UNIVERSIDAD DE MURCIA

Escuela Internacional de Doctorado

Técnicas de Optimización de Rutinas Paralelas de Álgebra Lineal en Sistemas Heterogéneos

D. Jesús Cámara Moreno

2020

A Elena, por estar siempre a mi lado.

Agradecimientos

Aún recuerdo el día en que Domingo y Javier me ofrecieron la oportunidad de realizar esta tesis doctoral, un momento que, sin duda, marcó un antes y un después en mi carrera profesional. Gracias a vosotros, durante estos tres años de formación predoctoral, he ido adentrándome poco a poco en el apasionante mundo de la investigación y he adquirido las destrezas y el conocimiento necesario para poder llevar a cabo este trabajo. Sin vosotros, esto no hubiese sido posible.

Gracias por vuestros consejos, vuestro saber hacer, por haber estado siempre dispuestos a ofrecerme cuanto he necesitado y, cómo no, por la inmensa paciencia que habéis tenido para guiarme y formarme durante la realización de esta tesis doctoral. Siempre estáis ahí cuando se os necesita, y os agradezco vuestro esfuerzo. Además de excelentes directores de tesis, sois unos auténticos profesionales en vuestro trabajo y esos son los valores que marcan la diferencia.

También quiero dedicar unas palabras a la mujer que, sin duda alguna, está siempre a mi lado apoyándome en todo. Gracias a ti, Elena, por confiar en mí, por hacerme crecer cada día y por haber aceptado en todo momento cualquier situación que hayamos tenido que afrontar para que pudiese culminar esta tesis de la mejor manera posible. Por último, no me quiero olvidar de todas las personas (profesores, investigadores, compañeros,...) que me han aportado su granito de arena en este camino para que pueda alcanzar el grado de Doctor en Informática.

A todos vosotros,

¡Gracias!

Índice general

1 Introducción 11

1.1 Contexto . . . 11

1.2 Estado del arte . . . 13

1.3 Objetivos . . . 18

1.4 Metodología . . . 22

1.5 Contribuciones . . . 23

1.6 Estructura de la tesis . . . 23

2 Herramientas computacionales 25

2.1 Herramientas hardware . . . 25

2.2 Herramientas software . . . 34

2.2.1 Librerías de álgebra lineal . . . 34

2.2.2 Rutinas de álgebra lineal . . . 36

2.2.2.1 Multiplicación de matrices . . . 36

2.2.2.2 Multiplicación de Strassen . . . 39

2.2.2.3 Factorización LU . . . 43

2.3 Conclusiones . . . 44

3 Técnicas de optimización y auto-optimización 47

3.1 Ideas generales . . . 47

3.2 Trabajos relacionados . . . 48

3.3 Clasificación general de las técnicas de optimización automática . 55

3.4 Optimización automática basada en el modelado del tiempo de ejecución . . . 57

3.5 Optimización automática híbrida: modelo y experimentación . . . 59
3.6 Optimización automática experimental 64
3.7 Conclusiones 67

4 Metodología de optimización jerárquica 69
4.1 Nociones generales de optimización jerárquica 70
4.2 Estructura de optimización multinivel 71
4.2.1 Jerarquía bidimensional: hardware × software 78
4.3 Multiplicación de matrices en multicore 84
4.4 Extensión en la jerarquía hardware: multiplicación de matrices en nodo CPU+GPU 86
4.5 Extensión en la jerarquía software 89
4.5.1 Multiplicación de Strassen 89
4.5.2 Factorización LU 91
4.6 Conclusiones 94

5 Ejemplos de aplicación de la metodología 97
5.1 Instalación de rutinas en elementos computacionales básicos . . . 98
5.1.1 Instalación de la multiplicación de matrices en un elemento computacional 98
5.1.2 Instalación de la multiplicación de Strassen en un elemento computacional 107
5.2 Instalación a nivel de nodo 115
5.2.1 Instalación de la multiplicación de matrices en un nodo . . 116
5.2.2 Instalación de la multiplicación de Strassen en un nodo . . 131
5.2.3 Utilización de la multiplicación de matrices híbrida en una factorización LU 134
5.3 Instalación en nivel 2 de hardware 137
5.3.1 Multiplicación de matrices en un cluster 138
5.3.2 Multiplicación de Strassen usando nodos virtuales 138
5.4 Uso de herramientas software para la obtención del valor de los parámetros algorítmicos 142
5.5 Conclusiones 147

6 Extensión de la metodología a librerías basadas en tareas con asignación dinámica 149

6.1 Extensión de la metodología a otras librerías numéricas 150

6.2 Estructura general de Chameleon 151

6.2.1 Uso de simulación 152

6.3 Optimización de Rutinas 154

6.3.1 Descomposición de Cholesky 154

6.3.2 Factorización LU 157

6.3.3 Uso de la metodología de optimización 160

6.3.4 Selección de las unidades de cómputo 168

6.3.5 Selección de la política de planificación 169

6.4 Conclusiones 173

7 Conclusiones y Trabajo Futuro 175

7.1 Conclusiones 175

7.2 Difusión de Resultados 178

7.3 Trabajo Futuro 181

Bibliografía 187

A Software de Instalación y Auto-Optimización Jerárquica 211

A.1 Introducción 211

A.2 Estructura de la Aplicación 213

A.3 Manual de Usuario 215

A.4 Manual de Desarrollador 221

A.4.1 Modificación de scripts 221

A.4.2 Modificación de código fuente 222

Índice de tablas

3.1 Tiempos de ejecución (en segundos) obtenidos con la rutina de multiplicación de matrices generando el número óptimo de threads (MKL-ORA); generando tantos threads MKL como cores de la plataforma (MKL-AC); activando la selección dinámica de threads MKL (MKL-DYN) y utilizando la multiplicación de dos niveles con la configuración de threads (entre paréntesis) seleccionada por el sistema de auto-tuning (MKL-ATS). 59

3.2 Tiempos de ejecución (en segundos) obtenidos con la rutina de factorización LU por bloques generando el número óptimo de threads (MKL-ORA); generando tantos threads MKL como cores de la plataforma (MKL-AC); activando la selección dinámica de threads MKL (MKL-DYN) y utilizando la multiplicación de dos niveles con la configuración de threads (entre paréntesis) seleccionada por el sistema de auto-tuning (MKL-ATS). 60

3.3 Tiempos de ejecución (en segundos) obtenidos en **hipatia** con la rutina de descomposición de Cholesky de PLASMA al utilizar tantos threads como cores y el tamaño de bloque por defecto, y utilizando el número de threads y el tamaño de bloque externo seleccionado con las técnicas de modelado empírico Mod-LS y Mod-NNLS, así como la desviación experimentada en el tiempo de ejecución con cada una respecto al valor óptimo. 62

3.4 Tiempos de ejecución (en segundos) obtenidos en **saturno** con la rutina de descomposición de Cholesky de PLASMA al utilizar tantos threads como cores y el tamaño de bloque por defecto, y utilizando el número de threads y el tamaño de bloque externo seleccionado con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`, así como la desviación experimentada en el tiempo de ejecución con cada una respecto al valor óptimo. 63
3.5 Tiempos de ejecución (en segundos) obtenidos en **joule** con la rutina de descomposición de Cholesky de PLASMA al utilizar tantos threads como cores y el tamaño de bloque por defecto, y utilizando el número de threads y el tamaño de bloque externo seleccionado con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`, así como la desviación experimentada en el tiempo de ejecución con cada una respecto al valor óptimo. 63
3.6 Tiempos de ejecución (en segundos) obtenidos en **hipatia** con la rutina QR de PLASMA al usar el número de threads y el tamaño de bloque externo e interno seleccionado con las técnicas de modelado empírico (`Mod-LS` y `Mod-NNLS`), y desviación producida en el tiempo de ejecución con cada una respecto al óptimo. 64
3.7 Tiempos de ejecución (en segundos) obtenidos en **saturno** con la rutina QR de PLASMA al usar el número de threads y el tamaño de bloque externo e interno seleccionado con las técnicas de modelado empírico (`Mod-LS` y `Mod-NNLS`), y desviación producida en el tiempo de ejecución con cada una respecto al óptimo. 64
3.8 Tiempos de ejecución (en segundos) obtenidos en **joule** con la rutina QR de PLASMA al usar el número de threads y el tamaño de bloque externo e interno seleccionado con las técnicas de modelado empírico (`Mod-LS` y `Mod-NNLS`), y desviación producida en el tiempo de ejecución con cada una respecto al óptimo. 65
3.9 Comparación del tiempo de ejecución (en segundos) obtenido en **hipatia** con la rutina LU de PLASMA, con la rutina LU de MKL y con la rutina LU de PLASMA usando el número de threads y los tamaños de bloque externo e interno seleccionados con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`. 65

3.10 Comparación del tiempo de ejecución (en segundos) obtenido en **saturno** con la rutina LU de PLASMA, con la rutina LU de MKL y con la rutina LU de PLASMA usando el número de threads y los tamaños de bloque externo e interno seleccionados con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`. 66

3.11 Comparación del tiempo de ejecución (en segundos) obtenido en **joule** con la rutina LU de PLASMA, con la rutina LU de MKL y con la rutina LU de PLASMA usando el número de threads y los tamaños de bloque externo e interno seleccionados con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`. 66

3.12 Tiempos de ejecución (en segundos) obtenidos en diferentes sistemas computacionales por la rutina de multiplicación de matrices con dos niveles de paralelismo utilizando una búsqueda exhaustiva y una búsqueda guiada (con diferentes porcentajes) para seleccionar el número de threads OpenMP y MKL a establecer en cada nivel. Entre paréntesis, combinación de threads OpenMP-MKL con la que se obtiene el tiempo de ejecución indicado. 67

3.13 Tiempos de ejecución (en segundos) obtenidos en **saturno** con la rutina de Cholesky de LAPACK al usar su propia selección del tamaño de bloque (LAPACK+MKL) y realizando la selección del tamaño de bloque y del número de threads con la propuesta de auto-optimización (LAPACK+AT). Entre paréntesis, número de threads OpenMP y combinación OpenMP-MKL con los que se obtiene el tiempo de ejecución indicado en cada caso. 68

4.1 Parámetros algorítmicos para los niveles 0 y 1 de la jerarquía, usando la multiplicación de matrices como rutina básica de nivel 0 y la multiplicación de Strassen y la factorización LU como ejemplo de rutinas de nivel 1. 83

5.1 Rutinas de álgebra lineal y unidades de cómputo consideradas en el estudio experimental. 98

5.2 Información almacenada en la instalación de la rutina de multiplicación de matrices para las diferentes unidades básicas de procesamiento de los nodos de cómputo del cluster **Heterosolar** (sólo se muestran las entradas correspondientes a matrices cuadradas). . . 100
5.3 Tiempo de instalación (en segundos) en los diferentes elementos computacionales de cada nodo del cluster **Heterosolar**. 103
5.4 Nivel de recursión con el que se obtiene el menor tiempo de ejecución de forma experimental (`Exp`), y nivel seleccionado con modelado empírico (`EmpMod`) y con instalación experimental (`ExpIns`) para diferentes tamaños de problema. 110
5.5 Comparación del nivel de recursión con el que se obtiene el menor tiempo de ejecución de forma experimental (`Exp`) con el obtenido por los métodos de selección `EmpMod`, `ExpIns` y `DynOpt`, para distintos tamaños de problema y sistemas computacionales y variando el número de threads utilizado en las llamadas a la rutina de multiplicación de matrices. 111
5.6 Tiempo de instalación (en segundos) con los métodos `EmpMod` y `ExpIns` en tres CPU multicore usando un máximo de seis threads. 112
5.7 Número de threads de CPU seleccionados en **saturno** con ejecución sólo en CPU y en CPU+GPU. 121
5.8 Número de threads de CPU seleccionados en **jupiter** con ejecución sólo en CPU y en CPU+multiGPU (desde 1 hasta 6 GPUs). . . . 123
5.9 Número de threads de CPU seleccionados en **venus** para distintas configuraciones de CPU sin y con coprocesadores. 124
5.10 Número de threads seleccionados en cada MIC de **venus** para distintas configuraciones de CPU con y sin GPU con 1 o 2 MIC. . . 124
5.11 Comparación de los valores de los parámetros algorítmicos seleccionados para la multiplicación matricial híbrida con distintos métodos de optimización y prestaciones obtenidas, en **jupiter**. . . . 126
5.12 Comparación de los valores de los parámetros algorítmicos seleccionados para la multiplicación matricial híbrida con distintos métodos de optimización y prestaciones obtenidas, en **venus**. 127
5.13 Tiempo de decisión (en segundos) obtenido en **jupiter** para distintos tamaños de problema, variando el número de GPUs. 130

5.14 Tiempo de decisión (en segundos) obtenido en **venus** para distintos tamaños de problema, variando el número y tipo de coprocesadores (GPU, MIC). 130

5.15 Comparación del tiempo de ejecución (en segundos) y valores seleccionados para los parámetros, aplicando la metodología de autooptimización con la multiplicación de matrices tradicional y con Strassen (con 1 nivel de recursión), para matrices cuadradas de tamaño 3500 y 7000 usando tres configuraciones diferentes en **jupiter**: sólo CPU, CPU+C2075, CPU+C2075+GTX590. 133

5.16 Método con el que se obtiene el menor tiempo de ejecución en **jupiter** para distintos tamaños de matriz, unidades de cómputo utilizadas y valores seleccionados para los parámetros en la multiplicación matricial básica. 134

5.17 Distribución de la carga entre diferentes unidades de cómputo de **jupiter** y prestaciones obtenidas en las sucesivas multiplicaciones de matrices que se llevan a cabo dentro de la rutina de factorización LU por bloques, para una matriz de tamaño 10000 y tamaño de bloque 1000. 136

5.18 Simulación del tiempo de ejecución (en segundos) de una factorización LU por bloques en un sistema CPU+multiGPU para una matriz de tamaño 10000, considerando diferentes prestaciones de las operaciones básicas en CPU para distintos tamaños de bloque. 137

5.19 Distribución de la carga de trabajo de la multiplicación de matrices entre cuatro nodos de cómputo del cluster **Heterosolar** cuando se instala en el nivel 2 de hardware usando los métodos `AT-L10` y `AT-L11` a nivel 1, y desviación en las prestaciones respecto a las obtenidas con un oráculo perfecto. 139

5.20 Comparación del tiempo de ejecución (en segundos) obtenido en **jupiter** por la multiplicación de Strassen con un nivel de recursión, para matrices de tamaño 3500 y 7000, usando paralelismo únicamente en la multiplicación (1 thread OpenMP) y con paralelismo anidado (2 threads OpenMP), estableciendo diferentes configuraciones del número de threads MKL con y sin el uso de GPU. . . . 140

5.21 Prestaciones obtenidas por OpenTuner y la metodología de optimización jerárquica con el mejor número de threads para cada tamaño de problema. 143
5.22 Parámetros algorítmicos (threads de CPU y distribución de la carga entre CPU y GPUs) y prestaciones obtenidas por OpenTuner para cada tamaño de problema, utilizando como límite de tiempo de búsqueda el empleado por la metodología jerárquica al realizar una instalación **exhaustiva** de tipo `S0H1-s0h0`. 144
5.23 Parámetros algorítmicos (threads de CPU y distribución de la carga entre CPU y GPUs) y prestaciones obtenidas por OpenTuner para cada tamaño de problema, utilizando como límite de tiempo de búsqueda el empleado por la metodología jerárquica al realizar una instalación **guiada** de tipo `S0H1-s0h0`. 145
5.24 Parámetros algorítmicos (threads de CPU y distribución de la carga entre CPU y GPUs) y prestaciones obtenidas por OpenTuner para cada tamaño de problema, utilizando como límite de tiempo de búsqueda el empleado por la metodología jerárquica al realizar una instalación **exhaustiva** en el nivel `S0H1`. 145

6.1 Prestaciones obtenidas por la rutina de factorización LU con los mejores valores de los parámetros algorítmicos (***nb*** e ***ib***) seleccionados con cada una de las estrategias de búsqueda para cada tamaño de problema. 166
6.2 Prestaciones obtenidas en **jupiter** con la rutina de descomposición de Cholesky con los valores de los parámetros seleccionados por la metodología de auto-optimización (***Tuned***) para cada tamaño de problema y con la configuración por defecto de la plataforma (***Default***), así como el porcentaje de mejora (%) obtenido con auto-optimización. 169
6.3 Prestaciones obtenidas en **jupiter** con la rutina de descomposición de Cholesky al utilizar, para cada tamaño de problema, la mejor política de planificación junto con el mejor valor de ***nb*** y la mejor configuración hardware (cores de CPU y GPUs). 171

Índice de figuras

2.1 Estructura de un sistema multicore con arquitectura NUMA. . . . 26
2.2 Organización jerárquica de cores en una GPU. 28
2.3 Organización jerárquica de memoria en una GPU. 28
2.4 Arquitectura del coprocesador Xeon Phi. 30
2.5 Estructura del cluster **Heterosolar**. 32
2.6 Distribución de la multiplicación matricial en un nodo compuesto por una CPU y un coprocesador. 37
2.7 Descomposición de tareas en una multiplicación de Strassen con un nivel de recursión. 41
2.8 Operaciones de computación realizadas por diferentes unidades de cómputo (CPU, GPU) en la factorización LU por bloques. 43

4.1 Esquema general de instalación jerárquica. 73
4.2 Metodología de auto-optimización jerárquica. 75
4.3 Opciones de instalación en una jerarquía multinivel bidimensional, con varios niveles hardware y software. 79
4.4 Instalación de la rutina de multiplicación de matrices variando la carga de trabajo en la CPU multicore de diferentes nodos. 85
4.5 Esquema de reparto de la carga de trabajo en la instalación de la rutina de multiplicación de matrices en un nodo compuesto por CPU+GPU. 88
4.6 Esquema recursivo para la factorización LU de una matriz. 93

5.1 Rendimiento obtenido y número de threads seleccionado durante la instalación de la rutina de multiplicación de matrices en **jupiter**, variando el número de cores lógicos para tamaños 3500 y 6500. . . 102

5.2 Rendimiento obtenido y número de threads seleccionados durante la instalación de la rutina de multiplicación de matrices en **saturno**, variando el número de cores lógicos para tamaño 6500. . . 103

5.3 Comparación de las prestaciones teóricas y experimentales en las CPU de cuatro nodos de **Heterosolar** variando el número de cores disponibles. A la izquierda, comparación de las prestaciones para matrices cuadradas de tamaños 3500, 6500 y 9500. A la derecha, cociente de las prestaciones para los tres tamaños. 106

5.4 Evolución del cociente de las prestaciones teóricas respecto a las experimentales de la rutina de multiplicación de matrices con cuBLAS en GPUs de distintos nodos de **Heterosolar** y para distintos tamaños de matrices cuadradas. 107

5.5 Cociente del tiempo de ejecución de la multiplicación directa con respecto a la multiplicación de Strassen con niveles de recursión 1, 2 y 3, para matrices de tamaño 8000 (arriba) y 12000 (abajo). . . 113

5.6 Cociente del tiempo de ejecución de la multiplicación de Strassen con el mejor nivel de recursión, obtenido experimentalmente, con respecto al tiempo de ejecución obtenido con `DynOpt`, para tamaños de matriz 8000 y 12000 en tres sistemas multicore. 114

5.7 Cociente del tiempo de ejecución de la multiplicación de cuBLAS con respecto a la multiplicación de Strassen con niveles de recursión 1 y 2, en tres GPUs con distinta capacidad computacional. 114

5.8 Cociente del tiempo de ejecución de la multiplicación directa con respecto al de la multiplicación de Strassen con niveles de recursión 1, 2 y 3, para matrices cuadradas de tamaño 8000 (arriba) y 12000 (abajo), en la CPU multicore de **jupiter** y **saturno**. Asimismo, se muestran algunos cocientes en GPUs de estos nodos. 116

5.9 Evolución del cociente de las prestaciones de la multiplicación de matrices en GPU con respecto a un core de CPU, en cuatro nodos de **Heterosolar**, cuando varía el tamaño de las matrices. 117

5.10 Porcentaje de carga asignado en la multiplicación de matrices a cada unidad básica de procesamiento de los cuatro nodos considerados de **Heterosolar**, teniendo en cuenta sólo la información obtenida de la instalación de nivel 0 de la multiplicación de matrices (`S0H1-s0h0`), variando el tamaño de las matrices. 118

5.11 Porcentaje de reparto de la carga de trabajo entre CPU y GPU en **jupiter**, con ejecuciones variando el número de GPUs. 119

5.12 Porcentaje de reparto de la carga de trabajo entre la CPU y los coprocesadores (GPU, MIC) de **venus**, con ejecuciones variando el número y tipo de coprocesadores. 120

5.13 Comparación de las prestaciones estimadas y experimentales para la multiplicación de matrices en los cuatro nodos considerados de **Heterosolar**, usando todas las unidades de cómputo en cada nodo y variando el tamaño de problema. A la izquierda, prestaciones obtenidas. A la derecha, cociente de las prestaciones estimadas respecto a las experimentales. 122

5.14 Cociente del menor tiempo experimental obtenido para la multiplicación híbrida en los nodos **jupiter** y **venus** con respecto a los tiempos que se obtienen al determinar la carga de trabajo teniendo en cuenta el coste de las transferencias entre la CPU y los coprocesadores, y sin tenerlas en cuenta. 129

5.15 Cociente del tiempo de ejecución de la rutina de multiplicación de matrices de MKL con respecto al de la multiplicación de Strassen con niveles de recursión 1 y 2, en **venus**, con dos coprocesadores Intel Xeon Phi. 135

5.16 Porcentaje de trabajo asignado a los coprocesadores Intel Xeon Phi en las multiplicaciones básicas de la multiplicación de Strassen cuando se realizan ejecuciones en **venus** combinando la CPU con uno o dos coprocesadores. 135

5.17 Tiempo de ejecución (en segundos) de la multiplicación de Strassen con asignación dinámica y nivel de recursión 1, con diferentes configuraciones de subnodos computacionales en **jupiter**, usando dos GPUs C2075 y una GTX590. 142

5.18 A la izquierda, prestaciones obtenidas por OpenTuner (***OT***) y por la metodología de optimización jerárquica (***HL***) al utilizar un subnodo de **jupiter** (CPU, Tesla C2075 y GeForce GTX590) variando el tamaño de problema y utilizando tres versiones diferentes para la búsqueda de los valores de los parámetros algorítmicos (***Exh***, ***L*1**, ***Guided***). Asimismo, se muestran las prestaciones máximas que se alcanzarían (***Peak***). A la derecha, pérdida de rendimiento experimentada por los métodos que explotan la jerarquía y por OpenTuner (con los límites de tiempo correspondientes) con respecto al método totalmente exhaustivo (HL_{exh}). 146

6.1 Esquema general de funcionamiento de la librería Chameleon. . . 152

6.2 Funcionamiento de Chameleon con el simulador SimGrid. 153

6.3 Operaciones realizadas sobre los elementos de una matriz en cada paso de la descomposición de Cholesky. 155

6.4 Factorización LU de una matriz. 157

6.5 Prestaciones obtenidas al instalar la rutina de descomposición de Cholesky de forma exhaustiva con un conjunto arbitrario de tamaños de bloque (arriba izquierda); prestaciones obtenidas con el mejor valor de ***nb*** para cada tamaño de problema (arriba derecha), y tiempo de instalación (en segundos) empleado con cada tamaño de problema (abajo). 161

6.6 Prestaciones obtenidas al instalar la rutina de descomposición de Cholesky aplicando una búsqueda guiada con poda (izquierda), y comparación de las prestaciones obtenidas con respecto a la instalación exhaustiva (derecha). 162

6.7 Estimación de las prestaciones obtenidas con la rutina de descomposición de Cholesky usando un enfoque simulado combinado con una estrategia exhaustiva (arriba); comparación de las prestaciones obtenidas con ambas estrategias (exhaustiva y guiada) usando el mejor valor de ***nb*** para cada tamaño de problema (abajo izquierda), y comparación de las prestaciones obtenidas con el mejor valor de ***nb*** para cada tamaño de problema usando ambos enfoques (empírico y simulado) con las citadas estrategias (abajo derecha). 164

6.8 Prestaciones obtenidas al instalar la rutina de factorización LU de forma exhaustiva con un conjunto arbitrario de tamaños de bloque (arriba izquierda); prestaciones obtenidas con la mejor combinación (***nb***, ***ib***) para cada tamaño de problema (arriba derecha), y tiempo de instalación empleado (en segundos) con cada tamaño de problema (abajo). 165
6.9 Prestaciones obtenidas con la mejor combinación (***nb***, ***ib***) seleccionada para cada tamaño de problema durante la instalación de la rutina de factorización LU utilizando la estrategia de búsqueda **Guiada_1D** (izquierda), y tiempo de instalación (en segundos) empleado para cada tamaño de problema (derecha). 166
6.10 Prestaciones obtenidas con la mejor combinación (***nb***, ***ib***) seleccionada para cada tamaño de problema durante la instalación de la rutina de factorización LU utilizando la estrategia de búsqueda **Guiada_2D** (arriba izquierda); tiempo de instalación (en segundos) empleado con cada tamaño de problema (arriba derecha), y comparativa de las prestaciones obtenidas con las tres estrategias de búsqueda (abajo). 167
6.11 Comparativa de las prestaciones obtenidas en **jupiter** de forma empírica y simulada con la rutina de factorización LU utilizando en cada caso la estrategia de búsqueda Guiada_2D. 168
6.12 Prestaciones obtenidas por la rutina de descomposición de Cholesky con diferentes políticas de planificación. 172
6.13 Prestaciones obtenidas por la rutina de descomposición de Cholesky al usar en cada política de planificación el mejor valor de ***nb*** para cada tamaño de problema (izquierda), y comparación de las prestaciones obtenidas usando la mejor política de planificación con el mejor valor de ***nb*** para cada tamaño de problema o usando la política de planificación por defecto (derecha). 173

A.1 Estructura de directorios de la aplicación **HATLib**. 214
A.2 Menú de la aplicación software. 215

Índice de algoritmos

1 Esquema algorítmico de la multiplicación de Strassen. 40

2 Instalación jerárquica en niveles superiores de la jerarquía. 74

3 Función para determinar los mejores valores de los parámetros algorítmicos y el rendimiento obtenido para un tamaño de problema n en el nivel $l > 0$ de la jerarquía. 76

4 Instalación de la rutina de multiplicación en CPU multicore. 85

5 Instalación de la rutina de multiplicación en un nodo compuesto por CPU+GPU. 88

6 Instalación de la rutina de multiplicación de Strassen en el nivel `S1H0` en una CPU multicore. 90

7 Instalación de la rutina de multiplicación de Strassen en el nivel `S1H1` en un nodo compuesto por CPU+GPU. 91

8 Función para obtener los mejores valores de los parámetros algorítmicos y el rendimiento con diferentes métodos de acceso, desde el nivel `S1H1` de la jerarquía, a la información de instalación almacenada en niveles inferiores. 92

9 Instalación de la rutina de factorización LU en el nivel `S1H0` en una CPU multicore. 94

10 Instalación de la rutina de factorización LU en el nivel `S1H1` en un nodo compuesto por CPU+GPU. 94

11 Algoritmo para la descomposición de Cholesky. 156

12 Algoritmo para la factorización LU. 158

Resumen

La resolución de problemas complejos del ámbito científico y con alto coste computacional requiere del uso de librerías numéricas de álgebra lineal sobre plataformas computacionales de altas prestaciones. La gran diversidad existente de este tipo de problemas, así como la continua aparición de sistemas computacionales con nuevas arquitecturas paralelas, ha motivado el desarrollo y la aplicación de métodos paralelos y técnicas de auto-optimización (*auto-tuning*) sobre rutinas de álgebra lineal incluidas en dichas librerías. Estas técnicas permiten la adaptación automática de las rutinas a las características del sistema de cómputo empleado, con el fin de conseguir una ejecución lo más eficiente posible. De esta manera, se facilita la obtención de buenas prestaciones por parte de usuarios no expertos, sin necesidad de disponer de conocimientos avanzados acerca de cómo paralelizar una aplicación o qué parámetros se han de considerar en función de las rutinas y las características del sistema de cómputo empleado.

Esta tesis se centra en el desarrollo, paralelización y auto-optimización de rutinas de álgebra lineal densa en sistemas heterogéneos formados por múltiples unidades de procesamiento paralelo de diferentes características (CPU, GPU o MIC) agrupadas de manera jerárquica como componentes de los nodos de cómputo que, a su vez, se agrupan en clusters. Esta agrupación jerárquica propicia que en la paralelización de las rutinas se puedan utilizar, a su vez, varios niveles y tipos de paralelismo, los cuales hay que combinar de forma armónica para su ejecución eficiente en sistemas computacionales jerárquicos de diferentes características: sistemas cc-NUMA con varios niveles en la jerarquía de memoria mediante el uso de rutinas de álgebra lineal densa por bloques y multinivel; paralelismo híbrido combinando paso de mensajes con memoria compartida y algoritmos multinivel; paralelismo multiCPU+multicoprocesadores, etc.

Dado que la paralelización por sí sola no garantiza la obtención de tiempos de ejecución cercanos al óptimo, resulta oportuno aplicar una metodología de auto-optimización que, mediante el uso de diferentes técnicas de optimización durante la fase de instalación, permita realizar una selección automática de los parámetros ajustables de la rutina que se está optimizando. De esta forma, y partiendo de los niveles más bajos de la jerarquía, se posibilita la minimización del tiempo de ejecución de la rutina en cada nivel de la jerarquía, decidiendo para los niveles superiores el grado de reutilización de la información de optimización proporcionada por los niveles inferiores frente a la obtención de sus propios valores de forma experimental, reduciendo así el tiempo de instalación de las rutinas.

Esta metodología se aplicará sobre librerías paralelas de álgebra lineal densa y sobre librerías numéricas basadas en tareas. Se estudiará el impacto que tienen diversos parámetros sobre el rendimiento, como el número de threads de ejecución o el tamaño de bloque de computación, cuando se ejecutan sobre sistemas computacionales complejos con una estructura jerárquica en la organización de sus elementos computacionales y una gran heterogeneidad, de forma que se sea capaz de determinar, en cada plataforma computacional, la versión adecuada de la rutina a utilizar.

El proceso de optimización y auto-optimización jerárquica se llevará a cabo sobre nodos CPU+GPU, CPU+multiGPU, CPU+MIC, CPU+multiMIC y, en general, en nodos CPU+multicoprocesadores, pudiendo incluir simultáneamente GPU y MIC. Los parámetros a determinar en cada tipo de nodo dependerán, por un lado, de la arquitectura hardware de las propias unidades de cómputo y, por otro, de las librerías que implementan las rutinas, e incluirán, a su vez, la distribución del trabajo entre las distintas unidades computacionales del nodo. Asimismo, los nodos se pueden combinar y agrupar en clusters, que tendrán normalmente una estructura heterogénea, con procesos de optimización distintos para cada nodo y nuevos parámetros a determinar a nivel de cluster, como el volumen de trabajo a asignar a cada nodo.

En cuanto a las rutinas de álgebra lineal, se considerarán inicialmente rutinas básicas, para pasar a continuación a rutinas de mayor nivel. Se comenzará con la rutina de multiplicación de matrices y, a continuación, se extenderá el estudio a rutinas que utilizan la multiplicación matricial como núcleo básico computacional.

Entre este tipo de rutinas podemos encontrar, por ejemplo, la multiplicación de Strassen y factorizaciones matriciales tipo LU, QR y Cholesky. Los parámetros a determinar para estas rutinas son diferentes que en el caso de la rutina básica de multiplicación de matrices, e incluyen el número de niveles de recursión (Strassen) o el tamaño de bloque de computación (en las factorizaciones). A su vez, las rutinas de nivel intermedio se pueden utilizar en rutinas de mayor nivel, por ejemplo, en la resolución simultánea de varios sistemas, por lo que en este caso habría que determinar el número de componentes computacionales a participar en la resolución de cada uno de los sistemas. De esta forma, la misma idea sobre la utilización apropiada de la información obtenida en niveles inferiores de la jerarquía para la optimización jerárquica dentro de una jerarquía de elementos computacionales, se aplicaría en una jerarquía de rutinas. Además, al disponer de múltiples librerías que trabajan sobre elementos computacionales distintos (MKL en multicore o MIC, PLASMA en multicore, MAGMA en GPU o MIC, cuBLAS en GPU, etc.), la metodología de auto-optimización jerárquica habría que adaptarla en función de la librería en uso, pues la implementación de una rutina en cada librería determina el tipo de parámetros a seleccionar.

En resumen, la metodología de auto-optimización propuesta se plantea de forma jerárquica tanto para el sistema hardware (elementos computacionales) como para el software (rutinas y librerías), lo que permite una fácil extensión de las técnicas de optimización a nuevos sistemas lógicos y físicos.

Summary

The solution of complex scientific problems with high computational cost, requires the use of numerical linear algebra libraries on high-performance computational platforms. The diversity of this type of problems, as well as the continuous appearance of computing systems with new parallel architectures, has led to the development and application of parallel methods and auto-tuning techniques to the linear algebra routines included in such libraries. These techniques allow the automatic adaptation of the routines to the characteristics of the computing system in hand to achieve efficient execution times. In this way, it is easier for non-expert users to obtain good performance without the need for advanced knowledge on how to parallelize an application or the parameters to use depending on the routines and the characteristics of the computing system.

The aim of this Thesis is the development of a generic and flexible method that allows to alleviate the deficiencies of the optimization techniques developed so far. For that, a hierarchical auto-tuning methodology is proposed for the optimization of dense linear algebra routines in heterogeneous systems, which are made up of several parallel processing units of different characteristics (CPU, GPU or MIC), hierarchically grouped in computation nodes which, in turn, are grouped in clusters. This hierarchical structure allows the use of multiple levels and types of parallelism to parallelize the routines, so different parallel paradigms need to be combined for efficient execution of the routines in hierarchical computing systems with different characteristics: cc-NUMA systems with several levels in the memory hierarchy, with multilevel and block-dense routines; hybrid parallelism combining message passing with shared memory and multilevel algorithms; multiCPU+multiprocessor parallelism, etc.

The use of parallelism does not guarantee near-optimal execution times, so it is necessary the application of some auto-tuning methodology that, through the use of various optimization techniques during an experimental phase, eases the automatic selection of the adjustable parameters of the routine being optimized. In this way, starting from the lowest levels of the hierarchy, the execution time of the routine is minimized for each level of the hierarchy, deciding for the higher levels the degree of reuse of the optimization information obtained in lower levels or if the values of the parameters are obtained experimentally at each level. The reuse of information from lower levels enables reducing the installation time of the routines.

The methodology is applied to parallel dense linear algebra libraries and task-based libraries. The impact of various parameters on performance is analyzed. For example, the number of threads and the block size are selected on systems with a hierarchical organization of the memory blocks. Theoretical and experimental analyses can be combined to determine, for each computational platform, the appropriate version of the routine to be used. The study can be extended to more complex computing systems, with a greater hierarchy in the organization of their computational elements and a greater degree of heterogeneity.

The hierarchical auto-tuning process is performed on CPU+GPU, CPU+MIC, CPU+multiGPU, CPU+multiMIC nodes and, in general, CPU+multicoprocessor nodes, which can include both GPU and MIC. The parameters to be determined for each type of node depend on the architecture of the computing units and the libraries that implement the routines. In particular, the workload distribution among the computing units of the node needs to be determined. In addition, nodes can be grouped into clusters, which usually have a heterogeneous structure, meaning that different optimization processes and parameter sets are needed for nodes of different types. Similarly, the workload for each node and new parameters need to be determined at cluster level.

As for the linear algebra routines, basic routines are initially considered, and then routines of higher levels. The basic routine considered is the matrix multiplication, and the study is extended to routines that use the matrix multiplication as a computing kernel. Some of these routines are the Strassen multiplication and matrix factorizations such as LU, QR and Cholesky. The parameters to be

determined for these routines are different from those for the basic matrix multiplication: the number of recursion levels (Strassen), and the block size (in the factorizations). In turn, intermediate level routines are used in higher level routines, for example, in the simultaneous resolution of several systems, so in this case the number of computing units working in the resolution of each of the systems could be selected. In this way, the same ideas about the appropriate use of the information obtained at different levels of a hierarchical computing system can be applied for a hierarchy of routines. In addition, having multiple libraries that work on different computational platforms (MKL in multicore or MIC, PLASMA in multicore, MAGMA in GPU or MIC, CUBLAS in GPU, etc.), the hierarchical auto-tuning methodology should be adapted depending on the library in use, because the implementation of a routine in each library determines the type of parameters to be selected.

The different chapters of the Thesis and the main conclusions for the chapters (those in which the multilevel hierarchical auto-tuning methodology is presented and experiments are summarized for several routines and libraries on several heterogeneous hardware configurations) are commented on:

- The general ideas of the Thesis are introduced in the first chapter. The context in which the work is carried out is presented, as well as the objectives and the methodology to achieve them. The state-of-the-art is also included in this chapter.
- The basic software and hardware tools used along the Thesis are listed in a second chapter. The hardware tools correspond to the basic computing units (multicore, GPU and MIC), their combination in nodes and the grouping of these ones in heterogeneous clusters, so conforming a hierarchy of computing units. Similarly, the characteristics of the linear algebra routines used for the experiments are discussed for the lowest level (matrix multiplication) and highest levels (Strassen multiplication and matrix factorizations).
- A chapter is devoted to the revision of several optimization and auto-tuning techniques for scientific software. There are three main approaches: theoretical, experimental and hybrid. The initial contributions for each of these techniques are shown, with references to the publications generated, which

are the starting point on which the hierarchical optimization methodology presented in the Thesis is based.

- The hierarchical methodology is presented below. It aims is to automatically obtain efficient execution of linear algebra software in the current heterogeneous systems, which are organized hierarchically. This can be considered the central part of the Thesis, and contains a description of the methodology working at different levels of the hierarchy, both for the software and the hardware. The possible parameters to be selected for the optimization of the routines at different levels are discussed, and several indications are given on how satisfactory values for those parameters can be obtained exploiting the hierarchical organization of routines and computing systems.

  The basic routine used is the efficient implementation of the typical three-loop matrix multiplication included in optimized dense linear algebra libraries (such as MKL and cuBLAS). This routine is used to explain the multilevel approach at three hardware levels: basic computing units, nodes and cluster. Also, virtual nodes are used as a group of computing units in a level (for example, subgroups of CPU cores together with one or more GPUs in a multicore+multiGPU node) to further optimize some routines. The extension to higher levels of software is shown with a Strassen multiplication and a LU factorization, both calling to the basic matrix multiplication routine. The methodology shows to be valid for a hierarchy of software and hardware, so resulting in a two-dimensional hierarchy of levels.

- To justify the applicability of the methodology, an exhaustive set of experiments are carried out with different linear algebra routines and hardware configurations. The routines and computing units experimented with are those presented in previous chapters. The goal is not to obtain highly efficient implementations of the routines, but to experimentally show that the methodology proposed is valid for both software and hardware at different levels. The experiments with different routines and hardware configurations show that the methodology is versatile and can be used for other routines and configurations and in larger hierarchical configurations. Furthermore, it can be combined with other optimization tools (OpenTuner) for the selection of satisfactory values for the algorithmic parameters of the routines.

- The versatility of the proposed methodology is also justified with the application of the hierarchical methodology to dense linear algebra libraries which relies on sequential task-based algorithms. The Chameleon library is used for the analysis. This library uses a runtime system (StarPU) which offers several scheduling policies to manage automatically tasks to the computing units (CPU cores and GPUs in this case). The parameters to be selected are the tile size (or tile sizes in routines with inner and outer sizes), the number of computing units used to solve the problem, and the scheduling policy of the runtime system. The routines used in the study are the Cholesky and LU factorizations, and the system is a multicore+multiGPU (Chameleon does not support Xeon Phi).

As a result, a basic auto-tuned library has been developed, including the matrix multiplication in its MKL (for multicore and Xeon Phi) and cuBLAS (for GPU) implementation. The Annex includes the user's and developer's guide, with the steps to follow for extending the library to other routines and hardware configurations. This version is freely available so that interested researchers can use it and give us feedback to improve the library.

In summary, the proposed methodology has shown to be valid for a hierarchy in both the hardware system (computational elements) and the software (routines and libraries), allowing an easy extension of the optimization techniques to new logical and physical systems. This conclusion is supported by exhaustive experiments with routines and libraries working over different hardware configurations.

Capítulo 1

Introducción

En este capítulo introductorio se comentan los aspectos generales de esta tesis. Inicialmente, se enmarca el trabajo en el contexto bajo el que se realiza y se lleva a cabo un repaso general del estado del arte, que se irá completando en sucesivos capítulos. A continuación, se describe el objetivo general de la tesis y se establecen los subobjetivos necesarios para su consecución, así como la metodología de trabajo propuesta. Finalmente, se enumeran las principales contribuciones derivadas del trabajo realizado, se comenta la estructura de la tesis y se resume el contenido de los distintos capítulos de que consta.

1.1 Contexto

Esta tesis doctoral se centra en el estudio de técnicas de optimización de rutinas de álgebra lineal en sistemas computacionales heterogéneos. Las principales razones que han motivado que se lleve a cabo esta propuesta son:

- Por un lado, la necesidad de disponer de rutinas optimizadas de álgebra lineal para que los usuarios que hacen uso de ellas obtengan tiempos de ejecución reducidos en la resolución de sus problemas científicos. Multitud de problemas en diversos campos de la ciencia y de la ingeniería se resuelven recurriendo a técnicas de computación numérica donde se utilizan rutinas de álgebra lineal, en algunos casos trabajando sobre matrices densas y de

gran dimensión. Con la optimización de este tipo de rutinas, se delega la explotación eficiente de los recursos computacionales a las rutinas y librerías de álgebra lineal que se utilizan para la resolución de los cálculos numéricos de sus problemas.

- Por otro lado, permitir el uso eficiente de los sistemas computacionales actuales explotando al máximo los recursos que ofrecen. Estos sistemas son cada vez más complejos y, en general, constan de un conjunto de nodos de cómputo formados por una CPU multicore y uno o varios coprocesadores de diferente tipo (GPU, Xeon Phi...). La complejidad de los nodos puede aumentar si, además, incluyen varias unidades computacionales de alguno de los tipos mencionados o combinaciones de ellas. Se dispone así de nodos multicore, multicore+GPU, multicore+MIC, multicore+multiGPU, multicore+multiMIC y, en general, multicore+multicoprocesadores, en los que conviven unidades computacionales heterogéneas con distinta arquitectura hardware, distinta capacidad computacional y velocidad de acceso a los niveles de su jerarquía de memoria, etc., dando lugar a nodos híbridos. A su vez, estos nodos suelen estar conectados entre sí por medio de una red (por ejemplo, Gigabit Ethernet o Infiniband) para formar un cluster de varios nodos, que pueden ser idénticos (cluster homogéneo) o, en el caso más general, de diferente tipo (cluster heterogéneo). Esta heterogeneidad conlleva que podamos plantear una visión jerárquica del sistema computacional, con elementos básicos (CPU multicore, GPU y MIC) que se combinan en componentes de mayor nivel (nodos) que, a su vez, se combinan en otros de un nivel superior (nodos organizados en un cluster).

La diversidad y complejidad de los sistemas paralelos computacionales hace que su programación no siga un único paradigma, sino que sea necesario combinar varios modelos y entornos de programación paralela en un modelo de programación híbrida. Una CPU multicore se puede programar usando, por ejemplo, el estándar *de facto* OpenMP [40, 155] para sistemas con memoria compartida, mientras que los coprocesadores con los que se combina en un nodo pueden seguir un paradigma de programación diferente y utilizar un entorno distinto al utilizado para la CPU. En GPU se suele utilizar el paradigma SIMD (*Simple Instruction Multiple Data*) para la programación en CUDA [60] y en Xeon Phi se puede ha-

cer uso de directivas OpenMP para la programación de la arquitectura Intel MIC (*Many Integrated Core Architecture*) siguiendo dicho paradigma. En cambio, para trabajar con varios nodos de un cluster es necesario disponer de un paradigma de paso de mensajes (que se puede usar también a nivel intra-nodo en una jerarquía de memoria compartida) que permita comunicar los distintos nodos, para lo que se suele utilizar el estándar *de facto* MPI [144]. Por tanto, para explotar de forma eficiente un sistema computacional complejo en la resolución de problemas científicos de gran demanda computacional, es necesario utilizar programación híbrida combinando varios paradigmas y entornos de programación paralelos.

Asimismo, el nivel de complejidad de dichos sistemas hace que la optimización de rutinas de álgebra lineal (o de cualquier tipo de rutinas) en ellos se vuelva más complicado, por lo que no se puede esperar que un usuario no experto en paralelismo, álgebra lineal o en cada uno de los sistemas computacionales y estilos de programación, pueda explotar eficientemente todo el sistema, tarea que incluso a veces se convierte en imposible para usuarios "expertos" en los distintos aspectos computacionales implicados.

De este modo, un enfoque basado en la utilización de técnicas de auto-optimización que sean capaces de adaptarse a los distintos niveles de complejidad del software (rutinas y librerías de álgebra lineal) y del hardware (cluster de nodos heterogéneos), va a permitir proporcionar a los usuarios un conjunto de rutinas que puedan utilizar en sus aplicaciones y que se adapten de forma autónoma a las características de los distintos componentes computacionales del sistema para obtener implementaciones eficientes con un uso adecuado de recursos del mismo.

1.2 Estado del arte

En esta sección se expone el contexto actual en el que se enmarcan los distintos aspectos que se tratan en esta tesis: por un lado, paralelismo, rutinas y librerías paralelas de álgebra lineal, y por otro, técnicas de optimización y auto-optimización de rutinas paralelas. Aspectos particulares del estado del arte de cada una de estas líneas de investigación, se discutirán a lo largo de la tesis en las secciones correspondientes.

La Ley de Moore [143] es la norma que ha establecido la tendencia con la que aumentarían las prestaciones de los sistemas computacionales. Ha habido discusiones sobre si las limitaciones físicas en el desarrollo de la tecnología pondrían fin a la validez de esta Ley [138, 172, 184], pero el uso de paralelismo ha hecho posible que siga cumpliéndose en cuanto al aumento de la velocidad de cómputo, multiplicada por dos cada año y medio, y por ende, la capacidad computacional de las unidades de procesamiento, permitiendo así abordar cada vez problemas de mayor dimensión y complejidad a costa de una mayor dificultad de programación para poder obtener las máximas prestaciones de los sistemas de los que se dispone. En la actualidad encontramos una gran variedad de plataformas computacionales destinadas a la aplicación de la computación paralela para la resolución de problemas de gran demanda computacional. Un listado actualizado con las más recientes se puede consultar, por ejemplo, en la lista TOP500 [141]. Estas plataformas incluyen CPUs multicore, GPUs [118, 157], Intel Xeon Phi [168], sistemas masivamente paralelos [119] o virtualizacion de GPUs [175].

En consonancia con el desarrollo de sistemas paralelos, se han ido sentando las bases de la programación paralela. Esto ha dado lugar, desde los años noventa, a la publicación de numerosos libros que intentan ofrecer un enfoque unificado [6, 11, 67, 78, 92, 94, 98, 131, 160, 164, 165, 169, 174, 199], algunos de ellos centrados en la aplicación de paralelismo en álgebra lineal o computación matricial [21, 142] debido a su gran importancia en la resolución de múltiples problemas de alto coste computacional, o directamente en la aplicación a problemas numéricos y científicos [26, 79, 122, 173]. Pero la variedad existente de entornos de computación lleva pareja una variedad en los entornos de programación. Así, para sistemas con una arquitectura basada en memoria compartida, se dispone de la librería Pthreads de C [150] y de OpenMP [5], estándar *de facto* para este tipo de sistemas; y para sistemas con memoria distribuida, la librería PVM [88] y MPI [112, 158, 176], estándar *de facto* actual para la programación mediante paso de mensajes y del que existen múltiples implementaciones, siendo OpenMPI [183], utilizada en este trabajo, la más difundida. También se encuentra disponible el entorno de desarrollo CUDA [151] para la programacion de las GPUs de NVIDIA, y OpenCL [179], que se postula como el estándar para GPUs de diferentes fabricantes.

En la actualidad, se están desarrollando nuevos entornos de programación paralela, así como diferentes optimizaciones de los existentes. Todo esto conlleva que sea preferible adoptar una metodología de trabajo que muestre su versatilidad en los entornos actuales y que, previsiblemente, pueda ser válida para nuevos entornos emergentes, como la computación en GPUs virtualizadas con rCUDA [38, 72], la mejora de las transferencias CPU $\longleftrightarrow$ GPU [196] o MPI 3.0 con comunicaciones en una dirección [61].

Como ya se ha indicado, la computación paralela se aplica en multitud de ámbitos científicos y de ingeniería donde los problemas tienen un alto coste computacional. En bioinformática, por ejemplo, se ha llevado a cabo su aplicación en *docking* de moléculas [13, 39, 63, 102, 105, 188], *virtual screening* [103, 104, 186], descubrimiento de fármacos [76, 140, 171], análisis de ADN [9], alineamiento de cadenas [134], etc. En ingeniería, por su parte, se puede encontrar su aplicación en análisis de electromagnetismo [14, 15, 120, 121, 203], simulación de sistemas multicuerpo [36], representación de terreno [22], modelado hidrodinámico [135, 197] o ingeniería agrícola [62], entre otros. En economía y estadística, en análisis envolvente de datos [18], autoregresión vectorial [37], modelos lineales [123] y modelos de ecuaciones simultáneas [74, 136, 137]. También existen trabajos sobre paralelización de rutinas básicas que se utilizan en otras aplicaciones, como la transformada Wavelet [25], la transformada rápida de Fourier [80, 81], trazado de rayos [192] y síntesis de imágenes [191], etc. En cuanto a problemas de optimización, se pueden encontrar trabajos sobre optimización combinatoria [181] y métodos metaheurísticos, que se paralelizan de múltiples formas [7, 10, 57, 75, 139] o para algún componente en particular, como GPU [182]. Y últimamente, en la era del *big data*, la computación paralela se hace imprescindible para acelerar el procesamiento de las enormes cantidades de datos que se generan continuamente [167].

En muchas de estas áreas de conocimiento, las operaciones básicas de computación se llevan a cabo con rutinas de álgebra lineal [91, 109], de ahí la multitud de trabajos de optimización llevados a cabo para estas rutinas sobre diferentes sistemas computacionales. Por ejemplo, con la rutina básica de multiplicación de matrices se han publicado trabajos para sistemas multicore [107], GPU [126, 127, 147, 154], esquema maestro-esclavo [69], clusters [12] y sistemas computacionales de altas prestaciones [189]. Dicha rutina constituye el componente bá-

sico sobre el que se implementan otras rutinas de álgebra lineal de mayor nivel, las cuales trabajan por bloques invocando a multiplicaciones implementadas de forma eficiente. Por este motivo, la investigación en la optimización de la rutina de multiplicación de matrices es fundamental. A lo largo de esta tesis se utilizará con profusión esta rutina básica para mostrar el funcionamiento de la metodología de optimización que se propone. Asimismo, se puede trabajar directamente en la optimización de rutinas de álgebra lineal de mayor nivel en sistemas concretos, como GPU [193], CPU+GPU [24, 187] o clusters [41, 66, 177]; o en la optimización de determinadas rutinas para distintos sistemas, como la evaluación de polinomios matriciales [28], la multiplicación de Strassen [89, 96, 97], sistemas de ecuaciones [156] o software matemático en general [31, 44, 90].

Así pues, existe un copioso trabajo en el desarrollo de librerías de álgebra lineal para la variedad de sistemas computacionales existentes. Las primeras aproximaciones definen un estándar de librerías de álgebra lineal básica, como BLAS (*Basic Linear Algebra Subroutines*) [64, 70], y posteriormente se va completando la jerarquía de librerías con la aparición de BLACS (*Basic Linear Algebra Communication Subroutines*) [71], PBLAS (*Parallel BLAS*) [42], LAPACK (*Linear Algebra PACKage*) [16] y ScaLAPACK (*Scalable LAPACK*) [27]. Otros trabajos desarrollan librerías con funcionalidad similar o intentan ampliar el abanico de sistemas para los que se implementan, como es el caso de PLAPACK (*Parallel LAPACK*) [190], LINPACK para CUDA [77, 195] y ScaLAPACK para sistemas heterogéneos [166]. Asimismo, existen implementaciones optimizadas de algunas de ellas (BLAS, LAPACK y ScaLAPACK), como la librería MKL (*Math Kernel Library*) de Intel [106]. También se pueden encontrar trabajos que usan métodos formales para el desarrollo de rutinas [93] o propuestas de librerías para computación grid [161]. Posteriormente, se han desarrollado librerías optimizadas para distintos tipos de arquitecturas paralelas, como CULA para GPU [73], PLASMA y MAGMA para multicore y multicore+GPU [4], respectivamente, MAGMA para Xeon Phi [68] y Chameleon para multicore+multiGPU [2]. También se podría adoptar un enfoque basado en el uso de polialgoritmos [132] o polilibrerías [8], seleccionando en cada caso, entre las librerías disponibles, la rutina con la que se espera obtener mejores resultados para el problema que se está resolviendo.

Inmersos en este amplio campo de estudio, no siempre está claro qué sistema es el más apropiado para resolver un problema [130], o cuál es la mejor manera de resolverlo en un sistema concreto. Por tanto, se requiere disponer de técnicas que permitan predecir las prestaciones de los algoritmos en sistemas de altas prestaciones [82], así como la utilización de técnicas de auto-optimización para que el software sea capaz de adaptarse de forma autónoma al sistema computacional sobre el que se ejecuta. Un trabajo pionero en este campo es la tesis de Brewer [30]. Más tarde, surgieron técnicas específicas para problemas numéricos [115, 116, 124] o algoritmos de álgebra lineal [125] y, de esta forma, en esta tesis se parte de trabajos previos sobre auto-optimización de rutinas paralelas de álgebra lineal [47, 48, 51, 65, 99, 145], adaptando las técnicas empleadas a las características de los sistemas computacionales actuales, en su mayoría de naturaleza heterogénea y con una organización jerárquica.

En la actualidad, existe una gran cantidad de investigación en este campo [146, 194], con *workshops* dedicados a auto-optimización en general [108] o en entornos específicos, como ATMG [19], dedicado a multicore y GPU. También se dispone de implementaciones de BLAS con capacidad de auto-optimización en sistemas concretos, como ATLAS (*Auto-Tuning Linear Algebra Subroutines*) [198], o técnicas para multicore [48, 52], manycore [87], sistemas heterogéneos [111, 128], sistemas de carga variable [46, 56], técnicas basadas en modelado [49, 50, 53, 83, 84, 86], diseño jerárquico de librerías [54, 55, 58, 85], algoritmos basados en *tiles* para multicore [32], algoritmos tipo *stencil* [59], algoritmos sobre matrices dispersas [114, 170], etc. En particular, se pueden encontrar algunos trabajos en la línea de investigación de esta tesis, en los que se presenta un enfoque jerárquico para la optimización de un algoritmo paralelo de multiplicación de matrices en sistemas de memoria distribuida [95]. Estas ideas de auto-optimización también se pueden utilizar en otro tipo de rutinas, como las destinadas al cálculo de valores propios [117] o la transformada rápida de Fourier [133], así como en aplicaciones de mayor nivel dedicadas a análisis climático [110] o procesado de señal [163].

1.3 Objetivos

El objetivo principal de esta tesis es el desarrollo de una metodología de auto-optimización jerárquica para rutinas de álgebra lineal en entornos computacionales heterogéneos. Esta metodología debe ser diseñada de forma que permita la obtención de rutinas eficientes, empezando desde el nivel más bajo de la jerarquía, tanto software como hardware, hasta llegar al nivel más alto en cada caso.

En lo referente al software (rutinas de álgebra lineal densa), se comienza por el nivel correspondiente a las rutinas básicas, principalmente la multiplicación de matrices, pues constituye el componente computacional básico sobre el que se implementan otras rutinas eficientes de álgebra lineal de nivel superior en los entornos computacionales actuales. A continuación, la metodología se extiende a rutinas de mayor nivel, como la multiplicación de Strassen (que en el caso base de su esquema algorítmico de tipo divide y vencerás utilizará la multiplicación de matrices previamente optimizada), o descomposiciones matriciales como la factorización LU. Del mismo modo, en un siguiente nivel de la jerarquía, se podría considerar la ejecución de códigos científicos que realicen llamadas a rutinas del nivel inferior, como pueden ser simulaciones en las que se lleve a cabo la resolución simultánea de varios sistemas de ecuaciones [36]. Además, dado que existen multitud de librerías de álgebra lineal optimizadas y, en algunos casos, paralelizadas para distintos sistemas, se hace uso de dichas rutinas y se analiza la aplicación de la metodología de auto-optimización jerárquica considerando diferentes librerías. De esta manera, se puede mostrar su validez independientemente de las rutinas y librerías básicas utilizadas.

En cuanto al hardware, los componentes que se consideran a más bajo nivel son los elementos computacionales básicos: CPU multicore, GPU y MIC (Xeon Phi). Los parámetros a tener en cuenta para optimizar las rutinas son distintos en cada uno de ellos, pero la metodología debe ser válida para todos. Estos elementos computacionales, a su vez, pueden agruparse para formar otros de mayor nivel (nodo). Generalmente, hoy en día un nodo computacional estándar suele estar formado por una CPU multicore y uno o varios coprocesadores, que en algunos casos son de distinto tipo (GPU y/o MIC) e incluso pueden tener diferente capacidad computacional, aunque sean del mismo tipo. La metodología, por tanto,

debe considerar cómo combinar la información de auto-optimización a nivel de elemento computacional básico, para guiar la optimización a nivel de nodo. Los nodos, a su vez, se pueden agrupar en clusters, homogéneos o heterogéneos, por lo que la metodología jerárquica ha de ser capaz, así mismo, de obtener la información de optimización a nivel de nodo para guiar la optimización de las rutinas a nivel de cluster.

El proceso de auto-optimización tiene como principal objetivo obtener rutinas eficientes en los sistemas computacionales para los que se diseñan, para lo cual es necesario seleccionar los valores a utilizar para cada uno de los parámetros del conjunto de parámetros ajustables (también llamados parámetros algorítmicos, o AP, *Algorithmic Parameters*) que lleven a la obtención de tiempos de ejecución cercanos al óptimo experimental. La elección del conjunto de parámetros depende tanto de la rutina como del entorno computacional. Por ejemplo, para una rutina de multiplicación de matrices en multicore o en Xeon Phi, habrá que decidir el número de threads a generar, pero en un nodo con multicore+multiGPU habrá que determinar la carga de trabajo a asignar al multicore y a cada una de las GPUs. Por otro lado, en una factorización LU por bloques, habrá que determinar el tamaño de bloque con el que se obtiene menor tiempo de ejecucion, y esto dependerá de las rutinas básicas a las que invoca internamente, tanto para la resolución de sistemas de ecuaciones lineales como para la multiplicación de matrices, que pueden, a su vez, estar diseñadas para ser auto-optimizadas. Por tanto, la metodología de auto-optimización debe garantizar que es válida para múltiples rutinas, librerías y sistemas computacionales, de forma que sea fácilmente extensible en cualquiera de estas dimensiones.

En resumen, para alcanzar el objetivo principal de la tesis, habrá que trabajar en dos direcciones, software y hardware, comenzando en cada caso desde el nivel más bajo de la jerarquía. Para ello, se plantean los siguientes subobjetivos:

- Estudio de técnicas de auto-optimización de la rutina básica de multiplicación de matrices:
  - En elementos computacionales básicos (CPU, GPU y Xeon Phi), determinando, en cada caso, tanto los parámetros algorítmicos como la técnica de selección de los valores óptimos de dichos parámetros.

  - En nodos compuestos por varios elementos computacionales básicos, donde el parámetro algorítmico a determinar corresponde a la carga de trabajo a asignar a cada elemento computacional. La obtención del valor del resto de parámetros se delega a la instalación realizada para cada uno de los elementos computacionales básicos, obteniendo así una instalación jerárquica que facilita el proceso de selección del valor de los parámetros.
  - En un cluster compuesto por varios nodos, determinando el volumen de trabajo que le corresponde a cada nodo y haciendo uso de la información de instalacion almacenada en el nivel inferior de la jerarquía para obtener la carga asignada a cada elemento computacional dentro de cada nodo. Asimismo, se podrían considerar parámetros algorítmicos relacionados con las comunicaciones, como el tipo de comunicación a realizar en cada momento (p2p, colectiva).

- Estudio de técnicas de auto-optimización de rutinas basadas en la multiplicación de matrices, como la multiplicación de Strassen o la factorización LU:

  - En elementos computacionales básicos (CPU, GPU y Xeon Phi), la optimización se puede delegar a rutinas computacionales básicas, como la multiplicación de matrices, y a otras rutinas de menor coste, como la resolución de sistemas triangulares múltiples (en la factorización LU). Asimismo, se puede considerar optimizar directamente la rutina a este nivel de la jerarquía de rutinas, invocando a una rutina LU paralelizada con *multithreading* para multicore para determinar el número de threads a usar en la paralelización de la LU y a nivel de rutinas de BLAS.
  - A nivel de nodo, la optimización se puede delegar a la rutina con mayor coste computacional dentro de la rutina a optimizar, como la multiplicación de matrices en la factorización LU, o se puede trabajar en determinar directamente los parámetros de las rutinas en este nivel de la jerarquía, en cuyo caso, además del tamaño de bloque, habría que determinar cómo asignar cada *kernel* computacional básico a cada elemento computacional.

  - En un cluster la situación es igual que a nivel de nodo: o bien se determina la carga de trabajo a asignar a cada nodo invocando a rutinas básicas optimizadas a nivel de cluster, o se optimiza directamente la rutina a este nivel, determinando el tamaño de bloque y el volumen de trabajo a asignar a cada nodo del cluster.

- Existe multitud de código científico que realiza llamadas a rutinas de álgebra lineal de este segundo nivel software. Si las llamadas son consecutivas, la optimización se consigue simplemente con la utilización de dichas rutinas optimizadas, pero si se pueden realizar computaciones simultáneas, hay que determinar cómo asignar cada una de ellas a los elementos computacionales:

  - En elementos computacionales básicos se debe considerar si es preferible llevar a cabo las operaciones de computación de forma consecutiva o simultáneamente, en cuyo caso habrá que determinar cómo dividir el sistema computacional para realizar cada una de ellas. Por ejemplo, en una CPU multicore con 12 cores, se pueden considerar tres grupos de 4 cores y que cada grupo trabaje en la resolución de un problema, resolviendo así tres problemas de forma simultánea.

  - Lo mismo ocurre a nivel de nodo, donde hay que determinar cómo agrupar los elementos computacionales que trabajarán en cada problema. Por ejemplo, en un nodo con una CPU multicore con 12 cores y dos GPUs, se pueden agrupar 6 cores con cada una de las GPU, y que cada grupo trabaje en la resolución de un problema, resolviendo así dos problemas de forma simultánea, lo que puede dar lugar a implementaciones más rápidas que si se usa todo el sistema (12 cores y 2 GPUs) para resolver los dos problemas de forma consecutiva.

  - En un cluster también habría que determinar cómo agrupar los nodos para asignarle trabajo a cada uno de ellos y que trabajen simultáneamente en la resolución del problema.

1.4 Metodología

La metodología de trabajo que se va a emplear, se corresponde con los distintos objetivos planteados: se comenzará trabajando con los niveles más bajos de la jerarquía, tanto software como hardware, y se pasará a continuación a los siguientes niveles hasta llegar al mayor nivel considerado en cada caso. En primer lugar, se trabajará con la multiplicación de matrices básica, tanto a nivel de elemento computacional como a nivel de nodo y de cluster. A continuación, con rutinas que invocan a la multiplicación de matrices en diferentes niveles hardware. Finalmente, se podría trabajar en la optimización de rutinas que invocan a esas rutinas intermedias, de nuevo en los tres niveles.

En cada caso, se considerarán varias rutinas y sistemas, de manera que las conclusiones sean generales y no dependan de la rutina o de un sistema en particular. Así, mientras que en el nivel básico de las rutinas puede ser suficiente trabajar con la multiplicación de matrices (al ser el núcleo computacional en el que se basan las rutinas por bloques); en el siguiente nivel se podrían considerar varias rutinas con parámetros de distinto tipo, como el nivel de recursión en la multiplicación de Strassen o el tamaño de bloque en las factorizaciones matriciales. Por otro lado, en un mismo nivel hardware del sistema computacional se considerarán varias configuraciones, por ejemplo, multicores con distinto número de cores con/sin *hyperthreading* activado, o agrupaciones de nodos, a nivel de cluster, que den lugar a configuraciones con mayor o menor grado de heterogeneidad.

Se analizará detenidamente la metodología de auto-optimización para cada nivel de rutinas y de sistema computacional, con diferentes rutinas en cada nivel y con experimentos en varios sistemas. Una vez analizadas las características de la metodología en ese nivel, se pasará al siguiente nivel de elementos de computación, analizando qué información del nivel anterior se puede utilizar para acelerar la toma de decisiones en el presente nivel manteniendo, al mismo tiempo, la calidad del proceso de optimización. Tras analizar las rutinas de un nivel, se pasará al siguiente nivel de la jerarquía software, recorriendo de nuevo, de forma incremental, los niveles de la jerarquía hardware.

1.5 Contribuciones

Al margen de las contribuciones en forma de presentaciones en congresos o publicaciones en revistas, se ha desarrollado un sistema software que permite realizar la instalación de rutinas auto-optimizadas de forma jerárquica. Este software se encuentra disponible en http://luna.inf.um.es/grupo_investigacion/software. En el Anexo A se incluyen los manuales de usuario y de desarrollador (disponibles también en el enlace anterior), donde se indican las características generales de este sistema software de *auto-tuning* y se describe su estructura y funcionamiento, de forma que quien pueda estar interesado, conozca los detalles necesarios para hacer uso del mismo.

1.6 Estructura de la tesis

A continuación se resume el contenido de cada uno de los capítulos de que consta esta tesis:

- En el Capítulo 2 se comentan las herramientas hardware y software utilizadas. En las herramientas hardware se encuentran las plataformas computacionales sobre las que se han realizado los experimentos, y en las herramientas software, los paradigmas de programación paralela, librerías de algebra lineal y rutinas consideradas en el estudio experimental.

- En el Capítulo 3 se discuten las técnicas de optimización y auto-optimización de software más ampliamente utilizadas, completando el estado del arte en estas líneas. Asimismo, se indican las técnicas básicas que se utilizan en la metodología jerárquica propuesta.

- En el Capítulo 4 se presenta la metodología jerárquica y describe con detalle cómo se aplica el proceso de auto-optimización en cada uno de los niveles hardware y software, mostrando con rutinas de bajo nivel su funcionamiento en sistemas con una complejidad limitada.

- En el Capítulo 5 se muestran ejemplos de aplicación de la metodología jerárquica variando los niveles hardware y software. Por un lado, se muestra cómo se lleva a cabo el proceso de instalación de algunas rutinas en diferentes niveles de la jerarquía hardware, y de forma similar se discute la estructura jerárquica de la metodología en el nivel software, analizando cómo se puede reutilizar la información de optimización de una rutina en otras de nivel superior que la utilizan como componente computacional básico. Estos ejemplos se muestran para diferentes librerías de álgebra lineal y con distintas técnicas de búsqueda de los valores de los parámetros algorítmicos.

- En el Capítulo 6 se analiza la extensión de la metodología a librerías de álgebra lineal con un diseño y estructura de funcionamiento diferente al ofrecido por librerías tradicionales. Concretamente, se muestra la aplicación de la metodología de optimización a rutinas de la librería Chameleon [101], con algoritmos basados en *tiles* y asignación dinámica de tareas.

- Finalmente, en el Capítulo 7 se muestran las principales conclusiones obtenidas y posibles líneas de trabajo futuro, así como la difusión de resultados (en congresos y publicaciones) llevada a cabo como consecuencia de las tareas de investigación realizadas en esta tesis.

- De forma adicional, el Anexo A contiene los manuales de usuario y de desarrollador del software de instalación jerárquica en el que se ha implementado la metodología de auto-optimización propuesta. Se muestra su funcionalidad con diferentes rutinas y librerías de álgebra lineal en un sistema con tres niveles hardware (elementos computacionales básicos, nodos y cluster).

Capítulo 2

Herramientas computacionales

En este capítulo se resumen las herramientas hardware y software utilizadas en esta tesis, y se describe su organización en niveles con el fin de reflejar la idea fundamental de este trabajo, que no es otra que la aplicación de técnicas de optimización mediante la adopción de un enfoque jerárquico.

Las herramientas hardware corresponden a los elementos básicos de computación (CPU multicore, GPU y Xeon Phi) y la forma en que éstos se combinan para constituir nodos que, a su vez, se agrupan para formar clusters.

Las herramientas software, por su parte, incluyen las librerías de álgebra lineal que se utilizan para llevar a cabo las operaciones básicas de computación, así como las rutinas en cuya optimización se trabaja.

2.1 Herramientas hardware

Tal como se ha indicado, los sistemas computacionales sobre los que se trabaja están organizados en niveles. En un primer nivel (en adelante referenciado como nivel cero) se encuentran las unidades básicas de procesamiento. Existen sistemas de diferentes tipos, pero en este trabajo el estudio realizado se centra en los tres que se pueden considerar más extendidos en la actualidad dentro del campo de la computación científica [141]. Sus características principales son:

- **Multicore**. Estos sistemas incluyen desde dos cores hasta unas pocas decenas. En la actualidad, los nodos computacionales básicos están formados por una CPU multicore. En algunos casos, disponen de *hyperthreading*, cuya activación permite duplicar el número de threads a usar en cada core físico. La figura 2.1 muestra el esquema de un multicore formado por cuatro hexacores con un total de 24 cores físicos. Cada uno de los hexacores es lo que se denomina *socket*. Dado que el sistema dispone de *hyperthreading* y se encuentra habilitado, cada core físico se ve como dos cores lógicos, lo que da lugar a un total de 48 cores lógicos. Asimismo, se observa que el sistema de memoria está organizado jerárquicamente, con niveles de memoria asignados a los cores y a los sockets, pero todos los cores lo ven como un sistema de memoria compartida. De entre los diferentes entornos de programación existentes para explotar el paralelismo y la arquitectura de memoria en este tipo de sistemas, se hará uso de OpenMP [155], considerado actualmente el estándar *de facto*.

Figura 2.1: Estructura de un sistema multicore con arquitectura NUMA.

- **GPU**. Estas unidades de procesamiento gráfico se llevan utilizando desde hace unos años para computación de propósito general (GPGPU). En la actualidad, se suelen incluir en nodos de cómputo que disponen de una CPU multicore. La arquitectura de una GPU difiere en varios aspectos respecto a la de un multicore, principalmente en el diseño y número de cores y en la organización de memoria. El número de cores es más elevado (de unos cientos a miles), y éstos están organizados de forma jerárquica (figura 2.2), con varios cores (*Streaming Processors*, SP) agrupados en un *Streaming Multiprocessor*, SM. La memoria, también está organizada jerárquicamente (figura 2.3), con una serie de registros, memoria compartida, constante y de texturas para cada uno de los SM, y con una memoria global a la que pueden acceder todos los cores, siendo más rápido el acceso si se realiza a zonas de memoria directamente asociadas al SM donde se encuentra el core. Esto hace que sea necesario optimizar los accesos a memoria para obtener implementaciones altamente eficientes en este tipo de arquitectura, pero dada la diversidad de tipos de GPU existentes, con arquitecturas y capacidades computacionales diferentes, la portabilidad y optimización de código se complica. Para su programación, se pueden encontrar entornos de desarrollo portables entre diferentes plataformas, como OpenCL [179], y otros dependientes del fabricante, como CUDA [60, 151] de NVIDIA [153], que es el que se usa en este trabajo debido a la amplia expansión de las GPUs de dicho fabricante en los sistemas computacionales actuales. El paradigma de programación que se sigue consiste en ejecutar el programa principal en CPU e ir lanzando núcleos computacionales (*kernels*) a las GPUs presentes en el sistema para ejecutar las zonas del código con una mayor carga computacional, accediendo a los datos conforme al paradigma SIMD (*Simple Instruction Multiple Data*) [92]. Esto hace que sea necesario utilizar un enfoque de programación heterogéneo, combinando OpenMP con CUDA.

- **MIC** (*Many-Integrated Cores*). Este tipo de sistemas permite disponer de mayor número de cores que los sistemas multicore. El más representativo con una arquitectura MIC, es el coprocesador Intel Xeon Phi [43]. La figura 2.4 muestra un esquema de su arquitectura. El número de cores físicos en un coprocesador de este tipo está comprendido entre 57 y 72, dependiendo de la generación a la que pertenezca. Cada core está basado en la arquitectura

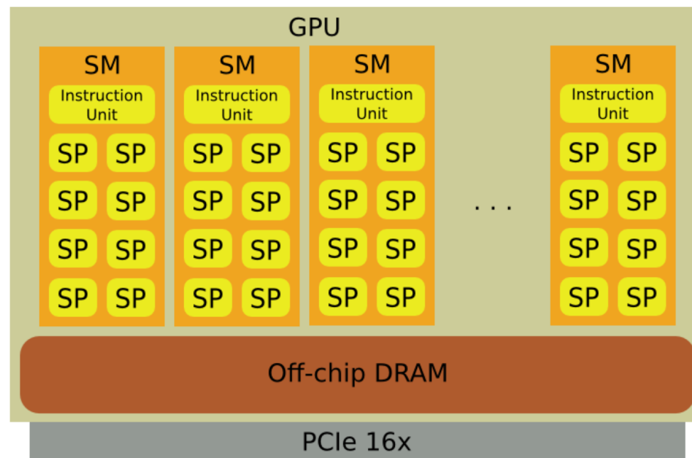

Figura 2.2: Organización jerárquica de cores en una GPU.

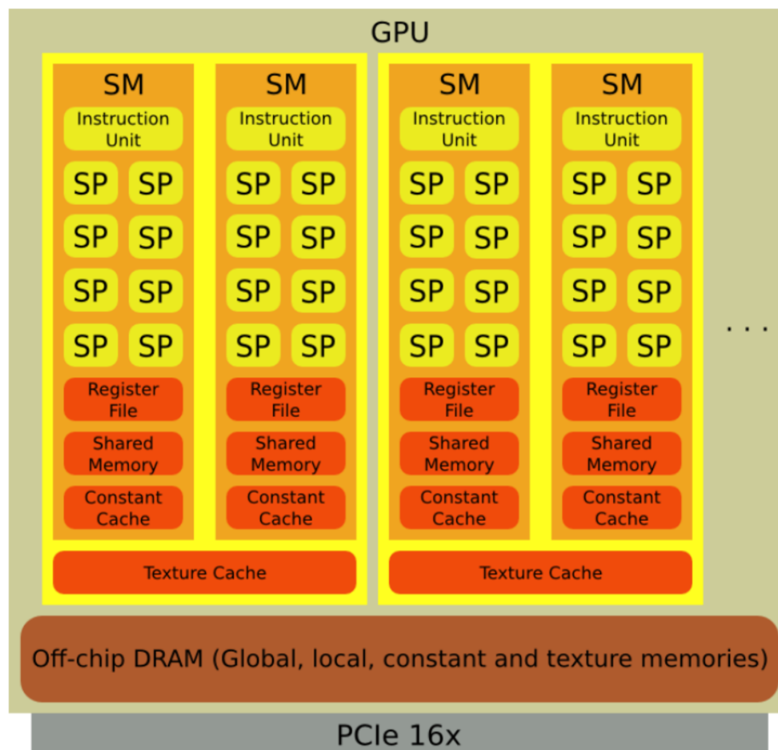

Figura 2.3: Organización jerárquica de memoria en una GPU.

x86-64 utilizada en los procesadores de Intel de propósito general, pero con una frecuencia de funcionamiento de 1.1 GHz en aras de reducir el consumo de energía. Esta reducción de la capacidad computacional se compensa con el número de threads que soporta cada core, pues cada uno permite hacer uso de hasta 4 threads hardware, lo que da lugar a un total de entre 228 y 288 threads cuando se utilizan todos los cores del coprocesador. Estos cores están interconectados por medio de un bus bidireccional en forma de anillo, en el que cada core comparte la porción de la cache L2 que le corresponde. Asimismo, cada uno dispone de su propia cache L1 y de una VPU (*Vectorial Processing Unit*) con 512 registros para explotar el paralelismo a nivel de instrucción (ILP). La arquitectura Intel MIC soporta dos modos de programación: *offload* y nativo, aunque también se puede usar MPI [149]. En modo nativo, el coprocesador trabaja de forma independiente (sin requerir la intervención de la CPU) y se puede hacer uso de OpenMP para explotar el paralelismo a nivel de threads. En modo *offload*, la ejecución se lleva a cabo de forma conjunta entre el coprocesador y la CPU multicore. Dado que los datos se encuentran en la memoria principal asociada a la CPU, es necesario utilizar un conjunto de directivas específicas para la gestión de las transferencias desde/hacia la CPU, permitiendo así transferir al Xeon Phi los datos necesarios para realizar la computación de dichas zonas del código. Este modo de programación es el que se usa en esta tesis, ya que va a permitir explotar tanto el paralelismo a nivel de hilo como la jerarquía de memoria compartida.

Estos elementos computacionales básicos se pueden agrupar de diferentes maneras para dar lugar a una unidad computacional de mayor nivel, los nodos, que constituyen el segundo nivel (nivel 1) en la jerarquía hardware de los sistemas computacionales. De esta forma, un nodo puede incluir una o varias CPU multicore y uno o varios coprocesadores, GPUs y MICs. En algunos casos, los coprocesadores pueden ser de distinto tipo (GPU y MIC), o del mismo tipo pero con arquitectura y capacidad computacional diferente. Esto ocasiona que se puedan tener sistemas con distintos grados de heterogeneidad. Por otro lado, la forma de programar cada una de las unidades de computación puede ser distinta, con lo que se necesita utilizar un esquema de programación híbrida, que combine OpenMP

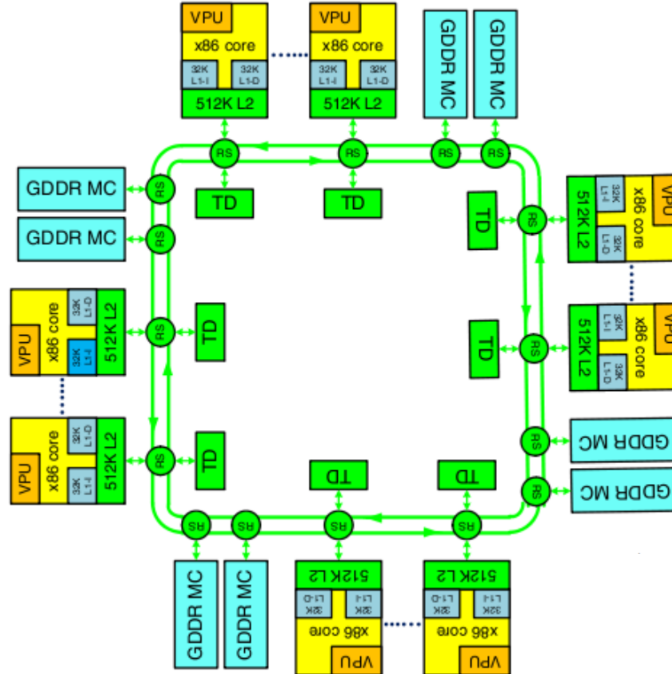

Figura 2.4: Arquitectura del coprocesador Xeon Phi.

en CPU con CUDA (en GPU), o con otro nivel de programación OpenMP junto a directivas offload (en Xeon Phi).

En general, el esquema de programación que se aplica consiste en utilizar threads OpenMP en el multicore, algunos de los cuales realizarán cómputo en CPU y otros se utilizarán para asignar trabajo a los coprocesadores, bien con llamadas a kernels en GPU o en modo offload en MIC. Para obtener prestaciones satisfactorias, habrá que determinar el volumen de trabajo a asignar a cada unidad computacional básica (elementos del nivel 0 de la jerarquía) y utilizar rutinas optimizadas para cada una, ya sea de desarrollo propio o pertenecientes a alguna librería disponible. Además, se puede considerar que esta jerarquía es reconfigurable, pues en algunos casos, se puede decidir formar elementos computacionales de nivel 1 usando subconjuntos de las unidades disponibles en un nodo. Por ejemplo, en un nodo con 12 cores y dos GPUs, se podrían formar dos subnodos, cada uno con 6 cores de CPU y una GPU. De esta forma, si hay que resolver dos problemas, sería preferible resolverlos simultáneamente, cada uno en un subnodo, en vez de resolver los dos problemas uno tras otro usando el nodo completo.

Varios de los elementos computacionales de nivel 1 (normalmente nodos, pero también subgrupos de unidades computacionales, tal como se ha dicho) se pueden agrupar para formar unidades de cómputo de nivel 2. Lo más común es tener un cluster compuesto por un conjunto de nodos interconectados por medio de una red Gigabit Ethernet o Infiniband. Esto requiere hacer uso del modelo de programación por paso de mensajes, para lo que se utiliza MPI [144], que es el estándar *de facto* para este paradigma de programacion. De esta forma, la programación se hace más híbrida y el sistema computacional más heterogéneo, ya que los nodos pueden ser de distinto tipo, con diferente número de cores y coprocesadores, que pueden tener distinta capacidad computacional y capacidad de memoria, etc. De nuevo, será necesario determinar el volumen de trabajo a asignar a cada nodo (o en general, a cada elemento de cómputo de nivel 1), delegando al nodo la explotación eficiente de las unidades que lo conforman, obteniendo de esta manera una optimización jerárquica del sistema completo.

En el estudio experimental se han utilizado varios clusters heterogéneos, pero principalmente se ha hecho uso del cluster **Heterosolar** (figura 2.5), del Grupo de Computacion Científica y Programación Paralela de la Universidad de Murcia [159]. El cluster está formado por un nodo de acceso (**luna**) y cinco nodos de cómputo con las siguientes características:

- **luna**. Actúa como *front-end* para permitir el acceso al cluster. Está virtualizado, por lo que no se puede utilizar como nodo de cómputo. Se encarga de exportar el sistema de ficheros al resto de nodos y permitir el acceso a ellos vía `ssh`. Asimismo, dispone de un sistema de colas para la planificación de los trabajos que se envían a los nodos de cómputo solicitados.

- **marte** y **mercurio**. Son nodos de cómputo idénticos, con una CPU AMD Phenom II X6 1075T (hexa-core) a 800 MHz y arquitectura x86-64, 16 GB de memoria RAM, caches L1 y L2 privadas de 64 KB y 512 KB, respectivamente, y una cache L3 de 6 MB compartida por todos los cores. Cada nodo dispone de una GPU NVIDIA GeForce GTX 480 (Fermi) con 1536 MB de Memoria Global y 480 cores CUDA (15 Streaming Multiprocessors y 32 Streaming Processors por SM).

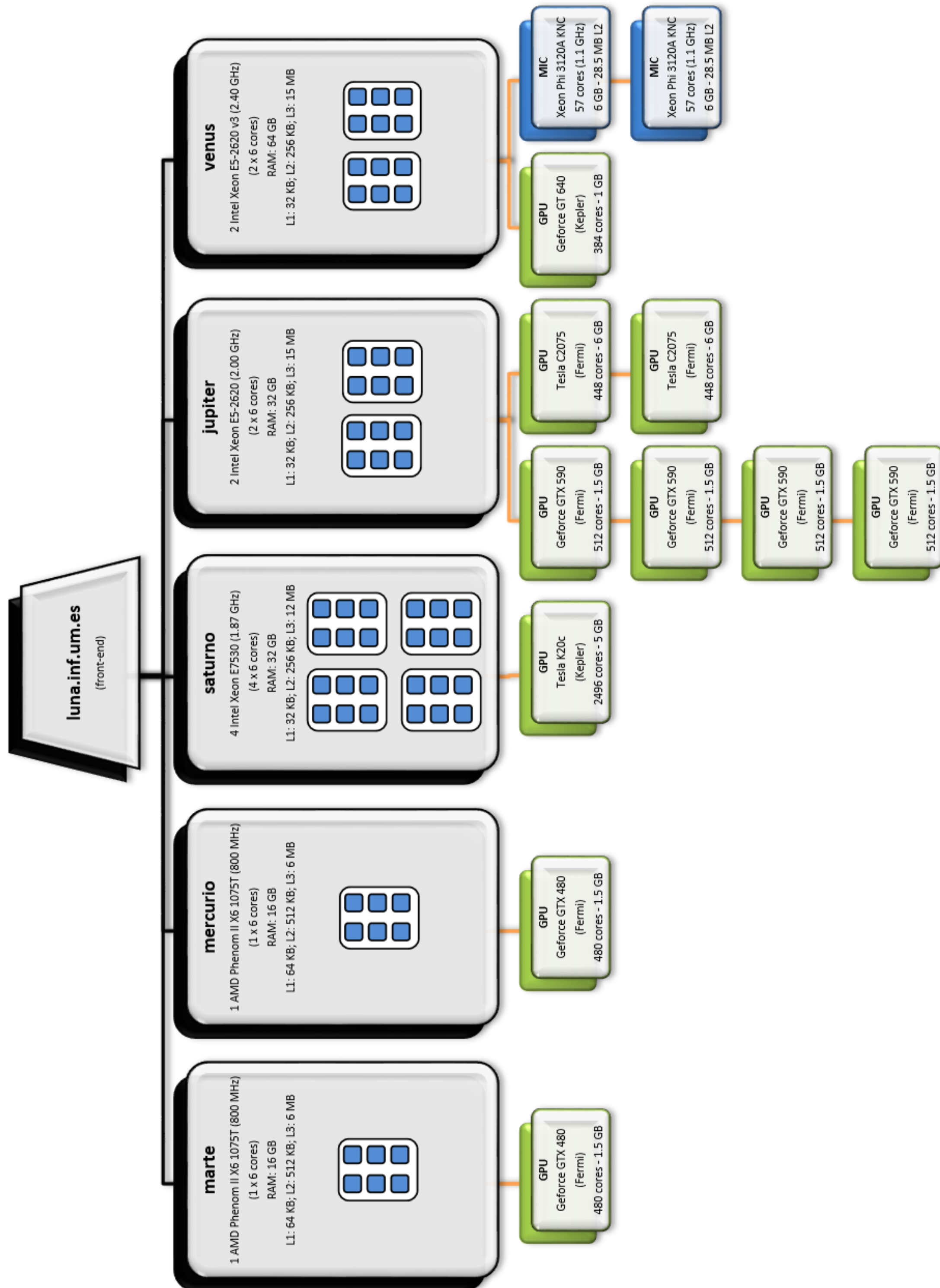

Figura 2.5: Estructura del cluster **Heterosolar**.

- **saturno**. Nodo de cómputo con 4 CPU Intel Xeon E7530 (hexa-core) a 1.87 GHz y arquitectura x86-64. Dispone de *hyperthreading* habilitado, por lo que cada core soporta hasta 2 threads de procesamiento, lo que hace un total de 48 cores lógicos. La arquitectura de memoria es de tipo NUMA con 32 GB de memoria RAM, caches L1 y L2 privadas por core de 32 KB y 256 KB, respectivamente, y una cache L3 de 12 MB compartida por todos los cores de cada socket (nodo NUMA). Dispone de una GPU NVIDIA Tesla K20c (Kepler) con 4800 MB de Memoria Global y 2496 cores CUDA (13 Streaming Multiprocessors y 192 Streaming Processors por SM).

- **jupiter**. Nodo de cómputo con 2 CPU Intel Xeon E5-2620 (hexa-core) a 2.00 GHz y arquitectura x86-64. Dispone de *hyperthreading* habilitado, por lo que cada core soporta hasta 2 threads de procesamiento, lo que hace un total de 24 cores lógicos. La arquitectura de memoria es de tipo NUMA con 32 GB de memoria RAM, caches L1 y L2 privadas por core de 32 KB y 256 KB, respectivamente, y una cache L3 de 15 MB compartida por todos los cores de cada socket (nodo NUMA). Dispone de seis GPUs: dos NVIDIA Tesla C2075 (Fermi) con 5375 MB de Memoria Global y 448 cores CUDA (14 Streaming Multiprocessors y 32 Streaming Processors por SM) y cuatro GPUs agrupadas en dos tarjetas, cada una con dos NVIDIA GeForce GTX 590 (Fermi) con 1536 MB de Memoria Global y 512 cores CUDA (16 Streaming Multiprocessors y 32 Streaming Processors por SM).

- **venus**. Nodo de cómputo con 2 CPU Intel Xeon E5-2620 (hexa-core) a 2.40 GHz y arquitectura x86-64. Tiene una arquitectura de memoria tipo NUMA con 64 GB de memoria RAM, caches L1 y L2 privadas por core de 32 KB y 256 KB, respectivamente, y una cache L3 de 15 MB compartida por todos los cores de cada socket (nodo NUMA). Dispone de una GPU NVIDIA GeForce GT 640 (Kepler) con 1024 MB de Memoria Global y 384 cores CUDA (2 Streaming Multiprocessors y 192 Streaming Processors por SM), y dos coprocesadores Intel Xeon Phi 3120A Knights Corner con 57 cores físicos (228 lógicos) a 1.1 GHz, 6 GB de memoria, caches L1 y L2 privadas por core de 32 KB y 28.50 MB, respectivamente, y VPUs de 512 bits de ancho.

Se observa que el cluster contiene nodos de distintas características, con CPUs multicore con diferente número de cores, con y sin *hyperthreading*, con GPUs con distinta capacidad computacional y, en algunos casos, con varias GPUs de distinto tipo o combinando GPU y coprocesadores Xeon Phi. Así pues, esta plataforma resulta apropiada para el análisis experimental de la metodología jerárquica propuesta para la optimización de rutinas de álgebra lineal en sistemas heterogéneos, que constituye el objetivo de la tesis. En algunos experimentos se utilizarán nodos completos o el cluster en su totalidad, pero en otros, se hará uso de subconjuntos del cluster o de los nodos, lo que permitirá experimentar con sistemas computacionales con distintos grados de complejidad y heterogeneidad.

Además del cluster descrito, se utilizarán puntualmente otros sistemas computacionales para mostrar resultados experimentales de algunos aspectos citados en la tesis, comentando las características principales de esos sistemas en los apartados correspondientes.

2.2 Herramientas software

Las herramientas software que se utilizan en esta tesis consisten en las rutinas con las que se ilustra la metodología de auto-optimización jerárquica propuesta (multiplicación de matrices, multiplicación de Strassen y factorización LU) y las librerías numéricas que se utilizan en los elementos computacionales básicos para invocar a rutinas básicas de álgebra lineal.

2.2.1 Librerías de álgebra lineal

Actualmente, las librerías de álgebra lineal densa se basan en algoritmos por bloques o *tiles* para obtener implementaciones eficientes de altas prestaciones [16, 91]. La operación básica que abarca la mayor parte del coste computacional de sus rutinas es la multiplicación de matrices, de ahí que el interés se centre, principalmente, en la optimización y paralelización de esta rutina. En este trabajo se toma como referencia dicha rutina para mostrar el funcionamiento de la metodología de auto-optimización jerárquica. Esta rutina se encuentra disponible

dentro de la librería BLAS, de la que existen implementaciones comerciales (Intel MKL [106], IBM ESSL [100], AMD ACML [1], etc.) que, en algunos casos, ofrecen versiones académicas con licencias de validez limitada, e implementaciones de libre distribución, tanto para multicore (OpenBLAS [201], PLASMA [32], Goto BLAS [189], etc.) como para GPU (MAGMA [185], CULA Tools [73], cuBLAS [152], etc.). Las características de algunas de ellas son:

- Para sistemas multicore:
  - Intel MKL [106]. Ofrece una implementación optimizada de las rutinas de BLAS, LAPACK y ScaLAPACK [27]. Asimismo, dispone de paralelización multihilo y posibilita que, o bien sea la propia rutina la que seleccione el número de threads a utilizar, o se fije dicho número al valor deseado. También se puede fijar el número de threads a usar en rutinas de BLAS y en rutinas de mayor nivel, consiguiendo así paralelismo de dos niveles. Además, se puede combinar paralelismo implícito de MKL con paralelismo explícito OpenMP o MPI. Esta librería se utilizará normalmente en el estudio experimental realizado en este tipo de sistemas para la ejecución de las rutinas básicas.
  - PLASMA y MAGMA [4]. Incluyen implementaciones de las rutinas de LAPACK. Ambas son de libre distribución y sólo se utilizarán en la realización de algunos experimentos.
- Para GPU, están disponibles las librerías mencionadas, como MAGMA o cuBLAS, que implementan las rutinas de BLAS e incluyen, a su vez, algunas de las rutinas propias de LAPACK. De entre ellas, esta última es la que se usará principalmente en los experimentos.
- Para Xeon Phi, existe una implementación de MKL [148] ofrecida por Intel, que es la que se utilizará en los experimentos.

Dada la gran variedad de librerías, la metodología propuesta debe ser general y válida, con independencia de las librerías que se utilicen para invocar a las rutinas básicas dentro de la estructura jerárquica. En este trabajo, se mostrará el funcionamiento de la metodología con experimentos que invocan a rutinas de las librerías mencionadas: MKL en CPU multicore y Xeon Phi, y cuBLAS en GPU.

2.2.2 Rutinas de álgebra lineal

En este apartado se resumen las características generales de las rutinas con las que se ilustrará, a lo largo de esta tesis, la metodología de optimización jerárquica propuesta. Entre estas rutinas, se encuentra la multiplicación de matrices, que es la rutina con la que se realizará la mayor parte del estudio, pues constituye la rutina básica y de mayor coste computacional sobre la que se basan rutinas de mayor nivel para obtener buenas prestaciones. De estas rutinas, se muestra la multiplicación de Strassen, como ejemplo para ilustrar rutinas que realizan llamadas simultáneas a la multiplicación de matrices básica, y la factorización LU, que sigue un esquema similar a otras factorizaciones (como QR y Cholesky) e invoca a la multiplicación de matrices como su componente computacional básico utilizando un esquema por bloques, frecuente en este tipo de rutinas, para explotar los diferentes niveles de la jerarquía de memoria.

2.2.2.1 Multiplicación de matrices

La multiplicación de matrices que se utiliza, se ejecutará sobre una serie de unidades computacionales (multicore, GPU, MIC, nodos...), por tanto, el objetivo será descomponer la computación matricial que realiza para que pueda ser asignada a cada una de dichas unidades, en las que la multiplicación subyacente se llevará a cabo con rutinas de otras librerías.

Sea la multiplicacion matricial $C = \alpha AB + \beta C$, $A \in R^{m \times k}$, $B \in R^{k \times n}$, $C \in R^{m \times n}$. Esta operación se puede llevar a cabo de diferentes maneras, con el fin de conseguir un reparto equilibrado de la carga de trabajo entre los diferentes elementos computacionales. Se decide replicar la matriz A entre los elementos computacionales, mientras que la matriz B se particiona en bloques de columnas. Evidentemente, la metodología que aquí se presenta se puede aplicar a otras versiones de la multiplicación matricial, y en general, a otras rutinas de álgebra lineal de nivel superior, como la factorización LU, QR o Cholesky, las cuales harán uso de la versión optimizada de la rutina sobre la que se ha aplicado la metodología propuesta. De esta forma, el proceso de auto-optimización se trasladaría de forma implícita a dichas rutinas [55].

Si se considera una plataforma simple, compuesta únicamente por un nodo con una CPU y un coprocesador, la multiplicación matricial se puede expresar como $C = \alpha(AB_1|AB_2) + \beta(C_1|C_2)$, con $B = (B_1|B_2)$ y $C = (C_1|C_2)$, $B_i \in R^{k\times n_i}$, $C_i \in R^{m\times n_i}$ y $n = n_1 + n_2$. La multiplicación $\alpha AB_1 + \beta C_1$ se asignaría al coprocesador y la $\alpha AB_2 + \beta C_2$ a la CPU (figura 2.6). En esta situación, las prestaciones de la rutina irán mejorando conforme el reparto que se haga de las matrices tienda a equilibrar la carga de trabajo entre la CPU y el coprocesador [77, 154, 202, 204].

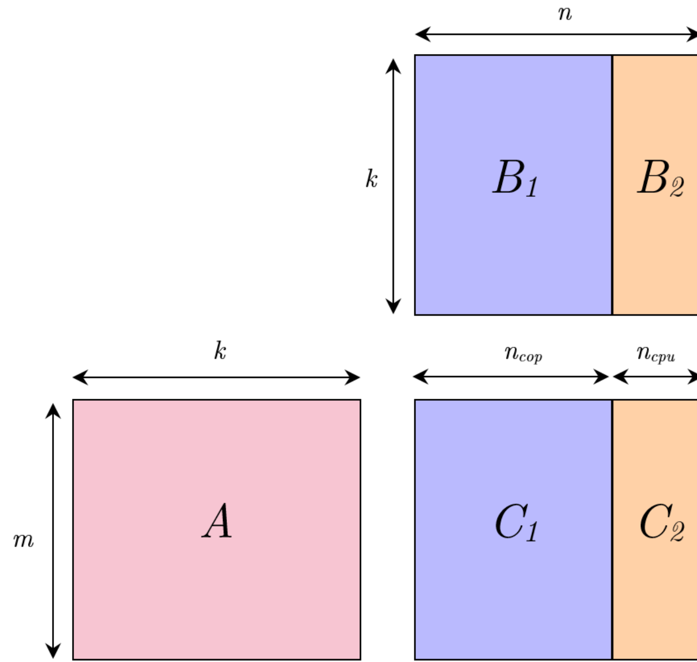

Figura 2.6: Distribución de la multiplicación matricial en un nodo compuesto por una CPU y un coprocesador.

En un nodo más heterogéneo, con una CPU y un conjunto de c coprocesadores, numerados de 1 a c, la multiplicación matricial se puede expresar como $C = \alpha(AB_1|\ldots|AB_{c+1}) + \beta(C_1|\ldots|C_{c+1})$, donde $\alpha AB_i + \beta C_i$, con $1 \leq i \leq c$, se asignaría al coprocesador i y $\alpha AB_{c+1} + \beta C_{c+1}$ a la CPU.

Finalmente, si se considera la situación global, donde la multiplicación matricial se ejecuta utilizando la plataforma completa, formada por N nodos heterogéneos, donde el nodo i está compuesto de una CPU multicore (con p_i cores) y c_i coprocesadores, el conjunto de parámetros ajustables cuyo valor se debe decidir con la metodología de optimización jerárquica al configurar la rutina para su

ejecución sería:

$$\begin{aligned}\{&n_{1,1},\ldots,n_{1,j},\ldots,n_{1,c_1},n_{1,cpu},\ldots,\\ &n_{i,1},\ldots,n_{i,j},\ldots,n_{i,c_i},n_{i,cpu},\ldots,\\ &n_{N,1},\ldots,n_{N,j},\ldots,n_{N,c_N},n_{N,cpu}\},\end{aligned} \tag{2.1}$$

siendo $n_{i,j}$ el tamaño de la submatriz de B asignada al coprocesador j del nodo i ($1 \leq i \leq N$, $1 \leq j \leq c_i$) y $n_{i,cpu}$ la parte asignada a la CPU en dicho nodo i. Por tanto, el espacio de búsqueda de los mejores valores para estos parámetros ajustables, siguiendo un método exhaustivo, sería enorme. Por ejemplo, en el reparto de bloques de columnas ($n_{i,j}$ en la ecuación 2.1) entre elementos computacionales básicos, si se considera como el conjunto de vecinos de una configuración, el formado por todas aquellas que se pueden obtener modificando en una cierta cantidad una de las entradas del vector de asignaciones, y se traspasa esa cantidad a otro de los elementos computacionales, esta vecindad sería de orden $O\left(\left(N+\sum_{i=1}^{N} c_i\right)^2\right)$.

Por otro lado, si además se busca la configuración óptima de los parámetros de cada elemento básico de procesamiento, el espacio de búsqueda aumentaría considerablemente. Por ejemplo, si se considera la búsqueda del número de threads en cada elemento computacional j de cada nodo i, $t_{i,j}$, el conjunto de parámetros ajustables planteado inicialmente en la ecuación 2.1, se expresaría ahora como:

$$\begin{aligned}\{&n_{1,1},\ldots,n_{1,j},\ldots,n_{1,c_1},n_{1,cpu},t_{1,1},\ldots,t_{1,j},\ldots,t_{1,c_1},t_{1,cpu},\ldots,\\ &n_{i,1},\ldots,n_{i,j},\ldots,n_{i,c_i},n_{i,cpu},t_{i,1},\ldots,t_{i,j},\ldots,t_{i,c_i},t_{i,cpu},\ldots,\\ &n_{N,1},\ldots,n_{N,j},\ldots,n_{N,c_N},n_{N,cpu},t_{N,1},\ldots,t_{N,j},\ldots,t_{N,c_N},t_{N,cpu}\}\end{aligned} \tag{2.2}$$

La metodología jerárquica que se propone, permite abordar el problema de forma separada en cada nivel de la plataforma computacional, correspondiendo el primer nivel a las unidades básicas de proceso, el siguiente nivel a los nodos híbridos formados por varias unidades de proceso y, finalmente, el tercer nivel a la plataforma computacional completa con su conjunto de nodos.

2.2.2.2 Multiplicación de Strassen

La multiplicación de Strassen data de 1969 [180]. Tiene un coste computacional de orden $\theta\left(n^{2.8074}\right)$, mientras que la multiplicación tradicional de matrices con tres bucles anidados, tiene coste $\theta\left(n^{3}\right)$. Tras la multiplicación de Strassen, han aparecido algoritmos asintóticamente más eficientes para la multiplicación de matrices [113, 129], pero la multiplicación de Strassen se conoce como "multiplicación rápida" debido a la dificultad de obtener implementaciones de estos nuevos algoritmos que sean competitivas en la práctica con la multiplicación tradicional.

La multiplicación de matrices es el núcleo computacional básico de multitud de rutinas de álgebra lineal que trabajan por bloques, y estas rutinas se utilizan en la resolución de problemas científicos y de ingeniería. Esto ha dado lugar a que existan muchos trabajos de optimización de la multiplicación de matrices tradicional basada en tres bucles, así como versiones altamente optimizadas que explotan el paralelismo de los nuevos sistemas computacionales. Como consecuencia, se ha vuelto más difícil implementar la multiplicación de Strassen de forma que sea competitiva con las rutinas altamente optimizadas para los sistemas actuales. La principal dificultad, estriba en la memoria adicional que la multiplicación de Strassen utiliza para las llamadas recursivas, con lo que el cociente de computación respecto a accesos a memoria disminuye.

Recientemente, se han publicado algunos trabajos sobre la optimización del algoritmo de Strassen para sistemas computacionales actuales [89, 96], pero en el caso que nos ocupa, la finalidad no es optimizar la multiplicación de Strassen, sino usarla como ejemplo de los factores a tener en cuenta para optimizar rutinas de álgebra lineal de nivel medio que realizan varias llamadas a rutinas de nivel inferior. En este caso, se utiliza la multiplicación de matrices tradicional (e implementaciones eficientes de ella) como rutina de bajo nivel.

El algoritmo 1 muestra cómo se lleva a cabo la multiplicación de Strassen, que sigue un esquema recursivo de tipo divide y vencerás [29, 45]. Las matrices que se multiplican (A y B) y la resultante (C), de tamaño $n \times n$, se dividen en cuatro submatrices de tamaño $\frac{n}{2} \times \frac{n}{2}$, y la rutina se invoca recursivamente siete veces. Por un lado, se producen diez sumas y restas de matrices de tamaño $\frac{n}{2} \times \frac{n}{2}$, que se llevan a cabo para generar las submatrices a multiplicar, y por otro, ocho

operaciones del mismo tipo para combinar las submatrices resultantes. En el caso base de la recursión, la multiplicación se realiza directamente mediante una multiplicación básica que normalmente será una versión eficiente y paralelizada de la multiplicación tradicional de matrices. La figura 2.7 muestra la descomposición de tareas en una multiplicación de Strassen con un nivel de recursión.

Algoritmo 1: Esquema algorítmico de la multiplicación de Strassen.

Input: Dimensión de las matrices, n.
Matrices A y B, de tamaño $n \times n$.
Tamaño del caso base, $base$.
Result: $C = AB$
if $n \leq base$ **then**
  C=multiplicacionBasica(A, B);
else
  strassen($A_{11} + A_{22}$, $B_{11} + B_{22}$, P, $\frac{n}{2}$, $base$);
  strassen($A_{21} + A_{22}$, B_{11}, Q, $\frac{n}{2}$, $base$);
  strassen(A_{11}, $B_{12} - B_{22}$, R, $\frac{n}{2}$, $base$);
  strassen(A_{22}, $B_{21} - B_{11}$, S, $\frac{n}{2}$, $base$);
  strassen($A_{11} + A_{12}$, B_{22}, T, $\frac{n}{2}$, $base$);
  strassen($A_{21} - A_{11}$, $B_{11} + B_{12}$, U, $\frac{n}{2}$, $base$);
  strassen($A_{12} - A_{22}$, $B_{11} + B_{22}$, V, $\frac{n}{2}$, $base$);
  $C_{11} = P + S - T + V$;
  $C_{12} = R + T$;
  $C_{21} = Q + S$;
  $C_{22} = P - Q + R + U$;
end

En un sistema computacional sobre el que se ha optimizado la multiplicación de matrices tradicional, la multiplicación de Strassen se podría llevar a cabo llamando a esa multiplicación básica utilizando todo el sistema, con lo que las multiplicaciones básicas se ejecutarían una tras otra explotando cada una el paralelismo del sistema. De esta forma, se trabajaría jerárquicamente en la optimización de la rutina de Strassen, delegando a la operación de nivel inferior (la multiplicación directa) parte de la optimización, pero quedando por determinar el valor de *base* con el que se obtiene el menor tiempo. El tiempo de ejecución viene dado por la ecuación de recurrencia $t(n) = 7t\left(\frac{n}{2}\right) + \frac{9}{4}n^2$, de la que se obtiene que el coste es de orden $\theta\left(n^{2.8074}\right)$. Para obtener el nivel de recursión óptimo (valor de *base* en

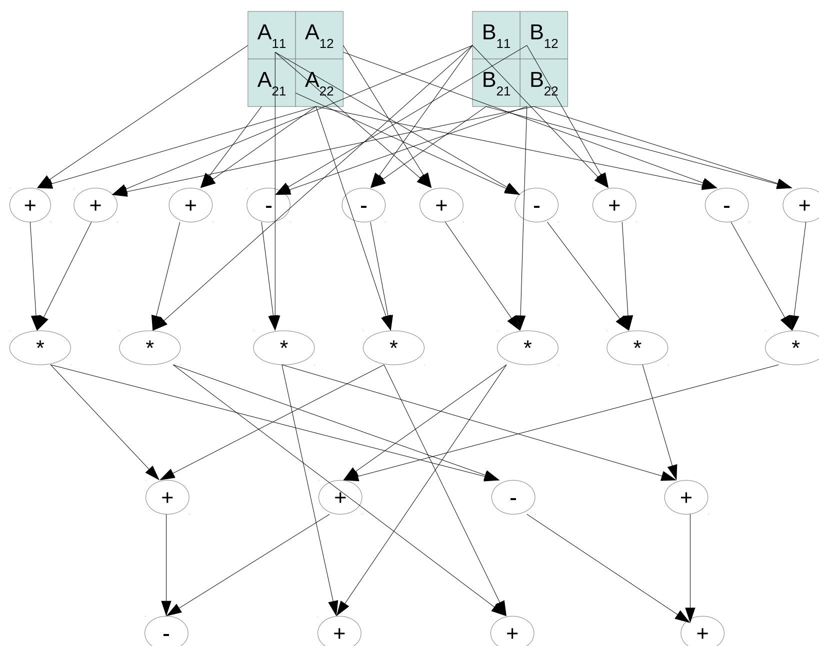

Figura 2.7: Descomposición de tareas en una multiplicación de Strassen con un nivel de recursión.

el esquema algorítmico), si expandimos la recursión se obtiene:

$$t(n) = 2k_3 n^{\log 7} base^{\log 7} + k_2 \frac{4}{3}\left(1 - \left(\frac{base}{n}\right)^2\right) \tag{2.3}$$

donde k_3 y k_2 representan los tiempos de ejecución de una multiplicación y una suma o resta de dos números en doble precisión. El valor óptimo de $base$ para una matrix de tamaño $n \times n$, se puede estimar derivando la ecuación 2.3 con respecto a $base$ e igualando a cero para obtener $base$ como función de n. Así, en un sistema computacional dado, si se estiman los valores de k_3 y k_2, el valor de $base$ se puede obtener con la función que relaciona $base$ con n. Pero el problema no es tan simple, pues los valores de k_3 y k_2 no son fijos, sino que dependen del tamaño de las matrices con las que se trabaja y la forma en que se almacenan los datos en memoria, y, cuando se explota el paralelismo, también dependen del número de cores que trabajan.

Adicionalmente, puede ser necesario considerar también la memoria consumida, que depende del orden en que se ejecutan las instrucciones en el algoritmo 1, y que tiene un coste:

$$m(n) = 3n^2 + \frac{s}{3}n^2\left(1 - \left(\frac{base}{n}\right)^2\right) \qquad (2.4)$$

donde s representa el número de submatrices usado para almacenar matrices temporales en las llamadas recursivas. Si estas llamadas se hacen en el orden en que aparecen en el algoritmo, sólo se necesitan dos matrices temporales, y la cantidad de memoria necesaria es $\frac{11}{3}n^2$. No obstante, puede ser preferible llevar a cabo las siete multiplicaciones simultáneamente (obteniéndose un paralelismo de mayor granularidad). En este caso, la memoria usada para una multiplicación no se reúsa para otra, y $s = 21$, lo que da una memoria total de alrededor de $10n^2$. Además, cada una de las multiplicaciones básicas no hará uso ahora de todo el sistema computacional, sino de un subsistema formado por varias de sus unidades computacionales, con lo que habría que determinar las unidades para cada una de las multiplicaciones. Esto da lugar a un problema de asignación del grafo de dependencias de la figura 2.7 a los posibles subsistemas del sistema computacional en que se trabaja. Por ejemplo, si realizamos siete multiplicaciones básicas en **jupiter**, con un total de siete unidades computacionales (CPU multicore y seis GPUs), se puede asignar una multiplicación a cada unidad. Pero de este modo, el tiempo vendría determinado por el de la unidad más lenta. Otra posible opción, es agrupar las unidades en dos subnodos, cada uno con seis cores, una GPU de las más rápidas y dos GPUs de las más lentas. De esta forma, las siete multiplicaciones se podrían ejecutar en tres pares de dos simultáneamente, cada una ejecutada con uno de los subnodos, y una última multiplicación con el nodo completo. Hay muchas más posibilidades de creación de subnodos y de agrupación de multiplicaciones y asignación a subnodos, por lo que la optimización a nivel de la rutina de Strassen se complica enormemente, aún usando multiplicaciones básicas optimizadas para los subnodos seleccionados. En este sentido, la multiplicación de Strassen nos servirá para mostrar las técnicas de optimización jerárquica en el caso de rutinas con llamadas simultáneas a rutinas de nivel inferior. Este tipo de rutinas aparecen con frecuencia en computación científica [23].

2.2.2.3 Factorización LU

La factorización LU sigue un esquema similar al de otras factorizaciones matriciales (QR y Cholesky) [91]. En su versión por bloques, tiene como componente computacional principal la rutina de multiplicación de matrices, por lo que se puede delegar a dicha rutina de multiplicación, la obtención de una versión optimizada para un sistema computacional heterogéneo.

La figura 2.8 muestra cómo se descomponen las operaciones de computación llevadas a cabo en un paso de la factorización por bloques, junto con una posible asignación de estas operaciones a las unidades computacionales básicas, considerando un nodo compuesto por CPU+GPU. En naranja se muestra la factorización de un panel y en verde la resolución de un sistema triangular múltiple. Estas dos operaciones pueden realizarse en CPU o GPU, y conllevan transferencias de datos entre las dos unidades computacionales. La multiplicación de matrices, en cambio, se descompone asignando una parte a la GPU (azul) y otra a la CPU (morado).

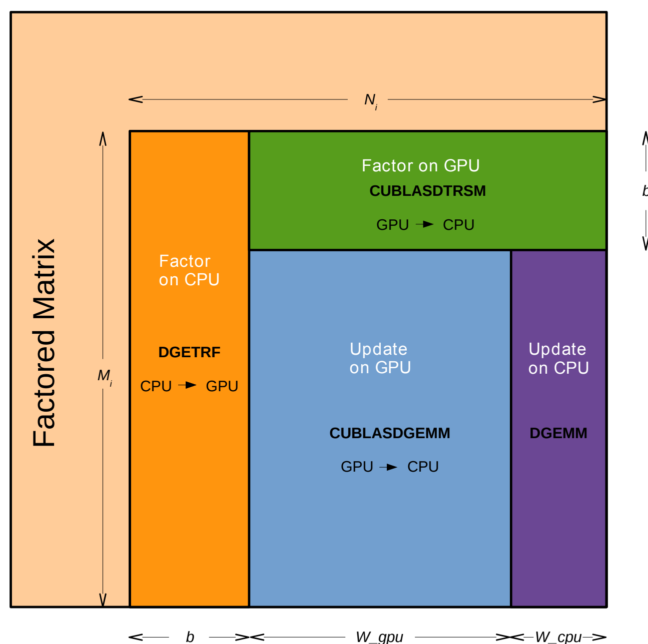

Figura 2.8: Operaciones de computación realizadas por diferentes unidades de cómputo (CPU, GPU) en la factorización LU por bloques.

Esta descomposición de la multiplicación se realizaría implícitamente si se invocase a una multiplicación de matrices ya optimizada para CPU+GPU, pues la propia multiplicación sería la encargada de decidir el reparto de trabajo entre las unidades de cómputo, que además puede ser diferente en cada paso de la descomposición por bloques, pues el tamaño de la matriz con el que se trabaja es distinto. No obstante, aún usando una multiplicación optimizada, es necesario determinar el tamaño de bloque, $\boldsymbol{b}$, con el que se obtiene el menor tiempo de ejecución, por lo que en esta rutina de nivel medio aparecería un nuevo parámetro a determinar, o una serie de parámetros, pues los valores de $\boldsymbol{b}$ también pueden ser distintos en los distintos pasos de la factorización. Además, se tienen varias operaciones cuya ejecución puede realizarse de forma simultánea, por lo que habrá que decidir, por un lado, cómo asignar estas operaciones a las unidades computacionales, y por otro, si las operaciones se llevan a cabo una tras otra explotando en cada caso el sistema completo, o si se utilizan subnodos para trabajar en algunas de las operaciones simultáneamente, al igual que en la multiplicación de Strassen.

2.3 Conclusiones

La metodología de optimización jerárquica que se analizará en capítulos posteriores ha sido diseñada para trabajar de forma simultánea en dos direcciones: hardware y software.

En el hardware, se pueden tener distintos niveles, desde el más bajo, compuesto por las unidades computacionales básicas (CPU, GPU, MIC), hasta otros que se obtienen pasando por niveles intermedios en los que se combinan elementos computacionales de niveles inferiores. Se han comentado las características generales de las unidades del nivel más bajo, de los nodos compuestos por dichas unidades básicas, y de clusters compuestos con nodos, dando lugar a una jerarquía de tres niveles. Aunque se trabajará normalmente con estos tres niveles descritos, la metodología propuesta es igualmente válida para otras jerarquías con estructura diferente.

Una situación similar se produce en la jerarquía software, donde se consideran las rutinas de nivel básico (como las ofrecidas por la librería BLAS) y otras de segundo nivel que se basan en ellas, pudiendo seguir ampliando la jerarquía con rutinas que se basan en las de nivel intermedio, etc. Se han comentado los parámetros que habría que determinar en cada nivel de rutinas para su optimización en un entorno computacional heterogéneo, y se han mostrado como ejemplos de rutinas, la multiplicación tradicional de matrices, la multiplicación de Strassen y la factorización LU, que serán las que se usen principalmente a lo largo de la tesis (junto con otras factorizaciones matriciales). No obstante, la metodología de auto-optimización es igualmente válida para otras rutinas con estructura similar.

Capítulo 3

Técnicas de optimización y auto-optimización

Este capítulo muestra, en primer lugar, un recorrido por las propuestas de optimización y auto-optimización de software científico de los últimos años. Tras ello, se presenta una posible clasificación de las técnicas de auto-optimización, en función de la metodología general que utilizan: teórica, experimental o híbrida. Finalmente, se describen las propuestas de auto-optimización básica consideradas en cada una de estas tres vertientes, que han supuesto el punto de partida del planteamiento de auto-optimización jerárquica presentado en esta tesis.

3.1 Ideas generales

Como se ha mencionado, los sistemas computacionales actuales tienen una gran complejidad, siendo habitualmente de naturaleza híbrida y heterogénea, pues disponen de componentes con arquitecturas y capacidades de cómputo muy diversas. Un ejemplo de ello se puede apreciar en la lista TOP500 [141]. Estos entornos computacionales incluyen unidades de procesamiento paralelo como GPUs [118, 157], MICs [43, 168], sistemas masivamente paralelos [119] o virtualización de GPUs [175]. Por tanto, para explotar eficientemente estos sistemas en la resolución de problemas científicos con alta demanda computacional, se hace necesario

que la programación se lleve a cabo combinando diferentes paradigmas y entornos de programación, lo que da lugar a un esquema de programación híbrida. De esta forma, el problema de optimización de software para estos sistemas computacionales requiere de un amplio conocimiento en diferentes campos de estudio: paralelismo, paradigmas de programación paralela, software propiamente dicho y arquitectura de los diferentes componentes del hardware. Frente a este planteamiento, una alternativa para obtener rutinas optimizadas consiste en delegar a las propias rutinas la explotación eficiente de los recursos computacionales, lo que requiere dotar a dichas rutinas de capacidad para adaptarse de forma autónoma a las características de los diferentes componentes computacionales donde se va a llevar a cabo su ejecución.

3.2 Trabajos relacionados

Esta sección traza un posible recorrido por las aportaciones más significativas que se han realizado en los últimos años en el campo de la auto-optimización (*auto-tuning*) de rutinas. Estas aportaciones se han mencionado en el Capítulo 1 en la sección relativa al estado del arte, por tanto, aquí se indicarán algunos de los avances destacables producidos en el campo de las técnicas de auto-optimización. Todas estas técnicas tienen como objetivo final dotar al software de capacidad para adaptarse de forma autónoma al sistema computacional donde se ejecuta. De manera general, la principal motivación que ha suscitado el nacimiento y desarrollo de este campo de estudio, ha sido el hecho de que no siempre está claro qué tipo de sistema es el más adecuado para resolver un problema [130] o cuál es la mejor forma de resolverlo en un sistema determinado. Por todo ello, uno de las principales retos en las diferentes propuestas suele ser cómo conseguir una buena predicción de las prestaciones que ofrecen las rutinas cuando se ejecutan en estos sistemas de altas prestaciones [82].

Un trabajo pionero en este campo se remonta al año 1994 con la tesis de Brewer [30], en la que se propone un método de selección automática del mejor algoritmo, entre varios previstos, para resolver cada problema planteado (poli-algoritmo). Del mismo modo, también selecciona automáticamente el valor de ciertos parámetros de estos algoritmos. El método en su conjunto se basa en el

modelado del tiempo de ejecución de los algoritmos, donde estos modelos se obtienen aunando dos técnicas: ejecución monitorizada de unos pocos problemas de pequeño tamaño (*profiling*) y modelado estadístico para generalizar el comportamiento de dichas ejecuciones. La propuesta se aplica en diversas plataformas paralelas homogéneas (multiprocesadores y clusters) a rutinas para la resolución de ecuaciones diferenciales parciales y para la ordenación de datos mediante el método *radix sort*. Más adelante, también en este tipo de plataformas, surgieron técnicas específicas para problemas numericos, como el proyecto ILIB-FIBER [115, 116, 124], donde se plantea una metodología de ajuste automático *ad hoc* para cada rutina (*auto-tuning software for dedicated usage*). Por otro lado, en [125], se proponen técnicas de paralelización+optimización automática de algoritmos de álgebra lineal basados en el modelado del flujo de datos entre tareas en CPUs multicore. En [99] se expone un planteamiento de auto-optimización para la rutina de multiplicación de matrices `PDGEMM`, perteneciente a la librería paralela PBLAS. El método propuesto, consiste en realizar una búsqueda heurística experimental del tamaño de bloque a utilizar. En [65], el prestigioso grupo de investigación del ICL en la Universidad de Tennessee, asentó las principales bases en el campo de optimización automática de software de altas prestaciones, estableciendo tres grandes retos en los que se viene trabajando desde entonces: las decisiones a nivel algorítmico, a nivel de la plataforma de ejecución y a nivel de la programación de las rutinas básicas (*kernels* computacionales).

En los últimos años, nuestro grupo de investigación [159] ha llevado a cabo el desarrollo de una serie de trabajos sobre auto-optimización de rutinas paralelas de álgebra lineal:

- En [145] se aborda el ajuste automático de rutinas paralelas desde un punto de vista unificado, en base al modelado teórico-experimental del tiempo de ejecución de cada rutina. Este modelo, contiene dos conjuntos de parámetros: por un lado, los parámetros del sistema, SP (*System Parameters*), que determinan la influencia de la plataforma en el tiempo de ejecución de la rutina y, por otro, los parámetros algorítmicos, AP (*Algorithmic Parameters*), cuyos valores se deben eligir de manera automática de cara a minimizar el tiempo de ejecución de dicha rutina. Para dotar de naturaleza teórico-experimental a este modelo, se parte de un diseño inicial basado

en la complejidad del algoritmo, al que se le confiere una naturaleza más realista mediante el cálculo experimental del valor de los SP que aparecen. Esta propuesta se aplica a sistemas paralelos homogéneos, tanto de memoria compartida como distribuida.

- En [47] se plantea un sistema de compilación múltiple (*Poly-Compilation Engine*, PCE) de rutinas paralelas para sistemas paralelos de memoria compartida. Dada una rutina, el PCE se encarga de generar una versión ejecutable de ésta por cada compilador disponible en la plataforma. De igual manera, el PCE crea el modelo teórico-experimental del tiempo de ejecución de cada versión, donde los SP reflejan tanto el coste de las operaciones numéricas, como los costes asociados a la creación y gestión de threads. En tiempo de ejecución, el PCE selecciona el mejor ejecutable de la rutina básica, así como los valores para el resto de los AP.

- En [48] se presentan un conjunto de ideas iniciales sobre cómo se podría modelar el tiempo de ejecución de las rutinas de manera que se contemple la heterogeneidad del tiempo de acceso a los datos en plataformas con acceso no uniforme a memoria (NUMA). El software objeto de la auto-optimización cuenta con una estructura paralela anidada que permite ajustar, en cierta medida, el mapa de reparto de la computación a la estructura jerárquica de la plataforma NUMA.

- En [51] se estudia la adaptación de la metodología de auto-optimización a sistemas heterogéneos formados por nodos que disponen de una CPU multicore junto a uno o varios aceleradores (coprocesadores). En tiempo de instalación, se realiza de forma experimental una búsqueda local de los mejores valores de los AP para cada uno de los tamaños de problema que se plantean como conjunto de instalación. Posteriormente, en tiempo de ejecución, dado un problema específico, los valores seleccionados para los AP estarán basados en los valores de estos AP para los tamaños del conjunto de instalación más cercanos al del problema a resolver. Por tanto, en este trabajo, no se diseña ni maneja ningún modelo del tiempo de ejecución de las rutinas. Todo el conocimiento del comportamiento de cada rutina se adquiere mediante experimentación.

Se comentan las características generales de algunos trabajos de optimización orientados a algunos tipos de problemas y/o algunos sistemas de cómputo específicos:

- ATLAS (*Auto Tuning Linear Algebra Subroutines*) [198]: implementación de BLAS con auto-optimización que dispone de un generador de código que determina experimentalmente, en tiempo de compilación, el tamaño de bloque y el grado de desenrrollado de bucles necesarios para obtener un núcleo interno de la multiplicación matricial, `GEMM`, que ofrezca las máximas prestaciones al ejecutarse *on-chip*. Por otro lado, implementa dos posibles algoritmos (*off-chip*) para el núcleo externo que envuelve al núcleo interno de dicha rutina. Esta parte externa es común para todas las plataformas.

- Técnicas para CPU multicore, como las presentadas en [48], donde se plantea cómo introducir la posible heterogeneidad del tiempo de acceso a los datos (plataformas NUMA) en el modelado del tiempo de ejecución de las rutinas. Por otro lado, en [52] se plantea una selección automática de la mejor versión compilada de cada rutina en función de las características de la plataforma.

- En sistemas manycore, como en [87], donde se realiza una búsqueda guiada experimental en tiempo de instalación de los mejores valores para los parámetros ajustables de cada coprocesador del sistema.

- En sistemas heterogéneos, como en [111, 128], donde partiendo de la metodología de distribución cíclica por bloques tradicionalmente utilizada en sistemas distribuidos homogéneos, y proponiendo un modelado de las prestaciones de cada procesador de la plataforma, se plantea una distribución heterogénea del trabajo en base a las diferentes capacidades de cómputo de los procesadores.

- En sistemas de carga variable, como en [46, 56], donde durante el proceso de instalación del software en una plataforma, se optimizan automáticamente las rutinas conforme a las características de dicha plataforma. Posteriormente, en tiempo de ejecución, los parámetros que definen estas características se ajustan a las condiciones presentes en el sistema, tanto de

carga de cómputo de los procesadores, como de carga de tráfico en la red de interconexión.

- Diferentes técnicas basadas en modelado, como en [53], donde se diseña un modelo analítico teórico práctico de rutinas paralelas para sistemas distribuidos que utilizan el paradigma de paso de mensajes mediante MPI. El modelo cuenta con una serie de parámetros donde recoger las características del sistema junto a otro conjunto de parámetros algorítmicos cuyos valores se pueden ajustar libremente para optimizar el tiempo de ejecución. Posteriormente, en [49] se realiza un refinamiento del modelado considerando los parámetros del sistema como funciones dependientes de la operación básica que se realice y del tamaño de problema utilizado. Por otro lado, en [84] se muestra cómo el modelo del tiempo de ejecución resulta igualmente de utilidad en sistemas heterogéneos, con más de una red de interconexión, quedando recogidas estas características en los parámetros del sistema. La decisión de cómo mapear los procesos en los diferentes procesadores aparece como un nuevo parámetro algorítmico para estas plataformas. A continuación, en [86] se plantea una técnica de remodelado de las rutinas básicas, utilizando regresión polinómica en aquellas situaciones donde los modelos originales no proporcionen información suficientemente precisa de cara a poder utilizarla para modelar, a su vez, aquellas rutinas de mayor nivel que hacen uso de las básicas. Finalmente, en [50] el planteamiento de modelado teórico-práctico se extiende a plataformas híbridas que cuentan con uno o varios coprocesadores, con la consiguiente variación en parámetros del sistema y parámetros ajustables.

- Técnicas basadas en el diseño jerárquico de librerías, como en [58], donde se plantea una visión jerárquica del modelado de rutinas de álgebra lineal en base a la jerarquía de librerías sobre las que se sustenta la librería ScaLAPACK. A continuación, en [55] se amplía la visión de la metodología de auto-optimización, considerando la jerarquía de librerías donde se encuentra inmersa la librería a la que pertenece cada rutina a tratar. De esta forma, dado que cada rutina tiene en su código diferentes llamadas a rutinas de librerías pertenecientes a niveles inferiores en la jerarquía, se puede utilizar la información de auto-optimización de estas rutinas básicas para confor-

mar la correspondiente a la rutina con la que se está tratando. Finalmente, en [85] se ahonda en la inclusión de técnicas de auto-optimización en la jerarquia de librerías sobre las que se sustenta ScaLAPACK, planteando un modelado de las rutinas de alto nivel en tres fases: una primera fase donde se realiza el modelo en base a los submodelos de las rutinas de nivel inferior utilizadas, una segunda fase donde se chequea la precisión del modelo obtenido, y una tercera fase donde, en el caso de querer mejorar la precisión del modelo, se realiza un remodelado de la rutina mediante diferentes técnicas de aproximación polinomial.

- Algoritmos por *tiles* en multicore [32], donde el algoritmo de cada rutina es representado como una secuencia de tareas que operan sobre bloques de datos (*tiles*). La distribución de estas tareas entre los distintos procesadores se realiza dinámicamente, teniendo en cuenta las dependencias temporales y de datos entre ellas, así como la disponibilidad de los recursos computacionales.

- Algoritmos tipo *stencil* [59], donde se realizan sucesivas iteraciones de cálculos sobre mallas de datos. En esta situación, resulta crucial (auto)ajustar una reordenación de las operaciones de cara a sacar máximo provecho de la jerarquía de memoria existente en la plataforma hardware utilizada.

- Algoritmos sobre matrices dispersas [114, 170], donde las características de los datos de entrada deben ser consideradas como un factor clave en la metodología de auto-optimización, ya que condicionan en gran medida cuáles son las mejores decisiones que se pueden tomar a la hora de ejecutar cada rutina.

- Trabajos en la línea de esta tesis, las técnicas de optimización jerárquica, como en [95], donde se presenta una propuesta para organizar jerárquicamente el esquema de comunicaciones de la rutina de multiplicacion de matrices de cara a sacar más provecho de las plataformas actuales (miles de cores interconectados de manera heterogénea). En [162] se puede encontrar una propuesta de organización jerárquica global del espacio de búsqueda de los valores óptimos para los parámetros ajustables de cada rutina, basándose en las dependencias que puedan existir entre estos parámetros. Por otro lado,

en [200] se plantea otra visión jerárquica del proceso de auto-optimización en base a realizar, para cada rutina, tanto las optimizaciones más internas (desenrollado de bucles, vectorización...) como aquellas a nivel algorítmico (descomposición del problema, número de threads...).

Las propuestas de auto-optimización, también se han extendido a otros tipos de rutinas básicas, por ejemplo:

- Cálculo de valores propios: en [117] se realiza un ajuste automático en 2 fases: en una primera fase se aproxima el tiempo de ejecución previsto mediante medidas experimentales, utilizando una función polinomial lineal y, posteriormente, en la fase de ejecución se realiza un equilibrado del reparto de la carga de trabajo en base a las prestaciones medidas para cada unidad de cómputo.

- Transformada rapida de Fourier: en [133] se propone una librería de altas prestaciones en base a la auto-optimización de esta rutina para varias GPUs.

De igual forma, resulta de igual utilidad la aplicación de auto-optimización a aplicaciones de mayor nivel, donde el código suele ser muy extenso y complejo, lo que conlleva que cualquier intento de optimización manual resulte, en general, inabordable. Algunos ejemplos serían:

- Análisis climático: en [110] se diseña un modelo del tiempo de ejecución de la aplicación. Los valores de los parámetros del sistema que aparecen en el modelo, se calculan en tiempo de instalación y, posteriormente, a la hora de ejecutar la rutina, se deciden automáticamente los valores de los parámatros ajustables del modelo: número de cores y número de subperiodos del tiempo que se quieren simular para tratar de forma simultánea.

- Procesado de señal: en [163] se integra un motor de búsqueda en el software para que, dada una plataforma de cómputo, encuentre la mejor formulación de resolución y la mejor implementación (polialgoritmo).

3.3 Clasificación general de las técnicas de optimización automática

Como colofón al recorrido realizado en la sección anterior a lo largo de las aportaciones llevadas a cabo en el campo de la optimización automática de software, se podría establecer una clasificación general de estas técnicas en función de la metodología que utilizan como núcleo principal de su propuesta:

- Por un lado, se encuentran las que se apoyan en algún tipo de modelado del tiempo de ejecución de la rutina a tratar, así como de la plataforma de ejecución. Para este tipo de propuestas, resulta crucial un conocimiento inicial del algoritmo que utiliza la rutina y, en cualquier caso, debe venir acompañado de un estudio experimental inicial que establezca, de alguna forma, la capacidad de cómputo y de comunicación de la plataforma, bien de manera general o bien específicamente para cada rutina.
- Por otro lado, se encuentran propuestas meramente experimentales, donde inicialmente no se precisa tener ningún conocimiento sobre el algoritmo o la implementación de cada rutina. En este tipo de técnicas, se lleva a cabo un conjunto de experimentos con cada rutina, lo que permite obtener una visión aproximada de su comportamiento, es decir, de las prestaciones que ofrece conforme varía el tamaño de problema a resolver y el valor de sus parámetros ajustables.
- Finalmente, existen también aproximaciones híbridas que combinan ambas técnicas, modelado y experimentación. Estas aproximaciones suelen tener una primera fase de modelado que sirve para obtener un visión general del comportamiento de la rutina en la plataforma. Tras ello, se lleva a cabo un conjunto reducido de experimentos dirigidos a perfilar esa imagen obtenida con el modelado.

A continuación se describirán de manera resumida los trabajos de auto-optimización que se han llevado a cabo en el marco de cada una de las tres líneas trazadas con esta clasificación. Estos trabajos han supuesto, en gran medida, el punto de

partida para el planteamiento global de la metodología jerárquica que se presenta en esta tesis. Tal y como se irá describiendo, estas propuestas centran su trabajo en la auto-optimización de software para plataformas NUMA, sistemas con una organización jerárquica de la memoria compartida, lo que conlleva un tiempo de acceso no uniforme a las diferentes posiciones de memoria. En estas plataformas, conseguir el uso eficiente de los diferentes niveles de la jerarquía de memoria no es una labor sencilla, lo que produce que las prestaciones obtenidas con implementaciones multihilo no crezcan de forma proporcional al número de cores que se están utilizando. De cara a mejorar esta situación, se pueden desarrollar rutinas con paralelismo multinivel que combinen paralelismo a nivel de thread OpenMP y a nivel de rutina básica, como por ejemplo rutinas tipo BLAS. Para este tipo de rutinas, se hace necesario diseñar técnicas de auto-optimización que sean capaces de seleccionar los valores de los principales parámetros ajustables, en este caso, el número de threads a utilizar en cada nivel de la rutina. Además, hay que tener en cuenta que, para conseguir mejorar las prestaciones en estos sistemas, se precisa utilizar frecuentemente nuevos métodos de programación, así como algoritmos que hagan un uso eficiente de este tipo de arquitecturas complejas. En el ámbito de álgebra lineal densa, PLASMA [32] y FLAME [93] son ejemplos de librerías que se han diseñado con dicho propósito.

Los experimentos mostrados en estos trabajos se han realizado en diferentes sistemas de memoria compartida, desde sistemas NUMA de tamaño medio hasta grandes sistemas cc-NUMA:

- **saturno**: nodo de cómputo del mencionado cluster **Heterosolar**. Este nodo es un sistema NUMA con 32 GB de memoria compartida y 24 cores, organizados en 4 sockets hexa-core Intel Xeon E7530 a 1.87GHz.

- **hipatia**: cluster de la Universidad Politécnica de Cartagena, formado por 14 nodos con 2 Intel Xeon Quad-Core (8 cores por nodo) a 2.80 GHz y 2 nodos con 4 Intel Xeon Quad-Core (16 cores por nodo) a 2.93 GHz, de lo que se han utilizado los nodos con 16 cores.

- **ben**: nodo del ya cerrado Centro de Supercomputación de Murcia. Este nodo contaba con 1.5 TB de memoria compartida y 128 cores organizados en procesadores Intel-Itanium-2 Dual-Core processors a 1.6 GHz.

- **pirineus**: sistema del antiguo CESCA (Centre de Serveis Científics i Acadèmics de Catalunya), con 6.14 TB de memoria compartida y 1344 cores organizados en procesadores Intel Xeon X7542 Hexa-Core a 2.67 GHz, de los cuales sólo se pueden utilizar un máximo de 256 cores en paralelo.
- **joule**: sistema NUMA de la Universidad Jaume I de Castellón, con 64 GB de memoria compartida y 4 procesadores AMD Opteron 6276 con 16 cores cada uno (64 cores en total) a 2.3 GHz.

3.4 Optimización automática basada en el modelado del tiempo de ejecución

En [34], el método de auto-optimización propuesto se encarga de seleccionar automáticamente el número óptimo de threads a usar en cada uno de los niveles de paralelismo cuando las rutinas de álgebra lineal se invocan desde código paralelizado con OpenMP. El método está basado en el modelado del tiempo de ejecución para software con dos niveles de paralelismo, cuando se ejecutan en plataformas que pueden llegar a tener una amplia jerarquía de memoria heterogénea, como ocurre en plataformas NUMA. La metodología se aplica a una rutina de multiplicación de matrices con dos niveles de paralelismo y a diferentes factorizaciones matriciales por bloques (LU, QR y Cholesky).

Para cada rutina, el núcleo del sistema de auto-optimización diseñado (*Automatic Tuning System*, ATS), contiene un modelo del tiempo de ejecución de la rutina, donde se incluyen las características de la plataforma (hardware + librerías básicas instaladas) en forma de Parámetros del Sistema, SP, y un conjunto de parámetros algorítmicos, AP, cuyo valor debe ser seleccionado de forma apropiada por el ATS con el fin de reducir el tiempo de ejecución de la rutina. Así pues, la fórmula del tiempo de ejecución es de la forma:

$$T_{exe} = f(SP, n, AP) \tag{3.1}$$

donde n representa el tamaño del problema a resolver.

Con carácter general, para llevar a cabo una operación aritmética básica en el interior de una rutina, la probabilidad de encontrar los operandos en el nivel de memoria más próximo a la CPU, depende de la localidad de los datos en el algoritmo implementado. Por esta razón, el tiempo promedio para realizar una operación básica (incluido el acceso a memoria) se representa en el modelo con un parámetro específico para cada rutina r, k_r, de cara a tener un modo sencillo, pero efectivo, de considerar esta situación.

En este trabajo, a partir de ese planteamiento general, se precisa una adaptación de cara a considerar la características propias de grandes plataformas NUMA. De esta forma, el tiempo de acceso a cada operando de la operación debe considerarse como una función de la distancia que hay entre el core que va a realizar el cálculo y el lugar físico donde se encuentra el dato.

La tabla 3.1 muestra una comparación de los tiempos de ejecución obtenidos en las plataformas **ben** y **saturno**. En las columnas se representa:

- `MKL-ORA`: mínimo tiempo que se podría obtener idealmente con un oráculo perfecto que siempre generase la configuración óptima de dos niveles de threads (OpenMP $\times$ MKL).
- `MKL-AC`: tiempo obtenido si se generan tantos threads MKL como cores tenga la plataforma.
- `MKL-DYN`: tiempo obtenido si se activa la selección dinámica de threads MKL.
- `MKL-ATS`: tiempo obtenido con la configuración de 2 niveles de threads que decide el sistema de auto-optimización ATS.

En ambas plataformas, se puede apreciar cómo la selección automática de los valores de los AP que realiza el ATS produce mejores resultados que los que obtendría un usuario que utilice la rutina MKL con tantos threads como cores de la plataforma o haciendo uso de la selección automática del número de threads que proporciona la propia librería MKL. En todos los casos, con ATS los tiempos obtenidos son muy cercanos a los óptimos ideales, y mejores que si se utiliza directamente la rutina de MKL.

n	MKL-ORA	MKL-AC	MKL-DYN	MKL-ATS
		saturno		
1000	0.014	0.044	0.049	0.016 (1×18)
2000	0.059	0.146	0.145	0.059 (2×12)
3000	0.119	0.141	0.146	0.127 (2×12)
4000	0.203	0.221	0.241	0.208 (2×12)
5000	0.251	0.335	0.333	0.306 (2×12)
		ben		
1000	0.024	0.091	0.098	0.012 (2×8)
2000	0.07	0.39	0.40	0.07 (4×16)
3000	0.23	0.82	0.81	0.23 (4×16)
4000	0.59	1.40	1.41	0.74 (4×32)
5000	1.12	2.11	2.10	1.44 (4×32)

Tabla 3.1: Tiempos de ejecución (en segundos) obtenidos con la rutina de multiplicación de matrices generando el número óptimo de threads (`MKL-ORA`); generando tantos threads MKL como cores de la plataforma (`MKL-AC`); activando la selección dinámica de threads MKL (`MKL-DYN`) y utilizando la multiplicación de dos niveles con la configuración de threads (entre paréntesis) seleccionada por el sistema de auto-tuning (`MKL-ATS`).

En la tabla 3.2 se muestra una comparación de los tiempos de ejecución obtenidos con la rutina de factorización LU por bloques. Al igual que sucedía con la rutina de multiplicación de matrices, ATS mejora el tiempo de ejecución que obtendría un usuario generando tantos threads como cores o dejando que MKL seleccionara dinámicamente el número de threads por sí mismo. La mejora es más patente para problemas de gran tamaño, para los que es más necesaria una reducción del tiempo de ejecución.

3.5 Optimización automática híbrida: modelo y experimentación

En [33] se plantea como situación de partida una rutina de la que no se encuentra disponible el modelo del tiempo de ejecución, por lo que es necesario llevar a cabo un conjunto de experimentos de cara a analizar el comportamiento de la rutina. Este modelo diseñado con experimentación podrá utilizarse posteriormente para determinar los valores apropiados de los parámetros algorítmicos

n	MKL-ORA	MKL-AC	MKL-DYN	MKL-ATS
saturno				
1000	0.06	0.06	0.06	0.12
2000	0.12	0.20	0.12	0.15
3000	0.26	0.26	0.27	0.33
4000	0.54	0.76	0.62	0.59
5000	0.88	1.00	0.98	0.88
6000	1.44	1.44	1.46	1.49
7000	1.98	2.19	2.22	1.98
8000	2.76	3.14	3.07	2.76
9000	4.00	4.42	4.50	4.00
10000	5.17	5.30	5.27	5.17
ben				
1000	0.06	0.34	0.35	0.25
2000	0.17	0.79	0.70	0.42
3000	0.54	2.14	2.13	0.85
4000	0.95	3.39	3.35	1.32
5000	1.64	6.36	6.32	2.21
6000	2.63	8.32	8.34	2.83
7000	3.84	10.20	10.25	4.55
8000	5.19	14.49	14.45	5.39
9000	6.95	20.21	20.35	7.30
10000	8.12	22.28	22.21	8.70

Tabla 3.2: Tiempos de ejecución (en segundos) obtenidos con la rutina de factorización LU por bloques generando el número óptimo de threads (`MKL-ORA`); generando tantos threads MKL como cores de la plataforma (`MKL-AC`); activando la selección dinámica de threads MKL (`MKL-DYN`) y utilizando la multiplicación de dos niveles con la configuración de threads (entre paréntesis) seleccionada por el sistema de auto-tuning (`MKL-ATS`).

que incluya (número de threads, tamaño/s de bloque/s. . .), así como para decidir la rutina más apropiada de entre varias posibles para resolver un problema.

Con el objetivo de mejorar las prestaciones en arquitecturas multicore, las rutinas de la librería PLASMA se basan en algoritmos por *tiles*, diseñados con un paralelismo de grano fino. Las prestaciones de esta librería dependen, en gran medida, de los valores seleccionados para ciertos parámetros algorítmicos: el tamaño de bloque externo (*outer block size*), que marcará la granularidad del paralelismo y la flexibilidad de distribución del trabajo entre los cores, y el tamaño de bloque interno (*inner block size*), que marcará la carga extra de memoria debido al

incremento de operaciones en punto flotante [32]. Este último parámetro sólo se utiliza en las factorizaciones LU y QR.

Por tanto, los parámetros cuyos valores deben ser seleccionados para cada tamaño de problema a resolver en cada sistema, serán el número de threads, el tamaño de bloque externo y el tamaño de bloque interno. El modelo seguido por las rutinas de PLASMA incluye diferentes combinaciones de n (tamaño de la matriz), t (número de threads), b (tamaño de bloque externo) y l (tamaño de bloque interno). Así pues, se pueden considerar todas las combinaciones $\{n^3, n^2, n, 1\} \times \{t, 1, \frac{1}{t}\} \times \{b^2, b, 1, \frac{1}{b}\} \times \{l^2, l, 1, \frac{1}{l}\}$. Los términos de menor orden no se han incluido en el modelo, y para n^3 solamente se ha considerado el término $\frac{n^3}{t}$. De esta forma, el esquema del modelo para una rutina de PLASMA sería:

$$\begin{aligned} T(n,t,b,l) = & k_1\frac{n^3}{t} + k_2\frac{n^3}{bl} + k_3\frac{n^3}{l} + k_4\frac{n^3}{b} + k_5 n^2 tb + k_6 n^2 t + k_7\frac{n^2 t}{bl} + k_8\frac{n^2 t}{l} + k_9\frac{n^2 t}{b} + \\ & k_{10} n^2 tl + k_{11} n^2 b + k_{12} n^2 l + k_{13}\frac{n^2}{bl} + k_{14}\frac{n^2}{l} + k_{15}\frac{n^2}{b} + k_{16} n^2 + k_{17}\frac{n^2}{t} + \\ & k_{18}\frac{n^2 b}{t} + k_{19}\frac{n^2 l}{t} + k_{20} ntb^2 + k_{21} ntbl + k_{22} ntb + k_{23} ntl^2 + k_{24}\frac{nt}{l} + k_{25}\frac{nt}{b} + \\ & k_{26} ntl + k_{27} nt + k_{28} nb^2 + k_{29} nbl + k_{30} nb + k_{31} nl^2 + k_{32} nl + k_{33} n \end{aligned} \quad (3.2)$$

A partir de este modelo esquemático, y llevando a cabo un conjunto de experimentos para diferentes tamaños de problema y número de threads, se puede realizar un ajuste por mínimos cuadrados (LS) o mínimos cuadrado no negativos (NNLS) para estimar los valores de los coeficientes k_i, obteniendo finalmente el modelo completo. Esto se realiza durante el proceso de instalación de cada rutina en el sistema. Para cada tamaño de problema especificado en un *Conjunto de Instalación*, CI, la rutina se ejecuta variando el número de threads y el tamaño de bloque externo e interno, llevando a cabo, a su vez, la estimación experimental de los coeficientes con LS o NNLS (`Mod-LS` o `Mod-NNLS`). De esta forma, gracias a la información que aporta este modelo generado durante la instalación, será posible seleccionar, en tiempo de ejecución, el número de threads y el tamaño de los *tiles* con los que se obtiene el menor tiempo para cada tamaño de problema.

Las tablas 3.3 (**hipatia**), 3.4 (**saturno**) y 3.5 (**joule**) muestran la configuración óptima de los AP (número de threads, tamaño de bloque externo) y la selección que realizan las técnicas de modelado experimental `Mod-LS` y `Mod-NNLS` para la rutina de descomposición de Cholesky con diferentes tamaños de problema de un *Conjunto de Validación*. Las columnas ***Dev*** muestran la desviación de cada método respecto al tiempo de ejecución óptimo. La columna ***Default*** muestra el tiempo de ejecución obtenido con esta rutina usando la configuración por defecto establecida por PLASMA para el tamaño de bloque externo, con tantos threads como cores tenga la plataforma.

n	*Optimo*	*Default*	`Mod-LS`	*Dev.*	`Mod-NNLS`	*Dev.*
2000	0.13 (12-200)	0.14	0.14 (8-280)	10%	0.13 (16-280)	4%
3000	0.27 (16-280)	0.32	0.27 (12-280)	2%	0.27 (16-280)	0%
4000	0.48 (16-280)	0.63	0.48 (16-280)	0%	0.48 (16-280)	0%
5000	0.79 (16-280)	1.05	0.79 (16-280)	0%	0.79 (16-280)	0%
6000	1.21 (16-280)	1.66	1.21 (16-280)	0%	1.21 (16-280)	0%
7000	1.75 (16-280)	2.58	1.75 (16-280)	0%	1.75 (16-280)	0%
8000	2.44 (16-280)	3.69	2.44 (16-280)	0%	2.44 (16-280)	0%
9000	3.35 (16-280)	5.14	3.35 (16-280)	0%	3.35 (16-280)	0%
10000	4.38 (16-280)	6.94	4.38 (16-280)	0%	4.38 (16-280)	0%

Tabla 3.3: Tiempos de ejecución (en segundos) obtenidos en **hipatia** con la rutina de descomposición de Cholesky de PLASMA al utilizar tantos threads como cores y el tamaño de bloque por defecto, y utilizando el número de threads y el tamaño de bloque externo seleccionado con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`, así como la desviación experimentada en el tiempo de ejecución con cada una respecto al valor óptimo.

Las diferencias en el tiempo de ejecución entre el óptimo y las técnicas propuestas, son mayores para problemas pequeños, para los cuales los modelos del tiempo de ejecución no son tan precisos como para problemas grandes. La técnica `Mod-LS` ofrece una desviación media respecto al óptimo de, aproximadamente, 1% en **hipatia**, 4% en **saturno** y 5% en **joule**, mientras que las desviaciones medias de `Mod-NNLS` son de, aproximadamente, 0,5% en **hipatia**, 0,4% en **saturno** y 7% en **joule**. En cualquier caso, los desviaciones son claramente inferiores que en el caso ***Default*** (27% en **hipatia**, 1% en **saturno** y 13% en **joule**). Del mismo modo, se muestra un estudio comparativo con la rutina de descomposición QR (tablas 3.6 (**hipatia**), 3.7 (**saturno**) y 3.8 (**joule**)) y la rutina de factorización LU (tablas 3.9 (**hipatia**), 3.10 (**saturno**) y 3.11 (**joule**)) de la librería PLASMA.

n	*Optimo*	*Default*	Mod-LS	*Dev.*	Mod-NNLS	*Dev.*
2000	0.08 (24-80)	0.08	0.09 (12-80)	17%	0.08 (24-80)	0%
3000	0.18 (24-80)	0.18	0.20 (18-80)	13%	0.18 (24-80)	0%
4000	0.35 (24-120)	0.35	0.35 (24-120)	0%	0.35 (24-80)	1%
5000	0.59 (24-120)	0.59	0.59 (24-160)	1%	0.59 (24-120)	0%
6000	0.92 (24-120)	0.92	0.93 (24-160)	2%	0.92 (24-120)	0%
7000	1.33 (24-160)	1.35	1.35 (24-200)	1%	1.35 (24-120)	1%
8000	1.88 (24-160)	1.90	1.91 (24-240)	1%	1.88 (24-160)	0%
9000	2.56 (24-200)	2.60	2.58 (24-240)	1%	2.59 (24-160)	1%
10000	3.37 (24-200)	3.45	3.40 (24-280)	1%	3.38 (24-160)	1%

Tabla 3.4: Tiempos de ejecución (en segundos) obtenidos en **saturno** con la rutina de descomposición de Cholesky de PLASMA al utilizar tantos threads como cores y el tamaño de bloque por defecto, y utilizando el número de threads y el tamaño de bloque externo seleccionado con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`, así como la desviación experimentada en el tiempo de ejecución con cada una respecto al valor óptimo.

n	*Optimo*	*Default*	Mod-LS	*Dev.*	Mod-NNLS	*Dev.*
2000	0.12 (32-40)	0.19	0.16 (16-200)	27%	0.15 (64-40)	18%
3000	0.22 (64-60)	0.29	0.23 (48-40)	7%	0.22 (64-60)	0%
4000	0.38 (64-60)	0.45	0.38 (64-60)	0%	0.38 (64-60)	0%
5000	0.60 (64-140)	0.68	0.64 (64-60)	8%	0.64 (64-60)	8%
6000	0.87 (64-140)	0.99	0.89 (64-60)	2%	1.05 (64-80)	22%
7000	1.24 (64-140)	1.43	1.26 (64-60)	2%	1.36 (64-80)	10%
8000	1.94 (64-200)	2.01	1.94 (64-200)	0%	1.97 (64-100)	1%
9000	2.58 (64-200)	2.67	2.58 (64-200)	0%	2.63 (64-100)	2%
10000	3.37 (64-200)	3.51	3.37 (64-200)	0%	3.43 (64-100)	2%

Tabla 3.5: Tiempos de ejecución (en segundos) obtenidos en **joule** con la rutina de descomposición de Cholesky de PLASMA al utilizar tantos threads como cores y el tamaño de bloque por defecto, y utilizando el número de threads y el tamaño de bloque externo seleccionado con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`, así como la desviación experimentada en el tiempo de ejecución con cada una respecto al valor óptimo.

n	*Optimo*	`Mod-LS`	*Dev.*	`Mod-NNLS`	*Dev.*
1512	0.19 (16-224-88)	0.25 (12-304-188)	30 %	0.22 (16-304-68)	13 %
2024	0.32 (16-264-88)	0.38 (16-304-188)	19 %	0.35 (16-304-68)	9 %
2536	0.50 (16-264-88)	0.58 (16-304-188)	16 %	0.55 (16-304-68)	11 %
3048	0.72 (16-264-88)	0.84 (16-304-168)	17 %	0.80 (16-304-88)	12 %
3560	1.11 (16-264-88)	1.18 (16-304-168)	7 %	1.13 (16-304-88)	2 %
4072	1.53 (16-304-108)	1.50 (16-304-168)	5 %	1.54 (16-304-88)	1 %
4584	2.07 (16-264-88)	2.43 (16-304-28)	17 %	2.12 (16-304-108)	3 %
5096	2.68 (16-304-88)	2.99 (16-304-28)	11 %	2.69 (16-304-108)	1 %
5608	3.45 (16-304-108)	3.95 (16-304-28)	15 %	3.45 (16-304-108)	0 %

Tabla 3.6: Tiempos de ejecución (en segundos) obtenidos en **hipatia** con la rutina QR de PLASMA al usar el número de threads y el tamaño de bloque externo e interno seleccionado con las técnicas de modelado empírico (`Mod-LS` y `Mod-NNLS`), y desviación producida en el tiempo de ejecución con cada una respecto al óptimo.

n	*Optimo*	`Mod-LS`	*Dev.*	`Mod-NNLS`	*Dev.*
1512	0.10 (24-104-28)	0.10 (24-104-28)	0 %	0.11 (24-104-68)	8 %
2024	0.18 (24-104-28)	0.18 (24-104-28)	0 %	0.19 (24-104-68)	6 %
2536	0.31 (24-144-48)	0.32 (24-144-28)	4 %	0.31 (24-144-48)	0 %
3048	0.49 (24-144-48)	0.49 (24-144-48)	0 %	0.49 (24-144-48)	0 %
3560	0.70 (24-144-48)	0.70 (24-144-48)	0 %	0.70 (24-144-48)	0 %
4072	1.01 (24-144-48)	1.02 (24-184-48)	1 %	1.02 (24-184-48)	1 %
4584	1.33 (24-184-48)	1.33 (24-184-48)	0 %	1.33 (24-184-48)	0 %
5096	1.80 (24-184-48)	1.81 (24-224-48)	1 %	1.81 (24-224-48)	1 %
5608	2.32 (24-144-48)	2.39 (24-224-48)	3 %	2.39 (24-224-48)	3 %

Tabla 3.7: Tiempos de ejecución (en segundos) obtenidos en **saturno** con la rutina QR de PLASMA al usar el número de threads y el tamaño de bloque externo e interno seleccionado con las técnicas de modelado empírico (`Mod-LS` y `Mod-NNLS`), y desviación producida en el tiempo de ejecución con cada una respecto al óptimo.

3.6 Optimización automática experimental

En [35] se plantea una instalación meramente experimental como alternativa al uso de modelos teóricos del tiempo de ejecución. Esta propuesta está orientada a aquellos casos en los que el código de la rutina no está disponible o es muy complicado obtener un modelo con suficiente precisión debido a la complejidad del sistema o del propio software.

n	*Optimo*	Mod-LS	*Dev.*	Mod-NNLS	*Dev.*
1512	0.12 (64-64-28)	0.14 (32-104-28)	14%	0.13 (64-64-48)	3%
2024	0.22 (48-64-28)	0.25 (64-104-48)	17%	0.24 (64-64-48)	10%
2536	0.33 (64-64-28)	0.34 (64-104-28)	2%	0.34 (64-104-48)	8%
3048	0.49 (64-64-28)	0.51 (64-104-28)	5%	0.52 (64-104-48)	6%
3560	0.70 (64-64-28)	0.70 (64-104-28)	1%	0.71 (64-104-48)	1%
4072	0.92 (64-144-48)	0.97 (64-104-28)	4%	0.98 (64-104-48)	6%
4584	1.22 (64-144-48)	1.31 (64-104-28)	7%	1.24 (64-144-28)	2%
5096	1.58 (64-144-48)	1.69 (64-104-28)	8%	1.61 (64-144-28)	2%
5608	2.01 (64-144-48)	2.23 (64-104-28)	11%	2.04 (64-144-28)	2%

Tabla 3.8: Tiempos de ejecución (en segundos) obtenidos en **joule** con la rutina QR de PLASMA al usar el número de threads y el tamaño de bloque externo e interno seleccionado con las técnicas de modelado empírico (`Mod-LS` y `Mod-NNLS`), y desviación producida en el tiempo de ejecución con cada una respecto al óptimo.

Una primera técnica que se podría utilizar, consiste en realizar una búsqueda experimental exhaustiva de la combinación de valores de los AP (en un rango o conjunto discreto predeterminado de antemano) que ofrezca las mejores prestaciones. Obviamente, la aplicación de esta técnica conlleva un tiempo de instalación elevado. De cara a reducir este tiempo de instalación, se realiza un refinamiento de esta técnica, transformando la búsqueda en un procedimiento guiado con un porcentaje de empeoramiento permitido en el avance local a partir de cada solución parcial. De esta forma, la precisión en la obtención de los mejores valores de los AP va a depender, en gran medida, del valor de dicho porcentaje, pues conforme sea mayor el valor de este umbral, más fácil será evitar óptimos locales.

n	PLASMA	MKL	Mod-LS	Mod-NNLS
2000	0.190	**0.147**	0.176 (16-120-20)	0.239 (16-280-40)
3000	**0.377**	0.378	0.433 (16-120-20)	0.447 (16-280-60)
4000	**0.669**	0.695	0.726 (16-280-100)	0.729 (16-280-60)
5000	1.127	1.196	1.124 (16-280-120)	**1.121** (16-280-80)
6000	1.820	1.853	**1.745** (16-280-240)	1.777 (16-280-80)
7000	2.779	2.659	2.621 (16-280-20)	**2.619** (16-280-80)
8000	4.088	3.863	**3.797** (16-280-20)	3.821 (16-280-40)

Tabla 3.9: Comparación del tiempo de ejecución (en segundos) obtenido en **hipatia** con la rutina LU de PLASMA, con la rutina LU de MKL y con la rutina LU de PLASMA usando el número de threads y los tamaños de bloque externo e interno seleccionados con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`.

n	PLASMA	MKL	`Mod-LS`	`Mod-NNLS`
3000	0.324	0.279	**0.214** (24-120-20)	0.464 (24-280-60)
4000	0.504	0.653	**0.383** (24-160-60)	0.647 (24-280-60)
5000	0.865	1.121	**0.692** (24-160-60)	0.859 (24-280-60)
6000	1.156	1.853	**1.144** (24-160-60)	1.184 (24-280-80)
7000	2.131	2.359	1.777 (24-160-120)	**1.770** (24-280-80)
8000	2.508	3.452	2.508 (24-200-20)	**2.498** (24-280-80)
9000	3.521	4.727	3.521 (24-200-20)	**3.464** (24-280-100)

Tabla 3.10: Comparación del tiempo de ejecución (en segundos) obtenido en **saturno** con la rutina LU de PLASMA, con la rutina LU de MKL y con la rutina LU de PLASMA usando el número de threads y los tamaños de bloque externo e interno seleccionados con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`.

n	PLASMA	MKL	`Mod-LS`	`Mod-NNLS`
3000	**0.596**	0.536	0.750 (64-120-20)	0.828 (64-280-80)
4000	0.848	0.938	**0.822** (64-160-160)	1.164 (64-280-60)
5000	1.168	1.775	**1.137** (64-200-200)	1.672 (64-280-60)
6000	1.493	2.736	**1.458** (64-200-200)	2.122 (64-280-60)
7000	2.029	3.826	**1.939** (64-200-200)	2.817 (64-280-80)
8000	**2.730**	5.066	**2.730** (64-200-20)	2.758 (64-280-80)
9000	**3.738**	8.259	**3.738** (64-200-20)	3.983 (64-280-80)

Tabla 3.11: Comparación del tiempo de ejecución (en segundos) obtenido en **joule** con la rutina LU de PLASMA, con la rutina LU de MKL y con la rutina LU de PLASMA usando el número de threads y los tamaños de bloque externo e interno seleccionados con las técnicas de modelado empírico `Mod-LS` y `Mod-NNLS`.

En la tabla 3.12 se muestran los tiempos de ejecución obtenidos tras aplicar la técnica de búsqueda guiada con diferentes valores para el umbral de permisibilidad. Los tiempos obtenidos son muy cercanos a los óptimos, especialmente para porcentajes del 10 % y del 20 %, y la reducción del tiempo de instalación que se consigue con respecto a utilizar una búsqueda exhaustiva está en torno al 30 %.

En rutinas que llevan a cabo una descomposición matricial (como Cholesky, LU o QR), se podría aplicar la metodología de auto-optimización a la rutina básica de multiplicación de matrices a la que invocan internamente para realizar las multiplicaciones matriciales por bloques. La tabla 3.13 muestra, para diferentes tamaño de problema, una comparativa del tiempo de ejecución obtenido directamente con la rutina de descomposición de Cholesky de LAPACK y mediante la

n	*Exhaustivo*	1 %	10 %	20 %	50 %
			saturno		
500	0.004 (6-4)	0.009 (2-2)	0.009 (2-2)	0.009 (2-2)	0.004 (2-12)
700	0.010 (6-4)	0.023 (2-2)	0.023 (2-2)	0.023 (2-2)	0.011 (2-12)
1000	0.029 (8-3)	0.033 (3-3)	0.033 (3-3)	0.033 (3-3)	0.030 (2-12)
2000	0.215 (6-4)	0.260 (3-3)	0.260 (3-3)	0.260 (3-3)	0.258 (2-10)
3000	0.521 (1-17)	0.834 (3-3)	0.834 (3-3)	0.834 (3-3)	0.709 (3-8)
			ben		
500	0.005 (23-1)	0.013 (1-4)	0.005 (4-6)	0.005 (4-6)	0.005 (1-20)
700	0.010 (7-4)	0.015 (1-10)	0.012 (5-6)	0.012 (5-6)	0.011 (1-19)
1000	0.018 (10-4)	0.030 (1-16)	0.018 (6-7)	0.018 (6-7)	0.025 (1-19)
2000	0.080 (10-5)	0.098 (3-15)	0.083 (6-8)	0.083 (6-8)	0.096 (3-16)
3000	0.219 (25-3)	0.230 (5-14)	0.238 (6-10)	0.238 (6-10)	0.230 (5-14)
			pirineus		
500	0.006 (1-24)	0.008 (4-4)	0.008 (4-4)	0.008 (4-4)	0.008 (4-4)
750	0.013 (2-16)	0.016 (4-4)	0.025 (4-6)	0.025 (4-6)	0.025 (4-6)
1000	0.026 (2-16)	0.033 (4-3)	0.026 (4-8)	0.026 (4-8)	0.026 (4-8)
2000	0.080 (5-12)	0.232 (4-8)	0.081 (4-15)	0.081 (4-15)	0.081 (4-15)
3000	0.275 (4-15)	0.275 (4-15)	0.275 (4-15)	0.275 (4-15)	0.275 (4-15)

Tabla 3.12: Tiempos de ejecución (en segundos) obtenidos en diferentes sistemas computacionales por la rutina de multiplicación de matrices con dos niveles de paralelismo utilizando una búsqueda exhaustiva y una búsqueda guiada (con diferentes porcentajes) para seleccionar el número de threads OpenMP y MKL a establecer en cada nivel. Entre paréntesis, combinación de threads OpenMP-MKL con la que se obtiene el tiempo de ejecución indicado.

propuesta de auto-optimización. Se observa que, en general, las decisiones que se toman para el tamaño de bloque y el número de threads de cada nivel (OpenMP y MKL), permiten obtener mejores prestaciones.

3.7 Conclusiones

En este capítulo se ha mostrado un recorrido por las principales aportaciones realizadas en el campo de la optimización automática de software. Se ha partido de las primeras ideas que se plantearon en este campo, hasta llegar a las propuestas más actuales.

	LAPACK+MKL		LAPACK+AT	
n	*bloque*	*tiempo*	*bloque*	*tiempo*
512	32	0.0039 (9)	128	0.0034 (1-14)
1024	96	0.0129 (12)	128	0.0115 (3-8)
1536	192	0.0246 (24)	64	0.0265 (6-4)
2048	384	0.0755 (24)	128	0.0621 (4-6)
2560	384	0.1091 (24)	64	0.0878 (6-4)
3072	512	0.2030 (21)	64	0.1457 (6-4)
3584	512	0.2792 (21)	256	0.2524 (4-6)
4096	512	0.3907 (21)	256	0.3645 (4-6)

Tabla 3.13: Tiempos de ejecución (en segundos) obtenidos en **saturno** con la rutina de Cholesky de LAPACK al usar su propia selección del tamaño de bloque (LAPACK+MKL) y realizando la selección del tamaño de bloque y del número de threads con la propuesta de auto-optimización (LAPACK+AT). Entre paréntesis, número de threads OpenMP y combinación OpenMP-MKL con los que se obtiene el tiempo de ejecución indicado en cada caso.

Tras este recorrido, se han presentado las tres propuestas de auto-optimización que han servido como punto de partida para el desarrollo de esta tesis, las cuales se pueden enmarcar en otras líneas generales dentro de este campo de investigación: estudio del tiempo de ejecución de las rutinas mediante su modelado, estudio experimental del comportamiento de las rutinas e hibridación de ambos enfoques.

Capítulo 4

Metodología de optimización jerárquica

En este capítulo se presenta la metodología de optimización jerárquica diseñada para la ejecución eficiente de código sobre plataformas computacionales heterogéneas. Esta propuesta surge tras detectar la necesidad de disponer de nuevas técnicas que permitan optimizar de forma automática la ejecución de kernels computacionales sobre plataformas compuestas por elementos de cómputo con arquitecturas paralelas de diferente naturaleza.

A lo largo de este capítulo, en primer lugar, se introducen una serie de nociones básicas sobre optimización jerárquica, abordándolas desde un punto de vista general. A continuación, se describe con detalle la metodología de optimización propuesta, ilustrando su funcionamiento en diferentes niveles de la jerarquía. Finalmente, se muestra cómo se podría aplicar la metodología tanto desde un punto de vista hardware (elementos de cómputo) como software (rutinas de álgebra lineal), indicando los posibles parámetros ajustables a considerar en cada uno de los enfoques y cómo se obtendría el valor de los mismos haciendo un uso jerárquico de la información de optimización generada en los diferentes niveles establecidos.

La metodología propuesta se ilustrará utilizando la rutina de multiplicación de matrices como caso de estudio, dado que constituye el kernel computacional básico de la mayoría de rutinas de álgebra lineal numérica contenidas en las

librerías existentes. Los ejemplos propuestos se aplicarán considerando ambos enfoques jerárquicos: hardware y software. En el enfoque hardware, se consideran unidades de cómputo individuales (CPU multicore, GPU...) y la agrupación de las mismas, y en el enfoque software, el uso de la rutina de multiplicación de matrices en rutinas de álgebra lineal de nivel superior.

4.1 Nociones generales de optimización jerárquica

Los sistemas computacionales actuales suelen estar formados por unidades de procesamiento paralelo con diferente arquitectura hardware (CPU, GPU o MIC). El uso eficiente de este tipo de plataformas requiere utilizar de forma adecuada los recursos computacionales de que disponen, de ahí que sea necesario rediseñar las técnicas de optimización existentes para que sean capaces de adaptarse de forma automática a las características del sistema empleado.

La estructura hardware de este tipo de plataformas hace posible que se pueda adoptar un enfoque jerárquico en el diseño de las técnicas de optimización, ya que los elementos de cómputo pueden agruparse de diferente forma y ser asociados a un nivel concreto de la jerarquía. De este modo, los elementos computacionales básicos se podrían agrupar en nodos y éstos, a su vez, podrían ser agrupados en clusters.

Esta agrupación jerárquica conlleva la necesidad de considerar diferentes parámetros en función del nivel de la jerarquía y de las unidades de cómputo utilizadas. Asimismo, dicha agrupación repercute en la forma de llevar a cabo la paralelización de las rutinas que se ejecutan en este tipo de sistemas para la resolución de problemas con elevados requisitos computacionales. Esto da lugar al uso de varios niveles y tipos de paralelismo para optimizar la ejecución de dichas rutinas. Sin embargo, la paralelización por sí sola no garantiza la obtención de tiempos de ejecución cercanos al óptimo, de ahí la importancia de aplicar un proceso de optimización jerárquico que permita realizar una selección automática de los parámetros ajustables de la rutina en cada nivel, de forma que, partiendo de los niveles más bajos de la jerarquía, sea capaz de minimizar el tiempo de ejecución

de la rutina en cada nivel de paralelismo, decidiendo para los niveles superiores de la jerarquía, el grado de reutilización de la información de optimización proporcionada por los niveles inferiores.

Los parámetros a determinar en cada nivel de la jerarquía dependerán tanto de la arquitectura hardware de las unidades de cómputo, como de las librerías que implementan las rutinas. Por tanto, la optimización se ha de aplicar combinando ambos enfoques, hardware y software, en cada nivel, estudiando el efecto que tienen diversos parámetros como el número de threads de ejecución o el tamaño de bloque a usar, la distribucion del trabajo entre las distintas unidades computacionales del nodo o el volumen de trabajo a asignar a cada nodo a nivel de cluster, de forma que sea posible determinar, en cada plataforma computacional, la versión adecuada de la rutina optimizada a utilizar.

Este enfoque se puede extender a sistemas computacionales más complejos, con una mayor jerarquía en la organización de sus elementos computacionales y una mayor heterogeneidad (CPU+GPU, CPU+multiGPU, CPU+MIC, CPU+multiMIC y, en general, nodos con CPU+multicoprocesadores), así como a rutinas de mayor nivel que hagan uso de rutinas básicas optimizadas en niveles inferiores de la jerarquía. De esta forma, las mismas ideas de la optimización jerárquica dentro de una jerarquía de elementos computacionales se puede aplicar en una jerarquía de rutinas.

4.2 Estructura de optimización multinivel

La metodología de optimización propuesta se ha diseñado siguiendo una estructura jerárquica multinivel. Con este diseño, se pretende que las rutinas se ejecuten en la plataforma computacional decidiendo de forma automática en cada nivel de la jerarquía, la mejor configuración a utilizar desde un punto de vista hardware (unidades de cómputo) y software (implementación de la rutina). Para ello, es necesario tomar una serie de decisiones en cada nivel, que vendrán determinadas por los parámetros algorítmicos a considerar en cada uno y que, a su vez, establecerán cómo se ha de ejecutar la rutina.

En sistemas computacionales con un elevado número de unidades de cómputo, tanto la cantidad de parámetros como el conjunto de posibles valores de éstos puede ser muy extenso. Esto hace que la exploración exhaustiva de todas las posibles combinaciones de valores de los parámetros se convierta en una tarea inviable. El enfoque multinivel adoptado permite abordar este problema organizando las unidades computacionales y/o las rutinas de forma jerárquica, de forma que las decisiones a tomar se repartan entre los diferentes niveles de la jerarquía, utilizando para la toma de decisiones en un nivel, la información almacenada en el nivel previo de la jerarquía. De esta manera se consigue descentralizar y segmentar el proceso de instalación, lo que repercutirá en una mayor escalabilidad de la metodología de optimización.

Así pues, para realizar este proceso siguiendo el enfoque multinivel propuesto, es necesario, en primer lugar, llevar a cabo la instalación jerárquica de la rutina en la plataforma computacional. Para ello, se ha adoptado un enfoque *bottom-up*, tomando como referencia los diferentes niveles de organización de las unidades de cómputo de las plataformas heterogéneas actuales. De esta forma, el proceso comenzará en el nivel más bajo de la jerarquía (unidades básicas de procesamiento) y ascenderá progresivamente hacia niveles superiores, haciendo uso, en cada uno de ellos, de la información de optimización almacenada en el nivel inferior. Todo ello en base a considerar la plataforma completa como una unidad de cómputo global compuesta, recursivamente, por un conjunto de unidades de cómputo más sencillas conectadas entre sí. La figura 4.1 muestra el esquema general de instalación propuesto, ilustrando las fases de que consta este proceso y la forma en que se lleva a cabo.

El proceso de instalación de la rutina comienza en el nivel de la plataforma en el que se encuentran las unidades básicas de procesamiento (CPU, GPU, MIC), el nivel inicial (`H0`). En este nivel, la rutina se ejecuta en cada unidad de procesamiento con cada tamaño de un conjunto predefinido de tamaños de problema, llamado Conjunto de Instalación (CI), variando el valor de los parámetros considerados en este nivel para optimizar la ejecución de la rutina, entre los que se encuentran el número de threads a usar en CPU y/o MIC, o el tamaño de bloque en GPU. Cada unidad de procesamiento se trata de forma aislada e independiente, por tanto, en la medición de las prestaciones no se considera el tiempo empleado

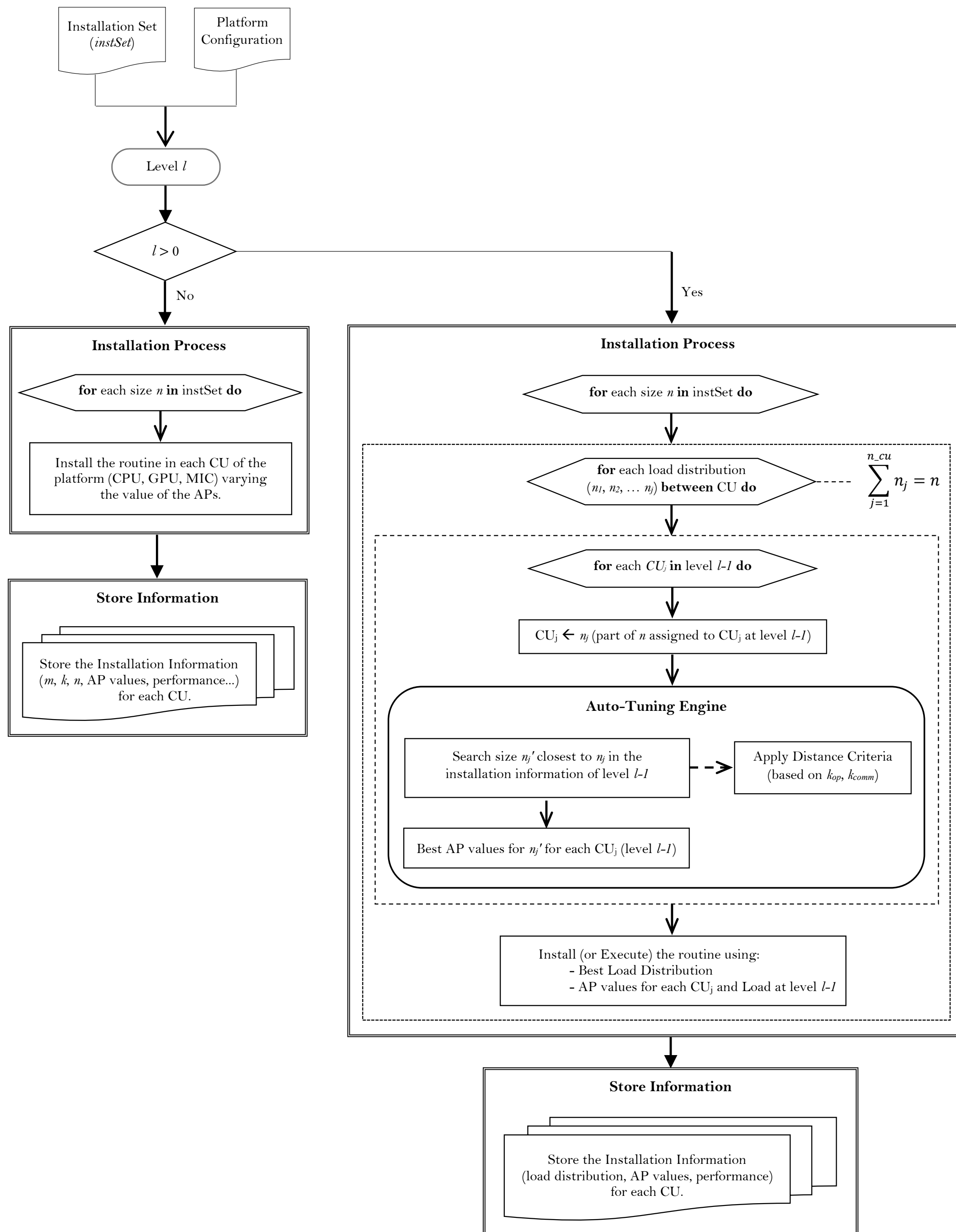

Figura 4.1: Esquema general de instalación jerárquica.

en las transferencias de datos entre la CPU y cada coprocesador (GPU o MIC). No obstante, dicho valor se almacena para tenerlo en cuenta en niveles superiores de la jerarquía. Al finalizar el proceso de instalación, se almacena, para cada tamaño de problema del CI, el tiempo de ejecución obtenido (y el rendimiento, en GFlops) con los valores óptimos de los parámetros y el tiempo de transferencia anteriormente mencionado.

Una vez instalada la rutina en el nivel inicial de la jerarquía, el proceso continúa hacia el siguiente nivel (`H1`), en el que la rutina se instala sobre los nodos de cómputo de la plataforma que contienen las unidades básicas de procesamiento sobre las que ya se ha realizado la instalación. Para optimizar la ejecución de la rutina en este nivel, se hace uso de la información de instalación almacenada en el nivel inferior para cada uno de los elementos de cómputo, teniendo en cuenta ahora las comunicaciones intra-nodo existentes, es decir, el coste de las transferencias entre la CPU y cada uno de los coprocesadores. En el nivel actual, el principal parámetro algorítmico a determinar es la cantidad de trabajo a asignar a cada unidad básica de procesamiento del nodo (CPU, GPU, MIC) para el tamaño de problema dado, pero también se consideran los parámetros propios de cada unidad básica para el tamaño de problema que se le ha asignado, cuyos valores serán determinados mediante la aplicación de un proceso de auto-optimización. La figura 4.2 muestra de forma gráfica los pasos que se llevarían a cabo durante la aplicación de este proceso. Estos pasos también quedan reflejados en los algoritmos 2 y 3.

Algoritmo 2: Instalación jerárquica en niveles superiores de la jerarquía.

```
Input: Installation Set, IS
for each platform level, l > 0 do
    for each Computing Unit, CU, at level l do
        AP: set of Algorithmic Parameters for CU
        for each n in IS do
            (Best_AP, Best_Perf) =
             SEARCH_BEST(l,CU,n,AP,METHOD)
            Performance_Installation[CU] += (n,Best_AP,Best_Perf)
        end
    end
end
```

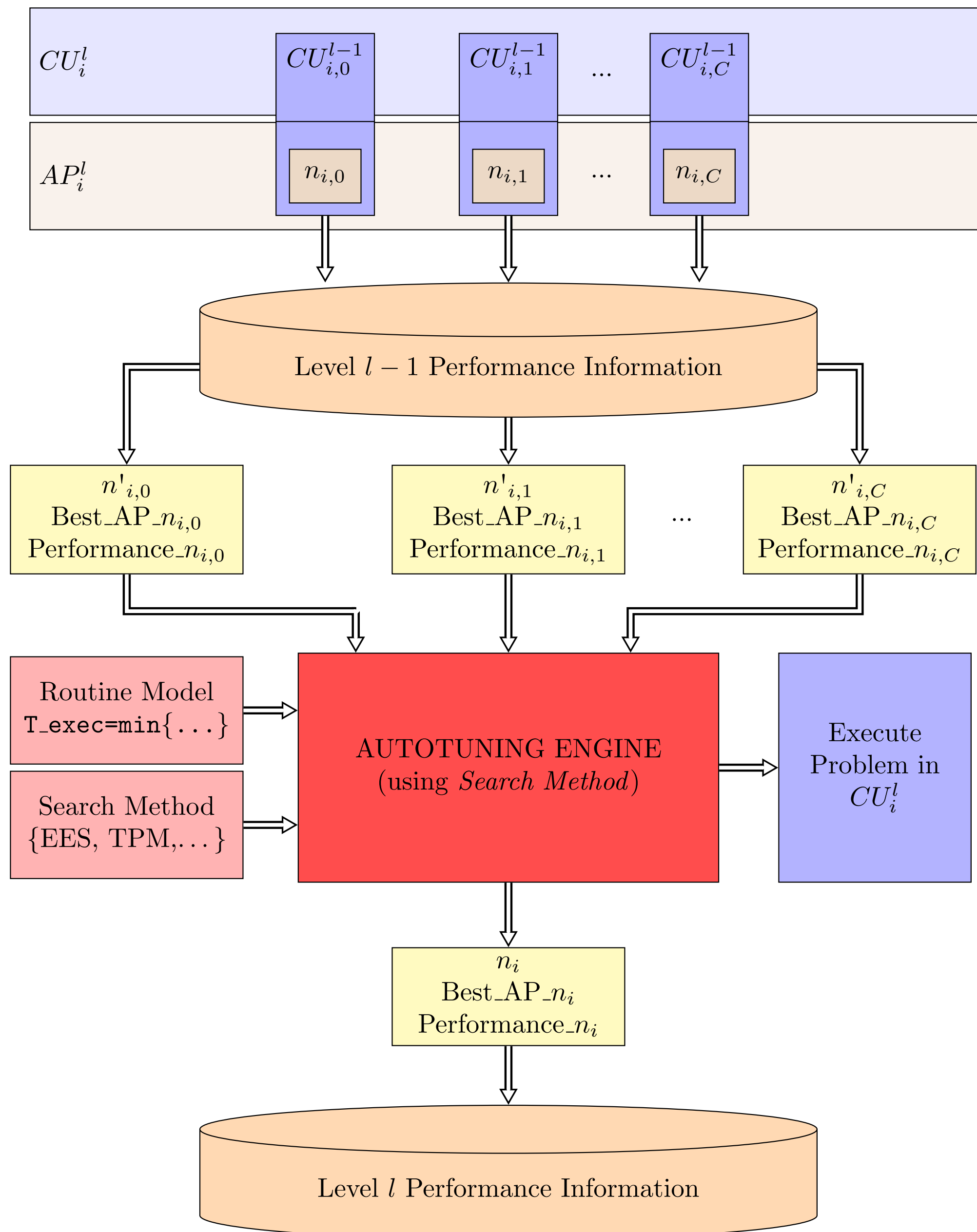

Figura 4.2: Metodología de auto-optimización jerárquica.

Algoritmo 3: Función para determinar los mejores valores de los parámetros algorítmicos y el rendimiento obtenido para un tamaño de problema n en el nivel $l > 0$ de la jerarquía.

SEARCH_BEST(l, CU, n, AP, $METHOD$)
Result: $Best_AP$, $Best_Performance$
for *each possible combination of* AP *values for* CU*:* AP_k **do**
    for *each computing unit inside* CU*:* CU_j **do**
        n_j: portion of n assigned to CU_j according to AP_k
        Get from $Performance_Information_j$:

- n'_j: problem size closest to n_j
- $Best_AP_n_j$: Best AP values of CU_j for n'_j
- $Performance_n_j$: Performance with $Best_AP_n_j$

    end

    if $METHOD$ = EES **then**
        $Performance_k$ = performance obtained by executing the problem (n, AP_k) in CU, with (n_j, $Best_AP_n_j$) in each CU_j inside CU
    end

    if $METHOD$ = TPM **then**
        $Performance_k$ = performance, according to model (equation 4.1), of the problem (n, AP_k) in CU, with (n_j, $Best_AP_n_j$) in each CU_j inside CU
    end

    if $METHOD$ = HES **then**
        $Performance_k$ = performance obtained by executing a local heuristic search starting with (n_j, $Best_AP_n_j$) for each CU_j inside CU
    end

    if $Performance_k > Best_Performance$ **then**
        $Best_Performance = Performance_k$
        $Best_AP = AP_k$
    end
end

Dado un tamaño de problema, n, a resolver en el nivel actual, $l > 0$, en una unidad de cómputo, UC, para cada posible combinación de valores de los parámetros ajustables de UC, AP_k, quedaría establecido el de reparto entre las unidades de cómputo UC_j que conforman UC. Pues bien, sea n_j, $j = 0, 1, \ldots, c$, la cantidad de n asignada a cada UC_j contenida en UC. Entonces, se determina el tamaño de problema del conjunto de instalación utilizado en el nivel inferior de la jerarquía, $l-1$, más cercano a n_j (n'_j) y se obtiene el mejor valor para los AP de UC_j basándose en la información de rendimiento previamente almacenada en el nivel $l-1$ de la jerarquía para un subproblema de tamaño n'_j en UC_j. Tras ello, el motor de auto-optimización de la metodología lleva a cabo la obtención de los mejores valores de los AP para UC mediante la aplicación del método *SEARCH_BEST* del algoritmo 3, que hace uso internamente de diferentes métodos de búsqueda (como los indicados tanto en el citado algoritmo como en la figura 4.2), cuya funcionalidad se describe a continuación:

- **EES**: búsqueda experimental exhaustiva. Realiza ejecuciones con cada posible reparto de n entre las UC_j de UC, considerando en cada ejecución los valores de los $Best_AP_n_j$ para cada una de las UC_j.

- **TPM**: modelado teórico-experimental. Considera que el tiempo de ejecución para resolver un problema de tamaño n en una UC, es el máximo que se obtiene de entre los empleados por cada UC_j con el mejor valor de los AP_j, para la porción n_j asignada a cada uno en base a la información de rendimiento almacenada en el nivel inferior, $l-1$, de la jerarquía. El tiempo de ejecución viene dado por la ecuación:

$$T(UC,n) = \min_{AP}\left\{K_{op}(UC,n,AP)\right\} = \min_{AP}\left\{\max_{UC_j}\left\{K_{op}(UC_j,n_j,AP_j) + K_{comm}(UC_j,n_j,AP_j)\right\}\right\} \quad (4.1)$$

  que considera tanto el coste de cómputo, (K_{op}) como el coste de las comunicaciones necesarias (K_{comm}) para el envío/recepción de los datos para resolver el tamaño de problema n_j asignado a cada UC_j. Este enfoque no requiere experimentación en el nivel actual, $l > 0$, pues se basa completamente en la información de instalación almacenada para el nivel inferior de

la jerarquía, en el que se llevarán a cabo ejecuciones en caso de tratarse del nivel inicial.

- **HES**: búsqueda experimental con heurística. Realiza una búsqueda local experimental (con valores crecientes y decrecientes) partiendo de la información de instalación obtenida para los $\boldsymbol{AP}$ de cada $\boldsymbol{UC}_j$ de $\boldsymbol{UC}$ con el valor de $\boldsymbol{n}_j$ asignado a cada uno.

Finalmente, como resultado del proceso de instalación se almacena, para cada tamaño de instalación $\boldsymbol{n}$ en el nivel actual $l > 0$, la configuración de reparto entre las unidades de cómputo de $\boldsymbol{UC}$ que da lugar a la obtención del menor tiempo de ejecución y el rendimiento obtenido (en GFlops).

El proceso descrito se podría aplicar de forma similar tanto si se consideran subniveles del nivel actual (mediante la agrupación de unidades básicas de procesamiento) como en niveles superiores de la jerarquía, en los que la plataforma computacional completa es considerada una única unidad de cómputo global compuesta por un conjunto de nodos que son vistos como cajas negras de la que sólo se obtiene la información de instalación necesaria para optimizar la ejecución de la rutina. En este caso, el coste de las comunicaciones difiere, pues corresponde al empleado en comunicar cada par de nodos, por tanto, ha de ser analizado experimentalmente con el fin de añadirlo a la información de rendimiento obtenida.

4.2.1 Jerarquía bidimensional: hardware × software

La presencia de diferentes escenarios tanto hardware como software al instalar las rutinas, puede dar lugar a diferentes formas de aplicar la metodología de optimización multinivel. Así pues, habrá casos en los que sólo se quiera instalar la rutina en un nivel utilizando un subconjunto de unidades de cómputo de la plataforma, o casos en los que se quiera instalar una rutina que internamente hace uso de la información de instalación almacenada para otra rutina de nivel inferior que ha sido previamente instalada en el sistema. De esta forma, se pueden considerar diferentes formas de llevar a cabo la instalación jerárquica, en función de cómo se utilice la información de instalación y de cómo se obtenga el valor de los parámetros algorítmicos en cada nivel.

La figura 4.3 ilustra las opciones que se podrían considerar en los diferentes niveles hardware y software de la jerarquía en función del tipo de instalación a realizar. Cada círculo (etiquetado como `SRHP`) representa que la rutina `S` de nivel `R` se instala en la plataforma `H` de nivel `P`. Cada flecha, etiquetada con `SRHP-srhp`, indica que la rutina `S` de nivel `R` se instala en la plataforma `H` de nivel `P` utilizando la información de instalación de la rutina `s` de nivel `r` previamente instalada en la plataforma `h` de nivel `p`. El esquema propuesto sigue el paradigma de programación dinámica, de forma que los valores obtenidos en el nivel 1 hardware y software se obtienen de los valores óptimos almacenados en los niveles inferiores.

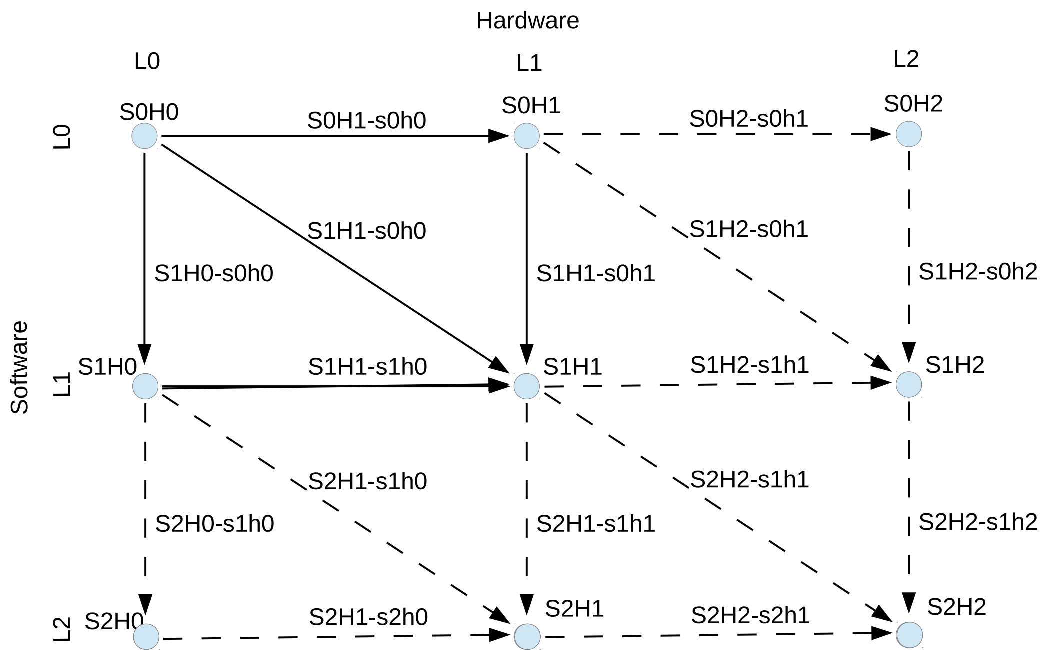

Figura 4.3: Opciones de instalación en una jerarquía multinivel bidimensional, con varios niveles hardware y software.

En el nivel inicial de la jerarquía, `S0H0` (nivel 0 de hardware y de software), la rutina se instala en cada unidad básica de procesamiento (CPU, GPU, MIC) con cada tamaño de problema del CI. En este nivel, la búsqueda del valor de los parámetros para cada unidad básica se realiza de forma exhaustiva y como resultado, se almacena, para cada tamaño de problema del CI, los valores de los

parámetros con los que se obtiene el menor tiempo de ejecución, junto con las prestaciones (en GFlops). Partiendo de este nivel, la instalación jerárquica puede avanzar hacia el siguiente nivel hardware o software de diferentes formas, donde cada una de ellas dará lugar a un tipo de instalación en la que los valores de los parámetros algorítmicos se determinarán directamente en el nivel de la instalación o accediendo a la información de instalación almacenada en el nivel inferior para los tamaños de problema con los que se instalaron las rutinas en cada una de las unidades básicas de procesamiento.

- Nivel `S0H1`: una rutina básica de nivel 0, `S0`, se instala en un nodo, `H1`. En este caso, tanto los valores de los parámetros en el nivel 1 hardware (cantidad de trabajo a asignar a cada UC del nodo), como los parámetros de cada UC para la carga de trabajo asignada, pueden ser obtenidos directamente de forma experimental en el nivel 1 (`S0H1`) variando la carga de trabajo asignada a cada unidad de cómputo básica o usando un enfoque jerárquico (`S0H1-s0h0`), delegando la selección de los parámetros a la instalación realizada en el nivel inferior para cada unidad de cómputo, utilizando la información de rendimiento almacenada (GFlops) para diferentes tamaños de problema. Para seleccionar la mejor partición teórica de la carga, se podría aplicar el enfoque jerárquico y, a continuación, realizar experimentos variando la cantidad de trabajo asignada a cada UC (método EES o HES), o bien usar la ecuación 4.1 sin experimentación (método TPM).
- Nivel `S1H0`: una rutina de nivel 1, `S1`, se instala en una UC de nivel 0, `H0`. En este caso, los parámetros algorítmicos a determinar dependen del tipo de rutina que se instale. La instalación podría llevarse a cabo directamente de forma experimental variando el valor de los parámetros tanto para la rutina de nivel 1 como para las rutinas básicas a las que invoca (`S1H0`), o utilizando un enfoque jerárquico (`S1H0-s0h0`) en el que, por un lado, se determina experimentalmente el valor de los parámetros de nivel 1 de la rutina y, por otro, se hace uso de la información almacenada en la instalación `S0H0` para obtener el valor de los parámetros de las rutinas básicas. No obstante, en ambos casos la selección de los parámetros propios de las UC básicas (como el número de threads de CPU o MIC) se obtendría directamente de la información de instalación generada en el nivel inferior (`S0H0`).

De esta forma, tanto en la instalación S0H1 como en la S1H0, la información de rendimiento del nivel inferior (S0H0) se reutiliza en niveles superiores del hardware (S0H1-s0h0) o del software (S1H0-s0h0), y podría también reutilizarse en otros nodos con las mismas unidades básicas de procesamiento o en otras rutinas que invoquen a las mismas rutinas básicas (como la multiplicación de matrices) con tamaños similares a los utilizados en la instalación S0H0.

- Nivel S1H1: una rutina de nivel 1, S1, se instala en el nodo, H1. La instalación podría llevarse a cabo directamente de forma experimental, variando el valor de los parámetros tanto a nivel hardware (cantidad de trabajo a asignar a cada UC del nodo y parámetros de cada UC para la carga de trabajo asignada) como software (rutina de nivel 1 y rutinas básicas a las que invoca) o utilizando un enfoque jerárquico. En este caso existen tres posibles opciones, en función de cómo se haga uso de la información de instalación almacenada en el nivel inferior de la jerarquía:

  - S1H1-s0h1: la rutina S1 utiliza la versión de la rutina de nivel inferior, s0, optimizada para el nivel 1 hardware, h1, por tanto, la explotación de la heterogeneidad del sistema se delega a s0. Los únicos parámetros a determinar son los propios de la rutina S1. Los parámetros a nivel hardware, H1, son obtenidos por la rutina optimizada a la que se invoca, haciendo uso de la información almacenada en la instalación S0H1.
  - S1H1-s0h0: cada rutina básica, s0, usada en S1, se asigna a una UC de nivel 0 hardware, h0, y se ejecuta en ésta con los valores de los parámetros almacenados en la instalación S0H0 para el tamaño de problema más cercano a la carga asignada a s0. En este caso, los parámetros a determinar son los propios de la rutina de nivel 1, S1, junto con la distribución de las rutinas básicas entre las UC de nivel 0, h0. Otra opción podría ser usar planificación dinámica [20], distribuyendo cada rutina básica, s0, entre las UC dinámicamente, comenzando con las unidades más potentes y usando la información de S0H0 para decidir la UC en la que ejecutar la rutina básica.
  - S1H1-s1h0: este caso es similar a S1H1-s0h0, con la única diferencia de que ahora, la rutina básica, s0, se reemplaza por la rutina de nivel

1, `s1`, cuyos valores de los parámetros fueron almacenados para cada UC básica, `h0`, tras la instalación `S1H0`.

Esta misma idea se podría aplicar en el siguiente nivel de la jerarquía (`S2H2`). En este caso, el número de opciones de instalación aumenta considerablemente, dado que se consideran todas las posibles alternativas desde el nivel anterior tanto en el plano hardware como software, lo que da lugar a diferentes formas de hacer uso de la información de instalación almacenada para obtener los parámetros algorítmicos de este nivel. Por tanto, la metodología de optimización multinivel descrita va a permitir, por un lado, optimizar la ejecución de la rutina en cada nivel de la jerarquía haciendo un uso racional y eficiente de la información almacenada en el nivel inferior, y por otro, reducir sucesivamente el tiempo de instalación conforme se avance hacia niveles superiores. Asimismo, permite aplicar en cada nivel diferentes métodos para llevar a cabo la toma de decisiones de los valores de los parámetros algorítmicos.

Las siguientes secciones del capítulo están destinadas a mostrar ejemplos de cómo se llevaría a cabo el proceso de instalación jerárquica en diferentes niveles hardware y software. Este proceso se describe de forma incremental, partiendo de un nivel hardware compuesto por una CPU multicore y un nivel software en el que se considera inicialmente la rutina básica de multiplicación de matrices. A continuación, se proponen diferentes casos donde se observa cómo se puede extender el proceso tanto a nivel hardware (mediante la agrupación de unidades básicas en nuevas unidades de cómputo) como software (mediante el uso de otras rutinas de nivel superior que utilizan internamente la multiplicación de matrices, como la multiplicacion de Strassen y la factorización LU). El objetivo no es optimizar en cada uno de los niveles hardware el algoritmo que realiza la multiplicación matricial, sino mostrar cómo se aplicaría la metodología de instalación jerárquica para que las rutinas que internamente utilizan esa operación, se ejecuten de forma eficiente en dicho nivel de la jerarquía con la versión optimizada de dicha rutina.

Del mismo modo, cabe destacar que los parámetros a determinar van a depender tanto del nivel hardware como software considerado. La tabla 4.1 resume los parámetros en los niveles 0 y 1 de hardware (CPU multicore, GPU y MIC como unidades de cómputo básicas de nivel 0, y la agrupación de éstas como

nodos de nivel 1) y software (multiplicación de matrices en el nivel 0 y la multiplicación de Strassen o la factorización LU en el nivel 1). En el caso de la rutina de multiplicación de matrices, los parámetros que se consideran son los indicados en la ecuación 2.2. En el caso de la multiplicación de Strassen, los parámetros propios de la rutina son el nivel de recursión y la implementación a usar (`s0h1`, `s0h0` o `s1h0`). En el caso de la multiplicación de Strassen basada en la ejecución simultánea de rutinas optimizadas en el nivel `H0` (`s0h0` or `s1h0`), un parámetro adicional sería el mapeo de dichas rutinas a unidades de cómputo de nivel 0. En la factorización LU, en cambio, el parámetro algorítmico a determinar será el tamaño de los bloques de computación. Asimismo, dado que existen múltiples librerías que trabajan sobre elementos computacionales distintos (MKL en multicore o MIC, PLASMA en multicore, MAGMA en GPU, CUBLAS en GPU), el proceso de optimización jerárquico habría que adaptarlo en función de la librería que se utilice, pues la implementación de una rutina en cada librería determina los parámetros (y el rango de posibles valores) a seleccionar.

S0H0 Número de hilos en CPU o MIC ($t_{i,j}$ en ecuación 2.2)	**S0H1** Distribución del Trabajo entre UC de `H0` ($n_{i,j}$ en ecuación 2.2)

S1H0
Strassen: nivel de recursión.
LU: tamaño de bloque.
	S1H1
	Instalación Jerárquica:
	`a) S1H1-s0h1`
	- Strassen: nivel de recursión.
	- LU: tamaño de bloque.
	`b) S1H1-s0h0`
	- Rutinas básicas a UC de `H0`
	`c) S1H1-s1h0`
	- Multiplicaciones de Strassen a UC de `H0`
	- Rutinas básicas de LU a UC de `H0`

Tabla 4.1: Parámetros algorítmicos para los niveles 0 y 1 de la jerarquía, usando la multiplicación de matrices como rutina básica de nivel 0 y la multiplicación de Strassen y la factorización LU como ejemplo de rutinas de nivel 1.

4.3 Multiplicación de matrices en multicore

Esta sección muestra cómo se llevaría a cabo la instalación de la rutina básica de multiplicación de matrices en una CPU multicore (`S0H0`), siguiendo la metodología de instalación jerárquica propuesta en la figura 4.3.

Este escenario corresponde al caso base de la jerarquía, en cuanto a hardware y software se refiere (`S0H0`), pues la multiplicación de matrices constituye uno de los kernels computacionales básicos utilizados por un gran número de rutinas de nivel superior, y la CPU multicore es una de las unidades básicas de procesamiento (junto con GPU y MIC) de que disponen los sistemas computacionales actuales.

Dado que la instalación se realiza sobre una unidad básica de procesamiento, los parámetros a tener en cuenta en este nivel serán los propios de dicha unidad que permitan optimizar la ejecución de la rutina. En este caso, se considera el número de threads, que está limitado por el número de cores lógicos del multicore, en función de si ofrece soporte para hacer uso de *hyperthreading*. Los threads se establecerán mediante la función proporcionada por la librería de álgebra lineal que se utilice para invocar a la operación de multiplicación.

El objetivo es instalar la rutina de forma que, cuando se quiera hacer uso de la versión optimizada para un tamaño de problema dado, permita su ejecución con tiempos cercanos al óptimo y con un uso eficiente de la plataforma. Para ello, la instalación se realiza subdividiendo la dimensión n de la matriz B en bloques de columnas, en función de un valor porcentual establecido que determina el grano de reparto a utilizar (*Distribution Grain Size*, DGS). De este modo, durante el proceso de instalación se irá obteniendo información de rendimiento para matrices rectangulares de distintos tamaños, que será útil tanto para rutinas de mayor nivel como en sistemas con un mayor número de unidades de cómputo. Por ejemplo, en la factorización LU (rutina de nivel 1) las multiplicaciones de matrices que se llevan a cabo son rectangulares, y lo mismo sucede cuando se realiza la operación de multiplicación combinando CPU y GPU en un nodo (sistema de nivel 1).

Cabe recordar que la operación de multiplicación se define como $C = \alpha AB + \beta C$, $A \in R^{m\times k}$, $B \in R^{k\times n}$, $C \in R^{m\times n}$. Así pues, dado un CI con los tamaños de problema definidos por las dimensiones de las matrices, (m, k, n), para cada par

(m, k), se llevarán a cabo $\frac{100}{DGS}$ ejecuciones de la rutina, con tamaños $n_i = \frac{i \cdot DGS}{100} \cdot n$, con $i = 1, \ldots, \frac{100}{DGS}$, y en cada una de ellas, el número de threads se variará hasta alcanzar el máximo número de cores lógicos del multicore. El algoritmo 4 indica los pasos que se llevarían a cabo para la instalación de la rutina en el nivel `SOHO`. La figura 4.4, por su parte, ilustra de forma gráfica este proceso. Durante el proceso de instalación, la información de rendimiento obtenida en la CPU para cada tamaño de problema utilizado en cada ejecución de la rutina, se almacena junto al número de threads utilizado. Esta información será utilizada para optimizar la ejecución de esta rutina (u otras de nivel superior) cuando se instalen en el siguiente nivel de la jerarquía hardware.

Algoritmo 4: Instalación de la rutina de multiplicación en CPU multicore.

Input: *Installation Set, IS*
for *each n in IS* **do**
  $load = \{\frac{i \cdot DGS}{100} \cdot n$, con $i = 1, \ldots, \frac{100}{DGS}\}$
  for *each load* **do**
    for *thr in* $1, \ldots, max_threads$ **do**
      $(time, gflops) = \text{EXEC_GEMM}(m,k,load,thr)$
      $Performance_Installation[CPU] \mathrel{+}= (m,k,load,thr,time,gflops)$
    end
  end
end

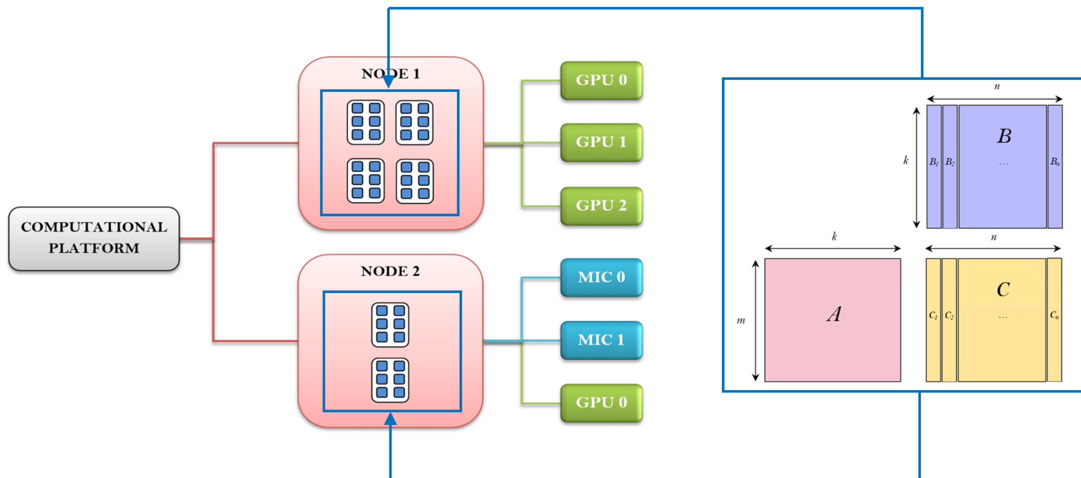

Figura 4.4: Instalación de la rutina de multiplicación de matrices variando la carga de trabajo en la CPU multicore de diferentes nodos.

El proceso descrito se realizaría de forma similar en el caso de considerar otras unidades básicas de procesamiento, como GPUs o coprocesadores Intel Xeon Phi. En este caso, habría que tener en cuenta los parámetros propios de estas unidades (como el tamaño de bloque en GPU o el número de threads en Xeon Phi) que permitan optimizar la ejecución de la rutina. Si se consideran varias unidades de procesamiento, la instalación se realizaría entonces en paralelo (poniendo en marcha tantos threads OpenMP como unidades de cómputo), pero la rutina se ejecutaría de manera independiente en cada una de ellas.

4.4 Extensión en la jerarquía hardware: multiplicación de matrices en nodo CPU+GPU

Esta sección aborda cómo se instalaría la rutina de multiplicación de matrices, aplicando la metodología de optimización multinivel cuando se avanza de forma jerárquica en la dimensión hardware (nivel `S0H1`). Para ello, se considera un nodo que contiene, además de una CPU multicore, una GPU. Asimismo, se asume que la rutina ha sido instalada en cada una de ellas en el nivel inferior de la jerarquía (`S0H0`), tal y como se ha descrito en la sección anterior.

En este nivel de la jerarquía, la operación de multiplicación de matrices $C = \alpha AB + \beta C$ se expresaría como $C = \alpha(AB_1|AB_2) + \beta(C_1|C_2)$, con $B = (B_1|B_2)$ y $C = (C_1|C_2)$, $B_i \in R^{k\times n_i}$, $C_i \in R^{m\times n_i}$ y $n = n_1 + n_2$. La multiplicación $\alpha AB_1 + \beta C_1$ se asignaría a la GPU y $\alpha AB_2 + \beta C_2$ a la CPU. Por tanto, el parámetro algorítmico a determinar será el reparto de la carga de trabajo entre ambas unidades de procesamiento (CPU y GPU). El rendimiento de la rutina será mayor cuanto más balanceado sea dicho reparto. Los parámetros correspondientes a la propia multiplicación no se consideran en este nivel, pues se hará uso de la información de rendimiento almacenada en la instalación de nivel inferior realizada en cada una de las unidades computacionales que se utilizan (`S0H1-s0h0`). Así pues, la instalación de la rutina se puede llevar a cabo de dos formas:

- **Exhaustiva**: la rutina se ejecuta para cada posible reparto de la carga de trabajo entre CPU y GPU, determinado por el valor de la variable *DGS*

establecido para este nivel de la jerarquía, y se hace uso de la información de instalación del nivel inferior para obtener el mejor valor de los parámetros algorítmicos de cada unidad de procesamiento.

- **Modelo Teórico**: el reparto de la carga entre CPU y GPU, así como el mejor valor de los parámetros de cada unidad con la carga asignada, se determina aplicando el método TPM del algoritmo 3 con la información de rendimiento almacenada en el nivel inferior de la jerarquía. Con el fin de disponer de las prestaciones obtenidas para su posterior uso desde niveles superiores de la jerarquía, la rutina se ejecuta con los valores de los parámetros así obtenidos.

La elección de un método u otro repercutirá en el tiempo de instalación necesario. En el caso exhaustivo el tiempo será superior, ya que la rutina se ejecuta con todas las posibles combinaciones de reparto entre CPU y GPU obtenidas a partir del valor de ***DGS***. Dado que las ejecuciones se llevan a cabo entre unidades de cómputo de diferente tipo (CPU y GPU), se tiene en cuenta el coste de las comunicaciones intra-nodo en el tiempo de ejecución obtenido. Al usar el modelo teórico, en cambio, el tiempo de instalación se reduce sustancialmente, pues se realiza una única ejecución de la rutina, pero en este caso, la dificultad radica en la obtención de un modelo preciso del tiempo de ejecución que contemple el coste de las transferencias CPU $\longleftrightarrow$ GPU.

El algoritmo 5 muestra cómo se llevaría a cabo la instalación de la rutina de multiplicación de matrices en el nivel `S0H1` de la jerarquía considerando un nodo compuesto por una CPU multicore y una GPU. La función ***SEARCH_BEST*** corresponde a la implementada en el algoritmo 3, que permite indicar, mediante el parámetro ***METHOD***, el método a usar (EES (exhaustivo) o TPM (modelo teórico)). Por otro lado, la figura 4.5 ilustra, de forma genérica, cómo se realizaría este proceso. Con el método exhaustivo, la matriz ***A*** se envía de forma completa a la GPU y la matriz ***B*** se va particionando y enviando de forma progresiva en bloques de columnas hasta considerarse de forma completa. Con el modelo teórico, en cambio, los envíos relativos a la matriz ***B*** no se producen, ya que el mejor reparto entre CPU y GPU se obtiene basándose en la información de instalación almacenada en el nivel inferior de la jerarquía. No obstante, tanto en un caso co-

mo en el otro, se accede a la información de instalación almacenada para la CPU con el fin de determinar el número de threads a usar para el bloque de columnas de B asignado. Este proceso se podría extender de la misma forma a nodos compuestos por diferente número y tipo de unidades de procesamiento (CPU+MIC, CPU+multiGPU, CPU+multiMIC, CPU+multiGPU+multiMIC. . .).

Una vez finalizado el proceso de instalación, se almacena el reparto de la carga entre CPU y GPU con el que se obtiene el menor tiempo de ejecución, así como las prestaciones alcanzadas (en GFlops). Esta información de optimización será incorporada para su uso en el siguiente nivel de la jerarquía, por ejemplo, al considerar la agrupación de varios nodos en un cluster.

Algoritmo 5: Instalación de la rutina de multiplicación en un nodo compuesto por CPU+GPU.

Input: *Installation Set, IS*
Installation Level, $l > 0$
CU: Node with CPU+GPU
DGS: Distribution Grain Size for Matrix B

for *each n in IS* **do**
    $AP = \{n_i = \frac{i \cdot DGS}{100} \cdot n, \text{ con } i = 0, \ldots, \frac{100}{DGS}\}$
    $(Best_AP, Best_Perf) = \text{SEARCH_BEST}(l, CU, n, AP, METHOD)$
    $Performance_Installation[CU] \mathrel{+}= (n, Best_AP, Best_Perf)$
end

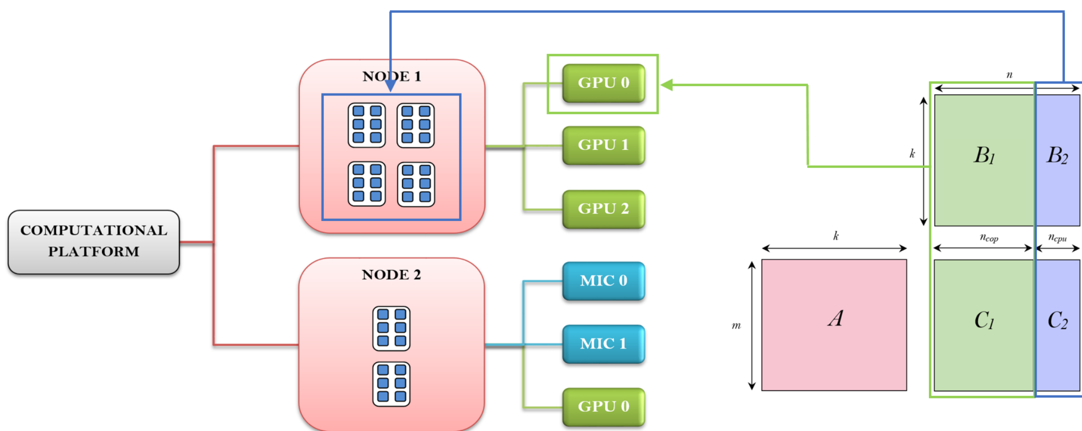

Figura 4.5: Esquema de reparto de la carga de trabajo en la instalación de la rutina de multiplicación de matrices en un nodo compuesto por CPU+GPU.

4.5 Extensión en la jerarquía software

Hasta ahora, los ejemplos propuestos se han centrado exclusivamente en mostrar cómo aplicar la metodología de instalación jerárquica con una rutina computacional básica en diferentes niveles hardware (niveles `S0H0` y `S0H1`). Esta sección, por su parte, completa dicho estudio extendiendo la jerarquía en la dimensión software (niveles `S1H0` y `S1H1`), considerando una organización jerárquica desde el punto de vista de las rutinas.

Para conseguir que los ejemplos que se muestran sean suficientemente ilustrativos y permitan reflejar de forma adecuada la aplicación de la metodología de optimización en ambas dimensiones (hardware y software), se han considerado rutinas que internamente invocan a la rutina básica de multiplicación de matrices (utilizada en los ejemplos de las secciones anteriores). Estas rutinas son la multiplicación de Strassen y la factorización LU. Se indicarán los principales parámetros algorítmicos a tener en cuenta en cada una de ellas, y se mostrará cómo se aplicaría la metodología de instalación en los niveles `S1H0` y `S1H1`, considerando diferentes formas de usar la información de instalación almacenada en niveles inferiores de la jerarquía para conseguir una ejecución optimizada de las mismas.

4.5.1 Multiplicación de Strassen

La multiplicación de Strassen se ha descrito con detalle en la sección 2.2 del Capítulo 2. Esta rutina se caracteriza por realizar una multiplicación de matrices mediante la descomposición de dicha operación en siete multiplicaciones básicas, que se combinan mediante operaciones aritméticas de multiplicación, suma y resta para obtener la matriz resultado. Esta descomposición se lleva a cabo utilizando un esquema algorítmico recursivo basado en divide y vencerás, de forma que se van realizando llamadas recursivas hasta que se alcanza el valor establecido para el caso base, tb, momento en el que se ejecuta la multiplicación básica con matrices de tamaño $tb \times tb$. Por tanto, las prestaciones obtenidas con cada valor del caso base utilizado para cada tamaño de problema, permitirán determinar el mejor nivel de recursión a utilizar, de ahí que éste sea el parámetro algorítmico a considerar en esta rutina.

Se parte de un escenario en el que ya se encuentra instalada la rutina básica de multiplicación de matrices en los niveles `S0H0` y `S0H1` para un determinado CI, tal y como se ha mostrado en la sección anterior. Por tanto, la extensión de la metodología en la jerarquía software se podría llevar a cabo aplicando la metodología de instalación jerárquica a la rutina de Strassen en los niveles `S1H0` y `S1H1`, utilizando en el caso base la versión optimizada de la multiplicación básica (a nivel de unidad de procesamiento o de nodo) y determinando, para cada tamaño de problema, el mejor valor a usar para el nivel de recursión.

En el nivel `S1H0`, la rutina se instalaría a nivel de unidad básica de procesamiento (CPU, GPU o MIC), utilizando en el caso base la versión optimizada de la multiplicación básica para dicha unidad con el mejor valor de los parámetros algorítmicos (número de threads en CPU o MIC, `S1H0-s0h0`) y determinando, a su vez, el nivel de recursión a usar para cada tamaño de problema del CI. El algoritmo 6 muestra cómo se llevaría a cabo la instalación considerando una CPU multicore. La función *STRASSEN_S1H0* ejecuta el algoritmo 1 con el valor del caso base establecido para la iteración actual del bucle, e internamente hará uso de la versión de la multiplicación básica optimizada para la CPU multicore. Como resultado, devuelve las prestaciones que se obtienen, que se usarán para determinar el mejor nivel de recursión para el tamaño de problema actual.

Algoritmo 6: Instalación de la rutina de multiplicación de Strassen en el nivel `S1H0` en una CPU multicore.

Input: *Installation Set*, *IS*
for *each n in IS* **do**
  $base = \left\{\frac{n}{i}, \text{ con } i = 1,2,4,\ldots\right\}$
  for *each base* **do**
    $Performance = \textbf{STRASSEN_S1H0}(n, base, CPU)$
    if $Performance > Best_Perf$ **then**
      $Best_Perf = Performance$
      $Best_AP = n/base$
    end
  end
  $Performance_Installation[CPU] \mathrel{+}= (n, Best_AP, Best_Perf)$
end

En el nivel `S1H1`, en cambio, la instalación se podría realizar de diferentes formas, en función de cómo se haga uso de la información de instalación alma-

cenada en el nivel inferior de la jerarquía. Existen tres posibles opciones: utilizar en el caso base la versión de la rutina básica de multiplicación de matrices optimizada para el nodo `S1H1-s0h1`, ejecutar de forma sucesiva multiplicaciones básicas en unidades de procesamiento básicas que exploten la heterogeneidad del nodo (`S1H1-s0h0`), o ejecutar multiplicaciones básicas de Strassen asignadas a las unidades básicas (`S1H1-s1h0`). Cada posibilidad correspondería a una implementación diferente de Strassen a nivel de nodo (`H1`). El algoritmo 7 muestra cómo se llevaría a cabo el proceso de instalación. Para ello, se hace uso de la función *SEARCH_S1H1* del algoritmo 8, que permite indicar, por medio del parámetro *METHOD*, cómo se ha de utilizar la información de nivel inferior para instalar la rutina en este nivel.

Algoritmo 7: Instalación de la rutina de multiplicación de Strassen en el nivel `S1H1` en un nodo compuesto por CPU+GPU.

Input: *Installation Set*, ***IS***
CU: Node with CPU+GPU

for *each n in* ***IS*** **do**
  $AP = \left\{base_i = \frac{n}{i}, \text{ con } i = 1,2,4,\ldots\right\}$
  $(Best_AP, Best_Perf) =$ **SEARCH_S1H1**$(n, CU, AP, METHOD)$
  $Performance_Installation[CU] \mathrel{+}= (n, Best_AP, Best_Perf)$
end

4.5.2 Factorización LU

La factorización (o descomposición) LU, es una operación algebraica que consiste en descomponer una matriz $A \in R^{n\times n}$, en una matriz triangular inferior, L, y una matriz triangular superior, U, obteniendo $A = LU$. Esta factorización es ampliamente utilizada para la resolución de sistemas de ecuaciones de la forma $Ax = b$, donde A es la matriz de coeficientes a factorizar, x es el array solución del sistema, y b el array de términos independientes.

Actualmente, existe un gran número de librerías de álgebra lineal densa que disponen de una rutina que permite realizar directamente la factorización LU de una matriz. Tal y como muestra la figura 4.6, la operación de factorización se lleva a cabo de forma recursiva, considerando en cada paso submatrices cuyo tamaño

Algoritmo 8: Función para obtener los mejores valores de los parámetros algorítmicos y el rendimiento con diferentes métodos de acceso, desde el nivel `S1H1` de la jerarquía, a la información de instalación almacenada en niveles inferiores.

SEARCH_S1H1(n, CU, AP, $METHOD$)
Result: $Best_AP$, $Best_Performance$
for *each possible combination of* AP *values for* CU: AP_k **do**
    if $METHOD$ = *s0h0* **then**
        $Performance_k$ = performance obtained by executing the problem (n, AP_k) with basic routines of level `s0` with $Best_AP$ for each CU_j inside CU
    end
    if $METHOD$ = *s0h1* **then**
        $Performance_k$ = performance obtained by executing the problem (n, AP_k) with basic routines of level `s0` with $Best_AP$ for CU
    end
    if $METHOD$ = *s1h0* **then**
        $Performance_k$ = performance obtained by executing the problem (n, AP_k) with routine of level `s1` with $Best_AP$ for CU
    end
    if $Performance_k > Best_Performance$ **then**
        $Best_Performance = Performance_k$
        $Best_AP = AP_k$
    end
end

viene determinado por un tamaño de bloque establecido. Por tanto, este será el parámetro algorítmico a determinar para esta rutina en este nivel de la jerarquía software. Asimismo, la implementación de esta rutina invoca internamente a otras rutinas de álgebra lineal de nivel inferior que intervienen en el cálculo de la factorización, como la rutina encargada de resolver un sistema triangular o la propia rutina de multiplicación de matrices, entre otras. El uso de estas rutinas de nivel inferior define la jerarquía software a considerar en la instalación de la rutina en la plataforma computacional. Así pues, la metodología de optimización podría ser aplicada de forma que la rutina de factorización LU se instale invocando a las rutinas de nivel inferior previamente instaladas en el sistema, haciendo uso de la información de optimización almacenada para el nivel de la jerarquía en el que está siendo instalada, en vez de invocar a las rutinas propias de la librería de álgebra lineal utilizada.

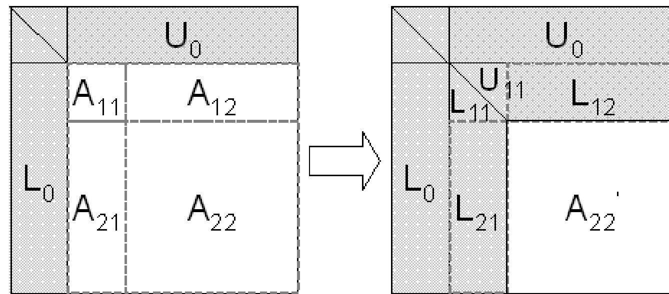

Figura 4.6: Esquema recursivo para la factorización LU de una matriz.

Teniendo en cuenta la jerarquía software descrita, el ejemplo propuesto se centra en mostrar cómo se instalaría la rutina de factorización LU aplicando la metodología de optimización jerárquica en los mismos niveles hardware y software considerados para la multiplicación de Strassen. Se parte de un escenario en el que ya se encuentra instalada la rutina básica de multiplicación de matrices en los niveles `S0H0` y `S0H1` para un determinado CI, por tanto, se dispone de la información de instalación almacenada en dicho nivel de la jerarquía.

Dado que el objetivo es mostrar cómo se puede extender la metodología en la jerarquía software mediante el uso de rutinas de mayor nivel, se considerará una implementación directa de la factorización LU con llamadas a las rutinas básicas que la constituyen, de forma que la rutina pueda hacer uso de la versión optimizada de la rutina de multiplicación de matrices cuando sea instalada de forma jerárquica en el sistema. De esta forma, en el nivel `S1H0`, la rutina se instalaría tal y como indica el algoritmo 9, variando el valor del tamaño de bloque para cada tamaño de problema. La función *LU_S1H0* ejecuta la implementación directa de la factorización con el tamaño de bloque actual, e invoca internamente a la versión de la multiplicación básica optimizada para la unidad de cómputo considerada (CPU multicore en este caso). Como resultado, devuelve las prestaciones obtenidas, que se usarán para determinar el mejor valor del tamaño de bloque para el tamaño de problema actual. En el nivel `S1H1`, en cambio, la instalación se llevaría a cabo con el algoritmo 10, de forma similar a como se ha realizado con la multiplicación de Strassen. Finalmente, la información obtenida con cada tipo de instalación se almacena en el nivel actual, con el fin de que pueda ser utilizada posteriormente en el siguiente nivel de la jerarquía software por parte de rutinas que invoquen a la factorización LU.

Algoritmo 9: Instalación de la rutina de factorización LU en el nivel `S1H0` en una CPU multicore.

Input: *Installation Set*, IS
DGS: Distribution Grain Size for Block_Size, nb

for *each* n *in* IS **do**
  $nb = \left\{ \frac{i \cdot DGS}{100} \cdot n, \text{ con } i = 0, \ldots, \frac{100}{DGS} \right\}$
  for *each* nb **do**
    $Performance = \mathbf{LU_S1H0}(n, nb, CPU)$
    if $Performance > Best_Perf$ **then**
      $Best_Perf = Performance$
      $Best_AP = nb$
    end
  end
  $Performance_Installation[CPU] \mathrel{+}= (n, Best_AP, Best_Perf)$
end

Algoritmo 10: Instalación de la rutina de factorización LU en el nivel `S1H1` en un nodo compuesto por CPU+GPU.

Input: *Installation Set*, IS
CU: Node with CPU+GPU

for *each* n *in* IS **do**
  $AP = \left\{ nb_i = \frac{i \cdot DGS}{100} \cdot n, \text{ con } i = 0, \ldots, \frac{100}{DGS} \right\}$
  $(Best_AP, Best_Perf) = \mathbf{SEARCH_S1H1}(n, CU, AP, METHOD)$
  $Performance_Installation[CU] \mathrel{+}= (n, Best_AP, Best_Perf)$
end

4.6 Conclusiones

Este capítulo describe la metodología de optimización multinivel propuesta en esta tesis para la instalación de rutinas de álgebra lineal sobre plataformas heterogéneas siguiendo un enfoque jerárquico tanto a nivel hardware como software.

Su funcionamiento se ilustra con diversos ejemplos en los que se consideran diferentes escenarios hardware y software. En ambos casos, se toma como referencia la rutina de multiplicación de matrices y se muestra, por una parte, cómo se instalaría de forma jerárquica haciendo uso de diferentes configuraciones hardware y, por otra, cómo se podría aplicar la metodología para instalar otras rutinas

de nivel superior (como la multiplicación de Strassen y la factorización LU) que invoquen internamente a la versión optimizada de dicha rutina.

El carácter genérico de la metodología propuesta, permite que pueda ser extendida tanto para el sistema hardware (elementos computacionales) como para el software (rutinas y librerías de álgebra lineal), considerando nuevos sistemas lógicos y físicos y garantizando siempre una ejecución optimizada de las rutinas en el sistema computacional empleado mediante el uso de la información de rendimiento almacenada en los diferentes niveles de la jerarquía.

En los siguientes capítulos se muestran experimentos que apoyan la validez de la metodología, tanto con distintos niveles en la jerarquía hardware y software, como con librerías de álgebra lineal diferentes a las utilizadas hasta el momento.

Capítulo 5

Ejemplos de aplicación de la metodología

En este capítulo, se mostrará el funcionamiento de la metodología de instalación jerárquica con algunas rutinas paralelas básicas de álgebra lineal. Este software está conformado por una jerarquía de rutinas de varios niveles, e incluye la multiplicación de matrices en el nivel más básico e implementaciones de la multiplicación de Strassen y de la factorización LU en un nivel superior. El sistema computacional sobre el que se llevará a cabo el estudio experimental es el cluster **Heterosolar**, compuesto por varios nodos de cómputo que incluyen CPUs multicore y uno o varios coprocesadores GPU y/o Intel Xeon Phi.

Cabe destacar que el objetivo de este trabajo no es obtener implementaciones altamente eficientes de estas rutinas, sino mostrar cómo la metodología que se propone puede ayudar a aproximarnos a ese objetivo de manera sistemática. Tampoco se trata de realizar un estudio exhaustivo de diferentes implementaciones y optimizaciones de las rutinas que se estudian en los sistemas computacionales que se consideran, sino que se mostrarán ejemplos puntuales de utilización de la metodología en distintos niveles de la jerarquía tanto software como hardware. La tabla 5.1 muestra, para cada rutina, las unidades de cómputo en las que se ha llevado a cabo este estudio experimental en cada uno de los niveles de la jerarquía considerados para ilustrar el funcionamiento de la metodología. Cada nivel se referencia utilizando la notación empleada en la figura 4.3 del capítulo anterior. Se

observa que en el estudio realizado con la multiplicación de Strassen se ha hecho uso de nodos virtuales en el nivel `H2`. Este concepto va a permitir definir nuevas subplataformas hardware como resultado de la agrupación de unidades de cómputo de diferentes formas, bien a nivel de unidades básicas (`H0`) dentro de cada nodo (`H1`), o a nivel de cluster (varios nodos de cómputo o nodos virtuales de nivel `H1`). De esta forma se dota al estudio de una mayor heterogeneidad, dado que se puede analizar cómo se comporta la metodología ante cualquier configuración hardware.

<table>
<tr><th></th><th>UC Básicas (H0)</th><th>Nodo (H1)</th><th>Plataforma (H2)</th></tr>
<tr><td rowspan="3">MM (S0)</td><td>CPU</td><td>CPU+multiGPU</td><td rowspan="3">cluster</td></tr>
<tr><td>GPU</td><td>CPU+multiMIC</td></tr>
<tr><td>MIC</td><td>CPU+GPU+multiMIC</td></tr>
<tr><td rowspan="2">Strassen (S1)</td><td>CPU</td><td>CPU+multiGPU</td><td rowspan="2">nodos virtuales</td></tr>
<tr><td>GPU</td><td>CPU+multiMIC</td></tr>
<tr><td>LU (S1)</td><td>-</td><td>CPU+multiGPU</td><td>-</td></tr>
</table>

Tabla 5.1: Rutinas de álgebra lineal y unidades de cómputo consideradas en el estudio experimental.

5.1 Instalación de rutinas en elementos computacionales básicos

Los elementos computacionales básicos que se van a considerar en el estudio experimental son: CPU multicore, GPU e Intel Xeon Phi. Estos elementos de cómputo se encuentran ubicados en el nivel `H0` y sobre ellos comenzaremos el estudio experimental considerando la instalación de la multiplicación de matrices (nivel `S0H0`) y de rutinas de mayor nivel que invocan a esta rutina básica y que, en este caso, trabajarán también con un único elemento computacional (nivel `S1H0`).

5.1.1 Instalación de la multiplicación de matrices en un elemento computacional

Como ya se ha comentado, la rutina de multiplicación de matrices hace llamadas a rutinas de librerías ya optimizadas (MKL en CPU multicore e Intel Xeon

Phi y cuBLAS en GPU), y los únicos parámetros algorítmicos que se consideran son el número de threads en CPU y en Xeon Phi.

Para instalar una rutina se establece un Conjunto de instalación (CI), que contiene los tamaños de problema con los que se va a experimentar para seleccionar los parámetros algorítmicos con los que se obtiene el menor tiempo de ejecución. Los valores de los parámetros que proporcionan el menor tiempo se almacenan, junto con las prestaciones obtenidas, para su posterior uso en la instalación en niveles superiores de software o hardware. Además, hay que tener en cuenta que durante la instalación se realizan experimentos con distintos valores posibles de los tamaños, lo que puede llevar a tiempos de ejecución elevados y, en algunos casos, prohibitivos.

Como ejemplo, se muestran resultados con un CI para los elementos computacionales básicos del cluster **Heterosolar**, con la matriz A de tamaño $n \times n$, con $n = \{1000, 2000, 3000, 4000, 5000, 6000\}$, y la matriz B de tamaño $n \times n_i$, con n_i obtenido variando un 2% el valor de n: $\frac{n}{50}$, $2\frac{n}{50}$, $\ldots$, n. Esto da lugar a un total de 300 tamaños con los que se hacen multiplicaciones durante la instalación. En GPU, al no tener que seleccionar ningún parámetro algorítmico, sólo se realizarán las multiplicaciones matriciales y se almacenarán los GFlops obtenidos. En CPU y Xeon Phi, en cambio, se varía el número de threads, por tanto, si se realiza una búsqueda exhaustiva y no se utiliza ninguna estrategia para disminuir el número de experimentos, se realizarán $300 \cdot t$ experimentos, con t el número máximo de threads soportados por la unidad de cómputo. La tabla 5.2 muestra, para cada componente computacional de los diferentes nodos del cluster, la información almacenada para los tamaños de matrices cuadradas ($n_i = n$). Las tablas completas incluirían 49 entradas más por cada valor de n. Se observa que:

- La capacidad computacional de cada componente dentro de cada nodo varía mucho. Por ejemplo, la GPU de **venus** es alrededor de 20 veces más lenta que uno de sus coprocesadores Intel Xeon Phi, y unas 13 veces más lenta que su CPU; mientras que la GPU de **saturno** es la más rápida y tiene una capacidad computacional aproximadamente 7 veces superior a su CPU, que dispone de 24 cores. Por tanto, la metodología de auto-optimización debe permitir tomar decisiones distintas dependiendo de la velocidad de los

marte mercurio	CPU		GTX480
n	*GFlops*	*threads*	*GFlops*
1000	58	6	97
2000	50	6	127
3000	50	6	139
4000	53	6	146
5000	55	6	148
6000	56	6	151

saturno	CPU		K20c
n	*GFlops*	*threads*	*GFlops*
1000	124	20	920
2000	133	22	969
3000	142	21	1033
4000	144	23	1020
5000	148	22	1017
6000	139	48	1041

jupiter	CPU		C2075	GTX590
n	*GFlops*	*threads*	*GFlops*	*GFlops*
1000	167	12	292	146
2000	159	24	310	152
3000	191	12	308	154
4000	177	24	309	155
5000	181	24	298	152
6000	182	24	296	152

venus	CPU		Xeon Phi		GT640
n	*GFlops*	*threads*	*GFlops*	*threads*	*GFlops*
1000	364	12	529	224	31
2000	279	12	641	220	31
3000	354	11	651	224	33
4000	304	12	632	224	32
5000	383	12	651	224	32
6000	437	12	670	220	33

Tabla 5.2: Información almacenada en la instalación de la rutina de multiplicación de matrices para las diferentes unidades básicas de procesamiento de los nodos de cómputo del cluster **Heterosolar** (sólo se muestran las entradas correspondientes a matrices cuadradas).

elementos computacionales con los que se trabaja y de las prestaciones de las librerías que se utilizan.

- Las prestaciones alcanzadas aumentan ligeramente con el tamaño del problema. Esto se nota especialmente si se incluyen las entradas en las que $\boldsymbol{B}$ es rectangular, pues cuanto menor es el tamaño de los bloques en que se divide la matriz, menor es el coste computacional y, por tanto, las prestaciones disminuyen. Esto hay que tenerlo en cuenta cuando en una multiplicación matricial se vaya a seleccionar el tamaño de los bloques a asignar a los elementos computacionales en un sistema hardware de mayor nivel (`SOH1`).

- El número de threads seleccionado es cercano al de cores físicos. En **marte**, por ejemplo, el número de threads seleccionado es siempre seis.

- El número preferido de threads puede disminuir para tamaños menores de la matriz $\boldsymbol{B}$. Por ejemplo, en **jupiter**, con $n = 1000$ y $n_i = \{20, 40, 60, 80, 100\}$, el número de threads seleccionado para los diferentes valores de n_i es 8, 8, 9, 12 y 8, que no son tan cercanos al número de cores.

- En **saturno** y **jupiter** el *hyperthreading* está habilitado, por lo que se ha experimentado hasta 48 y 24 threads, respectivamente, y los valores preferidos están cerca del número de cores físicos o del máximo número de threads soportado, pero las diferencias en las prestaciones son mínimas. El comportamiento con *hyperthreading* se observa mejor con los resultados de un experimento, que se muestran en las figuras 5.1 (**jupiter**) para tamaños 3500 y 6500 y 5.2 (**saturno**) para tamaño 6500. Las figuras muestran el número de threads (en rojo) con los que se obtiene el menor tiempo de ejecución en función del método de auto-optimización que utilicemos, y los GFlops (en azul) obtenidos experimentalmente usando ese número de threads cuando varía el máximo número de cores lógicos a utilizar. De este modo consideramos en cada nodo subnodos con un número variable de cores (desde 1 hasta el número máximo de cores en el nodo). Se observan fluctuaciones en la explotación del *hyperthreading* para un número de cores lógicos mayor que el de cores físicos en el sistema, por lo que no se explota satisfactoriamente esta característica cuando se utiliza la rutina de multiplicación de MKL. En cualquier caso, algunas de las fluctuaciones son propias

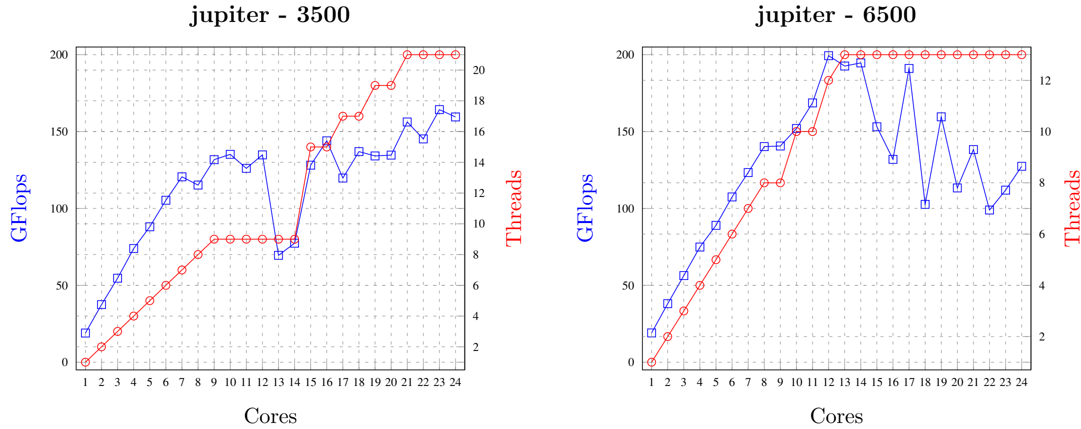

Figura 5.1: Rendimiento obtenido y número de threads seleccionado durante la instalación de la rutina de multiplicación de matrices en **jupiter**, variando el número de cores lógicos para tamaños 3500 y 6500.

de ejecuciones diferentes, al haberse realizado una única ejecución para cada número de cores.

- En el coprocesador Intel Xeon Phi se ha experimentado variando el número de threads en valores múltiplos de 4, pues cada uno de los cores de que dispone permite usar hasta cuatro threads hardware. El máximo valor alcanzado para el número de threads ha sido 224, ya que el coprocesador reserva uno de los cores para la gestión de las comunicaciones desde/hacia la CPU). Para los valores que se muestran en la tabla, el número de threads es 224 o 220 (en algunos casos). Al igual que en la CPU, para multiplicaciones rectangulares, con mucha menor carga computacional, el número de threads seleccionado puede ser mucho menor; por ejemplo, para $n = 1000$ y $n_i = \{20, 40\}$ los mejores resultados se obtienen con 68 y 52 threads, respectivamente.

Los tiempos de instalación se muestran en la tabla 5.3. Se observa que son mucho menores en GPU que en CPU y Xeon Phi. Esto se debe a que en GPU no se consideran parámetros algorítmicos, por tanto, simplemente se lleva a cabo una ejecución por cada entrada del CI para almacenar las prestaciones alcanzadas, de cara a usarlas en niveles superiores de la jerarquía de instalación. Cuanto mayor es el número de cores, mayor es el tiempo de instalación en CPU y Xeon Phi. En **saturno**, por ejemplo, el tiempo empleado es de 8 horas. Además, la tabla muestra los tiempos cuando se realiza una única ejecución por tamaño de

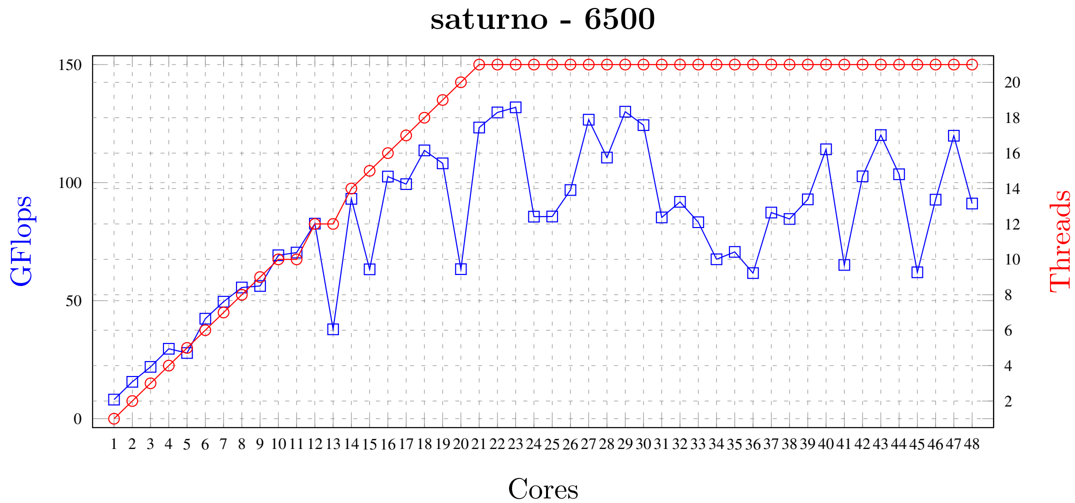

Figura 5.2: Rendimiento obtenido y número de threads seleccionados durante la instalación de la rutina de multiplicación de matrices en **saturno**, variando el número de cores lógicos para tamaño 6500.

problema y valor del parámetro algorítmico, pero puede ser preferible realizar varios experimentos para evitar fluctuaciones debidas a ejecuciones concretas. El software de instalación (Anexo A) permite elegir el número de experimentos. En este caso, la instalación se realizó con cinco experimentos, con lo que el tiempo en **saturno** aumentaría a 40 horas.

marte		**saturno**		**jupiter**			**venus**		
CPU	GT480	CPU	K20	CPU	C2075	GTX590	CPU	GT640	MIC
8491	791	29147	133	9655	183	297	2867	1241	9707

Tabla 5.3: Tiempo de instalación (en segundos) en los diferentes elementos computacionales de cada nodo del cluster **Heterosolar**.

El tiempo de instalación se puede reducir considerablemente si se tienen en cuenta las observaciones realizadas sobre el número de threads óptimo, que es cercano al número de cores, y si se utiliza una búsqueda guiada del valor de ese parámetro [51]. La idea básica de una búsqueda guiada es que se empieza buscando el valor del parámetro con el que se obtiene el menor tiempo de ejecución para el menor tamaño de problema del CI y se utiliza como punto de partida para una búsqueda local con el siguiente tamaño del CI. De esta forma, sólo se experimenta con todos los posibles valores para el tamaño más pequeño, pero

para el resto de tamaños se realizan menos experimentos y con valores de los parámetros con los que las ejecuciones son más rápidas, produciéndose así una reducción importante en el tiempo de instalación. Por ejemplo, en el experimento realizado en **jupiter** con $n = 1000$ y $n_i = \{20, 40, 60, 80, 100\}$, el número de experimentos pasaría de 120 (5 tamaños y número de threads entre 1 y 24) a 44.

En cualquier caso, como lo que se pretende es mostrar la validez de la metodología de instalación jerárquica, no se optimiza la instalación en este nivel inicial, aunque el software de instalación se está extendiendo para que contemple mejoras de este tipo.

Una posible mejora podría consistir en utilizar una metodología híbrida teórico-experimental, tal como se comentó en el Capítulo 3, para lo que sería necesario analizar y comparar las prestaciones obtenidas experimentalmente, con las que se predicen teóricamente con la información generada mediante los experimentos realizados con los tamaños del CI. Un enfoque similar se puede aplicar utilizando la metodología de instalación jerárquica, pues la información que se almacena tras la instalación puede ser utilizada posteriormente para decidir los valores de los parámetros algorítmicos a seleccionar para el tamaño de problema a resolver. De esta forma, para resolver un problema de tamaño n, se utilizarán los valores de los parámetros algorítmicos almacenados para el tamaño de problema n' más cercano a n.

La figura 5.3 compara, para la CPU de cada uno de los nodos en **Heterosolar**, las prestaciones teóricas obtenidas a partir de la información generada en la instalación para el CI inicial, con las obtenidas experimentalmente utilizando el número de threads seleccionado. Se ha usado un CI = $\{500, 1000, 1500, \ldots, 6000\}$ y se ha variado el número de cores, como en las figuras 5.1 y 5.2. Para un tamaño dado, n, se selecciona el tamaño del CI más cercano, n_c, y los GFlops esperados (teóricos) para n son los que se obtuvieron experimentalmente para n_c. Por otro lado, los GFlops experimentales para n se obtienen realizando la multiplicación matricial para este tamaño con el valor óptimo del parámetro (número de threads) para n_c. Normalmente las prestaciones esperadas (teóricas) son mayores que las experimentales y no se aprecian diferencias significativas para un tamaño del CI (3500), uno cercano a un tamaño del CI (6500 cercano a 6000), y otro alejado de los tamaños de instalación (9500). En cualquier caso, las diferencias no son muy

importantes (al margen de algunos resultados espurios debidos a fluctuaciones en la ejecución), especialmente cuando el número de cores no es elevado. Como se ha indicado, la realización de la instalación con varias ejecuciones para cada tamaño de problema y valor de los parámetros, podría reducir tales fluctuaciones, pero a costa de un mayor tiempo de instalación.

Teniendo en cuenta el estancamiento que se produce en las prestaciones, observado en la figura 5.3 al aumentar el número de cores, para rutinas en las que haya que llevar a cabo varias operaciones al mismo tiempo (por ejemplo, una multiplicación de Strassen, en la que se realizan siete multiplicaciones con matrices de dimensiones la mitad que las de la matriz original), puede ser preferible considerar subgrupos de cores y realizar operaciones simultáneas sobre subgrupos distintos. Este aspecto habrá que analizarlo en el caso de rutinas de mayor nivel que la multiplicación de matrices, considerando la CPU como combinación de varios elementos computacionales básicos, cada uno formado por un grupo de cores, lo que dará lugar a una instalación del tipo `S1H1`.

En el caso de las GPUs no se selecciona ningún parámetro, pero hay que tener en cuenta que para que las prestaciones teóricas se aproximen a las experimentales habrá que incluir el coste de las transferencias de datos entre CPU y GPU. La figura 5.4 muestra el cociente de las prestaciones teóricas considerando sólo la computación con respecto a la experimental, para los cuatro nodos de **Heterosolar** considerados. En **jupiter** se muestran los resultados para la GPU Tesla. Las prestaciones experimentales son siempre menores que las teóricas, lo que deberá tenerse en cuenta para realizar una buena selección del reparto de trabajo cuando se usen de forma conjunta CPU y GPU (lo mismo ocurre al utilizar el coprocesador Xeon Phi). Cuando el tamaño de las matrices aumenta, las diferencias disminuyen, por lo que tomar decisiones teniendo en cuenta sólo la computación puede ser satisfactorio para problemas de grandes dimensiones. Sin embargo, en **saturno** la desviación es mucho mayor, lo que dificulta la toma de decisiones si no se tienen en cuenta las comunicaciones. En resumen, el software de instalación debe tener en cuenta estas posibles diferencias en el comportamiento al usar CPU+GPU dentro de un nodo cuando se trabaje en el nivel 1 de hardware (`H1`).

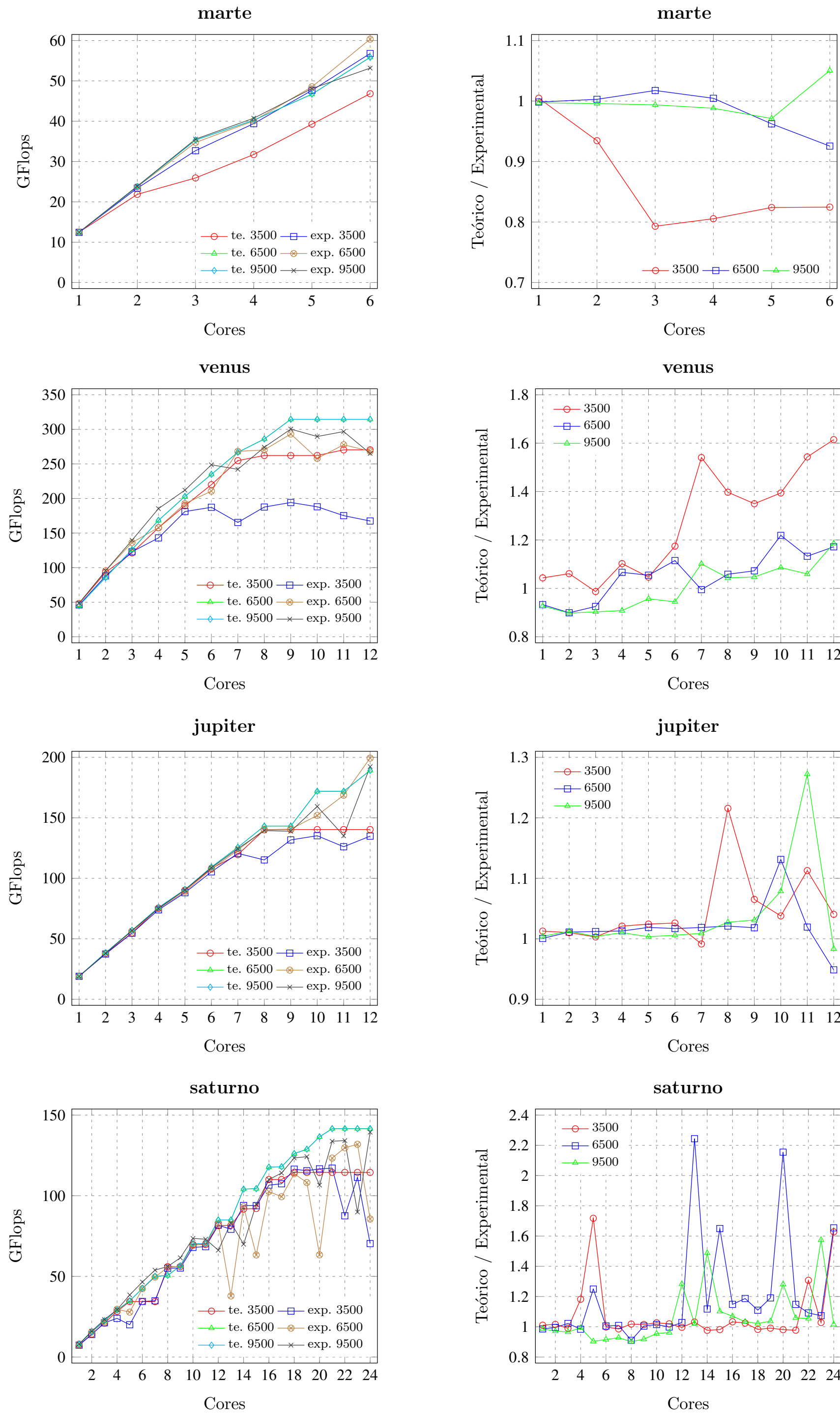

Figura 5.3: Comparación de las prestaciones teóricas y experimentales en las CPU de cuatro nodos de **Heterosolar** variando el número de cores disponibles. A la izquierda, comparación de las prestaciones para matrices cuadradas de tamaños 3500, 6500 y 9500. A la derecha, cociente de las prestaciones para los tres tamaños.

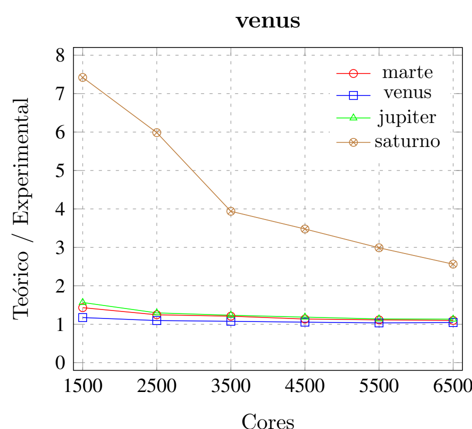

Figura 5.4: Evolución del cociente de las prestaciones teóricas respecto a las experimentales de la rutina de multiplicación de matrices con cuBLAS en GPUs de distintos nodos de **Heterosolar** y para distintos tamaños de matrices cuadradas.

5.1.2 Instalación de la multiplicación de Strassen en un elemento computacional

Cuando aumenta el nivel software (`S1H0`), hay rutinas que invocan internamente a rutinas de nivel 0, donde se ha considerado la multiplicación de matrices tradicional. Una rutina de nivel 1 puede ser, por ejemplo, la multiplicación de Strassen, que llama en el nivel base de la recursión a rutinas de nivel 0. El parámetro de nivel 1 a determinar es el nivel de recursión óptimo, siendo cero en el caso de que no se realice recursión (lo que indica que se utiliza la multiplicación directa). El valor óptimo de este parámetro se obtiene experimentalmente para los distintos tamaños de matriz del CI en cada elemento básico de proceso (CPU, GPU o Xeon Phi). La obtención de los parámetros de la rutina de nivel 0 (número de threads en CPU y Xeon Phi) se delega a la instalación de la rutina de nivel inferior en cada unidad de cómputo, con lo que el esquema de optimización jerárquica de la multiplicación de Strassen sería de la forma `S1H0-s0h0`.

Tal como se ha comentado, la auto-optimización se puede llevar a cabo básicamente de dos formas: con modelos empíricos del tiempo de ejecución (`EmpMod`) o con experimentos durante la instalación de la rutina (`ExpIns`) [34, 35]. Además, la optimización se puede hacer de forma estática, para decidir la forma de ejecución de la rutina antes de que se produzca, o dinámicamente durante la ejecución

de la misma. Se analizan brevemente los resultados de algunos experimentos sobre estos aspectos para la multiplicación de Strassen sobre multicore. Se puede encontrar más información en [89].

En `EmpMod`, el modelo del tiempo de ejecución de la ecuación 2.3 se puede utilizar con ejecuciones de la rutina de Strassen para tamaños de problema pequeños dentro del sistema computacional donde se instala junto con información de la multiplicación de matrices básica generada en su instalación en el multicore, con lo que se reutiliza información de nivel 0 de software, para estimar así los valores de los parámetros en la ecuación 2.3.

En `ExpIns`, se llevan a cabo experimentos cuando la rutina se instala en el sistema, y esta información se incorpora a la rutina justo al acabar la instalación. A diferencia de lo que se hace en `EmpMod`, en este caso se experimenta con la multiplicación de Strassen con tamaños mayores de problema, y la información que se usa en la ejecución proviene de las ejecuciones realizadas para los tamaños del CI, sin la utilización de modelos del tiempo de ejecución.

En el caso de optimización dinámica (`DynOpt`), los valores de los parámetros se seleccionan por medio de experimentos mientras se está ejecutando la rutina, por lo que habrá que mantener baja la sobrecarga de la gestión dinámica.

La rutina básica preferida para la multiplicación en el caso base del algoritmo 1, varía con el tamaño del caso base, y el valor k_3 de la ecuación 2.3 depende de la rutina que se usa, con lo que se podría utilizar una aproximación de polialgoritmos [8] en la que la rutina a utilizar es un nuevo parámetro a tener en cuenta en el proceso de auto-optimización.

Se muestran los resultados de experimentos en las CPU multicore de los nodos **marte**, **venus** y **saturno** del cluster **Heterosolar**, analizando en cada uno de ellos las tres técnicas de optimización comentadas. En primer lugar, se llevan a cabo experimentos en un único core del nodo, por tanto, sólo se compara el comportamiento de la multiplicación de Strassen con respecto a la básica, pues no se explota el paralelismo. Los resultados en los tres sistemas son similares, con variaciones debidas a la diferencias en la capacidad computacional.

Para aplicar el modelo del tiempo de ejecución de la ecuación 2.3 hay que estimar los valores de k_3 y k_2 para el sistema donde se trabaja. Se utilizan matrices pequeñas para que el tiempo de ejecución sea bajo. Estos tamaños de problema se incluirían entre los del CI a nivel 1. En **marte**, por ejemplo, si consideramos tamaño 1000 y la multiplicación de Strassen con una matriz de ese tamaño y nivel de recursión 1, los valores de k_3 y k_2 serían $k_3 = 0.0847$ μseg y $k_2 = 5.06$ μseg. Hay una gran diferencia entre estos dos valores, lo que puede deberse al uso de la rutina `dgemm` optimizada de MKL para las multiplicaciones básicas y a que las sumas y restas se implementan directamente. De todas formas, la metodología es válida para otras implementaciones, y si se experimenta con más tamaños y se aproximan los valores por mínimos cuadrados, se pueden obtener mejores resultados. Con los tamaños considerados, los resultados son satisfactorios y los experimentos tardan menos de un segundo.

Para `ExpIns`, se llevan a cabo experimentos en **marte** con CI = {1000, 3000, 5000, 7000} y se almacena el nivel de recursión óptimo (se ha considerado la multiplicación directa, nivel 0, y otros tres niveles en los experimentos) para usarlo en tiempo de ejecución. Para reducir el tiempo de instalación, para cada tamaño del CI se empieza ejecutando la rutina con nivel 0 y se aumenta el nivel mientras siga disminuyendo el tiempo de ejecución. De esta forma, se evitan las ejecuciones más costosas. El tiempo de instalación es de alrededor de 285 segundos, que es mucho mayor que con el modelado empírico, pero no es excesivo y con mejores predicciones.

Se comparan los métodos de instalación para el conjunto de tamaños {2000, 4000, 6000, 8000}, cuyos niveles de recursión óptimos obtenidos experimentalmente son 1, 2, 3 y 3, respectivamente, que coinciden con los obtenidos con los dos métodos de auto-optimización, con lo que ambos son satisfactorios y con menor tiempo de instalación para el modelado empírico. La media de la mejora de la multiplicación de Strassen con respecto al método directo está en torno al 37%. La diferencia media con respecto al mayor nivel con el que se ha experimentado (nivel 3) es de alrededor del 3.3%, por lo que utilizar por defecto este nivel es una buena opción, especialmente para problemas de mayor tamaño para los que ese nivel de recursión es el óptimo.

En sistemas más potentes es más difícil estimar de forma precisa el tiempo de ejecución, lo que dificulta la selección satisfactoria del nivel de recursión. Un core de **saturno** es unas dos veces más lento que uno de **marte** cuando realiza una multiplicación de matrices, mientras que un core de **venus** es unas 2.5 veces más rápido que uno de **marte**. Para el sistema más lento, los niveles de recursión también se seleccionan satisfactoriamente para los cuatro tamaños (2, 3, 3 y 3), y la mejora media respecto a la multiplicación directa es del 79.6 %, y los tiempos de instalación son de 0.6 y 437 segundos para `EmpMod` y `ExpIns`, respectivamente.

Para el sistema más rápido (**venus**), los resultados se muestran en la tabla 5.4. Se comparan los niveles seleccionados con los dos métodos y los mejores obtenidos experimentalmente (`Exp`). Menos de la mitad de las selecciones son correctas, y los niveles de recursión con los dos métodos de optimización son distintos. La diferencia con el respecto al mejor tiempo de ejecución es de 4.3 % para `EmpMod` y 1 % para `ExpIns`, lo que se puede considerar satisfactorio. Además, el tiempo de instalación queda en 0.2 y 60 segundos para `EmpMod` y `ExpIns`, respectivamente.

n	`Exp`	`EmpMod`	`ExpIns`
2000	1	1	1
4000	1	**2**	1
6000	2	**3**	**1**
8000	2	**3**	**1**

Tabla 5.4: Nivel de recursión con el que se obtiene el menor tiempo de ejecución de forma experimental (`Exp`), y nivel seleccionado con modelado empírico (`EmpMod`) y con instalación experimental (`ExpIns`) para diferentes tamaños de problema.

Los experimentos que se han mostrado hasta ahora corresponden a ejecuciones con un único core, por tanto, no incluyen paralelismo. Dado que MKL es una librería *multi-threading*, se puede explotar el paralelismo de forma implícita al llamar a su rutina `dgemm`. De esta forma, se reduce el tiempo de ejecución y será más difícil que la multiplicación de Strassen mejore a la multiplicación directa. La tabla 5.5 muestra cómo varía el nivel de recursión de la multiplicación de Strassen con el que se obtiene el menor tiempo de ejecución (`Exp`), para matrices de tamaño 4000, 6000 y 8000, en la CPU multicore de diferentes nodos y variando el número de threads que se usan dentro de la rutina `dgemm`. Se muestran también los niveles seleccionados con los métodos de optimización estática (`EmpMod` y `ExpIns`) y los obtenidos con ejecución dinámica (`DynOpt`). Se resaltan los casos

en que la selección no coincide con la que se obtiene menor tiempo de ejecución experimental. La tabla muestra resultados hasta seis threads, aunque **venus** tiene 12 cores y **saturno** 24. Algunas conclusiones a destacar son:

n:	4000						6000						8000					
threads:	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6
								saturno										
`Exp`	3	2	1	1	1	0	3	3	2	1	1	1	3	3	2	2	1	0
`EmpMod`	3	2	1	1	**0**	**1**	3	3	2	**2**	1	1	3	3	2	2	1	**2**
`ExpIns`	3	2	1	1	1	**1**	3	3	2	**2**	1	1	3	3	2	2	1	**1**
`DynOpt`	**2**	2	1	1	1	**1**	3	3	2	1	1	1	**4**	3	2	2	1	**1**
								marte										
`Exp`	2	2	1	0	0	0	3	1	1	1	1	1	3	3	2	1	1	1
`EmpMod`	2	2	1	**1**	**1**	**1**	3	**3**	**2**	1	1	**2**	3	3	2	**2**	**2**	**2**
`ExpIns`	2	**1**	1	0	0	**1**	3	**2**	**2**	1	1	**0**	3	**2**	2	1	1	**0**
`DynOpt`	2	**1**	1	**1**	**1**	**1**	3	**2**	1	1	1	1	3	**2**	2	1	1	1
								venus										
`Exp`	1	1	0	0	0	0	2	1	1	1	0	0	2	1	1	1	0	0
`EmpMod`	**2**	1	**1**	**1**	**1**	**1**	**3**	**2**	**2**	**2**	**2**	**1**	**3**	**2**	**2**	**2**	**2**	**2**
`ExpIns`	1	1	0	0	0	0	**1**	1	**0**	**0**	0	0	**1**	1	1	**0**	0	0
`DynOpt`	**2**	**2**	**1**	**1**	**1**	**1**	2	1	1	1	**1**	**1**	2	1	1	1	**1**	**1**

Tabla 5.5: Comparación del nivel de recursión con el que se obtiene el menor tiempo de ejecución de forma experimental (`Exp`) con el obtenido por los métodos de selección `EmpMod`, `ExpIns` y `DynOpt`, para distintos tamaños de problema y sistemas computacionales y variando el número de threads utilizado en las llamadas a la rutina de multiplicación de matrices.

- En general, el nivel de recursión óptimo disminuye para menor coste computacional: es menor para tamaños más pequeños, en sistemas más rápidos o cuando se usan más threads. Como consecuencia, el algoritmo de Strassen es competitivo con implementaciones *multi-threading* eficientes de `dgemm` con tamaños de problema muy grandes.
- La calidad de la predicción con modelado empírico empeora en sistemas más rápidos.
- El método de instalación experimental toma mejores decisiones, especialmente en sistemas rápidos, por lo que es preferible siempre que el tiempo de instalación se mantenga a niveles aceptables.

La tabla 5.6 muestra el tiempo de instalación (en segundos) de `EmpMod` y `ExpIns` con un máximo de seis cores en las tres CPUs multicore de los nodos considerados. En `EmpMod` los valores de k_3 y k_2 se estiman de forma independiente para cada número de threads entre 1 y 6, pero el tiempo de instalación es mucho menor que cuando esta se hace de forma experimental. Aún así, el tiempo de instalación de `ExpIns` en el sistema más lento es menor de 19 minutos, que es aceptable teniendo en cuenta que se toman mejores decisiones.

	venus	**marte**	**saturno**
`EmpMod`	0.40	1.04	1.58
`ExpIns`	143	708	1122

Tabla 5.6: Tiempo de instalación (en segundos) con los métodos `EmpMod` y `ExpIns` en tres CPU multicore usando un máximo de seis threads.

La figura 5.5 muestra el cociente del tiempo de ejecución de la multiplicación directa con respecto a la multiplicación de Strassen con niveles 1, 2 y 3, en las tres CPUs multicore consideradas, con matrices de tamaño 8000 y 12000, y variando el número de threads desde 1 hasta el máximo número de cores físicos del sistema. La mejora de la multiplicación de Strassen respecto a la directa está entre un 10% y un 30% cuando se usa sólo un thread, pero cuando el número de threads aumenta, la diferencia disminuye, y cuando coincide con el número de cores, las diferencias son mínimas, siendo preferible el método de Strassen con nivel de recursión 1 sólo en los sistemas con un número de cores no muy elevado.

Como hemos comentado, el nivel de recursión también puede seleccionarse de forma dinámica en tiempo de ejecución, con lo que no se necesitaría una fase de instalación de la rutina de Strassen. De esta forma, la rutina `strassen` del algoritmo 1 se llama sin caso base, y las siete multiplicaciones de tamaño la mitad se invocan aumentando el nivel de recursión, con $base = n$, $n/2$, $n/4\ldots$ Cuando el tiempo de ejecución es mayor que el de la llamada anterior, en llamadas sucesivas se utiliza el nivel con el que se obtuvo el menor tiempo. Dado que sólo hay siete llamadas, el método puede dar resultados satisfactorios sólo si el nivel de recursión no es alto, lo que sabemos que sucede teniendo en cuenta los resultados de la tabla 5.5. Una de las desventajas de este método es que no puede usar nivel 0 de recursión, que es el preferido en muchos casos. Aún así, el método se adapta bien a la velocidad del sistema y al tamaño de las matrices, tal como se muestra en

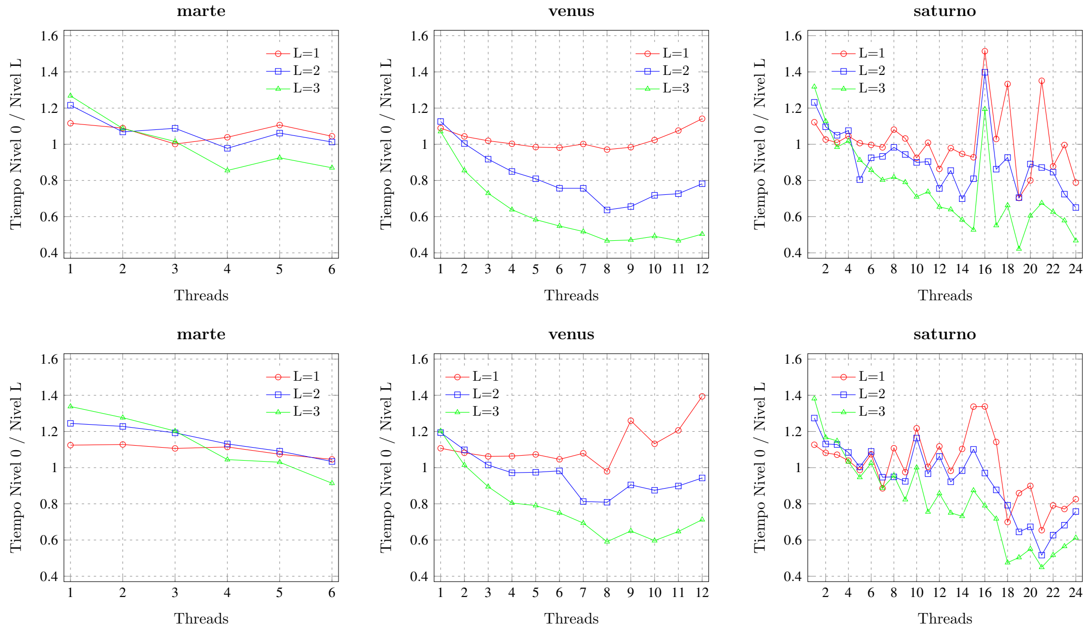

Figura 5.5: Cociente del tiempo de ejecución de la multiplicación directa con respecto a la multiplicación de Strassen con niveles de recursión 1, 2 y 3, para matrices de tamaño 8000 (arriba) y 12000 (abajo).

la figura 5.6, que muestra el cociente del tiempo de ejecución de la multiplicación de Strassen con el nivel de recursión óptimo obtenido experimentalmente, con respecto al obtenido con esta estrategia de ejecución dinámica. Los tiempos están muy próximos. Sólo para un número de threads alto (>14) los resultados con la selección dinámica no son satisfactorios, lo que se debe a que en estos casos el menor nivel de recursión es 0 (multiplicación directa).

En el caso de considerar el uso de GPUs, con mayor capacidad computacional que las CPUs multicore pero con el coste adicional de las transferencias entre CPU y GPU, la multiplicación de Strassen es todavía menos competitiva que implementaciones eficientes de la multiplicación tradicional. La figura 5.7 compara el tiempo de ejecución en GPU de la rutina de multiplicación de cuBLAS con una multiplicación de Strassen con niveles de recursión 1 y 2. Los resultados se muestran para tres GPUs con diferente capacidad computacional: Tesla K20c de **saturno**, con 4800 MBytes de memoria global y 2496 cores; GeForce GTX590 de **jupiter**, con 1536 MBytes de memoria global y 512 cores; y GeForce GT640 de **venus**, con 1 GB de memoria global y 384 cores. En todas las GPUs la tendencia

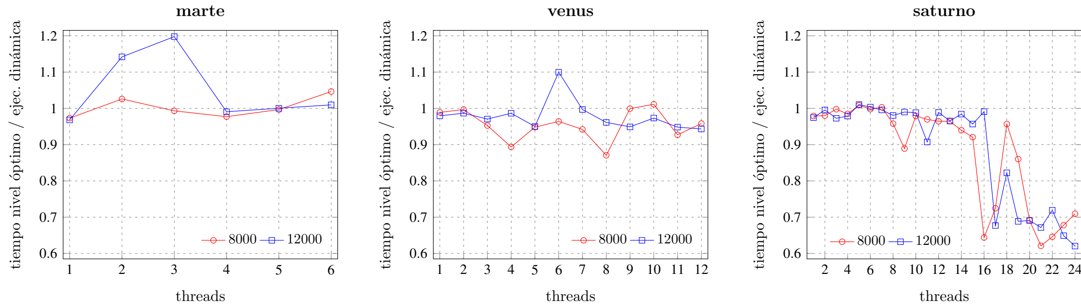

Figura 5.6: Cociente del tiempo de ejecución de la multiplicación de Strassen con el mejor nivel de recursión, obtenido experimentalmente, con respecto al tiempo de ejecución obtenido con `DynOpt`, para tamaños de matriz 8000 y 12000 en tres sistemas multicore.

es que el tiempo de ejecución de la multiplicación de Strassen se aproxime al de la multiplicación tradicional al aumentar el tamaño del problema. En las GTX590 y GT640, el tamaño máximo de problema que se utiliza es aproximadamente el que se podría almacenar en la memoria global de la GPU. Para la GPU más rápida (K20c), el tiempo de ejecución de la multiplicación de Strassen se encuentra bastante lejos del que se obtiene con cuBLAS, mientras que para la GPU más lenta (GT640) es preferible usar nivel de recursión 1 en la multiplicación de Strassen a partir de tamaño 4000. De esta forma, el método de optimización de la multiplicación de Strassen en GPU seleccionaría nivel de recursión cero (multiplicación directa con cuBLAS) en las Tesla K20c y GeForce GTX590, y en la GeForce GT640 sería cero hasta tamaño de problema 4000, y uno a partir de ese tamaño.

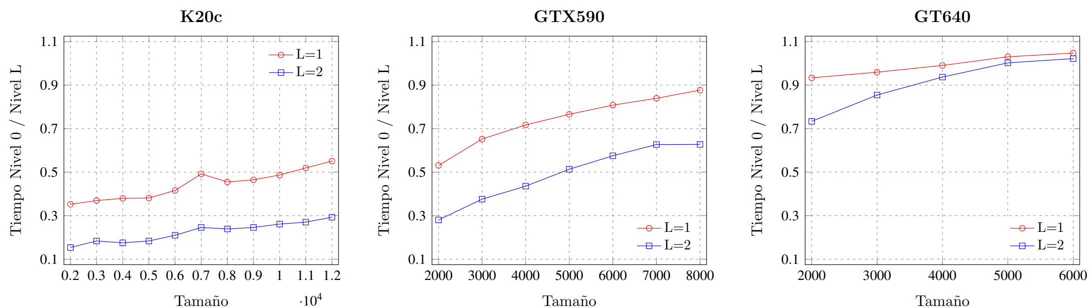

Figura 5.7: Cociente del tiempo de ejecución de la multiplicación de cuBLAS con respecto a la multiplicación de Strassen con niveles de recursión 1 y 2, en tres GPUs con distinta capacidad computacional.

Las CPU multicore, al tener normalmente menor capacidad computacional que las GPUs y no incurrir en el coste adicional de las transferencias, la situación puede ser algo distinta. La figura 5.8 muestra el cociente del tiempo de ejecución de la multiplicación directa con respecto al de la multiplicación de Strassen variando el nivel de recursión y el número de cores, para matrices de tamaño 8000 y 12000 en los nodos **jupiter** y **saturno**. Asimismo, se muestra el cociente para la GPU más lenta (GTX590, sólo para tamaño 8000 por restricciones de memoria) y la GPU más rápida (K20c). Como se ha comentado, dado que las GPUs disponen de mayor capacidad computacional, el nivel de recursión es cero (multiplicación directa), pero en CPUs multicore, cuando las matrices son suficientemente grandes, el nivel de recursión que selecciona el método de auto-optimización es uno.

5.2 Instalación a nivel de nodo

En un nodo que contiene una CPU multicore y algún coprocesador, los elementos computacionales pueden colaborar en la resolución de un problema (`H1`). Para ello, hay que tener en cuenta la velocidad relativa de cada uno de estos elementos básicos (`H0`), que se habrá obtenido de forma teórica o experimental al realizar la instalación de la rutina en cada uno de ellos. Como ejemplo, la figura 5.9 muestra la evolución del cociente de las prestaciones de la multiplicación de matrices en GPU con respecto a la obtenida con un core de CPU, en los cuatro tipos de nodos de **Heterosolar**, considerando en **jupiter**, de los dos tipos de GPUs de que dispone, la GPU Tesla. En **venus**, la GPU es aproximadamente un 35 % más lenta que un core de la CPU, lo que da como resultado que la GPU trabaje mucho más lenta que la CPU cuando se utilizan todos los cores (ver la figura 5.3), por tanto, en este caso el proceso de toma de decisiones en el nivel 1 (`H1`, el nodo) asignará muy poco trabajo (o ninguno) a la GPU. En los otros nodos, en cambio, la situación es diferente. La GPU de **marte** es aproximadamente 10 veces más rápida que un core de CPU, por lo que se espera que la matriz $\boldsymbol{B}$ se distribuya entre los dos elementos de cómputo para realizar la multiplicación. En **saturno**, las prestaciones en GPU aumentan con respecto a las de CPU cuando aumenta el tamaño de las matrices, por lo que se espera que el tamaño asignado a la CPU sea pequeño y disminuya conforme aumente el tamaño del problema.

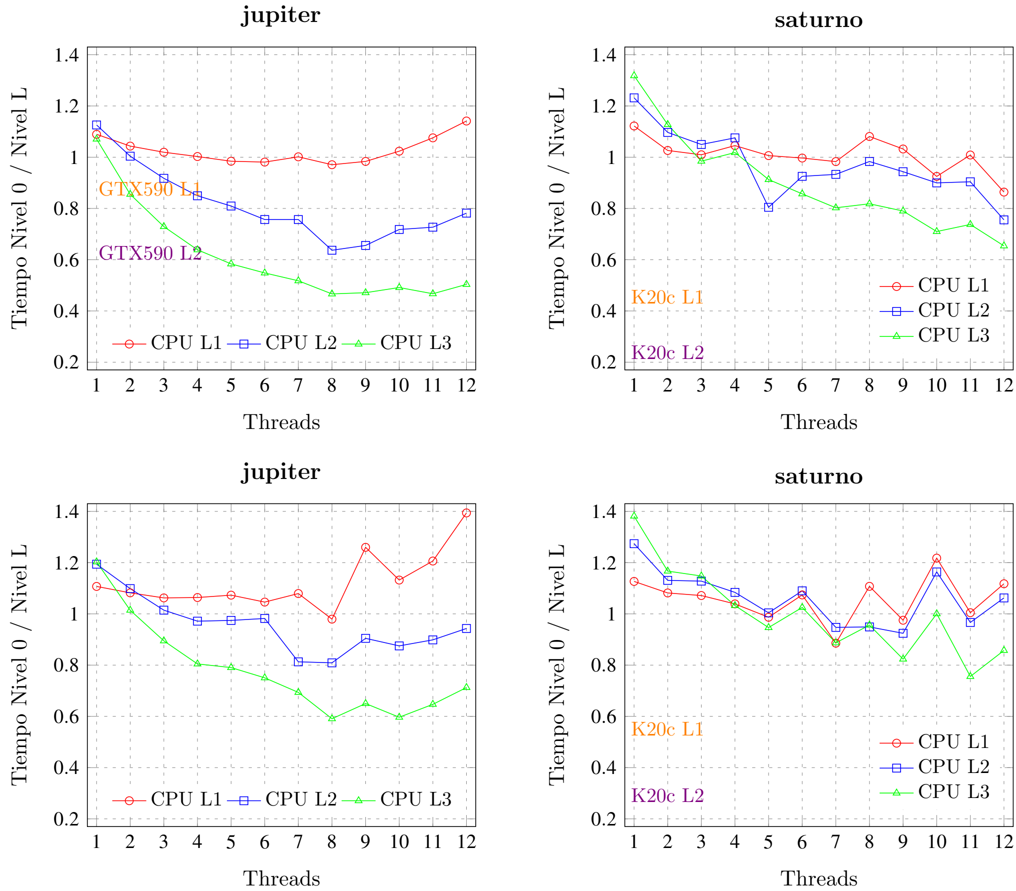

Figura 5.8: Cociente del tiempo de ejecución de la multiplicación directa con respecto al de la multiplicación de Strassen con niveles de recursión 1, 2 y 3, para matrices cuadradas de tamaño 8000 (arriba) y 12000 (abajo), en la CPU multicore de **jupiter** y **saturno**. Asimismo, se muestran algunos cocientes en GPUs de estos nodos.

5.2.1 Instalación de la multiplicación de matrices en un nodo

Teniendo en cuenta todo lo apuntado al comienzo de esta sección, la instalación de una rutina básica como la multiplicación de matrices (nivel `S0` software) a nivel de nodo (nivel `H1` hardware), `S0H1`, se podría llevar a cabo utilizando la información obtenida con los experimentos realizados en cada elemento computacional del nodo (nivel `H0`). De esta forma, se tendría una instalación `S0H1-s0h0`, donde la información del nivel inferior hardware puede ser utilizada de varias maneras.

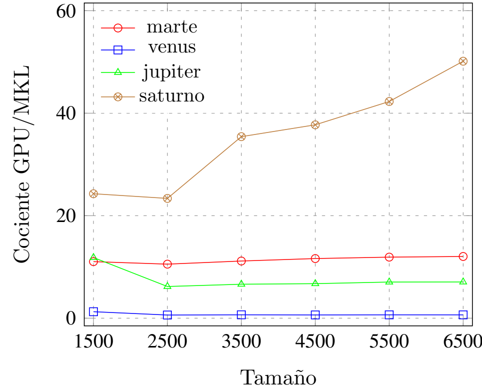

Figura 5.9: Evolución del cociente de las prestaciones de la multiplicación de matrices en GPU con respecto a un core de CPU, en cuatro nodos de **Heterosolar**, cuando varía el tamaño de las matrices.

Para la instalación en el nivel `H1`, se considera el siguiente CI-L1 = {1500, 2500, 3500, 4500, 5500, 6500, 7500}. Este conjunto de instalación puede ser distinto del que se utilizó para la instalación en el nivel `H0`, que es CI-L0 = {1000, 2000, 3000, 4000, 5000, 6000}. En el nivel `H1`, el parámetro algorítmico a determinar es el volumen de trabajo a asignar a cada elemento computacional del nodo, que a su vez viene determinado por el número de columnas de la matriz $\boldsymbol{B}$ que se asigna a cada elemento. Para cada tamaño de problema n_{H1}, el número de columnas de $\boldsymbol{B}$ a asignar a cada elemento viene dado por la velocidad relativa de los elementos de proceso cuando trabajan con el tamaño de CI-L0 más cercano a n_{H1}. Por ejemplo, en el CI-L1 considerado, si $n_{H1} = 1500$, entonces el tamaño de CI-L0 más cercano será $n_{L0} = 1000$. La figura 5.10 muestra, para los cuatro nodos de **Heterosolar**, el volumen de trabajo que se asigna a cada elemento computacional. La carga de trabajo es distinta en cada uno de los nodos, dada la diferencia tanto en el número de unidades computacionales como en su velocidad relativa.

Para tamaños grandes de problema, la carga tiende a ser proporcional a la capacidad computacional relativa de las diferentes unidades, pero no ocurre lo mismo para tamaños pequeños (especialmente en **saturno** y **venus**), que normalmente estarán más cercanos a los tamaños de los problemas a resolver cuando la rutina se utilice dentro de rutinas de mayor nivel, como es el caso de las multiplicaciones básicas en una multiplicación de Strassen o las multiplicaciones rectangulares que se utilizan para actualizar la matriz principal en factorizaciones

matriciales. En **jupiter**, las GPUs más potentes son las de tipo Tesla (GPU1 y GPU5). En **venus**, en cambio, la GPU es mucho más lenta que el resto de unidades de cómputo, por lo que el volumen de trabajo que se le asigna es mínimo.

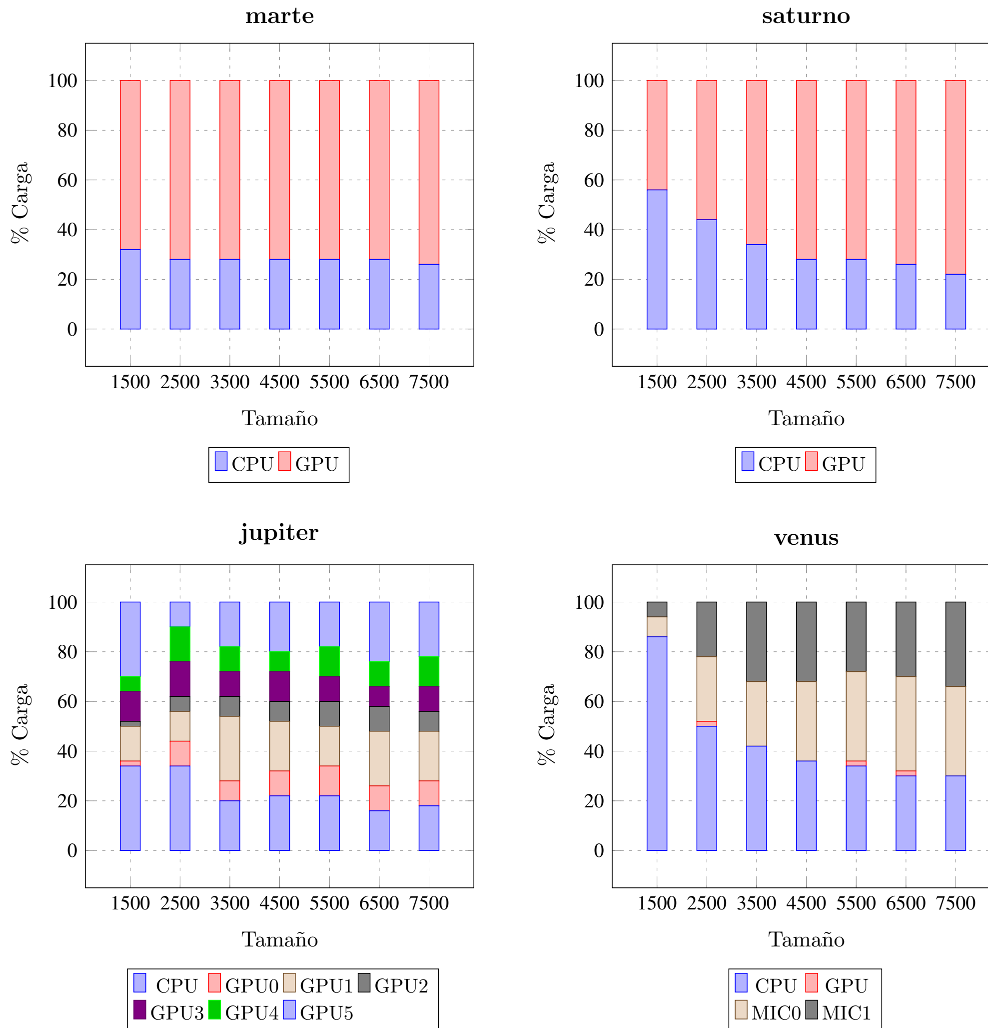

Figura 5.10: Porcentaje de carga asignado en la multiplicación de matrices a cada unidad básica de procesamiento de los cuatro nodos considerados de **Heterosolar**, teniendo en cuenta sólo la información obtenida de la instalación de nivel 0 de la multiplicación de matrices (`S0H1-s0h0`), variando el tamaño de las matrices.

Al usarse la información de nivel 0 para tomar las decisiones en el nivel 1, no se realizan experimentos para decidir el reparto de la carga, por tanto, el tiempo de instalación es muy reducido cuando el nodo contiene unas pocas unidades de cómputo. Cuando el número de unidades aumenta, el tiempo se incrementa

considerablemente, pues hay que obtener de entre todas las posibles distribuciones (o un subconjunto amplio de todas las posibles), la que teóricamente obtendría las mayores prestaciones. Por ejemplo, en **venus**, con cuatro unidades, el tiempo de instalación es de 0.28 segundos, mientras que en **jupiter**, con siete unidades, es de unos 126 segundos. Asimismo, en nodos con más de dos unidades de cómputo (como **venus** y **jupiter**), puede ser interesante analizar cómo varía el reparto de la carga al variar el número de unidades que se utilizan:

- En **jupiter** puede haber variaciones importantes dependiendo de las GPUs que se utilicen. La variación del porcentaje de la carga de trabajo asignada a la CPU y las GPUs se muestra en la figura 5.11. Al aumentar el tamaño de las matrices, la carga de trabajo que se asigna a la CPU disminuye ligeramente. Lo mismo sucede al incrementar el número de GPUs. Al usar varias GPUs, para tamaños grandes el reparto se estabiliza y las GPUs 1 y 5 reciben una cantidad de trabajo cercana al doble de la asignada a cada una de las restantes, lo que está en concordancia con la capacidad computacional relativa de las GPUs.

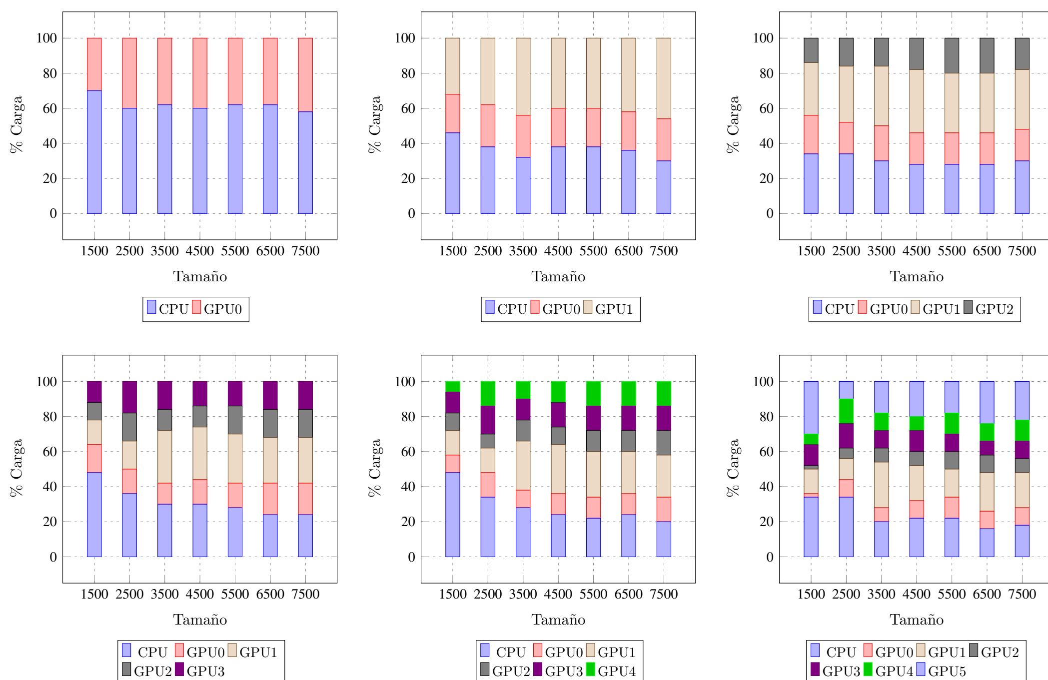

Figura 5.11: Porcentaje de reparto de la carga de trabajo entre CPU y GPU en **jupiter**, con ejecuciones variando el número de GPUs.

- En **venus**, que combina hasta tres coprocesadores de dos tipos con velocidades muy dispares, es comportamiento es ligeramente diferente. El porcentaje de la carga de trabajo asignada a la CPU y a los coprocesadores se muestra en la figura 5.12. Se asigna una parte muy pequeña (a veces ninguna) del trabajo a la GPU, debido a su poca capacidad computacional en comparación con la de la CPU y los coprocesadores Xeon Phi. Para tamaños pequeños de problema, se asigna más trabajo a la CPU que a los MIC, pero para tamaños mayores el trabajo se balancea, con una carga ligeramente mayor para MIC que para la CPU. Para tamaño 3500, se observa la mayor variación en el caso de un único MIC, y cuando se usan dos MIC, el cambio empieza a producirse en 2500.

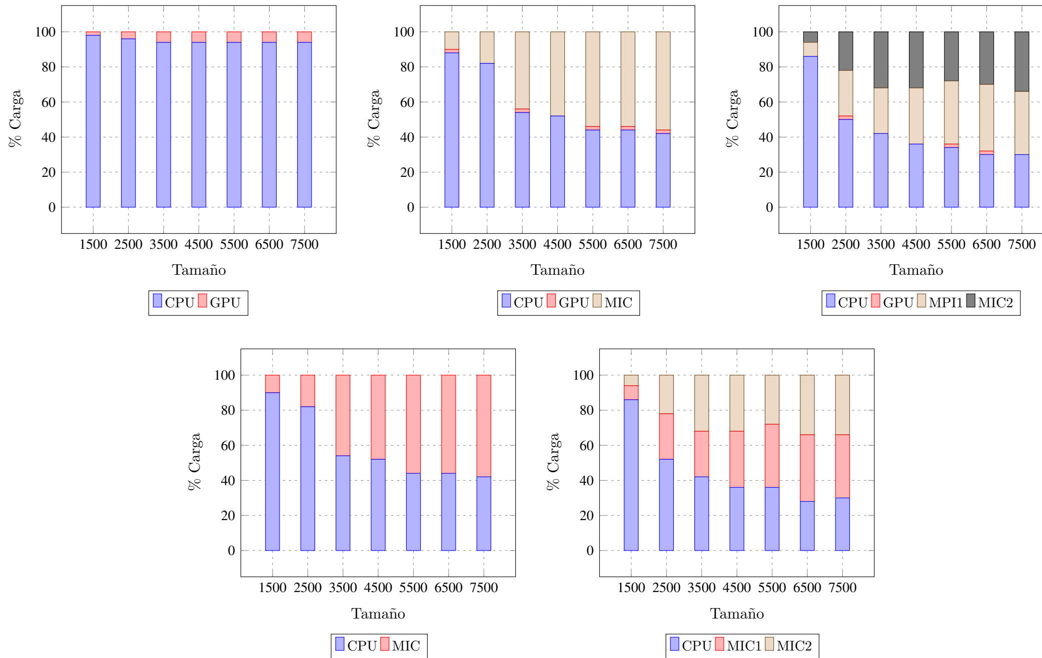

Figura 5.12: Porcentaje de reparto de la carga de trabajo entre la CPU y los coprocesadores (GPU, MIC) de **venus**, con ejecuciones variando el número y tipo de coprocesadores.

Tal como se ha comentado, tomar decisiones basándose sólo en las prestaciones computacionales, sin tener en cuenta las transferencias entre la CPU y los coprocesadores, puede dar lugar a la obtención de prestaciones teóricas alejadas de las experimentales, lo que puede llevar a una toma de decisiones errónea. La figura

5.13 compara las prestaciones estimadas y experimentales obtenidas en los cuatro nodos. Las estimaciones no son muy precisas, pero mejoran para problemas de mayor tamaño, donde el coste de las transferencias tiende a perder peso respecto al de computación. En los nodos con mayor número de coprocesadores, el coste de las transferencias (que no se ha tenido en cuenta para determinar la carga de trabajo a asignar a cada unidad computacional) es mayor, y las diferencia entre las prestaciones estimadas y experimentales aumenta.

El número de threads de CPU, cuya selección se delega a la instalación realizada en el nivel 0, puede verse afectado cuando la CPU trabaja en el nivel 1 junto con otras unidades computacionales, pues es necesario poner en marcha threads para lanzar trabajo a cada uno de los coprocesadores, lo que puede interferir con los threads de CPU que trabajan en la rutina de multiplicación de MKL. Esto puede llevar a pensar que es preferible realizar una reinstalación parcial a nivel 1 partiendo de la información obtenida de nivel 0. La tabla 5.7 muestra el número de threads de CPU seleccionado en **saturno** cuando se usa sólo CPU o CPU+GPU. En ambos casos, dicho valor es cercano al de cores físicos, pero usando sólo la CPU es ligeramente mayor (media de 22.7) que en el caso de usar también la GPU (media de 21.4). En cualquier caso, no hay una diferencia sustancial en el tiempo de ejecución por utilizar un thread más o menos. En **jupiter** y **venus**, que disponen de más unidades computacionales, la situación es más compleja:

n	CPU	CPU+GPU
1500	25	21
2500	20	22
3500	23	24
4500	23	21
5500	22	20
6500	22	21
7500	22	21

Tabla 5.7: Número de threads de CPU seleccionados en **saturno** con ejecución sólo en CPU y en CPU+GPU.

- La tabla 5.8 muestra el número de threads con el que se obtiene el menor tiempo de ejecución en **jupiter** cuando se usa sólo la CPU o cuando se usa añadiendo de forma incremental cada una de las GPUs de que dispone.

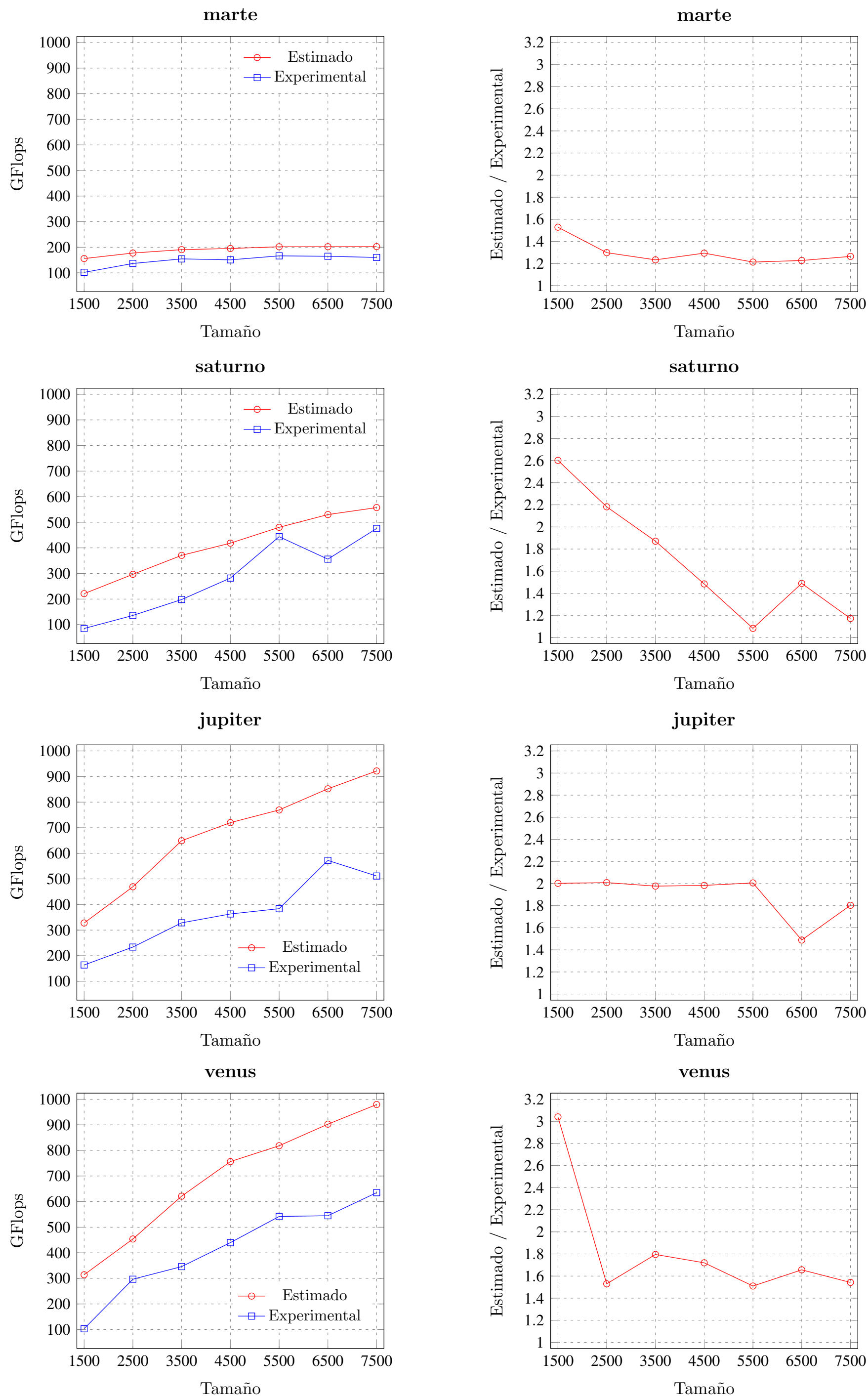

Figura 5.13: Comparación de las prestaciones estimadas y experimentales para la multiplicación de matrices en los cuatro nodos considerados de **Heterosolar**, usando todas las unidades de cómputo en cada nodo y variando el tamaño de problema. A la izquierda, prestaciones obtenidas. A la derecha, cociente de las prestaciones estimadas respecto a las experimentales.

En este caso no hay una tendencia clara en el número de threads seleccionado, cuyos valores se concentran cerca de 12 (entre 10 y 12) y de 24 (entre 21 y 24). Aunque el *hyperthreading* está habilitado, ya se ha visto que las prestaciones no aumentan al usar más threads que cores físicos y que a partir de 12 threads eran muy similares, de ahí que el valor se concentre en 12 y 24. En todo caso, parece que existe una ligera tendencia a usar un mayor número de threads cuando aumenta el tamaño de la matriz, y una ligera disminución al aumentar el número de GPUs. Por otro lado, los valores para los tamaños 6500 y 7500 coinciden, ya que en ambos casos se utiliza la información de instalación realizada con tamaño 6000.

	Número de GPUs						
n	0	1	2	3	4	5	6
1500	10	12	10	21	21	11	11
2500	22	22	11	11	10	10	12
3500	10	12	10	10	10	10	10
4500	22	23	12	22	22	12	23
5500	24	12	12	23	23	12	12
6500	24	12	12	12	12	12	12
7500	24	12	12	12	12	12	12

Tabla 5.8: Número de threads de CPU seleccionados en **jupiter** con ejecución sólo en CPU y en CPU+multiGPU (desde 1 hasta 6 GPUs).

- El número de threads de CPU seleccionado **en venus** cuando se combina de diferentes formas la CPU con los coprocesadores (GPU y MICs), se muestra en la tabla 5.9. No hay una tendencia clara en el número de threads seleccionado. Al igual que sucedía en **saturno** y **jupiter**, el valor seleccionado está cerca del número de cores, pero un poco más alejado en este caso. En cuanto a cada MIC, la tabla 5.10 muestra el número de threads seleccionado. El valor aumenta conforme se asigna más carga a los MIC, siendo normalmente 224 para los mayores tamaños de problema. Dado que un core (cuatro threads) de cada MIC se reserva para la gestión de las transferencias de datos entre CPU y MIC, para tamaños grandes se utilizan todos los recursos de cada MIC. Por otro lado, los valores con y sin GPU coinciden casi siempre, lo que se debe a la escasa cantidad de trabajo (a veces ninguno) que se asigna a la GPU, dado que su capacidad computacional es muy baja.

n	CPU	CPU MIC	CPU 2MIC	CPU GPU	CPU GPU MIC	CPU GPU 2MIC
500	9	9	10	9	10	10
2500	12	10	10	9	10	8
3500	8	8	9	10	8	9
4500	10	9	9	10	9	9
5500	10	9	9	10	9	9
6500	10	12	9	10	12	9
7500	10	8	9	10	8	9

Tabla 5.9: Número de threads de CPU seleccionados en **venus** para distintas configuraciones de CPU sin y con coprocesadores.

	CPU	CPU		CPU GPU	CPU GPU	
n	MIC	MIC1	MIC2	MIC	MIC1	MIC2
1500	112	112	92	112	112	92
2500	220	220	212	220	220	212
3500	224	216	224	220	216	224
4500	220	220	216	220	220	216
5500	220	224	224	220	224	224
6500	224	224	220	224	224	224
7500	224	224	220	224	224	220

Tabla 5.10: Número de threads seleccionados en cada MIC de **venus** para distintas configuraciones de CPU con y sin GPU con 1 o 2 MIC.

Una vez analizados los valores de los parámetros algorítmicos (en adelante, AP) seleccionados en el nivel 1 de la jerarquía para la rutina de multiplicación de matrices (`SOH1`), se van a comparar tres versiones de instalación de la rutina en dicho nivel. Los experimentos se centran en analizar cómo de lejos está el tiempo de ejecución con la selección de los parámetros algorítmicos realizada por el método de auto-optimización, con respecto a un "oráculo perfecto" que se obtendría al realizar experimentación exhaustiva para obtener el menor tiempo de ejecución. Los resultados se muestran para **jupiter** y **venus**, pues al ser los nodos más heterogéneos de **Heterosolar**, permiten llevar a cabo experimentos considerando diferentes combinaciones de GPUs y/o MICs. Dado que el rango de posibles valores para los AP es bastante amplio, se hace uso de un valor porcentual (10 % en este caso) que se aplica sobre el tamaño del problema para limitar los

posibles valores de distribución de la carga entre las unidades de cómputo. Las tablas 5.11 (**jupiter**) y 5.12 (**venus**) muestran los valores seleccionados para los AP y las prestaciones obtenidas en cada nodo, considerando CI-L0 = CI-L1 = {1000, 2000, . . . , 7000}, y para ejecuciones de la multiplicación híbrida en el nodo con el conjunto de tamaños intermedios {1500, 2500, . . . , 6500}. Se consideran tres versiones del proceso de auto-optimización:

- `AT-L0`: la instalación se realiza a nivel 0 (`S0H1-s0h0`), y en tiempo de ejecución, para un tamaño de problema n_R, se utiliza la información de nivel 0 referente a la capacidad computacional de la CPU y los coprocesadores, pero sin tener en cuenta transferencias. De esta forma, no se realiza instalación en el nivel 1.
- `AT-L10`: además de la instalación de nivel 0 de `AT-L0`, se hace instalación de nivel 1, pero sin ejecuciones para este nivel; sólo con un proceso de búsqueda de la mejor distribución para un CI-L1 usando la información de nivel 0. Tampoco se consideran tiempos de transferencia. En tiempo de ejecución, para un tamaño n_R, no es necesaria una búsqueda de la mejor distribución, sino que se usa la información de CI-L1 para el tamaño n'_R más cercano a n_R. De esta forma, la sobrecarga en tiempo de ejecución es menor que con `AT-L0`, pero las decisiones pueden ser peores, ya que se toman con información para tamaño n'_R, no para el tamaño que se está resolviendo, n_R.
- `AT-L11`: en la instalación a nivel 1 se ejecuta la rutina para los tamaños en CI-L1, por lo que el tiempo de instalación será mayor que con los métodos anteriores, pero las decisiones pueden ser mejores. En tiempo de ejecución funciona como `AT-L10`.

Algunas consideraciones sobre los resultados obtenidos con los tres métodos de instalación son:

- En general, los resultados muestran que cuando se usa la metodología de optimización jerárquica propuesta, las prestaciones están cercanas a las mejores obtenidas experimentalmente, especialmente para problemas de gran tamaño (cuando las técnicas de auto-optimización son más útiles). Estos resultados varían en nodos con diferentes características:

	valores de *AP* seleccionados con `AT-L0`							
n	(t_{CPU}, n_{CPU})	n_{GPU_0}	n_{GPU_1}	n_{GPU_2}	n_{GPU_3}	n_{GPU_4}	n_{GPU_5}	$GFlops$
1500	(22, 150)	150	300	150	150	150	450	208
2500	(19, 250)	250	500	250	250	250	750	312
3500	(11, 350)	350	700	350	350	350	1050	420
4500	(24, 450)	450	900	450	450	450	1350	435
5500	(22, 550)	550	1100	550	550	550	1650	541
6500	(13, 650)	650	1950	650	650	650	1300	572

	valores de *AP* seleccionados con `AT-L10`							
n	(t_{CPU}, n_{CPU})	n_{GPU_0}	n_{GPU_1}	n_{GPU_2}	n_{GPU_3}	n_{GPU_4}	n_{GPU_5}	$GFlops$
1500	(22, 150)	150	300	150	150	150	450	208
2500	(19, 250)	250	500	250	250	250	750	312
3500	(11, 350)	350	700	350	350	350	1050	420
4500	(24, 450)	450	900	450	450	450	1350	435
5500	(11, 550)	550	1100	550	550	550	1650	529
6500	(12, 650)	650	1950	650	650	650	1300	584

	valores de *AP* seleccionados con `AT-L11`							
n	(t_{CPU}, n_{CPU})	n_{GPU_0}	n_{GPU_1}	n_{GPU_2}	n_{GPU_3}	n_{GPU_4}	n_{GPU_5}	$GFlops$
1500	(0, 0)	0	450	300	300	0	450	275
2500	(24, 500)	0	750	250	250	0	750	321
3500	(0, 0)	350	1050	350	350	350	1050	481
4500	(0, 0)	450	1350	450	450	450	1350	567
5500	(0, 0)	550	1650	550	550	550	1650	603
6500	(0, 0)	650	1950	650	650	650	1950	685

	valores de *AP* seleccionado con un oráculo perfecto							
n	(t_{CPU}, n_{CPU})	n_{GPU_0}	n_{GPU_1}	n_{GPU_2}	n_{GPU_3}	n_{GPU_4}	n_{GPU_5}	$GFlops$
1500	(8, 300)	0	300	0	300	300	300	277
2500	(0, 0)	250	500	0	500	500	750	413
3500	(0, 0)	350	1050	350	700	0	1050	511
4500	(10, 900)	450	900	450	0	450	1350	570
5500	(0, 0)	550	1650	550	550	550	1650	603
6500	(0, 0)	650	1950	650	650	650	1950	685

Tabla 5.11: Comparación de los valores de los parámetros algorítmicos seleccionados para la multiplicación matricial híbrida con distintos métodos de optimización y prestaciones obtenidas, en **jupiter**.

n	valores de AP seleccionados con `AT-L0`				$GFlops$
	(t_{CPU}, n_{CPU})	n_{GPU}	(t_{MIC_0}, n_{MIC_0})	(t_{MIC_1}, n_{MIC_1})	
1500	(12, 450)	0	(112, 600)	(108, 450)	134
2500	(11, 500)	0	(216, 1000)	(208, 1000)	273
3500	(11, 700)	0	(216, 1400)	(216, 1400)	357
4500	(11, 900)	0	(224, 1800)	(220, 1800)	436
5500	(11, 1100)	0	(220, 2200)	(212, 2200)	510
6500	(11, 1300)	0	(224, 2600)	(224, 2600)	729

n	valores de AP seleccionados con `AT-L10`				$GFlops$
	(t_{CPU}, n_{CPU})	n_{GPU}	(t_{MIC_0}, n_{MIC_0})	(t_{MIC_1}, n_{MIC_1})	
1500	(12, 450)	0	(112, 600)	(112, 450)	145
2500	(11, 500)	0	(208, 1000)	(224, 1000)	283
3500	(11, 700)	0	(220, 1400)	(224, 1400)	374
4500	(9, 900)	0	(224, 1800)	(224, 1800)	421
5500	(11, 1100)	0	(224, 2200)	(220, 2200)	531
6500	(11, 1300)	0	(224, 2600)	(220, 2600)	717

n	valores de AP seleccionados con `AT-L11`				$GFlops$
	(t_{CPU}, n_{CPU})	n_{GPU}	(t_{MIC_0}, n_{MIC_0})	(t_{MIC_1}, n_{MIC_1})	
1500	(12, 300)	0	(112, 600)	(208, 600)	167
2500	(11, 500)	0	(208, 1000)	(224, 1000)	283
3500	(11, 700)	0	(220, 1400)	(224, 1400)	374
4500	(9, 900)	0	(224, 1800)	(224, 1800)	421
5500	(11, 1100)	0	(224, 2200)	(220, 2200)	531
6500	(11, 1300)	0	(224, 2600)	(220, 2600)	717

n	valores de AP seleccionados con un oráculo perfecto				$GFlops$
	(t_{CPU}, n_{CPU})	n_{GPU}	(t_{MIC_0}, n_{MIC_0})	(t_{MIC_1}, n_{MIC_1})	
1500	(12, 300)	0	(112, 900)	(108, 300)	227
2500	(11, 500)	0	(108, 500)	(216, 1500)	328
3500	(9, 700)	0	(216, 1750)	(216, 1050)	460
4500	(7, 900)	0	(224, 1800)	(220, 1800)	527
5500	(11, 1100)	0	(224, 2200)	(224, 2200)	600
6500	(11, 1300)	0	(224, 2600)	(224, 2600)	729

Tabla 5.12: Comparación de los valores de los parámetros algorítmicos seleccionados para la multiplicación matricial híbrida con distintos métodos de optimización y prestaciones obtenidas, en **venus**.

   - En **jupiter**, el conjunto de GPUs tiene mucha más capacidad computacional que la CPU, que puede llegar a no recibir trabajo. El reparto es heterogéneo también entre GPUs, siendo las C2075 (número 1 y 5) más rápidas que las GTX590.
   - En **venus**, la GPU es con diferencia la unidad más lenta, por lo que no recibe trabajo, y la CPU recibe aproximadamente la mitad del trabajo que cada MIC.

- En cuanto al tiempo de instalación y de toma de decisiones, por ejemplo, en **venus** fue de alrededor de 1 hora a nivel 0 con `AT-L0`; de 6 segundos a nivel 1 con `AT-L10` y de unos 40 minutos con `AT-L11`. El tiempo dedicado a la toma de decisión es despreciable, siendo de unos 200 milisegundos con `AT-L0`, y de unos 20 microsegundos con `AT-L11` o `AT-L10`.

Asimismo, una instalación con experimentos en donde se incluya el coste de las transferencias de datos, puede proporcionar predicciones más precisas y, consecuentemente, tomar mejores decisiones. Una posibilidad es reutilizar la información de instalación de nivel 0, pero incluyendo medidas del coste de las transferencias entre la CPU y los coprocesadores, y utilizar modelos del tiempo de ejecución que incluyan computación y transferencias [24]. El tiempo de ejecución sería, entonces, el máximo de los tiempos de computación más los de transferencias, para todas la unidades computacionales, es decir:

$$\min_{N_i, 1 \leq i \leq c+1 / \sum_{i=1}^{c+1} N_i = N} \{ \max_{1 \leq i \leq c+1} \{ T_{comp,i}(N_i, < t_i >) + T_{tran,i}(N_i) \} \} \tag{5.1}$$

La figura 5.14 compara en **jupiter** y **venus** la predicción realizada teniendo en cuenta las transferencias y sin considerarlas. La instalación a nivel 0 se ha realizado con tamaños de problema $\{1000, 2000, \ldots, 6000\}$ y las matrices rectangulares correspondientes que se generan. La comparación se hace con matrices cuadradas de tamaños 1500, 3500 y 5500. Cuando no se tienen en cuenta las transferencias, el número de columnas de la matriz B que se asignan a cada unidad de cómputo se obtiene a partir de la ecuación 5.1 con $T_{tran,c+1} = 0$. Para tener en cuenta las transferencias, los GFlops para cada entrada del CI se obtienen realizando ejecuciones para cada tamaño y distribución de la carga de trabajo,

de forma que se incluyen los costes $T_{tran,c+1}$. La figura muestra, para cada uno de los tamaños, el cociente del tiempo de ejecución con respecto al mínimo obtenido experimentalmente de forma exhaustiva variando la distribución de la carga. Las decisiones son mejores (el cociente está más cercano a 1) cuando se consideran las transferencias, y mejoran al aumentar el tamaño del problema.

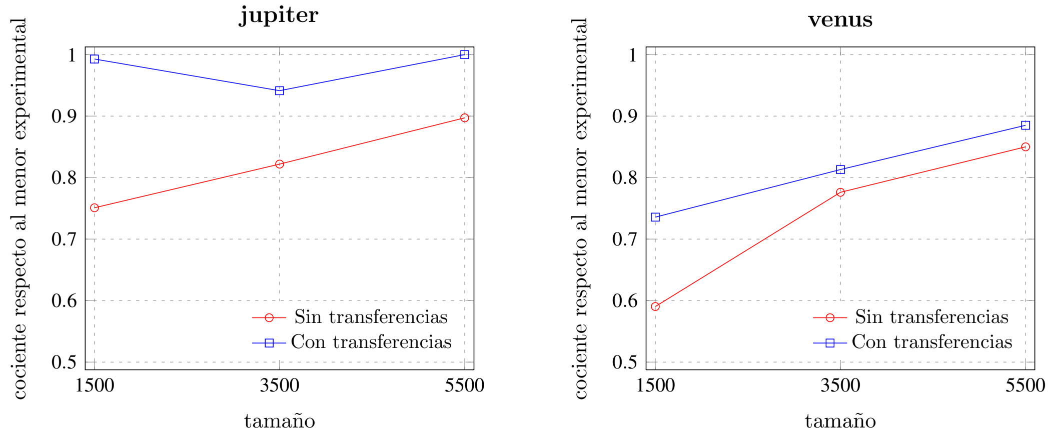

Figura 5.14: Cociente del menor tiempo experimental obtenido para la multiplicación híbrida en los nodos **jupiter** y **venus** con respecto a los tiempos que se obtienen al determinar la carga de trabajo teniendo en cuenta el coste de las transferencias entre la CPU y los coprocesadores, y sin tenerlas en cuenta.

El tiempo de decisión empleado para seleccionar el valor de los parámetros a utilizar en la resolución del problema, debe suponer una sobrecarga mínima. Así, si se considera la ecuación 5.1 con los tiempos de transferencia a cero (sólo se tiene en cuenta la velocidad relativa de las unidades computacionales para decidir el reparto), habría que sustituir los posibles repartos en la ecuación y seleccionar el que proporcione menor tiempo teórico. La tabla 5.13 muestra el tiempo de decisión en **jupiter** para los distintos tamaños de problema al variar el número de GPUs. El tiempo de la toma de decisión es asumible para pocos elementos de proceso, pero aumenta de forma exponencial al aumentar el número de elementos. Esos tiempos no son asumibles en tiempo de ejecución, por lo que habría que reducirlos. También hay un aumento (mucho menor) del tiempo de decisión al aumentar el tamaño de las matrices. En consecuencia, se hace necesario incluir la toma de decisiones en la instalación de nivel 1, guardando la información de los parámetros para los tamaños seleccionados en CI-L1, y en tiempo de ejecución tomar las decisiones usando la información generada en la instalación de nivel 1.

La tabla 5.14 muestra el tiempo de decisión en **venus** para los distintos tamaños de problema al variar el número y el tipo de coprocesadores. El tiempo de la toma de decisión con cuatro elementos de proceso (CPU+GPU+2MIC) es ligeramente superior que el obtenido en **jupiter** cuando se usan 4 elementos (CPU+3GPU), dado que para cada MIC se ha de obtener el número de threads a usar.

	Número de GPUs						
n	0	1	2	3	4	5	6
1500	0.0000	0.0001	0.0011	0.0164	0.2057	1.7851	15.3827
2500	0.0000	0.0002	0.0010	0.0136	0.1754	1.8473	16.5568
3500	0.0001	0.0001	0.0010	0.0141	0.1824	1.9186	17.1296
4500	0.0001	0.0002	0.0012	0.0145	0.1873	1.9668	17.5820
5500	0.0002	0.0002	0.0011	0.0149	0.1902	2.0048	17.9140
6500	0.0002	0.0002	0.0013	0.0168	0.2146	2.2415	19.7942
7500	0.0002	0.0002	0.0015	0.0189	0.2398	2.4964	21.9852

Tabla 5.13: Tiempo de decisión (en segundos) obtenido en **jupiter** para distintos tamaños de problema, variando el número de GPUs.

	CPU	CPU	CPU	CPU	CPU	CPU
		GPU		GPU		GPU
n			MIC	MIC	2MIC	CPU+2MIC
1500	0.0001	0.0001	0.0003	0.00017	0.0030	0.0370
2500	0.0001	0.0001	0.0003	0.00016	0.0030	0.0391
3500	0.0000	0.0001	0.0003	0.00014	0.0031	0.0324
4500	0.0001	0.0002	0.0007	0.00022	0.0026	0.0333
5500	0.0001	0.0001	0.0003	0.00018	0.0028	0.0340
6500	0.0002	0.0002	0.0009	0.00025	0.0045	0.0396
7500	0.0001	0.0001	0.0005	0.00022	0.0034	0.0464

Tabla 5.14: Tiempo de decisión (en segundos) obtenido en **venus** para distintos tamaños de problema, variando el número y tipo de coprocesadores (GPU, MIC).

Por último, también sería posible utilizar la información de nivel 0 para establecer un punto de partida a partir del cual llevar a cabo una búsqueda local que realizara experimentos para cada distribución a estudiar. De esta forma, en la instalación de nivel 1 se tendrían en cuenta las transferencias, pero el tiempo de instalación sería menor que con una instalación exhaustiva, ya que la búsqueda local evitaría ejecuciones y se centraría sólo en algunas con tiempo de ejecución reducido.

5.2.2 Instalación de la multiplicación de Strassen en un nodo

Si consideramos la multiplicación de Strassen en un nodo, estaríamos ante una configuración software y hardware de nivel 1 (`S1H1`). La multiplicación de Strassen puede ejecutar sucesivamente las multiplicaciones básicas usando el nodo completo, con lo que el único parámetro a determinar es el nivel de recursión. La distribución de trabajo entre los elementos computacionales encargados de realizar cada multiplicación básica, se determina al usar la información de instalación previamente almacenada para la multiplicación de matrices optimizada para el nodo, dando lugar a una instalación de tipo `S1H1-s0h1`. También es posible ejecutar cada multiplicación básica en una única unidad computacional, con lo que se tendría una instalación de tipo `S1H1-s0h0`. En este caso, aparecen otros parámetros algorítmicos de la rutina de nivel 1, que son el número de multiplicaciones básicas a asignar a cada elemento computacional, lo que puede hacerse usando la información de la rutina de nivel 0 en las unidades computacionales de nivel 0; o se puede realizar una asignación dinámica, con asignación de trabajo inicial en función de la capacidad de cómputo. En cambio, si las multiplicaciones básicas se sustituyen por multiplicaciones de Strassen en unidades computacionales, estaríamos ante el caso `S1H1-s1h0`.

Tal como se ha visto en la sección anterior, la elevada capacidad computacional de los sistemas de cómputo actuales junto al uso de librerías optimizadas de álgebra lineal (MKL y cuBLAS), hacen que la multiplicación de Strassen supere a la multiplicación tradicional sólo para tamaños de problema muy grandes. En un experimento inicial, la tabla 5.15 compara el tiempo de ejecución y la selección de parámetros que realiza la metodología de auto-optimización jerárquica con matrices cuadradas de tamaño 3500 y 7000, para la multiplicación tradicional y la de Strassen con un nivel de recursión. Los experimentos se han llevado a cabo en **jupiter**, considerando tres plataformas computacionales: sólo CPU, CPU+C2075 y CPU+C2075+GTX590. Algunas observaciones:

- En general, la versión de la multiplicación de Strassen con un nivel de recursión no mejora al método directo. Esto es debido a que el método directo tiene un coste de $2n^3$ flops y el de Strassen $\frac{7}{4}n^3 + \frac{9}{2}n^2$ flops, lo que significa

una mejora máxima del 12.5 %, que se ve reducida por la no paralelización y optimización de las rutinas de tipo BLAS de nivel 2 en la multiplicación de Strassen y por su mayor número de accesos a memoria.

- La versión híbrida CPU+GPU proporciona buenas prestaciones sólo cuando se usan pocas GPUs. De esta forma, se podría pensar en la utilización de nodos virtuales, cada uno formado por un subconjunto de cores de la CPU y una o varias GPUs. En nuestro experimento, las unidades computacionales de nivel 1 se componen de una (CPU), dos (CPU+C2075) o tres (CPU+C2075+GTX590) unidades de nivel 0.
- Los valores de los parámetros seleccionados con la metodología de auto-optimización son lógicos de acuerdo con los comentarios realizados en la sección 5.1, con más carga para la GPU Tesla, y con una carga similar para la CPU y la GPU de tipo GForce.

La tabla 5.16 muestra los parámetros seleccionados para un conjunto de tamaños de un CI. Se muestran resultados en **jupiter**, pero sin usar el nodo completo, ya que el uso de muchas GPUs degrada las prestaciones debido al coste de las transferencias. Las combinaciones de CPU y GPU que se han considerado son: CPU+C2075, CPU+GTX590, C2075+GTX590, CPU+C2075+GTX590 y CPU+2C2075, siendo esta última la configuración con la que se obtienen siempre los menores tiempos debido a que ofrece la mayor capacidad computacional. Para tamaños pequeños, la metodología de auto-optimización selecciona el método directo, pero debido a las limitaciones de memoria de las GPUs, el método de Strassen con un nivel de recursión es el más adecuado para tamaños mayores, siendo preferible para tamaños muy grandes utilizar un segundo nivel de recursión. La carga de trabajo del método de Strassen corresponde a la de la multiplicación básica: la mitad de la matriz o un cuarto de la misma para niveles 1 y 2, respectivamente. Los valores de esos parámetros se seleccionan dentro de la multiplicación de matrices a nivel de nodo (`S1H1-s0h1`).

Asimismo, se han realizado experimentos en **venus** haciendo uso de la CPU multicore (que ofrece gran capacidad de cómputo) y los dos coprocesadores Xeon Phi de que dispone. No se usa la GPU por su reducida capacidad computacional.

	3500 × 3500				
	tiempo	threads	CPU		
directo	1.55	9	3500		
Strassen	1.86	10	3500		
	tiempo	threads	CPU	C2075	
directo	0.56	10	1190	2310	
Strassen	0.57	12	630	1120	
	tiempo	threads	CPU	C2075	GTX590
directo	0.60	12	910	1750	840
Strassen	0.53	9	350	945	455
	7000 × 7000				
	tiempo	threads	CPU		
directo	4.41	12	7000		
Strassen	9.27	9	7000		
	tiempo	threads	CPU	C2075	
directo	3.88	12	2940	4060	
Strassen	3.25	10	1190	2310	
	tiempo	threads	CPU	C2075	GTX590
directo	1.86	12	1820	3500	1680
Strassen	2.44	12	910	1750	840

Tabla 5.15: Comparación del tiempo de ejecución (en segundos) y valores seleccionados para los parámetros, aplicando la metodología de auto-optimización con la multiplicación de matrices tradicional y con Strassen (con 1 nivel de recursión), para matrices cuadradas de tamaño 3500 y 7000 usando tres configuraciones diferentes en **jupiter**: sólo CPU, CPU+C2075, CPU+C2075+GTX590.

La figura 5.15 muestra el cociente del tiempo de ejecución del método directo (implementación de la multiplicación de MKL en CPU y Xeon Phi) con respecto a la multiplicación de Strassen con 1 y 2 niveles de recursión. A diferencia de **jupiter** (usando las GPUs), para matrices pequeñas el mejor método es Strassen con un nivel de recursión. La decisión del trabajo a asignar a cada unidad computacional también se hace ahora dentro de la multiplicación directa híbrida (`S1H1-s0h1`).

n	Método	Combinación de UC	CPU threads	**Distribución del Trabajo** CPU	1ª C2075	2ª C2075
2000	Direct	CPU+2 C2075	7	240	880	880
3000	Direct	CPU+2 C2075	11	600	1200	1200
4000	Direct	CPU+2 C2075	12	880	1520	1600
5000	Direct	CPU+2 C2075	10	1000	2000	2000
6000	StrassenL1	CPU+2 C2075	11	600	1200	1200
7000	StrassenL1	CPU+2 C2075	11	630	1470	1400
8000	StrassenL1	CPU+2 C2075	12	880	1520	1600
9000	StrassenL1	CPU+2 C2075	11	900	1800	1800
10000	StrassenL1	CPU+2 C2075	10	1000	2000	2000
11000	StrassenL1	CPU+2 C2075	12	1320	2090	2090
12000	StrassenL2	CPU+2 C2075	11	600	1200	1200

Tabla 5.16: Método con el que se obtiene el menor tiempo de ejecución en **jupiter** para distintos tamaños de matriz, unidades de cómputo utilizadas y valores seleccionados para los parámetros en la multiplicación matricial básica.

La figura 5.16 muestra la evolución de la carga de trabajo asignada a los coprocesadores Intel Xeon Phi cuando varía el tamaño de problema. Cuando se usan los dos Xeon Phi, se asigna menos trabajo a la CPU y el trabajo asignado a cada uno de ellos aumenta con el tamaño de la matriz. El comportamiento en este nodo es distinto que cuando se usan GPUs (figura 5.7), lo que confirma la necesidad de utilizar un método que permita seleccionar de forma automática tanto el mejor método (directo o Strassen y su nivel de recursión) como la mejor combinación de unidades computacionales.

5.2.3 Utilización de la multiplicación de matrices híbrida en una factorización LU

Esta sección ilustra los dos primeros métodos mencionados para la multiplicación de Strassen, para la auto-optimización de una rutina de nivel 1 (la factorización LU por bloques) en un nodo heterogéneo (`S1H1`).

En la rutina de factorización LU por bloques, el mayor coste computacional se encuentra en la multiplicación de matrices rectangulares que se lleva a cabo para actualizar las matrices cuadradas con las que se trabaja en cada paso de la factorización. Por ejemplo, dada una matriz de tamaño 10000, si se consideran

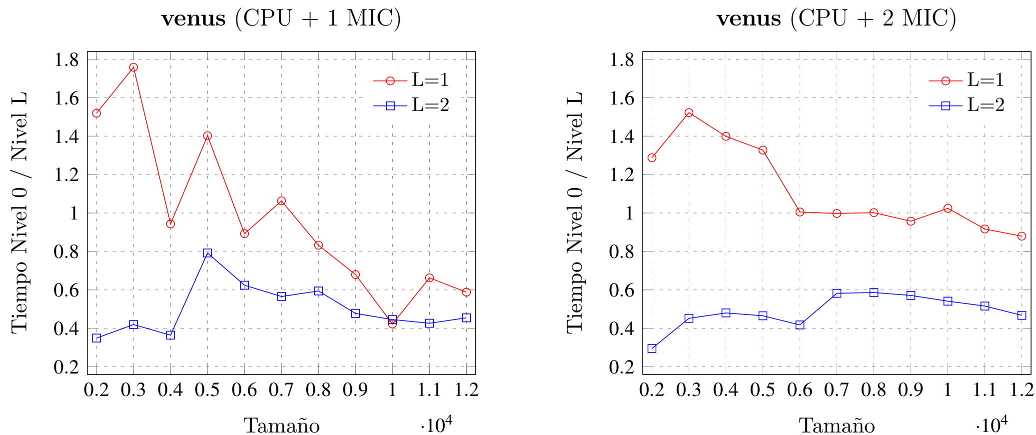

Figura 5.15: Cociente del tiempo de ejecución de la rutina de multiplicación de matrices de MKL con respecto al de la multiplicación de Strassen con niveles de recursión 1 y 2, en **venus**, con dos coprocesadores Intel Xeon Phi.

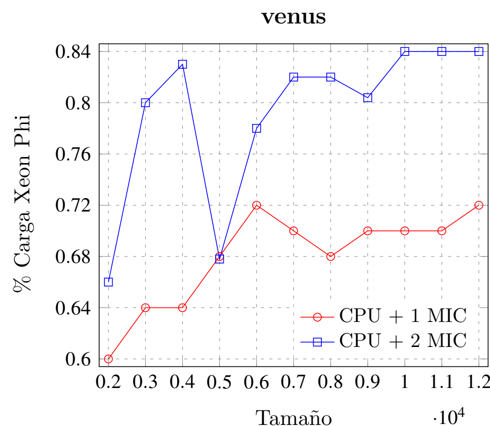

Figura 5.16: Porcentaje de trabajo asignado a los coprocesadores Intel Xeon Phi en las multiplicaciones básicas de la multiplicación de Strassen cuando se realizan ejecuciones en **venus** combinando la CPU con uno o dos coprocesadores.

bloques de tamaño 1000, la multiplicación de matrices se invoca nueve veces, con dimensiones $(s, 1000, s)$ y $s = 9000, 8000, \ldots, 1000$. En el estudio realizado, se considera una versión de la factorización LU en la que las únicas operaciones que usan el sistema completo (CPU+multiGPU) son las multiplicaciones de matrices, mientras que las otras operaciones básicas (factorizaciones en los bloques y resolución de sistemas de ecuaciones lineales), se llevan a cabo en la CPU multicore.

La tabla 5.17 muestra la distribución del trabajo y los GFlops obtenidos en **jupiter** en las nueve multiplicaciones matriciales que se realizan dentro de la factorización. La distribución del trabajo muestra el número de columnas de la matriz $\boldsymbol{B}$ asignadas a la CPU y a cada una de las GPUs. Se consideran las seis GPUs del nodo y la carga de trabajo se distribuye siguiendo el orden en que están numeradas, siendo la primera GPU una GTX590, la segunda una C2075, la tercera otra GTX590, etc. Debido al alto coste de las transferencias y a que el tamaño de las matrices rectangulares a multiplicar no es muy grande, no llegan a usarse más de tres GPUs. Se asigna más trabajo a la GPU Tesla C2075, y el asignado a la CPU y a las GeForce GTX590 es similar. Cuando el tamaño de las matrices disminuye, lo mismo pasa con las prestaciones y el número de GPUs que se usa; para tamaños 3000 y 2000 intervienen dos GPUs en la multiplicación, y para tamaño 1000 sólo trabaja una. En este caso, la optimización se realiza sobre una factorización LU con tamaño de bloque fijo, delegando a la multiplicación híbrida la selección del valor de los parámetros algorítmicos, que a nivel de nodo (nivel 1 de la jerarquía hardware, `H1`) es la distribución del trabajo y a nivel de elementos de proceso (nivel 0, `H0`) el número de threads MKL en CPU. De esta forma, el proceso de auto-optimización de la metodología sería de tipo `S1H1-s0h1` sin selección de parámetros en el nivel 1 software.

$n \times 1000 \times n$	**Distribución de la carga**	
n	CPU, GPU0, GPU1, GPU2	GFlops
9000	1980,1800,3420,1800	236
8000	1920,1600,3040,1440	377
7000	1540,1400,2660,1400	359
6000	1560,960,2280,1200	340
5000	1100,1000,1900,1000	320
4000	800,800,1600,800	292
3000	780,720,1500,0	273
2000	440,520,1040,0	211
1000	480,0,520,0	134

Tabla 5.17: Distribución de la carga entre diferentes unidades de cómputo de **jupiter** y prestaciones obtenidas en las sucesivas multiplicaciones de matrices que se llevan a cabo dentro de la rutina de factorización LU por bloques, para una matriz de tamaño 10000 y tamaño de bloque 1000.

Si se decide incluir la selección de algunos parámetros en el nivel 1 software, habría que seleccionar el tamaño de bloque de la factorización, el número de GPUs y la carga de trabajo a asignar a cada unidad computacional. Los dos últimos parámetros se delegan a la multiplicación de matrices híbrida a nivel de nodo, como en el caso anterior, por tanto, el único parámetro a determinar en la rutina de nivel 1 sería el tamaño de bloque, que se seleccionaría o bien cuando se instale la rutina LU o en tiempo de ejecución. En ambos casos, la información generada durante la instalación de las rutinas básicas de la factorización por bloques en el nodo o en las unidades computacionales básicas, se consultaría para estimar el tiempo de ejecución para diferentes tamaños de bloque, seleccionando aquel con el que se obtenga un menor tiempo de ejecución. La tabla 5.18 muestra una simulación del tiempo de ejecución de la factorización LU en un sistema CPU+multiGPU para una matriz de tamaño 10000, considerando diferentes costes de las implementaciones de la factorización LU y de la resolución de los sistemas de ecuaciones en CPU. Se resalta el menor tiempo de ejecución para cada una de las prestaciones consideradas. Cuando la CPU es lenta, se seleccionan bloques de tamaño 1500, pero cuando es más rápida el tamaño seleccionado es 2500.

	Tamaño de Bloque			
GFlops (en CPU)	1000	1500	2000	2500
50	2.094	**1.783**	2.056	2.299
100	1.994	**1.558**	1.522	1.674
150	1.960	1.482	1.522	**1.465**
200	1.943	1.445	1.455	**1.361**

Tabla 5.18: Simulación del tiempo de ejecución (en segundos) de una factorización LU por bloques en un sistema CPU+multiGPU para una matriz de tamaño 10000, considerando diferentes prestaciones de las operaciones básicas en CPU para distintos tamaños de bloque.

5.3 Instalación en nivel 2 de hardware

La extensión de la jerarquía a un segundo nivel hardware se puede llevar a cabo de dos formas: considerando el cluster como una agrupación de nodos o considerando subnodos dentro de un nodo como elementos hardware de nivel 1, lo que daría lugar a que el propio nodo fuese visto como un elemento de nivel 2.

5.3.1 Multiplicación de matrices en un cluster

El cluster completo constituye una unidad computacional de nivel 2 formada por unidades de nivel 1, los nodos. Una multiplicación de matrices básica que distribuye las columnas de la matriz B entre los nodos es una rutina de nivel 0 sobre un hardware de nivel 2 (`S0H2`). La tabla 5.19 muestra la distribución del trabajo en un cluster formado por cuatro nodos de **Heterosolar** conectados a través de una red Gigabit Ethernet. Se compara la instalación a nivel 2 cuando se usan dos métodos de instalación a nivel 1 (`AT-L10` y `AT-L11`). Se muestra también la decisión con un "oráculo perfecto" obtenido con experimentación exhaustiva, así como la desviación de las prestaciones respecto a las obtenidas con dicho oráculo. En general, las dos técnicas de instalación proporcionan buenas prestaciones, con desviaciones alrededor del 10 % para los tamaños de problema mayores. Además, el tiempo de instalación en este nivel 2 es muy bajo (menor a un minuto) debido al uso de la información de nivel 1 para la toma de decisiones.

5.3.2 Multiplicación de Strassen usando nodos virtuales

El uso de unidades computacionales virtuales permite trabajar con diferentes configuraciones hardware de la plataforma. De esta forma, se puede explotar el paralelismo OpenMP y MKL de manera simultánea mediante el uso de paralelismo anidado, con threads OpenMP en un primer nivel, y cada thread llamando a rutinas BLAS, cuBLAS o híbridas. En un experimento con la multiplicación de Strassen con un nivel de recursión, las siete multiplicaciones de tamaño $n/2$ en el algoritmo 1 se ejecutan en paralelo con un cierto número de threads OpenMP, y las multiplicaciones básicas se realizan con la multiplicación directa con auto-optimización sobre una unidad virtual. Para que no interfiera el paralelismo OpenMP con el de MKL, hay que habilitar el paralelismo anidado (`omp_set_nested(1)`) y deshabilitar la ejecución dinámica en MKL (`mkl_set_dynamic(0)`) [34]. La tabla 5.20 compara el tiempo de ejecución con y sin paralelismo anidado, para diferentes configuraciones del número de threads OpenMP y MKL, con y sin GPU (sólo se utiliza una de las Tesla C2075 en algunos de los casos). Las principales conclusiones son:

n	Distribución del Trabajo con `AT-L21` (con `AT-L10`)				% Desviación
	marte	**saturno**	**jupiter**	**venus**	de Prestaciones
1750	350	350	700	350	4
2250	450	450	900	450	8
2750	550	825	1375	0	7
3250	650	975	1625	0	3
3750	750	1125	1500	375	2
4250	850	1275	1700	425	9
4750	950	1425	1900	475	10
5250	1050	1575	2100	525	10
5750	575	1725	2300	1150	10

n	Distribución del Trabajo con `AT-L21` (con `AT-L11`)				% Desviación
	marte	**saturno**	**jupiter**	**venus**	de Prestaciones
1750	350	350	525	525	5
2250	450	450	675	675	9
2750	275	825	825	825	9
3250	325	975	975	975	8
3750	375	1125	1125	1125	12
4250	425	1275	1275	1275	11
4750	475	1425	1425	1425	13
5250	525	1575	1575	1575	10
5750	575	1725	1725	1725	10

n	Distribución del Trabajo con un Oráculo Perfecto				
	marte	**saturno**	**jupiter**	**venus**	
1750	350	175	350	875	
2250	675	0	0	1575	
2750	275	550	550	1375	
3250	325	650	975	1300	
3750	375	750	750	1875	
4250	425	850	1275	1700	
4750	475	950	950	2375	
5250	525	1050	1575	2100	
5750	575	1150	1725	2300	

Tabla 5.19: Distribución de la carga de trabajo de la multiplicación de matrices entre cuatro nodos de cómputo del cluster **Heterosolar** cuando se instala en el nivel 2 de hardware usando los métodos `AT-L10` y `AT-L11` a nivel 1, y desviación en las prestaciones respecto a las obtenidas con un oráculo perfecto.

- La agrupación de unidades de cómputo en nodos virtuales permite combinar varios elementos de proceso dentro de un nodo, por ejemplo, cores de la CPU con coprocesadores. Los nodos virtuales utilizados en el experimento incluyen 12 o 6 cores de CPU, que se usan en la rutina de multiplicación de MKL, y en algunos casos se incluye también una GPU Tesla C2075. Las seis GPUs de **jupiter** están numeradas de 0 a 5, siendo las de tipo Tesla las 1 y 5. Así pues, las unidades de proceso en el experimento son subconjuntos de 12 o 6 cores de CPU (columnas *12 threads MKL* y *6 threads MKL*), o un número de cores de CPU y una de las GPUs Tesla.

- En general, se obtiene menor tiempo de ejecución con paralelismo anidado (columna *2 threads OpenMP*) combinando threads OpenMP con el paralelismo de la multiplicación híbrida.

- El paralelismo de dos niveles en CPU (columna *2 threads OpenMP* y *6 threads MKL*) proporciona menor tiempo de ejecución que si se explota únicamente paralelismo MKL (columna *1 thread OpenMP* y *12 threads MKL*).

- El uso de dos nodos virtuales de doce cores cada uno (columnas *12 threads MKL*) permite explotar de forma más eficiente el *hyperthreading* de la CPU multicore (figura 5.1).

	1 thread OpenMP	
n	*12 threads MKL*	*12 threads MKL + C2075*
3500	1.86	0.57
7000	9.27	3.25

	2 threads OpenMP		
n	*12 threads MKL*	*6 threads MKL*	*6 threads MKL + C2075*
3500	0.83	0.95	0.51
7000	4.78	5.29	1.97

Tabla 5.20: Comparación del tiempo de ejecución (en segundos) obtenido en **jupiter** por la multiplicación de Strassen con un nivel de recursión, para matrices de tamaño 3500 y 7000, usando paralelismo únicamente en la multiplicación (1 thread OpenMP) y con paralelismo anidado (2 threads OpenMP), estableciendo diferentes configuraciones del número de threads MKL con y sin el uso de GPU.

En el tipo de plataformas que se han considerado (nodos computacionales estándar) se pueden diferenciar dos niveles de hardware, pero, como se ha indicado, las unidades computacionales dentro de un nodo pueden, a su vez, agruparse para formar unidades computacionales de nivel 1, las cuales, a su vez, pueden agruparse para formar unidades de nivel 2, etc., con lo que el concepto de múltiples niveles hardware también sería válido dentro de un nodo. De este modo, la estructura jerárquica que se muestra en la figura 4.3 se podría ver como un esquema de programación dinámica [29, 45] en el que la solución óptima para un cierto tamaño de problema (nivel de software y hardware) se obtiene a partir de las soluciones óptimas para problemas más pequeños en niveles previos de software y/o hardware. No obstante, dada la posibilidad de definir unidades virtuales, el problema se hace más complejo. Por ejemplo, en un nodo con c componentes básicos, el número de posibles subnodos es 2^c (o superior si los cores de la CPU se agrupan en subgrupos de cores) y la multiplicación de matrices básica tendría que instalarse en todos los posibles subnodos virtuales, por lo que el esquema de programación dinámica debería considerar todas las posibles asignaciones de la multiplicación a todos los posibles subnodos, lo que puede llegar a ser inviable. Una solución más simple es considerar subnodos con grupos de cores y de CPUs con similar capacidad computacional. Por ejemplo, consideremos una estrategia `S1H2` dentro del nodo, con la multiplicación de Strassen asignando multiplicaciones básicas dinámicamente a unidades computacionales de nivel 1, que en este caso son subnodos. La figura 5.17 compara el tiempo de ejecución de la multiplicación de Strassen con nivel 1 en **jupiter** para distintas configuraciones de subnodos: 1×(12c+2C2075), un subnodo compuesto por 12 cores de CPU y 2 GPU C2075; 2×(6c+C2075), dos subnodos, cada uno con 6 cores de CPU y 1 GPU C2075; y 2×(4c+C2075)+1×(4c+GTX590), tres subnodos, cada uno con 4 cores de CPU, dos de ellos con una GPU C2075 y el otro con una GPU GTX590. Se observa que la extensión de la jerarquía en el nivel hardware posibilita una reducción en el tiempo de ejecución. Además, la decisión dentro de cada subnodo se delega a la multiplicación básica, que puede tomar decisiones distintas dependiendo del subnodo donde se ejecuta: para una matriz de tamaño 10000, la ejecución con la GPU C2075 asigna 1000 columnas de la matriz $\boldsymbol{B}$ a la CPU y 4000 columnas a la GPU, y con la GTX590, la distribución es de 1600 columnas para la CPU y 3400 para la GPU.

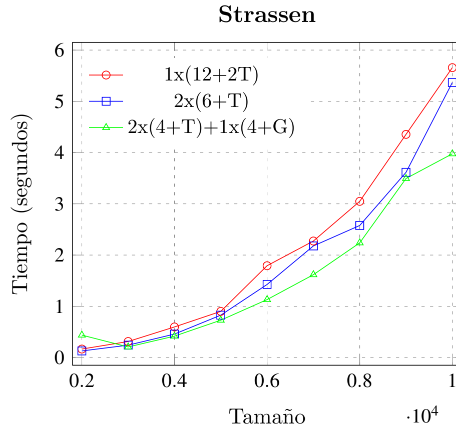

Figura 5.17: Tiempo de ejecución (en segundos) de la multiplicación de Strassen con asignación dinámica y nivel de recursión 1, con diferentes configuraciones de subnodos computacionales en **jupiter**, usando dos GPUs C2075 y una GTX590.

5.4 Uso de herramientas software para la obtención del valor de los parámetros algorítmicos

Hasta ahora, la obtención del valor de los parámetros algorítmicos se ha llevado a cabo aplicando de forma jerárquica la metodología de auto-optimización propuesta. Dado que la toma de decisiones para la obtención de dichos valores se puede llevar a cabo de diferentes formas, la metodología de auto-optimización se ha diseñado para permitir la inclusión de nuevas técnicas o herramientas software que permitan realizar la selección de los valores más apropiados para el tamaño de problema a resolver.

En esta sección se muestra cómo obtener el valor de determinados parámetros algorítmicos haciendo uso de la herramienta OpenTuner [17], integrándola dentro de la propia metodología de optimización jerárquica como una técnica más de búsqueda dentro del proceso de auto-optimización.

OpenTuner es un framework que permite definir auto-optimizadores multi-objetivo para la búsqueda de valores de parámetros algorítmicos en un dominio de valores concreto. Esta búsqueda la realiza mediante el uso simultáneo de diferentes técnicas heurísticas que incorpora y permite, a su vez, la definición e inclusión de nuevas técnicas de búsqueda.

Los experimentos realizados muestran los valores de los parámetros algorítmicos obtenidos por dicha herramienta para la rutina de multiplicación de matrices en diferentes niveles de la jerarquía hardware. Se ha considerado una plataforma compuesta por una CPU multicore y dos GPUs de diferente tipo (subnodo del nodo **jupiter**), con el fin de analizar cómo se comporta esta herramienta en un entorno heterogéneo a la hora de obtener el número de threads de CPU y la distribución de la carga de trabajo entre las unidades de cómputo (CPU y GPU).

Dado que el tiempo empleado para la obtención de los valores de los parámetros algorítmicos crece conforme aumenta el tamaño de problema y el dominio de búsqueda establecido, se ha limitado el tiempo de búsqueda estableciendo como cota superior el tiempo empleado por la propia metodología de auto-optimización en cada nivel de la jerarquía. Esto va a permitir, a su vez, analizar la calidad de la solución obtenida con esta herramienta software con respecto a usar directamente la metodología de optimización jerárquica.

La Tabla 5.21 muestra el número de threads y el rendimiento obtenido por OpenTuner y la metodología de optimización jerárquica para cada tamaño de problema en el nivel `S0H0` de la jerarquía, así como el tiempo de búsqueda empleado. Se observa que, aunque el valor obtenido por OpenTuner difiere para algunos tamaños de problema, el rendimiento es similar.

	OpenTuner		**Metodología Jerárquica**		
n	*threads*	*GFlops*	*threads*	*GFlops*	*Tiempo_Búsqueda*
1000	21	109.29	12	83.41	1
2000	23	161.78	12	163.53	5
3000	24	165.39	24	173.30	14
4000	13	189.69	24	179.13	33
5000	15	173.88	24	184.58	64
6000	24	183.77	24	186.55	108
7000	14	185.29	24	183.92	174

Tabla 5.21: Prestaciones obtenidas por OpenTuner y la metodología de optimización jerárquica con el mejor número de threads para cada tamaño de problema.

Las tablas 5.22, 5.23 y 5.24, por su parte, muestran los valores obtenidos para los parámetros algorítmicos en el nivel `S0H1` de la jerarquía (número de threads de CPU y distribución de la carga entre CPU y GPUs) cuando se hace uso de

OpenTuner estableciendo como límite de tiempo de búsqueda el empleado por la metodología de auto-optimización cuando se lleva a cabo la instalación de la rutina de multiplicación de diferentes formas:

- **Exhaustiva** en los parámetros de nivel 1 (reparto de la carga entre las unidades de cómputo) pero haciendo uso de la información de nivel 0 (`S0H1-s0h0`) para seleccionar el número de threads a usar en la CPU para la carga de trabajo asignada (tabla 5.22).
- **Guiada** (tanto en los parámetros de nivel 1 como de nivel 0) tomando como punto de partida, para cada tamaño de problema, la mejor configuración de los valores de dichos parámetros haciendo uso de la metodología de auto-optimización (tabla 5.23).
- Totalmente exhaustiva (tabla 5.24).

	OpenTuner (Límite Tiempo_Búsqueda = `S0H1-s0h0` sin Tiempo_Instalación `S0H0`)					
n	*threads*	n_{CPU}	n_{GPU0}	n_{GPU1}	$GFlops$	*Tiempo_Búsqueda*
1000	8	620	220	160	94.10	27
2000	19	720	440	840	307.23	117
3000	15	900	780	1320	366.63	362
4000	19	1280	880	1840	391.15	782
5000	15	1600	1300	2100	404.34	1474
6000	21	1920	1440	2640	460.00	2435
7000	19	2240	1680	3080	461.77	3740

Tabla 5.22: Parámetros algorítmicos (threads de CPU y distribución de la carga entre CPU y GPUs) y prestaciones obtenidas por OpenTuner para cada tamaño de problema, utilizando como límite de tiempo de búsqueda el empleado por la metodología jerárquica al realizar una instalación **exhaustiva** de tipo `S0H1-s0h0`.

En la figura 5.18 (izquierda) se muestra una comparativa del rendimiento obtenido por la metodología de optimización jerárquica, con las tres posibles versiones planteadas en esta sección, frente al ofrecido por OpenTuner cuando su tiempo de búsqueda se limita al de cada una de las versiones indicadas. Asimismo, se muestra el rendimiento máximo (***Peak***) que se alcanzaría si se suman las

n	**OpenTuner** (Límite Tiempo_Búsqueda = Tiempo_Instalación Guiada `S0H1-s0h0`)					
	threads	n_{CPU}	n_{GPU0}	n_{GPU1}	*GFlops*	*Tiempo_Búsqueda*
1000	0	0	440	560	120.44	1
2000	22	680	400	920	291.34	3
3000	9	360	900	1740	306.60	7
4000	16	1200	960	1840	374.30	11
5000	14	500	1700	2800	350.76	30
6000	9	1080	1680	3240	406.11	32
7000	15	2240	1540	3220	435.04	92

Tabla 5.23: Parámetros algorítmicos (threads de CPU y distribución de la carga entre CPU y GPUs) y prestaciones obtenidas por OpenTuner para cada tamaño de problema, utilizando como límite de tiempo de búsqueda el empleado por la metodología jerárquica al realizar una instalación **guiada** de tipo `S0H1-s0h0`.

n	**OpenTuner** (Límite Tiempo_Búsqueda = Tiempo_Búsqueda Exhaustiva en `H1`)					
	threads	n_{CPU}	n_{GPU0}	n_{GPU1}	*GFlops*	*Tiempo_Búsqueda*
1000	19	580	220	200	103.73	643
2000	18	720	440	840	320.32	3297
3000	9	720	840	1440	342.87	10111
4000	21	1280	880	1840	419.02	22394
5000	21	1600	1200	2200	434.57	42602
6000	23	1800	1440	2760	459.10	71603
7000	21	2240	1680	3080	460.23	108878

Tabla 5.24: Parámetros algorítmicos (threads de CPU y distribución de la carga entre CPU y GPUs) y prestaciones obtenidas por OpenTuner para cada tamaño de problema, utilizando como límite de tiempo de búsqueda el empleado por la metodología jerárquica al realizar una instalación **exhaustiva** en el nivel `S0H1`.

prestaciones de las unidades de cómputo consideradas. Se observa que el rendimiento obtenido por OpenTuner mejora conforme aumenta el límite establecido en el tiempo de búsqueda (en OT_{exh_time} el límite es mayor que en OT_{L1_Time}) y se aproxima al óptimo exhaustivo conforme aumenta el tamaño de problema, ya que puede explorar un mayor rango de valores en el espacio de búsqueda. Por otro lado, se observa que el uso de la metodología de optimización jerárquica permite obtener resultados similares a OpenTuner y cercanos al óptimo exhaustivo, pero

empleando un menor tiempo de búsqueda. Esto se consigue aplicando la versión de la búsqueda guiada descrita previamente (HL_{guided}). La figura 5.18 (derecha), por su parte, muestra la pérdida de rendimiento que se obtiene con respecto a usar una búsqueda completamente exhaustiva en el nivel 1 de la jerarquía (HL_{exh}). En el caso de OpenTuner, dicho valor es de un 11.2 % y un 15.7 % con los límites de tiempo de HL_{L1} y HL_{guided}, respectivamente. Cuando se usa la versión HL_{L1}, esta pérdida es de un 12.8 %, y con HL_{guided}, de un 4.8 %. Por tanto, a pesar de que OpenTuner ofrece resultados ligeramente mejores que HL_{L1} incluso realizando una búsqueda exhaustiva de los parámetros en el nivel `SOH1` de la jerarquía, el uso del enfoque jerárquico combinado con una búsqueda guiada de los valores de estos parámetros (HL_{guided}) permite mejorar aún más los resultados obtenidos, con una reducción en el tiempo de búsqueda.

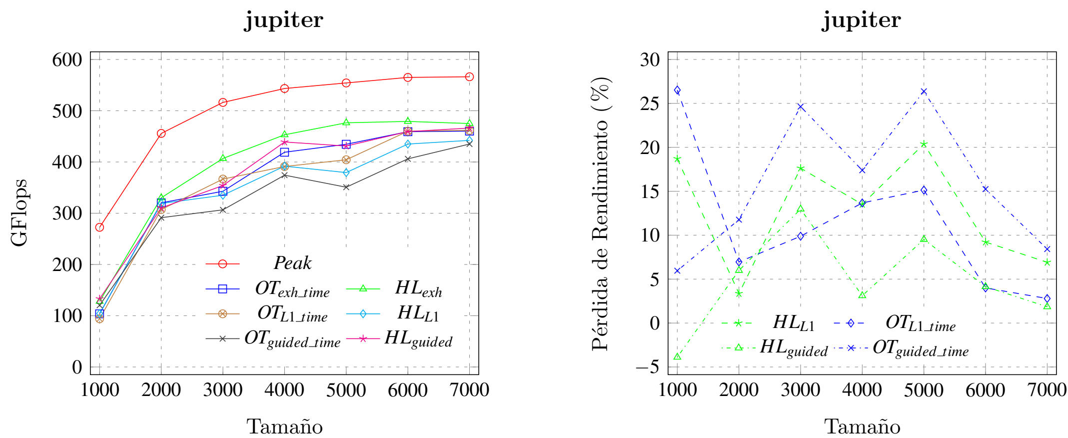

Figura 5.18: A la izquierda, prestaciones obtenidas por OpenTuner (*OT*) y por la metodología de optimización jerárquica (*HL*) al utilizar un subnodo de **jupiter** (CPU, Tesla C2075 y GeForce GTX590) variando el tamaño de problema y utilizando tres versiones diferentes para la búsqueda de los valores de los parámetros algorítmicos (*Exh*, *L*1, *Guided*). Asimismo, se muestran las prestaciones máximas que se alcanzarían (*Peak*). A la derecha, pérdida de rendimiento experimentada por los métodos que explotan la jerarquía y por OpenTuner (con los límites de tiempo correspondientes) con respecto al método totalmente exhaustivo (HL_{exh}).

Con los resultados de este estudio se pretende mostrar, por un lado, que la metodología de auto-optimización propuesta permite hacer uso de herramientas software externas para la obtención del valor de los parámetros algorítmicos en cada nivel de la jerarquía y, por otro, que el uso de la propia metodología permite obtener prestaciones similares y cercanas al óptimo en un tiempo menor.

5.5 Conclusiones

Tal como se indicó al comienzo del capítulo, la finalidad de este estudio experimental era mostrar la utilidad de la metodología de auto-optimización propuesta en los niveles hardware y software de la jerarquía.

Por un lado, se han considerado rutinas de diferentes niveles software: la multiplicación de matrices en el nivel más bajo y la multiplicación de Strassen y la factorización LU (con llamadas a la multiplicación básica) en un siguiente nivel.

Por otro lado, se han mostrado experimentos en tres niveles hardware de un cluster heterogéneo: unidades computacionales básicas (CPU, GPU y MIC), nodos CPU+multicoprocesador y plataforma completa.

El objetivo no es optimizar las rutinas, sino mostrar cómo la metodología propuesta facilita su optimización de forma automática en sistemas computacionales heterogéneos. Se podrían incluir también optimizaciones adicionales de las rutinas, considerando nuevos parámetros en algunos de los niveles de la jerarquía con el fin de obtener mejores tiempos de ejecución.

Asimismo, se ha mostrado cómo el diseño ofrecido por la propia metodología permite la inclusión de nuevas técnicas y herramientas software para la selección de los valores de los parámetros algorítmicos en cualquier nivel de la jerarquía. Todo ello, demuestra que la metodología propuesta puede extenderse tanto a otras rutinas y librerías de álgebra lineal, como a otras plataformas computacionales.

Capítulo 6

Extensión de la metodología a librerías basadas en tareas con asignación dinámica

Este capítulo surge a raíz de las tareas de investigación llevadas a cabo durante la estancia realizada en Burdeos en el Instituto Nacional de Investigación en Informática y Automática (INRIA Bordeaux Sud-Ouest). El estudio realizado consiste en la aplicación de técnicas de auto-optimización a librerías de álgebra lineal cuyas rutinas se ejecutan haciendo uso de un planificador de tareas que asigna, de forma dinámica, cada uno de los kernels computacionales básicos a los distintos elementos computacionales presentes en la plataforma computacional heterogénea.

En concreto, el estudio se centra en Chameleon [2, 101], una librería basada en tareas y cuyas rutinas están implementadas siguiendo un esquema algorítmico basado en *tiles*. Así pues, las técnicas de auto-optimización estarán encaminadas a la búsqueda del valor de determinados parámetros que afectan al rendimiento de las rutinas, como el tamaño del *tile* o la política de planificación escogida, entre otros. Se adoptarán dos enfoques diferentes, uno empírico y otro simulado, y se aplicarán las mismas estrategias de búsqueda con cada uno. Los experimentos se realizarán sobre una plataforma heterogénea compuesta por una CPU multicore y varias GPUs, dado que dicha librería no incluye soporte para Xeon Phi.

El capítulo comienza analizando cómo la metodología jerárquica propuesta puede ser extendida para dar soporte a este tipo de librerías de álgebra lineal. A continuación, se describe con detalle la estructura y funcionamiento de la librería Chameleon y se muestran diferentes ejemplos de aplicación de la metodología sobre rutinas de álgebra lineal, como la descomposición de Cholesky y la factorización LU, aplicando los dos enfoques mencionados, empírico y simulado, considerando diferentes parámetros algorítmicos para optimizar su ejecución. Finalmente, se muestran las principales conclusiones obtenidas.

6.1 Extensión de la metodología a otras librerías numéricas

En el Capítulo 4 se ha presentado la metodología de optimización multinivel, así como su posible extensión a nivel hardware (combinación de varias de unidades de cómputo posiblemente de tipos distintos) y software (rutinas de mayor nivel). En el estudio experimental realizado en el Capítulo 5, se ha mostrado su funcionamiento con diferentes librerías optimizadas de álgebra lineal (MKL y cuBLAS) en función del tipo de sistema computacional empleado (CPU multicore, GPU o Xeon Phi). Del mismo modo, se podría haber extendido su funcionalidad para hacer uso de otras librerías, como PLASMA (para multicore) o MAGMA (para multicore+GPU), teniendo en cuenta las características propias de cada una de ellas, pero la instalación de las rutinas en los diferentes niveles de la jerarquía se realizaría de forma similar.

El escenario cambia cuando se utilizan librerías que, aparte de las propias rutinas de álgebra lineal, incluyen un conjunto de componentes software que actúan de forma conjunta para la ejecución de dichas rutinas. Es el caso de librerías basadas en tareas, como Chameleon, objeto de estudio del presente capítulo. Este tipo de librerías disponen de un sistema de planificación en tiempo de ejecución que, mediante el uso de determinadas políticas, planifica y ejecuta de forma dinámica las tareas (rutinas computacionales básicas) en las diferentes unidades básicas de procesamiento (CPU, GPU…) haciendo uso, en cada una de ellas, de librerías optimizadas como las mencionadas anteriormente. Por tanto, dado que la ejecu-

ción de las tareas se delega directamente a la rutina correspondiente de dichas librerías, la metodología ha de ser adaptada en un nivel superior de la jerarquía para poder optimizar su ejecución en función de parámetros algorítmicos propios de las rutinas (como el tamaño de bloque), el tipo de planificador a usar o el número de unidades de cómputo a establecer. Esto va a permitir, por un lado, dotar a la metodología de mayor funcionalidad y, por otro, mostrar su validez en otro tipo de escenarios con independencia de la plataforma hardware o software subyacente.

6.2 Estructura general de Chameleon

Esta sección describe la estructura y funcionamiento de Chameleon, una librería de álgebra lineal densa que proporciona implementaciones eficientes de rutinas de BLAS y LAPACK para los sistemas computacionales actuales. Está desarrollada en lenguaje C y la implementación de las rutinas se deriva de la realizada en la librería PLASMA, que se caracteriza por seguir un diseño algorítmico basado en *tiles*, con el fin de agilizar el acceso a los bloques de memoria situados en los diferentes niveles de cache. Por tanto, la optimización de las rutinas incluidas en esta librería dependerá, en gran medida, del tamaño de *tile* utilizado para cada tamaño de problema.

La figura 6.1 muestra el esquema general de funcionamiento de la librería. Cada operación matricial realizada por las rutinas de álgebra lineal, se descompone en una serie de tareas que actúan sobre diferentes bloques de datos (*tiles*), organizados en forma de *layout*. A continuación, se crea un grafo de dependencias entre las tareas que actúan sobre dichos bloques. Estas tareas corresponden a los kernels computacionales básicos invocados por la rutina. Finalmente, estas tareas se planifican en tiempo de ejecución utilizando alguna de las políticas ofrecidas por el planificador y se asignan a las unidades de procesamiento disponibles, donde se ejecutarán utilizando la versión de la rutina ofrecida por las librerías optimizadas de álgebra lineal disponibles en el sistema.

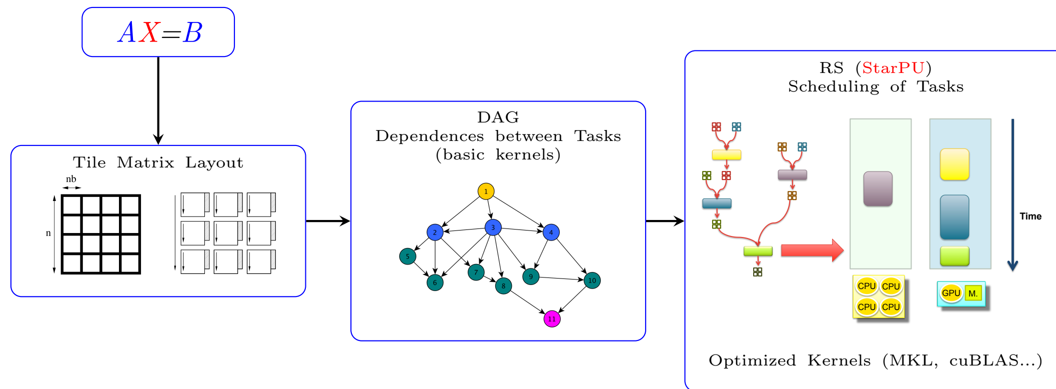

Figura 6.1: Esquema general de funcionamiento de la librería Chameleon.

Es importante remarcar que la propia librería no incorpora todos los componentes necesarios que intervienen durante la ejecución de las rutinas. Tanto el planificador como las librerías optimizadas a utilizar en cada elemento computacional han de ser instaladas previamente en la plataforma para que puedan ser enlazadas posteriormente con la propia librería durante su instalación. Así pues, la librería Chameleon puede ser considerada como la implementación de una interfaz para el acceso a las rutinas de PLASMA, adaptada para invocar a kernels optimizados de otras librerías mediante el uso de un planificador dinámico de tareas, como StarPU [20].

6.2.1 Uso de simulación

Hasta ahora, el funcionamiento de la librería Chameleon se ha descrito prestando especial atención al planificador dinámico de tareas de que dispone para gestionar la ejecución de los kernels computacionales de las rutinas en las unidades de cómputo del sistema. En cambio, esta librería también permite ser utilizada junto al simulador SimGrid [178]. De esta forma, se puede obtener una estimación del tiempo de ejecución (y del rendimiento) de las rutinas, evitando tener que disponer de los recursos hardware de la plataforma computacional cada vez que se quiera evaluar el rendimiento de una rutina.

La figura 6.2 muestra un esquema general de funcionamiento de Chameleon cuando se usa junto al simulador SimGrid. El simulador hace uso de modelos de rendimiento (denominados *codelets*) que son generados durante la instalación de

la rutina para cada kernel computacional de la misma. Cada *codelet* almacena información relativa al rendimiento obtenido por dicho kernel en cada unidad de cómputo (CPU, GPU) de la plataforma cuando se ejecuta con un tamaño de bloque concreto y un conjunto de tamaños de problema. Para obtener estimaciones precisas del rendimiento, estos modelos han de estar debidamente calibrados, lo que se consigue ejecutando varias veces cada experimento realizado con la rutina durante la fase de instalación. Inicialmente, este proceso supone un incremento adicional en el tiempo empleado durante la instalación de la rutina, pero una vez calibrados los *codelets*, ya pueden ser utilizados en cualquier computador personal para realizar nuevas estimaciones sin necesidad de volver a ejecutar la rutina en el sistema computacional.

Cabe indicar que la idea que subyace bajo el uso del simulador, se asemejaría bastante al uso de modelos teórico-experimentales de las rutinas, pero usando en este caso los modelos de rendimiento (*codelets*) obtenidos tras la ejecución de las rutinas con un sistema de asignación dinámico de tareas.

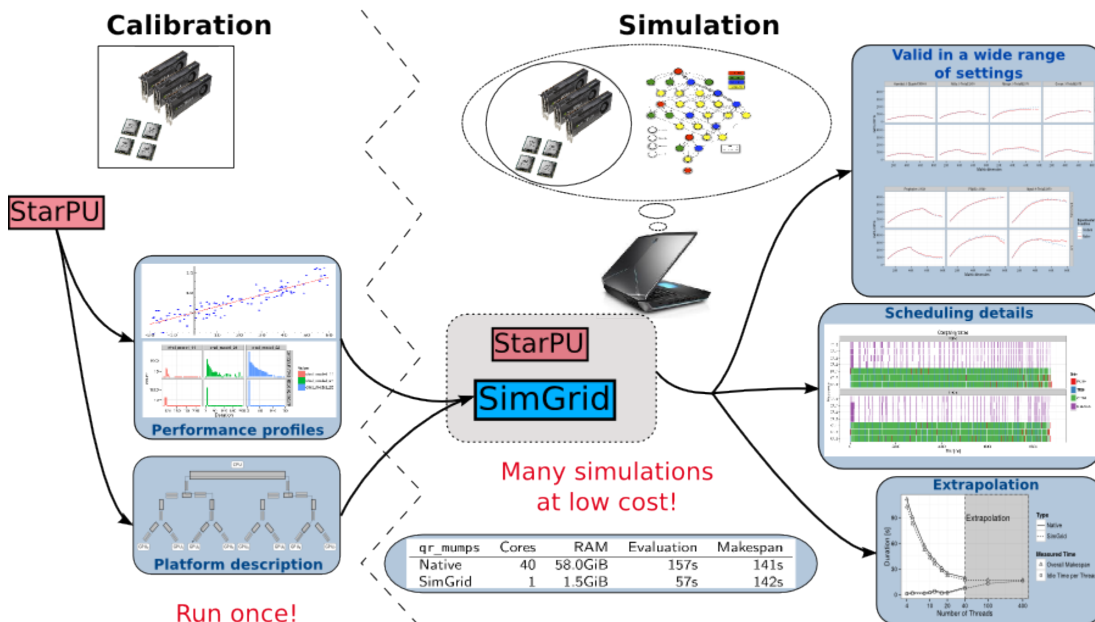

Figura 6.2: Funcionamiento de Chameleon con el simulador SimGrid.

6.3 Optimización de Rutinas

En esta sección se presentan diferentes estrategias de optimización para la selección del valor de los parámetros algorítmicos que afectan al rendimiento de determinadas rutinas de la librería Chameleon. Cada una de las estrategias se aplicará usando los enfoques empírico y simulado. En el caso empírico, se hará uso de la metodología de auto-optimización. Ésta se aplicará directamente en el nivel 1 (hardware y software) de la jerarquía, debido a que se delega la ejecución de las rutinas básicas en los elementos de cómputo del nivel 0 al planificador de tareas utilizado por la librería.

El estudio se ha llevado a cabo con la rutina de descomposición de Cholesky y con la factorización LU. Dado que ambas rutinas están implementadas siguiendo un esquema algorítmico basado en bloques (*tiles*), se analizará el impacto que tiene este parámetro en su rendimiento. Además, en la factorización LU se considerará un parámetro adicional, el tamaño de bloque interno, ya que la rutina hace uso del mismo dentro del tamaño de bloque *tile* durante la resolución de la factorización matricial.

Los experimentos se han llevado a cabo en **jupiter**, haciendo uso de todas las unidades de cómputo disponibles en el nodo (2 CPU hexa-core, 4 NVIDIA GeForce GTX590 y 2 Tesla C2075), pero se podría aplicar de forma similar en el resto de nodos de cómputo de la plataforma computacional.

En secciones posteriores se mostrará cómo aplicar técnicas de auto-optimización que permitan hacer un uso eficiente de los recursos computacionales, seleccionando el número apropiado de unidades de cómputo a utilizar en función del tamaño de problema.

6.3.1 Descomposición de Cholesky

La descomposición de Cholesky se utiliza para la resolución de operaciones matriciales del tipo $Ax = b$, donde A es la matriz de coeficientes, x es el vector de términos dependientes y b el vector de términos independientes. La operación de descomposición sólo se puede aplicar cuando la matriz A es simétrica definida

positiva. En tal caso, la factorización consiste en descomponer la matriz A en el producto de dos matrices, $A = LL^T$, donde L es una matriz triangular y L^T su traspuesta. La figura 6.3 muestra el proceso que se lleva a cabo para realizar la descomposición.

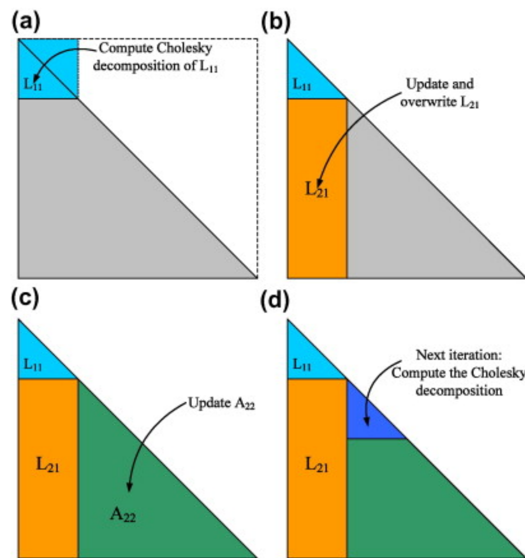

Figura 6.3: Operaciones realizadas sobre los elementos de una matriz en cada paso de la descomposición de Cholesky.

En el cálculo de la descomposición matricial intervienen varias operaciones algebraicas que se llevan a cabo invocando a diferentes rutinas básicas de álgebra lineal. El algoritmo 11 muestra los kernels a los que se invoca para realizar la descomposición de Cholesky. Durante la ejecución del algoritmo, cada uno de los kernels actúa sobre diferentes bloques de datos de la matriz, accediendo en modo lectura (***R***) o lectura+escritura (***RW***), cuyo tamaño viene determinado por el valor del tamaño de bloque, ***nb***. Por defecto, Chameleon considera un valor fijo para el tamaño de bloque, independientemente del tamaño de problema. Dado que la ejecución de los diferentes kernels en las unidades de procesamiento paralelo del nodo (CPU y GPU) la lleva a cabo el planificador de tareas de forma dinámica, el valor de ***nb*** seleccionado afectará al rendimiento obtenido en cada uno de ellos. Por tanto, el rendimiento de la rutina mejorará si se selecciona de forma apropiada el valor de este parámetro algorítmico para cada tamaño de problema. Para ello, se aplicarán las siguientes estrategias de búsqueda haciendo uso de la metodología de optimización:

- **Exhaustiva**: la rutina se ejecuta con un conjunto de tamaños de bloque para cada tamaño de problema, n, almacenando el tiempo de ejecución y el rendimiento obtenido para cada par (n, nb). Esta técnica permite analizar, a su vez, el comportamiento de la rutina conforme varía el tamaño de bloque.

- **Guiada**: la rutina se instala comenzando con el primer tamaño de problema del conjunto de instalación y el primer tamaño de bloque considerado, nb_1, y se incrementa su valor hasta que se obtiene una disminución en el rendimiento. A continuación, el proceso continúa con el siguiente tamaño de problema (en orden ascendente), tomando como punto de partida el valor de nb con el que se obtuvo el mejor rendimiento para el tamaño de problema anterior. Entonces, se lleva a cabo una búsqueda bidireccional en el valor de nb (con valores crecientes y decrecientes) hasta que se obtenga un valor en el rendimiento que empore el obtenido hasta el momento. Este proceso se repite con cada uno de los tamaños de problema del conjunto de instalación. Como resultado, se almacena para cada tamaño de problema, el tiempo de ejecución y el rendimiento obtenido con el mejor valor de nb. El uso de esta estrategia permite reducir considerablemente el tiempo de instalación de las rutinas respecto al uso de la estrategia anterior.

Algoritmo 11: Algoritmo para la descomposición de Cholesky.

```
for j = 0; j < n; j = j + nb do
    POTRF (RW,A[j][j]);
    for i = j + 1; i < n; i = i + nb do
        TRSM (RW,A[i][j], R,A[j][j]);
    end
    for i = j + 1; i < n; i = i + nb do
        SYRK (RW,A[i][i], R,A[i][j]);
        for k = j + 1; k < i; k = k + nb do
            GEMM (RW,A[i][k], R,A[i][j], R,A[k][j]);
        end
    end
end
```

6.3.2 Factorización LU

Al igual que sucede en la rutina de Cholesky, en el cálculo de la factorización LU de una matriz intervienen varias operaciones algebraicas que se llevan a cabo invocando a diferentes rutinas básicas de álgebra lineal. El algoritmo 12 muestra cómo se lleva a cabo la factorización LU, indicando las rutinas básicas a las que invoca en cada paso y el modo en que accede cada una de ellas (modo lectura (R) o lectura+escritura (RW)) a los diferentes bloques de datos de la matriz. Un ejemplo de su ejecución se puede observar en la figura 6.4, donde se muestra cómo se aplican las diferentes operaciones matriciales para realizar la factorización LU del bloque $A11$ de la matriz. El resultado es un bloque de datos que se puede expresar como el producto de dos matrices, $L11$ y $U11$, correspondientes a la parte triangular inferior y superior, respectivamente, obtenidas tras el cálculo de la factorización con dicho bloque. Tras ello, se actualiza el resto de la matriz aplicando de nuevo, en el mismo orden, las operaciones matriciales básicas del algoritmo, y el proceso continúa hasta que se alcanza el último bloque de datos de la matriz principal.

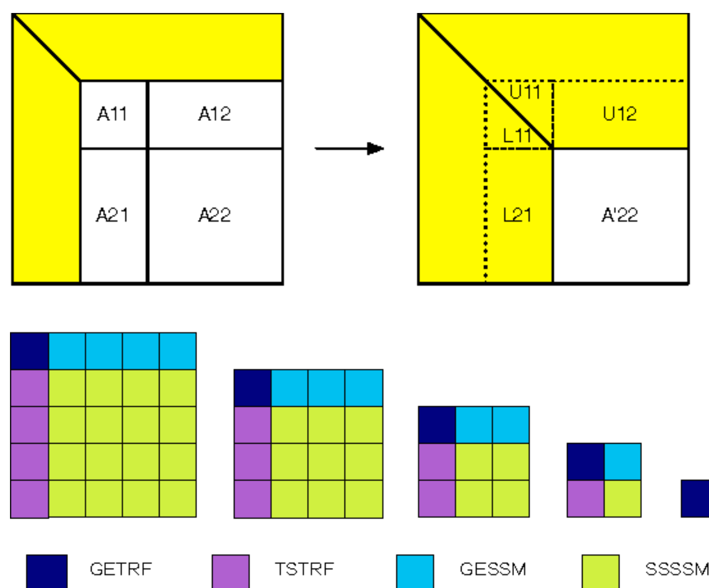

Figura 6.4: Factorización LU de una matriz.

A diferencia de la rutina de Cholesky, el algoritmo implementado en Chameleon para la factorización LU, hace uso de un tamaño de bloque interno dentro

Algoritmo 12: Algoritmo para la factorización LU.

```
for j = 0; j < n; j = j + nb do
    DGETRF (R,A[j][j], W,L[j][j], W,U[j][j]);
    for i = j + 1; i < n; i = i + nb do
        DGESSM (R,A[j][i], R,L[j][j], W,U[j][i]);
    end
    for i = j + 1; i < n; i = i + nb do
        DTSTRF (R,A[i][j], R,L[i][j], RW,U[j][j]);
        for k = j + 1; k < n; k = k + nb do
            DSSSSM (R,L[i][j], RW,U[j][k], RW,A[i][k]);
        end
    end
end
```

del propio tamaño de bloque. En el ejemplo propuesto, esto equivaldría a considerar cada elemento del bloque de datos de la matriz principal con el que se trabaja, como un sub-bloque dentro del mismo. Así pues, el rendimiento de la rutina estará condicionado, a su vez, por el valor de este parámetro adicional. Por tanto, la metodología de optimización se tendrá que aplicar teniendo en cuenta, además del tamaño de bloque, nb, el tamaño de bloque interno, ib. Por defecto, Chameleon considera un valor fijo para dicho parámetro, independientemente del tamaño de bloque y de problema, por tanto, el rendimiento de la rutina mejorará si se selecciona de forma apropiada, además del valor de nb, el valor de dicho parámetro algorítmico. Se considerarán las mismas estrategias de búsqueda que con la descomposición de Cholesky, pero teniendo en cuenta este nuevo parámetro en cada una de ellas:

- **Exhaustiva**: la rutina se ejecuta con un conjunto de tamaños de bloque para cada tamaño de problema, n, variando el valor del tamaño de bloque interno, ib, hasta alcanzar el valor de nb. Como resultado se almacena el tiempo de ejecución y el rendimiento obtenido para cada configuración de valores (n, nb, ib).
- **Guiada_1D**: la rutina se instala comenzando con el primer tamaño de problema del conjunto de instalación y el primer tamaño de bloque considerado, (n_1, nb_1), incrementando el tamaño de bloque interno, ib, hasta que se alcance el valor de nb_1, o se obtenga un valor en el rendimiento que empore

el obtenido hasta el momento. Para evitar que el proceso se estanque en un mínimo local, se hace uso de un valor porcentual que se aplica sobre el rendimiento obtenido para decidir si la búsqueda continúa hacia el siguiente valor de ***ib***. El proceso continúa con el siguiente valor de ***nb*** para el mismo tamaño de problema, tomando como punto de partida el valor de ***ib*** con el que se obtuvo el mejor rendimiento para el tamaño de bloque anterior. Entonces, se lleva a cabo una búsqueda bidireccional en el valor de ***ib*** (con valores crecientes y decrecientes) usando el valor porcentual indicado, hasta que se detecte una disminución en el rendimiento o se alcance el valor de ***nb***. Una vez finalizada la búsqueda con el último valor de ***nb***, se almacena el tiempo de ejecución y el rendimiento obtenido con la mejor configuración (***nb***, ***ib***) para el tamaño de problema actual. A continuación, se selecciona el siguiente tamaño de problema del conjunto de instalación (en orden ascendente) y se repite el proceso de búsqueda para ***ib***, comenzando de nuevo con el primer tamaño de bloque, $\boldsymbol{nb}_1$. Esta estrategia permite reducir considerablemente el tiempo de instalación de las rutinas respecto al uso de una estrategia exhaustiva.

- **Guiada_2D**: esta estrategia comienza la instalación de la rutina de forma similar a la anterior (Guiada_1D), pero cuando el proceso avanza hacia el siguiente tamaño de problema, la búsqueda bidireccional se aplica tanto en el tamaño de bloque, ***nb***, como en el tamaño de bloque interno, ***ib***, tomando como punto de partida la mejor configuración (***nb***, ***ib***) obtenida para el tamaño de problema anterior. En este caso, también se utiliza el valor porcentual para evitar que el proceso se estanque en un mínimo local, pero solamente se aplica cuando se realiza la búsqueda variando el tamaño de bloque interno. Con esta estrategia se consigue reducir aún más el tiempo de instalación de las rutinas, ya que se reduce considerablemente el espacio de búsqueda.

La principal diferencia entre la búsqueda Guiada_1D y la búsqueda Guiada_2D radica en los parámetros sobre los que se realiza la búsqueda bidireccional. En la Guiada_1D sólo se aplica sobre el tamaño de bloque interno (***ib***), en cambio, en la Guiada_2D se aplica tanto en el tamaño de bloque (***nb***), como en el tamaño de bloque interno (***ib***).

Las siguientes secciones muestran cómo se lleva a cabo la obtención de los valores de los parámetros algorítmicos para cada rutina (Cholesky y LU) aplicando la metodología de optimización con cada una de las estrategias de búsqueda, así como el rendimiento que se obtiene utilizando los dos enfoques propuestos (empírico y simulado).

6.3.3 Uso de la metodología de optimización

El estudio realizado muestra el rendimiento obtenido con los valores de los parámetros algorítmicos seleccionados por la metodología de optimización con cada una de las rutinas consideradas, combinando de forma empírica (en la plataforma considerada) y simulada (con el simulador SimGrid) las diferentes estrategias de búsqueda. La metodología se aplicará directamente en el nivel 1 de la jerarquía, tanto a nivel hardware como software (`S1H1`), ya que la ejecución de los diferentes kernels computacionales básicos en cada unidad de cómputo de nivel 0 es administrada en tiempo de ejecución por el planificador de tareas de la propia librería Chameleon.

Tanto en el estudio empírico como simulado, se ha considerado el siguiente conjunto de instalación de tamaños de problema: {2000, 4000, 8000, 12000, 16000, 20000, 24000, 28000} y el conjunto de tamaños de bloque: {208, 256, 288, 320, 384, 448, 512, 576}. Se podría haber utilizado cualquier otro conjunto, tanto de tamaños de bloque como de problema, pero se ha realizado de esta manera para poder analizar de una forma más homogénea, tanto el rendimiento de las rutinas como las decisiones tomadas por la metodología de optimización con cada estrategia de búsqueda. Los experimentos se han llevado a cabo en **jupiter**, nodo de cómputo del cluster **Heterosolar** compuesto por 2 CPU multicore (hexa-core) y 6 GPUs NVIDIA (4 GeForce GTX590 y 2 Tesla C2075).

La figura 6.5 (arriba izquierda), muestra las prestaciones obtenidas en **jupiter** al instalar la rutina de Cholesky siguiendo un enfoque empírico con una estrategia exhaustiva. Se muestra cómo el rendimiento obtenido al variar el tamaño de problema varía en función del tamaño de bloque utilizado, de ahí que la selección adecuada de dicho valor permita optimizar el rendimiento de la rutina. La figura 6.5 (arriba derecha) muestra, para cada tamaño de problema, el tamaño de

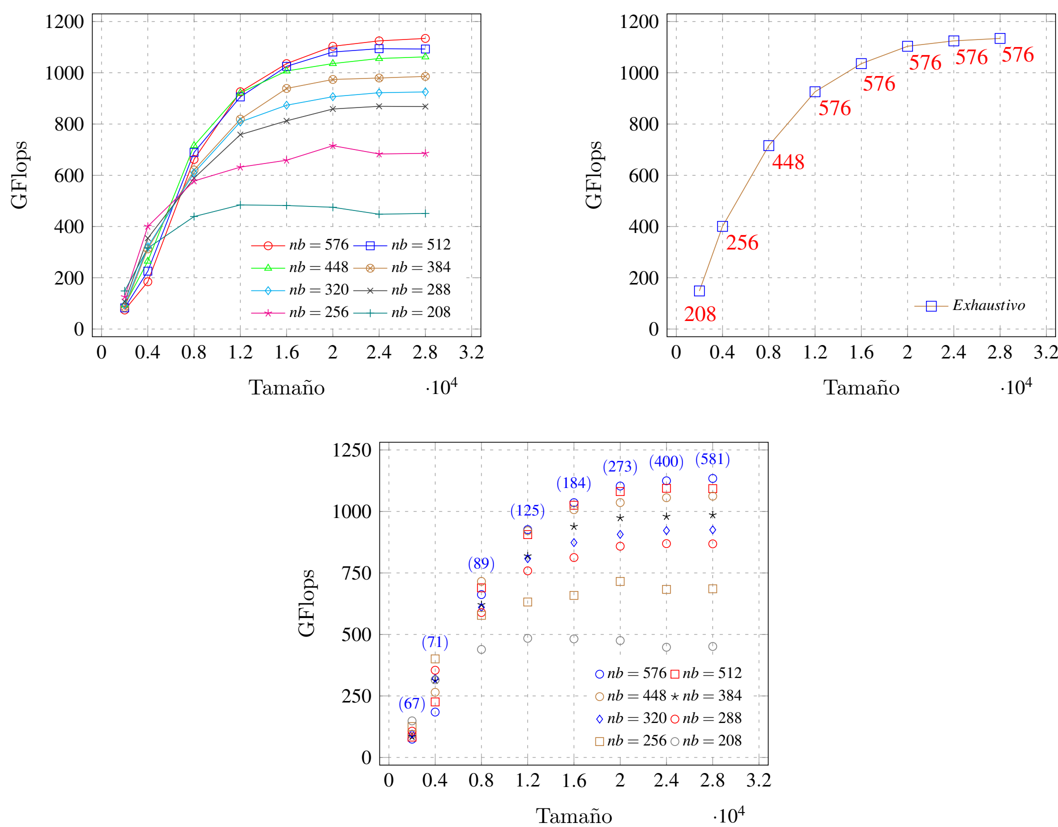

Figura 6.5: Prestaciones obtenidas al instalar la rutina de descomposición de Cholesky de forma exhaustiva con un conjunto arbitrario de tamaños de bloque (arriba izquierda); prestaciones obtenidas con el mejor valor de ***nb*** para cada tamaño de problema (arriba derecha), y tiempo de instalación (en segundos) empleado con cada tamaño de problema (abajo).

bloque (de entre los utilizados durante la instalación de la rutina) con el que se obtiene el mejor rendimiento. Se observa que, para tamaños de problema mayores a 8000, el mejor valor para ***nb*** corresponde siempre al mayor del conjunto utilizado. Asimismo, en la figura 6.5 (abajo) se puede observar el tiempo de instalación empleado para cada tamaño de problema con todos los tamaños de bloque del conjunto utilizado, lo que suma un total de, aproximadamente, 30 minutos.

La figura 6.6 (izquierda), muestra el rendimiento obtenido por la rutina de Cholesky con el valor de ***nb*** seleccionado para cada tamaño de problema al instalar la rutina con la metodología de optimización haciendo uso de una búsqueda guiada en el valor de ***nb***. La figura 6.6 (derecha), por su parte, compara el rendimiento obtenido con los mejores valores de ***nb*** para cada tamaño de problema al utilizar

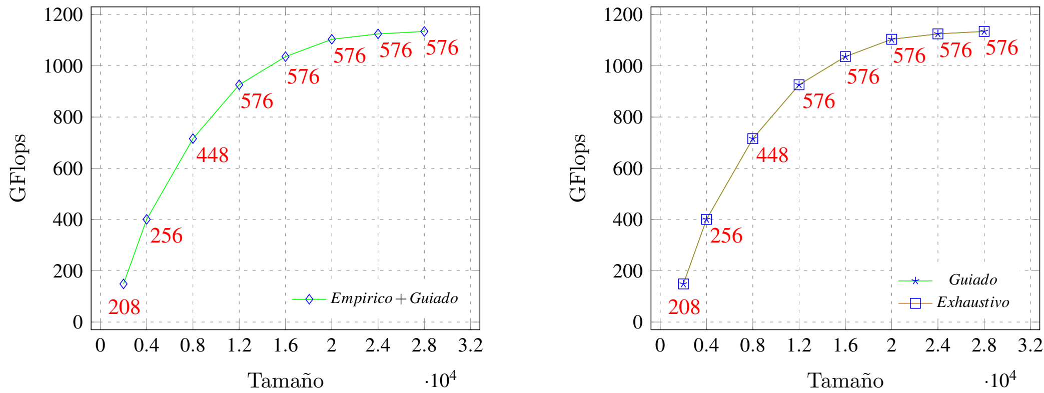

Figura 6.6: Prestaciones obtenidas al instalar la rutina de descomposición de Cholesky aplicando una búsqueda guiada con poda (izquierda), y comparación de las prestaciones obtenidas con respecto a la instalación exhaustiva (derecha).

el enfoque empírico con cada una de las estrategias de búsqueda (exhaustiva y guiada). Se observa que tanto el rendimiento como los valores obtenidos para ***nb*** coinciden al aplicar ambas estrategias, pero en la guiada, el tiempo empleado en la búsqueda de dichos valores es de tan sólo 154 segundos, mientras que en el caso exhaustivo alcanza los 30 minutos.

Hasta ahora, la instalación de la rutina se ha realizado de forma empírica, es decir, ejecutando la rutina sobre la plataforma computacional con cada estrategia de búsqueda. Esto se traduce en la necesidad de disponer, por un lado, de los recursos hardware de la propia plataforma y, por otro, del tiempo necesario para llevar a cabo toda la fase de experimentación. Dado que durante la fase de instalación se generan los modelos de rendimiento (*codelets*) de cada una de las rutinas computacionales básicas a las que invoca la rutina que se está instalando, se podría hacer uso del simulador SimGrid para estimar el rendimiento que se obtendría para cada tamaño de problema, haciendo uso de las mismas estrategias de búsqueda junto con la información almacenada tras la instalación de la rutina con cada una, evitando así el uso de la plataforma computacional.

La figura 6.7 (arriba izquierda), muestra cómo varía el rendimiento que se obtiene con la rutina de Cholesky al utilizar el enfoque simulado con una estrategia exhaustiva. La figura 6.7 (arriba derecha) simplifica la anterior, mostrando el rendimiento obtenido con el mejor valor de ***nb*** para cada tamaño de problema. Asimismo, la figura 6.7 (abajo izquierda) muestra el rendimiento obtenido con

ambas estrategias de búsqueda (exhaustiva y guiada) al usar el mejor valor para el tamaño de bloque para cada tamaño de problema. Tanto el rendimiento como los valores obtenidos para ***nb*** coinciden al aplicar ambas estrategias, pero el tiempo empleado con la búsqueda guiada es de tan sólo 84 segundos, frente a los 30 minutos que se requieren con la búsqueda exhaustiva. Finalmente, la figura 6.7 (abajo derecha), muestra una comparativa de las prestaciones obtenidas utilizando las citadas estrategias en ambos enfoques (empírico y simulado). Se observa que, salvo para algunos tamaños pequeños de problema en los que la desviación es mínima, tanto el rendimiento como los valores obtenidos para ***nb*** coinciden, lo que demuestra que el uso de un enfoque basado en simulación puede ser aplicado de forma eficiente, especialmente cuando se combina con estrategias de búsqueda guiada de los valores de los parámetros algorítmicos de las rutinas, pues permite reducir de forma sustancial el tiempo total de búsqueda empleado.

El estudio realizado con la descomposición de Cholesky se ha llevado a cabo del mismo modo con la rutina de factorización LU, pero considerando en este caso el nuevo parámetro algorítmico que incorpora esta rutina (el tamaño de bloque interno, ***ib***), en la aplicación de la metodología de optimización con las diferentes estrategias de búsqueda. La figura 6.8 (arriba izquierda), muestra las prestaciones obtenidas en **jupiter** al instalar dicha rutina de forma empírica haciendo uso de una estrategia exhaustiva. Para cada par (***n***, ***nb***), se representa el rendimiento obtenido con el mejor valor de ***ib***. Asimismo, también se observa que el rendimiento varía en función del tamaño de bloque (***nb***) utilizado, por tanto, la selección adecuada del valor de dichos parámetros algorítmicos es fundamental si se quiere optimizar el rendimiento de la rutina. La figura 6.8 (arriba derecha) muestra, para cada tamaño de problema, el par de valores (***nb***, ***ib***) con los que se obtiene el mejor rendimiento. En el caso de ***nb***, se observa que para tamaños de problema mayores a 8000, el mejor valor corresponde siempre al mayor del conjunto utilizado (576); en cambio, en el caso de ***ib*** los valores seleccionados son pequeños y varían entre 16 y 56 para los tamaños de bloque y de problema utilizados. En la figura 6.8 (abajo) se puede observar el tiempo de instalación (en segundos) empleado con cada tamaño de problema usando esta estrategia de búsqueda, siendo en total de 6 horas y 10 minutos.

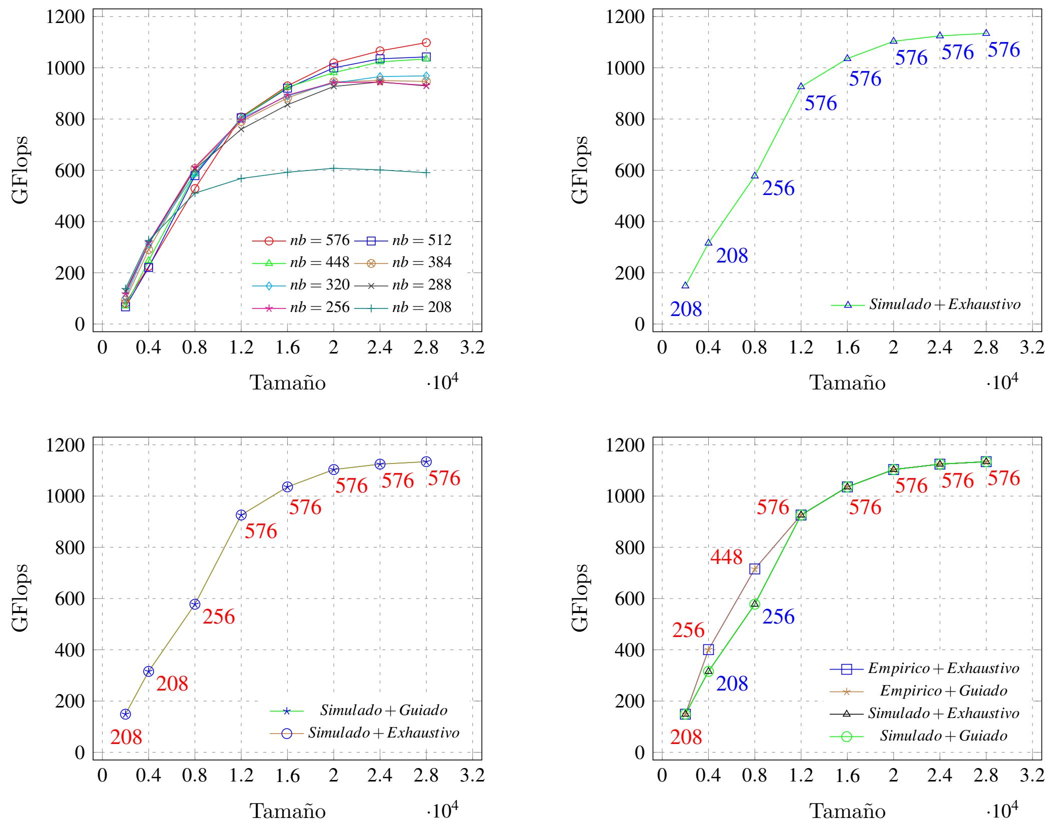

Figura 6.7: Estimación de las prestaciones obtenidas con la rutina de descomposición de Cholesky usando un enfoque simulado combinado con una estrategia exhaustiva (arriba); comparación de las prestaciones obtenidas con ambas estrategias (exhaustiva y guiada) usando el mejor valor de nb para cada tamaño de problema (abajo izquierda), y comparación de las prestaciones obtenidas con el mejor valor de nb para cada tamaño de problema usando ambos enfoques (empírico y simulado) con las citadas estrategias (abajo derecha).

La figura 6.9 (izquierda) muestra el rendimiento obtenido por la factorización LU con la mejor configuración de valores (nb, ib) para cada tamaño de problema. La rutina se ha instalado de forma empírica haciendo uso de la metodología de optimización con una búsqueda guiada en el valor de ib (Guiada_1D). La figura 6.9 (derecha), por su parte, muestra el tiempo de instalación (en segundos) empleado para cada tamaño de problema. En total, el tiempo requerido es de 32 minutos, frente a las 6 horas utilizadas con la estrategia exhaustiva. Este tiempo se puede reducir aún más si se aplica la metodología de optimización haciendo uso de la otra estrategia de búsqueda guiada (Guiada_2D). Esto se puede observar en la

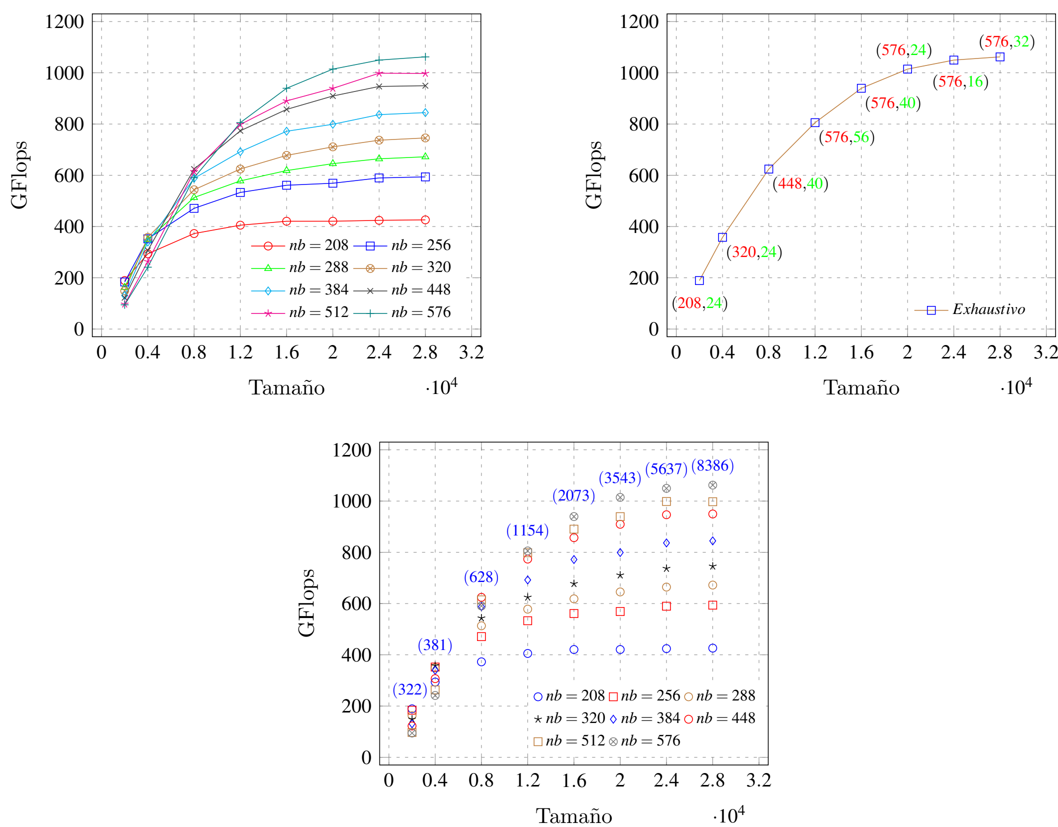

Figura 6.8: Prestaciones obtenidas al instalar la rutina de factorización LU de forma exhaustiva con un conjunto arbitrario de tamaños de bloque (arriba izquierda); prestaciones obtenidas con la mejor combinación (*nb*, *ib*) para cada tamaño de problema (arriba derecha), y tiempo de instalación empleado (en segundos) con cada tamaño de problema (abajo).

figura 6.10 (derecha). El tiempo total empleado es de tan sólo 6 minutos. La 6.10 (izquierda), por su parte, muestra el rendimiento obtenido usando dicha estrategia con la mejor configuración de valores (*nb*, *ib*) para cada tamaño de problema. Finalmente, en la figura 6.10 (abajo), se muestran las prestaciones obtenidas con esta rutina con cada una de las tres estrategias de búsqueda (Exhaustiva, Guiada_1D y Guiada_2D). Se observa que el rendimiento obtenido con las estrategias de búsqueda guiada prácticamente coincide con el obtenido de forma exhaustiva, lo que demuestra, de nuevo, que el uso de este tipo de técnicas permite obtener resultados satisfactorios y con un tiempo de instalación menor, especialmente si se aplica de forma simultánea en la búsqueda del valor de todos los parámetros algorítmicos, como en el caso de la estrategia Guiada_2D.

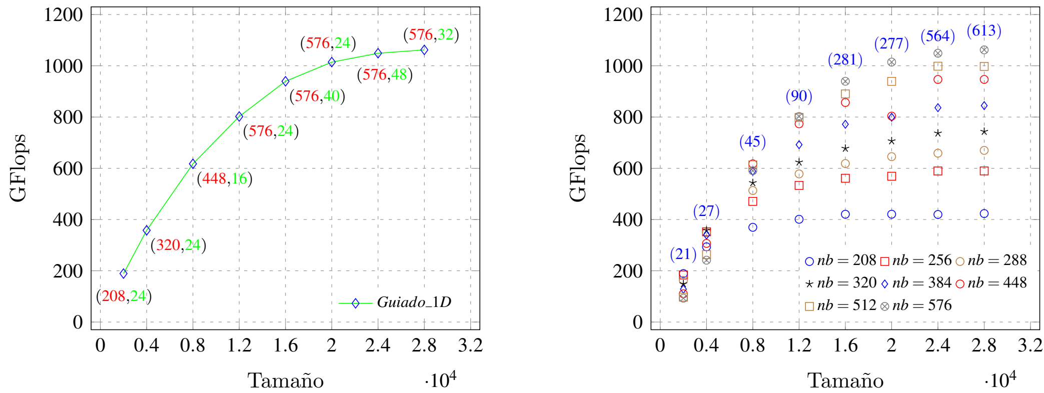

Figura 6.9: Prestaciones obtenidas con la mejor combinación (***nb***, ***ib***) seleccionada para cada tamaño de problema durante la instalación de la rutina de factorización LU utilizando la estrategia de búsqueda **Guiada_1D** (izquierda), y tiempo de instalación (en segundos) empleado para cada tamaño de problema (derecha).

La tabla 6.1 resume los valores obtenidos para ***nb*** e ***ib*** con cada una de las estrategias de búsqueda utilizadas. En algunos tamaños de problema se producen pequeñas variaciones en la selección del valor de estos parámetros, pero esto no afecta en gran medida al rendimiento obtenido, que es muy similar con cada una de ellas.

	Exhaustiva			**Guiada_1D**			**Guiada_2D**		
n	*nb*	*ib*	*GFlops*	*nb*	*ib*	*GFlops*	*nb*	*ib*	*GFlops*
2000	208	24	189	208	24	189	208	24	189
4000	320	24	358	320	24	358	256	24	352
8000	448	40	624	448	16	618	448	16	618
12000	576	56	806	576	24	802	576	24	802
16000	576	40	940	576	40	940	576	24	939
20000	576	24	1015	576	24	1015	576	24	1015
24000	576	16	1050	576	48	1049	576	16	1050
28000	576	32	1062	576	32	1062	576	32	1062

Tabla 6.1: Prestaciones obtenidas por la rutina de factorización LU con los mejores valores de los parámetros algorítmicos (***nb*** e ***ib***) seleccionados con cada una de las estrategias de búsqueda para cada tamaño de problema.

Asimismo, se ha llevado a cabo el estudio de las prestaciones obtenidas por la factorización LU haciendo uso del simulador SimGrid. La figura 6.11 compara los resultados obtenidos utilizando los enfoques empírico y simulado en combinación con la estrategia Guiada_2D. Se observa que el rendimiento estimado por el

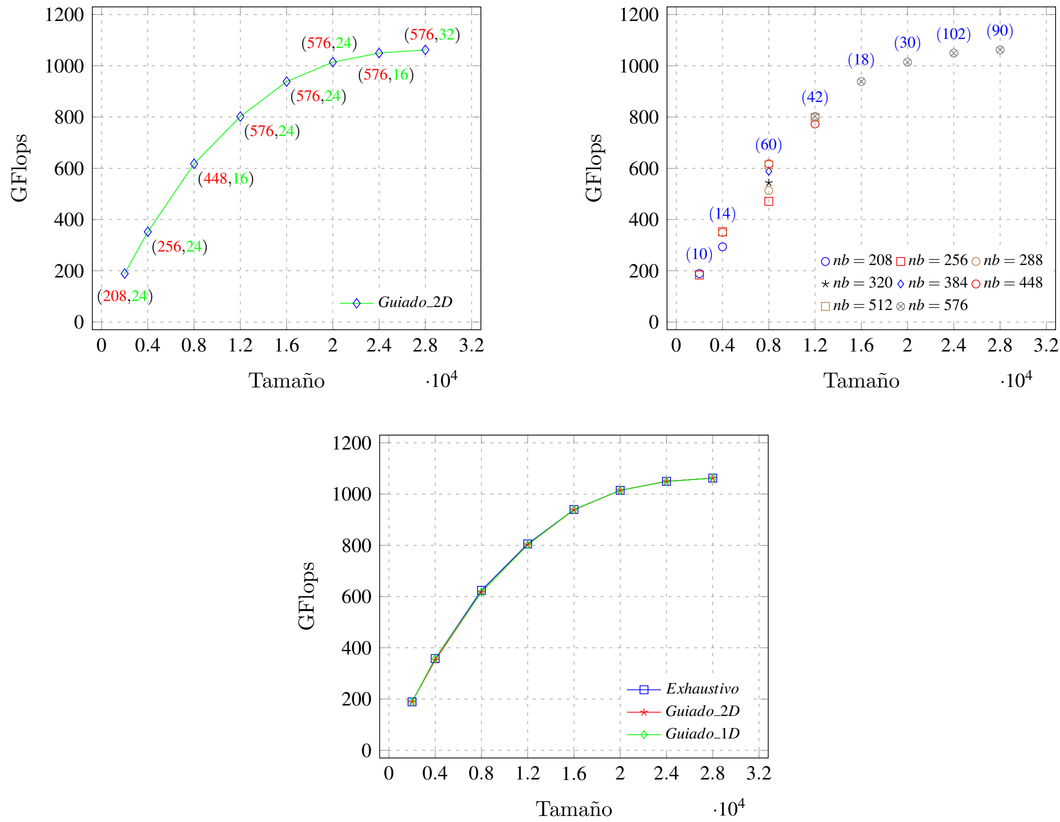

Figura 6.10: Prestaciones obtenidas con la mejor combinación (*nb*, *ib*) seleccionada para cada tamaño de problema durante la instalación de la rutina de factorización LU utilizando la estrategia de búsqueda **Guiada_2D** (arriba izquierda); tiempo de instalación (en segundos) empleado con cada tamaño de problema (arriba derecha), y comparativa de las prestaciones obtenidas con las tres estrategias de búsqueda (abajo).

simulador para cada tamaño de problema es satisfactorio, pues en algunos casos es prácticamente similar al obtenido de forma empírica y, en otros, se encuentra próximo al mismo.

Hasta ahora, el estudio se ha centrado en optimizar la ejecución de rutinas, como la descomposición de Cholesky o la factorización LU, mediante la aplicación de un conjunto de estrategias de búsqueda enfocadas a la selección de los valores de los parámetros algorítmicos que afectan a su rendimiento. No obstante, existen otra serie de parámetros que deben ser analizados si se quiere seguir reduciendo el tiempo de ejecución. Estos parámetros pueden ser propios del sistema (como el número de cores de CPU o el número y tipo de GPUs a usar) o de la librería

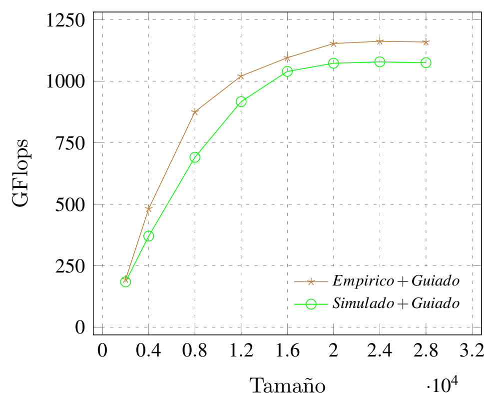

Figura 6.11: Comparativa de las prestaciones obtenidas en **jupiter** de forma empírica y simulada con la rutina de factorización LU utilizando en cada caso la estrategia de búsqueda Guiada_2D.

(como la política de planificación utilizada por el planificador de tareas presente en Chameleon). En las siguientes secciones se analiza el impacto de estos parámetros en el rendimiento de dichas rutinas.

6.3.4 Selección de las unidades de cómputo

Tal y como se ha comentado, además de los parámetros algorítmicos propios de las rutinas (como el tamaño de bloque), se puede analizar la influencia que tiene en el rendimiento la selección adecuada del valor de determinados parámetros del sistema, como el número de unidades de cómputo a usar. En el estudio realizado se consideran tanto el número de cores de CPU como el número de GPUs presentes en el nodo. El algoritmo de optimización que se aplica consiste en ir añadiendo, de forma sucesiva, unidades de cómputo en función de su capacidad computacional, siempre y cuando se mejore el rendimiento obtenido hasta el momento. Para la obtención del valor de los parámetros algorítmicos de la rutina, se utiliza la metodología de optimización con búsqueda guiada bidireccional descrita en la sección anterior. De esta forma, en cada paso del algoritmo se utilizará, como punto de partida para la configuración hardware actual, el mejor valor de ***nb*** obtenido para la configuración anterior.

La tabla 6.2 muestra los resultados obtenidos para la rutina de Cholesky en **jupiter**. Se observa cómo una adecuada selección de las unidades de cómputo del nodo, permite obtener una mejora en el rendimiento entre un 10 % y un 40 % respecto a la que obtendría la librería con la ejecución por defecto. Esto se debe a que el planificador asigna las tareas a las unidades de cómputo en tiempo de ejecución siguiendo un esquema basado en prioridades y dependencias entre los datos requeridos por los diferentes kernels computacionales, pero no contempla la capacidad computacional de las unidades de cómputo.

n	nb	CPU_Cores	$GPUs$	$Tuned$	$Default$	%
1000	112	12	$\{-\}$	76	46	40
2000	192	9	$\{1,5,0\}$	164	117	29
3000	192	8	$\{1,5,0,2\}$	285	196	31
4000	240	7	$\{1,5,0,2,3\}$	412	352	15
5000	256	6	$\{1,5,0,2,3,4\}$	545	465	15
6000	256	6	$\{1,5,0,2,3,4\}$	626	554	12
7000	320	6	$\{1,5,0,2,3,4\}$	687	608	12
8000	320	6	$\{1,5,0,2,3,4\}$	753	682	10

Tabla 6.2: Prestaciones obtenidas en **jupiter** con la rutina de descomposición de Cholesky con los valores de los parámetros seleccionados por la metodología de auto-optimización (***Tuned***) para cada tamaño de problema y con la configuración por defecto de la plataforma (***Default***), así como el porcentaje de mejora (**%**) obtenido con auto-optimización.

6.3.5 Selección de la política de planificación

En una librería como Chameleon, donde la ejecución de los kernels computacionales de las rutinas de álgebra lineal se lleva a cabo mediante un planificador dinámico de tareas en tiempo de ejecución, la selección adecuada de la política de planificación puede llevar a una reducción del tiempo de ejecución de dichas rutinas. StarPU [20], el sistema de planificación utilizado por esta librería, dispone de diferentes políticas de planificación. Cada una de estas políticas, establece el momento en que los kernels pasan a ejecutarse en las diferentes unidades de cómputo de la plataforma. Por tanto, en función del criterio adoptado, la rutina se ejecutará con mayor o menor eficiencia. Algunas políticas, utilizan la información de los modelos de rendimiento de las rutinas *codelets* para decidir cómo planificar

las tareas. Otras, en cambio, se basan en prioridades o en la disponibilidad de las unidades de cómputo. Todo esto da lugar a que este parámetro del sistema pueda ser considerado como un parámetro más a seleccionar durante la aplicación de la metodología de optimización. A continuación, se indican las características de las políticas de planificación consideradas en el estudio realizado:

- **eager**: usa una cola central de tareas y no permite la búsqueda de datos por adelantado (*prefetching*). Si una tarea tiene una prioridad $\neq 0$, se sitúa al comienzo de la cola.
- **random**: utiliza una cola por *worker* y distribuye las tareas de forma aleatoria entre ellos de acuerdo al rendimiento máximo soportado por cada uno. Para StarPU, un *worker* corresponde a una unidad básica de procesamiento (cada core de la CPU o una GPU) que permite ejecutar una operación de cómputo básica de álgebra lineal.
- **ws** (*work stealing*): utiliza una cola por *worker*. Cuando éste se queda ocioso, roba una tarea de la cola del *worker* con más carga.
- **lws** (*locality work stealing*): utiliza también una cola por *worker*, pero cuando éste se queda ocioso, roba una tarea de la cola de los vecinos. También tiene en cuenta prioridades.
- **dm** (*deque model*): tiene en cuenta los modelos de rendimiento de la tarea para su ejecución. Planifica las tareas tan pronto como están disponibles, sin tener en cuenta prioridades.
- **dmda** (*deque model data aware*): similar al anterior, pero además, tiene en cuenta el tiempo de transferencia de datos.

Los experimentos realizados muestran cómo varía el rendimiento de la rutina de Cholesky cuando se cambia la política de planificación utilizada por medio de la variable de entorno `STARPU_SCHED`. Se parte del hecho de que la rutina ya ha sido instalada en el sistema con cada una de las políticas (figura 6.12) y, por tanto, se dispone de la información de instalación almacenada. Esta instalación se ha realizado utilizando el mismo conjunto de tamaños de problema y de bloque que

en apartados anteriores. Por tanto, una primera aproximación podría consistir en utilizar la metodología de optimización para seleccionar el mejor valor del tamaño de bloque a usar para cada tamaño de problema con cada política de planificación (figura 6.13 (izquierda)). La figura 6.13 (derecha), por su parte, compara el rendimiento que se obtendría al aplicar la metodología de optimización seleccionando, además, la mejor política de planificación de entre todas las utilizadas, con el que se obtendría al utilizar directamente la política por defecto (*lws*). Se observa que el rendimiento obtenido al aplicar la metodología es ligeramente superior para todos los tamaños de problema, pero no se distancia mucho del obtenido con la política por defecto, ya que ésta proporciona un correcto balanceo de la carga entre las tareas incluso si no se dispone de los modelos de rendimiento de los kernels computacionales de las rutinas.

Finalmente, la tabla 6.3 muestra el rendimiento obtenido con los valores de los parámetros (algorítmicos y del sistema) seleccionados para cada tamaño de problema, tras aplicar la metodología de optimización con búsqueda guiada considerando todos los parámetros analizados en el estudio realizado en este capítulo: tamaño de bloque, unidades de cómputo y política de planificación. Se observa que tanto el planificador como el número de unidades de cómputo a usar y el tamaño de bloque seleccionado, difieren para cada tamaño de problema, de ahí la importancia de disponer de una metodología que permita seleccionar, en cada caso, los mejores valores a usar.

n	*Politica*	*nb*	*CPU_Cores*	*GPUs*	*GFlops*
2000	dmda	192	10	{1,5}	169
4000	dmda	256	7	{1,5,0,2,3}	403
8000	dmda	320	6	{1,5,0,2,3,4}	719
12000	dmda	576	6	{1,5,0,2,3,4}	896
16000	dmda	576	6	{1,5,0,2,3,4}	1010
20000	ws	672	6	{1,5,0,2,3,4}	1051
24000	ws	672	6	{1,5,0,2,3,4}	1120
28000	ws	672	6	{1,5,0,2,3,4}	1150

Tabla 6.3: Prestaciones obtenidas en **jupiter** con la rutina de descomposición de Cholesky al utilizar, para cada tamaño de problema, la mejor política de planificación junto con el mejor valor de *nb* y la mejor configuración hardware (cores de CPU y GPUs).

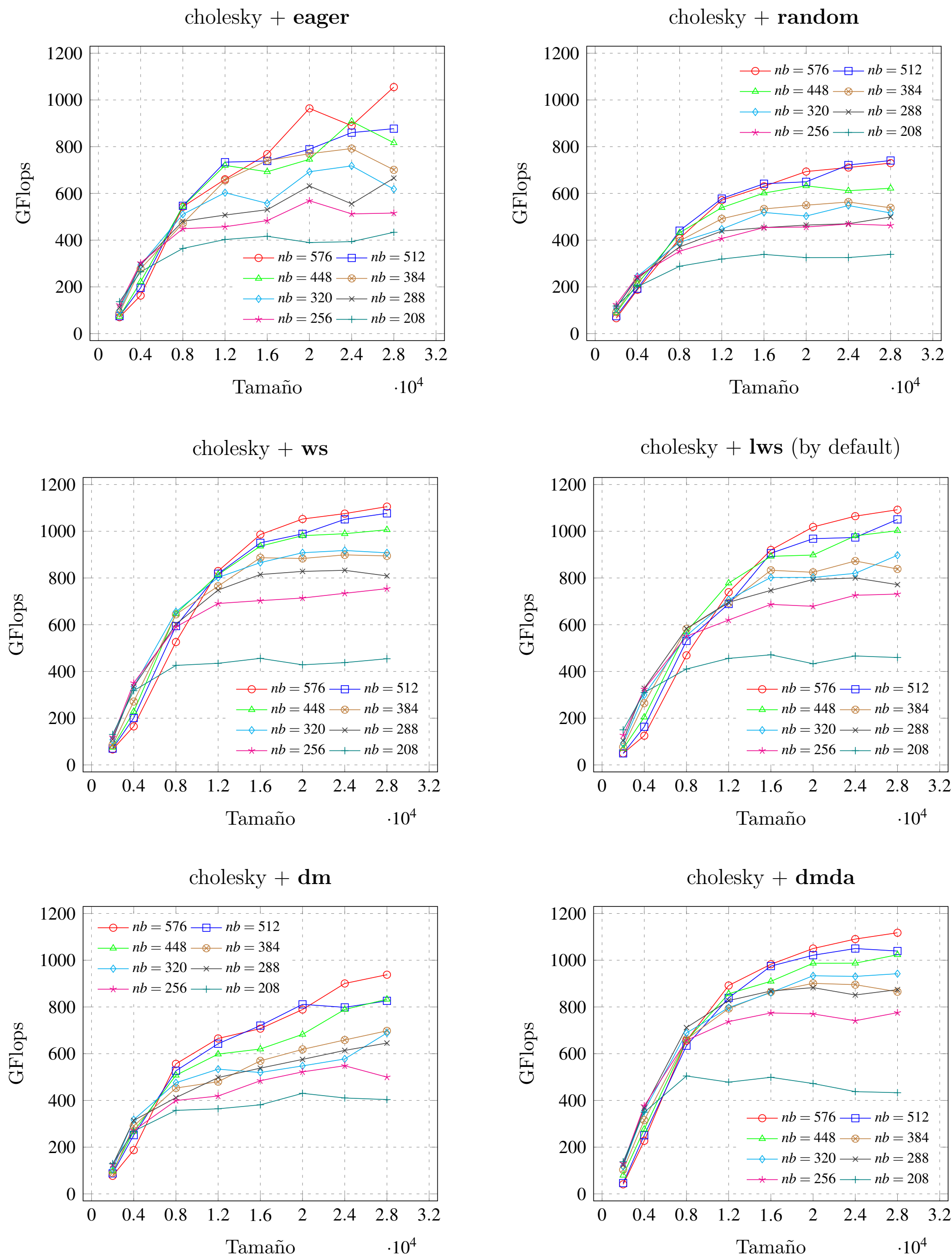

Figura 6.12: Prestaciones obtenidas por la rutina de descomposición de Cholesky con diferentes políticas de planificación.

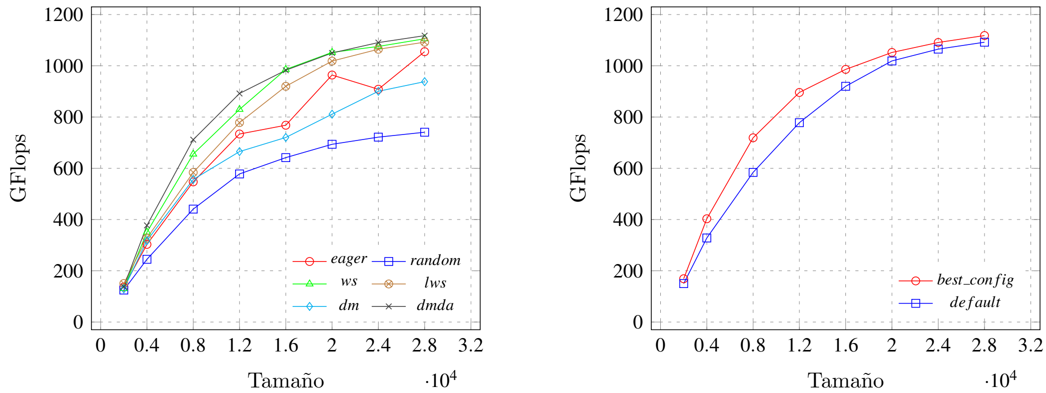

Figura 6.13: Prestaciones obtenidas por la rutina de descomposición de Cholesky al usar en cada política de planificación el mejor valor de ***nb*** para cada tamaño de problema (izquierda), y comparación de las prestaciones obtenidas usando la mejor política de planificación con el mejor valor de ***nb*** para cada tamaño de problema o usando la política de planificación por defecto (derecha).

6.4 Conclusiones

En este capítulo se ha mostrado cómo extender la metodología de auto-optimización propuesta para permitir el uso de librerías basadas en tareas, como Chameleon. Se han presentado diferentes estrategias de búsqueda de los valores de los parámetros algorítmicos de las rutinas y se ha aplicado la metodología de forma empírica y mediante el uso del simulador SimGrid.

Por un lado, se ha analizado el impacto que tiene en el rendimiento el valor de determinados parámetros algorítmicos como el tamaño de bloque, ***nb***, o el tamaño de bloque interno, ***ib***, en rutinas como la rutina de descomposición de Cholesky o la factorización LU, respectivamente. Los resultados obtenidos con cada una de las estrategias de búsqueda propuestas, tanto con un enfoque empírico como simulado, han permitido obtener buenas prestaciones, reduciendo incluso el tiempo de instalación de las rutinas guiando la búsqueda de los valores de dichos parámetros. Por otro lado, se han considerado parámetros adicionales, como el número de unidades de cómputo a usar o la política de planificación a establecer, y se ha aplicado de nuevo la metodología de optimización, demostrando la importancia que tiene seleccionar adecuadamente el valor de cada uno de ellos para reducir aún más el tiempo de ejecución de las rutinas.

De esta forma, queda patente la validez de la metodología de auto-optimización propuesta, así como la versatilidad que ofrece su diseño para adaptarse a cualquier tipo de sistemas hardware o software.

Capítulo 7

Conclusiones y Trabajo Futuro

En este capítulo se comentan las principales conclusiones obtenidas y las líneas de trabajo futuro a seguir, así como las publicaciones derivadas de la realización de la presente tesis doctoral.

7.1 Conclusiones

La gran variedad de sistemas computacionales existentes, así como de librerías que ofrecen implementaciones de rutinas de álgebra lineal utilizadas en códigos científicos para el cálculo de operaciones complejas, ha dado lugar al constante desarrollo de técnicas de optimización enfocadas a reducir el tiempo de ejecución de dichas rutinas en función del tipo de sistema computacional empleado.

La necesidad de disponer de un método genérico y flexible que permita paliar las posibles deficiencias de las técnicas de optimización desarrolladas hasta el momento, ha llevado al desarrollo de la metodología de auto-optimización jerárquica propuesta en esta tesis. Esta metodología puede ser aplicada en diferentes niveles hardware y software, lo que permite, por un lado, que la ejecución de las rutinas pueda ser optimizada para cualquier tipo de sistema (CPU multicore, CPU+GPU, CPU+MIC, CPU+multicoprocesadores..., combinación de los anteriores en nodos de cómputo o uso de todos los nodos como una sola unidad de cómputo (cluster heterogéneo)) y por otro, que dicho proceso de optimización se

pueda llevar a cabo de forma jerárquica entre rutinas de diferentes niveles software, permitiendo que rutinas de nivel superior (como la multiplicación de Strassen o la factorización LU) puedan ejecutarse utilizando versiones optimizadas de las rutinas matriciales básicas a las que invocan (como la multiplicación de matrices). Existen librerías de álgebra lineal que ofrecen implementaciones eficientes de estas rutinas básicas para cada tipo de unidad de procesamiento paralelo del sistema, por tanto, la metodología puede hacer uso de las mismas (como MKL para CPU o cuBLAS para GPU) para reducir, aún más, el tiempo de ejecución.

La optimización de las rutinas no sólo depende de la librería utilizada, sino que también es necesario determinar, en los diferentes niveles de la jerarquía, los parámetros algorítmicos y del sistema que afectan a su rendimiento. Estos parámetros pueden ser diferentes en función del nivel hardware y software considerado. Algunos de ellos son característicos del sistema utilizado (como el número de threads en CPU o Xeon Phi), otros son propios de las rutinas (como el nivel de recursión o el tamaño de bloque) y otros dependen del nivel en el que se aplique la metodología (como el reparto de la carga entre las unidades de cómputo). De ahí el interés por utilizar una metodología de auto-optimización que sea capaz de contemplar los diferentes escenarios. Esto va a permitir instalar las rutinas por niveles, considerando en cada nivel los parámetros que afectan al rendimiento. Como resultado, se dispondrá de un conjunto de rutinas instaladas en diferentes niveles de la jerarquía. De esta forma, la ejecución de rutinas en niveles superiores de la jerarquía invocando a rutinas previamente instaladas, se llevará a cabo de forma optimizada con el mejor valor de los parámetros en cada nivel y con un uso eficiente de las unidades computacionales del sistema.

El funcionamiento de la metodología propuesta se ha ilustrado realizando un estudio por niveles (tanto hardware como software), considerando diferentes escenarios (sistemas y rutinas) y parámetros en cada nivel. Los experimentos se han llevado a cabo en una plataforma computacional heterogénea compuesta por nodos con distinto número y tipo de unidades de procesamiento paralelo, mostrando la instalación de rutinas básicas (multiplicación de matrices) y de nivel medio (multiplicación de Strassen y factorización LU) en diferentes niveles hardware, haciendo uso de la información de instalación almacenada en niveles inferiores de la jerarquía para optimizar su ejecución. Los resultados obtenidos

son satisfactorios y muestran la efectividad de aplicar la metodología de auto-optimización siguiendo un enfoque jerárquico en cada uno de los niveles de las dimensiones hardware y software.

Asimismo, se ha extendido la funcionalidad de la metodología para hacer uso de herramientas software destinadas a la búsqueda del valor de los parámetros algorítmicos, como OpenTuner. Se ha mostrado cómo se pueden utilizar las técnicas metaheurísticas ofrecidas por dicho software para determinar el valor de los parámetros en cada nivel. El estudio se ha realizado con la rutina de multiplicación de matrices en un nodo heterogéneo, estableciendo como límite para el tiempo de búsqueda de los parámetros, el correspondiente a diferentes formas de realizar la instalación jerárquica. Las prestaciones que se obtienen con dicha herramienta son satisfactorias con los valores que selecciona, pero el uso de la metodología jerárquica en combinación con una búsqueda guiada, muestra que es posible obtener prestaciones ligeramente superiores y con un tiempo de búsqueda menor. No obstante, esta herramienta podría ser considerada para su aplicación en determinados niveles de la jerarquía.

Por otro lado, también se ha extendido la metodología para permitir su uso con librerías basadas en tareas, como es el caso de Chameleon. Se ha mostrado su aplicación con determinadas rutinas de álgebra lineal de esta librería, adoptando diferentes estrategias para la búsqueda de parámetros algorítmicos de las mismas (como el tamaño de bloque), así como parámetros del sistema (como el número de unidades de cómputo a usar) y parámetros propios de la librería (como la política de planificación utilizada por el planificador de tareas). El estudio se ha llevado a cabo sobre un nodo heterogéneo utilizando la rutina de descomposición de Cholesky y la factorización LU, adoptando tanto un enfoque empírico como simulado. Los resultados obtenidos tras aplicar la metodología de auto-optimización con diferentes estrategias de búsqueda, permiten obtener prestaciones similares a las óptimas experimentales, con una buena selección del valor de sus parámetros algorítmicos.

Así pues, el estudio realizado deja constancia tanto de la validez de la metodología de auto-optimización propuesta, como de su capacidad para adaptarse a cualquier tipo de sistema computacional y de ser extendida con otras rutinas, librerías y técnicas de selección de los valores de los parámetros (algorítmicos

y del sistema). Finalmente, cabe mencionar que toda la funcionalidad descrita se ha implementado en un software de instalación y optimización jerárquica. El Anexo A contiene una primera versión del manual de usuario y de desarrollador. Ambos se irán actualizando conforme se publiquen nuevas versiones del software, y estarán disponibles junto a éste para su libre descarga desde la página web http://luna.inf.um.es/grupo_investigacion/software.

7.2 Difusión de Resultados

En esta sección se especifican los congresos a los que se ha asistido durante el período de formación predoctoral, así como las publicaciones que han tenido lugar durante dicha etapa como resultado del trabajo desarrollado en esta tesis.

Cabe destacar determinadas publicaciones, tanto en Proceedings de congresos como en Revistas, que tuvieron lugar con anterioridad al inicio de esta tesis y que constituyen el punto de partida de la misma. Estas publicaciones se resumen en el Capítulo 3, y aportan resultados de investigación sobre el estudio inicial de técnicas de optimización de rutinas de álgebra lineal en plataformas computacionales precursoras de las actuales. A continuación se indica cada una de ellas:

Jesús Cámara, Javier Cuenca, Domingo Giménez, Antonio M. Vidal. Empirical Autotuning of Two-level Parallel Linear Algebra Routines on Large cc-NUMA Systems. *In Proceedings of the IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ISPA'12*, pages 843-844, 07 2012. Congreso CORE B (clase 3 conforme al GGS).

Jesús Cámara, Javier Cuenca, Luis-Pedro García, Domingo Giménez. Empirical Modelling of Linear Algebra Shared-Memory Routines. *In Proceedings of the 2013 International Conference on Computational Science, ICCS*, pages 110-119, 2013. Congreso CORE A (clase 3 conforme al GGS).

Jesús Cámara, Javier Cuenca, Luis-Pedro García, Domingo Giménez. Auto-tuned nested parallelism: A way to reduce the execution time of scientific software in NUMA systems. *Parallel Computing*, 40(7):309-327, 2014. Categoría Q1 en JCR.

Jesús Cámara, Javier Cuenca, Domingo Giménez, Luis-Pedro García, Antonio M. Vidal. Empirical Installation of Linear Algebra Shared-Memory Subroutines for Auto-Tuning. *International Journal of Parallel Programming*, 42(3):408-434, 06 2014. Categoría Q4 en JCR.

Asimismo, durante la realización de la tesis he tenido la oportunidad de asistir a diferentes congresos de carácter internacional para presentar, bien mediante comunicaciones orales o en formato póster, los resultados de investigación que se han ido obteniendo y que constituyen el núcleo de la labor investigadora realizada en esta tesis. La asistencia a los mismos se ha justificado dentro del proyecto de investigación al que está asociado el contrato predoctoral (en modalidad de beca FPI) del que he disfrutado. A continuación se detalla cada uno de ellos, indicando el lugar y la fecha de realización, así como el trabajo presentado y el capítulo de la tesis en el que quedaría enmarcado:

- 18th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2018), Cádiz, 9-14 Julio, 2018.
  - Contribución: *A Hierarchical Approach for Autotuning Linear Algebra Routines on Heterogeneous Clusters* (publicada en *Proceedings*).
  - Formato de la Presentación: Comunicación Oral.
  - Enmarcado en el Capítulo 4 de la Tesis. Presenta las ideas iniciales de la metodología de optimización jerárquica con un ejemplo de aplicación.
- XXIX Jornadas de Paralelismo (JP2018) de la Sociedad de Arquitectura y Tecnología de Computadores (SARTECO), Teruel, 12-14 Septiembre, 2018.
  - Contribución: *Optimización Automática de Rutinas de Álgebra Lineal en Clusters Heterogéneos: Un Enfoque Jerárquico.*
  - Formato de la Presentación: Comunicación Oral.
  - Enmarcado en el Capítulo 4 de la Tesis.
- 19th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2019), Cádiz, 1-6 Julio, 2019.
  - Contribución: *Hierarchical Automatic Optimization of High and Medium Level Linear Algebra Routines* (publicada en *Proceedings*).

  - Formato de la Presentación: Comunicación Oral.
  - Enmarcado en los capítulos 4 y 5 de la Tesis. Extiende el estudio realizado con la metodología de optimización jerárquica, mostrando ejemplos en diferentes niveles de la jerarquía hardware y software.

- International Conference on Parallel Computing (ParCo 2019), Praga, República Checa, 10-13 Septiembre, 2019. Congreso CORE C de clase 3 (*events of good quality*) conforme al GII-GRIN-SCIE (GGS).

  - Contribución: *On the Autotuning of Task-based Numerical Libraries for Heterogeneous Architectures* (publicada en *Proceedings* [3]).
  - Formato de la Presentación: Comunicación Oral.
  - Enmarcado en el Capítulo 6 de la Tesis. Presenta los resultados de investigación obtenidos durante la estancia realizada en Burdeos en el Instituto Nacional de Investigación en Informática y Automática (INRIA Bordeaux Sud-Ouest).

- SIAM Conference on Parallel Processing for Scientific Computing (PP20), Seattle, Washington, U.S., 12-15 Febrero, 2020. Congreso clasificado como *work in progress* según el GII-GRIN-SCIE (GGS).

  - Contribución: *Hybrid Empirical and Simulation-Based Autotuning of a Dense Linear Algebra Library for Heterogeneous Architectures.*
  - Formato de la Presentación: Comunicación Oral.
  - Enmarcado en el Capítulo 6 de la Tesis. Extiende los resultados de investigación presentados al Congreso ParCo 2019.

Finalmente, el trabajo realizado ha culminado con la publicación de un artículo en una revista de carácter científico e internacional, dentro del área de conocimiento en la que se enmarca la línea de investigación de la presente tesis:

Jesús Cámara, Javier Cuenca, Domingo Giménez. Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters. *The Journal of Supercomputing*, 03 2020. Categoría Q2 en JCR.

En esta publicación, se abordan los contenidos del núcleo de la tesis (capítulos 4 y 5) y se muestran los principales resultados de investigación obtenidos con la aplicación de la metodología de auto-optimización jerárquica propuesta.

Por otro lado, también se ha llevado a cabo la difusión de resultados preliminares de una de las tareas que se pretende abordar dentro de las líneas de trabajo futuro. La presentación se llevó a cabo, en formato póster, en el Congreso SIAM PP20 anteriormente citado. Este trabajo, titulado: *Adapting a Multibody System Simulator to Auto-Tuning Linear Algebra Routines*, propone cómo integrar la metodología de auto-optimización jerárquica en un software para la simulación de sistemas multicuerpo en el ámbito de la ingeniería mecánica.

Así pues, la difusión de resultados por medio de las contribuciones mencionadas, ha permitido aportar el soporte necesario para poder justificar el interés y la validez científica de las tareas de investigación llevadas a cabo en la presente tesis doctoral.

7.3 Trabajo Futuro

La labor de investigación realizada en esta tesis ha permitido establecer una base sólida sobre la que poder continuar el trabajo iniciado con la metodología de auto-optimización jerárquica propuesta. El objetivo es seguir avanzando en la misma línea, extendiendo la metodología con nueva funcionalidad y considerando su inclusión en software de mayor nivel que pueda hacer uso de ésta para conseguir una ejecución optimizada. Así pues, se podría establecer la siguiente clasificación en función del tipo de tareas a realizar y el orden en el que se pretenden conseguir:

- Incorporar nuevas funciones en la metodología de optimización jerárquica:
  - Extender la funcionalidad de la metodología con nuevos parámetros, rutinas y librerías de álgebra lineal.
  - Combinar la metodología con técnicas de modelado teórico del tiempo de ejecución de las rutinas.
  - Integrar OpenTuner como parte del motor de auto-optimización de la metodología, con el fin de poder hacer uso de las heurísticas de

búsqueda que ofrece para la selección de los valores de los parámetros algorítmicos de cada nivel de la jerarquía.
  - Adaptar la metodología para dar soporte a optimización bi-objetivo, tanto a nivel de rendimiento como de consumo de energía.

- Generar una versión estable y robusta del software desarrollado que incluya la nueva funcionalidad implementada.

- Extender el uso de la metodología jerárquica en software científico:

  - Integrar la metodología de optimización como un componente software dentro de la librería Chameleon.
  - Combinar la versión de Chameleon con auto-optimización con el simulador SimGrid, de forma que permita la resolución de problemas científicos con alta demanda computacional mediante el uso de los modelos de rendimiento optimizados de las rutinas de álgebra lineal y sin necesidad de requerir el uso de la propia plataforma computacional.
  - Utilizar la metodología de optimización jerárquica para la resolución eficiente de problemas de cinemática computacional en el área de ingeniería mecánica. Actualmente, se está trabajando en la integración de la metodología de auto-optimización en un software para la simulación de sistemas multicuerpo. Los resultados obtenidos hasta el momento fueron presentados, en formato póster, en el Congreso SIAM PPSC20.

Asimismo, se espera que los resultados que se deriven de los avances producidos en cada una de las líneas de trabajo futuro establecidas, puedan ser divulgados (mediante asistencia a congresos o publicación en revistas) y aplicados de forma efectiva en la resolución de problemas científicos y de ingeniería.

Conclusions

The variety of computing systems and libraries with implementations of linear algebra routines used in scientific codes for solving complex problems, has led to the development of optimization techniques focused on reducing the execution time of the routines depending on the type of computing system where they run.

In this Thesis, a generic and flexible methodology for auto-tuning parallel linear algebra routines on heterogeneous platforms is presented. This methodology can be applied at different hardware and software levels, allowing the optimization of the routines for multiple types of systems (multicore CPU, CPU+GPU, CPU+MIC, CPU+multicoprocessors, and heterogeneous clusters), and with routines at different levels of the software hierarchy, for example, the matrix multiplication at the lowest level and higher level routines (such as Strassen multiplication or LU factorization) executed by using optimized versions of the matrix multiplication. There are linear algebra libraries that include efficient implementations of these basic routines for each type of parallel processing unit in the system (for example, MKL for multicore or CUBLAS for GPU), so the methodology can make use of them to further reduce the execution time.

The optimization of routines does not only depend on the library used, but it is also necessary to determine, at the different levels of the hierarchy, the algorithmic and system parameters that affect their performance. These parameters can be different depending on the hardware and software levels considered. Some of them are characteristic of the system (for example, the number of threads in CPU or Xeon Phi), others are specific to the routines (for example, the recursion level or the block size) and others depend on the level at which the methodology is applied (for example, the workload distribution among the computing units). Therefore,

the auto-tuning methodology must consider the different scenarios. The routines are installed by levels, taking into account the parameters that affect performance at each level. As a result, there will be a set of routines installed at different levels of the hierarchy. In this way, the execution of routines in higher levels of the hierarchy is carried out in an optimized way by calling the routines previously installed at lower levels, which eases the efficient use of the computing units of the system.

To illustrate how the proposed methodology works, a study by levels is carried out for both the hardware and the software, considering different scenarios (systems and routines) and parameters at each level. The experiments have been carried out in a heterogeneous cluster composed of nodes with different number and type of parallel processing units (CPU+GPU, CPU+multiGPU, CPU+MIC, CPU+multiMIC and CPU+GPU+multiMIC), analysing the installation of basic routines (matrix multiplication) and medium level routines (Strassen multiplication and LU factorization) at different hardware levels by using the information stored in lower levels of the hierarchy. The results obtained are satisfactory (both in the hardware and the software hierarchy levels) and show the effectiveness of using the hierarchical auto-tuning approach.

The functionality of the methodology has been extended to consider the use of software tools for the search of the best values of the algorithmic parameters (OpenTuner). In this way, instead of using the information stored at lower levels of the hierarchy, the metaheuristic techniques offered by this software are used directly to determine the value of the parameters at each level. The study has been carried out with the matrix multiplication routine in a heterogeneous node, with limits for the searching time of the parameters corresponding to different ways of doing the hierarchical installation. The performance obtained with this software is satisfactory, but the use of the hierarchical methodology in combination with a guided search provides slightly higher performance with a shorter search time. Nevertheless, this software could be considered for application at certain levels of the hierarchy.

The methodology has also been extended to use numerical task-based libraries (Chameleon). Its application has been analysed with the Cholesky and LU routines of this library. The study has been carried out on a heterogeneous node using

both an empirical (or experimental) and simulated approaches in combination with several strategies for searching the values of different types of parameters: algorithmic parameters (block size), system parameters (number of computing units to be used) and parameters of the library itself (the scheduling policy used by the task scheduler). The results obtained with the auto-tuning methodology are satisfactory, with a good selection for the parameter values and with performances close to the experimental optima.

In summary, the study carried out shows that the proposed methodology is valid for both the hardware system and the software and allows an easy extension of the optimization techniques to new logical and physical systems, due to its ability to adapt to different types of computing units, linear algebra routines, libraries and software tools for the selection of satisfactory values for the algorithmic and system parameters.

The features described for the auto-tuning methodology has been implemented in a hierarchical installation and optimization software, which is freely available for download together with the user's and developer's guide from http://luna.inf.um.es/grupo_investigacion/software. Also, the user's and developer's manual are shown in Annex A.

The researching tasks carried out in this Thesis have established a solid base to continue working with the proposed hierarchical auto-tuning methodology. The goal is to advance in the same direction, extending the methodology with new features and considering the integration inside higher-level software that can use it to achieve an optimized execution. Here is a list of tasks ordered in function of its type and the time of achievement.

- Extend the auto-tuning methodology with new features:
  - Consider other parameters, routines and linear algebra libraries.
  - Combine the application of the methodology with theoretical modeling techniques of the execution time of the routines.
  - Integrate OpenTuner as a software component inside the auto-tuning engine of the methodology to use the different heuristics for search the best values of the algorithmic parameters at each level of the hierarchy.

- Adapt the methodology to support bi-objective optimization, both in performance and energy consumption.

- Extend the use of the auto-tuning methodology to scientific software:

  - Include the auto-tuning engine of the methodology as a software component inside the Chameleon library.
  - Combine the tuned version of Chameleon with the SimGrid simulator for solving scientific problems with high computing requirements using the optimized performance models of the linear algebra routines and without requiring the use of the computational platform.
  - Use the methodology for the efficient resolution of computational kinematics problems in mechanical engineering. Currently, we are working in the integration of the auto-tuning engine inside a software used for the simulation of multibody systems.

This work has culminated with the publication:

Jesús Cámara, Javier Cuenca, Domingo Giménez. Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters. *The Journal of Supercomputing*, 2020.

This article addresses the proposed hierarchical auto-tuning methodology and shows the main results obtained (chapters 4 and 5). In addition, some results (such as those in chapter 6) have been presented in international conferences [3], which justifies the scientific validity of the research tasks carried out. Finally, it is expected that the auto-tuning methodology can be applied for the efficient resolution of scientific and engineering problems.

Bibliografía

[1] Advanced Micro Devices. AMD Core Math Library User's Guide. http://developer.amd.com/wordpress/media/2013/05/acml.pdf. Online; accedido 20-03-2020.

[2] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better - a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*, volume 2. Morgan Kaufmann, 2010.

[3] Emmanuel Agullo, Jesús Cámara, Javier Cuenca, and Domingo Giménez. On the Autotuning of Task-Based Numerical Libraries for Heterogeneous Architectures. In *Proceedings of the 2019 International Conference on Parallel Computing*, volume 36 of *Advances in Parallel Computing*, pages 157–166, 2020.

[4] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, 08 2009.

[5] Shameem Akhter and Jason Roberts. *Multi-Core Programming: Increasing Performance through Software Multi-threading*. Intel Press, 2006.

[6] Selim G. Akl. *Diseño y Análisis de Algoritmos Paralelos*. RA-MA, 1992.

[7] Enrique Alba. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, 2005.

[8] Pedro Alberti, Pedro Alonso, Antonio M. Vidal, Javier Cuenca, and Domingo Giménez. Designing polylibraries to speed up linear algebra computations. *International Journal of High-Performance Computing and Networking*, 1(1-3):75–84, 01 2004.

[9] Francisco Almeida, Rumen Andonov, Luz Moreno, Vincent Poirriez, Melquíades Pérez, and Casiano Rodríguez-León. On the parallel prediction of the RNA secondary structure. In *Proceedings of the 2003 International Conference on Parallel Computing*, volume 13 of *Advances in Parallel Computing*, pages 525–532, 2004.

[10] Francisco Almeida, Domingo Giménez, and José-Juan López-Espín. A parameterized shared-memory scheme for parameterized metaheuristics. *The Journal of Supercomputing*, 58(3):292–301, 12 2011.

[11] Francisco Almeida, Domingo Giménez, José Miguel Mantas, and Antonio Vidal. *Introducción a la Programación Paralela*. Paraninfo, 2008.

[12] Pedro Alonso, Ravi Reddy, and Alexey Lastovetsky. Experimental Study of Six Different Implementations of Parallel Matrix Multiplication on Heterogeneous Computational Clusters of Multicore Processors. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 263–270, 2010.

[13] Serkan Altuntaş, Zeki Bozkus, and Basilio Fraguela. GPU Accelerated Molecular Docking Simulation with Genetic Algorithms. In *Proceedings of 19th European Conference on the Applications of Evolutionary Computation*, volume 9598 of *Lecture Notes in Computer Science*, pages 134–146, 2016.

[14] Alejandro Álvarez-Melcón, Domingo Giménez, Fernando-Daniel Quesada-Pereira, and Tomás Ramírez. Hybrid-Parallel Algorithms for 2D Green's Functions. In *Procedia Computer Science*, volume 18, pages 541–550, 2013.

[15] Alejandro Álvarez-Melcón, Fernando-Daniel Quesada-Pereira, Domingo Giménez, Carlos Pérez-Alcaraz, Tomás Ramírez, and José-Ginés Picón. On the development and optimization of hybrid parallel codes for integral equation formulations (EuCAP). In *Proceedings of the 7th European Conference on Antennas and Propagation*, pages 2648–2652, 2013.

[16] E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Third Edition)*. Society for Industrial and Applied Mathematics, 1999.

[17] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *23rd International Conference on Parallel Architectures and Compilation Techniques*, pages 303–316, 2014.

[18] Juan Aparicio, José-Juan López-Espín, Raúl Martínez-Moreno, and Jesús T. Pastor. Benchmarking in Data Envelopment Analysis: An Approach Based on Genetic Algorithms and Parallel Programming. *Advances in Operations Research*, 2014:9, 02 2014.

[19] ATMG. Auto-Tuning for Multicore and GPU. http://atrg.jp/atmg. Online; accedido 20-03-2020.

[20] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. STARPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 02 2011.

[21] Jacques Bahi, Sylvain Contassot-Vivier, and Raphael Couturier. *Parallel Iterative Algorithms*. Chapman & Hall, 2007.

[22] Marcos Barreto, Murilo Boratto, Pedro Alonso, and Domingo Giménez. Automatic routine tuning to represent landform attributes on multicore and multi-GPU systems. *The Journal of Supercomputing*, 70(2):733–745, 03 2014.

[23] Gregorio Bernabé, José-Carlos Cano, Javier Cuenca, Antonio Flores, Domingo Giménez, Mariano Saura-Sánchez, and Pablo Segado-Cabezos. Exploiting Hybrid Parallelism in the Kinematic Analysis of Multibody Systems Based on Group Equations. In *Procedia Computer Science*, volume 108, pages 576–585, 2017.

[24] Gregorio Bernabé, Javier Cuenca, Luis-Pedro García, and Domingo Giménez. Auto-tuning techniques for linear algebra routines on hybrid platforms. *Journal of Computational Science*, 10:299–310, 04 2015.

[25] Gregorio Bernabé, Javier Cuenca, and Domingo Giménez. An Autotuning Engine for the 3D Fast Wavelet Transform on Clusters with Hybrid CPU + GPU Platforms. *International Journal of Parallel Programming*, 43(6):1160–1191, 12 2015.

[26] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI.* Oxford University Press, 2004.

[27] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide.* Society for Industrial and Applied Mathematics, 1997.

[28] Murilo Boratto, Pedro Alonso, Domingo Giménez, and Alexey Lastovetsky. Automatic tuning to performance modelling of matrix polynomials on multicore and multi-GPU systems. *The Journal of Supercomputing*, 73(1):227–239, 01 2017.

[29] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics.* Prentice-Hall International, Inc., 1996.

[30] Eric A. Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1994.

[31] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov. The Impact of Multicore on Math Software. In *8th International Workshop on Applied Parallel Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10, 2006.

[32] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38–53, 01 2009.

[33] Jesús Cámara, Javier Cuenca, Luis-Pedro García, and Domingo Giménez. Empirical Modelling of Linear Algebra Shared-Memory Routines. In *Procedia Computer Science*, volume 18, pages 110–119, 2013.

[34] Jesús Cámara, Javier Cuenca, Luis-Pedro García, and Domingo Giménez. Auto-tuned nested parallelism: A way to reduce the execution time of scientific software in NUMA systems. *Parallel Computing*, 40(7):309–327, 2014.

[35] Jesús Cámara, Javier Cuenca, Domingo Giménez, Luis-Pedro García, and Antonio Vidal. Empirical Installation of Linear Algebra Shared-Memory Subroutines for Auto-Tuning. *International Journal of Parallel Programming*, 42(3):408–434, 06 2014.

[36] José-Carlos Cano, Javier Cuenca, Domingo Giménez, Mariano Saura-Sánchez, and Pablo Segado-Cabezos. A parallel simulator for multibody systems based on group equations. *The Journal of Supercomputing*, 75:1368–1381, 09 2018.

[37] Alfonso Castaño, Javier Cuenca, José-Matías Cutillas-Lozano, Domingo Giménez, José-Juan López-Espín, and Alberto Pérez-Bernabeu. Parallelism on Hybrid Metaheuristics for Vector Autoregression Models. In *Proceedings of the 2018 International Conference on High-Performance Computing & Simulation*, pages 828–835, 2018.

[38] Adrián Castelló, Rafael Mayo, Judit Planas, and Enrique S Quintana-Ortí. Exploiting Task-Parallelism on GPU Clusters via OmpSs and rCUDA Virtualization. In *1st IEEE International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms*, pages 160–165, 2015.

[39] José M. Cecilia, José-Matías Cutillas-Lozano, Domingo Giménez, and Baldomero Imbernón. Exploiting multilevel parallelism on a many-core system for the application of hyperheuristics to a molecular docking problem. *The Journal of Supercomputing*, 74:1803–1814, 03 2017.

[40] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kauffman, 2001.

[41] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self-Adapting Software for Numerical Linear Algebra and LAPACK for Clusters. *Parallel Computing*, 29(11–12):1723–1743, 11 2003.

[42] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and R. Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. Technical Report CS-95-292, Computer Science Dept. University of Tennessee, 05 1995.

[43] Chrysos, George. Intel Xeon Phi X100 Family Coprocessor - the Architecture. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner. Online; accedido 20-03-2020.

[44] Andrea Clematis, Gabriella Dodero, and Vittoria Gianuzzi. A Practical Approach to Efficient Use of Heterogeneous PC Network for Parallel Mathematical Computation. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, volume 2110 of *Lecture Notes in Computer Science*, pages 464–473, 2001.

[45] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 1990.

[46] Javier Cuenca, Luis García, Domingo Giménez, and Jack Dongarra. Processes Distribution of Homogeneous Parallel Linear Algebra Routines on Heterogeneous Clusters. In *2005 IEEE International Conference on Cluster Computing*, pages 1–10, 2005.

[47] Javier Cuenca, Luis-Pedro García, and Domingo Giménez. A proposal for autotuning linear algebra routines on multicore platforms. In *Procedia Computer Science*, volume 1, pages 515–523, 2010.

[48] Javier Cuenca, Luis-Pedro García, and Domingo Giménez. Improving Linear Algebra Computation on NUMA Platforms through Auto-tuned Nested Parallelism. In *Proceedings of 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 66–73, 2012.

[49] Javier Cuenca, Luis-Pedro García, Domingo Giménez, José González, and Antonio Vidal. Empirical Modelling of Parallel Linear Algebra Routines. In

Proceedings of the 5th International Conference on Parallel Processing and Applied Mathematics, volume 3019 of *Lecture Notes in Computer Science*, pages 169–174, 2003.

[50] Javier Cuenca, Luis-Pedro García, Domingo Giménez, and Francisco-José Herrera. Empirical Modelling: an Auto-tuning Method for Linear Algebra Routines on CPU+multiGPU Platforms. In *16th International Conference on Computational Methods in Science and Engineering*, 2016.

[51] Javier Cuenca, Luis-Pedro García, Domingo Giménez, and Francisco-José Herrera. Guided installation of basic linear algebra routines in a cluster with manycore components. *Concurrency and Computation: Practice and Experience*, 29(15):e4112, 03 2017.

[52] Javier Cuenca, Luis-Pedro García, Domingo Giménez, and Manuel Quesada-Martínez. Analysis of the Influence of the Compiler on Multicore Performance. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 170–174, 2010.

[53] Javier Cuenca, Domingo Giménez, and José González. Modeling the behaviour of linear algebra algorithms with message-passing. In *Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing*, pages 282–289, 2001.

[54] Javier Cuenca, Domingo Giménez, and José González. Towards the design of an automatically tuned linear algebra library. In *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing*, pages 201–208, 2002.

[55] Javier Cuenca, Domingo Giménez, and José González. Architecture of an automatically tuned linear algebra library. *Parallel Computing*, 30(2):187–210, 02 2004.

[56] Javier Cuenca, Domingo Giménez, José González, Jack Dongarra, and Kenneth Roche. Automatic Optimisation of Parallel Linear Algebra Routines in Systems with Variable Load. In *Proceedings of the 11th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 409–416, 2003.

[57] José-Matías Cutillas-Lozano and Domingo Giménez. Optimizing a parameterized message-passing metaheuristic scheme on a heterogeneous cluster. *Soft Computing*, 21(19):5557–5572, 09 2016.

[58] Krister Dackland and Bo Kågström. An hierarchical approach for performance analysis of ScaLAPACK-based routines using the distributed linear algebra machine. In *3rd International Workshop on Applied Parallel Computing*, pages 186–195, 1996.

[59] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms.* PhD thesis, Computer Science Division, University of California, Berkeley, 2009.

[60] NVIDIA Developers. *NVIDIA CUDA C Programming Guide 8.0.* NVIDIA Corporation, 2017.

[61] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience*, 28(17):4385–4404, 12 2016.

[62] Murilo do Carmo Boratto. *Modelos Paralelos para la Resolución de Problemas de Ingeniería Agrícola.* PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2015.

[63] Rafael Dolezal, Teodorico Ramalho, Tanos França, and Kamil Kuca. Parallel Flexible Molecular Docking in Computational Chemistry on High-Performance Computing Clusters. In *Computational Collective Intelligence*, pages 418–427. Springer, 2015.

[64] Jack Dongarra. The Future of the BLAS. In *PPSC*, 1999.

[65] Jack Dongarra, George Bosilca, Victor Eijkhout, Graham Fagg, Erika Fuentes, Julien Langou, Piotr Luszczek, Jelena Pjesivac-Grbovic, Keith Seymour, Haihang You, and Sathish Vadhiyar. Self-adapting numerical software (SANS) effort. *IBM Journal of Research and Development*, 50(2-3):223–238, 03 2006.

[66] Jack Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* Society for Industrial and Applied Mathematics, 1998.

[67] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of Parallel Computing.* Morgan Kaufmann Publishers, 2003.

[68] Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov. HPC Programming on Intel Many-Integrated-Core Hardware with MAGMA Port to Xeon Phi. *Scientific Programming*, 2015:1–11, 04 2015.

[69] Jack Dongarra, Jean-Francois Pineau, Yves Robert, and Frédéric Vivien. Matrix Product on Heterogeneous Master-Worker Platforms. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 53–62, 2008.

[70] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 03 1988.

[71] Jack J. Dongarra and R. Clint Whaley. A User's Guide to the BLACS v1.0. Technical Report CS-95-292, Dept. of Computer Sciences, University of Tennessee, Knoxville, TN 37996, 05 1997.

[72] José Duato, Antonio J Peña, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the 2010 International Conference on High-Performance Computing & Simulation*, pages 224–231, 2010.

[73] EM Photonics and NVIDIA. CULA GPU-accelerated Linear Algebra Libraries. http://www.culatools.com/. Online; accedido 20-03-2020.

[74] José Juan López Espín. *Aspectos Computacionales de la Resolución y Obtención de Modelos de Ecuaciones Simultáneas.* PhD thesis, Universidad de Murcia, 2009.

[75] Mokhtar Essaid, Lhassane Idoumghar, Julien Lepagnot, and Mathieu Brévilliers. GPU parallelization strategies for metaheuristics: a survey. *International Journal of Parallel, Emergent and Distributed Systems*, 34(5):497–522, 01 2018.

[76] Jianbin Fang, Ana Lucia Varbanescu, Baldomero Imbernón, José M. Cecilia, and Horacio Emilio Pérez Sánchez. 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*, pages 579–588, 2014.

[77] Massimiliano Fatica. Accelerating Linpack with CUDA on heterogenous clusters. In *2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 46–51, 2009.

[78] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[79] Len Freeman and Chris Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

[80] Matteo Frigo and Steven Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, 1998.

[81] Matteo Frigo and Stefan Kral. The Advanced FFT Program Generator GENFFT. *Report AURORA TR 2001-03*, 04 2001.

[82] Kyle Gallivan, William Jalby, Allen Malony, and Harry Wijshoff. Performance Prediction for Parallel Numerical Algorithms. *International Journal of High Speed Computing*, 1(3):31–62, 03 1991.

[83] Luis Pedro García. *Técnicas de Modelado y Optimización del Tiempo de Ejecución de Rutinas Paralelas de Álgebra Lineal*. PhD thesis, Departamento de Informática y Sistemas, Universidad de Murcia, 2012.

[84] Luis-Pedro García, Javier Cuenca, and Domingo Giménez. Auto-Optimization of Linear Algebra Parallel Routines: The Cholesky Factorization. In *Current & Future Issues of High-End Computing*, pages 229–236, 2005.

[85] Luis-Pedro García, Javier Cuenca, and Domingo Giménez. Including Improvement of the Execution Time in a Software Architecture of Libraries With Self-Optimisation. In *Proceedings of the 2nd International Conference on Software and Data Technologies*, pages 156–161, 2007.

[86] Luis-Pedro García, Javier Cuenca, and Domingo Giménez. Using Experimental Data to Improve the Performance Modelling of Parallel Linear Algebra Routines. In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 1150–1159, 2007.

[87] Luis-Pedro García, Javier Cuenca, Francisco-José Herrera, and Domingo Giménez. On Guided Installation of Basic Linear Algebra Routines in Nodes with Manycore Components. In *7th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 114–122, 2016.

[88] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, 11 1995.

[89] Domingo Giménez. Revisiting Strassen's Matrix Multiplication for Multicore Systems. *Annals of Multicore and GPU Programming*, 4(1):1–8, 2017.

[90] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. Society for Industrial and Applied Mathematics, 2001.

[91] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (Fourth Edition)*. The John Hopkins University Press, 2013.

[92] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (Second Edition)*. Addison-Wesley, 01 2003.

[93] John Gunnels, Fred Gustavson, Gwendolyn Henry, and Robert van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 12 2001.

[94] Salim Hariri and Manish Parashar. *Tools and Environments for Parallel and Distributed Computing*. Wiley-Interscience, 2004.

[95] Khalid Hasanov, Jean-Noël Quintin, and Alexey Lastovetsky. Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms. *The Journal of Supercomputing*, 71(11):3991–4014, 11 2015.

[96] Jianyu Huang, Tyler Smith, Greg Henry, and Robert van de Geijn. Strassen's Algorithm Reloaded. In *Proceedings of the SC16: International Conference for High-Performance Computing, Networking, Storage and Analysis*, pages 690–701, 2016.

[97] Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn. Implementing Strassen's Algorithm with CUTLASS on NVIDIA Volta GPUs. *CoRR*, abs/1808.07984, 08 2018.

[98] Cameron Hughes and Tracey Hughes. *Professional Multicore Programming: Design and Implementation for C++ Developers*. Wiley Publishing, 2008.

[99] Sascha Hunold and Thomas Rauber. Automatic Tuning of PDGEMM Towards Optimal Performance. In *Proceedings of the 11th International European Conference on Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 837–846, 2005.

[100] IBM. Engineering and Scientific Subroutine Library. https://www.ibm.com/support/knowledgecenter/en/SSFHY8_6.1/navigation/welcome.html. Online; accedido 20-03-2020.

[101] ICL Team. Chameleon: A Dense Linear Algebra Software for Heterogeneous Architectures. https://gitlab.inria.fr/solverstack/chameleon. Online; accedido 20-03-2020.

[102] Baldomero Imbernón, José M. Cecilia, and Domingo 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*, pages 50–58, 2016.

[103] Baldomero Imbernón, José M. Cecilia, Horacio Pérez-Sánchez, and Domingo Giménez. METADOCK: A parallel metaheuristic schema for virtual screening methods. *International Journal of High-Performance Computing Applications*, 03 2017.

[104] Baldomero Imbernón, Antonio Llanes-Castro, Jorge Peña-García, José L. Abellán, Horacio Pérez-Sánchez, and José M. Cecilia. Enhancing the Parallelization of Non-bonded Interactions Kernel for Virtual Screening on GPUs. In *Proceedings of the 3rd International Conference on Bioinformatics and Biomedical Engineering*, volume 9044 of *Lecture Notes in Computer Science*, pages 620–626, 2015.

[105] Baldomero Imbernón, Javier Prades, Domingo Giménez, José M. Cecilia, and Federico Silla. Enhancing large-scale docking simulation on heterogeneous systems: An MPI vs rCUDA study. *Future Generation Computer Systems*, 79:26–37, 02 2018.

[106] Intel Corporation. Intel Math Kernel Library. https://software.intel.com/en-us/mkl. Online; accedido 20-03-2020.

[107] Muhammad Ali Ismail, S. Mirza, Talat Altaf, Dr. Mirza, and Dr. Altaf. Concurrent Matrix Multiplication on Multi-Core Processors. *International Journal of Computer Science and Security*, 5:208–220, 05 2011.

[108] iWAPT. The International Workshop on Automatic Performance Tuning. http://iwapt.org/. Online; accedido 20-03-2020.

[109] Jack Dongarra. List of Freely Available Software for Linear Algebra on the Web. http://www.netlib.org/utk/people/JackDongarra/la-sw.html. Online; accedido 20-03-2020.

[110] Sonia Jerez, J. Montávez, and Domingo Giménez. Optimizing the execution of a parallel meteorology simulation code. In *23rd IEEE International Symposium on Parallel & Distributed Processing*, pages 1–6, 2009.

[111] Alexey Kalinov and Alexey Lastovetsky. Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers. *Journal of Parallel and Distributed Computing*, 61(4):520–535, 04 2001.

[112] George Karniadakis and Robert Kirby. *Parallel Scientific Computing in C++ and MPI.* Cambridge University Press, 2003.

[113] Elaye Karstadt and Oded Schwartz. Matrix Multiplication, a Little Faster. In *29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 101–110, 2017.

[114] Takahiro Katagiri, Yoshinori Ishii, and Hiroki Honda. RAO-SS: A Prototype of Run-time Auto-tuning Facility for Sparse Direct Solvers. *Technical Report of UEC-IS-2005-2*, 07 2005.

[115] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. FIBER: A Generalized Framework for Auto-tuning Software. In *5th International Symposium on High-Performance Computing*, pages 146–159, 2003.

[116] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. Effect of auto-tuning with user's knowledge for numerical software. In *Proceedings of the 1st Conference on Computing Frontiers*, pages 12–25, 2004.

[117] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. ABCLib_DRSSED: A parallel eigensolver with an auto-tuning facility. *Parallel Computing*, 32(3):231–250, 03 2006.

[118] Stephen Keckler, William Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 11 2011.

[119] David Kirk and Hwu Wen-Mei. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 2016.

[120] Branko Kolundzija, Dragan Olcan, Dusan Zoric, and Sladjana Maric. Accelerating WIPL-D numerical EM kernel by using graphics processing units. In *Proceedings of the 10th International Conference on Telecommunication in Modern Satellite Cable and Broadcasting Services*, pages 413–419, 2011.

[121] Branko Kolundzija, Miodrag Tasic, Dragan Olcan, Dusan Zoric, and Srdjan Stevanetic. Full-wave analysis of electrically large structures on desktop PCs. In *CEM11: Computational Electromagnetics International Workshop*, pages 122–127, 2011.

[122] Alice Koniges. *Industrial Strength Parallel Computing*. Morgan Kaufmann Publishers, 2000.

[123] Erricos Kontoghiorghes. *Parallel Algorithms for Linear Models: Numerical Methods and Estimation Problems*. Kluwer Academic Publishers, 01 2000.

[124] Hisayasu Kuroda, Takahiro Katagiri, and Yasumasa Kanada. Parallel numerical library project: how ILB is to be developed. In *Workshop on Scalable Solver Software*, 2001.

[125] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 01 2010.

[126] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. Autotuning GEMM kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 01 2012.

[127] Scott Larsen and David McAllister. Fast Matrix Multiplies Using Graphics Hardware. In *ACM/IEEE Conference on Supercomputing*, page 43, 2001.

[128] Alexey Lastovetsky and Ravi Reddy. A Novel Algorithm of Optimal Matrix Partitioning for Parallel Dense Factorization on Heterogeneous Processors. In *Proceedings of the 9th International Conference on Parallel Computing Technologies*, volume 4671 of *Lecture Notes in Computer Science*, pages 261–275, 2007.

[129] François Le Gall. Powers of Tensors and Fast Matrix Multiplication. In *39th International Symposium on Symbolic and Algebraic Computation*, pages 296–303, 2014.

[130] Victor Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38(3):451–460, 01 2010.

[131] Bruce P. Lester. *The Art of Parallel Programming*. Prentice Hall, 01 1993.

[132] Jin Li, Anthony Skjellum, Ioana Banicescu, Clayborne Taylor, Donna Reese, Susan Bridges, and Richard Koshel. A Poly-Algorithm For Parallel Dense Matrix Multiplication On Two-Dimensional Process Grid Topologies. *Concurrency and Computation: Practice and Experience*, 9(5):345–389, 8 1997.

[133] Yan Li, Yunquan Zhang, Yiqun Liu, Guo-Ping Long, and Hai-Peng Jia. MPFFT: An auto-tuning FFT library for OpenCL GPUs. *Journal of Computer Science and Technology*, 28(1):90–105, 01 2013.

[134] Chun-Yuan Lin, Chen Huang, Yeh-Ching Chung, and Chuan Tang. Efficient Parallel Algorithm for Optimal Three-Sequences Alignment. In *2007 International Conference on Parallel Processing*, page 14, 2007.

[135] Francisco López-Castejón and Domingo Giménez. Auto-optimization on parallel hydrodynamic codes: an example of COHERENS with OpenMP for multicore. In *XVIII International Conference on Water Resources*, 2010.

[136] José-Juan López-Espín and Domingo Giménez. Solution of Simultaneous Equations Models in high performance systems. In *Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2006.

[137] José-Juan López-Espín and Domingo Giménez. Message-Passing Two Steps Least Square Algorithms for Simultaneous Equations Models. In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 127–136, 2007.

[138] Mark Lundstrom. Moore's Law Forever? *Science*, 299(5604):210–211, 2003.

[139] Gabriel Luque and Enrique Alba. *Parallel Genetic Algorithms: Theory and Real World Applications*. Springer, 2011.

[140] Simon McIntosh-Smith, James Price, Richard Sessions, and Amaurys Ibarra. High performance in silico virtual drug screening on many-core processors. *The International Journal of High-Performance Computing Applications*, 29(2):119–134, 05 2015.

[141] Meuer, Hans and Strohmaier, Erich and Dongarra, Jack and Simon, Horst and Meuer, Martin. Top 500: The List. https://www.top500.org/. Online; accedido 20-03-2020.

[142] Jagdish J. Modi. *Parallel Algorithms and Matrix Computation*. Clarendon Press, 1988.

[143] Gordon Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *Solid-State Circuits Newsletter*, 11:33–35, 10 2006.

[144] MPI Forum. Activities of the Message Passing Interface. https://www.mpi-forum.org/. Online; accedido 20-03-2020.

[145] Javier Cuenca Muñoz. *Optimización Automática de Software Paralelo de Álgebra Lineal*. PhD thesis, Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, 2004.

[146] Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda. *Software Automatic Tuning. From Concepts to State-of-the-Art Results*. Springer, 2010.

[147] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An Improved Magma Gemm For Fermi Graphics Processing Units. *International Journal of High-Performance Computing Applications*, 24(4):511–515, 11 2010.

[148] Nguyen, Loc Q (Intel). Using Intel MKL Compiler Assisted Offload in Intel Xeon Phi Processor. https://software.intel.com/en-us/articles/using-intel-math-kernel-library-compiler-assisted-offload-in-intel-xeon-phi-processor. Online; accedido 20-03-2020.

[149] Nguyen, Loc Q (Intel). Using Intel MPI Library on Intel Xeon Phi Product Family. https://software.intel.com/en-us/articles/using-intel-mpi-library-on-intel-xeon-phi-product-family. Online; accedido 20-03-2020.

[150] Bradford Nichols, Dick Buttlar, and Jacqueline Farrel. *Pthreads Programming: A Posix Standard For Better Multiprocessing*. O'Reilly, 1996.

[151] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 03 2008.

[152] NVIDIA Corporation. cuBLAS Library User's Guide v8.0. https://docs.nvidia.com/cuda/archive/8.0/pdf/CUBLAS_Library.pdf. Online; accedido 20-03-2020.

[153] NVIDIA Corporation. Web Page. https://www.nvidia.com/es-es/. Online; accedido 20-03-2020.

[154] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba. Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment. In *Proceedings of the 7th International Conference on High-Performance Computing for Computational Science*, volume 4395 of *Lecture Notes in Computer Science*, pages 305–318, 2007.

[155] OpenMP ARB (Architecture Review Boards). The OpenMP API Specification for Parallel Programming. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Online; accedido 20-03-2020.

[156] James M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, 1989.

[157] John Owens, Mike Houston, David Luebke, Simon Green, John Stone, and James Phillips. GPU computing. In *Proceedings of the IEEE*, volume 96, pages 879–899, 05 2008.

[158] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[159] Parallelum. Scientific Computing and Parallel Programming Group. http://luna.inf.um.es/grupo_investigacion/. Online; accedido 20-03-2020.

[160] Behrooz Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, 2002.

[161] Antoine Petitet, L. Blackford, Jack Dongarra, Brett Ellis, Graham Fagg, Kenneth Roche, and Sathish Vadhiyar. Numerical Libraries and the Grid. *International Journal of High-Performance Computing Applications*, 15(4):359–374, 01 2001.

[162] Philip Pfaffe, Tobias Grosser, and Martin Tillmann. Efficient Hierarchical Online-Autotuning: A Case Study on Polyhedral Accelerator Mapping. In *Proceedings of the ACM/IEEE International Conference on Supercomputing*, pages 354–366, 2019.

[163] Markus Püschel, José Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert Johnson. A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *International Journal of High-Performance Computing Applications*, 18:21–45, 02 2004.

[164] Michael Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, 2004.

[165] Thomas Rauber and Gudula Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2012.

[166] Ravi Reddy and Alexey Lastovetsky. HeteroMPI+ScaLAPACK: Towards a ScaLAPACK (Dense Linear Solvers) on Heterogeneous Networks of Computers. In *Proceedings of the 13th International Conference on High-Performance Computing*, volume 4297 of *Lecture Notes in Computer Science*, pages 242–253, 2006.

[167] Daniel Reed and Jack Dongarra. Exascale Computing and Big Data. *Communications of the ACM*, 58(7):56–68, 07 2015.

[168] Reinders, J. An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors. https://www.cs.unc.edu/~prins/Classes/633/Readings/An-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_Reinders.pdf. Online; accedido 20-03-2020.

[169] Seyed H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer, 2000.

[170] Takao Sakurai, Takahiro Katagiri, Hisayasu Kuroda, Ken Naono, Mitsuyoshi Igai, and Satoshi Ohshima. A Sparse Matrix Library with Automatic Selection of Iterative Solvers and Preconditioners. *Procedia Computer Science*, 18:1332–1341, 12 2013.

[171] Santiago Sánchez, Erney Ramírez-Aportela, José Garzón, Pablo Chacón, Antonio S. Montemayor, and Raúl Cabido. FRODRUG: A Virtual Screening GPU Accelerated Approach for Drug Discovery. In *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 594–600, 02 2014.

[172] Robert R. Schaller. Moore's law: past, present, and future. *IEEE Spectrum*, 34(6):52–59, 06 1997.

[173] Ridgway Scott, Terry Clark, and Babak Bagheri. *Scientific Parallel Computing*. Princeton University Press, 2005.

[174] Ronald Shonkwiler and Lew Lefton. *An Introduction to Parallel and Vector Scientific Computation*. Cambridge University Press, 2006.

[175] Federico Silla, Javier Prades, Sergio Iserte, and Carlos Reaño. Remote GPU Virtualization: Is It Useful? In *2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era*, pages 41–48, 2016.

[176] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI. The Complete Reference (2nd Ed.)*. The MIT Press, 1998.

[177] Fengguang Song, Hatem Ltaief, Bilel Hadri, and Jack Dongarra. Scalable Tile Communication-Avoiding QR Factorization on Multicore Cluster Systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High-Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[178] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. *Concurrency and Computation: Practice and Experience*, 27:4075–4090, 2015.

[179] John Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(1-3):66–72, 05 2010.

[180] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[181] El-Ghazali Talbi. Parallel Evolutionary Combinatorial Optimization. In *Handbook of Computational Intelligence*, pages 1107–1125. Springer, 2015.

[182] El-Ghazali Talbi and Geir Hasle. Metaheuristics on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):1–3, 01 2013.

[183] The Open MPI Project. Open MPI: Open Source High-Performance Computing. https://www.open-mpi.org/. Online; accedido 20-03-2020.

[184] T.N. Theis and H.-S. Philip Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science & Engineering*, 19(2):41–50, 03 2017.

[185] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36:232–240, 06 2010.

[186] Oleg Trott and Arthur 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, 2009.

[187] Yaohung Tsai, Weichung Wang, and Ray-Bing Chen. Tuning block size for QR factorization on CPU-GPU hybrid systems. In *IEEE 6th International Symposium on Embedded Multicore Socs*, pages 205–211, 2012.

[188] Baldomero Imbernón Tudela. *Técnicas Metaheurísticas Paralelas en Acoplamiento de Compuestos Bioactivos*. PhD thesis, Facultad de Informática, Universidad de Murcia, 2015.

[189] Robert van de Geijn and Kazushige Goto. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):Article 12, 05 2008.

[190] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[191] Isidro Verdú, Domingo Giménez, and J.J. Sánchez. Three Dimensional Image Synthesis in Transputers. *Microprocessing and Microprogramming*, 40(10-12):919–922, 12 1994.

[192] Isidro Verdú, Domingo Giménez, and Juan-Carlos Torres. Ray Tracing for natural scenes in parallel processors. In *Proceedings of the International Conference on High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 297–305, 1996.

[193] Vasily Volkov and James Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the ACM/IEEE International Conference on Supercomputing*, pages 1–11, 2008.

[194] Richard Vuduc, James Demmel, and Jeff Bilmes. Statistical Models for Automatic Performance Tuning. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *Lecture Notes in Computer Science*, pages 117–126, 2001.

[195] Feng Wang, Can-Qun Yang, Yun-Fei Du, Juan Chen, Hui-Zhan Yi, and Wei-Xia Xu. Optimizing Linpack Benchmark on GPU-Accelerated Petascale Supercomputer. *Journal of Computer Science and Technology*, 26(5):854–865, 09 2011.

[196] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Singh, Sayantan Sur, and D.K. Panda. MVAPICH2GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science - Research and Development*, 26(3-4):257–266, 06 2011.

[197] Ping Wang, Tony Song, Yi Chao, and Hongchun Zhang. Parallel Computation of the Regional Ocean Modeling System. *International Journal of High-Performance Computing Applications*, 19(4):375–385, 01 2005.

[198] R. Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 01 2001.

[199] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (Second Edition)*. Prentice-Hall, 2005.

[200] Samuel Williams, Leonid Oliker, Jonathan Carter, and John Shalf. Extracting ultra-scale Lattice Boltzmann performance via hierarchical and distributed auto-tuning. In *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[201] Xianyi, Zhang and Kroeker, Martin. OpenBLAS: An Optimized BLAS Library. https://www.openblas.net/. Online; accedido 20-03-2020.

[202] Peng Zhang and Yuxiang Gao. Matrix Multiplication on High-Density Multi-GPU Architectures: Theoretical and Experimental Investigations. In *Proceedings of the 30th International Conference on High-Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 17–30, 2015.

[203] Y. Zhang, Tapan Sarkar, Heejeong Moon, A. De, and M.C. Taylor. Solution of large complex problems in computational electromagnetics using higher order basis in MOM with parallel solvers. In *IEEE Antennas and Propagation Society International Symposium*, pages 5620–5623, 2007.

[204] Ziming Zhong, Vladimir Rychkov, and Alexey Lastovetsky. Data Partitioning on Heterogeneous Multicore Platforms. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 580–584, 2011.

Anexo A

Software de Instalación y Auto-Optimización Jerárquica

A.1 Introducción

El presente documento describe la estructura y funcionamiento de la aplicación software desarrollada para la instalación y auto-optimización jerárquica de rutinas de álgebra lineal sobre plataformas computacionales heterogéneas. Esta aplicación ha sido bautizada con el nombre **HATLib** (*Hierarchical AutoTuning Library*), ya que podrá ser utilizada en forma de librería para la ejecución optimizada de las rutinas contenidas en códigos científicos.

El contenido que se describe corresponde a la versión inicial del software, que se irá actualizando con nuevas mejoras y funcionalidad adicional. La última versión estable estará siempre accesible para su libre descarga en la página web http://luna.inf.um.es/grupo_investigacion/software.

Este software permite realizar la instalación jerárquica de rutinas de álgebra lineal, auto-optimizando su ejecución conforme se avanza en los diferentes niveles hardware (agrupación de unidades de cómputo) y software (jerarquía de rutinas). Durante el proceso de instalación, por un lado, se determinan los valores de los parámetros algorímicos en el nivel actual y, por otro, se aplica una metodología de optimización que permite, mediante el uso de la información de instalación

almacenada en niveles inferiores de la jerarquía, ejecutar de forma eficiente la rutina que está siendo instalada. De esta forma se consigue, a su vez, reducir el tiempo de instalación.

El proceso de instalación se ha de llevar a cabo comenzando siempre desde el nivel más bajo de la jerarquía. En la dimensión hardware, este nivel corresponde a las unidades básicas de procesamiento (CPU, GPUs y/o MICs) presentes en los nodos de cómputo. El siguiente nivel (nivel 1) correspondería al nodo de cómputo en su totalidad o cualquier subconjunto de unidades de cómputo del mismo (nodos virtuales). El último nivel (nivel 2), equivaldría a usar la plataforma completa o un subconjunto de nodos (virtuales o no) de la misma, es decir, cualquier agrupación de unidades de cómputo de la plataforma, donde cada una puede estar formada, a su vez, por un subconjunto de unidades de cómputo. En la dimensión software, en cambio, esta versión inicial sólo permite la instalación de la rutina de multiplicación de matrices en los niveles hardware mencionados. Se ha considerado inicialmente esta rutina porque constituye el kernel computacional básico de la gran mayoría de rutinas de álgebra lineal numérica. Una vez instalada la rutina, la aplicación ofrece otras rutinas de nivel superior, como la multiplicación de Strassen o la factorización LU, que se pueden ejecutar en diferentes niveles usando internamente la rutina auto-optimizada de multiplicación de matrices. En siguientes versiones se extenderá su funcionalidad para permitir instalar rutinas de forma optimizada en cualquier nivel hardware de la plataforma, haciendo uso de una jerarquía de niveles de rutinas similar a la establecida en librerías de álgebra lineal como BLAS y LAPACK.

La aplicación software se ha diseñado de forma que sea independiente de la plataforma heterogénea utilizada y fácilmente extensible a otras rutinas o librerías de álgebra lineal. Actualmente soporta sistemas multiCPU, multiGPU, multiMIC y cualquier combinación de los anteriores. Asimismo, permite seleccionar por medio de la propia interfaz ofrecida por el software, la librería optimizada de álgebra lineal numérica a utilizar en cada unidad básica de procesamiento durante la instalación de las rutinas (MKL en CPU y Xeon Phi o cuBLAS en GPU). No obstante, el software dispone de algunas limitaciones. La primera ya se ha mencionado, referente a que esta versión inicial sólo contempla la instalación jerárquica de la rutina de multiplicación de matrices. La otra limitación radica en que sólo

admite usar hasta un máximo de siete unidades de cómputo, tanto a nivel de unidades básicas de procesamiento como a nivel de nodos de cómputo (o virtuales). Estas limitaciones irán desapareciendo conforme se extienda la funcionalidad de la aplicación. Cabe indicar que el software ya ha sido adaptado para permitir su extensión a otras rutinas, que pueden ser instaladas en cualquier nivel software de la jerarquía haciendo uso de otras librerías de álgebra lineal, como PLASMA (para CPU), MAGMA (para CPU+GPU) o Chameleon. El manual de desarrollador explica de forma detallada cómo se llevarían a cabo estas tareas.

El documento se ha organizado de la siguiente manera. En primer lugar, se describe en la sección A.2 la estructura del software desarrollado. A continuación, se muestran los manuales de usuario y de desarrollador en las secciones A.3 y A.4, respectivamente.

A.2 Estructura de la Aplicación

La figura A.1 (izquierda) muestra la estructura original de directorios de que consta la aplicación. La figura A.1 (derecha), por su parte, muestra los directorios que se crean tras realizar una primera instalación en el sistema computacional. A continuación se indica la finalidad de cada uno de ellos:

- `code`: contiene los ficheros de código fuente y *makefiles* necesarios para compilar todo el código. Estos ficheros están distribuidos en tres subdirectorios:
  - `install`: contiene los ficheros de código fuente necesarios para realizar la instalación jerárquica de las rutinas en el sistema.
  - `library`: contiene los ficheros necesarios para crear una librería estática con todos los datos de instalación, de forma que pueda ser enlazada, en tiempo de compilación, con cualquier código que quiera hacer uso de la información de las rutinas instaladas para ejecutar de forma optimizada las rutinas a las que invoca.
  - `test`: contiene ficheros para probar, por un lado, la metodología de optimización con un programa de ejemplo que invoca a rutinas instaladas en el sistema, y por otro, un código que permite determinar los

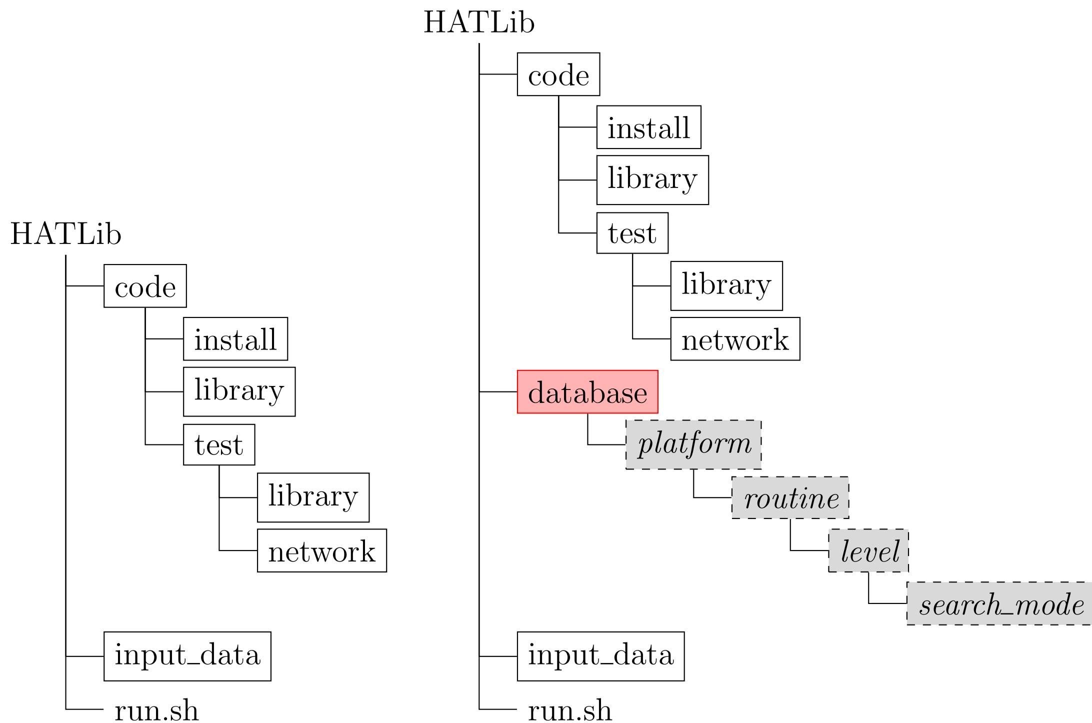

Figura A.1: Estructura de directorios de la aplicación **HATLib**.

parámetros del sistema que intervienen en el tiempo de comunicación entre cada par de nodos de un cluster.

- `input_data`: contiene ficheros de ejemplo con datos de entrada para cada nivel de la jerarquía hardware (unidades de cómputo a utilizar), así como un fichero que contiene un conjunto de tamaños de problema.

- `database` contiene la información de instalación de las rutinas en cada uno de los niveles hardware en los que haya sido instalada, con subdirectorios independientes para cada método (o herramienta software) de búsqueda (*search mode*) utilizado durante la instalación para la búsqueda del valor de los parámetros algorítmicos.

A.3 Manual de Usuario

La aplicación software desarrollada presenta una interfaz bastante intuitiva y sencilla de usar, con un diseño enfocado a facilitar la instalación de las rutinas en diferentes niveles de la plataforma. Esta instalación la puede llevar a cabo un usuario en su directorio de trabajo, o de forma global (en un directorio compartido) por parte del administrador del sistema. Esto último va a permitir que los usuarios puedan utilizar la información de instalación de las rutinas previamente instaladas para ejecutar de forma eficiente sus códigos en la plataforma computacional.

Para ejecutar la aplicación, el software dispone de un script con el nombre `run.sh`. Este script se encuentra en el directorio principal de la aplicación.

Una vez ejecutado (`source run.sh`), muestra un menú con las diferentes opciones que ofrece la aplicación. La figura A.2 muestra este menú. A continuación se comentan cada una de las opciones:

```
---------------------------------
      HATLib Library
---------------------------------
1. Install Routine
2. Show Installation Info
3. Test MPI Network Comms
4. Delete Installation
5. Create Library File
6. Test Library
7. Exit

Enter Your Choice [1-7]:
```

Figura A.2: Menú de la aplicación software.

- `Install Routine`: permite instalar una rutina en un nivel. Al seleccionar el nivel, se irá solicitando la información necesaria para instalar la rutina.
- `Show Installation Info`: muestra la información de instalación almacenada en el nivel seleccionado.

- `Test MPI Network Comms`: hace un chequeo de las comunicaciones entre todos los pares de nodos para estimar los parámetros del sistema que influyen en el tiempo de comunicación. Esto es útil, principalmente, cuando se instala la rutina a nivel de cluster.

- `Delete Installation`: elimina todos los ficheros generados durante la instalación de las rutinas, así como la propia información de instalación (directorio `database`).

- `Create Library File`: permite crear la librería estática `libhatlib.a` con toda la información de instalación almacenada en los diferentes niveles para cada rutina instalada en el sistema. Asimismo, proporciona una API con un conjunto de funciones que pueden ser invocadas desde el código del usuario para hacer uso de la versión auto-optimizada de las rutinas instaladas. Para hacer uso de esta librería, el usuario ha de compilar su código junto a esta librería. No obstante, si se reinstala alguna rutina previamente instalada con una nueva configuración (conjunto de instalación y/o plataforma diferente), se ha de volver a crear la librería desde el menú para que se añada la información de instalación generada a la ya existente.

- `Test Library`: permite ejecutar los `tests` proporcionados por la aplicación software para probar el funcionamiento de la metodología de auto-optimización. Estos `tests` contienen rutinas que invocan a rutinas básicas previamente instaladas en el sistema haciendo uso de las funciones de la API proporcionada por la librería estática `libhatlib.a`

- `Exit`: finaliza la ejecución de la aplicación.

Es importante indicar que el proceso de instalación jerárquica de la rutina se ha de llevar a cabo por niveles, comenzando por el nivel inferior de la jerarquía y continuando hacia niveles superiores, tal y como se ha indicado al comienzo del presente documento.

Al realizar la instalación de la rutina, el usuario tendrá que proporcionar determinada información, que irá solicitando la propia aplicación en función del nivel en el que se realice. Entre esta información se encuentra la librería optimizada de álgebra lineal a utilizar en cada unidad de cómputo, el método (o herramienta)

para la búsqueda de los parámetros algorítmicos, el número de ejecuciones a realizar durante la instalación (para evitar grandes fluctuaciones en el rendimiento obtenido) o el grano de distribución de la carga (si se trata del nivel 0 de la jerarquía). No obstante, en todos los niveles solicitará que se suministre un fichero con el conjunto de tamaños de problema y otro con las unidades de cómputo sobre las que realizar la instalación. La aplicación incluye un fichero de cada tipo en el directorio `input_data`, que utilizará por defecto si no se proporciona ninguno. La estructura de cada uno es la siguiente:

- `input_sizes`: contiene el conjunto de instalación con los tamaños de problema a utilizar. Se han de especificar las dimensiones m, k y n de las matrices, introduciendo tantas líneas como tamaños de problema se quieran usar.
- `input_config`: especifica los nodos y/o las unidades de cómputo que se utilizarán para realizar la instalación de la rutina. Su formato en cada uno de los niveles es el siguiente:
  - **Niveles 0 y 1**: contiene una línea por cada configuración de unidades de cómputo a usar en cada nodo. Su formato es:

    $$nodo\ n_cores\ n_gpu\ id_gpu_0 \ldots n_mics\ id_mic_0 \ldots$$

  - **Nivel 2**: contiene el nombre de los nodos a usar. Estos nodos pueden ser virtuales, es decir, obtenidos tras realizar una instalación a nivel de nodo con un subconjunto de unidades básicas del mismo. Si se especifican varios nodos virtuales pertenecientes a un mismo nodo, éstos han de ser disjuntos en cuanto a GPUs y/o MICs se refiere, es decir, no pueden hacer uso de la misma unidad básica de procesamiento. Asimismo, la suma del número de cores no ha de exceder el máximo soportado por la CPU multicore del sistema. Su formato es:

    $$nodo_1\ nodo_2 \ldots nodo_j$$

Tal como se ha mencionado, el software incluye unos *tests* que permiten probar el funcionamiento de la metodología de optimización haciendo uso de rutinas que invocan a rutinas básicas previamente instaladas en el sistema. Estos ficheros se encuentran en el directorio `HATLib/test/library/code/main`. La aplicación permite ejecutarlos desde la opción `Test Library` del menú principal.

No obstante, el usuario puede modificar estos *tests* o crear los suyos propios. Para ello, ha de disponer, en primer lugar, de la librería estática que permite generar el software con la opción `Create Library` del menú. A continuación, puede usar como referencia alguno de los *tests* disponibles y reutilizar tanto el Makefile contenido en `HATLib/test/library/compile` para compilar el código, como el fichero con los datos de entrada para el *test* en cuestión (contenido en `HATLib/test/library/input_data`).

Las rutinas utilizadas en los diferentes *tests* hacen uso de las funciones ofrecidas por la API de la librería `libhatlib`. Por un lado, contiene funciones que permiten establecer la configuración de la plataforma a utilizar, y por otro, dispone de funciones destinadas a seleccionar los valores de los parámetros algotítmicos con los que ejecutar las rutinas instaladas en el sistema con la configuración hardware establecida. Para hacer un uso correcto de las mismas, se han de invocar en el orden en que aparecen en los *tests*. Estas funciones son:

```
int HATLIB_setNodeConf(int nCores, int nGPU, int nMIC);

sNodeConfig *HATLIB_setProcessConf(int nThr, int nGPU, int nMIC, \
                                   int *gpuIds, int *micIds, \
                                   int *micThr);

int HATLIB_getDecision(int level, sNodeConfig *, char *routine, \
                       sRoutineParams *, sAutoDecision *);

double HATLIB_execRoutine(char *routine, sRoutineParams *, \
                          sAutoDecision *);

double HATLIB_tuneRoutine(int level, sNodeConfig *, \
                          char *routine, sRoutineParams *);
```

A continuación se describe la funcionalidad de cada una de ellas y se muestran las estructuras de datos necesarias para hacer uso de las mismas:

- `HATLIB_setNodeConf`: establece la configuración de ejecución a usar en el nodo, indicando el número de cores (físicos o lógicos), el número de GPUs y/o MICs. Admite la configuración en la que se indique el valor 0 para el número de cores de CPU y un valor > 0 para el número de GPUs y/o MICs.

- `HATLIB_setProcessConf`: permite establecer una subconfiguración de la plataforma establecida con la función anterior. Esta función es especialmente útil si el usuario pone en marcha varios threads OpenMP y quiere que cada uno gestione un subconjunto de unidades de cómputo del nodo para resolver una parte del problema. Podría considerarse como una forma de gestionar nodos virtuales mediante threads OpenMP.

- `HATLIB_getDecision`: aplica la metodología de auto-optimización para obtener la mejor configuración de ejecución a usar (valores de los parámetros algorítmicos) en la rutina de nivel inferior (`routine`) a la que invoca desde el nivel actual (`level`), utilizando la configuración hardware establecida (`sNodeConfig`) y el tamaño de problema asignado (`sRoutineParams`).

- `HATLIB_execRoutine`: utiliza los valores de los AP decididos por el proceso de auto-optimización aplicado en la función anterior (`sAutoDecision`) para ejecutar la rutina de nivel inferior a la que invoca (`routine`). Como resultado, devuelve el tiempo de ejecución que se obtiene. Si se ha indicado un valor 0 para el número de cores de CPU, el proceso de *auto-tuning* no se realiza para la CPU, pero sí se ponen en marcha threads OpenMP al ejecutar la rutina con la decisión tomada para cada GPU y/o MIC, dado que es necesario para llevar a cabo las transferencias de datos hacia/desde cada uno de los coprocesadores especificados.

- `HATLIB_tuneRoutine`: invoca internamente a las dos anteriores.

Las funciones descritas hacen uso de determinadas estructuras de datos que almacenan información referente tanto a la configuración hardware establecida (`sNodeConfig`) como a la decisión de ejecución (valores de los parámetros algorítmicos) obtenida por la metodología de auto-optimización (`sAutoDecision`). Concretamente, esta estructura de datos almacena el óptimo número de threads a usar en CPU y/o MIC para el tamaño de problema actual, así como la carga

de trabajo asignada a cada unidad de cómputo (CPU, GPU y MIC), el tiempo de ejecución estimado y el rendimiento (en GFlops) que se obtendría al ejecutar la rutina con dichos valores para los parámetros. Además, en dichas funciones se utiliza una estructura de datos para almacenar información propia de la rutina a la que se invoca (`sRoutineParams`). En este caso, esta estructura de datos tiene un campo de tipo `sGemmParams`, que se ha utilizado para almacenar los valores de los parámetros propios de la rutina de multiplicación de matrices instalada de forma jerárquica en la plataforma computacional. De forma similar se haría con otras rutinas instaladas en el sistema y que fuesen invocadas desde rutinas de mayor nivel. A continuación se muestran los campos que contiene cada una de las estructuras de datos mencionadas:

```
typedef struct _Node_Config {
        int nCores, nGPU, nMIC;
        sGPU_Config *gpu_conf;
        sMIC_Config *mic_conf;
} sNodeConfig;

typedef struct _Auto_Decision {
        int thrCPU, cpuLoad;
        int nGPU, *gpuIds, *gpuLoads;
        int nMIC, *micIds, *micLoads, *micThreads;
        double time, gflops;
} sAutoDecision;

typedef struct _Routine_Parameters {
        sGemmParams *gemm_params;
} sRoutineParams;
```

A.4 Manual de Desarrollador

En esta sección se abordan los aspectos principales para que cualquier usuario desarrollador pueda extender o mejorar la aplicación software presentada al comienzo del presente documento. El software ha sido desarrollado íntegramente utilizando el lenguaje de programación C. Por tanto, es necesario tener nociones básicas de programación en dicho lenguaje.

Se comenzará indicando cuáles son los principales scripts utilizados por la aplicación y cómo habría que modificarlos en el caso de querer utilizar otros compiladores y/o librerías. A continuación, se detallará cómo se han de modificar los ficheros de código fuente para extender la funcionalidad del software.

A.4.1 Modificación de scripts

El script principal a partir del cual se ejecuta el software y se pueden llevar a cabo las diferentes tareas comentadas en el manual de usuario, tiene por nombre `run.sh`. Este script se encuentra en el directorio principal de la aplicación y contiene el conjunto de funciones que se invocan en cada uno de los pasos que se visualizan con cada una de las opciones del menú principal. En función de la opción seleccionada, se invocarán diferentes scripts, algunos para inicializar el entorno de ejecución del sistema y otros para compilar los ficheros de código fuente necesarios e instalar las rutinas de forma jerárquica.

La mayoría de estos scripts se han creado de la forma más genérica posible, siendo capaces de detectar de forma automática las características de la plataforma computacional completa, compilar la aplicación (resolviendo dependencias) y ejecutarla mediante el sistema de colas existente, sin necesidad de intervención alguna por parte del usuario. Por tanto, sólo se indican aquellos scripts que habría que modificar en caso de querer utilizar compiladores o librerías diferentes de las establecidas por defecto.

Concretamente, estos scripts son `init_env.sh` e `init_env_gpu.sh`, que se encuentran almacenados tanto en el directorio `HATLib/code/install/compile` como en `HATLib/code/test/library`. La diferencia entre ambos scripts radica

en que `init_env.sh` incluye el módulo que carga los compiladores y librerías comunes a todos los nodos (por ejemplo, `intel/2017_u1` y `openmpi/1.6.4-intel`) e `init_env_gpu.sh` sólo incluye el módulo que carga el entorno de ejecución (compiladores y variables de entorno) y la librería utilizada en los nodos que contienen GPUs (por ejemplo, `cuda/7.5` de NVIDIA). Por tanto, si se usan otros compiladores (o versiones de los mismos) o librerías de álgebra lineal, habría que especificar en estos scripts el módulo correspondiente disponible en el sistema. Además, habría que modificar los valores de las variables utilizadas para la compilación en el Makefile contenido en dichos directorios, con el fin de compilar adecuadamente los ficheros de código fuente correspondientes.

A.4.2 Modificación de código fuente

Tal como se ha comentado, esta versión inicial de la aplicación permite llevar a cabo la instalación jerárquica de la rutina de multiplicación de matrices en diferentes niveles hardware de la plataforma heterogénea. Cada vez que se realiza la instalación en cada uno de los niveles, se generan un conjunto de ficheros que se utilizan durante el proceso de instalación. Algunos contienen información referente al sistema y otros se utilizan para enviar el trabajo al sistema de colas. Estos ficheros se pueden localizar en los directorios `def_config` y [`sh | pbs`]`_files`, respectivamente, del directorio `HATLib/code/install`.

La información que se almacena tras la instalación, difiere en función del nivel hardware de la plataforma en el que se haya instalado la rutina. En el nivel inferior de la jerarquía (nivel 0), se genera un fichero con la información de instalación para cada elemento básico de procesamiento (CPU, GPU y MIC), dado que la instalación se lleva a cabo de forma simultánea e independiente (sin tener en cuenta las comunicaciones intra-nodo) en cada uno de ellos. La instalación también se puede realizar en los elementos de procesamiento de varios nodos de forma simultánea. En el nivel 1, la información que se almacena para cada nodo (o nodo virtual) depende de la técnica (*exhaustiva, guiada...*) que se utilice para la obtención del valor de los parámetros algorítmicos. Lo mismo sucede en nivel 2 de la jerarquía. No obstante, en todos los casos la información de instalación contenida en los ficheros se almacena en forma de array de structs, con el fin

de poder incluirla en ficheros cabecera al realizar la instalación en el siguiente nivel de la jerarquía. Se ha optado por esta forma de almacenamiento porque permite acelerar la búsqueda de información al aplicar la metodología de auto-optimización. Cabe recordar que estos ficheros se encuentran dentro del directorio `database` en el subdirectorio correspondiente al nivel en el que se ha realizado la instalación. Asimismo, en estos directorios se almacenan un conjunto de ficheros de *log* para registrar las instalaciones realizadas en las unidades de cómputo de cada nivel y evitar así realizar instalaciones duplicadas.

Por otro lado, el software desarrollado también permite hacer uso de la metodología de auto-optimización con rutinas de mayor nivel (como la multiplicación de Strassen) que invocan a la rutinas básicas instaladas en el sistema para conseguir una ejecución eficiente. No obstante, la funcionalidad del software se puede seguir extendiendo para que considere nuevas rutinas o librerías de álgebra lineal. Para ello, hay que adaptar determinados ficheros de código fuente y añadir otros nuevos. A continuación se indican los pasos a seguir:

- **Añadir Rutina**:
  - Adaptar los ficheros `.c` de `HATLib/code/install/code/main` para que contemple la rutina que se quiere incluir, es decir, añadir las funciones y/o líneas de código del mismo modo en que aparecen para la rutina `gemm` de multiplicación de matrices.
  - Hacer lo propio con los ficheros `.c` del directorio `io` contenido en el directorio anterior. Además, añadir en el fichero `routines_structs.h` una estructura de datos similar a `sGemmParams` pero con los campos necesarios para almacenar los parámetros de la rutina actual. Asimismo, incluir un campo del tipo creado en la estructura `sRoutineParams`.

  En el directorio `HATLib/code/install/code/include`:

  - Definir el coste de la rutina en el fichero `flops.h`, en términos del número total de instrucciones de suma y multiplicación ejecutadas. Dicha información se puede obtener de http://www.netlib.org/lapack/lawnspdf/lawn41.pdf.

– Si la rutina añade nuevos parámetros algorítmicos (como el tamaño de bloque, el nivel de recursión...) en algún nivel de la jerarquía, añadir el campo correspondiente en la estructura de datos `sInstData` (nivel 0), `sInstDataNode` (nivel 1) o `sInstDataCluster` (nivel 2) del fichero `install_structs.h`.

– Incluir en el fichero `get_data_L1.h` la definición de la rutina híbrida para la rutina actual, de forma similar a la que ya contiene para la rutina de multiplicación de matrices.

– Crear una copia del fichero `hybrid_gemm_mpi.h` y adaptar su nombre y su contenido para que se corresponda con la rutina actual.

– Añadir en el fichero `routines_structs.h` una estructura de datos similar a `sGemmParams` pero con los campos necesarios para almacenar los parámetros de la rutina actual. Asimismo, incluir un campo del tipo creado en la estructura `sRoutineParams`.

En el directorio `HATLib/code/install/code/src`:

– Crear los ficheros de código fuente relativos a GPU y MIC para los niveles 0 `L0` y 1 `L1` de la jerarquía.

– Modificar el fichero `get_data_L1.c` para que también contemple la llamada a la función híbrida en MPI para la nueva rutina.

– Crear una copia del fichero `hybrid_gemm_mpi.c` y adaptar su nombre y su contenido para que implemente las funciones para la rutina actual.

– Añadir en el fichero `hybrid_routine.c` la función que implementa el esquema algorítmico híbrido para la nueva rutina.

– Adaptar los ficheros `install_L1.c`, `install_L2_aux.c`, `wd_methods.c` e `install_routine_L0.c`, para que contemplen la rutina que se quiere incluir, es decir, añadir las funciones y/o líneas de código del mismo modo en que aparecen para la rutina `gemm` de multiplicación matricial.

En el directorio `HATLib/code/library`:

  - Teniendo en cuenta las indicaciones anteriores para los ficheros de los directorios `include` y `src` del directorio `HATLib/code/install/code`, llevar a cabo los pasos que sean necesarios con los ficheros incluidos en los directorios `include` y `src` del directorio actual.

- **Añadir Librería**: en los ficheros `.c` en los que se invoque en la CPU a la rutina `dgemm` de la librería MKL, incluir la rutina correspondiente de la nueva librería estableciendo una distinción de casos para que se ejecute la rutina adecuada en función de la librería utilizada. Asimismo, incluir el fichero de cabecera `.h` de la librería usando directivas de compilación condicional (#ifdef... #elif... #endif). En el caso de usar una librería para GPU, crear un fichero `.c` similar al existente para la rutina `dgemm` de la librería cuBLAS y realizar la implementación utilizando la rutina ofrecida por la nueva librería.