Explotación de Paralelismo Multinivel e Híbrido en Metaheurísticas Híbridas

José M. Cecilia, Baldomero Imbernón
Bioinformatics and High Performance Computing Research Group (BIO-HPC)
Polytechnic School, University Católica San Antonio of Murcia (UCAM)
Murcia, Spain
{jmcecilia,bimbernon}@ucam.edu

Javier Cuenca
Department of Engineering and Technology of Computers
University of Murcia
Murcia, Spain
jcuenca@um.es

José-Matías Cutillas-Lozano, Domingo Giménez
Department of Computing and Systems
University of Murcia
Murcia, Spain
{josematias.cutillas,domingo}@um.es

***Resumen*—Los sistemas computacionales actuales están formados por nodos que comprenden CPUs *multicore* junto con uno o varios coprocesadores (normalmente GPUs pero ocasionalmente también MICs), de forma que se dispone de un sistema híbrido y heterogéneo. Por otro lado, el desarrollo de metaheurísticas híbridas y de hiperheurísticas que trabajan sobre ellas también sigue una estructura híbrida, en la que se puede explotar paralelismo multinivel para llevar a cabo computaciones con distinto volumen de cómputo (heterogéneas). En este trabajo analizamos la combinación de paralelismo híbrido y heterogéneo a los dos niveles, de software y de hardware, en la aplicación de metaheurísticas híbridas. Se realizan experimentos con la aplicación de metaheurísticas a dos problemas, uno de *docking* de moléculas y otro de modelos de autoregresión vectorial, sobre nodos multicore+multiGPU.**

***Index Terms*—metaheurísticas, hiperheurísticas, paralelismo híbrido, paralelismo heterogéneo, docking de moléculas, modelos de autoregresión vectorial**

I. INTRODUCCIÓN

Las metaheurísticas se utilizan para la aproximación de soluciones de problemas de gran dificultad computacional [1]–[3]. Hay variedad de métodos metaheurísticos, básicamente agrupados en dos clases: distribuidos o basados en poblaciones [4], [5], y de búsqueda local, y también métodos híbridos [6], [7] que combinan las características de diferentes metaheurísticas para una mejor adaptación al problema con el que se trabaja. Así, el *software* es híbrido al combinar distintas técnicas, y heterogéneo en el sentido de que cada uno de sus componentes tiene un coste computacional distinto.

Con la aparición de la computación paralela se han desarrollado versiones paralelas de las metaheurísticas, o nuevas metaheurísticas paralelas [8]. Los nodos de la mayoría de los sistemas computacionales actuales están formados por sistemas multinúcleo junto con coprocesadores, que son principalmente GPUs (Graphics Processing Unit)) y MICs (Many Integrated Core), por lo que se está trabajando en la adaptación y optimización de software para este tipo de sistemas híbridos, y, en particular, hay trabajos sobre metaheurísticas para GPU [9], [10]. Además, el *hardware* sobre el que se trabaja es híbrido y heterogéneo, con elementos computacionales con arquitecturas distintas, en distinto número y distinta velocidad, y está organizado en muchos casos de forma jerárquica (un *cluster* con nodos híbridos multicore+GPU, GPUs organizadas en *grids* y bloques...).

Esta situación propicia la línea de trabajo que se presenta: análisis de la combinación de las características híbridas, heterogéneas y jerárquicas de *software* y *hardware* para explotar el paralelismo y obtener beneficios en cuanto a reducir el tiempo de ejecución o mejorar las soluciones obtenidas.

Trabajamos con metaheurísticas híbridas desarrolladas a partir de un esquema parametrizado [11], que combina características de metaheurísticas distribuidas y de búsqueda local, y se ha aplicado a diversidad de problemas: p-hub, asignación de tareas a procesos, y modelado de ecuaciones simultáneas [11]; consumo de electricidad en explotación de pozos de agua [12]; determinación de constantes en ecuaciones cinéticas [13], etc. Además, al estar el esquema parametrizado, se puede utilizar para diseñar hiperheurísticas que buscan en el espacio de los parámetros que determinan las metaheurísticas [14].

El esquema parametrizado se puede ampliar a esquemas paralelos que incluyen parámetros de paralelismo dependiendo del sistema computacional destino (memoria compartida [15] o paso de mensajes [16]), con una paralelización unificada para las distintas metaheurísticas híbridas del esquema.

Analizamos la adaptación del esquema metaheurístico parametrizado a sistemas más complejos: nodos multicore+multiGPU, posiblemente con GPUs de distintas características. Y los problemas a los que se aplican las metaheurísticas tienen una característica que facilita el uso eficiente de la alta capacidad computacional de los coprocesadores: alto coste de cálculo del *fitness*, que puede delegarse a

Subvencionado por MINECO y fondos FEDER de la Comisión Europea, TIN2015-66972-C5-3-R y TIN2016-78799-P (AEI/FEDER, UE).

las GPUs.

Se utilizan dos problemas: *docking* de moléculas y modelado de autoregresión vectorial; y un tercer problema es la búsqueda por medio de hiperheurísticas (programadas como metaheurísticas) de metaheurísticas apropiadas para estos dos problemas básicos. El mayor coste computacional reside en el cálculo del *fitness*, y en las hiperheurísticas el cálculo del *fitness* se hace a su vez con la aplicación de metaheurísticas al problema básico.

El resto del trabajo está estructurado de la siguiente manera. La Sección II muestra el esquema metaheurístico parametrizado que se utiliza, y se comentan las características de las hiperheurísticas sobre este esquema. Las posibilidades de paralelización del esquema para multicore+multiGPU se comentan en la Sección III. Los dos problemas básicos usados como casos de prueba se describen en la Sección IV, y la Sección V muestra algunos resultados experimentales en la aplicación a los dos problemas básicos y a hiperheurísticas para el problema del *docking*. Las conclusiones y posibles trabajos futuros se resumen en la Sección VI.

II. Esquema Parametrizado de Metaheurísticas e Hiperheurísticas

Resumimos las ideas generales de los esquemas metaheurísticos parametrizados usados en este trabajo. Se pueden encontrar más detalles del esquema, de su aplicación a varios problemas de optimización y un estudio estadístico de la influencia de los parámetros en [11]. El esquema se muestra en el Algoritmo 1. Incluye Parámetros Metaheurísticos en cada una de las funciones básicas. No es un esquema general que incluya todas las posibles metaheurísticas (incontables) sino que depende de la implementación de las funciones básicas y de los parámetros que se incluyan en ellas. Instancias concretas de los parámetros pueden dar lugar a metaheurísticas básicas o a hibridaciones suyas. Por ejemplo, en [11] se implementan versiones de Algoritmos Genéticos, Búsqueda Dispersa y GRASP (Greedy Randomized Adaptive Search Procedure), y otras de las referencias mencionadas incluyen también Búsqueda Tabú. La mayoría de los parámetros son generales para el esquema, pero puede haber algunos propios del problema al que se aplican las metaheurísticas. Una descripción detallada se encuentra en las referencias previas.

Las hiperheurísticas que consideramos son a la vez metaheurísticas con el mismo esquema parametrizado, pero el significado de los parámetros es distinto, ya que depende del problema al que se aplican, que en este caso es el de búsqueda de metaheurísticas híbridas satisfactorias para el problema básico considerado. El esquema metaheurístico para implementar hiperheurísticas se muestra en el Algoritmo 2. Es igual al del Algoritmo 1 con la única diferencia los elementos con los que se trabaja (ahora son vectores de parámetros metaheurísticos) y el *fitness* asociado a cada elemento (metaheurística), que se obtiene con la aplicación de la metaheurística representada por el vector de parámetros metaheurísticos a uno o varios problemas de entrenamiento. Algunas formas de calcular el *fitness* se discuten en [14].

III. Esquemas Paralelos para multicore+multiGPU

Hemos desarrollado esquemas paralelos para memoria compartida [15] y paso de mensajes [16], y hemos analizado la utilización de esquemas parametrizados en la resolución de problemas particulares en nodos con coprocesadores. Por ejemplo, en [17] se analiza la aplicación de hiperheurísticas al problema de *docking* en un MIC. El esquema paralelo es el de memoria compartida, pero los MIC pueden ser considerados *manycore* más que multicore al ser el número de procesos mucho más elevado, y esto hace que la explotación del paralelismo a varios niveles (en la hiperheurística y las metaheurísticas donde ella busca) sea más adecuado para estos sistemas.

En el caso de la explotación de paralelismo de GPUs el esquema paralelo es distinto de los dos anteriores, pues la programación en GPUs sigue un modelo SIMD. El desarrollo de metaheurísticas para GPUs es un campo de investigación actual [9], [10], y hemos utilizado el esquema parametrizado para el problema del *docking* en nodos con GPUs, ya sean locales o virtualizadas [18], lo que acelera la computación y facilita encontrar soluciones mejores en menor tiempo.

En este trabajo analizamos por primera vez el diseño de metaheurísticas parametrizadas de forma general para explotar toda la capacidad computacional que proporcionan los nodos multicore+multiGPU. Para analizar el problema de forma general consideramos dos problemas, que además tienen alto coste computacional, lo que es necesario para que la explotación eficiente de los coprocesadores muestre todo su potencial.

Las metaheurísticas y las hiperheurísticas que trabajan sobre ellas siguen el mismo esquema parametrizado (algoritmos 1 y 2), con diferencias en la implementación de las funciones y los parámetros, dependiendo del problema subyacente a que se aplican. Así, la versión de memoria compartida es común a metaheurísticas e hyperheurísticas (Algoritmo 3), y consiste en la paralelización independiente de las funciones básicas del esquema, con inclusión de parámetros de paralelismo que determinan el número de hilos a usar en cada función. Hay funciones donde se tratan varios elementos dentro de un bucle, y se paraleliza el bucle estableciendo un determinado número de hilos. En las funciones de mejora se tratan varios elementos, en un bucle que se paraleliza con un cierto número de hilos, y para cada elemento se analiza su vecindad, lo que se hace con otro bucle con un número de hilos diferente en un segundo nivel de paralelismo. Como las hiperheurísticas llaman a metaheurísticas para el cálculo del *fitness*, llegamos a tener en algunos casos hasta cuatro niveles de paralelismo, y se podría determinar en número de hilos en cada nivel para la mayor reducción del tiempo de ejecución teniendo en cuenta las características del sistema computacional [17].

En sistemas con paralelismo híbrido multicore+multiGPU el código correrá en la CPU, que delegará a GPU partes con alto coste computacional. El paralelismo de GPU puede explotarse a distintos niveles de entre los que aparecen en el esquema de memoria compartida. Los datos con los que trabajar en GPU se

Algorithm 1 Esquema metaheurístico parametrizado

```
Initialize(S,ParamIni) //Generación del conjunto inicial, posiblemente con mejora de elementos
while (not EndCondition(S,ParamEnd)) do
  SS=Select(S,ParamSel) //Selección de elementos para combinación
  SS1=Combine(SS,ParamCom) //Combinación de pares de elementos
  SS2=Improve(S,SS1,ParamImp) //Mejora de algunos elementos, posiblemente con diversificación
  S=Include(S,SS1,SS2,ParamInc) //Actualización del conjunto de referencia
end while
```

Algorithm 2 Esquema metaheurístico parametrizado para la implementación de hiperheurísticas

```
Initialize(S,ParamIni) //Generación de un conjunto inicial de metaheurísticas
while (not EndCondition(S,ParamEnd)) do
  SS=Select(S,ParamSel) //Selección de metaheurísticas para combinación
  SS1=Combine(SS,ParamCom) //Combinación de pares de metaheurísticas para obtener nuevas metaheurísticas
  SS2=Improve(S,SS1,ParamImp) //Mejora y diversificación de metaheurísticas para obtener otras más apropiadas
    para el problema con que se trabaja
  S=Include(S,SS1,SS2,ParamInc) //Actualización del conjunto de metaheurísticas
end while
```

Algorithm 3 Versión de Memoria Compartida del esquema metaheurístico parametrizado

```
Initialize(S,ParamIni,HilosIni) //Número de hilos en la generación, y número de hilos sobre los elementos a
  mejorar y un segundo nivel para hilos que analizan la vecindad
while (not EndCondition(S,ParamEnd)) do
  SS=Select(S,ParamSel,HilosSel) //Número de hilos en bucle de selección
  SS1=Combine(SS,ParamCom,HilosCom) //Número de hilos en bucle de combinación
  SS2=Improve(S,SS1,ParamImp,HilosImp) //Número de hilos sobre los elementos a mejorar o diversificar, y un
    segundo nivel para hilos que analizan la vecindad
  S=Include(S,SS1,SS2,ParamInc,HilosInc) //Número de hilos en bucles para tratar los elementos a incluir
end while
```

transfieren de CPU a GPU, y los resultados a la inversa, y las transferencias CPU-GPU pueden ser costosas, por lo que habrá que realizar implementaciones donde se solapen computación y transferencias, y además el volumen de trabajo a realizar en GPU en cada llamada debe ser grande. Algunos niveles a los que pueden trabajar las GPUs en el esquema del Algoritmo 3 son:

- El mayor nivel corresponde al esquema de islas, con una isla por GPU en un sistema multicore+multiGPU, dividiendo los conjuntos en subconjuntos y asignando cada subconjunto a una GPU. El trabajo que se delega a las GPU es de propsito general y no sigue el modelo SIMD, que es el más adecuado para obtener buenas prestaciones en GPU.
- Si bajamos a nivel de paralelización dentro de cada función básica, el esquema se ejecuta en CPU y las funciones trabajan sobre elementos por medio de bucles, con cada iteración del bucle realizada por un hilo. Cada hilo puede tener asociada una GPU a la que delega el tratamiento del elemento, que se transfiere a la GPU, que devuelve a CPU el resultado. Las funciones básicas tienen costes distintos, por lo que habrá que determinar para cada una, dependiendo de cómo esté implementada, si es conveniente delegar el trabajo a GPU o realizarlo en CPU. Además, algunas funciones conllevan comparaciones y bifurcaciones en el análisis de la vecindad, por lo que, de nuevo, no tenemos un modelo SIMD.
- En un siguiente nivel las GPUs se utilizarían para el cálculo del *fitness*. Como en algunas funciones hay que calcular el *fitness* de los objetos obtenidos (en la generación inicial, al combinar, al analizar vecindades en las funciones de mejora), se pueden agrupar los cálculos de los *fitness* de todos los elementos, que se realizarían de nuevo en un bucle, con cada hilo delegando el cálculo de un elemento a la GPU que tiene asociada. De esta forma el trabajo que realizan las GPUs puede seguir el modelo SIMD si el cálculo del *fitness* se realiza a través de funciones matriciales, como es el caso de los problemas con los que experimentamos en este trabajo.
- En un último nivel serían todas las GPUs las que colaboraran en el cálculo del *fitness* de cada individuo. Podría ser una buena opción cuando este cálculo tenga un alto coste computacional, lo que puede ocurrir con las hiperheurísticas, donde el cálculo conlleva la aplicación completa de una metaheurística a una instancia del problema básico.

La mejor opción puede ser una combinación de las dos versiones intermedias, determinando el nivel al que trabajan las

GPU dependiendo del coste de la rutina básica y del número de elementos que se trate en cada nivel. El Algoritmo 4 muestra los puntos en los que se pueden usar las GPUs. Para cada rutina se indica con CPU o GPU el componente donde se ejecuta. Las rutinas sobre CPU usan paralelismo OpenMP. En las rutinas de mejora hay que determinar en cuál de los dos niveles de paralelismo es preferible utilizar las GPUs.

Algorithm 4 Asignación de trabajo a CPU y GPU en un esquema parametrizado de metaheurísticas para multicore+multiGPU

```
InitializeCPU(Sini,ParamIni)
ComputefitnessGPU(Sini,ParamIni)
ImproveGPU(Sini,Sref,ParamImpIni) //2 posibles niveles
while not EndConditionCPU(Sref,ParamEndCon) do
   SelectCPU(Sref,Ssel,ParamSel)
   CombineCPU(Ssel,Scom,ParamCom)
   ComputefitnessGPU(Scom,ParamCom)
   DiversifyCPU(Sref,Scom,Sdiv,ParamDiv)
   ComputefitnessGPU(Sdiv,ParamDiv)
   ImproveGPU(Scom,Sdiv,ParamImp) //2 posibles niveles
   IncludeCPU(Scom,Sdiv,Sref,ParamInc)
end while
```

IV. Problemas Básicos

Se comentan las características generales de los dos problemas de optimización que se han utilizado en los experimentos.

IV-A. Docking de Moléculas

Se dispone de una molécula de gran dimensión (receptor) y otra menor (ligando), y se quiere encontrar la mejor forma en que el ligando se acopla con el receptor, de acuerdo con una función de *scoring*. En la bibliografía se encuentran varias funciones que se pueden usar [19], y en ellas se mide el acoplamiento considerando cada átomo en el receptor y el ligando, con lo que son funciones de alto coste y con un esquema de cómputo regular, que es apropiado para ser utilizado en GPU. Cada posición del ligando viene determinada por tres coordenadas para indicar la posición en el espacio de un átomo de referencia, y otras tres para indicar los giros con respecto a esta referencia; además, el ligando puede tener puntos donde se puede flexionar. Así, cada elemento viene determinado por las variables que indican la posición del ligando. Además, hay varias regiones independientes (*spots*) en la superficie del receptor donde el ligando puede acoplarse, por lo que la búsqueda en los *spots* se realiza de forma idependiente, lo que significa que las metaheurísticas realizan la búsqueda con poblaciones independientes para cada *spot*. Teniendo en cuenta que el número de *spots* puede ser de unos cientos, y el número de átomos de receptor y ligando de unos miles y unos cientos, el coste computacional de la aplicación de metaheurísticas a este problema es alto, y es conveniente la utilización de sistemas de altas prestaciones del tipo multicore+multiGPU, posiblemente con GPUs virtualizadas para disponer de mayor potencia computacional [18].

IV-B. Modelos de Autoregresión Vectoriales

Consideramos d parámetros de los que se han tomado valores en t instantes de tiempo. Los valores para el instante k, $1 \leq k \leq t$, se almacenan en un vector $y^{(k)} = \left(y_1^{(k)}, \ldots, y_d^{(k)}\right) \in R^{1\times d}$. Y los t vectores de datos se pueden organizar en una matriz $Y \in R^{t\times d}$:

$$\begin{pmatrix} y_1^{(1)} & \ldots & y_d^{(1)} \\ \vdots & \ddots & \vdots \\ y_1^{(t)} & \ldots & y_d^{(t)} \end{pmatrix} \tag{1}$$

Consideramos también dependencias temporales con i instantes de tiempo anteriores. Puede haber variables externas al modelo, vector z, con e variables. La dependencia de vectores de datos de un instante j en función de los vectores de datos de instantes anteriores (hasta i para las variables internas y k para las externas) se puede representar como

$$y^{(j)} \approx y^{(j-1)}A_1 + y^{(j-2)}A_2 + \ldots + y^{(j-i)}A_i + z^{(j-1)}B_1 + z^{(j-2)}B_2 + \ldots + z^{(j-k)}B_k + a \tag{2}$$

donde A_l, $1 \leq l \leq i$, son matrices de dimensión $d \times d$ que representan la dependencia de los datos con los valores de l instantes anteriores, B_l, $1 \leq l \leq k$, son matrices de tamaño $e \times d$, y a es un vector de términos independientes.

Cada elemento del espacio de búsqueda de una metaheurística viene dado por los valores de las matrices A, B y a, y el *fitness* se obtiene con multiplicaciones matriciales y la norma de la diferencia entre los valores de la serie temporal (ecuación 1) y los que se obtienen en cada instante de tiempo utilizando los instantes anteriores y dichas matrices (ecuación 2) [20]. Así, también ahora las operaciones son apropiadas para el modelo SIMD, pero las dimensiones (d, e, i, k) son bastante menores que en el problema anterior.

V. Resultados Experimentales

Se resumen los resultados de los experimentos realizados en nodos computacionales con CPUs y GPUs de distintos tipos y uno de ellos con varias GPUS:

- **marte**: contiene un hexa-core CPU AMD Phenom II X6 1075T a 3.00 GHz, con arquitectura x86-64, 16 GB de RAM, cachés privadas L1 y L2 de 64 KB y 512 KB, y una caché L3 de 6 MB compartida por todos los núcleos. Incluye una GPU NVidia GeForce GTX 480 (Fermi) con 1536 MB de memoria global y 480 núcleos CUDA (15 Streaming Multiprocessors y 32 Streaming Processors por SM).
- **saturno**: tiene 4 CPU Intel Xeon E7530 (hexa-core) a 1.87 GHz, con arquitectura x86-64, 32 GB de RAM, cachés privadas L1 y L2 de 32 KB y 256 KB por núcleo, y una caché L3 de 12 MB compartida por los núcleos de cada socket. Incluye una GPU NVidia Tesla K20c (Kepler) con 4800 MB de memoria global y 2496 núcleos CUDA (13 Streaming Multiprocessors y 192 Streaming Processors por SM).

- **jupiter**: con 2 CPU Intel Xeon E5-2620 (hexa-core) a 2.00 GHz, con arquitectura x86-64, 32 GB de memoria RAM, cachés privadas L1 y L2 de 32 KB y 256 KB por núcleo, y una caché L3 de 15 MB compartida por los núcleos en cada socket. Incluye 6 GPUs: 2 NVidia Tesla C2075 (Fermi) con 5375 MB de memoria global y 448 núcleos CUDA (14 Streaming Multiprocessors y 32 Streaming Processors per SM), y 4 GPUs en dos placas, cada una es una con 2 NVidia GeForce GTX 590 (Fermi) con 1536 MB de memoria global y 512 núcleos CUDA (16 Streaming Multiprocessors y 32 Streaming Processors por SM).
- **venus**: con 2 CPU Intel Xeon E5-2620 (hexa-core) a 2.40 GHz, arquitectura x86-64, 64 GB de memoria RAM, cachés privadas L1 y L2 de 32 KB y 256 KB por núcleo, y una caché L3 de 15 MB compartida por todos los núcleos de un socket. Incluye una GPU NVidia GeForce GT 640 (Kepler) con 1024 MB de Global Memory y 384 núcleos CUDA (2 Streaming Multiprocessors y 192 Streaming Processors por SM).

La Tabla I compara los tiempos obtenidos en **jupiter** con paralelismo GPU (sólo se usa una Tesla C2075) y con CPU con un núcleo, para seis metaheurísticas híbridas aplicadas al problema de *docking* de moléculas con el par receptor-ligando COMT (se pueden consultar los detalles en [21]). Se alcanza un speed-up de GPU con respecto a CPU de alrededor de 43. La ventaja de usar GPUs es clara, incluso en este caso en que la GPU se usa sólo en el cálculo de la función de *scoring*.

Tabla I
TIEMPO DE EJECUCIÓN (EN SEGUNDOS) DE VERSIONES CPU Y GPU, Y SPEED-UP DE GPU RESPECTO A CPU, PARA DIFERENTES METAHEURÍSTICAS APLICADAS AL PAR RECEPTOR-LIGANDO COMT, EN **JUPITER**

Metaheuristic	CPU	GPU	CPU/GPU
M1	140.48	10.42	39.35
M2	193.67	13.81	43.55
M3	1,911.52	9.62	54.16
M4	209.56	9.59	34.43
M5	262.65	8.55	35.51
M6	1,379.93	10.39	53.89

El par con el que se ha experimentado para los resultados de la Tabla I es pequeño pero los tiempos de ejecución pueden ser grandes dependiendo de la metaheurística. En el caso de las hiperheurísticas, el tiempo crece enormemente dado que se busca en un espacio de metaheurísticas. En una hiperheurística se puede tener hasta cuatro niveles de paralelismo (dos en la hiperheurística y hasta dos en la metaheurística), por lo que es necesario experimentar con las distintas combinaciones del número de hilos en los cuatro niveles para obtener la mejor combinación. Para simplificar la experimentación dado que el tiempo de cada experimento es bastante largo, hemos considerado los tres niveles de paralelismo de mayor nivel, con lo que las metaheurísticas usan sólo paralelismo de un nivel. La Figura 1 compara el tiempo de ejecución (segundos) con la versión de memoria compartida para una hiperheurística particular, con diferentes combinaciones de número de hilos en tres niveles de paralelismo, con un total de seis hilos y en los cuatro nodos considerados. Se obtienen resultados similares con otras hiperheurísticas. La mejor combinación de hilos es $1 \times 1 \times 6$ en todos los nodos, con algunas diferencias en la comparación de las distintas combinaciones dependiendo del nodo, lo que es normal dada la distinta capacidad computacional de los nodos y que hay aleatoriedad en las ejecuciones. Para evitar la aleatoriedad se deberían realizar varias ejecuciones por cada combinación, pero, dado el alto coste computacional, esto conllevaría unos tiempos de experimentación excesivos.

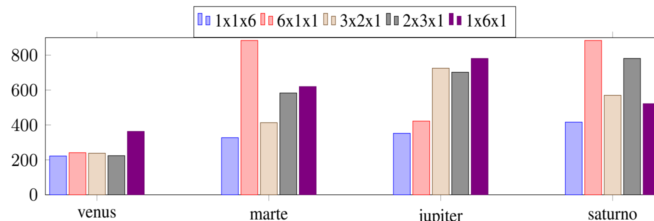

Figura 1. Tiempo de ejecución (segundos) con la versión de memoria compartida para una hiperheurística, con diferentes combinaciones de número de hilos en tres niveles de paralelismo, con un total de seis hilos y en los cuatro nodos considerados.

Tabla II
COMPARACIÓN DEL TIEMPO DE EJECUCIÓN (EN SEGUNDOS) DE TRES HIPERHEURÍSTICAS CON PARALELISMO DE MEMORIA COMPARTIDA EN CPU Y CON PARALELISMO GPU, EN LOS CUATRO NODOS CONSIDERADOS.

	H1	H2	H3
	marte		
multicore	13035	4541	23957
multicore+GPU	903	335	1348
CPU/GPU	14.44	13.56	17.77
	venus		
multicore	4021	1423	6346
multicore+GPU	3197	1857	5065
CPU/GPU	1.26	0.77	9.67
	jupiter		
multicore	8045	3509	14115
multicore+GPU	990	514	1459
CPU/GPU	8.13	6.83	9.67
	saturno		
multicore	6150	1595	8582
multicore+GPU	563	200	823
CPU/GPU	10.92	7.98	10.43

La Tabla II compara el tiempo de ejecución de tres hiperheurísticas cuando se explota paralelismo de memoria compartida con un número de hilos igual al número de cores en el nodo y cuando se usa GPU para el cálculo de las funciones de *scoring*. Se ha usado de nuevo el par COMT. Se comprueba que el tiempo de ejecución es mucho mayor que el de una única metaheurística, y el tiempo de ejecución con la explotación del paralelismo híbrido multicore+GPU es en general mucho menor que con paralelismo de memoria compartida. Sólo en el caso de **venus**, que tiene la CPU más rápida y la GPU más lenta, son los tiempos comparables.

El uso de GPU también puede dar resultados satisfactorios para el problema de modelos de autoregresión vectorial, pero

en este caso el coste computacional es mucho menor que en el *docking* de moléculas, por lo que el uso de GPU resulta ventajoso sólo para problemas muy grandes que seguramente no se corresponden con casos de aplicaciones reales. Por ejemplo, en **jupiter**, cuando se utiliza una única GPU en el nivel más bajo (cálculo del *fitness*, que en este caso es una multiplicación de matrices y el cálculo de la norma), con valores de los parámetros $t = 200$, $d = 4$, $e = 2$, $i = 3$ y $k = 2$ el speed-up de GPU respecto a CPU es 0.83, con lo que es preferible no usar GPU. Pero para problemas mayores el speed-up aumenta: para $d = 5$, $e = 3$, $i = 3$ y $k = 2$ es 0.92, y para $d = 5$, $e = 3$, $t_y = 4$ y $t_z = 2$ llega a 1.25.

El uso de más GPUs reduce el tiempo de ejecución sólo para tamaños muy grandes, por lo que para este problema es necesario optimizar más la utilización de GPUs. Un ejemplo es la asignación dinámica de trabajo. En una ejecución en **jupiter** con cuatro GPUs (dos de cada tipo) y tamaños mayores que en el experimento anterior ($t = 500$), el tiempo de ejecución fue 1854 segundos cuando se asignó el cálculo del *fitness* a las GPUs con el mismo número de elementos para cada GPU. Al asignar de forma estática el doble de elementos a las GPUs más rápidas (en teoría son el doble de rápidas) el tiempo subió a 2249 segundos. Y el menor tiempo se obtuvo con asignación dinámica, con 1598 segundos.

VI. Conclusiones y Trabajos Futuros

En este trabajo se discute la utilización de paralelismo híbrido multicore+multiGPU en la aplicación de metaheurísticas híbridas basadas en un esquema parametrizado de metaheurísticas. En el esquema hay paralelismo en distintos niveles, y se analiza en qué niveles sería conveniente la utilización de GPUs. Además, sobre el esquema se pueden desarrollar hiperheurísticas que se implementan como metaheurísticas con ese esquema y que buscan en el espacio de metaheurísticas para un problema básico implementadas con el mismo esquema. Como el esquema paralelo parametrizado tiene dos niveles de paralelismo, la implementación de hiperheurísticas da lugar a un máximo de cuatro niveles, y se debe analizar la mejor combinación de hilos en cada nivel y en qué partes del esquema paralelo utilizar GPU.

Se han considerado dos problemas básicos para analizar el esquema: *docking* de moléculas y modelos de autoregresión vectorial. Los dos problemas tienen alto coste computacional y el cálculo del *fitness* se hace con computación matricial que sigue un esquema SIMD, por lo que se puede explotar paralelismo GPU. Para el primer problema, dado el tamaño de las moléculas con las que se realiza el *docking*, el tiempo de ejecución es muy elevado, y se alcanza una gran reducción del tiempo con respecto a CPU cuando se utiliza paralelismo híbrido multicore+GPU. En el otro problema, el uso de GPU es ventajoso sólo para problemas grandes, que no se corresponden con aplicaciones en uso.

El mismo esquema puede ser utilizado en otros problemas, y estamos centrando nuestra investigación en el análisis de cómo determinar en función de los costes del problema de optimización la forma óptima de utilizar el esquema: número de hilos y forma de uso de las GPUs en cada nivel.

Referencias

[1] Z. Michalewicz and D. B. Fogel, *How to Solve It: Modern Heuristics*. Springer, 2002.

[2] J. Hromkovič, *Algorithmics for Hard Problems*. Springer, second ed., 2003.

[3] J. Dréo, A. Pétrowski, P. Siarry, and E. Taillard, *Metaheuristics for Hard Optimization*. Springer, 2005.

[4] F. Glover and G. A. Kochenberger, *Handbook of Metaheuristics*. Kluwer, 2003.

[5] E.-G. Talbi, *Metaheuristics - From Design to Implementation*. New York: Wiley, 2009.

[6] G. R. Raidl, "A unified view on hybrid metaheuristics," in *Hybrid Metaheuristics, Third International Workshop, LNCS*, vol. 4030, pp. 1–12, October 2006.

[7] C. Blum, J. Puchinger, G. R. Raidl, and A. Roli, "Hybrid metaheuristics in combinatorial optimization: A survey," *Appl. Soft Comput.*, vol. 11, no. 6, pp. 4135–4151, 2011.

[8] E. Alba, *Parallel Metaheuristics: A New Class of Algorithms*. New York: Wiley-Interscience, 2005.

[9] E.-G. Talbi and G. Hasle, "Metaheuristics on GPUs," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 1–3, 2013.

[10] M. Essaid, L. Idoumghar, J. Lepagnot, and M. Brévilliers, "GPU parallelization strategies for metaheuristics: a survey," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 0, no. 0, pp. 1–26, 2018.

[11] F. Almeida, D. Giménez, J.-J. López-Espín, and M. Pérez-Pérez, "Parameterised schemes of metaheuristics: basic ideas and applications with Genetic algorithms, Scatter Search and GRASP," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 43, no. 3, pp. 570–586, 2013.

[12] L.-G. Cutillas-Lozano, J.-M. Cutillas-Lozano, and D. Giménez, "Modeling shared-memory metaheuristic schemes for electricity consumption," in *Distributed Computing and Artificial Intelligence - 9th International Conference*, pp. 33–40, 2012.

[13] J. Cutillas-Lozano and D. Giménez, "Determination of the kinetic constants of a chemical reaction in heterogeneous phase using parameterized metaheuristics," in *Proceedings of the International Conference on Computational Science*, pp. 787–796, 2013.

[14] J. Cutillas-Lozano, D. Giménez, and F. Almeida, "Hyperheuristics based on parametrized metaheuristic schemes," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 361–368, 2015.

[15] F. Almeida, D. Giménez, and J.-J. López-Espín, "A parameterized shared-memory scheme for parameterized metaheuristics," *The Journal of Supercomputing*, vol. 58, no. 3, pp. 292–301, 2011.

[16] J. Cutillas-Lozano and D. Giménez, "Optimizing a parameterized message-passing metaheuristic scheme on a heterogeneous cluster," *Soft Comput.*, vol. 21, no. 19, pp. 5557–5572, 2017.

[17] J. M. Cecilia, J. Cutillas-Lozano, D. Giménez, and B. Imbernón, "Exploiting multilevel parallelism on a many-core system for the application of hyperheuristics to a molecular docking problem," *The Journal of Supercomputing*, vol. 74, no. 5, pp. 1803–1814, 2018.

[18] B. Imbernón, J. Prades, D. Giménez, J. M. Cecilia, and F. Silla, "Enhancing large-scale docking simulation on heterogeneous systems: An MPI vs rcuda study," *Future Generation Comp. Syst.*, vol. 79, pp. 26–37, 2018.

[19] G. Schneider, "Virtual screening and fast automated docking methods," *Drug Discovery Today*, vol. 7, pp. 64–70, Jan. 2002.

[20] A. L. Castaño, J. Cuenca, J.-M. Cutillas-Lozano, D. Giménez, J. J. López-Espín, and A. Pérez-Bernabeu, "Parallelism on hybrid metaheuristics for vector autoregression models," in *International Conference on High Performance Computing & Simulation*, August 2018.

[21] B. Imbernón, J. M. Cecilia, and D. Giménez, "Enhancing metaheuristic-based virtual screening methods on massively parallel and heterogeneous systems," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 50–58, 2016.