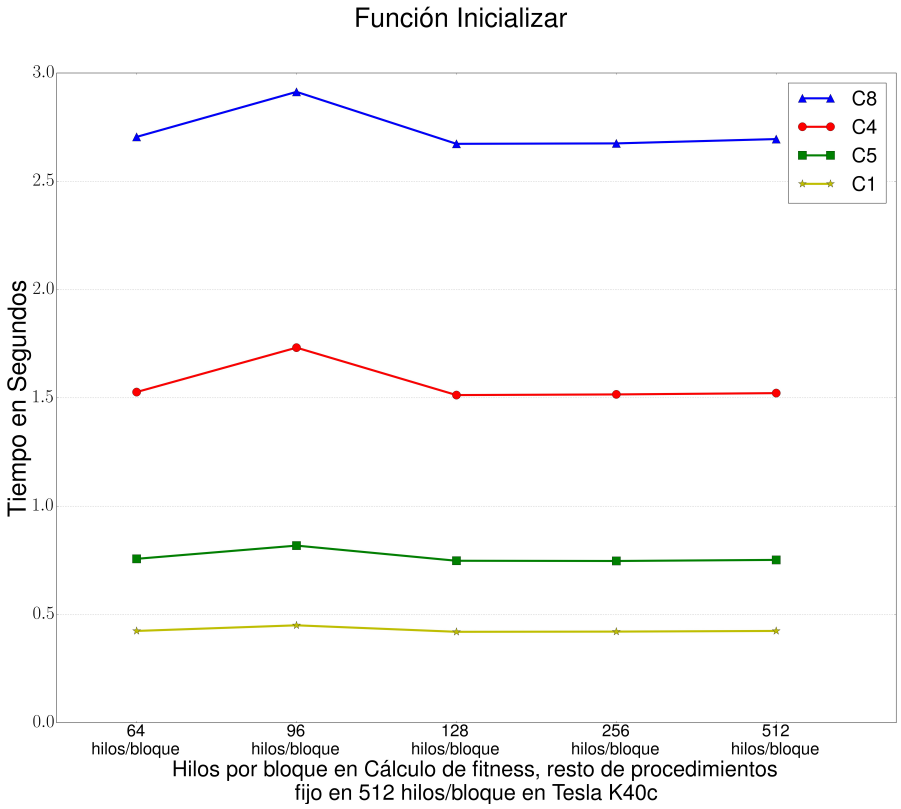

Figura 5.2: Tiempo de ejecución de la función Inicializar en el ordenador Hertz, con GPU Tesla K40c de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), con número de hilos por bloque en el cálculo de fitness variable según eje X, y en el resto de procedimientos fijo en 512 hilos por bloque.

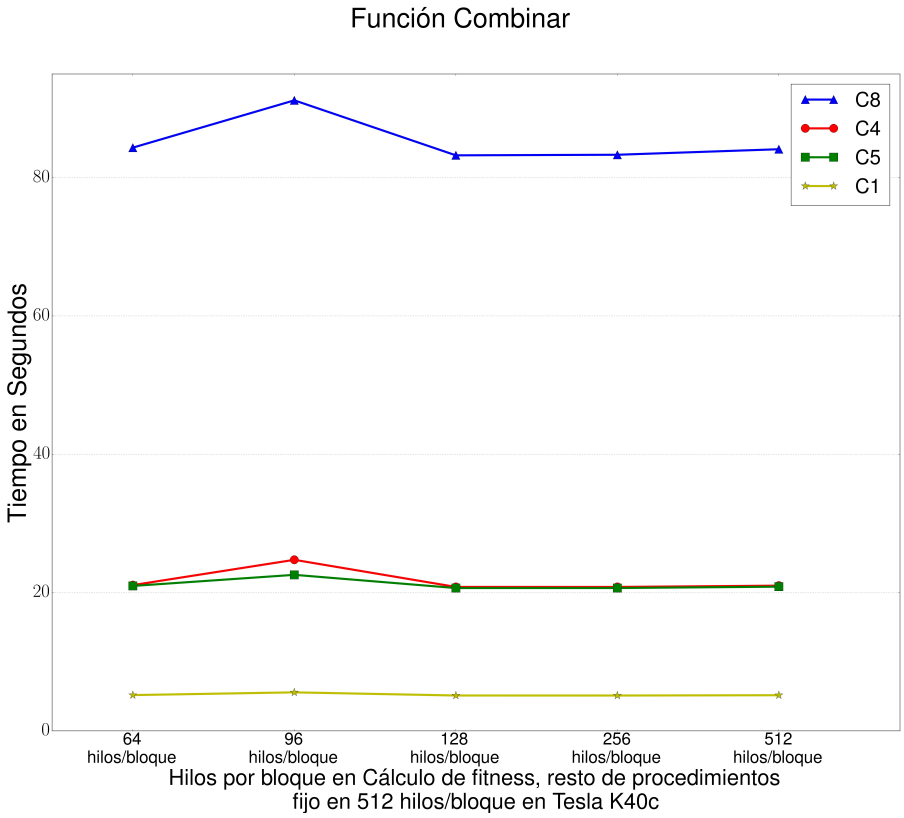

Figura 5.3: Tiempo de ejecución de la función Combinar en el ordenador Hertz, con GPU Tesla K40c de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 8 (C8), con número de hilos por bloque en el cálculo de fitness variable según eje X, y en el resto de procedimientos fijo en 512 hilos por bloque.

5.3.1.2. Modelo paralelo en OpenMP vs Modelo paralelo en OpenMP y GPU

Como se muestra en las tablas 5.1 y 5.2, disminuimos de manera sustancial el tiempo del esquema paralelo multicore en CPU. Por tanto el uso de la GPU en el problema de acoplamiento molecular es fundamental para abordar la mejora de su rendimiento. Además podemos ver que la tarjeta Tesla K40c se comporta cada vez mejor, con respecto a la Tesla K20c cuando vamos teniendo mayor número de conformaciones para trabajar en las diferentes funciones, caso de las combinaciones 2 y 6.

COMBINACIONES	Versión Paralela Multicore	Versión Paralela Multicore+GPU	Speed-up
1	47.86	5.54	8.6
2	28.48	3.31	8.6
3	39.66	4.72	8.2
4	188.25	23.24	8.1
5	187.21	24.01	7.8
6	107.49	13.43	8.0
7	132.65	16.79	7.9
8	755.46	93.26	8.1

Tabla 5.1: Comparativa del tiempo de ejecución entre la versión paralela en Multicore y versión paralela Multicore y en GPU de las combinaciones de la tabla 3.2. Ejecuciones realizadas en el ordenador Hertz con Tesla K40c con la combinación de hilos por bloque 512-128. Tiempo medido en segundos.

COMBINACIONES	Versión Paralela Multicore	Versión Paralela Multicore+GPU	Speed-up
1	47.86	5.68	8.4
2	28.48	4.45	6.4
5	187.21	29.65	6.3
6	107.49	15.89	6.7

Tabla 5.2: Comparativa del tiempo de ejecución entre la versión paralela en Multicore y versión paralela Multicore y en GPU de las combinaciones 1, 2, 5 y 6 de la tabla 3.2. Ejecuciones realizadas en el nodo Saturno con Tesla K20c con la combinación de hilos por bloque 512-128. Tiempo medido en segundos.

5.3.1.3. Análisis del *fitness*

El uso de la GPU proporciona mejoras del rendimiento muy importantes a nuestra aplicación, además de mantener los valores de *fitness* con respecto a versiones anteriores en multicore. En la tabla 5.3 se ofrecen los resultados en el ordenador Hertz con el dispositivo Tesla K40c.

	COMBINACIONES							
	1	2	3	4	5	6	7	8
Fitness	-120.14	-126.54	-180.78	-198.65	-205.84	-186.48	-207.17	-173.57
Tiempo (seg)	47.32	35.81	28.12	110.46	87.41	65.23	101.48	186.74

Tabla 5.3: Valores de *fitness* y tiempo de ejecución de cada una de las combinaciones de la tabla 4.2. en el ordenador Hertz con la Tesla K40c

5.3.2. Análisis del coste computacional de los compuestos *PROT_rec.mol2* y *ligando_lig.mol2*

En el cuerpo humano existen compuestos con un número de átomos muy elevado. Este es el caso de un receptor particular, denominado *PROT_rec.mol2*, que está formado por un total de 105648 átomos, mientras que el estudiado hasta este momento y que actuaba como compuesto receptor, dispone de una estructura formada por 3264 átomos. Por tanto, para aplicar nuestro esquema metaheurístico parametrizado al mundo real, debemos enfrentarnos a este tipo de compuestos. El aumento del coste computacional sería considerable, ya que estamos hablado de un compuesto más de treinta veces superior al que estamos analizando hasta ahora, por lo que las estructuras en memoria pueden sobrepasar el límite del dispositivo, a la vez que el tiempo de ejecución aumentará en todas las funciones del esquema.

5.3.2.1. Estudio experimental del modelo paralelo en OpenMP y en GPU

Vamos a proceder a ejecutar las combinaciones 1, 2 y 3 de la tabla 3.2 con este compuesto una sola pasada del esquema parametrizado paralelo en multicore y en GPU, para analizar qué funciones debemos tratar de elevar a un nivel superior de cómputo para acometer este tipo de cálculos.

FUNCIONES	COMBINACION 1	COMBINACION 2	COMBINACION 3
Inicializar	150.256	150.589	539.46
Seleccionar	0.005	0.004	0.003
Combinar	2310.651	1247.653	1285.08
Mejorar	0	12484.79	292.84
Incluir	0.461	0.279	0.003

Tabla 5.4: Tiempo de ejecución en segundos de cada una de las funciones, para las distintas combinaciones consideradas. Ejecuciones realizadas en el ordenador Hertz en la Tesla K40c.

Como observamos en los resultados de la tabla 5.4, los principales costes se sitúan en las funciones del esquema metaheurístico que integran el cálculo del *fitness*, como Inicializar, Combinar y Mejorar, que es el más costoso de todo el proceso. Por ello es necesario buscar técnicas que aumenten el nivel de paralelismo para enfrentarnos a

compuestos de este tipo, dado que con tal cantidad de átomos, una sola GPU tarda varias horas en realizar una sola pasada al esquema.

5.4. Auto-optimización en GPU

La investigación en el desarrollo de aplicaciones con sistemas de auto-optimización para GPU, también ha sido explorado en librerías de álgebra lineal como MAGMA [37], en trabajar con transformadas de fourier en CUDA de la manera más óptima [38] o, resolver sistemas tridiagonales de una forma más eficiente [13]. Para que una aplicación que se ejecuta en la GPU obtenga el mejor rendimiento se debe trabajar en la optimización de sus parámetros, ya que el conseguir mejor o peor rendimiento depende totalmente de la configuración que se aplique al dispositivo donde se realice la ejecución. Tanto su computabilidad como el número de hilos por bloque al configurar la ejecución de los diferentes *kernels*, van a determinar mayormente el rendimiento que va a proporcionar el dispositivo.
Como se ha procedido en el capítulo anterior, se va a realizar un conjunto de pruebas significativas en la primera ejecución real. Con estas pruebas vamos a encontrar el tamaño óptimo de bloque en los diferentes *kernels* para el dispositivo donde se va a realizar la ejecución y lo guardamos en un fichero de configuración. Dado que es un sistema híbrido entre multicore y GPU, los parámetros paralelos para trabajar en multicore se optimizarán como se ha descrito en la versión para multicore en la sección 4.4. En las siguientes ejecuciones en esta misma máquina con ese mismo dispositivo, se extraen los parámetros de ese fichero previamente generado, y se aplican a la ejecución. Las fases que comprenden este proceso son:

- Generar un conjunto de entrenamiento.
- Para cada función que utilice multicore
  - Establecer el número de hilos al máximo.
  - Ir progresivamente disminuyendo el valor del número de hilos y ejecutar con cada valor la función hasta llegar a la unidad, almacenando el mejor resultado.
- Para cada *kernel* ejecutarlo con valores de hilos por bloque entre 64 y 512, en intervalos de 64 hilos.
  - Almacenar el valor óptimo de hilos por bloque para cada uno de los *kernels*.
- Generar el fichero de configuración para ese dispositivo, con el número de hilos óptimo en procedimientos multicore, y el número de hilos por bloque en cada *kernel*.

El tiempo dedicado a esta fase de optimización hará que se maximice el nivel de ocupación de los multiprocesadores, con la mayor cantidad de bloques activos posibles. El resultado a modo de ejemplo de una de estas ejecuciones en la GPU, se encuentra en el Anexo B.

Como se muestra en el Anexo, se siguen las fases que se han descrito anteriormente, primeramente se genera el conjunto de entrenamiento y, a continuación, se procede a calcular el valor óptimo de hilos para las funciones de generación de elementos iniciales dentro de la función de Inicializar y en la función de Seleccionar, ambas con un nivel de paralelismo. Seguidamente se realiza este mismo estudio para la función de Combinar con dos niveles. La obtención del número óptimo de hilos se obtiene de la misma forma que la descrita en la sección 4.4.
A continuación, como se observa en el anexo, se obtiene el número óptimo de hilos por bloque para cada uno de los siete *kernels* de la aplicación, variando el número de hilos por bloque entre 64 y 512 hilos en tramos de 64 hilos. Finalmente se muestra el total de tiempo empleado junto con el número de hilos óptimo en las funciones con multicore, y el valor de hilos por bloque óptimo en cada uno de los *kernels*.
En la tabla 5.5 se muestran las tres configuraciones de parámetros paralelos, que se van a evaluar en la fase de auto-optimización. Con estas ejecuciones se va a demostrar que el gasto en tiempo de ejecución en esta fase, compensará en el resto de ejecuciones con un mejor rendimiento. En las dos primeras columnas se aportan dos configuraciones manuales de parámetros paralelos y, en la ultima, los parámetros óptimos obtenidos en la fase de auto-optimización. El criterio de elección de las configuraciones manuales de las columnas 1 y 2, son las que podría colocar un usuario con conocimiento del entorno de las GPUs de NVIDIA, y en los valores por donde podrían encontrarse los valores óptimos. Esto significa que no sabría donde se concentran los costes computacionales de la aplicación por lo que iría al azar en su asignación.

Parámetros Paralelos	Configuración Manual 1	Configuración Manual 2	Configuración Óptima
Threads1Ini	12	12	11
Threads1Sel	12	12	8
ThreadsblockFit	512	256	64
ThreadsblockMoveIni	256	512	448
ThreadsblockRot	128	256	512
ThreadsblockQuat	256	128	384
ThreadsblockRandom	128	512	128
Threads1Com	6	12	11
Threads2Com	2	1	1
ThreadsblockMoveImp	512	256	128
ThreadsblockIncImp	256	128	128
Threads1Inc	12	12	8

Tabla 5.5: Combinaciones de parámetros paralelos para el sistema híbrido multicore + GPU

La tabla 5.6 muestra el SPEED-UP obtenido con respecto a las configuraciones manuales al usar la óptima según la fase de auto-optimización. El tiempo dedicado para la fase de auto-optimización es 105.77 segundos. La ganancia obtenida es menor, aunque ejecutando 10 veces la aplicación con diferentes compuestos en la misma máquina se habrá compensando el gasto de optimización. Esto nos lleva a la conclusión, que

el usuario final de la aplicación debería mantener la auto-optimización activada para lograr un rendimiento mejor de la misma a corto plazo.

Combinaciones	Configuración Manual 1	Configuración Manual 2	Configuración Óptima	SPEED-UP
4	45.41	42.14	39.75	1.1
8	164.99	162.25	155.65	1.1

Tabla 5.6: Tiempo de ejecución de las combinaciones 4 y 8 de la tabla 3.2 en el nodo Jupiter con los valores de los parámetros paralelos de la tabla 5.5. SPEED-UP entre la configuración óptima y las dos configuraciones manuales consideradas. Tiempo medido en segundos.

5.5. Conclusiones

Después de realizar el estudio sobre la aplicación de la GPU al esquema basado en memoria compartida, extraemos una serie de conclusiones:

- Aplicando técnicas masivamente paralelas obtenemos una mejora superior a ocho veces con respecto al uso de memoria compartida, por lo que el uso de la GPU está totalmente justificado y es recomendable para el acoplamiento molecular. Las proporciones de mejora figuran en las tablas 5.1 y 5.2.

- Elevando los valores de los parámetros metaheurísticos, como los expuestos en la tabla 4.2, logramos resultados de *fitness* muy cercanos al óptimo en tiempos más razonables que los derivados de las ejecuciones en multicore exclusivamente. Los resultados en general han mejorado en todas las combinaciones.

- Dedicar tiempo a la auto-optimización de los parámetros de paralelismo en la GPU puede mejorar el rendimiento conforme vaya creciendo su uso. Ya la mejora puede ser muy sutil en compuestos pequeños, pero conforme se vaya haciendo más intensiva el uso de la GPU, los porcentajes de mejora irán subiendo. En las figuras 5.3 y 5.4 del estudio experimental observamos esa tendencia de mejora, en elegir un tamaño de bloque en el cálculo del *fitness* de 96 hilos por bloque a otro de 128 hilos. Como vemos, el de 96 hilos por bloque es penalizado.

- Para usar nuestro esquema metaheurístico con compuestos biológicos presentes en la mayoría de las células, que pueden tener una enorme cantidad de átomos, es recomendable proceder a utilizar nodos que permitan trabajar en multiGPU, ya que con este tipo de compuestos los tiempos de ejecución son muy elevados para obtener los primeros resultados.

Capítulo 6

Técnicas masivamente paralelas aplicadas al esquema metaheurístico híbrido parametrizado en multicore + multiGPU

En este capítulo se usan técnicas basadas en multiGPU en las funciones donde se concentran los mayores costes computacionales y que suponen un cuello de botella cuando usamos compuestos receptores con decenas de miles de átomos. Se van a describir el conjunto de algoritmos que nos van proporcionar la aplicación de estas técnicas, trabajando en un esquema híbrido en multicore y multiGPU. Seguidamente se van a explicar los dos tipos de computación que se van tratar y la forma de gestionar cada uno de ellos. A continuación se va a realizar un estudio experimental con ambos tipos y las conclusiones que se derivan del mismo.
También en este caso se abordarán mecanismos de auto-optimización para cada tipo de computación, junto con un reparto de la carga de trabajo en los dispositivos en función de una fase previa de entrenamiento.

6.1. Tipos de computación en multiGPU y reparto de cargas de trabajo.

En este trabajo se han distinguido dos formas de abordar la computación en multiGPU en un nodo, dependiendo del tipo de dispositivos de que disponga el sistema:

- *Computación homogénea.* Cuando se elige esta forma de cómputo, se explora el nodo y se buscan los diferentes tipos de GPUs con soporte CUDA de que dispone el sistema. Seguidamente se elige para computar un determinado grupo de GPUs similares.

- *Computación heterogénea.* En este caso se explora por completo el nodo, igual que el caso anterior, pero se asume que se van a usar todas las GPUs que soporten CUDA. A la hora de computar se utilizan todos los dispositivos que han sido validados para el cómputo de nuestra aplicación.
  Hoy en día existe una gran diversidad de GPUs, cada una con unas capacidades y rendimientos totalmente distintos. Se hace necesario ir un paso más allá en la configuración de los dispositivos para cómputo de lo que se ha hecho hasta ahora, teniendo en cuenta la diversidad de características que nos podemos encontrar.

La distribución de cargas de computación en función del rendimiento de cada uno de los dispositivos, la debemos tener presente cada vez que usemos estos esquemas de computación.

- En la *computación homogénea*, al ser todos los dispositivos involucrados en la computación de similares características, se divide la carga de trabajo de forma equitativa entre todos los dispositivos.
- En *modo heterogéneo* tenemos dos modos de trabajo. El primero consistiría en detectar los dispositivos CUDA aptos para cómputo y repartir el trabajo de forma homogénea entre todos ellos. Un segundo modo de trabajo configura una misma ejecución en cada uno de los dispositivos. A partir de ese momento y, dependiendo el porcentaje de tiempo que haya consumido cada uno de los dispositivos, se asigna la carga de trabajo de forma que los dispositivos más rápidos asuman una carga mayor del cómputo.

6.2. Esquema híbrido usando sistemas de memoria compartida en CPU y multiGPU

El esquema de asignación de trabajo a las diferentes GPUs va a ser similar al algoritmo de un solo nivel de paralelismo, donde cada hilo trabajará con una parte de los datos y enviará a una GPU diferente. Este esquema queda definido en el algoritmo 6.1. El esquema es aplicable tanto al modo de computación homogénea como al modo heterogéneo.

Algoritmo 6.1 Esquema de trabajo con multicore + multiGPU

```
1: omp_set_num_threads(numero_GPUs)
2: #pragma omp parallel for
3: for i = 1 to numero_GPUs do
4:       Tratamiento a realizar por cada GPU
5: end for
```

El esquema del algoritmo 6.2 describe la estructura general de trabajo con multiGPU. Los nombres de variables con nombre similar a las descritas en el algoritmo 4.1 mantienen las mismas funciones, al igual que el nombre de los parámetros paralelos en GPU. Como en capítulos anteriores, para facilitar la lectura y comprensión de los algoritmos, recordamos la definición una serie de variables:

- n al número total de conformaciones o individuos.
- m al número de puntos candidatos o puntos de acoplamiento del receptor.
- r al número de átomos de la proteína o receptor.
- l al número de átomos del ligando.
- o al número de elementos mejores en la función Combinar.
- p al número de elementos peores en la función Combinar.
- q al número de elementos mejores-peores en la función Combinar.
- d al número de GPUs.

Algoritmo 6.2 Esquema parametrizado paralelo híbrido multicore y multiGPU

```
1: Inicializar_multiGPU(S,Devices,ParamIni,Threads1Ini,ThreadsblockMoveIni,
   ThreadsblockRot,ThreadsblockQuat,ThreadsblockRandom,
   ThreadsblockMoveImp,ThreadsblockIncImp,ThreadsblockFit)
2: while no Fin(S) do
3:    Seleccionar(S,Ssel,ParamSel,Threads1Sel)
4:    Combinar_multiGPU(Ssel,Scom,Devices,ParamCom,Threads1Com,
      Threads2Com,ThreadsblockFit)
5:    Mejorar_multiGPU(Scom,Devices,ParamImp,ThreadsblockMoveImp,
      ThreadsblockRot,ThreadsblockIncImp,ThreadsblockFit)
6:    Incluir(Scom,S,ParamInc,Threads1Inc)
7: end while
```

Dados los estudios realizados con una GPU, detectamos que el cuello de botella se puede producir cuando realizamos el cálculo del *fitness* con compuestos receptores de gran tamaño como el *PROT_rec.mol2* con decenas de miles de átomos en su estructura. Por tanto, se va a abordar el tratamiento con multiGPU en este cálculo, debido a que para el resto de *kernels* su tiempo de ejecución no es relevante para el rendimiento de este problema concreto, como se viene observando en capítulos anteriores.
Los algoritmos que no incorporan el cálculo del *fitness* no sufren ningún cambio en su estructura. A continuación se van a ir describiendo las funciones con nuevas estructuras algorítmicas.

Inicializar_multiGPU(S,Devices,ParamIni,Threads1Ini, ThreadsblockMoveIni,ThreadsblockRot,ThreadsblockQuat, ThreadsblockRandom,ThreadsblockMoveImp,ThreadsblockIncImp, ThreadsblockFit)

La función Inicializar_multiGPU tiene los mismos parámetros que en su versión para GPU, excepto que el componente denominado *Devices* almacena todas las características de cada dispositivo con el que se va a computar, en vez de solo un dispositivo como

ocurría en el capítulo anterior. Previamente se ha establecido el modo de computación en la GPU y se han extraído y almacenado las características de cada dispositivo en el vector *Devices*.

Algoritmo 6.3 Pseudocódigo función *Inicializar_multiGPU(S,Devices,ParamIni, Threads1Ini,ThreadsblockMoveIni,ThreadsblockRot, ThreadsblockQuat,ThreadsblockRandom,ThreadsblockMoveImp, ThreadsblockIncImp,ThreadsblockFit)*

```
1: Host_To_GPU(S,Stmp)
2: Init_Rot<conformaciones/ThreadsblockQuat,ThreadsblockQuat>
   (Stmp,ParamIni)
3: Init_Random<conformaciones/ThreadsblockRandom,ThreadsblockRandom>
   (Stmp,ParamIni)
4: Mover<conformaciones/ThreadsblockMoveIni,ThreadsblockMoveIni>
   (Stmp,ParamIni)
5: Rotar<conformaciones/ThreadsblockRot,ThreadsblockRot>
   (Stmp,ParamIni)
6: GPU_To_Host(S,Stmp)
7: if PEMIni >0 then
8:    Mejorar_multiGPU(S,ParamIni,ThreadsblockMoveImp,ThreadsblockRot,
      ThreadsblockIncImp,ThreadsblockFit)
9: end if
10: omp_set_num_threads(d)
11: #pragma omp parallel for
12: for i = 1 to d do
13:    Seleccionar_dispositivo(Devices[i].id)
14:    Host_To_GPU(S,Stmp)
15:    Calcular_fitness<Devices[i].conformaciones/Devices[i].Threadsblock,
       Devices[i].Threadsblock>
       (Stmp+Devices[i].desplazamiento,ParamIni)
16:    GPU_To_Host(S,Stmp)
17: end for
```

Con se observa en el algoritmo 6.3, se aplica el esquema definido en el algoritmo 6.1 en el cálculo del *fitness*. Los parámetros del número de conformaciones, tamaños de bloque y parte de los datos de que se encarga cada uno de los dispositivos, son definidos previamente a la llamada de la función. Dependiendo del porcentaje de carga que asuma cada dispositivo, se le asignarán para su cómputo un cierto número de conformaciones. El número de hilos por bloque también está asociado al dispositivo para posteriores tareas de optimización, y mejorar el rendimiento de los diferentes *kernels*.
Las funciones que no quedan distribuidas bajo el esquema algoritmo 6.1 mantienen la misma estructura que en la versión con una única GPU. También mantiene la misma estructura interna el cálculo de *fitness* que el algoritmo 5.3, dado que el dispositivo que se hará cargo del mismo y los parámetros de lanzamiento del *kernel* son independientes de su estructura.

Algoritmo 6.4 Pseudocódigo función *Mejorar_multiGPU(S,Devices,ParamIni, ThreadsblockMoveImp,ThreadsblockRot,ThreadsblockFit, ThreadsblockIncImp)*

```
1: K = 0
2: Host_To_GPU(S,Stmp)
3: while K < IMEIni do
4:    Mover_mejora<conformaciones/ThreadsblockMoveImp,
      ThreadsblockMoveImp>(Stmp,ParamIni)
5:    Rotar<conformaciones/ThreadsblockRot,ThreadsblockRot>
      (Stmp,ParamIni)
6:    GPU_To_Host(S,Stmp)
7:    omp_set_num_threads(d)
8:    #pragma omp parallel for
9:    for i = 1 to d do
10:      Seleccionar_dispositivo(Devices[i].id)
11:      Host_To_GPU(S,Stmp)
12:      Calcular_fitness<Devices[i].conformaciones/Devices[i].Threadsblock,
         Devices[i].Threadsblock>(Stmp+Devices[i].desplazamiento,ParamIni)
13:      GPU_To_Host(S,Stmp)
14:   end for
15:   Incluir_si_Mejora<conformaciones/ThreadsblockIncImp,
      ThreadsblockIncImp>(Stmp,ParamIni)
16:   K = K + 1
17: end while
```

La función Mejorar_multiGPU, definida en el algoritmo 6.4, mantiene básicamente la estructura del algoritmo 5.4 variando exclusivamente la parte del cálculo del *fitness*, como sucede en la función Inicializar. La configuración del cálculo sigue su misma estructura, enviando a cada uno de los dispositivos una parte de la computación.

Combinar_multiGPU(Ssel,Scom,Devices,ParamCom,Threads1Com, Threads2Com,ThreadsblockFit)

La función combinar en multiGPU que figura en el algoritmo 6.5 y que forma parte del esquema del algoritmo 6.2, hereda la estructura del algoritmo 5.5 variando en la ejecución del cálculo de *fitness*. Se mantiene un ejecución híbrida, pero en este caso la combinación es multicore + multiGPU. El cálculo de *fitness* queda definido con la misma estructura que en los algoritmos 6.3 y 6.4.

Algoritmo 6.5 Pseudocódigo función *Combinar_multiGPU(Ssel,Scom,Devices, ParamCom,Threads1Com,Threads2Com,ThreadsblockFit)*

```
1: omp_set_nested(1)
2: omp_set_num_threads(Threads1Com)
3: #pragma omp parallel for
4: for i = 1 to m do
5:    omp_set_num_threads(Threads2Com)
6:    #pragma omp parallel for
7:    for j = 1 to o do
8:       Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
9:    end for
10:   #pragma omp parallel for
11:   for k = 1 to p do
12:      Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
13:   end for
14:   #pragma omp parallel for
15:   for l = 1 to r do
16:      Combinar_dos_elementos(Ssel,Scom,elemento_1,elemento_2,ParamCom)
17:   end for
18: end for
19: omp_set_num_threads(d)
20: #pragma omp parallel for
21: for i = 1 to d do
22:    Seleccionar_dispositivo(Devices[i].id)
23:    Host_To_GPU(Scom,Stmp)
24:    Calcular_fitness<Devices[i].conformaciones/Devices[i].Threadsblock,
       Devices[i].Threadsblock>
       (Stmp+Devices[i].desplazamiento,ParamIni)
25:    GPU_To_Host(Scom,Stmp)
26: end for
```

Mejorar_multiGPU(Scom,Devices,ParamImp,ThreadsblockMoveImp, ThreadsblockRotImp,ThreadsblockIncIni,ThreadsblockFit)

La función del esquema del algoritmo 6.2 equivaldría a la descrita en el algoritmo 6.4 dentro de la función Inicializar_multiGPU, y utiliza los mismos parámetros paralelos en su entrada, variando los parámetros metaheurísticos. La diferencia radica en el conjunto de datos de entrada, dado que son fruto de la combinación de los elementos y no procedentes de la generación inicial de los mismos y, también en los parámetros metaheurísticos IMEImp y PEMImp, que son específicos para la función.

6.3. Resultados computacionales en multicore en CPU y multiGPU

En esta sección se va a seguir la línea de ejecuciones del capítulo 5 para comprobar la mejora en tiempo de ejecución con el uso de multiGPU. Vamos a comenzar el estudio con los compuestos *2bsm_rec.mol2* y *2bsm_lig.mol2*, realizando una comparativa entre multicore + multiGPU, multicore + GPU y, exclusivamente, multicore en CPU, de cada una de las combinaciones de la tabla 3.2. Se evaluará tanto el valor de *fitness* como el tiempo de ejecución, viendo el impacto de aumentar el número de GPUs en el cálculo en ciertas combinaciones de la tabla 3.2. Este estudio se realizará para los modos homogéneo y heterogéneo en sus dos modos de trabajo en el nodo Jupiter. Seguidamente se va a trabajar con los compuestos *PROT_rec.mol2* y *ligando_lig.mol2*, donde el receptor tiene un tamaño molecular muy elevado, con decenas de miles de átomos. Vamos a valorar este compuesto con multiGPU y a realizar una comparativa con los valores obtenidos en la tabla 5.4 con una GPU de gran capacidad de cómputo como es la Tesla K40c. Al terminar el estudio se expondrán las conclusiones extraídas del mismo. Los tiempos de las tablas que se van a mostrar, corresponden a la ejecución de una sola vez de cada una de las funciones del esquema.

6.3.1. Análisis del coste computacional y cálculo de fitness de los compuestos *2bsm_rec.mol2* y *2bsm_lig.mol2* en multiGPU

Primeramente se va a distinguir los dos modos de cómputo en multiGPU. En el modo homogéneo se va a analizar el tamaño óptimo del bloque de las funciones básicas de Inicializar y Combinar, del mismo modo que en el capítulo anterior. Vamos a realizar este estudio con los dos tipos de GPUs del nodo Jupiter. En el modo heterogéneo vamos a coger los resultados del modo homogéneo donde se extrae el número óptimo de hilos por bloque en ambos tipos de GPUs. Seguidamente analizaremos la evolución de los valores de *fitness* extraídos en multiGPU, con la versión en GPU.

6.3.1.1. Estudio experimental del modelo paralelo en OpenMP y en multiGPU

En el modo homogéneo se van a estudiar el número de hilos óptimos de las funciones Inicializar y Combinar. En la función Inicializar se va a distinguir por un lado el número de hilos por bloque en la generación de elementos y por otro los relativos al cálculo del *fitness*. Con respecto a la función Combinar, se utiliza multicore en CPU para combinar las conformaciones, y otro valor para realizar el cálculo del *fitness* en GPU después de combinarlos. Esto implica que en la función de Inicializar variaremos dos parámetros y en Combinar solo uno.
En el modo heterogéneo se va a seleccionar el número de GPUs para obtener el rendimiento óptimo de nuestra aplicación, ya que cuando no tenemos suficientes conformaciones para satisfacer a todos los dispositivos, el rendimiento puede caer. En este caso vamos a buscar el número óptimo para ejecutar los compuestos *2bsm_rec.mol2* y *2bsm_lig.mol2*, según las combinaciones 1, 4, 5 y 6 de la tabla 3.2. En el modo heterogéneo se distinguirán los dos modos de trabajo, optimizando la carga o repartiéndola de forma homogénea entre todos los dispositivos.

Modo homogéneo

En las figuras 6.1 y 6.2 se observa la diferencia en rendimiento en las funciones Inicializar y Combinar si variamos el número de hilos por bloque en las GPUs GeForce GTX 590, de las que se dispone de cuatro dispositivos, y en las figuras 6.3 y 6.4 con las GPUs Tesla C2075, de las que se dispone de dos dispositivos.
En estas cuatro figuras, variamos el número de hilos por bloque de todos los *kernels* excepto del que calcula el *fitness* que se mantiene fijo en 128 hilos por bloque. La elección de un valor u otro de los estudiados para calcular la parte de generación inicial de posiciones, y otros *kernels* de cómputo interno, no afecta el rendimiento de la aplicación. Las líneas de las gráficas son prácticamente horizontales en la función Combinar, aunque en la fase de Inicializar sí existe una pequeña variación con mejora si usamos un tamaño de 256 hilos por bloque en las GPUs GeForce GTX 590 y Tesla C2075, aunque en estas últimas no apreciable.
En la figuras 6.5 y 6.6 se aporta el estudio en las funciones de Inicializar y Combinar referente exclusivamente al cálculo del *fitness* en GPUs GeForce GTX 590. En este caso si colocamos valores diferentes de hilos por bloque en esta función, sí que afecta al rendimiento de la aplicación, situando el número óptimo de hilos por bloque en 256 con los cuatro dispositivos GeForce GTX 590 cuando existe mayor carga de cómputo en la fase de Inicializar. En la fase de Combinar los diferentes tamaños de hilos por bloque tienen mayor igualdad, aunque cuando mayor es la carga computacional, el valor más se aproxima a 128 hilos por bloque.
En la figuras 6.7 y 6.8 se muestra el estudio de las funciones de Inicializar y Combinar para el cálculo del *fitness* en las GPUs Tesla C2075. A diferencia de los modelos GTX 590, el tamaño óptimo de bloque para este cálculo en estos dos dispositivos es de 256 hilos por bloque.
En estos estudios presentados no se detecta que el tamaño de bloque tenga especial impacto con el rendimiento de la aplicación, excepto en la fase de Inicializar con los

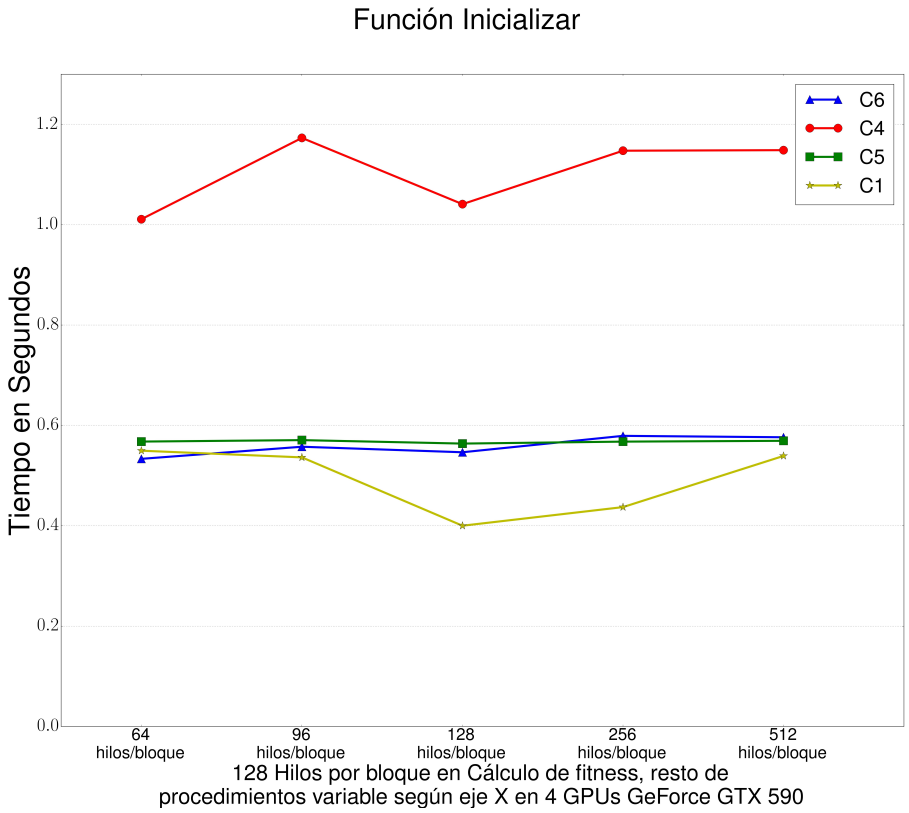

Figura 6.1: Tiempo de ejecución de la función Inicializar en el nodo Jupiter, con 4 GeForce GTX 590 de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6), con número de hilos por bloque en el cálculo de fitness fijo en 128 hilos por bloque, en el resto de procedimientos variable según eje X.

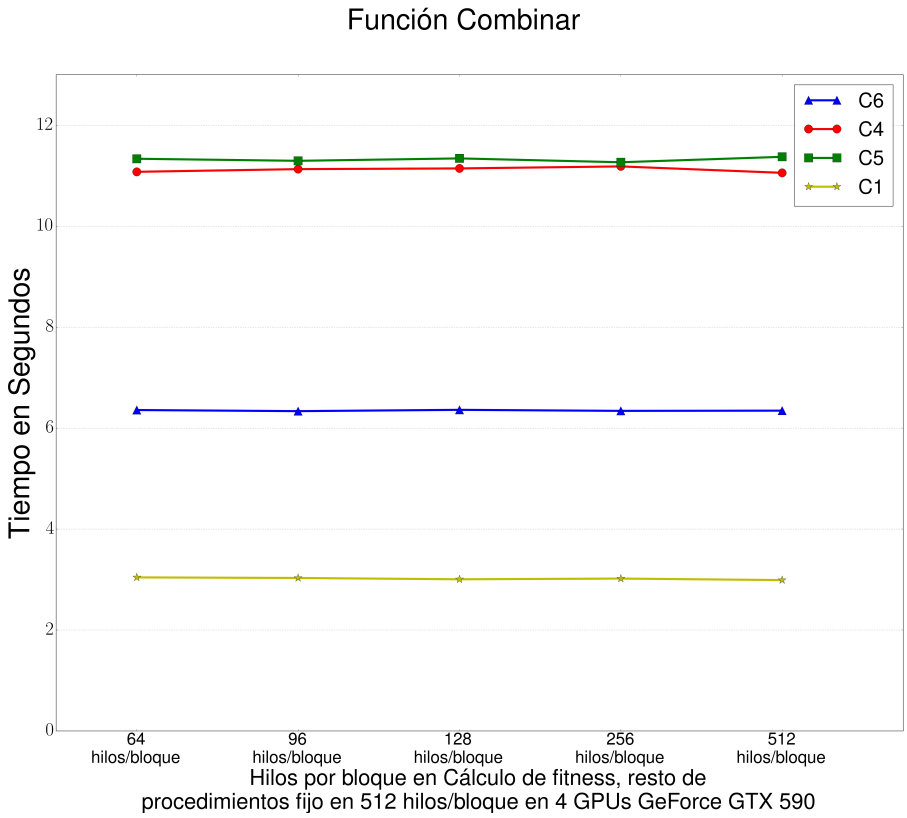

Figura 6.2: Tiempo de ejecución de la función Combinar en el nodo Jupiter, con 4 GPUs GeForce GTX 590 de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6), con número de hilos por bloque en el cálculo de fitness fijo en 128 hilos por bloque, en el resto de procedimientos variable según eje X.

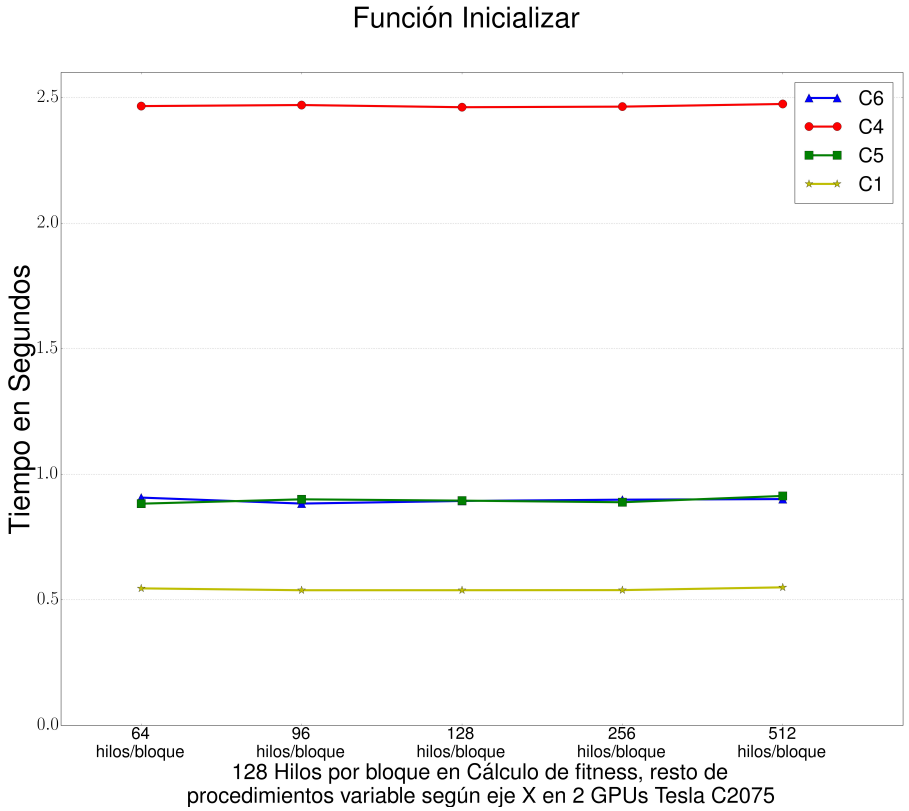

Figura 6.3: Tiempo de ejecución de la función Inicializar en el nodo Jupiter, con 2 GPUs Tesla C2075 de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6), con número de hilos por bloque en el cálculo de fitness fijo en 128 hilos por bloque, en el resto de procedimientos variable según eje X.

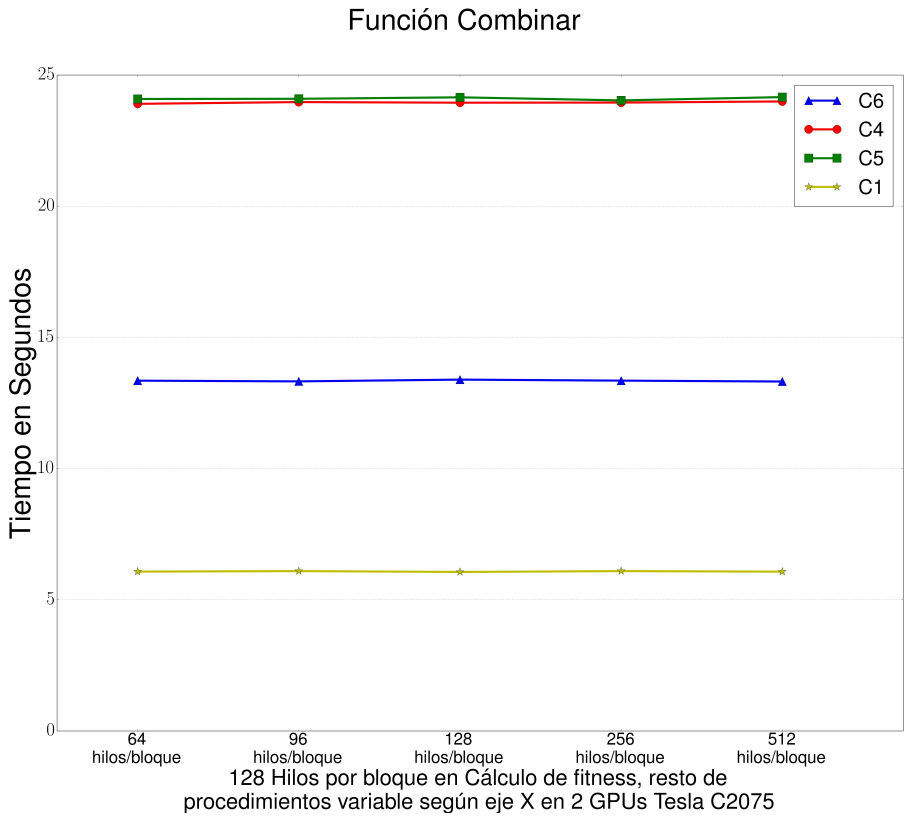

Figura 6.4: Tiempo de ejecución de la función Combinar en el nodo Jupiter, con 2 GPUs Tesla C2075 de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6), con número de hilos por bloque en el cálculo de fitness fijo en 128 hilos por bloque, en el resto de procedimientos variable según eje X.

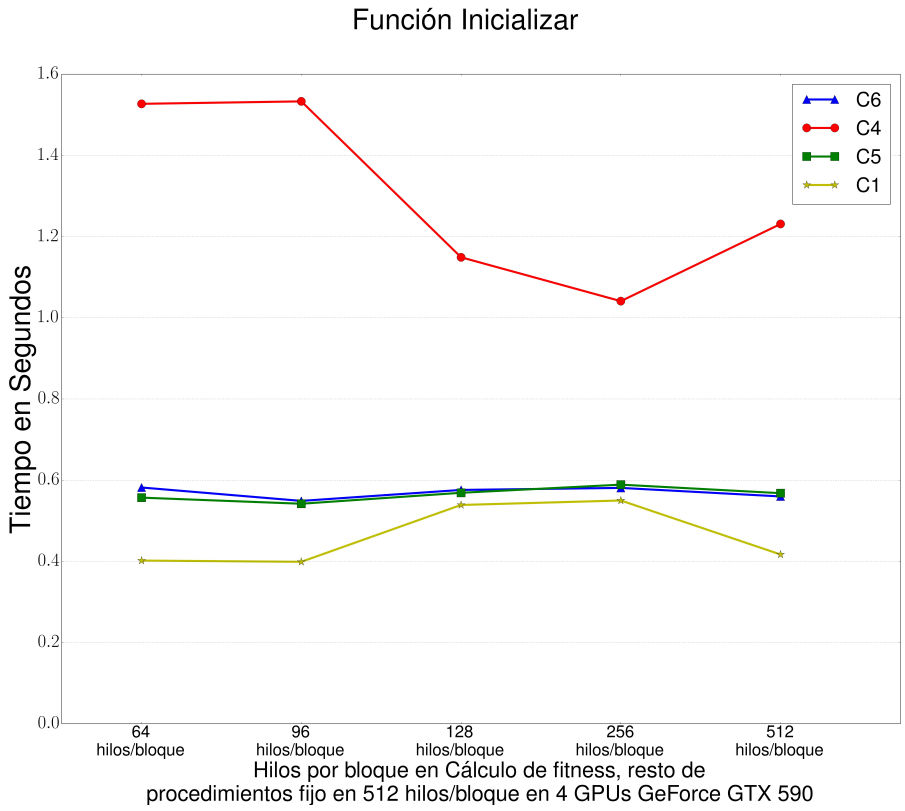

Figura 6.5: Tiempo de ejecución de la función Inicializar en el nodo Jupiter, con 4 GeForce GTX 590 de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6), con número de hilos por bloque en el cálculo de fitness variable según eje X, y en el resto de procedimientos fijo en 512 hilos por bloque.

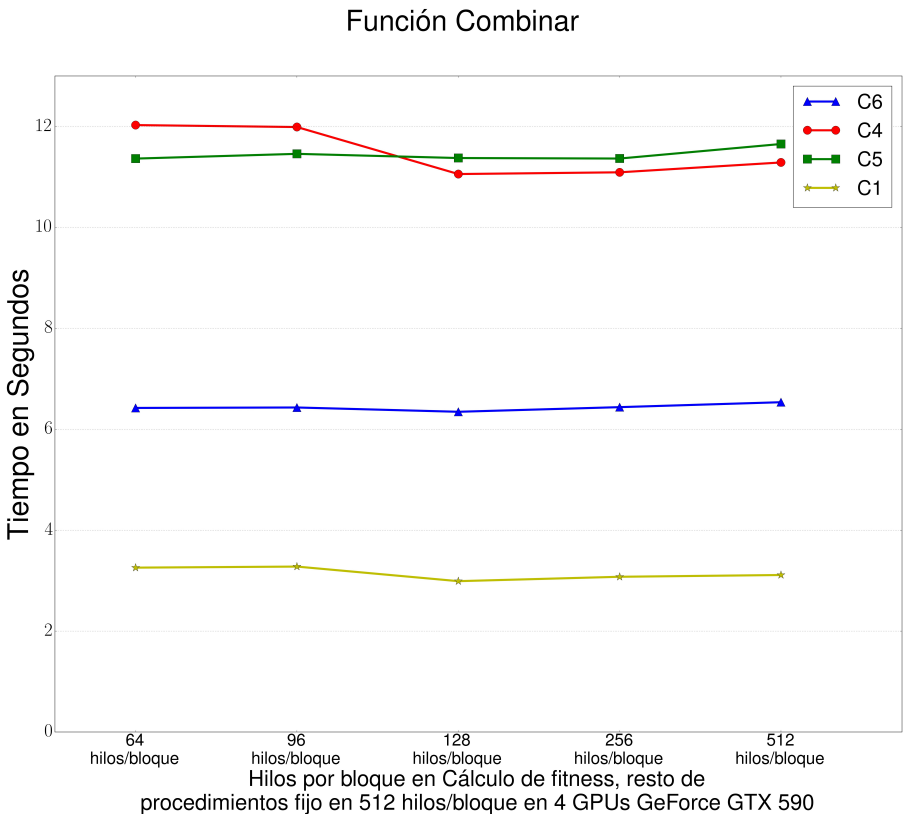

Figura 6.6: Tiempo de ejecución de la función Combinar en el nodo Jupiter, con 4 GPUs GeForce GTX 590 de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6), con número de hilos por bloque en el cálculo de fitness variable según eje X, y en el resto de procedimientos fijo en 512 hilos por bloque.

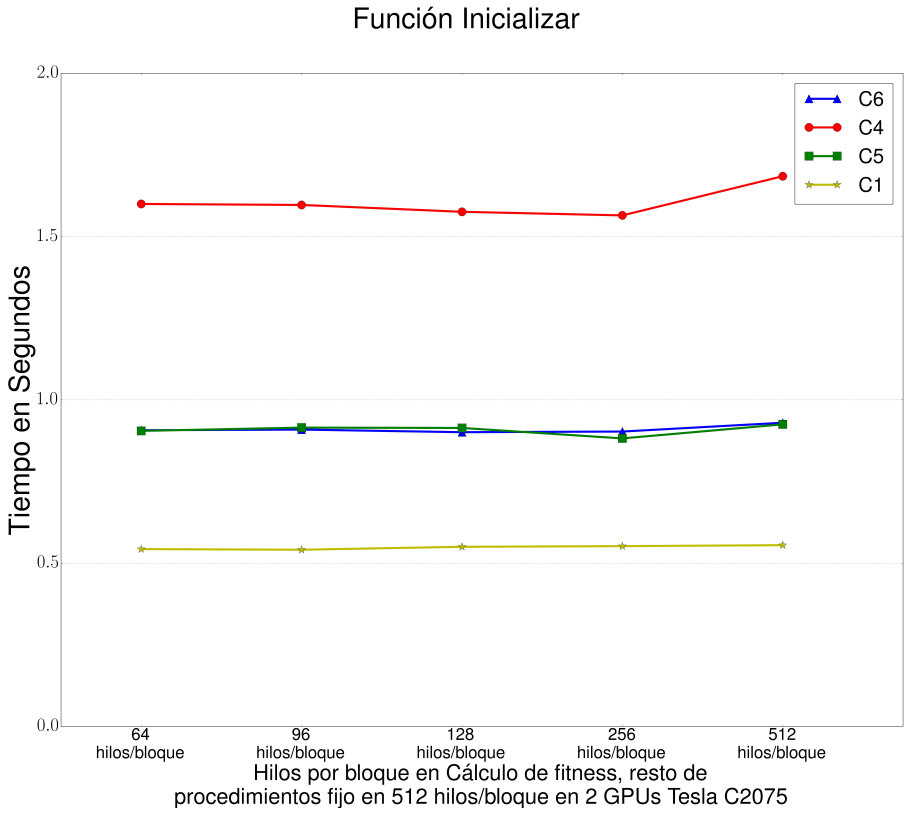

Figura 6.7: Tiempo de ejecución de la función Inicializar en el nodo Jupiter, con 2 GPUs Tesla C2075 de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6), con número de hilos por bloque en el cálculo de fitness variable según eje X, y en el resto de procedimientos fijo en 512 hilos por bloque.

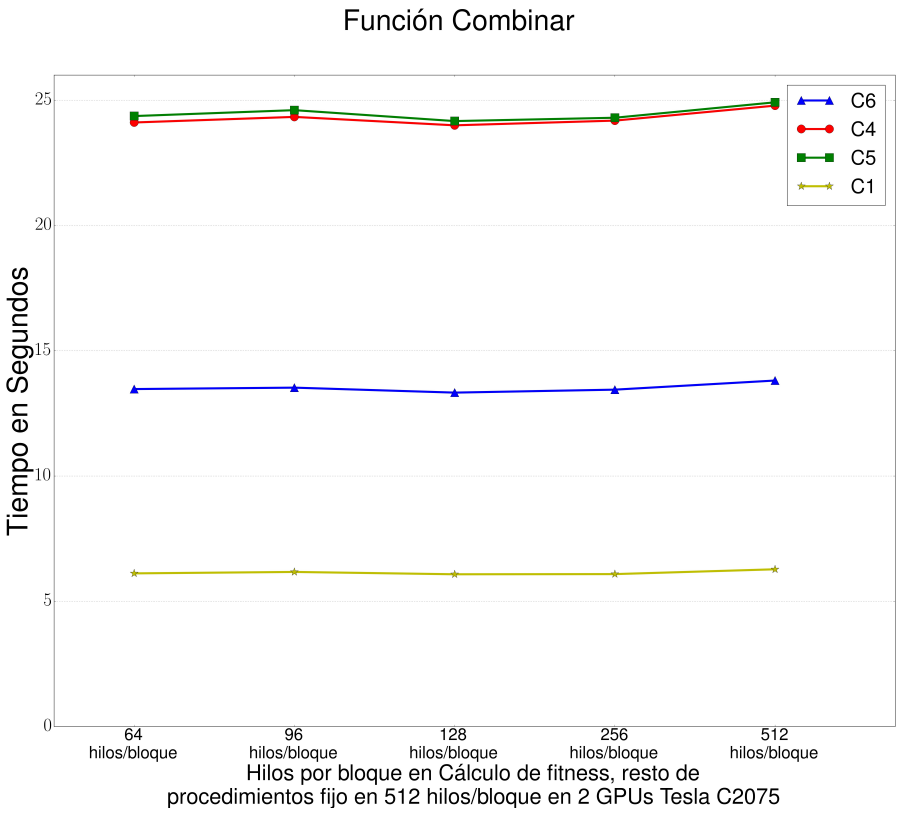

Figura 6.8: Tiempo de ejecución de la función Combinar en el nodo Jupiter, con 2 GPUs Tesla C2075 de las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6), con número de hilos por bloque en el cálculo de fitness variable según eje X, y en el resto de procedimientos fijo en 512 hilos por bloque.

dispositivos GeForce GTX 590. Esta anomalía se percibe en la figura 6.3, con un peor rendimiento con valores distintos a 256 hilos por bloque en el cálculo de *fitness* con la combinación 4, siendo en ella donde se realiza una mejora en la fase de Inicializar con un 20 % de conformaciones. Este número es muy pequeño para dividir entre los cuatro dispositivos, y puede causar este resultado. El resto de combinaciones obtienen su mejor rendimiento con valores de 96 hilos por bloque.

Modo heterogéneo

En este modo se va a heredar el tamaño óptimo de hilos por bloque obtenido en el modo homogéneo con los dispositivos GeForce GTX 590 y Tesla C2075. En las figuras 6.9 y 6.10 se observa la evolución del rendimiento cuando se varía el número de GPUs en las funciones Inicializar y Combinar con reparto de carga proporcional a cada una de los dispositivos según sus características.
Como se aprecia en la figura 6.9, el utilizar para computar los cuatro dispositivos más rápidos, es decir, las GeForce GTX 590 es la mejor opción para afrontar el cómputo de este compuesto en concreto. Es importante hacer constar que este número puede variar totalmente cuando procesamos otro compuesto con diferente tamaño molecular, haciendo necesaria una fase de auto-optimización para elegir el tipo de dispositivo del sistema con el que computar, si queremos obtener el mejor rendimiento.
En la figura 6.10, que representa a la función Combinar, aunque también se obtiene buen resultado usando exclusivamente las cuatro GeForce GTX 590, utilizar las seis GPUs disponibles en el nodo ofrece mejores prestaciones. La explicación radica en que un mayor reparto de conformaciones, la respuesta es más rápida, aunque las GPUs Tesla C2075 sean más lentas.
En este punto el reparto de carga cobra una gran importancia, dado que al llevar menor parte de computación las GPUs más lentas los tiempos se equiparan, ofreciendo un mayor rendimiento de forma global. El reparto se realiza antes de la fase de Inicializar, ejecutando un conjunto de entrenamiento en cada una de los dispositivos disponibles. Cuando han finalizado y tenemos el tiempo de ejecución en cada dispositivo, se reparte el porcentaje de carga en función del tiempo invertido.

6.3.1.2. Modelo paralelo en OpenMP vs GPU vs multiGPU

En este punto vamos a comparar las tres versiones paralelas y ver la mejor opción para ejecutar la aplicación con este compuesto, mostrando dos tablas comparativas. En la tabla 6.1 se muestran las versiones ejecutada en multiGPU en modo homogéneo con las 4 GPUs GeForce GTX 590 con su número de hilos por bloque óptimo y en versión heterogénea usando las 6 GPUs disponibles en el sistema. Ambos modos se van a comparar con los modelos paralelos en OpenMP puro y OpenMP + GPU.
Podemos sacar algunas conclusiones si observamos la tabla. La primera es en relación a que se obtiene igual o superior rendimiento en la versión heterogénea en las combinaciones donde el número de elementos a trabajar en la fase de Combinar es más elevado, caso especial de las combinaciones 4 y 8. También destacar que en combinaciones donde el cómputo es menor y se usan dispositivos más lentos, como es nuestro caso, la distribución de carga no surte el efecto deseado y tiende a ralentizarse un poco

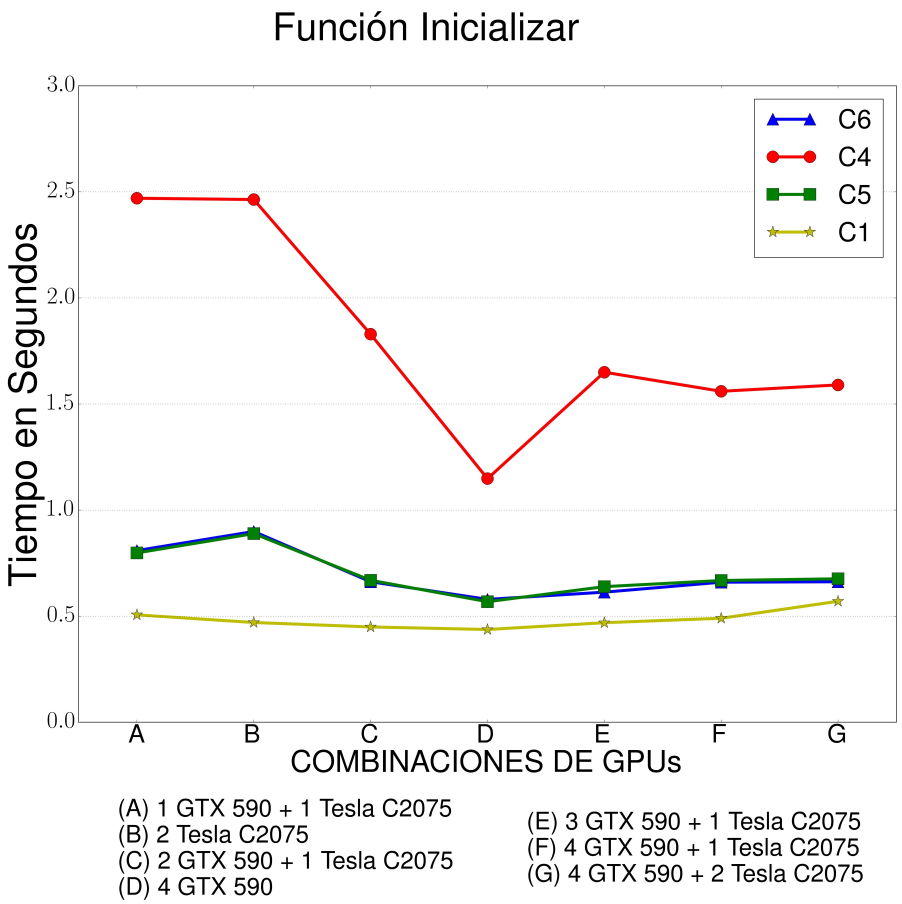

Figura 6.9: Tiempo de ejecución de la función Inicializar en el nodo Jupiter con todas las GPU disponibles, 4 GPUs GeForce GTX 590 y 2 GPUs Tesla C2075, con las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6)

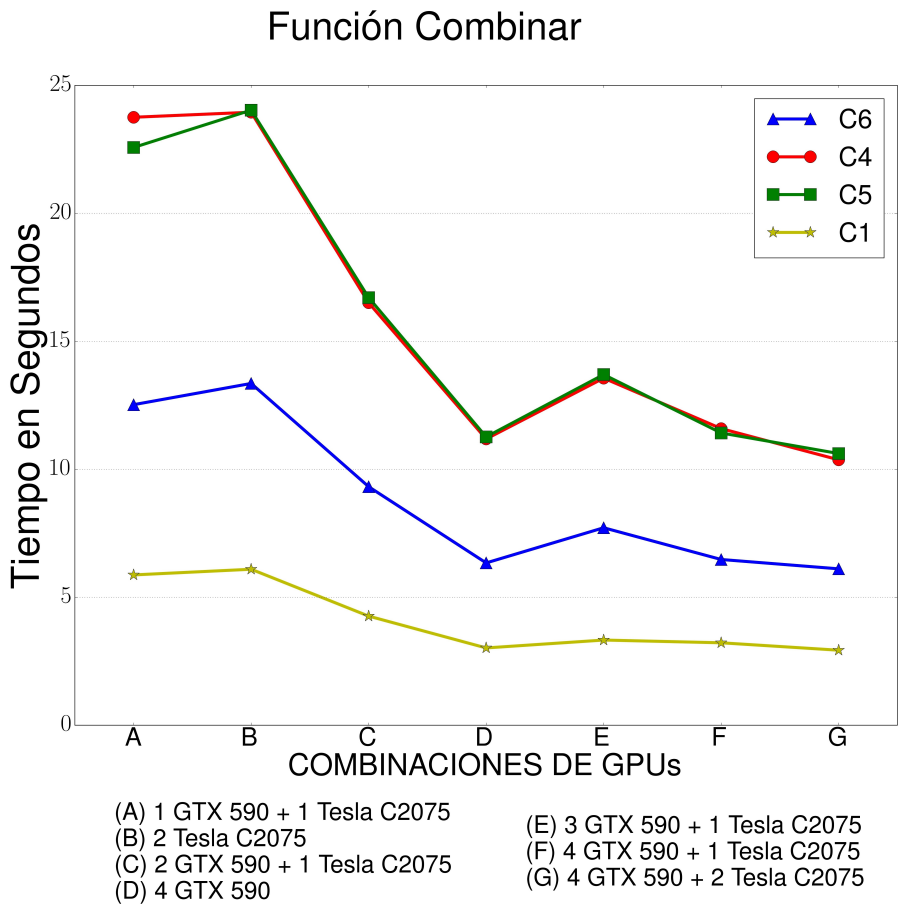

Figura 6.10: Tiempo de ejecución de la función Combinar en el nodo Jupiter con todas las GPU disponibles, 4 GPUs GeForce GTX 590 y 2 GPUs Tesla C2075, con las Combinaciones 1 (C1), 4 (C4), 5 (C5) y 6 (C6)

COMBINACIONES	Versión Paralela Multicore	Versión Paralela Multicore + GPU	Versión HOMOGÉNEA Paralela Multicore + multiGPU	Versión HETEROGÉNEA Paralela Multicore + multiGPU	SPEED - UP
1	47.86	5.54	3.86	**3.46**	12.3
2	28.48	3.31	**2.53**	2.66	11.2
3	39.66	4.72	**5.09**	6.22	7.8
4	188.25	23.24	13.60	**11.85**	13.8
5	187.21	24.01	11.83	**10.88**	15.8
6	107.49	13.43	7.17	**7.11**	14.9
7	132.65	16.79	**12.88**	12.95	10.2
8	755.46	93.26	48.87	**42.54**	15.4

Tabla 6.1: Comparativa del tiempo de ejecución entre la versión paralela en Multicore, versión paralela Multicore y GPU, versión paralela Multicore y multiGPU en modo homogéneo y versión paralela Multicore y multiGPU en modo heterogéneo de las combinaciones de la tabla 3.2. El mejor tiempo de cada combinación es marcado en negrita y el SPEED-UP de cada fila es entre la versión paralela en multicore y el marcado en negrita. Ejecuciones realizadas en el nodo Jupiter con 4 GPUs GTX 590 en la versión homogénea, y 4 GPUs GTX 590 + 2 Tesla C2075 en la versión heterogénea. Tiempo medido en segundos.

la ejecución en el modo heterogéneo. Este último caso se ve reflejado si observamos las combinaciones 2 y 3.

6.3.1.3. Análisis del *fitness* en multiGPU

El desarrollo en un sistema con multiGPU no supone mejora en el *fitness* de la aplicación, aportando solamente mejoras muy importantes en el proceso de cómputo. Los resultados obtenidos son similares en las ejecuciones de la tabla 5.3. Otro estudio que se va a realizar es ejecutar ambas versiones por un tiempo limitado, observando su resultado final. Se va a proceder a establecer el sistema de alarma. En la figura 6.11 se ofrece una comparativa de como evoluciona el *fitness* conforme va aumentando el tiempo de ejecución hasta que se cumplen determinadas condiciones de parada. En este caso se ha ejecutado en multiGPU en modo heterogéneo con reparto de carga la combinación 2 de la tabla 4.2. También se ha ejecutado la misma combinación de parámetros metaheurísticos en una sola GPU, en este caso una GeForce GTX 590 del nodo Jupiter. Se han ido tomando valores de *fitness* máximos en intervalos de 1 segundo, ya que para este compuesto el cálculo con ambas plataformas es muy rápido, pero podemos ver y analizar la evolución del *fitness*.

Se aprecia de forma clara, que para un mismo instante de tiempo, el modo en multiGPU obtiene un valor de *fitness* mejor. Esto significa que al ser más rápido en este modo, da tiempo a realizar mayores pasadas en el esquema y mejorar el valor de manera más rápida. En esta ejecución concreta el valor de multiGPU en 12 segundos es de -150.497 y el de GPU -112.982.

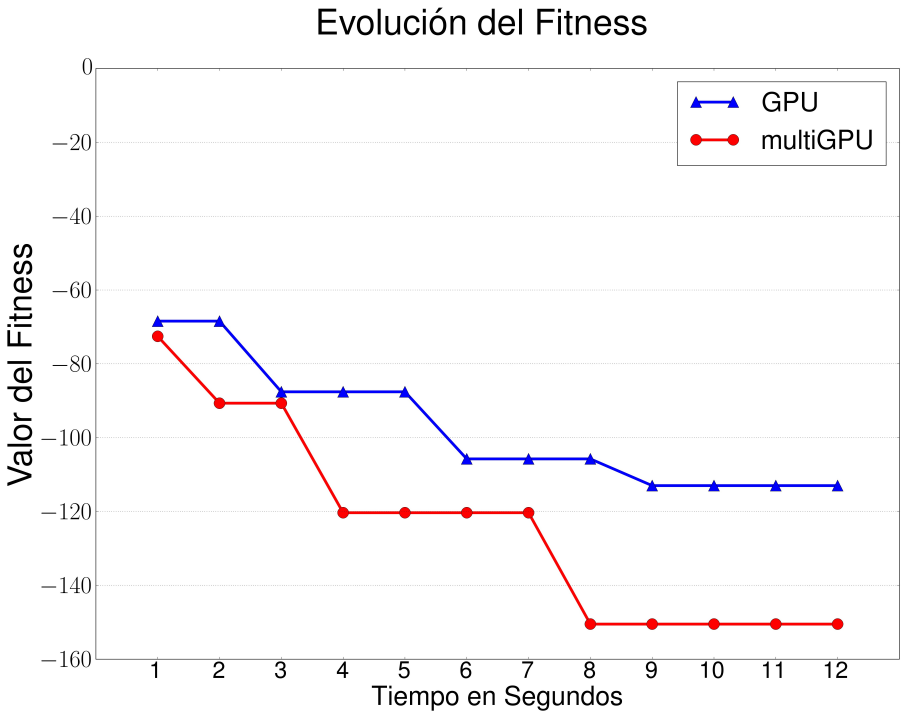

Figura 6.11: Evolución del *fitness* de la combinación 2 de la tabla 4.2 en el nodo Jupiter con multiGPU en modo heterogéneo (4 GPUs GeForce GTX 590 y 2 GPUs Tesla C2075) con reparto de carga, y una GPU modelo GeForce GTX 590. Valor medido en intervalos de 1 segundo durante 12 segundos.

6.3.2. Análisis del coste computacional de los compuestos *PROT_rec.mol2* y *ligando_lig.mol2* en multiGPU

En el capítulo anterior se ha procesado este compuesto cuyo tamaño es de 105648 átomos, muy superior al *2bsm_rec.mol2* con 3264 átomos. Se ejecutó una sola pasada del esquema parametrizado paralelo en multicore y en GPU de las combinaciones 1, 2 y 3 de la tabla 3.2, para ver el tiempo de cada una de las funciones individualmente. Esta ejecución arrojó tiempos de ejecución bastante elevados en las funciones en que se calculaba el *fitness*. Estos resultados llevaron a la necesidad de usar mayores recursos para intentar disminuir este tiempo, desarrollando el sistema en multiGPU.

FUNCIONES	COMBINACIONES					
	Versión Paralela Multicore + GPU			Versión Paralela HETEROGENEA Multicore + multiGPU		
	1	2	3	1	2	3
Inicializar	150.25	150.58	539.46	53.39	53.05	311.64
Seleccionar	0.005	0.004	0.003	0.002	0.002	0.006
Combinar	2310.65	1247.65	1285.08	835.20	480.32	452.55
Mejorar	0	484.79	292.84	0	187.75	0
Incluir	0.461	0.279	0.382	0.002	0.003	0.003
Total	2461.366	1882.653	2117.765	888.42	721.13	928.03

Tabla 6.2: Tiempo de ejecución en segundos de cada una de las funciones, para las distintas combinaciones consideradas en los esquemas híbridos multicore + GPU y multicore + multiGPU con reparto de carga. Ejecuciones realizadas en el nodo Jupiter con las 6 GPUs disponibles, 4 GPUs GeForce GTX 590 y 2 GPUs Tesla C2075. Los resultados de la ejecución multicore + GPU han sido extraídos de la tabla 5.4, de la ejecución en el ordenador Hertz con la GPU Tesla K40c.

Como podemos ver en la tabla 6.2, se produce un ahorro sustancial en el tiempo de ejecución en cada una de las funciones del esquema. Estos datos justifican el uso del esquema híbrido con multiGPU y sus grandes beneficios en rendimiento. En la tabla 6.3 se muestra el total de los tiempos de cada una de las combinaciones de la tabla 6.2 junto con su SPEED-UP. En todos los casos es superior a dos, por lo que su aplicación a este problema concreto queda demostrada experimentalmente. Hay que tener en cuenta la gran diferencia tecnológica entre las GPUs del nodo Jupiter a la del ordenador Hertz, denotando la eficacia de disponer de más dispositivos, aunque sean mucho más lentos, que uno muy rápido y mucho más costoso económicamente.

6.4. Auto-optimización en multiGPU

La importancia que se está dando a la optimización automática de parámetros paralelos en este trabajo prosigue cuando vamos a utilizar varias GPUs, optimizando tanto cada una de las tarjetas individualmente como el global de reparto de cómputo entre ellas. Al comienzo del capítulo se han diferenciado, además de los dos tipos de computación en multiGPU, dos modos de trabajo dentro del tipo heterogéneo, que también se van a tra-

Combinaciones	Versión paralela Multicore CPU + GPU	Versión paralela Multicore CPU + multiGPU	SPEED-UP
1	2461.366	888.42	2.9
2	1882.653	721.13	2.6
3	2117.765	928.03	2.3

Tabla 6.3: Tiempo de ejecución total en segundos y SPEED-UP para las distintas combinaciones consideradas en los esquemas híbridos multicore + GPU y multicore + multiGPU de la tabla 6.3. Ejecuciones realizadas con el esquema multicore + multiGPU en el nodo Jupiter con las 6 GPUs disponibles, 4 GPUs GeForce GTX 590 y 2 GPUs Tesla C2075. Los resultados de la ejecución multicore + GPU han sido extraídos de la tabla 5.4, de la ejecución en el ordenador Hertz con la GPU Tesla K40c.

tar de manera independiente en esta sección. Todas estas tareas de auto-optimización y balanceo de carga se van a ejecutar una sola vez en cada una de las máquinas donde se ejecute la aplicación, y se va a generar un fichero con todos estos parámetros paralelos optimizados, que se guardará junto al ejecutable de la aplicación para evitar el gasto computacional que esta fase de auto-optimización requiere.
Cuando utilizamos la computación en multiGPU, parte de la fase de auto-optimización se encarga de adaptar el cómputo a estos dispositivos, ya que pueden tener características muy distintas y, por tanto, adaptando la carga de cómputo a estas particularidades podemos obtener mejores valores de rendimiento. Una segunda parte de la fase de auto-optimización se va a dedicar a determinar el número óptimo de hilos por bloque de cada dispositivo que interviene en el cómputo. Por tanto vamos a distinguir dos fases:

- *Fase de reparto de cargas.* En los datos de entrada a la aplicación se definen algunos parámetros de la fase de calentamiento o auto-optimización. Cuando elegimos la opción de multiGPU heterogénea con reparto de carga se genera un conjunto de entrenamiento para medir los tiempos de ejecución de los dispositivos cuando deben calcular el *fitness* de todas esas conformaciones. El proceso que se sigue es el siguiente:

  - Generamos el conjunto de entrenamiento.
  - Se ejecuta el cálculo de *fitness* en cada dispositivo para cada conformación del conjunto de entrenamiento y se miden los tiempos.
  - Ordenamos por tiempo y calculamos el porcentaje del tiempo total que corresponde a cada dispositivo.
  - En función de dicho porcentaje se asigna la carga de trabajo a cada dispositivo.

  Los valores que se obtienen en esta fase, son personalizados en cada dispositivo y se guardan todos en la estructura *Devices*. Cada vez que se calcule el *fitness* el número de conformaciones totales se dividirá según los porcentajes establecidos en la fase de auto-optimización.
  También se puede optar por un reparto homogéneo de cargas, y dividir el total de

conformaciones a las que queremos calcular su *fitness* entre todos los dispositivos disponibles.

- *Fase de optimización del número de hilos por bloque.* En cada uno de los dispositivos, se analizan el valor óptimo de hilos por bloque en cada uno de los *kernels*. El proceso es similar al realizado con una sola GPU descrito en la sección 5.4.

En compuestos pequeños como *2bsm_rec.mol2* y *2bsm_lig.mol2* no tenemos suficiente carga computacional para distinguir de forma inequívoca el funcionamiento del reparto de carga en los dispositivos del nodo Jupiter, dado que tienen rendimientos similares en pequeñas ejecuciones. Por tanto vamos a usar el compuesto más grande *PROT_rec.mol2* y *ligando_lig.mol2*, para realizar este experimento.
En la tabla 6.4 se muestra el tiempo de cada una de las fases del esquema con multiGPU en sus versión heterogénea sin reparto de carga y con reparto de carga para las combinaciones 1, 2 y 3 de la tabla 3.2. Como se observa, existe cierta ganancia en tiempo de ejecución, que cuantificamos en la tabla 6.5 mostrando el porcentaje de ahorro.

FUNCIONES	COMBINACIONES					
	Versión HETEROGENEA SIN reparto de carga Multicore + multiGPU			Versión HETEROGENEA CON reparto de carga Multicore + multiGPU		
	1	2	3	1	2	3
Inicializar	56.817	56.87	345.87	53.39	53.05	311.64
Seleccionar	0.002	0.002	0.001	0.002	0.002	0.006
Combinar	889.84	485.09	497.66	835.02	480.32	452.55
Mejorar	0	192.25	175.54	0	187.75	163.84
Incluir	0.002	0.002	0.004	0.002	0.003	0.002
Total	946.66	734.21	1019.07	888.42	721.13	928.03

Tabla 6.4: Tiempo de ejecución en segundos de cada una de las funciones del esquema parametrizado paralelo con las combinaciones 1, 2 y 3 de la tabla 3.2. Las configuraciones son multicore + multiGPU sin reparto de carga y con reparto de carga entre los diferentes dispositivos según sus características. Ejecuciones realizadas en el nodo Jupiter con las 6 GPUs disponibles, 4 GPUs GeForce GTX 590 y 2 GPUs Tesla C2075.

En la tabla 6.5 se muestra el porcentaje de mejora de usar reparto de carga o no. El coste del reparto de carga, dentro de la fase de auto-optimización, es de 7.15 segundos. Esto lleva a la conclusión que es recomendable realizar este gasto con compuestos grandes, ya que conforme disminuimos el computo la ganancia en tiempo de ejecución se reduce. Esto último lo podemos comprobar en la tabla 6.4 en la fase de Inicializar, donde los tiempos son prácticamente similares. La diferencia en tiempo de ejecución en la fase de auto-optimización entre ambos modelos de dispositivos que se encuentran en el nodo, no varía en más del 5 % para este problema concreto. Una diferencia mayor entre ambos dispositivos aumentaría esta brecha.
En el Anexo C se muestra una ejecución de la fase de auto-optimización completa en modo heterogéneo con reparto de carga y optimización de hilos por bloque de cada *kernel* en cada uno de los dispositivos. Como se puede apreciar, primero se calculan

Combinaciones	Versión HETEROGENEA SIN reparto de carga Multicore + multiGPU	Versión HETEROGENEA CON reparto de carga Multicore + multiGPU	Mejora(%)
1	946.66	888.42	8
2	734.21	721.13	2
3	1019.07	928.03	9

Tabla 6.5: Tiempo de ejecución total en segundos y porcentaje de mejora para las combinaciones 1, 2 y 3 en el esquema multicore + multiGPU sin reparto de carga y con reparto de carga de la tabla 6.4. Ejecuciones realizadas en el nodo Jupiter con las 6 GPUs disponibles, 4 GPUs GeForce GTX 590 y 2 GPUs Tesla C2075.

cargas en función del tiempo de ejecución empleado en calcular el *fitness* del conjunto de entrenamiento. A continuación se calcula de la misma forma que en la sección 5.4, el número óptimo de hilos para los procesos que usan multicore y, seguidamente, el número óptimo de hilos por bloque para cada uno de los *kernels* en cada dispositivo. Para finalizar se muestra el tiempo final y un resumen de los parámetros paralelos finales para cada uno de los dispositivos.

6.5. Conclusiones

Para finalizar este capítulo y después de finalizar el estudio en multiGPU, podemos extraer una serie de conclusiones:

- Aplicando técnicas de multiGPU podemos dividir el trabajo entre todas ellas para mejorar el rendimiento. El reparto de cargas de trabajo entre los diferentes dispositivos permite dotar de una mayor carga de cómputo a los dispositivos más rápidos para intentar equiparar su finalización con los dispositivos más lentos. Esto conlleva la mejora con respecto a la versión de una sola GPU mucho más potente, como es la Tesla K40c, en más del 75 % de los casos en compuestos pequeños. En compuestos de grandes dimensiones siempre es aconsejable esta opción, como se comprueba en la tabla 6.3.

- Cuando usamos el modo de computación heterogéneo en la función Combinar se observa, a través de la figura 6.10, que su rendimiento es mayor conforme aumentamos el número de dispositivos involucrados en el cómputo, dado que el tamaño del conjunto combinado es mayor que el conjunto procesado en la fase de Inicializar. Por contra, en la fase de Inicializar se penaliza el uso de los dispositivos más lentos, ya que no hay suficiente carga de trabajo para hacer que se equiparen las GPUs más rápidas con mayor carga a los dispositivos más lentos.

- El uso de multiGPU en modo heterogénea con compuestos biológicos muy grandes, como sucede en el punto 6.3.2, se hace necesario para disminuir el coste temporal que conlleva el cálculo del *fitness*, como podemos observar en la tabla 6.3. Con estos resultados, queda justificado el uso de mayor número de recursos

para abordar este tipo de problemas, demostrando que repartir la carga computacional entre un mayor número de GPUs, aunque sean más lentas, lleva asociado generalmente un gran beneficio.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo se van a exponer las conclusiones finales de este trabajo fin de máster, junto a unas líneas futuras de trabajo para ir ahondando en el proceso de optimización del rendimiento.

7.1. Conclusiones

El proceso de acoplamiento molecular entre dos compuestos se ha demostrado que es un problema perfectamente abordable a través de metaheurísticas paralelas parametrizadas. Se ha demostrado que el uso de estos procesos metaheurísticos ofrece valores muy cercanos al óptimo en un tiempo realmente reducido usando técnicas masivamente paralelas en GPU.
Esta búsqueda de puntos de acoplamiento entre dos compuestos, tiene un amplio campo de aplicación en el mundo bioinformático mejorando los procesos de *docking* o cribado virtual. A continuación vamos a enumerar algunas conclusiones generales que podemos extraer de este trabajo, viendo el nivel de cumplimiento de los objetivos planteados en la parte inicial:

- Se ha optimizado el cálculo del potencial de Lennard-Jones aplicando técnicas masivamente paralelas en GPU y multiGPU, obteniendo *speed-up* de más de 15x en algunos casos con respeto a versiones paralelas en multicore usando memoria compartida, cumpliendo nuestro primer objetivo. Es importante tener en cuenta que más del 98 % del cómputo de la aplicación corresponde a este cálculo y, por tanto, se deben concentrar la mayoría de esfuerzos en optimizar su rendimiento.

- Ha sido desarrollado un esquema metaheurístico paralelo parametrizado, evolucionando de uno secuencial, mejorando su rendimiento en cada una de estas evoluciones, aplicando multicore usando memoria compartida y técnicas masivamente paralelas en esquemas híbridos con multicore + GPU y multicore + multiGPU.

- Se han analizado y obtenido experimentalmente los valores óptimos de rendimiento en los nodos y GPUs en los que se ha ejecutado la aplicación. Además se ha desarrollado un sistema de auto-optimización de parámetros paralelos para establecer los valores óptimos en la máquina en que se instala nuestra aplicación.

Este punto se ha desarrollado al demostrarse que la asignación de valores erróneos a parámetros paralelos, puede suponer un gran detrimento en el rendimiento de la aplicación. Con ello cumplimos todos los objetivos planteados inicialmente.

El trabajar con estos dispositivos, hace pensar que el modelo de programación establecido hasta ahora va a ir evolucionando de forma inexorable hacia una programación heterogénea (GPU o multiGPU) en los campos de investigación en que pueda ser posible su uso.

7.2. Trabajo futuro

Partiendo de la base de este trabajo fin de máster, se pueden extraer nuevas líneas de investigación que permitan ampliar funcionalidades y exportar a otros entornos. Vamos a enumerar algunas de ellas:

- El diseño de hiper-heurísticas [2] para obtener la mejor combinación de parámetros para poder aplicar al problema que estamos resolviendo sería muy útil para detectar la mejor combinación de los mismos, y aplicar por tanto la mejor estrategia posible.
- Extensión a un esquema híbrido de memoria compartida con paso de mensajes y GPU o multiGPU donde podamos explotar todos recursos de un cluster, pudiendo conseguir un mayor número de GPUs o procesadores para realizar nuestros cálculos.
- Pasar a un esquema de trabajo multi-objetivo para incrementar la confianza del acoplamiento en una determinada posición del receptor, utilizando otros cálculos para apoyar o desestimar esta predicción. Estos cálculos pueden consistir en aplicar más potenciales relacionados con la fuerza del enlace o relacionados con la interacción atómica, como el potencial electrostático.
- Perfeccionar los mecanismos de auto-optimización incluidos en este trabajo, estableciendo una serie de fases iniciales donde realizar un conjunto de *benchmarks* representativos y adaptar en fase de compilación el código.
- El uso de computación en multiGPU, y la comprobación en este trabajo de que su uso mejora el rendimiento de problema, nos hacer usar entornos que proporcionen una variedad de dispositivos para permitir este modo de computación. Amazon EC2 *Amazon Elastic Compute Cloud* [3] es una de estas vías a explorar, donde podemos obtener recursos en la nube de forma escalable y económicamente viable, ya que podemos acceder a recursos de computación de altas prestaciones sin necesidad de un gran desembolso económico. Amazon puede proveernos de instancias de GPU, para beneficiarnos de las capacidades de cómputo que nos proporciona la GPU y paralelizar nuestro algoritmo en ella, ahorrando costes y proporcionándonos rendimientos muy atractivos.

Bibliografía

[1] Luciano Alibrandi. Los procesadores tesla convierten la supercomputación en algo personal, 2008.

[2] Francisco Almeida, Domingo Giménez, José Juan López-Espín, and Melquiades Pérez-Pérez. Parameterized schemes of metaheuristics: Basic ideas and applications with Genetic Algorithms, Scatter Search, and GRASP. *on Systems, Man, and Cybernetics: Systems, IEEE Transactions*, 43(3):570–586, 2013.

[3] Inc. Amazon Web Services. Amazon Elastic Compute Cloud. Disponible en: `http://aws.amazon.com/es/ec2/`, 2015.

[4] AMBER. Amber 14 Nvidia gpu. Disponible en: `http://http://ambermd.org/gpus/`, 2014.

[5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, and Samuel Webb Williams. The landscape of parallel computing research: A view from Berkeley. Technical report, UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[6] Thomas Back, David B Fogel, and Zbigniew Michalewicz. *Handbook of evolutionary computation*. IOP Publishing Ltd., 1997.

[7] John E Beasley. Lagrangian relaxation. In *Modern heuristic techniques for combinatorial problems*, pages 243–303. John Wiley & Sons, Inc., 1993.

[8] Hans-Joachim Böhm. The development of a simple empirical scoring function to estimate the binding constant for a protein-ligand complex of known three-dimensional structure. *Journal of Computer-Aided Molecular Design*, 8(3):243–256, 1994.

[9] José M Cecilia, José M García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

[10] Richard D Cramer III and Svante B Wold. Comparative molecular field analysis (CoMFA), 1991. US Patent 5,025,388.

[11] José-Matías Cutillas-Lozano and Domingo Giménez. Modelling shared-memory metaheuristics and hyperheuristics for auto-tuning. *Multicore and GPU Programming*, page 19, 2015.

[12] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[13] Andrew Davidson, Yao Zhang, and John D Owens. An auto-tuned method for solving large tridiagonal systems on the gpu. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 956–965. IEEE, 2011.

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

[15] Barcelona Supercomputing Center. Centro Nacional de Supercomputación. `http://www.bsc.es`, 2014.

[16] Marco Dorigo. Optimization, learning and natural algorithms. *Ph. D. Thesis, Politecnico di Milano, Italy*, 1992.

[17] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, 2006.

[18] Todd JA Ewing, Shingo Makino, A Geoffrey Skillman, and Irwin D Kuntz. DOCK 4.0: search strategies for automated molecular docking of flexible molecule databases. *Journal of computer-Aided Molecular Design*, 15(5):411–428, 2001.

[19] Jianbin Fang, Ana Lucia Varbanescu, Baldomero Imbernon, Jose M Cecilia, and Horacio Perez-Sanchez. Parallel computation of non-bonded interactions in drug discovery: NVIDIA GPUs vs. Intel Xeon Phi. In *Proceedings of the 2nd International Work-Conference on Bioinformatics and Biomedical Engineering (IWBBIO'14)*, 2014.

[20] Adriana Favieri. Introducción a los cuaterniones. *Facultad Regional Haedo, Universidad Tecnológica Nacional, Argentina*, 2008.

[21] Thomas A Feo and Mauricio GC Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[22] Patrick Gaillard, Pierre-Alain Carrupt, Bernard Testa, and Alain Boudon. Molecular lipophilicity potential, a tool in 3D QSAR: method and applications. *Journal of Computer-Aided Molecular Design*, 8(2):83–96, 1994.

[23] Fred Glover and Gary A Kochenberger. *Handbook of metaheuristics*. Springer Science & Business Media, 2003.

[24] Fred Glover and Manuel Laguna. *Tabu Search*. Springer, 1999.

[25] David E Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion Wesley*, 1989.

[26] David S Goodsell and Arthur J Olson. Automated docking of substrates to proteins by simulated annealing. *Proteins: Structure, Function, and Bioinformatics*, 8(3):195–202, 1990.

[27] Jiri Jaros. Multi-GPU island-based genetic algorithm for solving the Knapsack problem. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2012.

[28] Holland John. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, 1992.

[29] Romano T Kroemer, Peter Hecht, and Klaus R Liedl. Different electrostatic descriptors in comparative molecular field analysis: A comparison of molecular electrostatic and Coulomb potentials. *Journal of Computational Chemistry*, 17(11):1296–1308, 1996.

[30] Eugene L Lawler. The traveling salesman problem: a guided tour of combinatorial optimization. *Wiley-Interscience Series in Discrete Mathematics*, 1985.

[31] Esteban López-Camacho, María Jesús García Godoy, José García-Nieto, Antonio J Nebro, and José F Aldana-Montes. Solving molecular flexible docking problems with metaheuristics: A comparative study. *Applied Soft Computing*, 28:379–393, 2015.

[32] Simone L Martins and Celso C Ribeiro. Metaheuristics and applications to optimization problems in telecommunications. In *Handbook of optimization in telecommunications*, pages 103–128. Springer, 2006.

[33] Belén Melián, José A Moreno Pérez, and J Marcos Moreno Vega. Metaheurísticas: una visión global. *Revista Iberoamericana de Inteligencia Artificial*, 19:7–28, 2003.

[34] Elaine C Meng, Brian K Shoichet, and Irwin D Kuntz. Automated docking with grid-based energy evaluation. *Journal of Computational Chemistry*, 13(4):505–524, 1992.

[35] Gabriela F Minetti, Carolina Salto, Hugo Alfonso, and Fernando Sanz Troiani. Técnicas metaheurísticas avanzadas aplicadas a la resolución de problemas bioinformáticos. In *XIV Workshop de Investigadores en Ciencias de la Computación*, 2012.

[36] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[37] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.

[38] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 30. ACM, 2009.

[39] NVIDIA. CUDA/C. Best Practices Guide v.5.5. Disponible en: http://developer.download.nvidia.com/compute/cuda/5_5/rel/docs/CUDA_Toolkit_Release_Notes.pdf, 2012.

[40] NVIDIA. CUDA/C. Best Practices Guide v.6.5. Disponible en: http://developer.download.nvidia.com/compute/cuda/6_5/rel/docs/CUDA_Getting_Started_Windows.pdf, 2012.

[41] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110. Whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[42] NVIDIA. CUDA/C. Best Practices Guide v.7.0. Disponible en: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, 2015.

[43] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[44] Panos M Pardalos and Mauricio GC Resende. *Handbook of applied optimization*. Oxford University Press, 2001.

[45] V Adrian Parsegian. *Van der Waals forces*. Cambridge Univ. Press, 2006.

[46] PDB. Protein Data Bank. Portal to Biological Macromolecular Structures. http://www.rcsb.org/pdb/home/home.do, 2015.

[47] Günther R Raidl. A unified view on hybrid metaheuristics. In *Hybrid Metaheuristics*, pages 1–12. Springer, 2006.

[48] Colin R Reeves. *Modern heuristic techniques for combinatorial problems*. John Wiley & Sons, Inc., 1993.

[49] Romelia Salomon-Ferrer, David A Case, and Ross C Walker. An overview of the AMBER biomolecular simulation package. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 3(2):198–210, 2013.

[50] Randall S Sexton, Bahram Alidaee, Robert E Dorsey, and John D Johnson. Global optimization for artificial neural networks: a Tabu search application. *European Journal of Operational Research*, 106(2):570–584, 1998.

[51] Brian K Shoichet. Virtual screening of chemical libraries. *Nature*, 432(7019):862–865, 2004.

[52] Edward A Silver, R Victor, V Vidal, and Dominique de Werra. A tutorial on heuristic methods. *European Journal of Operational Research*, 5(3):153–162, 1980.

[53] Sergio Filipe Sousa, Pedro Alexandrino Fernandes, and Maria Joao Ramos. Protein–ligand docking: current status and future challenges. *Proteins: Structure, Function, and Bioinformatics*, 65(1):15–26, 2006.

[54] Thomas G Stützle. *Local search algorithms for combinatorial problems: analysis, improvements, and new applications*, volume 220. Infix Sankt Augustin, Germany, 1999.

[55] Eric Taillard, Nouredine Melab, El-Ghazali Talbi, et al. Parallelization strategies for hybrid metaheuristics using a single GPU and multi-core resources. In *Parallel Problem Solving from Nature-PPSN XII*, pages 368–377. Springer, 2012.

[56] Liong Seng Tee, Sukehiro Gotoh, and Warren E Stewart. Molecular parameters for normal fluids. Lennard-Jones 12-6 potential. *Industrial & Engineering Chemistry Fundamentals*, 5(3):356–363, 1966.

[57] L Tripos. Tripos mol2 file format. *St. Louis, MO: Tripos*, 2007.

[58] Oleg Trott and Arthur J Olson. Autodock VINA: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of Computational Chemistry*, 31(2):455–461, 2010.

[59] Rob JM Vaessens, Emile HL Aarts, and Jan Karel Lenstra. A local search template. *Computers & Operations Research*, 25(11):969–979, 1998.

[60] Marcel L Verdonk, Jason C Cole, Michael J Hartshorn, Christopher W Murray, and Richard D Taylor. Improved protein–ligand docking using GOLD. *Proteins: Structure, Function, and Bioinformatics*, 52(4):609–623, 2003.

[61] Pablo Vidal and Enrique Alba. A multi-GPU implementation of a cellular genetic algorithm. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1–7. IEEE, 2010.

[62] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[63] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. ProQuest, 2008.

[64] Stelios H Zanakis, James R Evans, and Alkis A Vazacopoulos. Heuristic methods and applications: a categorized survey. *European Journal of Operational Research*, 43(1):88–110, 1989.

Anexo A

Ejemplo de ejecución de la fase de *warm-up* en multicore

```
CONFIG FILE: config-2bsm.met

Modo:  2
Tipo de cómputo:  0 HOMOGENEO
Identificador de dispositivo:  0
Fase de Warm_UP:  0
Conformaciones GPU(warm_up):  32768
Conformaciones Multicore CPU(warm_up):  64
Porcentaje de Conformaciones Multicore para Seleccion(warm_up):  50
Porcentaje de Conformaciones Multicore para Incluir(warm_up):  50
Limite tamaño de bloque (warm_up):  512
Pasadas optimizacion tamaño de bloque (warm_up):  1000
Fichero configuracion OPEN-MP:  ./openmp
Fichero configuracion GPU:  ./gpu
Fichero configuracion multiGPU:  ./multigpu
Alarma activada: ¡¡¡SI!!!
Duración de la alarma: 5 minutos)
Input Dir:  ./input
Output Dir:  ./output
Protein File:  ./input/2bsm_rec.mol2
Ligand File:  ./input/2bsm_lig.mol2
VdW Params:  ./input/vdw_params
Shift:  0.050000
Rotation angle:  180
NEIIni:  64
PEMIni:  20
IMEIni:  10
NEFMIni:  50
NEFPIni:  50
NEMSel:  100
```

```
NEPSel:  0
NMMCom:  100
NMPCom:  0
NPPCom:  0
PEMSel:  0
IMEImp:  0
NEMInc:  100
NIRFin:  3
NMIFin:  1
Hilos primer nivel Inicializar:  1
Hilos primer nivel Calculo Fitness:  1
Hilos segundo nivel Calculo Fitness:  1
Hilos primer nivel Seleccionar:  1
Hilos primer nivel Combinar:  1
Hilos segundo nivel Combinar:  1
Hilos primer nivel Mejorar:  1
Hilos primer nivel incluir:  1

OPEN-MP MODE...
***********
Mode: 2

**Conjunto de entrenamiento GENERADO**

**Optimización valores paralelos CALCULO DE FITNESS**
Hilos N1: 12  Tiempo: 4.291937
Hilos N1: 11  Tiempo: 4.672549
Hilos N1: 10  Tiempo: 5.113740
Hilos N1: 9  Tiempo: 5.725803
Hilos N1: 8  Tiempo: 6.156027
Hilos N1: 7  Tiempo: 7.103314
Hilos N1: 6  Tiempo: 8.361069
Hilos N1: 5  Tiempo: 9.802348
Hilos N1: 4  Tiempo: 12.412308
Hilos N1: 3  Tiempo: 16.004802
Hilos N1: 2  Tiempo: 23.385111
MEJOR Hilos N1: 12 tiempo mejor: 4.291937
Hilos N1: 6  Hilos N2 2  Tiempo: 4.368809
Hilos N1: 2  Hilos N2 4  Tiempo: 7.090845
MEJOR Hilos N1: 6, Hilos N2: 2 tiempo mejor: 4.368809

**Optimización de la fase de INICIALIZAR**
Hilos N1: 12  Tiempo: 0.000435
Hilos N1: 11  Tiempo: 0.000201
Hilos N1: 10  Tiempo: 0.000221
```

```
Hilos N1: 9  Tiempo: 0.000259
Hilos N1: 8  Tiempo: 0.000273
Hilos N1: 7  Tiempo: 0.000318
Hilos N1: 6  Tiempo: 0.000364
Hilos N1: 5  Tiempo: 0.000473
Hilos N1: 4  Tiempo: 0.000534
Hilos N1: 3  Tiempo: 0.000720
Hilos N1: 2  Tiempo: 0.001058
MEJOR Hilos N1 función INICIALIZAR: 11 tiempo mejor: 0.000201

**Optimización de la fase de SELECCIONAR**
Hilos N1: 12  Tiempo: 0.000304
Hilos N1: 11  Tiempo: 0.000025
Hilos N1: 10  Tiempo: 0.000025
Hilos N1: 9  Tiempo: 0.000024
Hilos N1: 8  Tiempo: 0.000020
Hilos N1: 7  Tiempo: 0.000027
Hilos N1: 6  Tiempo: 0.000028
Hilos N1: 5  Tiempo: 0.000030
Hilos N1: 4  Tiempo: 0.000034
Hilos N1: 3  Tiempo: 0.000037
Hilos N1: 2  Tiempo: 0.000138
MEJOR Hilos N1 función SELECCIONAR: 8 tiempo mejor: 0.000020

**Optimización valores paralelos de la fase COMBINAR**
Hilos N1: 12  Tiempo: 1.813925
Hilos N1: 11  Tiempo: 1.752418
Hilos N1: 10  Tiempo: 1.752541
Hilos N1: 9  Tiempo: 1.758126
Hilos N1: 8  Tiempo: 1.752778
Hilos N1: 7  Tiempo: 1.759511
Hilos N1: 6  Tiempo: 1.758917
Hilos N1: 5  Tiempo: 1.758630
Hilos N1: 4  Tiempo: 1.758794
Hilos N1: 3  Tiempo: 1.757673
Hilos N1: 2  Tiempo: 1.758302
MEJOR Hilos N1 función COMBINAR: 11 tiempo mejor: 1.752418
Hilos N1: 6  Hilos N2 2  Tiempo: 1.758163
Hilos N1: 2  Hilos N2 4  Tiempo: 1.757737
MEJOR Hilos N1 función COMBINAR: 3, Hilos N2: 4 tiempo mejor: 1.757737

**Optimización de la fase de MEJORAR**
Hilos N1: 12  Tiempo: 0.002640
Hilos N1: 11  Tiempo: 0.005233
Hilos N1: 10  Tiempo: 0.000174
```

```
Hilos N1: 9  Tiempo: 0.000152
Hilos N1: 8  Tiempo: 0.000163
Hilos N1: 7  Tiempo: 0.000191
Hilos N1: 6  Tiempo: 0.000220
Hilos N1: 5  Tiempo: 0.000261
Hilos N1: 4  Tiempo: 0.000322
Hilos N1: 3  Tiempo: 0.000418
Hilos N1: 2  Tiempo: 0.000583
MEJOR Hilos N1 función MEJORAR: 9 tiempo mejor: 0.000152

**Optimización de la fase de INCLUIR**
Hilos N1: 12  Tiempo: 0.004529
Hilos N1: 11  Tiempo: 0.003181
Hilos N1: 10  Tiempo: 0.000021
Hilos N1: 9  Tiempo: 0.000019
Hilos N1: 8  Tiempo: 0.000015
Hilos N1: 7  Tiempo: 0.000018
Hilos N1: 6  Tiempo: 0.000020
Hilos N1: 5  Tiempo: 0.000022
Hilos N1: 4  Tiempo: 0.000024
Hilos N1: 3  Tiempo: 0.000029
Hilos N1: 2  Tiempo: 0.000064
MEJOR Hilos N1 función INCLUIR: 8 tiempo mejor: 0.000015

*****************
WARM_UP: 137.431502 seg
*****************
Valores óptimos parámetros paralelos:
Nivel 1. Función Inicializar: 11
Nivel 1. Función Seleccionar: 8
Nivel 1. Función Combinar: 11
Nivel 2. Función Combinar: 1
Nivel 1. Función Mejorar: 9
Nivel 1. Función Incluir: 8
Nivel 1. Cálculo de Fitness: 12
Nivel 2. Cálculo de Fitness: 1
```

Anexo B

Ejemplo de ejecución de la fase de *warm-up* en GPU

```
CONFIG FILE: config-2bsm.met

Modo:  0
Tipo de cómputo:  0 HOMOGENEO
Identificador de dispositivo:  0
Fase de Warm_UP:  0
Conformaciones GPU(warm_up):  32768
Conformaciones Multicore CPU(warm_up):  64
Porcentaje de Conformaciones Multicore para Seleccion(warm_up):  50
Porcentaje de Conformaciones Multicore para Incluir(warm_up):  50
Limite tamaño de bloque (warm_up):  512
Pasadas optimizacion tamaño de bloque (warm_up):  1000
Fichero configuracion OPEN-MP:  ./openmp
Fichero configuracion GPU:  ./gpu
Fichero configuracion multiGPU:  ./multigpu
Alarma activada: ¡¡¡SI!!!
Duración de la alarma: 5 minutos)
Input Dir:  ./input
Output Dir:  ./output
Protein File:  ./input/2bsm_rec.mol2
Ligand File:  ./input/2bsm_lig.mol2
VdW Params:  ./input/vdw_params
Shift:  0.050000
Rotation angle:  180
NEIIni:  64
PEMIni:  20
IMEIni:  10
NEFMIni:  50
NEFPIni:  50
NEMSel:  100
```

```
NEPSel:  0
NMMCom:  100
NMPCom:  0
NPPCom:  0
PEMSel:  0
IMEImp:  0
NEMInc:  100
NIRFin:  3
NMIFin:  1
Hilos primer nivel Inicializar:  1
Numero de Hilos por bloque función Cálculo de FITNESS:  128
Numero de Hilos por bloque función MOVE Inicializar:  128
Numero de Hilos por bloque función ROTACION Inicializar:  128
Numero de Hilos por bloque función QUAT Inicializar:  128
Numero de Hilos por bloque función CURAND Inicializar:  128
Hilos primer nivel Combinar:  1
Numero de Hilos por bloque función MOVE mejorar:  128
Numero de Hilos por bloque función INCLUIR mejorar:  128
Hilos primer nivel incluir:  1

GPU SOLVER MODE...
******************
Mode: 0

**Conjunto de entrenamiento GENERADO**

GeForce GTX 590

**Optimización de la fase de INICIALIZAR**
Hilos N1: 12  Tiempo: 0.000481
Hilos N1: 11  Tiempo: 0.000208
Hilos N1: 10  Tiempo: 0.000221
Hilos N1: 9  Tiempo: 0.000252
Hilos N1: 8  Tiempo: 0.000273
Hilos N1: 7  Tiempo: 0.000324
Hilos N1: 6  Tiempo: 0.000410
Hilos N1: 5  Tiempo: 0.000435
Hilos N1: 4  Tiempo: 0.000536
Hilos N1: 3  Tiempo: 0.000716
Hilos N1: 2  Tiempo: 0.001058
MEJOR Hilos N1 función INICIALIZAR: 11 tiempo mejor: 0.000208

**Optimización de la fase de SELECCIONAR**
Hilos N1: 12  Tiempo: 0.000313
Hilos N1: 11  Tiempo: 0.000024
```

```
Hilos N1: 10  Tiempo: 0.000025
Hilos N1: 9  Tiempo: 0.000025
Hilos N1: 8  Tiempo: 0.000021
Hilos N1: 7  Tiempo: 0.000026
Hilos N1: 6  Tiempo: 0.000033
Hilos N1: 5  Tiempo: 0.000031
Hilos N1: 4  Tiempo: 0.000033
Hilos N1: 3  Tiempo: 0.000042
Hilos N1: 2  Tiempo: 0.000088
MEJOR Hilos N1 función SELECCIONAR: 8 tiempo mejor: 0.000021

**Optimización valores paralelos de la fase COMBINAR**
Hilos N1: 12  Tiempo: 1.872790
Hilos N1: 11  Tiempo: 1.821579
Hilos N1: 10  Tiempo: 1.827899
Hilos N1: 9  Tiempo: 1.836653
Hilos N1: 8  Tiempo: 1.845418
Hilos N1: 7  Tiempo: 1.853441
Hilos N1: 6  Tiempo: 1.860088
Hilos N1: 5  Tiempo: 1.872802
Hilos N1: 4  Tiempo: 1.886503
Hilos N1: 3  Tiempo: 1.892379
Hilos N1: 2  Tiempo: 1.901445
MEJOR Hilos N1 función COMBINAR: 11 tiempo mejor: 1.821579
Hilos N1: 6  Hilos N2 2  Tiempo: 1.909625
Hilos N1: 2  Hilos N2 4  Tiempo: 1.924650
MEJOR Hilos N1 función COMBINAR: 6, Hilos N2: 2 tiempo mejor: 1.909625

**Optimización de la fase de INCLUIR**
Hilos N1: 12  Tiempo: 0.000289
Hilos N1: 11  Tiempo: 0.000023
Hilos N1: 10  Tiempo: 0.000023
Hilos N1: 9  Tiempo: 0.000022
Hilos N1: 8  Tiempo: 0.000019
Hilos N1: 7  Tiempo: 0.000021
Hilos N1: 6  Tiempo: 0.000031
Hilos N1: 5  Tiempo: 0.000029
Hilos N1: 4  Tiempo: 0.000035
Hilos N1: 3  Tiempo: 0.000044
Hilos N1: 2  Tiempo: 0.000082
MEJOR Hilos N1 función INCLUIR: 8 tiempo mejor: 0.000019

Tiempo 1.481778 para tamaño de bloque CON 64
Tiempo 1.516049 para tamaño de bloque CON 128
Tiempo 1.530438 para tamaño de bloque CON 192
```

```
Tiempo 1.529554 para tamaño de bloque CON 256
Tiempo 2.208976 para tamaño de bloque CON 320
Tiempo 1.916988 para tamaño de bloque CON 384
Tiempo 1.605887 para tamaño de bloque CON 448
Tiempo 1.563655 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en Cálculo de FITNESS: 64

Tiempo 0.003655 para tamaño de bloque CON 64
Tiempo 0.003679 para tamaño de bloque CON 128
Tiempo 0.003691 para tamaño de bloque CON 192
Tiempo 0.003676 para tamaño de bloque CON 256
Tiempo 0.003628 para tamaño de bloque CON 320
Tiempo 0.003628 para tamaño de bloque CON 384
Tiempo 0.003599 para tamaño de bloque CON 448
Tiempo 0.003627 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Inicializar: 448

Tiempo 0.003038 para tamaño de bloque CON 64
Tiempo 0.003019 para tamaño de bloque CON 128
Tiempo 0.003038 para tamaño de bloque CON 192
Tiempo 0.003034 para tamaño de bloque CON 256
Tiempo 0.003056 para tamaño de bloque CON 320
Tiempo 0.003035 para tamaño de bloque CON 384
Tiempo 0.003037 para tamaño de bloque CON 448
Tiempo 0.003001 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en ROTACION en Inicializar: 512

Tiempo 0.003019 para tamaño de bloque CON 64
Tiempo 0.002989 para tamaño de bloque CON 128
Tiempo 0.003013 para tamaño de bloque CON 192
Tiempo 0.003030 para tamaño de bloque CON 256
Tiempo 0.003026 para tamaño de bloque CON 320
Tiempo 0.002988 para tamaño de bloque CON 384
Tiempo 0.003020 para tamaño de bloque CON 448
Tiempo 0.003005 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en QUAT en Inicializar: 384

Tiempo 0.002934 para tamaño de bloque CON 64
Tiempo 0.002884 para tamaño de bloque CON 128
Tiempo 0.002935 para tamaño de bloque CON 192
Tiempo 0.002972 para tamaño de bloque CON 256
Tiempo 0.002907 para tamaño de bloque CON 320
Tiempo 0.002943 para tamaño de bloque CON 384
Tiempo 0.002969 para tamaño de bloque CON 448
Tiempo 0.002913 para tamaño de bloque CON 512
```

```
Valor Optimizado HILOS/BLOQUE en CURAND en Inicializar: 128

Tiempo 0.003274 para tamaño de bloque CON 64
Tiempo 0.003205 para tamaño de bloque CON 128
Tiempo 0.003210 para tamaño de bloque CON 192
Tiempo 0.003256 para tamaño de bloque CON 256
Tiempo 0.003239 para tamaño de bloque CON 320
Tiempo 0.003223 para tamaño de bloque CON 384
Tiempo 0.003223 para tamaño de bloque CON 448
Tiempo 0.003236 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Mejorar: 128

Tiempo 0.003557 para tamaño de bloque CON 64
Tiempo 0.003597 para tamaño de bloque CON 128
Tiempo 0.003591 para tamaño de bloque CON 192
Tiempo 0.003604 para tamaño de bloque CON 256
Tiempo 0.003550 para tamaño de bloque CON 320
Tiempo 0.003566 para tamaño de bloque CON 384
Tiempo 0.003591 para tamaño de bloque CON 448
Tiempo 0.003649 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en INCLUIR en Mejorar: 128

*****************
WARM_UP: 105.770509 seg
*****************
Valores óptimos de parámetros paralelos:
Nivel 1. Función Inicializar: 11
Nivel 1. Función Seleccionar: 8
Nivel 1. Función Combinar: 11
Nivel 2. Función Combinar: 1
Nivel 1. Función Incluir: 8
Device 0, modelo GeForce GTX 590
Número de hilos por bloque función CALCULO DE FITNESS 64
Número de hilos por bloque función MOVE de Inicializar 448
Número de hilos por bloque función ROT de Inicializar 512
Número de hilos por bloque función QUAT de Inicializar 384
Número de hilos por bloque función CURAND de Inicializar 128
Número de hilos por bloque función MOVE de Mejorar 128
Número de hilos por bloque función INCLUIR de Mejorar 128
```

Anexo C

Ejemplo de ejecución de la fase de *warm-up* en multiGPU

```
CONFIG FILE: ./config-2bsm.met

Modo:  3
Tipo de Computo:  1 HETEROGENEO
Identificador de dispositivo:  0
Fase de Warm_UP:  1
Fase de cálculo Reparto de carga:  1
Conformaciones GPU(warm_up):  65536
Conformaciones Multicore CPU(warm_up):  64
Porcentaje de Conformaciones Multicore para Seleccion(warm_up):  50
Porcentaje de Conformaciones Multicore para Incluir(warm_up):  50
Limite tamaño de bloque (warm_up):  512
Pasadas optimizacion tamaño de bloque (warm_up):  1000
Fichero configuracion OPEN-MP:  ./openmp
Fichero configuracion GPU:  ./gpu
Fichero configuracion multiGPU:  ./multigpu
Alarma activada: NO
Duración de la alarma: 60 minutos)
Input Dir:  ./input
Output Dir:  ./output
Protein File:  ./input/2bsm_rec.mol2
Ligand File:  ./input/2bsm_lig.mol2
VdW Params:  ./input/vdw_params
Shift:  0.110000
Rotation angle:  360
NEIIni:  64
PEMIni:  0
IMEIni:  0
NEFMIni:  0
NEFPIni:  100
```

```
NEMSel:  50
NEPSel:  50
NMMCom:  25
NMPCom:  0
NPPCom:  0
PEMSel:  0
IMEImp:  0
NEMInc:  100
NIRFin:  10
NMIFin:  10
Hilos primer nivel Inicializar:  12
Hilos primer nivel Seleccionar:  12
Numero de Hilos por bloque función Cálculo de FITNESS:  64
Numero de Hilos por bloque función MOVE Inicializar:  448
Numero de Hilos por bloque función ROTACION Inicializar:  512
Numero de Hilos por bloque función QUAT Inicializar:  384
Numero de Hilos por bloque función CURAND Inicializar:  128
Hilos primer nivel Combinar:  3
Hilos segundo nivel Combinar:  1
Numero de Hilos por bloque función MOVE mejorar:  128
Numero de Hilos por bloque función INCLUIR mejorar:  128
Hilos primer nivel incluir:  12

MULTI-GPU MODE...
***********
Mode: 3

**Conjunto de entrenamiento GENERADO**

GeForce GTX 590
Device 0 name GeForce GTX 590
Device 1 name Tesla C2075
Device 2 name GeForce GTX 590
Device 3 name GeForce GTX 590
Device 4 name GeForce GTX 590
Device 5 name Tesla C2075

*** FASE REPARTO DE CARGA ***

Device 0 Modelo: GeForce GTX 590 Tiempo GPU: 2.979179 seg
Device 1 Modelo: Tesla C2075 Tiempo GPU: 3.642564 seg
Device 2 Modelo: GeForce GTX 590 Tiempo GPU: 3.031161 seg
Device 3 Modelo: GeForce GTX 590 Tiempo GPU: 3.038443 seg
Device 4 Modelo: GeForce GTX 590 Tiempo GPU: 3.042753 seg
Device 5 Modelo: Tesla C2075 Tiempo GPU: 3.639866 seg
```

```
Device 4 modelo: GeForce GTX 590 Carga de trabajo 19
Device 3 modelo: Tesla C2075 Carga de trabajo 19
Device 2 modelo: GeForce GTX 590 Carga de trabajo 16
Device 0 modelo: GeForce GTX 590 Carga de trabajo 16
Device 5 modelo: GeForce GTX 590 Carga de trabajo 15
Device 1 modelo: Tesla C2075 Carga de trabajo 15


**Optimización de la fase de INICIALIZAR**
Hilos N1: 12  Tiempo: 0.000456
Hilos N1: 11  Tiempo: 0.000202
Hilos N1: 10  Tiempo: 0.000220
Hilos N1: 9  Tiempo: 0.000250
Hilos N1: 8  Tiempo: 0.000270
Hilos N1: 7  Tiempo: 0.000314
Hilos N1: 6  Tiempo: 0.000365
Hilos N1: 5  Tiempo: 0.000434
Hilos N1: 4  Tiempo: 0.000569
Hilos N1: 3  Tiempo: 0.000715
Hilos N1: 2  Tiempo: 0.001058
MEJOR Hilos N1 función INICIALIZAR: 11 tiempo mejor: 0.000202

**Optimización de la fase de SELECCIONAR**
Hilos N1: 12  Tiempo: 0.000330
Hilos N1: 11  Tiempo: 0.000026
Hilos N1: 10  Tiempo: 0.000025
Hilos N1: 9  Tiempo: 0.000027
Hilos N1: 8  Tiempo: 0.000021
Hilos N1: 7  Tiempo: 0.000029
Hilos N1: 6  Tiempo: 0.000029
Hilos N1: 5  Tiempo: 0.000031
Hilos N1: 4  Tiempo: 0.000029
Hilos N1: 3  Tiempo: 0.000047
Hilos N1: 2  Tiempo: 0.000113
MEJOR Hilos N1 función SELECCIONAR: 8 tiempo mejor: 0.000021

**Optimización valores paralelos de la fase COMBINAR**
Hilos N1: 12  Tiempo: 1.872530
Hilos N1: 11  Tiempo: 1.822587
Hilos N1: 10  Tiempo: 1.828964
Hilos N1: 9  Tiempo: 1.837347
Hilos N1: 8  Tiempo: 1.846127
Hilos N1: 7  Tiempo: 1.855730
Hilos N1: 6  Tiempo: 1.861383
```

```
Hilos N1: 5  Tiempo: 1.877957
Hilos N1: 4  Tiempo: 1.883816
Hilos N1: 3  Tiempo: 1.893049
Hilos N1: 2  Tiempo: 1.902188
MEJOR Hilos N1 función COMBINAR: 11 tiempo mejor: 1.822587
Hilos N1: 6  Hilos N2 2  Tiempo: 1.916262
Hilos N1: 2  Hilos N2 4  Tiempo: 1.918688
MEJOR Hilos N1 función COMBINAR: 6, Hilos N2: 2 tiempo mejor: 1.916262

**Optimización de la fase de INCLUIR**
Hilos N1: 12  Tiempo: 0.000299
Hilos N1: 11  Tiempo: 0.000020
Hilos N1: 10  Tiempo: 0.000019
Hilos N1: 9  Tiempo: 0.000021
Hilos N1: 8  Tiempo: 0.000016
Hilos N1: 7  Tiempo: 0.000016
Hilos N1: 6  Tiempo: 0.000025
Hilos N1: 5  Tiempo: 0.000022
Hilos N1: 4  Tiempo: 0.000023
Hilos N1: 3  Tiempo: 0.000029
Hilos N1: 2  Tiempo: 0.000074
MEJOR Hilos N1 función INCLUIR: 8 tiempo mejor: 0.000016


GeForce GTX 590
Tiempo 2.947522 para tamaño de bloque CON 64
Tiempo 2.965285 para tamaño de bloque CON 128
Tiempo 2.994031 para tamaño de bloque CON 192
Tiempo 2.979076 para tamaño de bloque CON 256
Tiempo 4.288132 para tamaño de bloque CON 320
Tiempo 3.739621 para tamaño de bloque CON 384
Tiempo 3.150610 para tamaño de bloque CON 448
Tiempo 3.063914 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en Cálculo de FITNESS: 64
Tiempo 0.003391 para tamaño de bloque CON 64
Tiempo 0.003364 para tamaño de bloque CON 128
Tiempo 0.003358 para tamaño de bloque CON 192
Tiempo 0.003387 para tamaño de bloque CON 256
Tiempo 0.003349 para tamaño de bloque CON 320
Tiempo 0.003286 para tamaño de bloque CON 384
Tiempo 0.003288 para tamaño de bloque CON 448
Tiempo 0.003310 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Inicializar: 384
Tiempo 0.003021 para tamaño de bloque CON 64
Tiempo 0.003024 para tamaño de bloque CON 128
```

```
Tiempo 0.002989 para tamaño de bloque CON 192
Tiempo 0.003000 para tamaño de bloque CON 256
Tiempo 0.002985 para tamaño de bloque CON 320
Tiempo 0.002994 para tamaño de bloque CON 384
Tiempo 0.003004 para tamaño de bloque CON 448
Tiempo 0.002995 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en ROTACION en Inicializar: 320
Tiempo 0.002841 para tamaño de bloque CON 64
Tiempo 0.002912 para tamaño de bloque CON 128
Tiempo 0.002867 para tamaño de bloque CON 192
Tiempo 0.002857 para tamaño de bloque CON 256
Tiempo 0.002857 para tamaño de bloque CON 320
Tiempo 0.002884 para tamaño de bloque CON 384
Tiempo 0.002899 para tamaño de bloque CON 448
Tiempo 0.002889 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en QUAT en Inicializar: 64
Tiempo 0.002891 para tamaño de bloque CON 64
Tiempo 0.002933 para tamaño de bloque CON 128
Tiempo 0.002912 para tamaño de bloque CON 192
Tiempo 0.002936 para tamaño de bloque CON 256
Tiempo 0.002885 para tamaño de bloque CON 320
Tiempo 0.002930 para tamaño de bloque CON 384
Tiempo 0.002924 para tamaño de bloque CON 448
Tiempo 0.002938 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en CURAND en Inicializar: 64
Tiempo 0.003095 para tamaño de bloque CON 64
Tiempo 0.003047 para tamaño de bloque CON 128
Tiempo 0.003076 para tamaño de bloque CON 192
Tiempo 0.003086 para tamaño de bloque CON 256
Tiempo 0.003075 para tamaño de bloque CON 320
Tiempo 0.003080 para tamaño de bloque CON 384
Tiempo 0.003121 para tamaño de bloque CON 448
Tiempo 0.003083 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Mejorar: 64
Tiempo 0.003459 para tamaño de bloque CON 64
Tiempo 0.003461 para tamaño de bloque CON 128
Tiempo 0.003488 para tamaño de bloque CON 192
Tiempo 0.003445 para tamaño de bloque CON 256
Tiempo 0.003485 para tamaño de bloque CON 320
Tiempo 0.003462 para tamaño de bloque CON 384
Tiempo 0.003509 para tamaño de bloque CON 448
Tiempo 0.003453 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en INCLUIR en Mejorar: 64

*****************
```

```
WARM_UP: 240.044559 seg
*****************

GeForce GTX 590
Tiempo 2.947818 para tamaño de bloque CON 64
Tiempo 2.968167 para tamaño de bloque CON 128
Tiempo 2.991071 para tamaño de bloque CON 192
Tiempo 2.986225 para tamaño de bloque CON 256
Tiempo 4.295405 para tamaño de bloque CON 320
Tiempo 3.733149 para tamaño de bloque CON 384
Tiempo 3.157335 para tamaño de bloque CON 448
Tiempo 3.068333 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en Cálculo de FITNESS: 64
Tiempo 0.003345 para tamaño de bloque CON 64
Tiempo 0.003346 para tamaño de bloque CON 128
Tiempo 0.003357 para tamaño de bloque CON 192
Tiempo 0.003357 para tamaño de bloque CON 256
Tiempo 0.003348 para tamaño de bloque CON 320
Tiempo 0.003240 para tamaño de bloque CON 384
Tiempo 0.003261 para tamaño de bloque CON 448
Tiempo 0.003257 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Inicializar: 384
Tiempo 0.002975 para tamaño de bloque CON 64
Tiempo 0.002964 para tamaño de bloque CON 128
Tiempo 0.002971 para tamaño de bloque CON 192
Tiempo 0.002998 para tamaño de bloque CON 256
Tiempo 0.002979 para tamaño de bloque CON 320
Tiempo 0.002995 para tamaño de bloque CON 384
Tiempo 0.002972 para tamaño de bloque CON 448
Tiempo 0.003007 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en ROTACION en Inicializar: 128
Tiempo 0.002844 para tamaño de bloque CON 64
Tiempo 0.002882 para tamaño de bloque CON 128
Tiempo 0.002871 para tamaño de bloque CON 192
Tiempo 0.002851 para tamaño de bloque CON 256
Tiempo 0.002862 para tamaño de bloque CON 320
Tiempo 0.002873 para tamaño de bloque CON 384
Tiempo 0.002831 para tamaño de bloque CON 448
Tiempo 0.002827 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en QUAT en Inicializar: 512
Tiempo 0.002836 para tamaño de bloque CON 64
Tiempo 0.002904 para tamaño de bloque CON 128
Tiempo 0.002904 para tamaño de bloque CON 192
Tiempo 0.002893 para tamaño de bloque CON 256
Tiempo 0.002906 para tamaño de bloque CON 320
```

```
Tiempo 0.002888 para tamaño de bloque CON 384
Tiempo 0.002893 para tamaño de bloque CON 448
Tiempo 0.002877 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en CURAND en Inicializar: 512
Tiempo 0.003051 para tamaño de bloque CON 64
Tiempo 0.003037 para tamaño de bloque CON 128
Tiempo 0.003052 para tamaño de bloque CON 192
Tiempo 0.003074 para tamaño de bloque CON 256
Tiempo 0.003045 para tamaño de bloque CON 320
Tiempo 0.003049 para tamaño de bloque CON 384
Tiempo 0.003061 para tamaño de bloque CON 448
Tiempo 0.003065 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Mejorar: 512
Tiempo 0.003463 para tamaño de bloque CON 64
Tiempo 0.003392 para tamaño de bloque CON 128
Tiempo 0.003490 para tamaño de bloque CON 192
Tiempo 0.003462 para tamaño de bloque CON 256
Tiempo 0.003472 para tamaño de bloque CON 320
Tiempo 0.003453 para tamaño de bloque CON 384
Tiempo 0.003508 para tamaño de bloque CON 448
Tiempo 0.003430 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en INCLUIR en Mejorar: 512

*****************
WARM_UP: 239.832207 seg
*****************

GeForce GTX 590
Tiempo 2.950296 para tamaño de bloque CON 64
Tiempo 2.967751 para tamaño de bloque CON 128
Tiempo 2.985212 para tamaño de bloque CON 192
Tiempo 2.982981 para tamaño de bloque CON 256
Tiempo 4.287941 para tamaño de bloque CON 320
Tiempo 3.735324 para tamaño de bloque CON 384
Tiempo 3.159607 para tamaño de bloque CON 448
Tiempo 3.070610 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en Cálculo de FITNESS: 64
Tiempo 0.003505 para tamaño de bloque CON 64
Tiempo 0.003491 para tamaño de bloque CON 128
Tiempo 0.003495 para tamaño de bloque CON 192
Tiempo 0.003478 para tamaño de bloque CON 256
Tiempo 0.003486 para tamaño de bloque CON 320
Tiempo 0.003386 para tamaño de bloque CON 384
Tiempo 0.003415 para tamaño de bloque CON 448
Tiempo 0.003406 para tamaño de bloque CON 512
```

```
Valor Optimizado HILOS/BLOQUE en MOVE en Inicializar: 384
Tiempo 0.003139 para tamaño de bloque CON 64
Tiempo 0.003140 para tamaño de bloque CON 128
Tiempo 0.003123 para tamaño de bloque CON 192
Tiempo 0.003111 para tamaño de bloque CON 256
Tiempo 0.003103 para tamaño de bloque CON 320
Tiempo 0.003127 para tamaño de bloque CON 384
Tiempo 0.003099 para tamaño de bloque CON 448
Tiempo 0.003122 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en ROTACION en Inicializar: 448
Tiempo 0.003039 para tamaño de bloque CON 64
Tiempo 0.003039 para tamaño de bloque CON 128
Tiempo 0.003049 para tamaño de bloque CON 192
Tiempo 0.003026 para tamaño de bloque CON 256
Tiempo 0.003059 para tamaño de bloque CON 320
Tiempo 0.003021 para tamaño de bloque CON 384
Tiempo 0.003011 para tamaño de bloque CON 448
Tiempo 0.003018 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en QUAT en Inicializar: 448
Tiempo 0.002974 para tamaño de bloque CON 64
Tiempo 0.003034 para tamaño de bloque CON 128
Tiempo 0.003054 para tamaño de bloque CON 192
Tiempo 0.003055 para tamaño de bloque CON 256
Tiempo 0.003075 para tamaño de bloque CON 320
Tiempo 0.003026 para tamaño de bloque CON 384
Tiempo 0.003039 para tamaño de bloque CON 448
Tiempo 0.003006 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en CURAND en Inicializar: 64
Tiempo 0.003200 para tamaño de bloque CON 64
Tiempo 0.003214 para tamaño de bloque CON 128
Tiempo 0.003171 para tamaño de bloque CON 192
Tiempo 0.003178 para tamaño de bloque CON 256
Tiempo 0.003188 para tamaño de bloque CON 320
Tiempo 0.003204 para tamaño de bloque CON 384
Tiempo 0.003164 para tamaño de bloque CON 448
Tiempo 0.003181 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Mejorar: 64
Tiempo 0.003557 para tamaño de bloque CON 64
Tiempo 0.003565 para tamaño de bloque CON 128
Tiempo 0.003604 para tamaño de bloque CON 192
Tiempo 0.003622 para tamaño de bloque CON 256
Tiempo 0.003585 para tamaño de bloque CON 320
Tiempo 0.003514 para tamaño de bloque CON 384
Tiempo 0.003661 para tamaño de bloque CON 448
Tiempo 0.003560 para tamaño de bloque CON 512
```

```
Valor Optimizado HILOS/BLOQUE en INCLUIR en Mejorar: 64

*****************
WARM_UP: 240.471805 seg
*****************

GeForce GTX 590
Tiempo 2.951231 para tamaño de bloque CON 64
Tiempo 2.982391 para tamaño de bloque CON 128
Tiempo 3.001578 para tamaño de bloque CON 192
Tiempo 2.995944 para tamaño de bloque CON 256
Tiempo 4.304501 para tamaño de bloque CON 320
Tiempo 3.745844 para tamaño de bloque CON 384
Tiempo 3.155741 para tamaño de bloque CON 448
Tiempo 3.071718 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en Cálculo de FITNESS: 64
Tiempo 0.003510 para tamaño de bloque CON 64
Tiempo 0.003489 para tamaño de bloque CON 128
Tiempo 0.003486 para tamaño de bloque CON 192
Tiempo 0.003486 para tamaño de bloque CON 256
Tiempo 0.003488 para tamaño de bloque CON 320
Tiempo 0.003406 para tamaño de bloque CON 384
Tiempo 0.003404 para tamaño de bloque CON 448
Tiempo 0.003397 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Inicializar: 512
Tiempo 0.003132 para tamaño de bloque CON 64
Tiempo 0.003129 para tamaño de bloque CON 128
Tiempo 0.003126 para tamaño de bloque CON 192
Tiempo 0.003127 para tamaño de bloque CON 256
Tiempo 0.003096 para tamaño de bloque CON 320
Tiempo 0.003141 para tamaño de bloque CON 384
Tiempo 0.003105 para tamaño de bloque CON 448
Tiempo 0.003123 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en ROTACION en Inicializar: 320
Tiempo 0.003019 para tamaño de bloque CON 64
Tiempo 0.003014 para tamaño de bloque CON 128
Tiempo 0.003051 para tamaño de bloque CON 192
Tiempo 0.003036 para tamaño de bloque CON 256
Tiempo 0.003028 para tamaño de bloque CON 320
Tiempo 0.003062 para tamaño de bloque CON 384
Tiempo 0.003014 para tamaño de bloque CON 448
Tiempo 0.003025 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en QUAT en Inicializar: 128
Tiempo 0.003006 para tamaño de bloque CON 64
Tiempo 0.003024 para tamaño de bloque CON 128
```

```
Tiempo 0.003034 para tamaño de bloque CON 192
Tiempo 0.003037 para tamaño de bloque CON 256
Tiempo 0.003019 para tamaño de bloque CON 320
Tiempo 0.003052 para tamaño de bloque CON 384
Tiempo 0.003058 para tamaño de bloque CON 448
Tiempo 0.003048 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en CURAND en Inicializar: 64
Tiempo 0.003179 para tamaño de bloque CON 64
Tiempo 0.003200 para tamaño de bloque CON 128
Tiempo 0.003163 para tamaño de bloque CON 192
Tiempo 0.003175 para tamaño de bloque CON 256
Tiempo 0.003236 para tamaño de bloque CON 320
Tiempo 0.003210 para tamaño de bloque CON 384
Tiempo 0.003228 para tamaño de bloque CON 448
Tiempo 0.003182 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Mejorar: 64
Tiempo 0.003578 para tamaño de bloque CON 64
Tiempo 0.003562 para tamaño de bloque CON 128
Tiempo 0.003642 para tamaño de bloque CON 192
Tiempo 0.003565 para tamaño de bloque CON 256
Tiempo 0.003633 para tamaño de bloque CON 320
Tiempo 0.003602 para tamaño de bloque CON 384
Tiempo 0.003601 para tamaño de bloque CON 448
Tiempo 0.003584 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en INCLUIR en Mejorar: 64

*****************
WARM_UP: 240.166813 seg
*****************

Tesla C2075
Tiempo 3.527547 para tamaño de bloque CON 64
Tiempo 3.560318 para tamaño de bloque CON 128
Tiempo 3.583489 para tamaño de bloque CON 192
Tiempo 3.582599 para tamaño de bloque CON 256
Tiempo 5.103173 para tamaño de bloque CON 320
Tiempo 4.448448 para tamaño de bloque CON 384
Tiempo 3.772721 para tamaño de bloque CON 448
Tiempo 3.688367 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en Cálculo de FITNESS: 64
Tiempo 0.003349 para tamaño de bloque CON 64
Tiempo 0.003364 para tamaño de bloque CON 128
Tiempo 0.003376 para tamaño de bloque CON 192
Tiempo 0.003385 para tamaño de bloque CON 256
Tiempo 0.003375 para tamaño de bloque CON 320
```

```
Tiempo 0.003301 para tamaño de bloque CON 384
Tiempo 0.003298 para tamaño de bloque CON 448
Tiempo 0.003318 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Inicializar: 448
Tiempo 0.003006 para tamaño de bloque CON 64
Tiempo 0.002984 para tamaño de bloque CON 128
Tiempo 0.002995 para tamaño de bloque CON 192
Tiempo 0.003000 para tamaño de bloque CON 256
Tiempo 0.003010 para tamaño de bloque CON 320
Tiempo 0.002989 para tamaño de bloque CON 384
Tiempo 0.003009 para tamaño de bloque CON 448
Tiempo 0.003003 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en ROTACION en Inicializar: 128
Tiempo 0.002850 para tamaño de bloque CON 64
Tiempo 0.002912 para tamaño de bloque CON 128
Tiempo 0.002956 para tamaño de bloque CON 192
Tiempo 0.002898 para tamaño de bloque CON 256
Tiempo 0.002870 para tamaño de bloque CON 320
Tiempo 0.002948 para tamaño de bloque CON 384
Tiempo 0.002878 para tamaño de bloque CON 448
Tiempo 0.002864 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en QUAT en Inicializar: 64
Tiempo 0.002886 para tamaño de bloque CON 64
Tiempo 0.002930 para tamaño de bloque CON 128
Tiempo 0.002929 para tamaño de bloque CON 192
Tiempo 0.002953 para tamaño de bloque CON 256
Tiempo 0.002951 para tamaño de bloque CON 320
Tiempo 0.002888 para tamaño de bloque CON 384
Tiempo 0.002969 para tamaño de bloque CON 448
Tiempo 0.002920 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en CURAND en Inicializar: 64
Tiempo 0.003088 para tamaño de bloque CON 64
Tiempo 0.003106 para tamaño de bloque CON 128
Tiempo 0.003064 para tamaño de bloque CON 192
Tiempo 0.003088 para tamaño de bloque CON 256
Tiempo 0.003089 para tamaño de bloque CON 320
Tiempo 0.003112 para tamaño de bloque CON 384
Tiempo 0.003105 para tamaño de bloque CON 448
Tiempo 0.003082 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Mejorar: 64
Tiempo 0.003445 para tamaño de bloque CON 64
Tiempo 0.003428 para tamaño de bloque CON 128
Tiempo 0.003509 para tamaño de bloque CON 192
Tiempo 0.003502 para tamaño de bloque CON 256
Tiempo 0.003480 para tamaño de bloque CON 320
```

```
Tiempo 0.003509 para tamaño de bloque CON 384
Tiempo 0.003449 para tamaño de bloque CON 448
Tiempo 0.003451 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en INCLUIR en Mejorar: 64

*****************
WARM_UP: 295.149048 seg
*****************

Tesla C2075
Tiempo 3.530321 para tamaño de bloque CON 64
Tiempo 3.562289 para tamaño de bloque CON 128
Tiempo 3.596330 para tamaño de bloque CON 192
Tiempo 3.591611 para tamaño de bloque CON 256
Tiempo 5.111317 para tamaño de bloque CON 320
Tiempo 4.437913 para tamaño de bloque CON 384
Tiempo 3.767245 para tamaño de bloque CON 448
Tiempo 3.701093 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en Cálculo de FITNESS: 64
Tiempo 0.003509 para tamaño de bloque CON 64
Tiempo 0.003539 para tamaño de bloque CON 128
Tiempo 0.003487 para tamaño de bloque CON 192
Tiempo 0.003492 para tamaño de bloque CON 256
Tiempo 0.003498 para tamaño de bloque CON 320
Tiempo 0.003428 para tamaño de bloque CON 384
Tiempo 0.003421 para tamaño de bloque CON 448
Tiempo 0.003408 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Inicializar: 512
Tiempo 0.003133 para tamaño de bloque CON 64
Tiempo 0.003114 para tamaño de bloque CON 128
Tiempo 0.003131 para tamaño de bloque CON 192
Tiempo 0.003109 para tamaño de bloque CON 256
Tiempo 0.003120 para tamaño de bloque CON 320
Tiempo 0.003135 para tamaño de bloque CON 384
Tiempo 0.003131 para tamaño de bloque CON 448
Tiempo 0.003092 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en ROTACION en Inicializar: 512
Tiempo 0.002999 para tamaño de bloque CON 64
Tiempo 0.003016 para tamaño de bloque CON 128
Tiempo 0.002987 para tamaño de bloque CON 192
Tiempo 0.003015 para tamaño de bloque CON 256
Tiempo 0.003019 para tamaño de bloque CON 320
Tiempo 0.002977 para tamaño de bloque CON 384
Tiempo 0.002992 para tamaño de bloque CON 448
Tiempo 0.002997 para tamaño de bloque CON 512
```

```
Valor Optimizado HILOS/BLOQUE en QUAT en Inicializar: 384
Tiempo 0.002972 para tamaño de bloque CON 64
Tiempo 0.003025 para tamaño de bloque CON 128
Tiempo 0.003046 para tamaño de bloque CON 192
Tiempo 0.003021 para tamaño de bloque CON 256
Tiempo 0.003021 para tamaño de bloque CON 320
Tiempo 0.003069 para tamaño de bloque CON 384
Tiempo 0.003039 para tamaño de bloque CON 448
Tiempo 0.003001 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en CURAND en Inicializar: 64
Tiempo 0.003190 para tamaño de bloque CON 64
Tiempo 0.003165 para tamaño de bloque CON 128
Tiempo 0.003198 para tamaño de bloque CON 192
Tiempo 0.003262 para tamaño de bloque CON 256
Tiempo 0.003205 para tamaño de bloque CON 320
Tiempo 0.003176 para tamaño de bloque CON 384
Tiempo 0.003168 para tamaño de bloque CON 448
Tiempo 0.003165 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en MOVE en Mejorar: 64
Tiempo 0.003620 para tamaño de bloque CON 64
Tiempo 0.003572 para tamaño de bloque CON 128
Tiempo 0.003614 para tamaño de bloque CON 192
Tiempo 0.003589 para tamaño de bloque CON 256
Tiempo 0.003604 para tamaño de bloque CON 320
Tiempo 0.003620 para tamaño de bloque CON 384
Tiempo 0.003584 para tamaño de bloque CON 448
Tiempo 0.003554 para tamaño de bloque CON 512
Valor Optimizado HILOS/BLOQUE en INCLUIR en Mejorar: 64

******************
WARM_UP: 295.372062 seg
******************
Valores parámetros paralelos:
Device 4, modelo GeForce GTX 590
Número de hilos por bloque función CALCULO DE FITNESS 64
Número de hilos por bloque función MOVE de Inicializar 384
Número de hilos por bloque función ROT de Inicializar 320
Número de hilos por bloque función QUAT de Inicializar 64
Número de hilos por bloque función CURAND de Inicializar 64
Número de hilos por bloque función MOVE de Mejorar 64
Número de hilos por bloque función INCLUIR de Mejorar 64
Device 3, modelo Tesla C2075
Número de hilos por bloque función CALCULO DE FITNESS 64
Número de hilos por bloque función MOVE de Inicializar 384
Número de hilos por bloque función ROT de Inicializar 128
```

```
Número de hilos por bloque función QUAT de Inicializar 512
Número de hilos por bloque función CURAND de Inicializar 512
Número de hilos por bloque función MOVE de Mejorar 512
Número de hilos por bloque función INCLUIR de Mejorar 512
Device 2, modelo GeForce GTX 590
Número de hilos por bloque función CALCULO DE FITNESS 64
Número de hilos por bloque función MOVE de Inicializar 384
Número de hilos por bloque función ROT de Inicializar 448
Número de hilos por bloque función QUAT de Inicializar 448
Número de hilos por bloque función CURAND de Inicializar 64
Número de hilos por bloque función MOVE de Mejorar 64
Número de hilos por bloque función INCLUIR de Mejorar 64
Device 0, modelo GeForce GTX 590
Número de hilos por bloque función CALCULO DE FITNESS 64
Número de hilos por bloque función MOVE de Inicializar 512
Número de hilos por bloque función ROT de Inicializar 320
Número de hilos por bloque función QUAT de Inicializar 128
Número de hilos por bloque función CURAND de Inicializar 64
Número de hilos por bloque función MOVE de Mejorar 64
Número de hilos por bloque función INCLUIR de Mejorar 64
Device 5, modelo GeForce GTX 590
Número de hilos por bloque función CALCULO DE FITNESS 64
Número de hilos por bloque función MOVE de Inicializar 448
Número de hilos por bloque función ROT de Inicializar 128
Número de hilos por bloque función QUAT de Inicializar 64
Número de hilos por bloque función CURAND de Inicializar 64
Número de hilos por bloque función MOVE de Mejorar 64
Número de hilos por bloque función INCLUIR de Mejorar 64
Device 1, modelo Tesla C2075
Número de hilos por bloque función CALCULO DE FITNESS 64
Número de hilos por bloque función MOVE de Inicializar 512
Número de hilos por bloque función ROT de Inicializar 512
Número de hilos por bloque función QUAT de Inicializar 384
Número de hilos por bloque función CURAND de Inicializar 64
Número de hilos por bloque función MOVE de Mejorar 64
Número de hilos por bloque función INCLUIR de Mejorar 64


*****************
TOTAL WARM_UP:  1688.554501 seg
*****************
```