UNIVERSIDAD POLITECNICA DE VALENCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

Optimización y Paralelización del Software para el Cálculo de Campos Electromagnéticos MEATSS

José Ginés Picón López

Directores

Vicente Boria Esbert
Alejandro Álvarez Melcón
Fernando David Quesada Pereira
Domingo Giménez Cánovas

Valencia, 11 de septiembre de 2012

Resumen

En este proyecto se aborda la optimización y paralelización del software MEATSS (Microstrip Electromagnetic Analysis Tool for Space Systems) para el cálculo de los campos electromagnéticos en líneas de transmisión de tipo microstrip. La necesidad del estudio de este tipo de estructuras parte de un proyecto para la Agencia Espacial Europea (ESA) titulado New Investigations of RF Breakdown in Microwave Transmission Lines (referencia No. AO-5086). El objetivo de dicho estudio es la investigación de los fenómenos de descarga de Corona y Multipactor que pueden ocasionarse en el interior de componentes de microondas construidos en tecnología microstrip utilizados en satélites espaciales.

Las altos costes computacionales de los métodos empleados hacen que sea necesaria la optimización y paralelización del código con el fin de abordar geometrías complejas. Para ello, en primer lugar, se ha realizado un estudio de *profiling* del código con el objetivo de detectar las secciones más lentas empleando el software `gprof`. Una vez detectadas las zonas críticas, se ha procedido a la optimización mediante la sustitución de funciones de MEATSS por las equivalentes optimizadas de la librería Intel MKL. A continuación se muestra el estudio y paralelización con OpenMP y finalmente se ha explotado el paralelismo a 2 niveles empleando MKL y OpenMP junto con algunos conceptos de autooptimización.

Para aplicar de forma eficiente código paralelo en sistemas computacionales de gran dimensión es necesario estudiar la optimización y autooptimización del código. Para la ejecución eficiente de MEATSS se deben considerar combinaciones adecuadas de paralelismo OpenMP y MKL y seleccionar la rutina MKL a utilizar en la resolución de los sistemas de ecuaciones que en él aparecen. De esta forma el tiempo de ejecución se reduce de forma importante, lo que posibilita abordar problemas de mayor dimensión en sistemas de memoria compartida.

Índice general

Capítulo 1

Introducción

En este capítulo introductorio describiremos brevemente los efectos de Corona y Multipactor que se producen en los dispositivos bajo estudio. Seguidamente realizaremos una descripción del software de partida y se plantearán los objetivos que se desean conseguir. A continuación se expondrá el estado del arte, en el que se hará referencia a los paquetes software que se contemplaron en una fase inicial y que llevaron a la decisión de desarrollar MEATSS, y se analizarán técnicas de paralelismo utilizadas en la resolución de problemas científicos, especialmente de electromagnetismo, así como algunos trabajos de optimización y autooptimización de código paralelo. En las últimas secciones se planteará la metodología de trabajo y el contenido de la memoria.

1.1. Efectos de corona y multipactor

Los fenómenos de descarga de Corona y Multipactor [Pé09] son fenómenos físicos inherentes al comportamiento de los electrones o gases ionizados en presencia de campos electromagnéticos intensos producidos por niveles de potencia muy elevados en el interior de estructuras metálicas cerradas, y conducen a serios daños en los dispositivos electrónicos. Por ello, resulta de vital importancia poder predecirlos adecuadamente y evitar este tipo de problemas. Los altos niveles de potencia previamente mencionados son necesarios en los procesos de comunicaciones espaciales, por lo que no es conveniente su reducción si se quieren mantener unos niveles de señal a ruido adecuados. Por lo tanto, la única opción que resta para lograr tal cometido es una estimación precisa del comportamiento electromagnético de este tipo de dispositivos microstrip encapsulados, de forma

que sea posible predecir las posibles trayectorias de los electrones inmersos en los campos del dispositivo para el caso del efecto Multipactor. La meta final de estos estudios es la determinación de los elementos y geometrías que resulten críticos para los niveles de los campos electromagnéticos, de forma que sea posible realizar diseños que minimicen los mismos y con ello los riesgos de aparición de las descargas.

1.2. Software de partida

MEATSS [VPDM07] es un software basado en la técnica de la ecuación integral con equivalente de volumen para el tratamiento de los dieléctricos y la formulación de equivalente superficial para tratar los conductores usando las funciones de Green. Su objetivo es la predicción de dichos efectos de alta potencia. El uso de este tipo de funciones de Green es adecuado para el tratamiento de geometrías complejas, pero el número de incógnitas crece rápidamente con la complejidad de la estructura. El reto más importante desde el punto de vista computacional es la inversión y llenado de grandes sistemas lineales de ecuaciones complejas densas. Debido a esta gran carga computacional se hace necesaria la optimización y paralelización del código.

El código está escrito en los lenguajes C++ y FORTRAN-90. Los datos de entrada y salida se almacenan en ficheros de texto en los que se especifica la geometría y el tipo de análisis a realizar. Debido a la amplitud del código, el proyecto se ha centrado en el estudio del módulo que realiza el análisis de los parámetros de dispersión, también llamados de *scattering*, en un rango de frecuencias. Las técnicas aplicadas pueden emplearse de manera análoga para otros tipos de análisis incluidos en MEATSS, por ejemplo para el cálculo de los campos electromagnéticos en una determinada zona de la estructura.

Antes de continuar es necesario tener en consideración varios aspectos sobre las llamadas a subrutinas FORTRAN desde C++:

- **Parámetros.** Cuando se quiere construir una interface entre C++ y FORTRAN hay que tener en cuenta que C++ pasa los parámetros por valor mientras que FORTRAN los pasa por referencia. Esto significa que en el momento de definir las subrutinas que serán llamadas desde C++ habrá que

definir y pasarle a las rutinas de FORTRAN los parámetros por referencia para que estas puedan trabajar. Pasar un parámetro por referencia significa pasar la dirección de memoria donde este parámetro se encuentra.

- **Nombre de la subrutina.** Otro de los aspectos a tener en cuenta en la construcción de la interface C++-FORTRAN es el nombre de las subrutinas. El compilador de FORTRAN agrega un guión bajo a todas las subrutinas en el momento de generar los objetos a partir del código fuente. Esto se tiene que tener presente, ya que al realizar la llamada a la subrutina de FORTRAN se tendrá que concatenar un guión bajo a la misma.

- **Índices de matrices y vectores.** Es importante notar que el orden en que se almacenan los valores en los vectores y matrices cambia de acuerdo al lenguaje que se esté utilizando. Así, `Arr(i,j)` en FORTRAN sería en C++ algo similar a `Arr[j][i]`. También es importante recordar que los indices de los vectores y matrices en FORTRAN empiezan con el uno, a diferencia de en C++, que empieza con el cero.

Los detalles de formulación están desarrollados de manera detallada en [Pé09] y quedan fuera del alcance del proyecto. Básicamente, la matriz obtenida por el método de los momentos es dividida en dos matrices: una estática, válida para todas las frecuencias, y otra dinámica que necesita ser calculada para cada frecuencia. La matriz estática requiere más puntos de integración porque tiene la singularidad de las funciones de Green. Sin embargo, solamente necesita ser calculada una vez. Por el contrario, la parte dinámica de la matriz puede ser calculada con pocos puntos en la integración numérica para conseguir una precisión adecuada, pero debe evaluarse para cada frecuencia.

En la figura 1.1 se muestra una posible estructura que será utilizada para la realización de las simulaciones a lo largo del proyecto. Se trata de una línea microstrip simple modelada con una versión de evaluación del software para la generación de mallados Gid [Gid]. Se han definido dos puertos en extremos opuestos para inyectar y recibir potencia respectivamente.

1.3. Objetivos

El objetivo principal del proyecto consiste en reducir los tiempos de ejecución del software MEATSS mediante la optimización y paralelización de su código

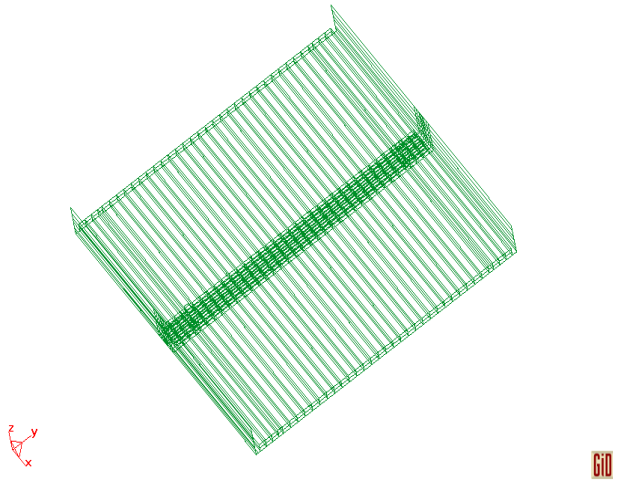

Figura 1.1: Vista 3D de la geometría utilizada en los experimentos

fuente. El cumplimiento de este objetivo nos va a permitir tanto aumentar la resolución de los problemas actuales, utilizando una discretización más fina, como el poder abordar otros problemas, hasta ahora no contemplados debido a sus altas necesidades computacionales. Para conseguir este objetivo se han planteado varios subobjetivos:

- Realización de un análisis de *profiling* para detectar las regiones con mayor carga computacional para trabajar en su paralelización y optimización.

- Identificación y reemplazo de funciones del código por sus equivalentes de la librería Intel MKL, estudiando las distintas posibilidades en el uso de rutinas básicas de álgebra lineal y el número de *cores* a utilizar para reducir el tiempo de ejecución.

- Identificación de zonas del código que permitan paralelización con el paradigma de memoria compartida, codificación y optimización en OpenMP.

- Estudio de la paralelización híbrida OpenMP+MKL, analizando la mejora en el tiempo de ejecución con paralelismo de dos niveles con respecto al de un único nivel.

1.4. Estado del arte

Antes de tomar la decisión de empezar a desarrollar el software MEATSS fueron analizados varios paquetes para el cálculo de campos electromagnéticos en las estructuras bajo estudio. Entre la lista de paquetes software que se evaluaron se encontraban ADS [Tec], Ansoft - HFSS [Ans], Microstripes [Mic] y FEKO [FEK]. Aunque estas soluciones comerciales permiten simular los campos electromagnéticos en estructuras arbitrarias, no hay ningún software actualmente que permita realizar estimaciones del comportamiento de los electrones.

Una vez analizados los paquetes para el análisis electromagnético anteriormente mencionados en las estructuras bajo estudio, y viendo que ninguno se adaptaba a las necesidades y a la flexibilidad esperada, el Grupo de Electromagnetismo Aplicado a las Telecomunicaciones (GEAT) decidió elaborar de forma propia el software de simulación MEATSS, utilizando técnicas eficientes de Ecuación Integral que permitían obtener resultados precisos reduciendo el coste computacional con respecto a las soluciones comerciales.

Por otra parte, otra de las razones por las que se desarrolló MEATSS fue porque se quería tener acceso al código fuente para tener control total sobre la precisión y las capacidades del software. En particular, se quería poder dar solución a posibles problemas con estructuras muy particulares. La elaboración de este software supuso un reto para el Grupo de Electromagnetismo Aplicado a las Telecomunicaciones, ya que para la elaboración del mismo fue necesaria la integración y mejora de diversas técnicas de análisis desarrolladas por el grupo en los últimos años.

En este proyecto se aborda la paralelización de MEATSS para reducir el tiempo de resolución de los problemas que con él se resuelvan. El paralelismo se utiliza desde hace aproximadamente tres décadas para la aceleración de la resolución de problemas científicos y de ingeniería [GGKK03, AGMV08], en algunos casos con aplicaciones en electromagnetismo [JSL97, PAGAMQ12, ZSM+07, ZOK].

Hay varios paradigmas de programación paralela según el tipo de sistema computacional que se utilice. En sistemas donde los distintos elementos computacionales tienen acceso directo a todos los bloques de la memoria se utiliza el paradigma de memoria compartida, para el que el estándar actual es el entorno OpenMP [Opeb, CMD+02]. Cuando la memoria está dividida en partes que están

asociadas a elementos de computación distintos, es necesaria la comunicación entre estos elementos para acceder a datos en zonas de memoria a las que no se tiene acceso directo. En este caso se usa el paradigma de paso de mensajes, del que el estándar de facto actual es el entorno MPI (Message-Passing Interface) [MPI, SG98]. Más recientemente se ha empezado a realizar computación de proposito general en tarjetas gráficas (GPU, Graphic Processing Unit), que permiten una computación de altas prestaciones con un coste reducido pero que tienen normalmente una mayor dificultad de programación. Se puede considerar que usan un paradigma SIMD (Simple Instruction Multiple Data), y se programan con CUDA [CUD] si son de NVidia o con OpenCL [Opea], que pretende convertirse en un estándar para este tipo de programación. Es también posible utilizar paralelismo híbrido, combinando distintos paradigmas, de forma que se pueden utilizar todos los componentes de un sistema computacional jerárquico como puede ser un *cluster* con varios nodos *multicore* y con tarjetas gráficas.

En este proyecto se trabajará en la paralelización en sistemas *multicore* (memoria compartida) del software MEATSS. En la actualidad los componentes básicos de los sistemas computacionales son *multicore*, desde portátiles duales o quad, a servidores con varios procesadores *multicore* que comparten la memoria, hasta grandes sistemas NUMA (*Non-Uniform Memory Access*) donde los distintos *cores* pueden acceder directamente a todas las zonas de memoria pero con un coste distinto dependiendo de dónde se encuentren los datos. En todos estos sistemas se utiliza paralelismo OpenMP, pero también es posible utilizar paralelismo de forma implícita con llamadas a rutinas *multithread* que ya estén paralelizadas. En sistemas de gran dimensión, donde el acceso a la memoria no es uniforme, una práctica habitual para mejorar el uso de la memoria y consecuentemente reducir el tiempo de ejecución es usar paralelismo multinivel [GL09, CGG12, CCGV12], con el que se combina paralelismo OpenMP con rutinas *multithread* con paralelismo implícito.

Dado que el software paralelo se utiliza en la resolución de grandes problemas científicos en grandes sistemas, se estudian técnicas de optimización de código para intentar que el software use estos sistemas de forma eficiente y que se reduzca el tiempo de resolución de los problemas. El tener a nuestra disposición códigos paralelos eficientes no asegura la utilización del código de forma óptima y el uso adecuado del software en sistemas computacionales complejos, por lo que se utilizan técnicas de autooptimización para desarrollar software paralelo con capacidad de adaptarse a las características del sistema y del problema a resolver.

De esta manera se hace un uso eficiente del software independientemente del sistema donde se utilice y de los conocimientos de paralelismo que tenga el usuario. Las técnicas de optimización y autooptimización se aplican en distintos campos [Fri98, GAMR03, KKH04, Dat09], y especialmente en rutinas de álgebra lineal [WPD01, HR05, CGG04], que constituyen el componente básico en la mayoría del software científico, y que en particular se usan desde MEATSS.

1.5. Metodología

Para alcanzar los objetivos se va a seguir la siguiente metodología de trabajo:

- En primer lugar se va a realizar un exhaustivo análisis del código utilizando el paquete de *profiling* `gprof` [FS].
- De los resultados extraídos del análisis se procederá a la localización de las secciones de código más lentas y a la sustitución de las mismas por rutinas optimizadas de la librería MKL [Int]. La sustitución se realizará tanto con la versión secuencial como con la versión para memoria compartida de la librería.
- Una vez concluido este apartado procederemos a paralelizar el código utilizando OpenMP. Para estudiar la reducción de tiempos se trabajará con mallados de distintos tamaños.
- Concluiremos combinando el paralelismo de la librebría MKL con el de OpenMP.

Para realizar todas las pruebas se va a utilizar el supercomputador Ben Arabí [Fun] del Centro de Supercomputación de la Fundación Parque Científico de Murcia, lo cual nos va a permitir realizar y validar los experimentos en dos tipos de sistemas: nodos de un *cluster* Intel Xeon con memoria distribuida y una máquina Intel Itanium con memoria compartida. Adicionalmente, en algunos casos se utilizarán otros sistemas *multicore* (portátiles y servidores) para algunos experimentos puntuales, lo que nos permitirá concluir resultados generales y no dependientes del sistema particular en que se trabaje. También se harán experimentos variando la complejidad de los problemas, para obtener conclusiones independientes del tamaño del problema y que nos permitan predecir el comportamiento del software en la resolución de problemas de gran dimensión.

1.6. Contenido de la memoria

Al margen de este capítulo inicial, la memoria del proyecto consta de un primer capítulo (capítulo 2) sobre las herramientas computacionales empleadas en la realización del proyecto. El capítulo se ha dividido en dos apartados, uno para las herramientas software y otro para las herramientas hardware.

A continuación el capítulo 3 se dedica a la utilización de la librería MKL. Se explica la estructura del software MEATSS, apoyándonos en el *profiler* del capítulo anterior. Una vez aclarada la estructura, se procederá a la sustitución de rutinas por otras optimizadas de MKL y a la realización de un estudio experimental variando la complejidad de las estructuras bajo estudio. Los experimentos se realizarán en las dos arquitecturas de Ben Arabí. El capítulo concluirá mostrando las conclusiones sobre la utilización de la librería MKL en los sistemas utilizados.

En el capítulo 4 se estudiará la utilización de paralelismo de memoria compartida, mostrando los cambios realizados en el software de partida para poder obtener una versión OpenMP, que también se estudiará en el supercomputador Ben Arabí.

El paralelismo híbrido OpenMP+MKL se estudia en el capítulo 5, donde se analizará la mejor manera de combinar los dos tipos de paralelismo, de forma que se mejore al acceso a las distintas zonas de la memoria compartida pero de acceso no uniforme (lo que es especialmente interesante en Ben, que tiene una jerarquía de memorias más amplia). Se estudia así la posibilidad de reducción del tiempo de ejecución de las versiones con un único nivel de paralelismo.

El capítulo 6 incluye algunas recomendaciones sobre la instalación y ejecución del software paralelo desarrollado, de manera que se puedan utilizar técnicas de optimización y autooptimización que ayuden a su ejecución de forma eficiente.

En el capítulo final se resumen las conclusiones del trabajo realizado y se comentan algunas posibles líneas de trabajo futuro.

Capítulo 2

Herramientas computacionales

En este capítulo se realizará una descripción de las herramientas utilizadas en el desarrollo del proyecto. Las herramientas se dividirán en herramientas software y herramientas hardware. En el apartado de herramientas software se presentará el paquete para el análisis de *profiling* `gprof`, la librería matemática Intel MKL, el *debugger* de Intel `idb` [Int] y la API para memoria compartida OpenMP. En la sección de herramientas hardware se presentará la arquitectura del supercomputador Ben Arabí de la Fundación Parque Científico de Murcia, que será el que se utilice en la mayoría de los experimentos, y del servidor Saturno, del Grupo de Computación Científica y Programación Paralela de la Universidad de Murcia, que se utilizará en algunos experimentos de forma ocasional.

2.1. Herramientas software

A continuación se revisan las herramientas software que se utilizan en la optimización y paralelización del software de partida, cuyas versiones son:

- GNU gprof 2.16.91.0.5
- idb 11.1
- icc 11.1.059
- MKL 10.2.2.025
- ifort 11.1.059

2.1.1. gprof

Los *profiler* nos ofrecen un detallado análisis temporal de nuestros programas. Del análisis de los mismos se desprende información muy útil como, por ejemplo, cuántas veces ha sido llamada una función, quién la llamó y cuánto tiempo se gastó en su ejecución. Esta información nos puede ayudar a la hora de localizar las secciones más lentas de nuestro código. El software empleado para este fin es `gprof` [FS].

El primer paso para generar información de *profiling* es compilar y *linkar* con la opción de *profiling* activada: `-p -g`. El siguiente paso es ejecutar el programa. La ejecución del programa será más lenta de lo habitual debido al tiempo consumido en recopilar y generar datos de *profiling*. El programa debe terminar normalmente para que la información obtenida sea útil. Al terminar la ejecución se crea el archivo `gmon.out` en el directorio de trabajo. Si ya existiera un archivo con nombre `gmon.out` será sobrescrito. Este archivo contiene la información de *profiling* que será interpretada por `gprof`.

Una vez obtenido `gmon.out` podemos extraer la información de él mediante el comando `gprof`:

gprof opciones [ejecutable] gmon.out

La información que se nos devuelve se divide en dos tablas. Una primera llamada `flat profile`, en la que se nos muestra cuánto tiempo gasta nuestro programa en cada función y cuántas veces ha sido llamada dicha función. Y una segunda tabla denominada `call graph`, que nos da información de, para cada función, desde qué funciones ha sido llamada y a qué funciones llama, y además nos dice cuántas veces.

A continuación se muestran unos fragmentos de los *profiles* obtenidos para el programa de partida con 204 frecuencias y utilizando un mallado con el que se genera un problema de alta complejidad:

Flat profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ks/call  Ks/call  name
 66.24   6380.83  6380.83   672384     0.00     0.00  zgemm_
```

```
 10.50   7392.02  1011.19 128520612    0.00     0.00  zdd_din_rt_
  7.40   8104.89   712.87 45524709     0.00     0.00  zgemv_
  4.53   8541.72   436.83 159306048    0.00     0.00  zrd_din_rt_
  3.15   8844.81   303.09 1584114134   0.00     0.00  quad3drec_
  1.26   8966.65   121.84 2328327468   0.00     0.00  green_functions3d_
  0.92   9055.39    88.74                             __intel_new_memcpy
  0.88   9140.52    85.13                             cexp
  0.71   9208.67    68.15      204     0.00     0.01  dynamicpart1_
  0.70   9276.45    67.78 49481424     0.00     0.00  zrr_din_micro_
  0.51   9325.81    49.36 341950455    0.00     0.00  intpoints_cell2cell_
  0.43   9367.15    41.34                             __intel_new_memset
  0.41   9406.35    39.21 419998260    0.00     0.00  quad3dbck2_
  0.39   9444.04    37.69  4065366     0.00     0.00  zcopy_
  0.26   9468.97    24.93 260532860    0.00     0.00  quad3dr2_
  0.21   9488.91    19.94                             _intel_fast_memcpy.J
  0.19   9507.50    18.59  1052436     0.00     0.00  zgeru_
  0.18   9524.40    16.90                             _intel_fast_memcpy.A
  0.17   9541.16    16.76                             get_fp_controlword
```

Observando el extracto anterior del `flat profile`, vemos que el 66 % del tiempo total de la ejecución se consume en la rutina de multiplicación de matrices densas complejas de doble precisión de BLAS `zgemm`, seguida de la rutina `zdd_din_rt` con un 10 % del tiempo. Por tanto, podría pensarse que una mejora en la multiplicación de matrices producirá una reducción importante en el tiempo de ejecución, pero hay que tener en cuenta que el tiempo invertido en las multipicaciones (6380 segundos) se divide en un gran número de multiplicaciones, con lo que cada multiplicación consume unos milisegundos, con lo que hay poco margen de mejora, y es preferible paralelizar a un nivel mayor de computación.

La información proporcionada por el `flat profile` es parcial e insuficiente para poder interpretar qué es lo que está ocurriendo en nuestro código, al no saber quién llama a cada función. Para obtener una información más completa tenemos que recurrir al `graph profile`.

En el `graph profile` cada función o subrutina es representada en forma de árbol de llamadas. La línea que representa una función en su árbol de llamadas se llama `function line` y se identifica por un índice entre corchetes situado en la columna más a la izquierda. Las líneas por encima de ella son las `parent lines` y las líneas por debajo de ella se llaman `descendant lines`. Se muestra el `graph profile` correspondiente al `flat profile` anterior:

Graph profile:

```
  index % time    self  children    called     name
                                                <spontaneous>
[1]       96.9    0.00 9330.85                 main [1]
                0.00 9330.85       1/1           microstrip(int, char* const*) [2]
-----------------------------------------------
                0.00 9330.85       1/1           main [1]
[2]       96.9    0.00 9330.85       1         microstrip(int, char* const*) [2]
                0.00 7173.44     204/204         dynamicpart2_ [3]
               68.15 2054.11     204/204         dynamicpart1_ [9]
                0.00   35.13       1/1           staticmeatss_ [23]
                0.00    0.01     204/204         scatteringparameter_ [84]
                0.00    0.00       1/1           Read_file(char const*, double&, ... int&) [782]
-----------------------------------------------
                0.00 7173.44     204/204         microstrip(int, char* const*) [2]
[3]       74.5    0.00 7173.44     204         dynamicpart2_ [3]
                0.00 7173.39     204/204         solve_system_ [5]
                0.01    0.04     204/204         excitation_evaluation_ [71]
-----------------------------------------------
                0.00 7173.39     408/408         solve_system_ [5]
[4]       74.5    0.00 7173.39     408         zsysv_ [4]
                0.01 7135.44     204/204         zsytrf_ [6]
                0.03   37.91     204/204         zsytrs_ [21]
                0.00    0.00     816/277234284     lsame_ [50]
                0.00    0.00     408/612         ilaenv_ [109]
-----------------------------------------------
                0.00 7173.39     204/204         dynamicpart2_ [3]
[5]       74.5    0.00 7173.39     204         solve_system_ [5]
                0.00 7173.39     408/408         zsysv_ [4]
-----------------------------------------------
                0.01 7135.44     204/204         zsysv_ [4]
[6]       74.1    0.01 7135.44     204         zsytrf_ [6]
                0.84 7134.43   16524/16524       zlasyf_ [7]
                0.01    0.16     204/204         zsytf2_ [58]
                0.00    0.00     408/277234284     lsame_ [50]
                0.00    0.00     204/612         ilaenv_ [109]
-----------------------------------------------
                0.84 7134.43   16524/16524       zsytrf_ [6]
[7]       74.1    0.84 7134.43   16524         zlasyf_ [7]
             6380.83    0.01  672384/672384      zgemm_ [8]
              696.39    0.82 44472273/45524709     zgemv_ [11]
               37.69    0.00 4065366/4065366     zcopy_ [22]
                5.38   10.52 2835775/2854189     izamax_ [31]
                2.77    0.00 1037492/2097636     zscal_ [36]
                0.02    0.00  192269/322261      zswap_ [75]
                0.00    0.00   16524/277234284     lsame_ [50]
-----------------------------------------------
             6380.83    0.01  672384/672384      zlasyf_ [7]
[8]       66.2 6380.83    0.01  672384         zgemm_ [8]
                0.01    0.00 4034304/277234284     lsame_ [50]
-----------------------------------------------
               68.15 2054.11     204/204         microstrip(int, char* const*) [2]
[9]       22.0   68.15 2054.11     204         dynamicpart1_ [9]
             1011.19  305.91 128520612/128520612     zdd_din_rt_ [10]
              436.83  165.78 159306048/159306048     zrd_din_rt_ [12]
               67.78   12.16 49481424/49481424     zrr_din_micro_ [17]
               49.11    0.00 340282404/341950455     intpoints_cell2cell_ [18]
                2.93    1.20 1831104/1831104     zwd_dynamic_rt_fine_ [37]
                0.94    0.26 1129344/1129344     zwr_dynamic_fine_ [46]
                0.01    0.00    6528/6528        zrw_dynamic_fine_ [81]
                0.00    0.00    7344/7344        zww_dynamic_fine_ [97]
-----------------------------------------------
             1011.19  305.91 128520612/128520612     dynamicpart1_ [9]
[10]      13.7 1011.19  305.91 128520612         zdd_din_rt_ [10]
              197.33    0.00 1031377640/1584114134     quad3drec_ [13]
               82.64    0.00 1579143600/2328327468     green_functions3d_ [14]
               23.97    0.00 256812336/419998260     quad3dbck2_ [20]
                0.28    1.43 2746656/2746656     quad3dtetra2_ [43]
                0.08    0.18  200328/540600      quad3dr_1p_generic_ [53]
-----------------------------------------------
```

Podemos observar en la `function line` 4, que para resolver el sistema de ecuaciones, la rutina `solve_system` realiza dos llamadas para cada una de las

frecuencias a la rutina `zsysv`. La primera llamada con el argumento `LWORK=-1` nos devuelve el tamaño óptimo de la memoria que tenemos que reservar. En la segunda llamada se realizan los cálculos correspondientes a la resolución del sistema. A su vez `zsysv` realiza llamadas a `zsytrs`, la cual es una rutina de BLAS2 (operaciones matriz-vector) que resuelve un sistema de ecuaciones lineales. El uso de `zsytr2`, rutina equivalente de BLAS3 (operaciones matriz-matriz), en lugar de `zsytrs` nos daría una mejora en los tiempos de ejecución. En capítulos posteriores se realizará el reemplazo de la función `zsysv` por la correspondiente de MKL, relegando a la misma el control y selección de estas rutinas llamadas desde `zsysv`.

Del análisis completo de la información de *profiling* se puede extraer el gráfico que se muestra en la figura 2.1. Como es de esperar, la estructura de llamadas a funciones se mantiene entre simulaciones. Lo que cambia son los porcentajes de tiempo en cada rama según el tamaño del mallado. Podemos observar que las rutinas `dynamicpartX` suponen hasta un 96.5 % del tiempo total de ejecución, repartiéndose de la siguiente forma: un 22 % en dynamicpart1 y un 75 % en dynamicpart2, para el caso del mallado bajo estudio más complejo.

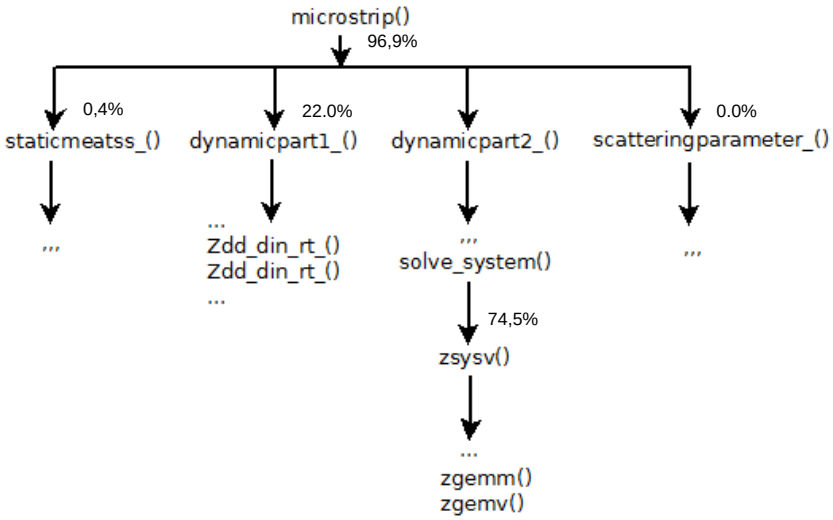

Figura 2.1: Árbol de llamadas y porcentajes de tiempo para mallado complejo en un nodo de Arabí con 204 frecuencias

Para más información acerca del significado de los campos que aparecen en el

`flat` y el `graph profile` se puede consultar el apéndice A.

2.1.2. Intel MKL

La librería Intel Math Kernel Library (Intel MKL) [Int] es una librería matemática altamente optimizada y *multithread* para aplicaciones que requieren el máximo rendimiento. El núcleo de las funciones matemáticas incluye BLAS [BLA], LAPACK [LAP], ScaLAPACK [Sca] y transformada rápida de Fourier, entre otras muchas. A partir de la versión 10.0, Intel MKL emplea un modelo de capas, en el cual se pueden distinguir cuatro partes esenciales de la librería:

- Interface Layer
- Threading Layer
- Computational Layer
- Compiler Support Run-time Libraries

Para *linkar* correctamente nuestro software es necesario elegir al menos una librería de cada una de las 3 primeras capas y, si es necesario, añadir librerías *run-time*. En algunos casos especiales (ScaLAPACK) es necesario *linkar* con más de una librería de cada capa. La capa de *Interface* hace referencia a la codificación con 32 o con 64 bits del tipo de datos entero. En la capa de *Threading* se selecciona el modo paralelo o secuencial de la librería, y en la capa computacional se eligen las rutinas matemáticas que se quieren usar.

En nuestro caso trabajaremos con la rutina `zsysv` de LAPACK, que resuelve un sistema de ecuaciones de coeficientes complejos simétrico múltiple. Esta rutina llama a otras rutinas de LAPACK y BLAS (entrada [4] del `graph profile`), entre ellas `zgemm`, que quedarán ocultas al sustituir el código de las rutinas de LAPACK y BLAS en el programa de partida por la correspondiente llamada a `zsysv`. Además, al utilizar una versión *multithread* de LAPACK se explotará directamente el paralelismo de memoria compartida usando `zsysv`.

La tabla de la figura 2.2 (extraída de [Int]) muestra las distintas librerías dinámicas que se pueden elegir en cada una de las capas para el caso de la arquitectura IA-64. El caso de uso de librerías estáticas es análogo a éste.

File	Contents
Dynamic Libraries	
Interface layer	
libmkl_gf_ilp64.so	ILP64 interface library for the GNU Fortran compiler
libmkl_gf_lp64.so	LP64 interface library for the GNU Fortran compiler
libmkl_intel_ilp64.so	ILP64 interface library for the Intel compilers
libmkl_intel_lp64.so	LP64 interface library for the Intel compilers
libmkl_intel_sp2dp.so	SP2DP interface library for the Intel compilers
Threading layer	
libmkl_gnu_thread.so	Threading library for the GNU Fortran and C compilers
libmkl_intel_thread.so	Threading library for the Intel compilers
libmkl_sequential.so	Sequential library
Computational layer	
libmkl_core.so	Library dispatcher for dynamic load of processor-specific kernel library
libmkl_i2p.so	Kernel library for the IA-64 architecture
libmkl_lapack.so	LAPACK and DSS/PARDISO routines and drivers
libmkl_scalapack_ilp64.so	ScaLAPACK routine library supporting the ILP64 interface
libmkl_scalapack_lp64.so	ScaLAPACK routine library supporting the LP64 interface
libmkl_vml_i2p.so	VML kernel for the IA-64 architecture
RTL	
libguide.so	Intel® Legacy OpenMP* run-time library for dynamic linking
libiomp5.so	Intel® Compatibility OpenMP* run-time library for dynamic linking
libmkl_blacs_intelmpi_ilp64.so	ILP64 version of BLACS routines supporting Intel MPI 2.0/3.x and MPICH2
libmkl_blacs_intelmpi_lp64.so	LP64 version of BLACS routines supporting Intel MPI 2.0 and 3.x and MPICH2
locale/en_US/mkl_msg.cat	Catalog of Intel MKL messages in English
locale/ja_JP/mkl_msg.cat	Catalog of Intel MKL messages in Japanese

Figura 2.2: Librerías de las capas de MKL

Para el desarrollo de éste proyecto se ha utilizado tanto la versión secuencial como la paralela de la librería. Cabe destacar algunas particularidades de la versión paralela de la misma:

- Es *thread-safe*, por lo que todas las funciones MKL trabajan correctamente durante las ejecuciones simultáneas en múltiples *threads*.

- Utiliza OpenMP, y para establecer el número de hilos que trabajarán se puede utilizar la variable de entorno `OMP_NUM_THREADS` o la llamada a la función OpenMP equivalente en tiempo de ejecución: `omp_set_num_threads()`.

- En las últimas versiones de MKL podemos ejercer más control al ofrecer otras variables independientes de OpenMP como `MKL_NUM_THREADS`, que indica el número de *threads* que trabajarán en las llamadas a rutinas de MKL. Se puede combinar la utilización de varios *threads* OpenMP y MKL si se quiere explotar el paralelismo a varios niveles. En caso de conflicto siempre Intel MKL tiene preferencia sobre OpenMP y a su vez las funciones tienen preferencia sobre las variables de entorno. Si no se encuentra ninguna directiva, por defecto se elige el número de *threads* igual al número de *cores* físicos con el fin de obtener el máximo rendimiento, lo que (como se verá más adelante) no es siempre lo más adecuado.

- La variable de entorno `MKL_DYNAMIC` permite a la libreria MKL cambiar dinámicamente el número de *threads*. El valor por defecto es TRUE. Cuando `MKL_DYNAMIC` está fijada a TRUE, Intel MKL intenta usar el número de *threads* que considera óptimo hasta el número máximo especificado. Por ejemplo, si `MKL_DYNAMIC` está fijado a TRUE y el número de *threads* excede el número de *cores* físicos (quizás por el uso de *hyper-threading*), Intel MKL rebajará el número de *threads* hasta el número de *cores* físicos. Cuando `MKL_DYNAMIC` es FALSE, Intel MKL intenta no variar el número de *threads* que el usuario ha solicitado. Sin embargo, el fijar el valor a FALSE no nos asegura que Intel MKL use el número de *threads* que hemos especificado. La librería puede examinar el problema y usar un número diferente de *threads*. Por ejemplo, si intentamos hacer una multiplicación de enteros en 8 *threads*, la librería elegirá usar solamente un *thread* porque considera que no es práctico utilizar 8.

- Si Intel MKL es llamada en una región paralela, solamente utilizará un *thread* por defecto. Si queremos utilizar paralelismo anidado, como será el caso de la última parte del proyecto, debemos fijar `MKL_DYNAMIC` a falso e ir variando manualmente el número de *threads*.

Se ha realizado un *linkado* estático de la librería. La aplicación del modelo de capas anteriormente expuesto se traduce en añadir al *makefile* alguna de las líneas que se muestran en los códigos del 1 al 4, según la arquitectura y la versión secuencial o multithread de MKL a usar.

Código: 1 Linkado con MKL secuencial en el superdome (Ben)

```
$(LD) $(COMP) $(OBJ) -o $(EXEC) -L
/opt/intel/Compiler/11.1/059/lib/ia64/
-lifcore -lifport -ldl -lmkl_intel_lp64
-lmkl_sequential -lmkl_core -lpthread
```

Código: 2 Linkado con MKL paralelo en el superdome (Ben)

```
$(LD) $(COMP) $(OBJ) -o $(EXEC) -L
/opt/intel/Compiler/11.1/059/lib/ia64/
-lifcore -lifport -ldl -lmkl_intel_lp64
-lmkl_intel_thread -lmkl_core -liomp5 -lpthread
```

Código: 3 Linkado con MKL secuencial en el cluster (Arabí)

```
$(LD) $(COMP) $(OBJ) -o $(EXEC) -L
/opt/intel/Compiler/11.1/072/lib/intel64/
-lifcore -lifport -ldl -lsvml -lmkl_intel_lp64
-lmkl_sequential -lmkl_core -lpthread
```

Podemos comprobar si el *linkado* es correcto con el comando `ldd` seguido del ejecutable. El comando `ldd` muestra las librerías compartidas de las que depende un programa y nos avisa en caso de no encontrarlas. El resultado de su ejecución se muestra en el código 5.

2.1.3. OpenMP

OpenMP es una API (*Application Programming Interface*) que permite añadir concurrencia a las aplicaciones mediante paralelismo con memoria compartida

Código: 4 Linkado con MKL paralelo en el cluster (Arabí)

```
$(LD) $(COMP) $(OBJ) -o $(EXEC) -L
/opt/intel/Compiler/11.1/072/lib/intel64/ -lifcore
-lifport -ldl -lsvml -lmkl_intel_lp64 -lmkl_intel_thread
-lmkl_core -liomp5 -lpthread
```

Código: 5 Resultado del comando `ldd`

```
ldd Meatss
        linux-gate.so.1 =>  (0xa000000000000000)
        libifcore.so.6 => /opt/intel/Compiler/11.1/059//lib/ia64/libifcore.so.6 (0x200000000004c000)
        libifport.so.6 => /opt/intel/Compiler/11.1/059//lib/ia64/libifport.so.6 (0x20000000002e4000)
        libdl.so.2 => /lib/libdl.so.2 (0x2000000000358000)
        libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x2000000000370000)
        libm.so.6.1 => /lib/libm.so.6.1 (0x2000000000530000)
        libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x2000000000600000)
        libunwind.so.7 => /lib/libunwind.so.7 (0x2000000000620000)
        libc.so.6.1 => /lib/libc.so.6.1 (0x200000000066c000)
        libimf.so.6 => /opt/intel/Compiler/11.1/059//lib/ia64/libimf.so.6 (0x20000000008c0000)
        libintlc.so.6 => /opt/intel/Compiler/11.1/059//lib/ia64/libintlc.so.6 (0x2000000000b88000)
        /lib/ld-linux-ia64.so.2 (0x2000000000000000)
```

utilizando C, C++ y FORTRAN en la mayoría de arquitecturas de procesadores y de sistemas operativos. OpenMP consiste en una serie de directivas de compilación, librería y variables de entorno que influyen en el compartamiento de la ejecución del programa. OpenMP se basa en el modelo *fork-join* (figura 2.3), paradigma que procede de los sistemas Unix, donde una tarea muy pesada se divide en una serie de hilos (función `fork`) con menor peso, para luego recolectar sus resultados al final y unirlos en un solo resultado (función `join`).

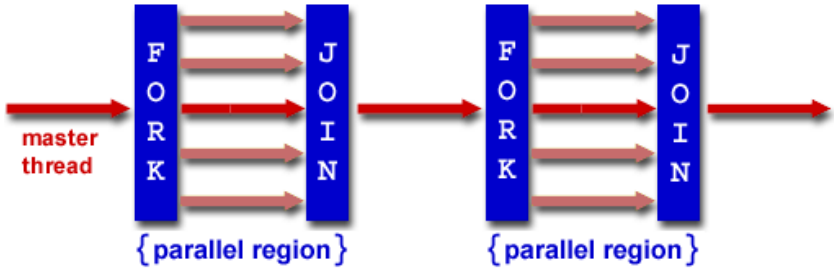

Figura 2.3: Modelo *fork-join* de ejecución paralela

Para la compilación de código con directivas OpenMP, como usaremos los compiladores de Intel, tenemos que añadir el *flag* `-openmp` y fijar el número

de *threads* exportando y asignándole valor a la variable `OMP_NUM_THREADS`. Lo anteriormente dicho se traduce en añadir la siguiente línea al *script* de envío de trabajo al supercomputador:

```
# export OMP_NUM_THREADS = X
```

Tal como se ha comentado, también es posible establecer el número de *threads* dentro del programa con la función `omp_set_num_threads()`, y el valor establecido en la función prevalece sobre el de la variable de entorno.

2.1.4. Debugger de Intel idb

Dada la gran complejidad del software bajo estudio, ha sido necesaria la utilización de una herramienta de *debugging* o depuración, la cual nos permita una ejecución controlada del código con el fin de localizar potenciales errores. Los *debugger* permiten al programador encontrar errores en tiempo de ejecución. Estos errores incluyen códigos mal escritos, goteos de memoria, desbordamiento de pila, excepciones inesperadas y otros problemas algorítmicos. Típicamente, los depuradores también ofrecen funciones más sofisticadas tales como ejecutar un programa paso a paso, parar el programa añadiendo un *breakpoint* y examinar el estado actual de los valores de las variables.

La información de *debugging* es puesta en los ficheros `.o` por el compilador. Con los compiladores de Intel es necesario añadir la opción `-g` a la compilación. Por ejemplo, utilizando `icpc`:

```
# icpc -g ejemplo.cpp
```

Para iniciar el *debugger* en modo línea de comandos basta con tener cargados los compiladores de Intel y escribir `idbc` seguido del programa ejecutable. A continuación hay que ejecutar el comando `run` con los parámetros para ejecutar el software en modo *debugging*. En el caso particular del software MEATSS la secuencia de comandos quedaría como sigue:

```
# idbc ../OBJS/Meatss
# run --in_meatss=../Configure.in --in_gid=../meshes/alex1.msh
--program_mode=scattering --in_green=../greencav.in
```

A partir de este momento el programa empezará su ejecución, la cual solamente será interrumpida en caso de fallo o en caso de que hayamos introducido algún *checkpoint*. Si estamos en el primer caso la instrucción `backtrace` nos dará información detallada de la ubicación del problema, el árbol de llamadas e incluso de la línea del código fuente en la que se ha detectado el fallo. La cantidad de comandos de que dispone `idbc` es muy amplia y puede consultarse introduciendo el *flag* `--help`.

2.2. Herramientas hardware

En este proyecto se trabaja en la paralelización y optimización del software MEATSS en sistemas de memoria compartida. Se ha utilizado principalmente el supercomputador Ben Arabí del Centro de Supercomputación de la Fundación Parque Científico de Murcia, y ocasionalmente el servidor Saturno del grupo de Computación Científica y Programación Paralela de la Universidad de Murcia. Se detallan a continuación las características principales de estos sistemas.

2.2.1. Ben Arabí

El sistema Ben Arabí está formado por dos arquitecturas claramente diferenciadas:

- Un nodo central HP Integrity Superdome SX2000, llamado Ben, con 128 núcleos del procesador Intel Itanium-2 dual-core Montvale (1.6 Ghz, 18 MB de caché L3) y 1.5 TB de memoria compartida.
- Un cluster llamado Arabí, formado por 102 nodos de cálculo, que ofrecen un total de 816 núcleos del procesador Intel Xeon Quad-Core E5450 (3 GHz y 6 MB de caché L2) y 1072 GB de memoria distribuida.

A lo largo del proyecto utilizaremos indistintamente Ben o Superdome para referirnos a la máquina de memoria compartida y Arabí o Cluster para referirnos al *cluster* de memoria distribuída. La figura 2.4 ilustra la arquitectura del supercomputador. En las tablas 2.1 y 2.2 se resumen las especificaciones de los dos sistemas que lo componen.

La arquitectura del Superdome es de 64 bits reales con una capacidad de cálculo por núcleo de 6,4 GFlops. Por tanto el rendimiento máximo del sistema es de

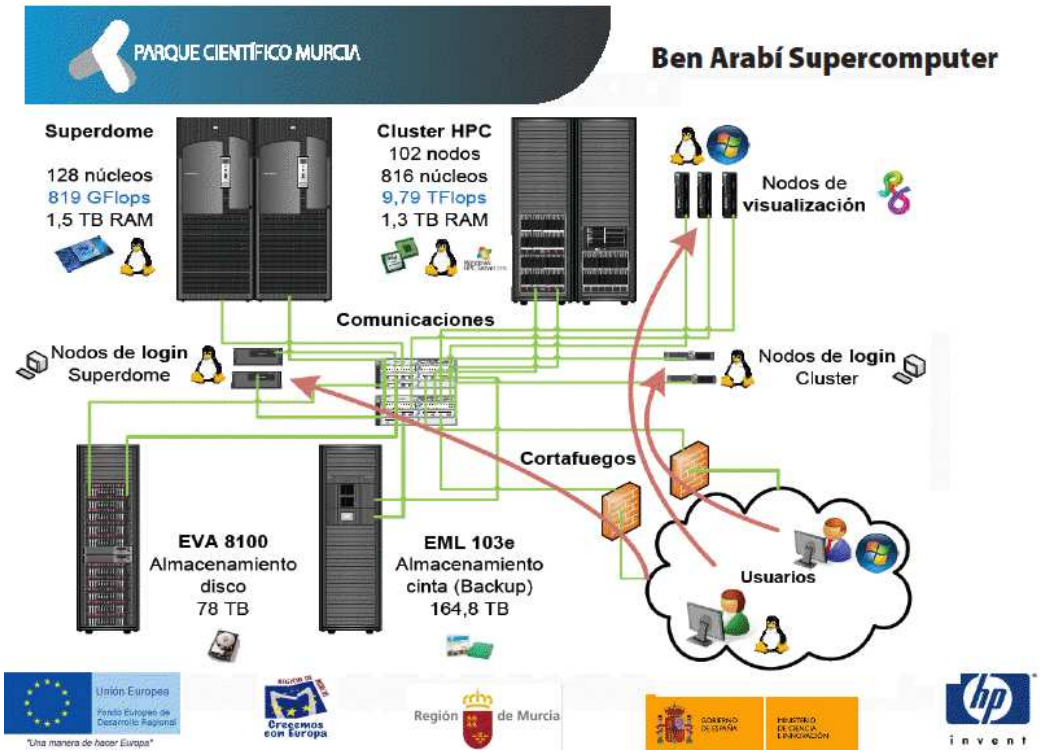

Figura 2.4: Arquitectura de Ben Arabí

Tabla 2.1: Características del Superdome

Capacidad	819 Gflops
Procesador	Intel Itanium-2 Dual-Core Montvale
Nº de núcleos	128
Memoria Compartida	1.5 TB DDR-2
Memoria Caché	18 MB L3
Frecuencia de Reloj	1.6 GHz
Discos de Trabajo Temporal	40 $\times$ 146 GB SAS = 5.84 TB

Tabla 2.2: Características del Cluster

Capacidad	9.72 Tflops
Procesador	Intel Xeon Quad-Core E5450
Nº de núcleos	102
Memoria/Nodo	32 nodos de 16 GB y 70 de 8 GB
Memoria Caché	3 MB (6 MB compartidos entre 2 núcleos)
Frecuencia de Reloj	3 GHz

819 GFlops, obteniendo un rendimiento en el benchmark Linpack de 737 GFlops.

En la arquitectura de memoria compartida todos los procesadores del sistema pueden acceder directamente a todas las posiciones de memoria. El espacio de memoria física es único y global. La utilización de este espacio de memoria común evita la duplicación de datos y el lento trasvase de información entre los procesos. El modelo de programación de memoria compartida es una extensión natural del modelo convencional (secuencial). La comunicación entre tareas/procesos es sencilla, al realizarse por medio de variables compartidas.

La memoria principal es compartida por todos los nodos siguiendo una arquitectura ccNUMA [NUM]. El sistema está organizado en células o nodos, las cuales están formadas por un procesador y una memoria. Los procesadores pueden acceder a la memoria de las demás células a través de la red de interconexión, teniendo en cuenta que el acceso a la memoria local (en la misma célula) es más rápido que el acceso remoto (en una célula diferente).

Por otra parte, el *cluster* tiene un rendimiento pico máximo de 9.792 TFlops. En este sistema, el espacio de memoria es distribuido, cada procesador tiene su espacio de direcciones exclusivo y es preciso el reparto de datos de entrada a cada procesador. El procesador que envía los datos tiene que gestionar la comunicación aunque no tenga implicación directa en el cálculo del procesador destino. La necesidad de sincronización dificulta la programación porque la paralelización se realiza de forma manual. Los procesos se van a comunicar entre sí enviando y recibiendo mensajes y si el tamaño de los datos a procesar es muy grande hay que hacer un particionado que aloje una parte de ellos en la memoria local de cada unidad de procesamiento. En este proyecto, al trabajar con memoria compartida,

se utilizarán nodos de Arabí de forma individual.

Los nodos del *cluster* se encuentran interconectados mediante una red infiniband de altas prestaciones xDDR 20 Gbs *full-duplex*. Se trata de una red de fibra óptica que interconecta todos los nodos de Ben Arabí. La figura 2.5 muestra la topologia de dicha red.

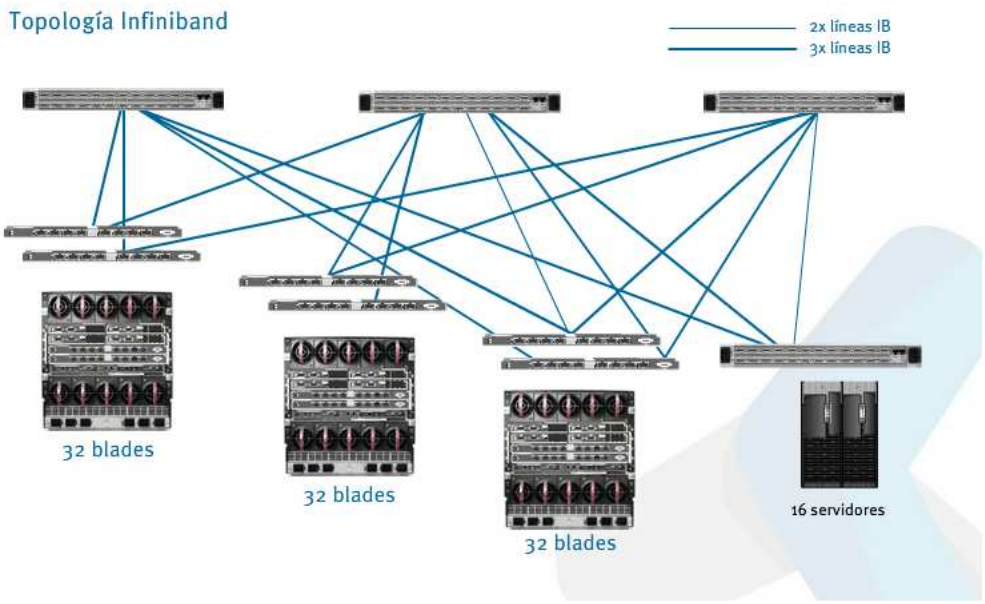

Figura 2.5: Red Infiniband

Los nodos de computación están reservados para la ejecución de problemas de los usuarios. El supercomputador tiene además dos nodos de *login*, los cuales son los puntos a los que el usuario se conecta y desde los cuales se pueden mandar sus trabajos y ver el estado de los mismos. El acceso al supercomputador se realiza por medio de dos nodos de *login* (uno para cada arquitectura) utilizando el protocolo ssh:

- **ben.fpcmur.es** para acceder al Superdome
- **arabi.fpcmur.es** para acceder al Cluster

Desde Linux o Mac podemos acceder simplemente abriendo una consola y tecleando:

ssh usuario@nodo_de_login

Desde los nodos de *login* podemos compilar código fuente, enviar o consultar los trabajos en cola y crear y editar ficheros.

El sistema utilizado para establecer el entorno de ejecución para las aplicaciones, librerías, compiladores, etc disponibles en Ben Arabí, esto es, cargar las variables de entorno necesarias para su ejecución, está basado en el paquete Modules [mod], el cual permite la modificación del entorno del usuario de forma dinámica. Los comandos que nos permiten modificar nuestro entorno son:

- **module avail**: muestra los entornos de trabajo disponibles.
- **module load nombre_app**: carga el entorno de trabajo para la utilización de la aplicación, librería o compilador `nombre_app`.
- **module list**: muestra los entornos de trabajo que tiene cargados.
- **module unload nombre_app**: descarga el entorno de trabajo para la utilización de la aplicación librería o compilador `nombre_app`.
- **module show nombre_app**: Muestra las variables de entorno que carga la aplicación.

Como la mayoría de los trabajos requieren más recursos que los disponibles para los procesos interactivos, es necesario un sistema para la gestión y planificación de trabajos. En Ben Arabí se utiliza LSF (*Load Sharing Facility*) [LSF]. Los trabajos se envían a una cola a través de un *script* utilizando el comando `bsub`. Las opciones más comunes del comando `bsub` se resumen en la tabla 2.3.

Seguidamente se muestra el contenido de un script tipo para el envío de trabajos en Ben Arabí utilizando el gestor de colas LSF:

Tabla 2.3: Opciones del comando `bsub`

-J nombre_trabajo	Asigna un nombre al trabajo
-u email	Indica la dirección de correo
-B	Envía un correo al empezar un trabajo
-N	Envía un correo al finalizar el trabajo
-e fichero_error	Redirige stderr al fichero especificado
-o fichero_salida	Redirige stdout al fichero especificado
-q nombre_cola	Especifica la cola que se va a usar
-n num_core	Especifica el número de cores

```
#!/bin/bash

#BSUB -J nombre_jobs # Nombre del trabajo
#BSUB -o salida.%J.out # Fichero de salida
#BSUB -e error.%J.err # Fichero de error
#BSUB -q arabi_8x24h # Nombre de la cola
#BSUB -n 8 # Nº de cores reservados

source /etc/profile.d/modules.sh
module load intel # Carga de módulos

export MKL_NUM_THREADS=8  # Número de threads

time ./meatss_scattering
```

El *script* anterior manda un trabajo con nombre `nombre_jobs` al gestor de colas LFS. La salida se vuelca en un fichero con nombre `salida.JOBID.out` y los errores se almacenan en un fichero llamado `error.JOBID.err`. Para la ejecución se realiza una reserva de 8 *cores* (`-n 8`) y se envía a la cola `arabi_8x24h`, por lo que puede estar en ejecución hasta 24 horas. Pasado ese tiempo, el propio gestor de colas se encargará de finalizar el trabajo. Las siguientes líneas se encargan de cargar los módulos requeridos por el software, en este caso los compiladores de Intel, fijar el número de *threads* a 8 y finalmente de lanzar la ejecución del

programa, en este caso `meatss_scattering`.

El sistema de colas también nos permite monitorizar el estado de los trabajos con el comando `bjobs` y eliminarlos con `bkill`.

2.2.2. Saturno

Saturno es un servidor con cuatro procesadores *hexacores*, con un total de 24 *cores*. Su topología se muestra en la figura 2.6, obtenida con el software `hwloc` [Por]. Se observa que cada uno de los *cores* tiene asociados dos elementos de proceso, al tener *hyperthreading*. A pesar de que la memoria es compartida, se encuentra distribuida entres los cuatro procesadores, y hay tres niveles de cache, lo que hace que se pueda ver este sistema como intermedio entre Ben, que es cc-NUMA y con una jerarquía amplia de memoria, y un nodo de Arabí, que consta de ocho *cores* con varios niveles de cache. Por tanto, este sistema se usará ocasionalmente para comprobar la validez y generalidad de algunas de las técnicas estudiadas en Ben Arabí en otros sistemas de memoria compartida, para lo que también se realizará algún experimento en un portátil *quadcore*.

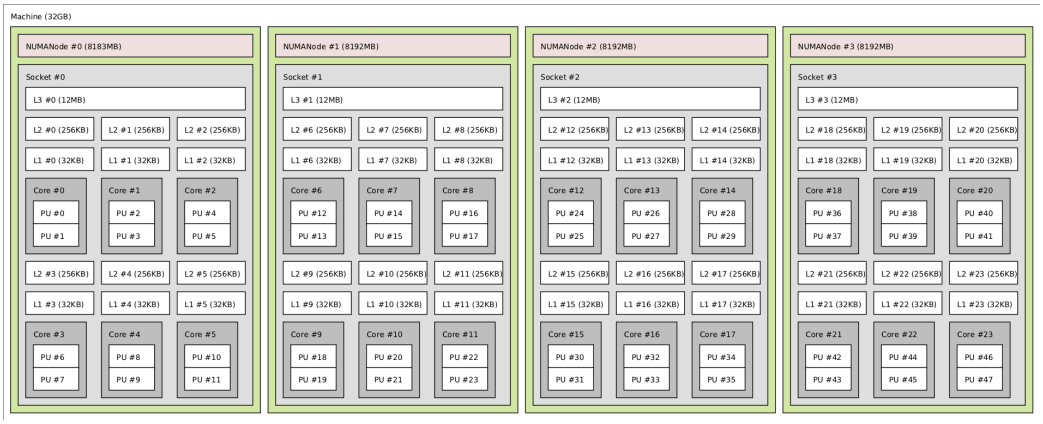

Figura 2.6: Topología de Saturno

Capítulo 3

Utilización de la librería MKL

En este capítulo empezaremos realizando un breve análisis de la estructura del programa, aprovechando la información expuesta en el capítulo anterior. A continuación se analizará la paralelización con MKL y la posibilidad de utilización de dos rutinas distintas para resolver los sistemas de ecuaciones. Se realizará un estudio experimental utilizando mallas de distintas complejidades y utilizando las dos arquitecturas presentes en el Centro de Supercomputación de la Fundación Parque Científico de Murcia, y adicionalmente se compararán las prestaciones de diferentes rutinas de resolución de sistemas de ecuaciones en la máquina Saturno.

3.1. Estructura del programa

El esquema general del programa se muestra en el algoritmo 1.

Algorithm 1 Bucle en frecuencia de MEATSS

staticpart
for $i = 0 \rightarrow num\_freq$ **do**
    $freq = init\_freq + i * step$
    dynamicpart1($freq$)
    dynamicpart2
    scatteringparameters
end for

En la implementación del Método de los Momentos elegida para resolver el sistema de ecuaciones integrales que se obtiene de la aplicación del método de la

ecuación integral de volumen/superficie (EIVS) [RWG82, SWG84] se ha decidido emplear el conocido método de Galerkin, por el cual se utiliza para el proceso de test el mismo conjunto de funciones base ya seleccionado. La ventaja de esta técnica es que conduce a matrices de momentos con forma simétrica, facilitándose con ello el proceso de construcción de la matriz. Además, los algoritmos específicos para la resolución de sistemas de ecuaciones simétricos permiten dividir aproximadamente por la mitad el tiempo de inversión con respecto al que se requiere para un sistema genérico [Pé09]. Aunque en primera instancia la elección de una rutina para resolución de sistemas de matrices simétricas parece lo más coherente, veremos que, bajo determinadas condiciones, puede ser ventajoso utilizar una rutina genérica que no contemple esta simetría.

La elección de la función de Green de espacio libre para la formulación EIVS utilizada, se sustenta en la intención de poder flexibilizar al máximo el tipo de geometrías que pueden ser analizadas, al no imponer esta función ninguna restricción sobre el tipo del medio bajo estudio. Se ha decidido utilizar una descomposición eficiente de la misma, de forma que se gane en coste computacional y precisión de cálculo. Para ello, se ha recurrido a la separación en una parte estática y una parte dinámica. Esto da lugar a una división de la matriz de momentos completa en dos partes:

$$Z_{m,n} = Z_{m,n}^{din} + Z_{m,n}^{est} \tag{3.1}$$

La rutina `staticpart` calcula la parte estática de la matriz de momentos mediante la resolución de integrales analíticas. Esta rutina solamente se ejecuta una vez y el resultado es utilizado en cada una de las iteraciones en frecuencia.

La rutina `dynamicpart1` realiza el llenado de la matriz de momentos calculando las integrales numéricas entre las funciones de base y de test junto con las funciones de Green de la parte dinámica:

$$\int \left(\int \left(Ft(r) \cdot \hat{\hat{G}} \left(r, r' \right) \cdot Fb \left(r' \right) \cdot dr' \right) \cdot dr \right) \tag{3.2}$$

donde Fb es la función de base, Ft es la función de test y G es la función de Green.

La parte estática del algoritmo pone en G solamente la parte estática de la función de Green, que no depende de la frecuencia y que se calcula sólo una vez para cada estructura. La parte estática tiene una expresión más sencilla que la parte

dinámica, lo que permite un cálculo analítico para las integrales correspondientes. En la rutina `staticpart` están implementadas las expresiones analíticas para el cálculo de estas integrales.

Con la rutina `dynamicpart1` se introduce para cada frecuencia la función de Green completa pero extrayendo la parte estática que ya se calculó antes. Para estas integrales no hay solución analítica y por eso lo que hay implementado en esa rutina son integrales numéricas extendidas a tetraedros o hexaedros para las integrales de volumen y triángulos o rectángulos para integrales de superficie, dependiendo de las celdas (aunque el software implementa dominios en tetraedros y en triángulos, las estructuras investigadas en este proyecto no utilizan estos elementos). Al no llevar la parte estática no hay singularidades en esas integrales, por lo que se pueden calcular de forma numérica sin demasiadas complicaciones debido al reducido número de puntos de integración. Como estas integrales dependen de la frecuencia hay que calcularlas para cada punto de frecuencia.

La rutina `dynamicpart2` realiza la resolución del sistema de ecuaciones lineales utilizando tanto la parte estática como la dinámica (ecuación 3.1), y se calculan los parámetros eléctricos del circuito con la función `scatteringparameters`.

Si partimos de una malla de tamaño $m \times n$ y las matrices de la ecuación 3.1 son matrices simétricas de tamaño $N \times N$, el coste de las funciones `staticpart` y `dynamicpart1` es cuadrático de orden mn o N^2, pues se completan las N^2 posiciones de las matrices $Z^{est}_{m,n}$ y $Z^{din}_{m,n}$ a partir de los mn puntos en la malla. Los sistemas que se resuelven en `dynamicpart2` tienen dos vectores de términos independientes, y el coste de la resolución de cada sistema es de orden N^3. Si llamamos f al número de frecuencias, el coste total del algoritmo es:

$$t(N, f) = k_{sta}N^2 + f\left(k_{dyn1}N^2 + k_{dyn2}N^3\right) \tag{3.3}$$

donde k_{sta}, k_{dyn1} y k_{dyn2} representan los costes de una operación aritmética básica en cada una de las funciones `staticpart`, `dynamicpart1` y `dynamicpart2`. No se ha considerado la función `scatteringparameters` por tener un coste mucho menor. Independientemente de los valores relativos de los parámetros en la ecuación 3.3, cuando el número de las frecuencias crece, la parte correspondiente al bucle en el algoritmo 1 es la más costosa, y será la que habrá que paralelizar y optimizar. Del mismo modo, al crecer el tamaño del ma-

llado y de las matrices, el coste de `dynamicpart2` es el más importante, por lo que empezaremos analizando la optimización de la resolución de los sistemas de ecuaciones con paralelismo implícito usando la implementación de LAPACK *multithread* de Intel MKL.

3.2. Uso de MKL

Como se ha expuesto en el capítulo anterior, podemos observar que la carga computacional se concentra principalmente en las llamadas a la rutina del programa `solve_system` (que corresponde a `dynamicpart2` en el algoritmo 1), la cual resuelve un sistema lineal de ecuaciones llamando a la función de LAPACK `zsysv`. Esta función a su vez hace múltiples llamadas a `zgemm` y `zgemv`. El código de partida lleva las rutinas de LAPACK y BLAS incrustadas.

En esta sección del proyecto se ha procedido al reemplazo de dicha rutina por la función equivalente de la librería Intel MKL [Int] intalada en el supercomputador. Para ello se ha eliminado la rutina `zsysv` y se ha realizado el *linkado* correspondiente, tal como se explica en la sección 2.1.2, tanto para la versión *multithread* como para la secuencial. La gestión de la librería MKL se realiza, al igual que el resto de las aplicaciones instaladas en el supercomputador, con el paquete `Module`. Mediante este paquete se establece el entorno de ejecución adecuado para poder utilizar las librerías. Para cargar el entorno de MKL basta con incluir el comando `module load intel` en el script de envío del trabajo:

```
# module load intel
```

Con lo que se cargan los compiladores de Intel para C++ y FORTRAN, además de la librería MKL y el *debugger* idb.

Las simulaciones se han realizado fijando la variable de entorno `MKL_DYNAMIC` tanto con valor TRUE (valor por defecto) como a FALSE, resultando los tiempos obtenidos similares, por lo que los resultados de los experimentos que se muestren corresponderán siempre que no se diga lo contrario al caso `MKL_DYNAMIC=FALSE`, ya que será de esta forma como habrá que ejecutarlo para obtener paralelismo anidado.

3.3. Estudio experimental

Para la realización del estudio experimental se han utilizado tres mallados de complejidad creciente y se ha efectuado un barrido discreto en frecuencia. Los mallados simple, medio y complejo nos producen unas matrices cuadradas $N \times N$ de tamaños $N = 733$, $N = 3439$ y $N = 5159$ respectivamente.

Como cabría esperar, hay un aumento lineal del tiempo conforme aumentamos el número de frecuencias a calcular. Esto es debido a que cada frecuencia conforma un cálculo totalmente independiente. Por esta razón, a partir de ahora sólo se tomarán tiempos para un número fijo de frecuencias, suponiendo, sin pérdida de generalidad, que el aumento de tiempo es proporcional al número de frecuencias calculadas (ecuación 3.3).

Concretamente se ha estudiado la modalidad del software MEATSS para el cálculo de los parámetros de scattering, `meatss_scattering`. El estudio realizado y las técnicas empleadas son fácilmente extrapolables a otros módulos del programa.

Para la correcta medida de tiempos en las ejecuciones *multithreads* es importante medir el `wallclock time`. El código original de MEATSS realizaba medidas de `CPU time`, por lo que se ha modificado la toma de tiempos. Se ha optado por utilizar la librería `wtime` [WTI]. El `wallclock time` es la suma del `CPU time`, la entrada-salida y el retardo del canal de comunicaciones.

Los resultados mostrados en las siguientes tablas se corresponden con los tiempos para una sola frecuencia de la versión inicial del software sin optimizar, la versión con la librería MKL secuencial y por último para la versión paralela de la librería MKL variando el número de *threads*, hasta 8 en el Cluster y hasta 32 en el Superdome. Los datos han sido obtenidos para ambas arquitecturas y son mostrados en segundos. Los tiempos obtenidos deben de tomarse siempre como valores orientativos, ya que están sujetos a la carga computacional que hay en cada momento en el sistema. Aunque los recursos de ejecución se asignan en exclusiva, el almacenamiento es compartido entre todos los usuarios y, dependiendo del uso de entrada-salida del programa, el tiempo de ejecución puede variar de manera sensible.

La tablas 3.1 y 3.2 muestran los tiempos de la parte estática y las dos funcio-

nes dinámicas obtenidos en el Cluster y en el Superdome respectivamente. Las subrutinas `staticpart` y `dynamicpart1` no han sido modificadas en esta sección del proyecto, por lo que permanecen fijas y no se volverá a tomar tiempos de las mismas. Recordemos que estas subrutinas realizan un llenado de matrices. En las tablas se muestra también el coste relativo de cada una de las rutinas. Podemos observar que la importancia relativa de la parte estática, aunque de mayor peso, decrece conforme aumenta la complejidad del mallado. En ambas arquitecturas el comportamiento es análogo, siendo mayor el tiempo empleado en la rutina estática. Como cabría esperar, el tiempo de ejecución se incrementa conforme aumentamos la complejidad de la geometría.

Tabla 3.1: Tiempos (en segundos) de `staticpart`, `dynamicpart1` y `dynamicpart2` y coste relativo de cada parte, en Arabí para mallados de distinta complejidad

	sta-par		dyn-par1		dyn-par2	
	tiempo	%	tiempo	%	tiempo	%
simple	7.6	89.6	0.65	7.7	0.23	2.7
media	106.49	76.6	12.5	8.9	21.77	15.5
compleja	70.31	40.4	31.1	17.9	72.72	41.8

Tabla 3.2: Tiempos (en segundos) de `staticpart`, `dynamicpart1` y `dynamicpart2` y coste relativo de cada parte, en Ben para mallados de distinta complejidad

	sta-par		dyn-par1		dyn-par2	
	tiempo	%	tiempo	%	tiempo	%
simple	10.82	77.8	2.37	17.1	0.71	5.1
media	145.53	56.6	44.78	17.4	67.13	26.1
compleja	149.54	31.9	95.32	20.3	223.6	47.3

La ganancia obtenida por el uso de MKL se puede comprobar en las tablas 3.3 (Arabí) y 3.4 (Ben), donde se muestran los tiempos de ejecución en `dynamicpart2` cuando se inserta el códido de las rutinas de BLAS y LAPACK y cuando se llama a la rutina `zsysv` de MKL con un único *thread*. Cabe recordar

que la subrutina `dynamicpart2` realiza la resolución de un sistema de ecuaciones lineales complejo. En primer lugar llama la atención la mejora obtenida por el hecho de utilizar la subrutina secuencial correspondiente de MKL, pasando de 72.72 a 19.51 segundos en el caso del Cluster y de 223.6 a 34.17 en el Superdome. Esta mejora es justificable debido a que la rutina `zsysv` incrustada en el código inicial hacía uso de BLAS2, mientras que la de MKL utiliza la correspondiente rutina de BLAS3, diseñada para realizar operaciones matriz-matriz. Se muestra también la ganancia por el uso de MKL. Los procesadores de Ben son más lentos que los de Arabí, razón por la cual todos los tiempos de ejecución son mayores en el Superdome. En ambos casos se observa una ganancia importante por el uso de MKL, que depende del sistema donde se trabaje, estando en Arabí el aumento de velocidad alrededor de 3 y en Ben de 6.

Tabla 3.3: Tiempos (en segundos) de `dynamicpart2` con código insertado y con llamada a `zsysv` de MKL, y speed-up conseguido con el uso de MKL, en Arabí para mallados de distinta complejidad

	insertado	MKL	speed-up
simple	0.23	0.07	3.28
media	21.77	5.90	3.69
compleja	72.72	19.51	3.73

Tabla 3.4: Tiempos (en segundos) de `dynamicpart2` con código insertado y con llamada a `zsysv` de MKL, y speed-up conseguido con el uso de MKL, en Ben para mallados de distinta complejidad

	insertado	MKL	speed-up
simple	0.71	0.12	5.92
media	67.13	10.43	6.44
compleja	223.60	34.17	6.54

En la tablas 3.5 y 3.6 podemos observar la evolución de los tiempos obtenidos en la ejecución de `dynamicpart2` al variar el número de *cores* y utilizando la rutina `zsysv` de MKL en su versión *multithread*, con lo que se está utilizando paralelismo implícito. Se puede observar que el nivel de reducción de tiempos

al aumentar el número de *cores* es mayor conforme aumentamos la complejidad del mallado. Este aumento en la complejidad de la geometría se traduce en matrices más grandes, cuya computación es más fácilmente paralelizable. Para mallados simples la mejora de tiempos conforme se aumenta el número de hilos no es significativa, llegando incluso a empeorar. En las tablas se ha resaltado, para las diferentes complejidades del mallado, el menor tiempo de ejecución obtenido al variar el número de *cores*. El número de *cores* con que se obtiene el menor tiempo no es siempre el máximo del sistema o el máximo número de *cores* con el que se está experimentando, por lo que sería conveniente tener alguna metodología de selección automática (autooptimización) para seleccionar el número de hilos a utilizar en la resolución de un problema en función de su tamaño y del sistema donde se esté resolviendo.

Tabla 3.5: Tiempos de ejecución (en segundos) de `dynamicpart2` con MKL, en Arabí variando el número de *cores*

Mallado	Sec	2 c	4 c	8 c
Simple	0.07	0.07	**0.05**	0.09
Media	5.90	3.61	2.47	**2.00**
Compleja	19.51	11.96	7.02	**5.64**

Tabla 3.6: Tiempos de ejecución (en segundos) de `dynamicpart2` con MKL, en Ben variando el número de *cores*

Mallado	Sec	2 c	4 c	8 c	16 c	24 c	32 c
Simple	0.12	0.10	0.08	**0.07**	**0.07**	0.08	0.08
Media	10.43	6.68	4.39	3.23	2.70	**2.66**	2.69
Compleja	34.17	20.75	12.55	8.76	7.18	6.89	**6.58**

Aunque se obtiene una reducción en el tiempo de ejecución con el uso del paralelismo, esta reducción no es muy importante, lo que se puede observar en las figuras 3.1 (Arabí) y 3.2 (Ben), donde se muestra el speed-up de la rutina `zsysv` para los distintos mallados y variando el número de *cores*. La escalabilidad de la rutina `zsysv` no es todo lo buena que cabría esperar, alcanzándose ganancias de velocidad máximas de alrededor de 3 en Arabí y de 5 en Ben, que tiene un total

de 128 *cores*, con lo que la explotación del paralelismo obtenida con `zsysv` es muy reducida.

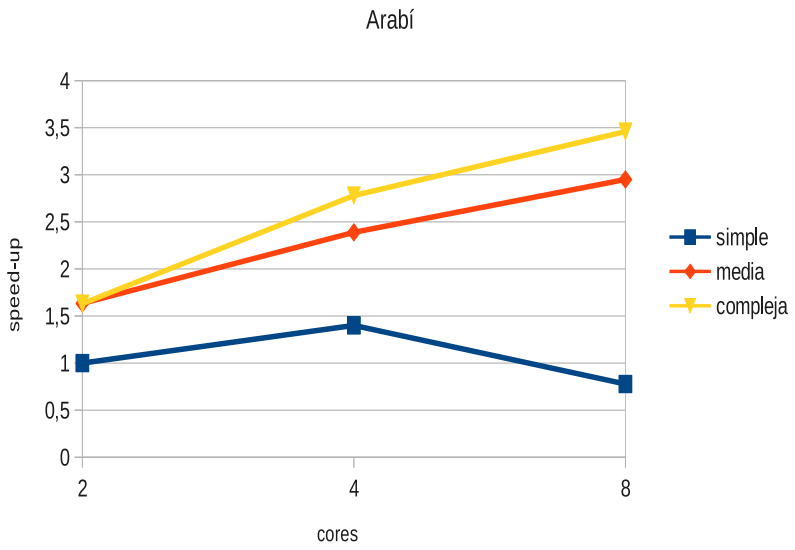

Figura 3.1: *Speed-up* de la rutina `zsysv` de MKL en Arabí

La rutina `zsysv` parece ser la mejor opción dadas las características de nuestras matrices (complejas simétricas), pero hemos comprobado que el paralelismo se explota muy escasamente en esta rutina, al menos en la implementación *multithread* de MKL, por lo que probaremos con otra rutina de resolución de sistemas lineales más genérica (`zgesv`) para ver si el comportamiento se repite. En las tablas 3.7 y 3.8 se pueden observar los tiempos obtenidos en la función `dynamicpart2` cuando se usa la rutina `zgesv` *multithread*. Las rutinas `zsysv` y `zgesv` tienen un coste de orden cúbico, pero en secuencial `zsysv` es el doble de rápida que `zgesv` al tener la mitad de operaciones aritméticas [GL90], lo que hace que los tiempos de `zgesv` sean mayores que los de `zsysv` para un número pequeño de *cores*. Sin embargo, conforme aumentamos el número de los mismos, la tendencia es la de reducir con `zgesv` el tiempo obtenido respecto a la subrutina `zsysv`. Las figuras 3.3 y 3.4 muestran los *speed-up* obtenidos con la rutina `zgesv` en Arabí y Ben. Comparando con los *speed-ups* de `zsysv` de las figuras 3.1 y 3.2 se observa un mayor aprovechamiento del paralelismo de la rutina que no considera la matriz simétrica, llegándose a alcanzar para problemas de gran

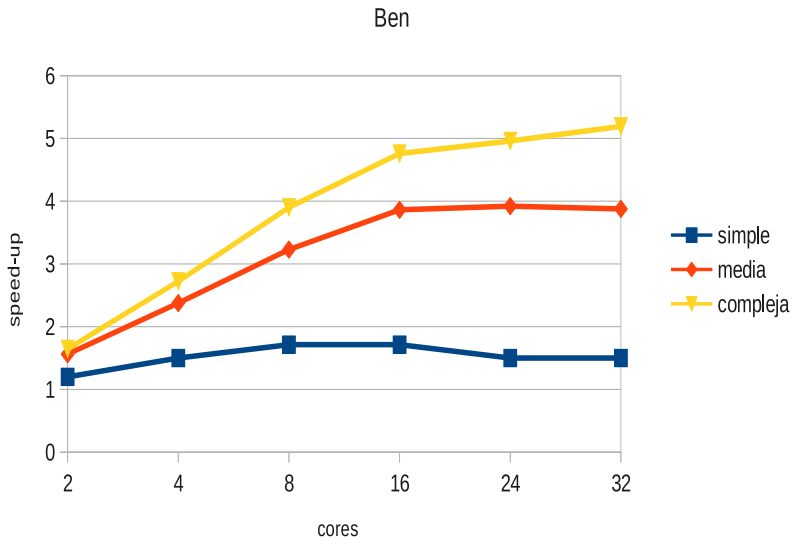

Figura 3.2: *Speed-up* de la rutina `zsysv` de MKL en Ben

dimensión un *speed-up* de alrededor de 16 en el Superdome, lo que significa una ganancia de velocidad respecto al algoritmo secuencial de `zsysv` de alrededor de 8, claramente mayor que el aumento de un 5 que se obtenía con `zsysv`.

Tabla 3.7: Tiempos de ejecución (en segundos) de `dynamicpart2` con `zgesv`, en Arabí variando el número de *cores*

Mallado	2 c	4 c	8 c
Simple	0.07	0.05	**0.03**
Media	5.31	2.75	**1.59**
Compleja	17.21	8.77	**4.90**

La diferencia en el comportamiento de las rutinas `zsysv` y `zgesv` es debida a la diferencia en las rutinas de descomposición que utilizan para factorizar la matriz de coeficientes antes de la resolución de los múltiples sistemas de ecuaciones. En la rutina `zsysv` se utiliza la factorización LDL^T, y en la `zgesv` la LU. En el primer caso se explota la simetría de la matriz, con lo que el número

Tabla 3.8: Tiempos de ejecución (en segundos) de `dynamicpart2` con `zgesv`, en Ben variando el número de *cores*

Mallado	2 c	4 c	8 c	16 c	24 c	32 c
Simple	0.16	0.16	0.19	0.16	**0.14**	0.16
Media	9.43	4.97	2.96	1.85	1.41	**1.26**
Compleja	31.08	16.00	8.78	5.97	4.85	**4.55**

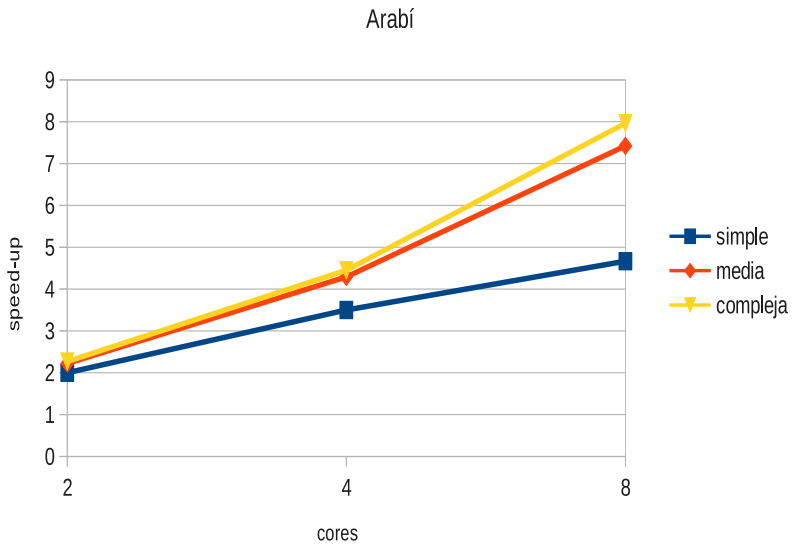

Figura 3.3: *Speed-up* de la rutina `zgesv` de MKL en Arabí

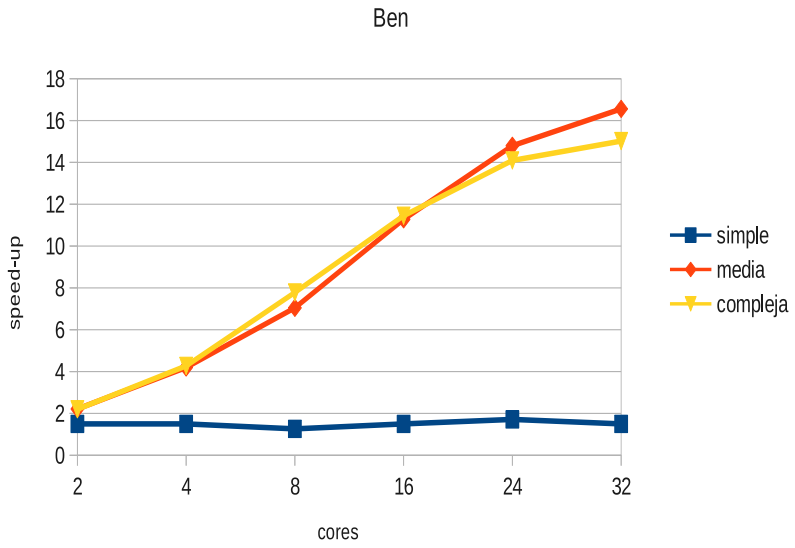

Figura 3.4: *Speed-up* de la rutina `zgesv` de MKL en Ben

de operaciones aritméticas es la mitad del de la segunda factorización. Sin embargo, la explotación de la simetría dificulta la paralelización, lo que hace a la rutina `zgesv` preferible cuando el número de *cores* crece. El número de *cores* a partir del cual es preferible usar `zgesv` se muestra en la tabla 3.9, para Arabí y Ben y con los tres mallados previamente considerados, y en la tabla 3.10 para Saturno y en un portátil *quadcore*, con distintos tamaños de matriz y comparando también el comportamiento de las correspondientes rutinas para datos de doble precisión. En general, observamos que las rutinas `gesv` son más escalables que las `sysv` y prácticamente en todos los casos llegan a ser preferibles (la única excepción es el mallado simple en Ben), pero al aumentar el coste de la computación (mayor tamaño del problema y tipo complejo en vez de doble) también aumenta el número de *cores* necesario para que las rutinas para matrices generales mejoren los tiempos de las que explotan la simetría. Esto se ve claramente en la figura 3.5, donde se comparan en Saturno los tiempos de ejecución de las rutinas `dsysv` y `dgesv` (rutinas para reales de doble precisión), y `zsysv` y `zgesv`, para tamaños de problema 1024 y 2048.

Tabla 3.9: Número de *cores* a partir del cual la rutina `zgesv` obtiene menor tiempo de ejecución que la `zsysv`, en Arabí y Ben y para mallados de distinta complejidad

Mallado	Arabí	Ben
Simple	4	-
Media	8	8
Compleja	8	16

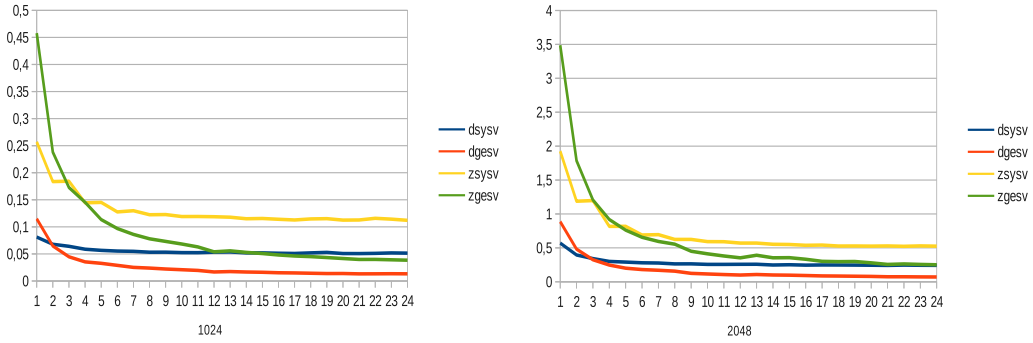

Figura 3.5: Comparación del tiempo de ejecución de rutinas de resolución de sistemas de ecuaciones lineales, variando el número de *cores*, para matrices de tamaño 1024 (izquierda) y 2048 (derecha), en Saturno

Tabla 3.10: Número de *cores* a partir del cual la rutina `gesv` obtiene menor tiempo de ejecución que la `sysv`, en Saturno y en un portátil *quadcore*, para distintos tamaños de matriz y para datos de tipo doble y complejo-doble

	Saturno		quadcore	
size	`dsysv-dgesv`	`zsysv-zgesv`	`dsysv-dgesv`	`zsysv-zgesv`
384	2	3	1	2
512	2	3	2	2
640	2	3	2	2
896	3	3	2	3
1024	2	3	1	2
1152	3	3	2	3
1408	3	3	2	3
1536	3	3	2	3
1664	3	3	2	3
1920	3	5	2	3
2048	3	5	2	3
2176	4	5	2	3
2432	4	5	2	3
2560	4	5	2	3
2688	4	5	2	3
2944	5	7	2	3
3072	3	7	3	3
3200	5	7	2	3
3456	5	7	3	3
3584	5	9	2	4
3712	5	9	3	3
3968	5	9	3	3
4096	5	9	3	3

3.4. Resumen y conclusiones

En este capítulo se ha presentado una descripción del código bajo estudio y se ha hecho uso de la libreria MKL en la parte de resolución de sistemas de ecuaciones, que se ha detectado tras el análisis de *profiling* como la de mayor coste para problemas de gran tamaño. Se han realizado simulaciones variando el número de frecuencias y el mallado, utilizando tanto la versión secuencial como la paralela de MKL.

Tras un primer análisis sustituyendo la función `zsysv` por su correspondiente llamada a la librería MKL, se han obtenido unos valores de *speed-up* máximos de 5.2 y de 3.45 para el caso del Superdome y del Cluster, respectivamente; valores muy alejados del *speed-up* máximo teórico. La mejora en los tiempos de ejecución se hace patente especialmente con los mallados complejos, los cuales dan lugar a matrices de tamaños mayores.

Con el objetivo de mejorar los resultados, se ha escogido otra rutina para la resolución de sistemas de ecuaciones lineales más general, `zgesv`, a priori peor al no tener en cuenta la simetría de las matrices bajo estudio. El resultado ha sido una mejora en los tiempos respecto a la subrutina `zsysv` a partir de un determinado número de *cores*, extrayendo como conclusión que la subrutina `zgesv` escala mejor.

Con las tres mejoras introducidas (utilización de las rutinas optimizadas de MKL, uso de paralelismo intrínseco con rutinas *multithread*, y uso de la rutina para matrices no simétricas) se obtiene una reducción importante en el tiempo de ejecución en `dynamicpart2`. La tabla 3.11 muestra el cociente entre el tiempo de ejecución del código original y el mejor tiempo obtenido utilizando MKL. Se muestra la mejora en la parte dinámica (funciones `dynamicpart2`) y en la ejecución del programa completo, en este caso con 10 y 100 frecuencias. Se obtiene una mejora importante (para la malla de mayor complejidad se pasa de unas 100 horas de ejecución a aproximadamente 30 horas en Ben con 100 frecuencias), pero muy lejos de la máxima alcanzable, que viene determinada por el número de *cores* que se utilizan. En los siguientes capítulos se explotará el paralelismo OpenMP y OpenMP+MKL para intentar reducir más el tiempo de ejecución.

Tabla 3.11: Aceleración en el tiempo de ejecución respecto al programa original, usando paralelismo MKL, en Arabí y Ben, para mallados de distinta complejidad en la parte dinámica y en el programa completo con 10 y 100 frecuencias

	Arabí		
Mallado	dinámica	total 10 frec.	total 100 frec.
Simple	1.29	1.14	1.15
Media	2.43	1.82	1.89
Compleja	2.88	2.58	2.71
	Ben		
Mallado	dinámica	total 10 frec.	total 100 frec.
Simple	1.26	1.18	1.20
Media	2.43	2.09	2.19
Compleja	3.19	2.91	3.09

Capítulo 4

Paralelismo con OpenMP

El capítulo que nos ocupa trata sobre la utilización de la librería de memoria compartida OpenMP en el código bajo estudio. Como la ejecución se va a realizar en paralelo, para poder hacer uso de dicha librería se ha realizado una profunda modificación del código, con el fin de evitar escrituras simultáneas de variables. En primer lugar se mostrará la metodología a seguir para convertir el código de MEATSS en uno apto para ser paralelizado. Una vez hechos los cambios, se expondrá la nueva estructura de MEATSS. A continuación se realizará un estudio experimental variando el número de *threads* y el mallado para cada una de las arquitecturas del Supercomputador Ben Arabí y se mostrarán las conclusiones.

4.1. Reestructuración del programa

MEATSS está programado de tal manera que todas las subrutinas hacen uso de variables globales. Una variable global es aquella que se define fuera del cuerpo de cualquier función, normalmente al principio del programa o en un módulo. El ámbito de una variable global son todas las funciones que componen el programa y cualquier función puede acceder a dichas variables para leer y escribir en ellas. Es decir, se puede hacer referencia a su dirección de memoria en cualquier parte del programa. A pesar de que aparentemente nos parezca muy útil, el uso de variables globales no es aconsejable por varias razones:

- Empeora la legibilidad.
- El uso indiscriminado de variables globales produce efectos colaterales. Esto sucede cuando existe una alteración no deseada del contenido de una va-

riable global dentro de una función. La corrección de dichos errores puede ser muy ardua.

- Va en contra de uno de los principios de la programación, la modularidad, la cual supone no compartir espacios de memoria con otras funciones, y potenciar el paso de información (llamadas) para que la función trate la información localmente.

Las variables declaradas dentro de una función sólo existen mientras se ejecuta la función. Cuando se invoca se crean estas variables en la pila y se destruyen cuando la función termina. Si queremos que nuestro programa sea paralelo, el uso de variables globales agrava aún más los problemas antes mencionados, produciéndose escrituras indeseadas desde distintos hilos de ejecución. Por esta razón, antes de realizar la paralelización con OpenMP, se ha realizado una completa reescritura del código eliminando las variables globales que implican escritura y pasándolas como parámetros a las rutinas correspondientes. El proceso seguido ha consistido en lo siguiente:

- Identificación de todas las subrutinas implicadas en la ejecución. Para ello ha resultado de gran utilidad la herramienta de *profiling* `gprof` explicada en los primeros capítulos del proyecto.

- Para cada subrutina examinar todas las variables globales. Si en algún momento son escritas, se crea una copia local que será usada en lugar de la variable global. En caso de que la variable solamente se lea no se realiza modificación alguna, ya que no hay problemas de coherencia al no escribir ningún hilo en ella.

- Si una subrutina lee alguna variable global que ha sido modificada por otra subrutina antecesora, deben modificarse todas las subrutinas desde esta antecesora hasta la que lee la variable. Se realizará una copia local de la variable en la subrutina antecesora y se pasará como parámetro en todas las llamadas hasta llegar a la subrutina que originariamente leía la variable.

A continuación se muestra el árbol de funciones que han sido modificadas:

- dynamicpart1
    - zdd_din_rt
        - green_functions3d

    - quad3drec
    - quad3dr_1p_generic
        - green_functions3d
        - quad3dt
    - quad3dbck2
    - quad3dtetra2
        - det
- zrd_din_rt
    - green_functions3d
    - quad3drec
    - quad3dr_1p_generic
        - green_functions3d
        - quad3dt
    - quad3dbck2
    - quad3dr2
- zrr_din_micro
    - green_functions3d
    - quad3dr2
    - quad3dt
- zwd_dynamic_rt_fine
    - green_functions3d
    - quad3drec
    - quad3dbck2
    - gauleg3d
- zwr_dynamic_fine
    - green_functions3d
    - quad3dr2
    - gauleg3d
- zww_dynamic_fine
    - green_functions3d
    - gauleg3d

- zrw_dynamic_fine
  - green_functions3d
  - quad3dr2
  - gauleg3d
- intpoints_cell2cell

- dynamicpart2
  - excitation_evaluation
    - eimicro
      - gauleg3d

- scatteringparameter
  - circuital_parameters
    - gaussj
      - nrutil_mp_outerprod_z
      - nrutil_mp_swap_zv
      - nrutil_mp_outerand
      - nrutil_mp_assert_eq3

Vemos que ha sido necesario modificar 28 funciones, con un máximo de cuatro niveles en el árbol, lo que da idea del trabajo realizado. Además, en todas las modificaciones ha habido que validar los cambios con la ejecución de varios tests de prueba.

Con el fin de independizar secciones de código, y dado que el número de argumentos de algunas funciones varía debido a los cambios introducidos, también se ha realizado una duplicación de funciones. Un ejemplo de dicha duplicación se produce con la función `intpoints_cell2cell`, la cual mantiene el nombre cuando es llamada desde `static_part` y se renombra a `intpoints_cell2cell2` cuando se llama, con más argumentos, desde la parte dinámica. De esta manera nos evitamos modificar la parte estática minimizando la aparición de posibles errores. Además de los cambios indicados anteriormente se ha optimizado el uso de la memoria cargando solamente las variables necesarias de los módulos en lugar de cargarlos en su totalidad. El esquema general del

Algorithm 2 Reestructuración del bucle en frecuencia de MEATSS en 3 bucles

```
for i = 0 → num_freq do
    fillmatrix(i,init_freq,step)
end for

for i = 0 → num_freq do
    solvesystem(i)
end for

for i = 0 → num_freq do
    circuitalparameters(i,init_freq,step)
end for
```

programa reestructurado se muestra en el algoritmo 2.

Podemos apreciar que se ha dividido el bucle inicial en 3 bucles, uno que realiza el llenado de la matriz de momentos, otro que resuelve el sistema de ecuaciones y un último que calcula los parámetros circuitales. De los 3 bucles, hemos visto que la carga computacional se concentra en los dos primeros. Con el fin de evaluar la conveniencia de utilizar OpenMP en el bucle de llenado de la matriz, se ha realizado la toma de tiempos para el caso más favorable (geometría compleja) obteniéndose una mejora muy baja (217.287 segundos con 1 *thread* frente a 199.979 segundos con 2 *threads)*, por lo cual se ha descartado la paralelización del mismo, centrándonos únicamente en paralelizar el bucle que resuelve el sistema de ecuaciones. Esta ganancia tan reducida con el uso del paralelislo puede deberse a que el coste de computación y de acceso a memoria es del mismo orden ($O\left(N^2\right)$), y el acceso a las distintas zonas de memoria no sigue un patrón regular, lo que propicia una ralentización en el acceso a los datos por su continuo trasiego entre los distintos niveles de la jerarquía de memoria.

En este algoritmo las funciones de cada uno de los bucles han sido adaptadas de tal forma que en una sola estructura de datos se almacenan las matrices de todas las frecuencias involucradas. Esto es necesario ya que el bucle donde se resuelven los sistemas tiene que tener las matrices de todas las frecuencias para poder realizar su resolución en paralelo. El uso de esta aproximación hace que los requerimientos de memoria sean mucho mayores que en el código inicial donde

sólo se reservaba espacio para los datos de una frecuencia: si en el caso secuencial la ocupación de memoria era N^2 ahora es fN^2, lo que hace que para muchas frecuencias el coste de memoria pueda llegar a ser muy alto.

El caso de que la memoria supusiera una limitación se podría plantear un reparto de computación por hilos en lugar de por frecuencias, con un esquema como el que se muestra en el algoritmo 3. En este caso habría que modificar las funciones en el esquema para que trabajaran con t (t representa el número de hilos que trabajan en la resolución del problema y en general será menor que el número total de frecuencias, f) matrices de tamaño $N \times N$, de manera que para cada valor del índice j se accediera a la matriz $j \equiv \text{mód } t$.

Algorithm 3 Reestructuración del bucle en frecuencia de MEATSS en 3 bucles para cada grupo de t frecuencias consecutivas

```
for i = 0 → num_freq/t do
    for j = i * t → (i + 1) * t − 1 do
        fillmatrix(j,init_freq,step)
    end for

    for j = i * t → (i + 1) * t − 1 do
        solvesystem(j)
    end for

    for j = i * t → (i + 1) * t − 1 do
        circuitalparameters(j,init_freq,step)
    end for
end for
```

Una vez realizados los cambios anteriores estamos en disposición de abordar la paralelización con OpenMP del segundo bucle. Para ello hay que cargar el módulo $< omp.h >$ en el `main.cpp` y añadir los flags `-openmp` en el makefile, tanto en la sección de FORTRAN como en la de C++. El bucle 2 del algoritmo 2 una vez paralelizado quedaría como se muestra en el algoritmo 4.

Algorithm 4 Bucle de resolución de sistemas con OpenMP

```
# pragma omp parallel for private(i)
for i = 0 → num_freq do
    solvesystem(i,init_freq,step)
end for
```

4.2. Estudio experimental

Debido al elevado coste de las simulaciones se plantea seguir una estrategia para reducir el número de las mismas. Dado que nos interesa obtener el número de *threads* con el que se consigue el menor tiempo de ejecución, la metodología a seguir consiste en realizar una toma de tiempos para un número de *threads* intermedio y a partir de ahí repetir los experimentos localmente para un número mayor y menor de *threads*. En las sucesivas simulaciones se utilizará el valor intermedio del intervalo en el que se muestre una mejora de tiempo mayor. Las simulaciones de mallados de más complejidad comenzarán utilizando el número de *threads* óptimo del mallado anterior, ya que, generalmente, al aumentar la complejidad también aumenta el número de *threads* óptimo.

En este capítulo todas las ejecuciones se han realizado fijando las variables de entorno a los valores `OMP_NESTED=FALSE`, `MKL_DINAMIC=FALSE`, `MKL_NUM_THREADS=1`, y se ha variado el número de hilos OpenMP con la variable `OMP_NUM_THREADS`. Todas estas variables han sido fijadas en el fichero de envío de trabajo que se muestra en el código 6 para el Cluster y en el código 7 para el Superdome.

La tabla 4.1 muestra los tiempos de ejecución de la subrutina `solvesystem` para 8 frecuencias obtenidos en el sistema Arabí variando el número de *threads* siguiendo la estrategia comentada para reducir el número de ejecuciones, por lo que aparecen huecos en la tabla. Podemos observar que el caso más favorable se produce siempre para el mayor número de *threads*.

La tabla 4.2 muestra los mismos experimentos en Ben y con 128 frecuencias. La arquitectura de Ben, al disponer de hasta 128 *cores*, nos da un posible rango de ejecuciones mayor. En esta ocasión, para el mallado simple, obtenemos un tiempo óptimo cuando utilizamos 48 *cores*. Al aumentar la complejidad de la geometría este valor se incrementa hasta los 64 *cores* para el mallado de complejidad me-

Código: 6 Fichero `bsub` para el Cluster

```
#!/bin/bash

#BSUB -o depuracion.%J.out
#BSUB -e depuracion.%J.err
#BSUB -q arabi16_120h
#BSUB -n 8

source /etc/profile.d/modules.sh
module load intel

export MKL_DYNAMIC=FALSE
export OMP_NESTED=TRUE
export MKL_NUM_THREADS=1
export OMP_NUM_THREADS=1

time ./meatss_scattering
```

Tabla 4.1: Tiempos de ejecución (en segundos) del bucle de resolución de sistemas con llamadas a `zsysv` de MKL, en Arabí para 8 frecuencias con OpenMP

Mallado	2 c	4 c	8 c
Simple	1.43	0.99	**0.88**
Media	-	-	**10.78**
Compleja	-	-	**33.47**

Código: 7 Fichero `bsub` para el Superdome

```
#!/bin/bash

#BSUB -o depuracion.%J.out
#BSUB -e depuracion.%J.err
#BSUB -q ben_128x24h
#BSUB -n 64

source /etc/profile.d/modules.sh
module load intel

export MKL_DYNAMIC=FALSE
export OMP_NESTED=TRUE
export MKL_NUM_THREADS=1
export OMP_NUM_THREADS=64

time ./meatss_scattering
```

dia, manteniéndose para el mallado más complejo. Aunque idealmente el número óptimo de *threads* es 128, la sobrecarga introducida debido a la generación y gestión de hilos y por la gestión de los accesos a memoria hace que en la práctica este valor óptimo sea menor que el número de *cores* en el sistema, con lo que sería conveniente tener alguna estrategia para poder decidir el número de *threads* a utilizar en la resolución del problema.

Tabla 4.2: Tiempos de ejecución (en segundos) del bucle de resolución de sistemas con llamadas a `zsysv` de MKL, en Ben para 128 frecuencias con OpenMP

Mallado	16 c	32 c	48 c	56 c	64 c	96 c	128 c
Simple	3.76	2.90	**1.93**	-	3.79	-	-
Media	-	-	59.41	61.90	**45.25**	81.6	91.55
Compleja	-	-	-	-	**123.54**	171.25	-

Para comparar la ganancia de velocidad en la resolución de los sistemas de

ecuaciones correspondientes a las diferentes frecuencias con paralelismo MKL y OpenMP se compararía el *speed-up* alcanzado con los dos esquemas, pero dado que en este caso hemos reducido el número de ejecuciones no disponemos de todos los datos necesarios, y utilizaremos lo que podemos llamar "eficiencia normalizada":

$$E_n\left(p_1, p_2\right) = \frac{t_{p_1} p_1}{t_{p_2} p_2} \tag{4.1}$$

donde se pondera el tiempo de ejecución (t_{p_i}) obtenido con distinto número de *threads* (p_i) multiplicándolo por este número. De esta forma, para el caso secuencial tendríamos el tiempo secuencial dividido por el tiempo paralelo multiplicado por el número de *threads*, lo que corresponde a la eficiencia, con valor entre 0 y 1, y los valores más próximos a 1 corresponden a un mejor uso del paralelismo. La tabla 4.3 compara los valores obtenidos para tres pares de número de *threads* en Arabí y Ben para el mallado simple. Dado que se puede utilizar las rutinas `zsysv` o `zgesv` de MKL en la resolución de los sistemas, se muestran los valores para estas dos rutinas. Se observa que el paralelismo se explota mejor con OpenMP que con MKL cuando aumenta el tamaño del sistema computacional (el número de *cores*), aunque con un número reducido de *threads* el paralelismo que se obtiene con `zgesv` es muy satisfactorio.

Tabla 4.3: Eficiencia normalizada para distintos pares de números de threads para mallados simples con las rutinas `zsysv` y `zgesv` de MKL y con OpenMP, en Arabí y Ben

	Arabí		Ben
	2 c / 4 c	4 c / 8 c	16 c / 32 c
`zsysv`	0.70	0.27	0.43
`zgesv`	0.70	0.83	0.50
OpenMP	0.72	0.56	0.64

4.3. Resumen y conclusiones

En este capítulo se ha mostrado el uso de la librería para memoria compartida OpenMP. Dada la estructura inicial del programa, en la que se hacía uso extensivo de variables globales, se ha tenido que hacer una reescritura del código para evitar

accesos simultáneos a zonas de memoria antes de poder abordar la paralelización propiamente dicha. Una vez el código ha sido reestructurado se ha procedido a la realización de un estudio experimental en el que se han lanzado ejecuciones variando el número de hilos para distintas geometrías y para las dos arquitecturas del Centro de Supercomputación de la Fundación Parque Científico de Murcia.

Para el cluster, en todos los casos se ha obtenido el mejor tiempo utilizando el número máximo de *cores* disponible, 8. En cambio, para las ejecuciones en el Superdome, el número de *cores* para el cual el tiempo de ejecución es óptimo es 64, valor que se corresponde con la mitad de su capacidad. Por tanto, concluimos que, para nuestro caso particular, no merece la pena lanzar ejecuciones para un número de *cores* mayor de 64 en el Superdome.

Para hacernos una idea de la mejora de tiempos que supone aplicar OpenMP podemos hacer un simple cálculo: Si por ejemplo, según el capítulo 3, el Superdome emplea 34.17 segundos en resolver el sistema de ecuaciones para la geometría compleja y una sola frecuencia, para resolver 128 tardaría aproximadamente 4373 segundos. Para este mismo caso, utilizando OpenMP tardamos solamente 123 segundos ejecutando en 64 *cores*, lo que supone un *speed-up* de más de 35. Las tablas 4.4 y 4.5 muestran una comparativa de los tiempos obtenidos con llamadas a la librería MKL secuencial, los tiempos con OpenMP para el caso mejor y el *speed-up*. Dado que, como hemos visto, el paralelismo se explota mejor con OpenMP que con MKL, el *speed-up* es mayor que el obtenido en el capítulo anterior con paralelismo MKL, pero los valores alcanzados para el mallado de mayor complejidad están entre alrededor de un medio y un cuarto del máximo alcanzable teniendo en cuenta el número de *cores* en el sistema. Esto hace que consideremos la posibilidad de utilizar paralelismo de dos niveles, combinando paralelismo OpenMP y MKL, lo que se analizará en el siguiente capítulo.

Tabla 4.4: Tiempos en segundos para el cálculo de 8 frecuencias en el caso de utilizar MKL secuencial, OpenMP con el número óptimo de *threads* y *speed-up*, en Arabí para mallados de distinta complejidad

Mallado	MKL sec	OpenMP	speed-up
Simple	0.56	0.88	0.63
Media	47.2	10.78	4.38
Compleja	156.08	33.47	4.66

Tabla 4.5: Tiempos en segundos para el cálculo de 128 frecuencias en el caso de utilizar MKL secuencial, OpenMP con el número óptimo de *threads* y *speed-up*, en Ben para mallados de distinta complejidad

Mallado	MKL sec	OpenMP	speed-up
Simple	15.36	1.93	7.95
Media	1335.04	45.25	29.5
Compleja	4373.76	123.54	35.4

Capítulo 5

Paralelismo híbrido OpenMP+MKL

Dado que los resultados obtenidos en el capítulo anterior distan del valor óptimo teórico, en este capítulo nos planteamos la posibilidad de combinar los dos niveles de paralelismo estudiados (*threads* OpenMP y dentro llamadas a rutinas paralelas de MKL) para intentar mejorar la escalabilidad de MEATSS.

5.1. Paralelismo de dos niveles

El paralelismo multinivel es una técnica que se viene aplicando recientemente de manera satisfactoria en rutinas de álgebra lineal [GL09, CGG12]. Las opciones de paralelismo de 2 niveles son muchas: OpenMP+OpenMP, MPI+OpenMP, etc. En entornos de memoria compartida el uso de varios niveles de paralelismo puede ayudar a aprovechar mejor la jerarquía de memorias, a balancear el trabajo entre los distintos *cores* en el sistema distribuyendo trabajos de distinto coste entre los distintos componentes computacionales, y a explotar el paralelismo de grano grueso y de grano fino en los diferentes niveles.

Mientras que en otros trabajos se ha estudiado paralelismo OpenMP+MKL en rutinas básicas de álgebra lineal (multiplicación de matrices y descomposiciones factoriales) [CCGV12, CGG12], en este capítulo nos proponemos aplicar dichas técnicas aprovechando que tenemos implementados dos tipos de paralelismo distintos en el código. Por un lado un paralelismo implícito en las rutinas de MKL (la resolución de cada sistema en el algoritmo 4) y por otro un paralelismo explícito con OpenMP (el bucle en frecuencias en dicho algoritmo). Con la combinación de ambos tipos de paralelismo surge un paralelismo híbrido OpenMP+MKL.

El esquema de paralelización es el mismo de los algoritmos 2 y 4, pero en este caso indicando la forma en que interactúan los dos tipos de paralelismo y estableciendo el número de *threads* a usar en cada nivel, con lo que obtenemos el esquema que se muestra en el algoritmo 5, que es una modificación del secuencial con tres bucles (algoritmo 2), donde se paraleliza el bucle donde se resuelven los sistemas (algoritmo 4) y se establece que se usará paralelismo anidado, que no hay selección dinámica del número de *threads* MKL, y el número de *threads* OpenMP y MKL. El paralelismo anidado puede que esté establecido por defecto, pero es conveniente asegurarse. El valor por defecto de la variable que indica la selección dinámica del número de *threads* por MKL es normalmente TRUE, lo que produce, en las versiones de MKL con las que se ha experimentado en distintos sistemas, que se pierda el paralelismo MKL dentro de un bucle OpenMP [GL09], por lo que es necesario deshabilitar la selección dinámica de MKL. El número de *threads* establecido para OpenMP ($ntomp$) será el que se use en la creación de *threads* esclavos con el constructor `parallel`, y el establecido para MKL ($ntmkl$) se usará en cada llamada a la rutina de `zsysv` de MKL dentro de la función `solvesystem`.

5.2. Estudio experimental

Para el estudio experimental se realizarán las simulaciones activando la variable de entorno que nos permite el paralelismo anidado `OMP_NESTED` y se irá variando el número de *threads* de MKL y de OpenMP para mallados de distinta complejidad. Como se ha visto en el capítulo 3, el comportamiento de las subrutinas `zsysv` y `zgesv` es distinto desde el punto de vista de la escalabilidad, razón por la cual repetiremos los experimentos para cada una de estas subrutinas. En el caso del Superdome, al tener 128 *cores*, el número de combinaciones de *threads* OpenMP y MKL se dispara, por lo que no se experimentará con todas las combinaciones posibles. De igual forma se omitirán aquellos casos en los que se vea evidente que la tendencia es la de empeorar el tiempo de ejecución.

Las tablas 5.1 y 5.2 muestran los tiempos obtenidos utilizando la rutina `zsysv` en Arabí y en Ben respectivamente. En las columnas de las tablas se muestra el número de *threads* OpenMP y MKL siguiendo el siguiente formato: número de *threads* OpenMP - número de *threads* MKL. En los dos casos se experimenta con un número de frecuencias coincidente con el número de *cores* en el sistema. En

Algorithm 5 Paralelismo híbrido en MEATSS con 3 bucles

```
omp_set_nested(1)
mkl_set_dynamic(0)
omp_set_num_threads(ntomp)
mkl_set_num_threads(ntmkl)

for i = 0 → num_freq do
    fillmatrix(i,init_freq,step)
end for

# pragma omp parallel for private(i)
for i = 0 → num_freq do
    solvesystem(i)
end for

for i = 0 → num_freq do
    circuitalparameters(i,init_freq,step)
end for
```

Arabí se utilizan los 8 *cores* del nodo con el que se trabaja, mientras que en Ben se utilizan 64 *cores*, pues hemos observado en los experimentos de los capítulos precedentes que el uso de todo el sistema no es siempre la mejor opción para reducir el tiempo de ejecución.

Tabla 5.1: Tiempos en segundos para el cálculo de 8 frecuencias utilizando programación híbrida OpenMP+MKL con subrutina `zsysv` en Arabí para mallados de distinta complejidad

Mallado	8-1	4-2	2-4	1-8
Simple	0.99	**0.36**	0.48	0.84
Media	**11.82**	11.91	16.13	24.14
Compleja	**33.57**	35.29	42.20	65.10

Tabla 5.2: Tiempos en segundos para el cálculo de 128 frecuencias utilizando programación híbrida OpenMP+MKL con subrutina `zsysv` en Ben para mallados de distinta complejidad

Mallado	64-1	32-2	16-4
Simple	3.08	**2.22**	2.85
Media	**48.53**	63.66	97.65
Compleja	**114.68**	152.91	241.79

Del análisis de las tablas 5.1 y 5.2 comprobamos que sólo obtenemos mejoras con el uso del paralelismo híbrido en la geometría más simple, para la que en Arabí es mejor utilizar paralelismo anidado con 4 *threads* OpenMP y 2 *threads* MKL, y en Ben con 32 y 2 *threads*. Para geometrías complejas, como la ganancia obtenida con OpenMP es muy alta y el paralelismo de la rutina `zsysv` bastante pobre, el caso óptimo lo obtenemos con la utilización de paralelismo OpenMP (el número de *threads* MKL es uno).

Dado que, como se vio en el capítulo 3, la rutina `zgesv` explota mejor el paralelismo que la `zsysv`, es posible que el paralelismo híbrido se comporte mejor con el uso de la rutina para matrices generales. En las tablas 5.3 y 5.4 se muestran los tiempos obtenidos repitiendo los experimentos con la rutina `zgesv` en lugar de `zsysv`. El comportamiento mostrado es ligeramente distinto al obtenido con

zsysv. Para geometrías simples la mejor opción sigue siendo el uso de paralelismo híbrido (4 y 2 *threads* OpenMP y MKL en Arabí y 32 y 2 *threads* en Ben). Sin embargo, debido al mejor uso del paralelismo de zgesv, en el caso de Ben, donde usamos un mayor número de *cores*, la mejor opción para las geometrías de complejidad media y alta la mejor opción ya no es usar OpenMP solamente, sino que el tiempo mínimo de ejecución lo alcanzaríamos con 32 *threads* OpenMP y 2 *threads* de MKL, con una reducción del tiempo de ejecución respecto al uso de paralelismo OpenMP de aproximadamente el 20 por ciento.

Tabla 5.3: Tiempos en segundos para el cálculo de 8 frecuencias utilizando programación híbrida OpenMP+MKL con subrutina zgesv en Arabí para mallados de distinta complejidad

Mallado	8-1	4-2	2-4	1-8
Simple	1.09	**0.76**	1.41	2.21
Media	**15.50**	20.14	25.40	34.07
Compleja	**46.25**	53.56	66.75	-

Tabla 5.4: Tiempos en segundos para el cálculo de 128 frecuencias utilizando programación híbrida OpenMP+MKL con subrutina zgesv en Ben para mallados de distinta complejidad

Mallado	64-1	32-2	16-4
Simple	4.94	**4.93**	6.12
Media	96.49	**81.36**	89.04
Compleja	222.01	**171.42**	193.46

Comparando las tablas 5.1 y 5.3 (Arabí) y las 5.2 y 5.4 (Ben), comprobamos que cuando se utiliza paralelismo híbrido usando la rutina zgesv no se mejora el tiempo de ejecución obtenido usando la de matrices simétricas, lo que es debido a que el número óptimo de *threads* MKL es reducido, lejos del que es necesario para que zgesv mejore a zsysv.

5.3. Otras posibilidades de explotación del paralelismo híbrido

Se ha realizado un conjunto reducido de experimentos para comprobar el comportamiento del paralelismo híbrido dentro del software con el que trabajamos, pero hay algunos otros casos en que la combinación de los dos niveles de paralelismo puede ser aconsejable. Vemos un par de situaciones de este tipo.

Cuando el número de frecuencias (f) es menor que el número de *cores* del sistema (c) el *speed-up* máximo teórico alcanzable con OpenMP es f, pues no se pueden poner a trabajar más de f *threads* en el bucle de resolución de los sistemas. Si embargo, si en la resolución de cada sistema colaboran c/f *threads* MKL, trabajarán los c cores, con un *speed-up* máximo de c. Es verdad que también se obtiene un *speed-up* máximo teórico de c si se utiliza sólo paralelismo MKL, pero, como hemos visto en los capítulos anteriores, se consigue una mejor explotación del paralelismo con OpenMP, por lo que experimentalmente es preferible utilizar este tipo de paralelismo al mayor nivel posible.

Si el número de frecuencias no es múltiplo o divisor del número de *cores* en el sistema, este no es explotado completamente con los esquemas vistos hasta ahora. Por ejemplo, si consideramos $f = 107$ y $c = 24$, si se utiliza paralelismo OpenMP con 24 *threads*, algunos de ellos resolverían 4 sistemas, y otros 5. El tiempo de ejecución secuencial es 107 veces el tiempo de resolver un sistema, mientras que el paralelo es el tiempo de resolver 5 sistemas, que es el de los *cores* con más trabajo. El desbalanceo en el reparto de trabajo hace que el *speed-up* máximo alcanzable sea 21.4, mientras que el máximo del sistema es 24. Para mantener todos los *cores* del sistema trabajando e igualmente balanceados durante toda la computación se podría usar un esquema como el del algoritmo 6, donde no se utiliza un número fijo de *threads* OpenMP y MKL durante toda la ejecución. Se utiliza un bucle para recorrer los divisores del número de *cores* del mayor al menor. El número de *threads* OpenMP coincide con el valor del divisor, y el de *threads* MKL con el índice del bucle. Se utilizan tres variables para indicar en cada paso la frecuencia inicial y final con las que se trabaja ($inifrec$ y $finfrec$) y el número de frecuencias restantes con las que trabajar ($nfres$). De esta forma, se entra en el bucle c veces, se actualizan los valores de las variables cuando el índice del bucle es divisor de c (el algoritmo `div` representa la división entera), y se trabaja en los pasos en los que hay frecuencias para las que calcular, lo que ocurre cuando

$finfrec \geq inifrec$.

Para facilitar la comprensión del funcionamiento del algoritmo 6, en la tabla 5.5 se muestran los valores que toman las variables una ejecución con $f = 107$ y $c = 24$. Se incluye una fila por cada paso por el bucle en el que el índice del bucle es divisor del número de *cores*. Sólo se realiza computación con los valores de $ntmkl$ 1 (96 frecuencias), 3 (8 frecuencias) y 8 (3 frecuencias).

Tabla 5.5: Valores de las variable en una ejecución del algoritmo 6 con $f = 107$ y $c = 24$

$ntmkl$	$ntomp$	$inifrec$	$finfrec$	$nfres$
		0	0	107
1	24	1	96	11
2	12	97	96	11
3	8	97	104	3
4	6	105	104	3
6	4	105	104	3
8	3	105	107	0
12	2	108	107	0
24	1	108	107	0

Con un esquema como el planteado se consigue tener todos los *cores* trabajando durante toda la ejecución, y el trabajo está balanceado entre ellos. Aún así, no podemos asegurar que este esquema nos proporcione el menor tiempo de ejecución experimental, pues sabemos que suele ser preferible utilizar paralelismo OpenMP a paralelismo MKL, y que sólo en algunos casos es preferible el paralelismo híbrido. Sería necesaria una experimentación exhaustiva en cada sistema, variando el número de *threads* OpenMP y MKL y el número de frecuencias y el tamaño del problema, para determinar la combinación que nos proporciona el menor tiempo de ejecución.

5.4. Resumen y conclusiones

Es posible utilizar en el software MEATSS paralelismo híbrido combinando OpenMP en el bucle en frecuencias con MKL en la resolución de cada uno de los sistemas. Los resultados experimentales obtenidos con paralelismo híbrido son

Algorithm 6 Paralelismo híbrido en MEATSS con 3 bucles y con número variable de *threads* OpenMP y MKL

```
omp_set_nested(1)
mkl_set_dynamic(0)

for i = 0 → num_freq do
    fillmatrix(i,init_freq,step)
end for

inifrec=0
finfrec=0
nfres=f
for ntmkl = 1 → c do
    if c múltiplo de ntmkl then
        ntomp=c div ntmkl
        omp_set_num_threads(ntomp)
        mkl_set_num_threads(ntmkl)
        inifrec=finfrec+1
        finfrec=inifrec+nfres div ntomp *ntomp -1
        # pragma omp parallel for private(i)
        for i = inifrec → finfrec do
            solvesystem(i)
        end for
        nfres=nfres -(finfrec - inifrec +1)
    end if
end for

for i = 0 → num_freq do
    circuitalparameters(i,init_freq,step)
end for
```

normalmente peores que los obtenidos con paralelismo OpenMP. Sólo en el problema de menor complejidad, el uso del paralelismo de dos niveles mejora los tiempos obtenidos con paralelismo OpenMP, con una configuración óptima de 32 *threads* OpenMP y 2 *threads* MKL en Ben y 4 *threads* OpenMP y 2 *threads* MKL en Arabí. Para problemas de tamaño medio o grande, aunque la mejor opción utilizando `zgesv` se obtiene empleando paralelismo híbrido OpenMP+MKL, el tiempo de ejecución sigue siendo mayor que el obtenido con `zgesv` (171 frente a 114 segundos para el mallado complejo en Ben).

Pese a que los resultados con paralelismo híbrido son principalmente negativos, hay otras posibilidades de paralelización, variando el número de *threads* OpenMP y MKL durante la ejecución del programa, con las que se pueden llegar a obtener mejores resultados que con paralelismo OpenMP. Para determinar la mejor combinación es necesario un análisis experimental sistemático, para lo que se pueden utilizar técnicas de autooptimización o *autotuning*.

Capítulo 6

Algunos aspectos de optimización y autooptimización

A lo largo de los capítulos que conforman el proyecto hemos ido obteniendo diferentes versiones optimizadas del software MEATSS. Cada una de estas versiones ha sido obtenida empleando una aproximación distinta del paralelismo y ha resultado ser especialmente eficiente en algún caso en particular, por lo que cabe plantearse la pregunta de qué versión del software elegir en cada caso, dependiendo del número de frecuencias y de la dimensión de los sistemas a resolver y del sistema computacional que se esté utilizando.

Mediante la aplicación de técnicas de autooptimización se pretende determinar la combinación más adecuada de *threads* OpenMP y MKL a establecer en cada nivel de paralelismo, así como la elección de la rutina MKL óptima, con el fin de conseguir una ejecución lo más eficiente posible y con un tiempo de instalación mínimo. En este capítulo veremos algunas ideas generales de autooptimización de software paralelo y analizaremos su posible utilización en el software MEATSS.

6.1. Autooptimización en Software Paralelo

Las técnicas de autooptimización se han venido aplicando en diferentes campos [Bre94, Fri98], especialmente a rutinas de álgebra lineal [CDLR03, WPD01], que son especialmente importantes por utilizarse como rutinas básicas en multitud de problemas científicos, como por ejemplo el que nos ocupa. Con la aplicación de estas técnicas se consiguen rutinas que se adaptan al sistema (muchas veces

sistemas distintos para los que fueron creados las rutinas originalmente) de forma automática. Ya que los principales usuarios potenciales de sistemas paralelos de alto rendimiento son científicos e ingenieros que tienen problemas de alto coste computacional, pero sin conocimientos profundos de paralelismo, otra motivación para el uso de las técnicas de *autotuning* es abstraer a los usuarios de los problemas que plantea el paralelismo.

En este trabajo se ha colaborado con el grupo de Computación Científica y Programación Paralela de la Universidad de Murcia, que trabaja en técnicas de optimización y autooptimización de software paralelo y en su aplicación a distintos problemas científicos. Se ha trabajado en problemas básicos de álgebra lineal [CGG+03, AAV+04, CGG04, Muñ05] y en diversas aplicaciones (metaheurísticas [CLCLG12], modelado climático [JMG09], hidrodinámica [LCG10], programación dinámica [CGMG05], etc.), y más recientemente en la adaptación de estas técnicas a sistemas *multicore* [CCGV12, CGG12]. En este capítulo se analiza la posible aplicación al software MEATSS en sistemas *multicore* y NUMA de las técnicas de autooptimización previamente utilizadas por el grupo con otros problemas.

6.1.1. Ciclo de vida de un software con autooptimización

Utilizando las ideas de [CGG04, Muñ05], podemos dividir el ciclo de vida de un software con capacidad de autooptimización en tres fases:

- En la fase de **diseño** se desarrolla la rutina (o librería), y se utiliza alguna técnica que permita incluir autooptimización en ella, lo que incluye diseñar algún procedimiento de instalación de la rutina. Con esta técnica se pretende obtener el comportamiento general de la rutina en los sistemas donde se instale, y puede ser básicamente de dos tipos:

  - **Empírica**, basada en una serie de ejecuciones para tamaños de problema y del sistema determinados.
  - **Modelado**, con obtención de un modelo teórico del tiempo de ejecución de la rutina, el cual se podrá adaptar a distintos sistemas por medio de un conjunto reducido de ejecuciones.

- La **instalación** de la rutina en un sistema computacional determinado se realizará utilizando la técnica de instalación diseñada en la fase anterior, y la información generada se almacenará de alguna manera para ser usada en la fase siguiente. Puede consistir en la ejecución para rangos de tamaños de problema y sistema suficientemente amplios para dar información completa del comportamiento de la rutina (instalación empírica) o con valores que permitan aproximar satisfactoriamente los valores de los parámetros que reflejan el comportamiento del sistema computacional en el modelo del tiempo de ejecución (instalación basada en modelado). La información generada se almacena de algún modo, y lo más útil puede ser insertarla en la rutina incluyendo en ella una capa de decisión, y recompilar obteniendo así una rutina con capacidad de autooptimización.

- En la **ejecución** de la rutina, cuando se quiere resolver un problema de un tamaño determinado y en un sistema concreto, se utiliza la capa de decisión y la información que se ha incluido en ella para seleccionar la mejor forma teórica de ejecutar la rutina con la finalidad de reducir su tiempo de ejecución. Las decisiones a tomar pueden consistir en seleccionar el algoritmo básico a utilizar si se dispone de varios, el número de *threads* en cada nivel de paralelismo, etc.

6.1.2. Modelado del tiempo de ejecución

Una de las técnicas utilizadas para el desarrollo de rutinas con capacidad de autooptimización es la de parametrización del modelo del tiempo de ejecución [Muñ05]. Se construye un modelo analítico del tiempo de ejecución de la rutina, de manera que este modelo sea una herramienta útil para decidir los valores de unos parámetros con los que se obtenga una ejecución eficiente, y con ello minimizar su tiempo de ejecución. Para esto, el modelo debe reflejar las características del cómputo y de comunicaciones de los algoritmos y del sistema sobre el que se ejecutará. El modelo es de la forma $t(s) = f(s, AP, SP)$, donde s representa el tamaño de la entrada, SP (System Parameters) los parámetros que reflejan las características del sistema, y AP (Algorithmic Parameters) los que intervienen en el algoritmo y cuyos valores deben ser seleccionados adecuadamente para obtener un tiempo de ejecución reducido.

Parámetros SP típicos son: el coste de una operación aritmética, o de operaciones de distintos tipos o niveles (BLAS 1, 2 y 3 en rutinas de álgebra lineal), los tiempos de inicio de las comunicaciones y de envío de un dato en rutinas con

operaciones de comunicación, o los de creación y gestión de *threads* en rutinas en memoria compartida. Con la utilización de estos parámetros se reflejan las características del sistema de cómputo y de comunicaciones, tanto físico (hardware) como lógico (las librerías que se utilizan para llevar a cabo las comunicaciones y operaciones de cómputo básicas).

Dependiendo del tipo de rutina y de la arquitetura del sistema computacional para el que se haya desarrollado, algunos parámetros AP típicos son: el número de procesadores o *cores* a utilizar de entre todos los disponibles, qué procesadores utilizar si el sistema es heterogéneo, el número de procesos a poner en marcha y su mapeo en el sistema físico, parámetros que identifiquen la topología lógica de los procesos (como puede ser el número de filas y columnas de procesos en un algoritmo para malla lógica 2D), el tamaño de los bloques de comunicación o de particionado de los datos entre los procesos, el tamaño de los bloques de computación en algoritmos de álgebra lineal que trabajan por bloques, el número de *threads* en cada nivel en rutinas con paralelismo multinivel, etc. El valor de todos estos parámetros debe ser seleccionado para obtener tiempos de ejecución reducidos, y no podemos pretender que usuarios no expertos tengan suficiente conocimiento (del problema, del algoritmo y del sistema) como para realizar una selección satisfactoria.

Como hemos indicado, los valores de los SP se obtendrán en el momento de instalar la rutina en un sistema particular. Para esto, el diseñador de la rutina debe haber desarrollado el modelo del tiempo de ejecución, haber identificado los parámetros SP que intervienen en el modelo, y haber diseñado una estrategia de instalación, que incluye para cada SP los experimentos a realizar para su estimación y los parámetros AP y sus valores con los que hay que experimentar. Los valores obtenidos para los SP se incluyen junto con el modelo del tiempo de ejecución en la rutina que se está optimizando, que se instala de esta forma con información del sistema para el que se está optimizando.

En tiempo de ejecución, para un tamaño de la entrada concreto, la rutina obtiene de manera automática valores de los AP con los que se obtiene una ejecución óptima según el modelo de ejecución de la rutina y los valores de los SP obtenidos en la instalación. El tiempo de obtención de estos valores debe ser reducido debido a que incrementa el tiempo de ejecución de la rutina. Si particularizamos estos parámetros a nuestro código, tendríamos que los valores de AP serían el algoritmo a utilizar (`zsysv` o `zgesv`) y el número de *threads* OpenMP y MKL.

6.1.3. Instalación empírica

En algunos casos no es posible obtener un modelo satisfactorio del tiempo de ejecución y es necesario realizar una instalación empírica. Esto sucede, por ejemplo, cuando trabajamos con una rutina de la que no conocemos su código al no haberla diseñado nosotros, o cuando la rutina o el sistema son suficientemente complejos para que no sea sencillo modelar teóricamente su comportamiento. En este caso, la instalación consistirá en realizar ejecuciones para un conjunto de tamaños del problema (conjunto de instalación), variando los posibles valores de los parámetros algorítmicos, obteniendo una tabla de los parámetros AP que proporcionan los menores tiempos de ejecución para cada tamaño. Así los parámetros del sistema no se obtienen ni se utilizan, sino que será la tabla la que refleje el comportamiento de la rutina en el sistema. El problema es que si tenemos muchos valores en el conjunto de instalación o el tiempo de ejecución para esos valores es muy alto, y además el número de valores de los parámetros algorítmicos es alto, el tiempo de ejecución se dispara, y será necesario utilizar alguna técnica para reducirlo [CCGV12].

En nuestro caso el tamaño del problema viene dado por la complejidad del mallado y el número de frecuencias, con lo que el conjunto de instalación contendría pares de valores (n, f). Para cada uno de estos pares se realizarían experimentos para cada posible conjunto de posibles valores de los parámetros algorítmicos, pero, tal como indicamos en capítulos anteriores, no es necesario realizar los experimentos para todos los valores, sino que se puede reducir el número de experimentos de varias formas:

- Para seleccionar entre `zsysv` y `zgesv` basta obtener el número de *cores* a partir del cual `zgesv` es preferible (capítulo 3).

- Para seleccionar el número de *threads* se puede obtener el óptimo con experimentación exhaustiva para el tamaño menor con el que se experimenta, y tomar ese valor como punto de partida para el siguiente tamaño. De esta forma se obtienen tablas de ejecución como las del capítulo 4, donde no se hacen todas las ejecuciones posibles y además las que se hacen son de las que consumen menos tiempo.

- Es posible utilizar un modelo teórico del tiempo para seleccionar unos valores de partida con los que experimentar. Por ejemplo, si tenemos un modelo cúbico $t(n,p) = \frac{n^3}{p} + n^2 p$, con n el tamaño del problema y p el número de procesos, para un problema particular, $n = n_0$, se iguala a cero la derivada del tiempo respecto a p, obteniendo un valor de p cercano al que proporcionaría el óptimo experimental: $p = \sqrt{n_0}$.

6.2. Posibilidades de autooptimización en el software MEATSS

De los resultados de los experimentos en los capítulos anteriores hemos concluido que en líneas generales es mejor utilizar únicamente paralelismo OpenMP para el bucle en frecuencias con un número de *threads* no excesivamente alto (8 en Arabí y 64 en Ben) y con la rutina `zsysv`, y que cuando se resuelve un problema con una única frecuencia se usa MKL *multithread* con la rutina `zgesv`.

Aunque esas recomendaciones a un posible usuario de la rutina pueden ser suficientes para que obtenga resultados satisfactorios, no podemos asegurar que proporcionen las ejecuciones óptimas en otros casos y sistemas con los que no hemos experimentado. Por ejemplo, ¿cuál es la mejor opción si el número de frecuencias es reducido? ¿y si no es múltiplo o divisor del número de *cores*? ¿y si disponemos de una nueva versión de MKL en la que se explote mejor el paralelismo en la rutina `zsysv`? ¿y si el número de *cores* en el sistema difiere mucho del de los sistemas donde hemos experimentado (por ejemplo un sistema con 1000 *cores*)? ¿y si el coste de las operaciones básicas de computación y gestión de *threads* es sustancialmente diferente del de los sistemas utilizados? ... Una opción es que al instalar el software en un nuevo sistema el usuario repitiera los experimentos aquí realizados para este nuevo sistema, pero puede ser preferible incluir capacidad de autooptimización en el software. En esta sección analizamos un par de ideas sobre cómo se podría realizar la autooptimización en el software MEATSS.

6.2.1. Selección de la rutina de resolución de sistemas

En el capítulo 3 comparamos el comportamiento de las rutinas `zsysv` y `zgesv` en Saturno, y comprobamos que con pocos *cores* es preferible utilizar

`zsysv`, y con un número suficientemente grande es preferible `zgesv`. Analizamos cómo se pueden aplicar en este caso las dos técnicas de instalación comentadas previamente.

Instalación empírica

Si seleccionamos como conjunto de instalación para los tamaños de los sistemas a resolver el $\{256, 768, 1280, 1792, 2304, 2816, 3328, 3840\}$, y hacemos una instalación exhaustiva consistente en realizar para cada uno de los tamaños en el conjunto ejecuciones con todos los posibles números de *threads*, el tiempo de instalación es de 79 segundos para `dsysv`, de 62 para `dgesv`, de 217 para `zsysv` y de 231 para `zgesv`. Este tiempo no es muy grande, pero se puede reducir usando una búsqueda guiada del número de *threads* óptimo, tal como se explica en [CCGV12] para la multiplicación de matrices. Por ejemplo, una búsqueda guiada para `dsysv` con un umbral de parada de un 1 % (cuando la ejecución para un número de *threads* excede en más de un 1 % el óptimo con los valores anteriores no se sigue variando el número de *threads*) produce un tiempo de instalación de 9 segundos, y cuando el umbral es de un 10 % el tiempo de instalación es de 36 segundos. La reducción en el tiempo de instalación puede producir una peor selección del número de *threads*, pero vemos en la tabla 6.1 que la selección con el umbral 10 % es satisfactoria. La última fila muestra la desviación media con respecto al óptimo de cada método de instalación y del número óptimo de *threads*. La búsqueda exhaustiva da una desviación de aproximadamente un 2 %, y por tanto no podemos esperar una desviación mejor que esa con ningún método de instalación. La misma desviación se obtiene con búsqueda guiada con umbral 10 %. Con el umbral 1 % la desviación aumenta ligeramente al 3 %, lo que no es relevante y además se obtiene con un menor tiempo de instalación. Estos resultados se deben a la forma del tiempo de ejecución de la rutina en Saturno (figura 3.5), donde vemos que el tiempo de ejecución es prácticamente constante a partir de un determinado número de *threads*. La desviación media del número óptimo de *threads* es 2 para la búsqueda exhaustiva y guiada con umbral 10 %, y con umbral 1 % crece a aproximadamente 5, lo que representa una peor selección que prácticamente no influye en el tiempo de ejecución debido a la forma del tiempo de ejecución de la rutina.

La instalación guiada se puede extender para instalar varias rutinas juntas, aprovechando la información generada para una rutina para reducir el tiempo de

	óptimo		bús. exhaustiva		bús. guiada 1 %		bús. guiada 10 %	
tamaño	tiempo	*threads*	tiempo	*threads*	tiempo	*threads*	tiempo	*threads*
384	0.00556	5	0.00556	12	0.0056	5	0.0056	12
512	0.0109	10	0.0177	12	0.0116	5	0.0117	12
640	0.0155	16	0.0163	12	0.0172	5	0.0163	12
896	0.0323	16	0.0323	16	0.0352	6	0.0323	16
1024	0.0506	21	0.0512	16	0.0553	6	0.0512	16
1152	0.0565	16	0.0565	16	0.0635	6	0.0565	16
1408	0.0881	21	0.0935	19	0.0939	11	0.0935	19
1536	0.1265	21	0.1312	19	0.1329	11	0.1312	19
1664	0.1275	24	0.1339	19	0.1379	11	0.1339	19
1920	0.1765	20	0.1808	22	0.1765	20	0.1808	22
2048	0.2411	21	0.2458	22	0.2437	20	0.2458	22
2176	0.2335	24	0.2384	22	0.2341	20	0.2384	22
2432	0.3022	20	0.3079	22	0.3139	24	0.3079	22
2560	0.3988	20	0.4054	22	0.4207	24	0.4054	22
2688	0.3691	20	0.3816	22	0.3753	24	0.3816	22
2944	0.4513	20	0.4513	20	0.4649	24	0.4513	20
3072	0.5881	22	0.5922	20	0.5919	24	0.5911	20
3200	0.5564	21	0.5759	20	0.5568	24	0.5759	20
3456	0.6707	20	0.6707	20	0.6847	23	0.6707	20
3584	0.8348	24	0.8397	20	0.8382	23	0.8397	20
3712	0.7241	20	0.7841	20	0.802	23	0.7841	20
3968	0.924	20	0.9243	20	0.9419	23	0.9337	21
4096	1.1419	22	1.1501	20	1.1432	23	1.154	21
desviación			0.022	1.91	0.031	5.26	0.022	1.91

Tabla 6.1: Comparación del tiempo de ejecución y del número de *threads* óptimo con los obtenidos con distintas técnicas de instalación, para varios tamaños de validación, para la rutina `dsysv` en Saturno

ejecución de las otras. Por ejemplo, si primero se instala `dsysv` y el número de *threads* seleccionado se utiliza para empezar la búsqueda en `dgesv`, el tiempo de instalación de las dos rutinas se reduce de 141 a 45 segundos.

La rutina a usar en la solución de un problema se selecciona a partir de la información generada en la instalación. Para esto, se almacena para cada tamaño de problema del conjunto de instalación el número de *threads* para el que la rutina `gesv` tiene por primera vez menor tiempo de ejecución que la `sysv`. Si en la ejecución, para un tamaño de problema dado, consideramos que el cambio de `sysv` a `gesv` ocurre con el mismo número de *threads* que en el problema de tamaño más cercano del conjunto de instalación, el número de *threads* seleccionado se muestra en la tabla 6.2, que muestra los resultados en Saturno y en un *quadcore*. Para los tamaños de problema en el que el cambio no ocurre en el punto seleccionado se muestran los valores experimental y estimado. En Saturno el punto en que se produce el cambio se predice bien en 35 de 46 casos, y en el *quadcore* en 30 de 46. El comportamiento en los dos sistemas es ligeramente diferente, y el número de *threads* en que ocurre el cambio aumenta con el tamaño del problema más rápido en Saturno que en el *quadcore*, lo que puede deberse a la diferencia en la jerarquía de memorias de los dos sistemas. Además, en el *quadcore* se observa un comportamiento extraño, pues las rutinas `gesv` son preferibles a las `sysv` para problemas pequeños, lo que no es predecible de los experimentos en otros sistemas y sin embargo es detectado en la instalación.

Instalación basada en el modelo del tiempo de ejecución

El tiempo de ejecución secuencial de las rutinas de resolución de sistemas de ecuaciones consideradas es de orden $O(n^3)$, y en el modelo aparecerán términos en n^3, n^2 y n. En las versiones paralelas, si t representa el número de *threads*, el término de mayor orden (n^3) se dividirá por t, y otros términos pueden aparecer multiplicados por t. Así, podemos considerar las distintas combinaciones de $\{n^3, n^2, n, 1\} \times \{t, 1, \frac{1}{t}\}$, pero para n^3 sólo incluimos $\frac{n^3}{t}$, y los términos de menor orden ($\frac{n}{t}$, t, 1 y $\frac{1}{t}$) no se incluyen, con lo que el modelo del tiempo de ejecución es:

$$T(n,t) = k_1\frac{n^3}{t} + k_2n^2t + k_3n^2 + k_4\frac{n^2}{t} + k_5nt + k_6n \tag{6.1}$$

Se puede realizar ajuste por mínimos cuadrados para estimar los valores de

	Saturno		*quadcore*	
tamaño	`dsysv-dgesv`	`zsysv-zgesv`	`dsysv-dgesv`	`zsysv-zgesv`
384	2	3 / 2	1	2 / 1
512	2	3 / 2	2 / 1	2 / 1
640	2	3	2 / 1	2
896	3 / 2	3	2 / 1	3 / 2
1024	2	3	1	2
1152	3	3	2	3
1408	3	3	2	3
1536	3	3	2	3
1664	3	3	2	3
1920	3	5 / 3	2	3
2048	3	5 / 3	2	3
2176	4	5	2	3
2432	4	5	2	3
2560	4	5	2	3
2688	4	5	2	3
2944	5 / 4	7	2	3
3072	3 / 4	7	3 / 2	3
3200	5 / 4	7	2 / 3	3
3456	5 / 4	7	3	3
3584	5 / 4	9 / 7	2 / 3	4 / 3
3712	5	9	3 / 4	3 / -
3968	5	9	3 / 4	3 / -
4096	5	9	3 / 4	3 / -
desviación	0.26	0.35	0.30	0.43

Tabla 6.2: Comparación del número de *threads* para el que `gesv` mejora a `sysv` y el seleccionado con la información almacenada en la instalación

los coeficientes k_i para una rutina particular en un sistema particular. Se realizan experimentos para diferentes tamaños de problema y del sistema, y la función que representa la desviación entre los tiempos teóricos y experimentales es:

$$F\left(k_1,k_2,k_3,k_4,k_5,k_6\right)=\sum_{i=1}^{r}\left(T_i-T\left(n_i,t_i\right)\right)^2 \tag{6.2}$$

donde r es en número de experimentos, T_i el tiempo experimental del experimento i, y n_i y t_i el tamaño de la matriz y el número de *threads* en el experimento i. Haciendo las derivadas parciales de F con respecto a cada k_i e igualando a cero se obtiene el sistema:

$$\begin{aligned}
k_1\sum_{i=1}^{r}\frac{n_i^6}{t_i^2}+k_2\sum_{i=1}^{r}n_i^5+k_3\sum_{i=1}^{r}\frac{n_i^5}{t_i}+k_4\sum_{i=1}^{r}\frac{n_i^5}{t_i^2}+k_5\sum_{i=1}^{r}n_i^4+k_6\sum_{i=1}^{r}\frac{n_i^4}{t_i}&=\sum_{i=1}^{r}T_i\frac{n_i^3}{t_i}\\
k_1\sum_{i=1}^{r}n_i^5+k_2\sum_{i=1}^{r}n_i^4t^2+k_3\sum_{i=1}^{r}n_i^4t_i+k_4\sum_{i=1}^{r}n_i^4+k_5\sum_{i=1}^{r}n_i^3t_i^2+k_6\sum_{i=1}^{r}n_i^3t_i&=\sum_{i=1}^{r}T_in_i^2t_i\\
k_1\sum_{i=1}^{r}\frac{n_i^5}{t_i}+k_2\sum_{i=1}^{r}n_i^4t_i+k_3\sum_{i=1}^{r}n_i^4+k_4\sum_{i=1}^{r}\frac{n_i^4}{t_i}+k_5\sum_{i=1}^{r}n_i^3t_i+k_6\sum_{i=1}^{r}n_i^3&=\sum_{i=1}^{r}T_in_i^2\\
k_1\sum_{i=1}^{r}\frac{n_i^5}{t_i^2}+k_2\sum_{i=1}^{r}n_i^4+k_3\sum_{i=1}^{r}\frac{n_i^4}{t_i}+k_4\sum_{i=1}^{r}\frac{n_i^4}{t_i^2}+k_5\sum_{i=1}^{r}n_i^3+k_6\sum_{i=1}^{r}\frac{n_i^3}{t_i}&=\sum_{i=1}^{r}T_i\frac{n_i^2}{t_i}\\
k_1\sum_{i=1}^{r}n_i^4+k_2\sum_{i=1}^{r}n_i^3t_i^2+k_3\sum_{i=1}^{r}n_i^3t_i+k_4\sum_{i=1}^{r}n_i^3+k_5\sum_{i=1}^{r}n_i^2t_i^2+k_6\sum_{i=1}^{r}n_i^2t_i&=\sum_{i=1}^{r}T_in_it_i\\
k_1\sum_{i=1}^{r}\frac{n_i^4}{t_i}+k_2\sum_{i=1}^{r}n_i^3t_i+k_3\sum_{i=1}^{r}n_i^3+k_4\sum_{i=1}^{r}\frac{n_i^3}{t_i}+k_5\sum_{i=1}^{r}n_i^2t_i+k_6\sum_{i=1}^{r}n_i^2&=\sum_{i=1}^{r}T_in_i
\end{aligned} \tag{6.3}$$

En la tabla 6.3 se compara el punto en el que se produce el cambio de `sysv` a `gesv`, obtenido experimentalmente y el predicho por el modelo. Para doble precisión los parámetros de la ecuación 6.1 se han estimado con 30 ejecuciones aleatorias, pero para las rutinas complejas no se obtenían resultados satisfactorios con un número reducido de ejecuciones, y han sido necesarias 200 ejecuciones. Para las rutinas con reales el número de *threads* se elige correctamente en más de la mitad de los casos, y para las complejas sólo en unos pocos casos, pero con los dos tipos la desviación media es baja. La diferencia es mayor para problemas pequeños, para los que el modelo no es tan satisfactorio como para problemas grandes. El tiempo de instalación es de aproximadamente 15 y 540 segundos para rutinas con reales y complejos.

size	dsysv-dgesv	zsysv-zgesv
256	2 / 13	2 / 3
384	2 / 8	3
512	2 / 5	3 / 4
640	2 / 4	3 / 4
768	2 / 3	3 / 4
896	3	3 / 4
1024	2 / 3	3 / 4
1152	3	3 / 4
1280	3	3 / 5
1408	3	3 / 5
1536	3	3 / 5
1664	3	3 / 5
1792	3	3 / 5
1920	3	5
2048	3	5
2176	4 / 3	5 / 6
2304	4 / 3	5 / 6
2432	4	5 / 6
2560	4	5 / 6
2688	4	7 / 6
2816	4	7
2944	5 / 4	7
3072	4	7
3200	5	7
3328	4	7
3456	5 / 4	7
3584	5 / 4	8 / 7
3712	5	9 / 8
3840	5	9 / 8
3968	5	9 / 8
4096	5 / 6	9 / 8
desviación	0.96	0.87

Tabla 6.3: Comparación del número de *threads* para el que `gesv` mejora a `sysv` y el seleccionado con el modelo, en Saturno.

6.2.2. Selección del número de *threads* en rutinas híbridas

La selección del número óptimo de *threads* de la versión con paralelismo OpenMP se puede realizar de forma similar a la utilizada en la subsubsección anterior. En el capítulo 4 comprobamos que la información generada para el mallado de menor complejidad es útil para empezar la búsqueda en mallados de mayor complejidad con un número de *threads* más cercano al óptimo, obteniéndose así una reducción importante en el tiempo de instalación. Así, vemos que en las tablas 4.1 y 4.2 muchas entradas no han sido computadas, y algunas de las que hay se podrían evitar con una búsqueda guiada dependiendo del umbral utilizado, con lo que se evitan las ejecuciones más costosas y se produce una gran reducción en el tiempo de instalación. En principio habría que realizar experimentos para varias frecuencias (f) y distintos tamaños del problema, pero si hacemos coincidir el número de frecuencias con el de *cores* en el sistema se determina el número de *threads* con el que se obtienen las mejores prestaciones (t_{opt}), con lo que cuando $f \geq t_{opt}$ se usarán t_{opt} *threads*, y cuando $f < t_{opt}$ se usará un número de *threads* igual al de frecuencias. Así, en Arabí (tabla 4.1) $t_{opt} = 8$ en todos los casos, y se usarán 8 *threads* si el $f \geq 8$ y f *threads* si $f < 8$. En Ben (tabla 4.2) $t_{opt} = 32$ para el mallado más simple, y $t_{opt} = 64$ en los otros casos (sólo consideramos divisores de 128).

Como el t_{opt} puede ser menor que el número de *cores* en algunos sistemas (principalmente si estos son grandes como ocurre con Ben), en el algoritmo 6 se sustituirá el valor c por t_{opt} en el bucle `for` en el que se recorren los divisores de c. Y en la capa de toma de decisiones se incluiría en el algoritmo en las zonas donde se selecciona el número de *threads* OpenMP y MKL. El número de *threads* OpenMP se selecciona como aparece en el algoritmo, usando un índice ind distinto de $ntmkl$ para recorrer los divisores de t_{opt}. Teniendo en cuenta el número de *cores* disponible tras seleccionar $ntomp$, se selecciona el valor de $ntmkl$ y la rutina `zsysv` o `zgesv` que proporcionan el menor tiempo de ejecución teórico para la resolución de los sistemas de ecuaciones usando MKL *multithread* según la información y para un máximo de $c/ntomp$. Por ejemplo, si consideramos $f = 317$ en Ben con $c = 128$ y un tamaño mediano del problema, para el que $t_{opt} = 64$, en la tabla 6.4 se muestran las frecuencias que se resuelven en cada pasada, junto con los valores seleccionados de *ntomp* y *ntmkl* y el método de resolución. El método y el valor de *ntmkl* se han seleccionado usando la información de la tabla 3.9.

Tabla 6.4: Frecuencias resueltas y valores de los parámetros algorítmicos seleccionados en cada pasada del algoritmo 6, en Ben con $f = 317$, $c = 128$ y para un problema de complejidad media

frecuencias	$ntomp$	$ntmkl$	método
1-256	64	2	`zsysv`
257-288	32	4	`zsysv`
289-304	16	8	`zgesv`
305-312	8	16	`zgesv`
313-316	4	32	`zgesv`
317-317	1	128	`zgesv`

6.3. Resumen y conclusiones

En este capítulo se han presentado algunos conceptos básicos de autooptimización de software paralelo, y se han analizado algunas ideas sobre cómo aplicarlos al software MEATSS. No se ha generado una versión de MEATSS con autooptimización, pero con las ideas presentadas no debe ser difícil generarla, y sería necesario validar el correcto funcionamiento de la versión autooptimizada usando otros tamaños de problema y en otros sistemas computacionales distintos.

Capítulo 7

Conclusiones y trabajos futuros

El objetivo principal de este proyecto de reducir el tiempo de ejecución del software MEATSS por medio de la programación paralela se ha cubierto con éxito. Para conseguirlo se ha realizado un análisis del código, y se han desarrollado diferentes versiones optimizadas del software MEATSS aplicando distintos tipos de paralelismo, dando lugar a códigos optimizados que hacen un uso eficiente de los recursos en sistemas de grandes dimensiones (Supercomputador Ben Arabí de la Fundación Parque Científico de Murcia). Concretamente se han producido enormes mejoras en la parte de la resolución de sistemas, pasando de tardar, 28620 segundos en la resolución de 128 frecuencias, empleando el código inicial, a tan solo 123 segundos en la versión paralelizada con OpenMP.

Del análisis de los datos obtenidos utilizando la herramienta de *profiling* `gprof` se detectó que algunas de las secciones más lentas del código se correspondían con rutinas que podían sustituirse por sus homólogas de la librería de Intel MKL. Se han realizado simulaciones variando el número de frecuencias y el mallado, utilizando tanto la versión secuencial como la paralela de MKL. Sustituyendo la función `zsysv` por su correspondiente llamada a la librería MKL, se han obtenido unos valores de *speed-up* máximos de 5.2 y de 3.4 para el caso del Superdome y del Cluster, respectivamente. La mejora en los tiempos de ejecución se hace patente especialmente con los mallados complejos, los cuales dan lugar a matrices de tamaños mayores. Con el objetivo de mejorar los resultados anteriores, se escogió otra rutina para la resolución de sistemas de ecuaciones lineales más general, `zgesv`, a priori peor al no tener en cuenta la simetría de las matrices bajo estudio. El resultado ha sido una mejora en los tiempos respecto a la subrutina `zsysv` a partir de un determinado número de *cores*, extrayendo como

conclusión que la subrutina `zgesv` escala mejor. Con las mejoras introducidas (utilización de las rutinas optimizadas de MKL, uso de paralelismo intrínseco con rutinas *multithread*, y uso de la rutina para matrices no simétricas) se obtiene una reducción importante en el tiempo de ejecución en `dynamicpart2`, llegando a pasar de 34.17 segundos por frecuencia a 4.55 segundos para el mejor caso.

Para obtener una versión con paralelismo OpenMP, dada la estructura inicial del programa, en la que se hacía uso extensivo de variables globales, se ha tenido que hacer una total reescritura del código para evitar accesos simultáneos a zonas de memoria. Para el Cluster, en todos los casos se ha obtenido el mejor tiempo utilizando el número máximo de *cores* disponible, 8. En cambio, para las ejecuciones en el Superdome, el número de *cores* para el cual el tiempo de ejecución es óptimo es 64, valor que se corresponde con la mitad de su capacidad. Por tanto, concluimos que, para nuestro caso particular, no merece la pena lanzar ejecuciones para un número de *cores* mayor de 64 en el Superdome, y en general en sistemas NUMA grandes el número óptimo de *threads* estará lejos del número de *cores*. Para aplicaciones con varias frecuencias, la utilización de paralelismo OpenMP produce una mayor aceleración que el uso de MKL en la resolución de los sistemas, llegándose en algunos casos a un *speed-up* de 35 en el Superdome.

Con el fin de intentar mejorar aún más los resultados se han realizado experimentos utilizando paralelismo híbrido OpenMP+MKL, llegando a la conclusión de que en el sistema en el que se ha trabajado solamente para el caso de problemas pequeños merece la pena la utilización de programación híbrida, y en el resto de casos siempre es recomendable utilizar solamente OpenMP debido a la gran ganancia obtenida con éste.

El código de MEATSS ha sido estudiado y optimizado para un hardware concreto, los sistemas Ben y Arabí del Centro de Supercomputación del Parque Científico de Murcia. Como trabajo futuro se podría realizar la implementación de técnicas cómo las expuestas en el capítulo 6, que aseguren el uso eficiente, sin intervención humana, en sistemas para los que no ha sido originalmente creado (*autotuning*).

Además de las ideas del párrafo anterior, existen varias líneas de trabajo futuro en las que podría ser interesante profundizar. Algunas de ellas serían: seguir la misma metodología para la optimización y paralelización de otros módulos de MEATSS, como por ejemplo el que calcula los campos electromagnéticos; au-

mentar la complejidad de las geometrías (en base a lo expuesto podemos predecir que los resultados continuarán mejorando); realizar una asignación de los recursos por *threads* en lugar de por frecuencias en caso de que la memoria sea un factor limitante; utilizar librerías alternativas a MKL tales como ATLAS [Atl] o PLASMA [PLA]; realizar una implementación con CUDA [CUD] que aproveche las modernas tarjetas gráficas; o incluso añadir paralelismo por paso de mensajes con MPI [MPI].

Apéndice A
Descripción de los campos del flat y del graph profile

Flat Profile

Los campos que aparecen en el flat profile son los siguientes:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
```

%time: Es el porcentaje del tiempo total de ejecución que el programa gasta en la función.

cumulative seconds: Es la suma del número total de segundos que se ha gastado en la ejecución de la función y de las funciones que se encuentran por encima de ésta.

self seconds: Número de segundos empleados en la ejecución de la función solamente.

calls: Número total de veces que la función ha sido llamada.

self s/call: Nos indica el promedio de tiempo, en segundos, gastado en la función por llamada.

total s/call: Este campo representa el promedio de tiempo, en segundos, empleado por la función y sus descendientes por llamada.

name: Indica el nombre de la función.

Campos del graph profile

Como se ha comentado en el apartado de *profiling*, cada entrada de la tabla consta de varias líneas. La línea situada más a la izquierda con el número de índice entre corchetes lista la función actual. Las líneas por encima de ella son las funciones que llaman a dicha función, también llamadas `parent lines`. Las líneas que se encuentran debajo muestran las funciones que han sido llamadas desde la función y se denominan `descendant lines`.

Dependiendo de si se trata de una línea de función, una `parent line` o una `descendant line` el significado de los campos será distinto.

Línea de función

```
index % time    self  children    called     name
```

index: Número entero que se le asigna a cada función.

% time: Porcentaje del tiempo total gastado por la función y sus hijos, incluyendo las subrutinas llamadas desde ésta.

self: Cantidad de tiempo total gastada por la función. Debe ser idéntica a la mostrada en el campo `self seconds` del `flat profile`.

children: Tiempo total gastado en las llamadas a subrutinas hechas por la función. Debe ser igual a la suma de los campos **self** y **children** listados en las `descendant lines`. (Cantidad de tiempo propagada en la función por sus hijos)

called: Número de veces que la función ha sido llamada.

name: Nombre de la función.

Functions parents

self: Cantidad de tiempo que fue propagada directamente desde la función a su padre.

children: Cantidad de tiempo que fue propagada desde los hijos de la función a su padre.

called: Número de veces que el padre llama a la función / número total que la función ha sido llamada.

name: Nombre del padre. El índice del padre está escrito justo después.

Functions children

self: Cantidad de tiempo que fue propagada directamente desde el hijo a la función.

children: Cantidad de tiempo que fue propagada desde los hijos de los hijos a la función.

called: Número de veces que la función llama a este hijo / número total de veces que el hijo ha sido llamado.

name: Nombre del hijo. El índice del hijo está escrito justo después.

Bibliografía

[AAV+04] 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(1/2/3):75–84, 2004.

[AGMV08] Francisco Almeida, Domingo Giménez, José Miguel Mantas, and Antonio M. Vidal. *Introducción a la programación paralela*. Paraninfo Cengage Learning, 2008.

[Ans] Ansys. `http://www.ansoft.com/products/hf/hfss/`.

[Atl] Atlas. `http://math-atlas.sourceforge.net/`.

[BLA] BLAS. `http://netlib.org/blas/`.

[Bre94] E. A. Brewer. Portable high-performance supercomputing: High-level platform-dependent optimization, Ph.D. Thesis, 1994.

[CCGV12] Jesús Cámara, Javier Cuenca, Domingo Giménez, and Antonio M. Vidal. Empirical autotuning of two-level parallel linear algebra routines on large cc-NUMA systems. In *ISPA*, 2012.

[CDLR03] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self Adapting Software for Numerical Linear Algebra and LAPACK for Clusters. *Parallel Computing*, 29:1723–1743, 2003.

[CGG+03] Javier Cuenca, Domingo Giménez, José González, Jack Dongarra, and Kenneth Roche. Automatic optimisation of parallel linear algebra routines in systems with variable load. In *PDP*, pages 409–416, 2003.

[CGG04] J. Cuenca, D. Giménez, and J. González. Architecture of an automatic tuned linear algebra library. *Parallel Computing*, 30(2):187–220, 2004.

[CGG12] Javier Cuenca, Luis-Pedro García, and Domingo Giménez. Improving linear algebra computation on NUMA platforms through auto-tuned nested parallelism. In *Proceedings of the 2012 EUROMICRO Conference on Parallel, Distributed and Network Processing*, 2012.

[CGMG05] J. Cuenca, D. Giménez, and J. P. Martínez-Gallar. Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. *Parallel Computing*, 31:717–735, 2005.

[CLCLG12] Luis-Gabino Cutillas-Lozano, José-Matías Cutillas-Lozano, and Domingo Giménez. Modeling shared-memory metaheuristic schemes for electricity consumption. In *DCAI*, 2012.

[CMD+02] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *OpenMP C and C++ Application Program Interface*. OpenMP Architecture Review Board. `http://www.openmp.org/drupal/mp-documents/cspec20.pdf`, 2002.

[CUD] CUDA Zone. `http://www.nvidia.com/cuda`.

[Dat09] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, Computer Science Division, U. C. Berkeley, December 2009.

[FEK] FEKO. `http://www.feko.info/`.

[Fri98] M. Frigo. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the ICASSP Conference*, volume 3, page 1381, 1998.

[FS] Jay Fenlason and Richard Stallman. `http://www.cs.utah.edu/dept/old/texinfo/as/gprof-toc.html`.

[Fun] Fundación Parque Científico de Murcia. http://www.parquecientificomurcia.es/web/centro-de-supercomputacion.

[GAMR03] D. González, F. Almeida, L. Moreno, and C. Rodríguez. Toward the automatic optimization mapping of pipeline algorithms. *Parallel Computing*, 29(2):241–254, 2003.

[GGKK03] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.

[Gid] Gid. http://www.gidhome.com/.

[GL90] G. Golub and C. F. Van Loan. *Matrix Computations*. The John Hopkins University Press, 1990.

[GL09] Domingo Giménez and Alexey Lastovetsky. On the behaviour of the MKL library in multicore shared-memory systems. In *XXI Jornadas de Paralelismo*, 2009.

[HR05] S. Hunold and T. Rauber. Automatic tuning of pdgemm towards optimal performance. In *Euro-Par, LNCS*, volume 3648, pages 837–846, 2005.

[Int] Intel MKL web page. http://software.intel.com/en-us/intel-mkl/.

[JMG09] Sonia Jerez, Juan-Pedro Montávez, and Domingo Giménez. Optimizing the execution of a parallel meteorology simulation code. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*. IEEE, May 2009.

[JSL97] U. Jakobus, I. Sulzer, and F. M. Landstorfer. Parallel implementation of the hybrid MoM/Green's function technique on a cluster of workstations. In *IEE 10th International Conference on Antennas and Propagation*, page 182–185, April 1997.

[KKH04] T. Katagiri, K. Kise, and H. Honda. Effect of auto-tuning with user's knowledge for numerical software. In J. L. Gaudiot S. Vassiliadis and V. Piuri, editors, *Proceedings of the First Conference on Computing Frontiers*, pages 12–25, 2004.

[LAP] LAPACK. http://www.netlib.org/lapack/.

[LCG10] 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 Computational Methods in Water Resources*, June 2010.

[LSF] LSF Platform. http://www.platform.com/workload-management/high-performance-computing.

[Mic] Microstripes. http://www.cst.com/content/products/MST/Overview.aspx.

[mod] http://modules.sourceforge.net/.

[MPI] MPI Forum. http://www.mpi-forum.org/.

[Muñ05] Javier Cuenca Muñoz. Optimización automática de software paralelo de álgebra lineal. Departamento de Ingeniería y Tecnología de los Computadores de la Universidad de Murcia, Ph.D. Thesis, 2005.

[NUM] NUMA en Wikipedia. http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access.

[Opea] OpenCL web page. http://www.khronos.org/opencl/.

[Opeb] OpenMP web page. http://openmp.org/wp/.

[Pé09] Francisco Javier Pérez Soler. *Investigación En Técnicas Numéricas Aplicadas Al Estudio De Nuevos Dispositivos Para Sistemas De Comunicaciones*. PhD thesis, UPCT, SP, 2009.

[PAGAMQ12] Carlos Pérez-Alcaraz, Domingo Giménez, Alejandro Álvarez-Melcón, and Fernando D. Quesada. Parallelizing the computation of Green functions for computational electromagnetism problems. In *PDSEC Workshop, 26th IEEE International Parallel and Distributed Processing Symposium*, 2012.

[PLA] PLASMA. http://icl.cs.utk.edu/plasma/.

[Por] Portable Hardware Locality. http://runtime.bordeaux.inria.fr/hwloc/.

[RWG82] S. M. Rao, D. R. Wilton, and A. W. Glisson. Electromagnetic scattering by surfaces of arbitrarily shape. Technical report, IEEE, 1982.

[Sca] ScaLAPACK. http://www.netlib.org/scalapack/.

[SG98] Marc Snir and William Gropp. *MPI. The Complete Reference. 2nd edition*. The MIT Press, 1998.

[SWG84] D. H. Schaubert, D. R. Wilton, and A. W. Glisson. A tetrahedralmodeling method for electromagnetic scattering by arbitrarily shaped inhomogeneous dielectric bodies. Technical report, IEEE, 1984.

[Tec] Agilent Technologies. http://www.home.agilent.com/agilent/.

[VPDM07] Carlos P. Vicente Quiles, F. Q. Pereira, J. S. G. Díaz, and Michael Mattes. New Investigations of RF Breakdown in Microwave Transmission Lines: Selection of Critical Structures. Technical report, ASAT/ESA, 2007.

[WPD01] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[WTI] WTIME. people.sc.fsu.edu/jburkardt/fsrc/wtime/wtime.html.

[ZOK] Dusan P. Zoric, Dragan I. Olcan, and Branko M. Kolundzija. Solving electrically large EM problems by using Out-of-Core solver accelerated with multiple Graphical Processing Units. Technical report, School of Electrical Engineering University of Belgrade, Serbia.

[ZSM+07] Y. Zhang, T. K. Sarkar, H. Moon, A. De, and M. C. Taylor. Solution of large complex problems in computational electromagnetic using higher order basis in MoM with parallel solvers. In *Proc. IEEE Antenna and Propagation Symp.*, page 5620–5623, 2007.