Tesis de máster

Máster en Computación Paralela y Distribuida

Paralelismo de tareas aplicado a algoritmos matriciales en memoria compartida

Autor: Alberto Martínez Pérez

Director: José E. Román Moltó

Universidad Politécnica de Valencia

Departamento de Sistemas Informáticos y Computación

Fecha: 18/02/2013

UNIVERSIDAD POLITECNICA DE VALENCIA

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Índice

Lista de Figuras

1 Introducción

1.1 Motivación

A día de hoy, los procesadores multicore están presentes en gran cantidad de ordenadores, siendo corrientes procesadores con 4 cores, desde productos electrónicos para el consumo personal a grandes instalaciones de servidores y supercomputadores.

Debido a esto, y a que el número de cores en un procesador seguirá aumentando es de vital importancia intentar aprovechar toda la capacidad que nos ofrecen los recursos de computación que disponemos con modelos de programación fáciles de entender y de usar [9].

Con este objetivo, durante años se han ido creando varias librerías que dan soporte a varios hilos de ejecución, basándose en el paralelismo de tareas: Quark, OpenMP, X-KAAPI, Intel Threading Building Blocks, por nombrar unos pocos.

Estos productos software son capaces de explotar el paralelismo de los algoritmos usando:

- Algoritmos que aprovechen los sistemas multicore, eviten la comunicación y sean asíncronos

Algoritmos que tengan en cuenta los procesadores multicore, es decir, que tengan una mayor escalabilidad, este objetivo se consigue mediante el solapamiento de etapas de comunicación con las de computación.

Dentro de un nodo, el coste de transferencia de datos es relativamente bajo, pero la afinidad temporal continua siendo un aspecto muy importante en el uso efectivo de la memoria cache.

Por lo tanto sea hace necesario el desarrollo de nuevos algoritmos que aumenten la localidad de datos.

Aunque complejidad algorítmica normalmente se expresa como el número de operaciones en vez de como la cantidad de datos que se mueven en memoria.

Esto es una contradicción, ya que coste de la computación proviene principalmente del movimiento de datos en memoria principal, ya que este resulta muy costoso en comparación con las operaciones de cálculo, la única forma de solventar el problema del movimiento de datos es mejorar el uso de la jerarquía de memoria, ya que el tiempo de acceso a cache es mucho menor que el tiempo de acceso a memoria principal.

- Soluciones a los problemas de balanceo de carga

La planificación de tareas basada en grafos acíclicos dirigidos (DAG) requiere nuevas estrategias para optimizar la utilización de recursos sin que peligre la localidad espacial en caché.

Con una planificación dinámica (asignación de tareas en tiempo de ejecución) el trabajo se asigna a hilos basándose en si están disponibles los datos en ese momento. La idea de expresar el computo a través de un grafo de tareas ofrece la flexibilidad de explorar el DAG en tiempo de ejecución.

Existe un concepto importante de camino crítico, que define el límite máximo de paralelización alcanzable y se debe intentar recorrerlo a la máxima velocidad.

Esto es una notable diferencia respecto a los modelos de programación fork-join y de paralelismo de datos, donde puntos de sincronización son necesarios para el correcto funcionamiento del software, por lo que se crean partes secuenciales del código en las que varios cores se encuentran desocupados/parados.

1.2 Objetivo del trabajo

El objetivo del trabajo es demostrar las ventajas que aporta el paralelismo de tareas empleando un modelo flujo de datos en vez de usar un modelo basado en fork-join. Con esta idea, se han utilizado varias librerías para realizar una comparación más detallada.

Para ello se han plantean dos casos de estudio que deben ser paralelizados: el cálculo del coseno de una matriz mediante aproximantes de Padé y el estudio de las atmósferas estelares.

1.3 Estructura de la memoria

El capítulo 2 explica los principios utilizados para la tesis. El capítulo 3 resume algunos de los productos software usados para desempeñar una paralelización mediante tareas, profundizando en las diferencias entre ellos. En el capítulo 4 se desarrolla el cálculo del coseno mediante aproximantes de Padé. Durante el capítulo 5 se realiza un estudio sobre la paralización del problema de las atmósferas estelares. Para finalizar se incluyen las conclusiones más relevantes del trabajo realizado.

2 Modelos de paralelismo

Existen muchos modelos de programación paralela, pero a la hora de separarlos por categorías se aprecian niveles de solapamiento. Durante este resumen, nosotros debemos preocuparnos de los contrastes entre estos dos modelos de paralelismo principalmente.

- Paralelismo de datos

  -El "bucle paralelo", la concurrencia se obtiene cuando se aplica la misma operación sobre todos los elementos de un conjunto.

- Paralelismo de tareas

  -Fork-join, es una descomposición del problema en partes, su asignación como tareas, una solución a estas tareas, y una sincronización para mantener la coherencia de los datos a los que se accede.

  -Tareas con dependencias dirigidas mediante flujo de datos.

2.1 Paso de mensajes

El modelo de programación de paso de mensajes esta basado en la abstracción de un sistema paralelo con un espacio de direcciones distribuido, donde cada procesador tiene una memoria local, que es únicamente accesible desde ese procesador.

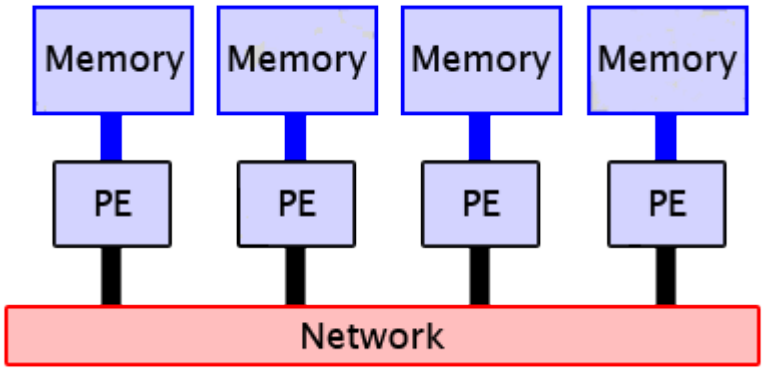


Figura 1. Modelo de Memoria distribuida

No existe una memoria global. El intercambio de datos se debe realizar a través del paso de mensajes. Para transferir datos de la memoria local del procesador A a la memoria local del pro-

cesador B, A debe enviar a B los datos, y B debe recibir los datos que serán almacenados en su memoria local.

Para garantizar la portabilidad de los programas no se asume ninguna topología de interconexión de red. De hecho, se asume que todos los procesadores pueden enviar mensajes a todos procesadores de forma directa.

Un programa basado en paso de mensajes ejecuta varios procesos, cada uno de estos proceso tiene sus propios datos locales. Normalmente, un proceso se ejecuta en un core de la máquina.

El número de procesos a ejecutar se especifica al iniciar el programa, cada proceso puede acceder a sus datos locales y puede intercambiar sus datos locales con otros procesos por medio del envío y recepción de mensajes.

En teoría, cada uno de los procesos puede ejecutar un programa diferente(MPMD, multiple program multiple data), pero por facilidad de programación, se asume que cada uno de los procesos ejecuta el mismo programa (SPMD, single program, multiple data).

En la práctica, esto no es realmente una restricción, ya que cada proceso puede ejecutar diferentes partes del programa, dependiendo, por ejemplo, del rango del proceso.

Los procesos que usan el modelo de paso de mensajes pueden intercambiar datos usando las operaciones de comunicación, como por ejemplo, operaciones de transferencia punto-a-punto o de comunicación global como broadcast, reduce, etc... Normalmente estas operaciones vienen implementadas en alguna librería de comunicación, como por ejemplo MPI.

Este modelo de paralelismo no se utilizará durante este trabajo.

2.2 Paralelismo de datos

Muchas de las aplicaciones de software que procesan grandes cantidades de datos y que, por lo tanto, tienen un tiempo de ejecución muy grande se dedican a modelar fenómenos del mundo real. Por ejemplo, las imágenes y los fotogramas de vídeo son “capturas” de un instante donde las diferentes partes de la fotografía capturan eventos físicos independiente y simultáneos.

Se puede diseñar un algoritmo de manera que la misma operación se aplique de forma concurrente sobre un conjunto de elementos de una estructura de datos. La concurrencia está en los datos.

Es decir, una operación se puede realizar de forma simultanea en varios elementos de las estructuras de datos, con la seguridad de que no se alterará el resultado final.

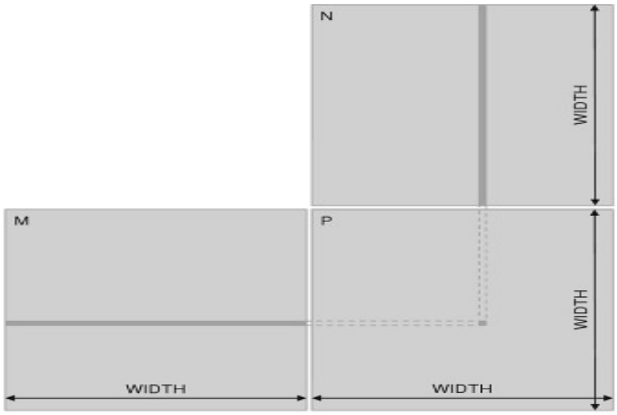


Figura 2. Multiplicación matriz-matriz

Como ejemplo de este tipo de paralelismo, puede ser la multiplicación matriz-matriz.

Cada elemento de la matriz P de tamaño MxN es generado a partir del producto escalar de una columna de la matriz N por una fila de la matriz M.

Este modelo lo utilizan algunas librerías paralelas de cálculo matricial como MKL.

2.3 Memoria compartida

El paradigma de memoria compartida significa que todos los procesos comparten un espacio único de direcciones de memoria.

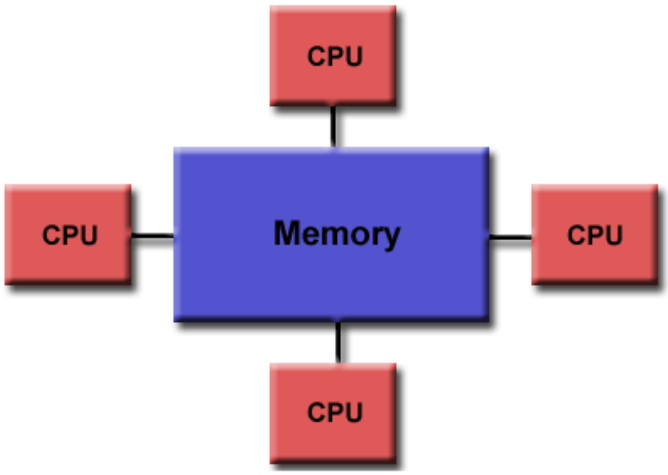


Figura 3. Modelo de memoria compartida

Una forma natural de programar en este tipo de arquitectura es con un modelo basado en threads en el que todos tienen acceso a variables compartidas. Estas variables serán usadas como medio de intercambio de información entre threads. Para coordinar el acceso a las variables compartidas se utilizan mecanismos de sincronización con el objetivo de evitar condiciones de carrera en caso de accesos concurrentes.

Los paradigmas de programación en memoria compartida pueden variar de unos a otros con respecto a los modelos de concurrencia, las variables compartidas y el soporte para sincronización.

2.4 Hilos

Un hilo es un flujo de instrucciones dentro del programa que pueden ser ejecutadas. El modelo de programación basado en hilos ofrece ventajas importantes sobre el modelo inspirado en el paso de mensajes, pero como en todo también existen desventajas. Aquí resumiré algunas de las características más importantes [12].

- Portabilidad de software: las aplicaciones basadas en threads pueden ser desarrolladas en máquinas secuenciales y ser ejecutadas en máquinas paralelas sin necesidad de hacer ningún cambio. La habilidad para migrar programas entre diversas plataformas es una ventaja muy importante en favor de las API's basadas en threads, debido a que la capacidad de acceso a un supercomputador puede ser limitado y costoso, y no se puede estar malgastando en tiempo de desarrollo.

- Ocultación de latencia: Uno de los mayores costes que tienen los programas (tantos secuenciales como paralelos) es la latencia de acceso, tanto a memoria como a dispositivos I/O. Puesto que se ejecutan múltiples hilos en el mismo procesador, las aplicaciones basadas en hilos permiten ocultar esta latencia. De hecho, mientras un hilo está esperando a que termine una operación de comunicación, los otros hilos pueden utilizar la CPU, así ocultando la espera.

- Planificación y balanceo de carga: Cuando se están escribiendo programas usando memoria compartida, el programador debe expresar la concurrencia de manera que el coste

de sincronización sea mínimo y que los procesadores se mantengan ocupados la mayoría del tiempo.

En muchas aplicaciones con datos estructurados la tarea de asignar la misma cantidad de trabajo a los procesos es sencilla, pero en aplicaciones con datos sin estructura y de carácter dinámico esta tarea resulta mucho más complicada.

Las APIs basadas en hilos permiten al programador especificar un gran número de tareas concurrentes y también soportan la asignación dinámica de tareas a procesadores de forma que se minimiza el tiempo que un core pasa desocupado. Esta capa de abstracción a nivel de sistema evita que el programador tenga que preocuparse de forma explícita de una planificación y un balanceo de carga.

- Facilidad de programación: Como su uso es generalizado debido a las ventajas mencionadas anteriormente, los programas basados en hilos son significativamente más fáciles de programar que los que utilizan un modelo de paso de mensaje.

2.5 OPENMP

En el apartado anterior, hemos resumido el uso de APIs basadas en hilos para programar maquinas de memoria compartida. Aunque estas APIs (POSIX threads) estén estandarizadas y tengan un gran soporte en la comunidad, su uso está prácticamente restringido a programadores de sistemas en vez programador de aplicaciones de alto nivel.

Un gran número de aplicaciones pueden ser implementadas mediante directivas de alto nivel, lo cual evita que el programador tenga que manipular threads a mano.

OpenMP es una API basada en directivas que puede ser usada con FORTRAN, C, C++ para crear programas que usen memoria compartida. La principal ventaja de OpenMP es su facilidad de uso ya que permite crear concurrencia, sincronización, y tratar los datos sin necesidad de tener que preocuparnos por mutexes, variables de condición y otros aspectos de bajo nivel.

2.6 Paralelismo de tareas

En un modelo de programación basado el paralelismo de tareas, el programador debe definir y manipular tareas concurrentes. Los problemas se descomponen en tareas que pueden ser ejecutadas de forma concurrente, y entonces asignarlas a threads para que se ejecuten en paralelo.

Esto resulta muy sencillo cuando las tareas no dependen entre ellas, pero en este modelo de programación también se usa cuando las tareas comparten datos entre sí.

Existen dos partes a las que se debe prestar total atención: coherencia de datos y balanceo de carga.

Una categoría especialmente importante del paralelismo de tareas iría dedicado a mantener las dependencias de datos: Read after Write (RaW), Write after Read (WaR) and Write after Write (WaW).

Debido a que la ejecución de un conjunto de tareas termina cuando la última tarea ha terminado se debe tener en cuenta que las tareas pueden tener requisitos significativamente diferentes, por lo que su distribución para que todas terminen en al mismo tiempo es una tarea difícil. Este es el problema del balanceo de carga.

2.7 Grafos de dependencias

En informática, se conoce a un grafo de dependencias como un grafo dirigido que representa las relaciones de los nodos entre ellos.

Un grafo dirigido se define como un par de conjuntos G = (V, E) :

- Un conjunto V, de elementos llamados vértices o nodos.
- Un conjunto E, de pares de vértices ordenados, llamadas aristas.

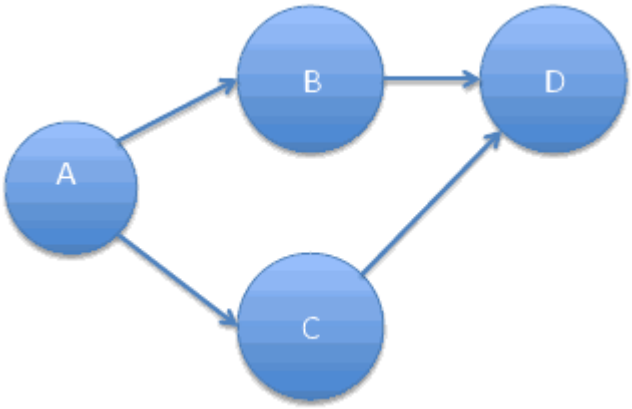


Figura 4. Grafo dirigido

En este ejemplo: $V=\{A,B,C,D\}$ y $E=\{\{A,C\},\{A,B\},\{B,D\}\{C,D\}\}$

Una dependencia de datos es una situación en la que los datos necesarios para una instrucción dependen de datos creados por otras instrucciones.

Existen tres tipos de dependencia de datos entre las instrucciones:

- Dependencia de flujo
- Antidependencia
- dependencia de salida

La dependencia de flujo ocurre cuando un dato producido por una instrucción se necesita en instrucciones futuras. Normalmente se conoce como la dependencia RAW (read-after-write) porque se lee el dato después de escribirlo.

```
1. R3 = R2 + R1
2. R4= R3 - 1
```

Existe dependencia de datos entre las dos instrucciones por el uso de R3.
La antidependencia ocurre cuando una instrucción escribe sobre una variable que acabe de ser leída por la instrucción anterior. Esta se conoce como la dependencia WAR (write-after-read).

```
1. R3 = R2 + R1
2. R2 = R5 -1
```

La instrucción 2 no debe guardar su resultado en R2 antes que la instrucción 1 lea R2, de lo contrario la instrucción 1 usaría un valor producido por la instrucción 2 en vez del valor previo de R2.

La dependencia de salida ocurre cuando una variable se escribe por dos instrucciones. Esta también es conocida como la dependencia WAW (write-after-write).

```
1. R3 = R2 + R1
2. R2 = R3 - 1
3. R3 = R2 + R5
```

La instrucción 1 debe guardar el resultado en R3 antes que la instrucción 3 guarde resultado en R3, de lo contrario la instrucción 2 podría usar el valor erróneo de R3.

Esta dependencia tiene una solución muy sencilla, usar dos variables diferentes.

3 Paralelismo de tareas

Si no consideramos algoritmos especiales, si se desea alcanzar un buen rendimiento, se consideran necesarios tres puntos principalmente.

Granularidad fina para alcanzar un gran nivel de paralelismo, ejecución asíncrona que evite barreras de sincronización, y un planificador que se base en el flujo de datos para garantizar la ejecución de las tareas en el momento en que sus datos de entrada se encuentren disponibles.

Existen frameworks especializados que soportan paralelismo de tareas, Cilk, Threading Building Blocks (TBB) de Intel, OpenMP-3.0.

En todos estos productos la creación de tares es una operación no-bloqueante, el programa crea la tarea y continúa ejecutando el programa. Además, la sintaxis sigue siendo secuencial. Pero de nuevo se plantea el problema de cómo solucionar la sincronización entre partes del código y su balanceo de carga.

Con respecto al balanceo de carga, todos los frameworks se basan en algún tipo de algoritmo *work-stealing*, basado en coger las tareas pendientes de la cola/pila de los hilos.

Pero cabe destacar que existen librerías, como Quark, X-KAAPI y StarPU que son capaces de generar una secuencia de tareas que acceden a memoria compartida. Y a partir de esta secuencia, son capaces de extraer tareas independientes y mandarlas a ejecutar a cores inactivos.

3.1 Directiva task de OpenMP

3.1.1 Origen de las tareas en OpenMP

La versión de OpenMP 2.5 está basada en hilos y utiliza un modelo de ejecución basado en fork-join para ejecutar todos los hilos que tendrán acceso al mismo espacio de memoria [3].

La directiva *parallel* se usa para crear un equipo de hilos. Las directivas de reparto de trabajo como *for, sections*, y *single* se usan para asignar trabajo entre los hilos del equipo.

Cada unidad de trabajo se asigna a un hilo en concreto del grupo, y se ejecuta desde el principio hasta el final por el mismo hilo. Un hilo no puede suspenderse y parar de ejecutar una unidad de trabajo para ponerse a ejecutar otro trabajo distinto.

Con la versión de OpenMP 3.0 se cambió el enfoque hacía las tareas. La versión 3.0 introduce la directiva *task*, que permite especificar de forma explícita al programador una unidad de trabajo que puede ser ejecutada en paralelo.

Las tareas explícitas son útiles cuando se trata de expresar paralelismo en unidades de trabajo generadas dinámicamente, tareas que deben ser añadidas y ejecutadas en un grupo de hilos en particular.

La tarea será ejecutada por uno de los hilos del equipo, pero diferentes partes de la tarea puede que sean ejecutadas por diferentes hilos del mismo equipo, a diferencia de lo que ocurría en versiones anteriores de OpenMP.

3.1.2 La directiva task

La sintaxis de la directiva *task*, se demuestra con el siguiente ejemplo:

```
1.  #include <stdio.h>
2.  #include <omp.h>
3.
4.  int fib(int n)
5.  {
6.    int i, j;
7.    if (n<2)
8.      return n;
9.    else
10.     {
11.       #pragma omp task shared(i) firstprivate(n)
12.       i=fib(n-1);
13.
14.       #pragma omp task shared(j) firstprivate(n)
15.       j=fib(n-2);
16.
17.       #pragma omp taskwait
18.       return i+j;
19.     }
20. }
21. int main()
22. {
23.   int n = 10;
24.
25.   omp_set_dynamic(0);
```

```
26.    omp_set_num_threads(4);
27.
28.    #pragma omp parallel shared(n)
29.    {
30.      #pragma omp single
31.      printf ("fib(%d) = %d\n", n, fib(n));
32.    }
33.  }
```

Cuando un hilo se encuentra con la directiva *task*, se crea una nueva tarea, es decir, se genera instancia específica con su código y datos de su entorno. Una tarea puede ser ejecutada por cualquier hilo del equipo, de forma paralela respecto a otras tareas, su ejecución puede ser inmediata o se puede retrasar. La tarea que se está ejecutando en ese momento en el un hilo se conoce como tarea actual.

Se crea un nuevo almacenamiento para cada variable *private y firstprivate*, y todas las referencias a la variable se reemplazan por referencias a las nuevas variables, dentro del ámbito de la tarea. Las variable *firstprivate* mantienen el valor original antes de entrar en la región de la tarea, las privadas no.

Las variables compartidas no listadas junto a la directiva *task* no comparten las reglas normales de OpenMP, se detectan de la siguiente manera:

Si una directiva de *task* se encuentra dentro de una región paralela, y una variable era *shared* en la región paralela, lo continúa siendo. El resto de variables se convierten en *firstprivate*.

Las tareas permiten la creación de tareas anidadas. Por lo tanto se debe especificar que una tarea es hija de la tarea que la creo.

El ámbito de la tareas hija es diferente de la del padre, lo que nos da la oportunidad de compartir una variable que era *private* en la tarea padre con sus tareas hijas.

En esto caso, la tarea hija se ejecuta de forma concurrente con la tarea generadora, el programador tiene la responsabilidad de añadir la sincronización necesaria para evitar condiciones de carrera, o permitir que la variable deje de existir antes que la hija termine.

La directiva *taskwait* se usa para sincronizar la ejecución de tareas de forma mucho más fina, por ejemplo en la figura, aquí se mantiene el recorrido en post-orden del árbol, y se evita que las variables compartidas dejen de existir.

```
1.  int traverse(struct node *p, int postorder) {
2.    int l,r;
3.    l = r = 0;
4.    if (p->left){
5.      #pragma omp task shared(l)
6.        traverse(p->left, postorder);
7.    }
8.    if (p->right){
9.      #pragma omp task shared(r)
10.       traverse(p->right, postorder);
11.   }
12.   #pragma omp taskwait
13.   return l + r+ proceso(p)
14. }
```

La directiva *taskwait* suspende la ejecución de las tarea, hasta que todas las tareas hijas de la tarea actual hayan terminado. Sólo se esperan a las tareas hijas inmediatas, a sus descendientes.

Por lo tanto es responsabilidad del usuario insertar puntos de sincronización para las tareas cuando sea necesario, para evitar que las variables se liberen antes que la tarea haya terminado de usarlas.

3.1.2.1 Ejecución de tareas

Cuando un hilo del equipo actual empieza a ejecutar una tarea, los dos se “enlazan” mutuamente, es decir, el mismo thread ejecutará esa tarea desde el principio hasta el final.

Esto no quiere decir que la ejecución vaya a ser continua, un hilo puede suspenderse en un punto de sincronización mediante la directiva *taskwait*, pero puede reanudarse mas tarde, cuando las tareas usan la clausula *tied* (por defecto), la planificación de las tareas puede ocurrir en varios puntos.

En estos puntos la información de una variable del *privatethread* o información del hilo, como su número(en caso de que la tarea fuera *untied*) puede cambiar.

Cuando se encuentran la directiva *taskwait*, en un *barrier*, o cuando termina la tarea. En el momento en el que un hilo se suspende, se realiza un intercambio de tareas, es decir, se reanuda la ejecución de una tarea que había sido previamente suspendida, o se empieza a ejecutar una tarea completamente nueva.

Existe la posibilidad de que la tarea generadora de tareas quede suspendida, para que se ejecuten sus tareas hijas, y así dejar espacio para que se creen nuevas tareas, pero esto puede llevar al problema de que no esté bien balanceado el trabajo.

Supongamos que se suspende la tarea padre, y se empieza a ejecutar una de las tareas hijas en ese hilo, y resulta que esta tarea hija es muy costosa, mientras tanto el resto de hilos han terminado(y se encuentran desocupados) y están esperando a nuevas tareas, pero la tarea padre que es la encargada de generar las tareas está suspendida. Hasta que no termine la tarea hija o llegue a un punto de sincronización no se podrían crear nuevas tareas.

Una forma de hacer que la carga este distribuida de forma más o menos similar es que la tarea generadora esté declarada como *untied*.

3.2 Librerías basadas en grafos

3.2.1 QUARK

Esta librería es parte del proyecto PLASMA, pero se considera a efectos prácticos un proyecto diferente. Su finalidad crear a una ejecución paralela a partir de una ejecución secuencial basada en bucles [6].

Esta basada en el paralelismo sobre tareas, ofrece un planificador dinámico capaz de ejecutar las tareas en orden diferente al que se crearon, siempre y cuando se respeten las dependencias de datos entre las tareas.
Ahora describiré de forma breve las técnicas de planificación usadas por QUARK.

Para que un planificador sea capaz de resolver las dependencias entre las tareas, este necesita saber cómo se están usando los datos de cada tarea.

A continuación muestro un ejemplo reducido de una multiplicación de matrices:

```
1.  //LLAMADA QUE EJECUTA LAS TAREAS
2.  void matmul_quark_task( Quark *quark )
3.  {
4.     double *A, *B, *C;
5.     int NB;
6.     quark_unpack_args_4( quark, A, B, C, NB );
7.     GEMM( A, B, C, NB );
8.  }
9.
10. //LLAMADA QUE INSERTAR LAS TAREAS
11. void matmul_quark_call( Quark *quark, double *A, double *B, double *C, int NB )
12. {
13.    QUARK_Insert_Task( quark, matmul_quark_task, NULL,
14.                       sizeof(double)*NB*NB, A, INPUT,
15.                       sizeof(double)*NB*NB, B, INPUT,
16.                       sizeof(double)*NB*NB, C, INOUT,
17.                       sizeof(int), &NB, VALUE,
18.                       0 );
19. }
20.
21. void main(){
22.   int THREADS=4;
23.   Quark *quark = QUARK_New(THREADS);
24.   for (i = 0; i < BB; i++)
25.     for (j = 0; j < BB; j++)
26.       for (k = 0; k < BB; k++)
27.             matmul_quark_call( quark, &Ablk[NB*NB*i + NB*NB*BB*k],
          &Bblk[NB*NB*k + NB*NB*BB*j],
                          &C_quark_blk[NB*NB*i + NB*NB*BB*j],                                    NB);
28.   QUARK_Delete(quark);
29. }
```

La llamada a *QUARK_Insert_Task* es la encargada de comenzar a crear las tareas, cuando estas tareas tengan sus datos disponibles, la librería ejecutará la función dada en este caso *matmul_quark_task,* donde se extraen los parámetros y se ejecuta la función de calculo.

Ahora pasaremos a analizar los posibles valores de la llamada *QUARK_Insert_Task [16]:*

Los argumentos de una función pueden ser *VALUE*, una copia del valor en la tarea, o pueden ser del tipo *INPUT, OUTPUT* o *INOUT,* estos argumentos referencian a una variable en memoria, que cumple que actúa como entrada/salida/entrada-salida de datos dependiendo del nombre para esa función.

También se pueden añadir *flags*, con la intención de aumentar la localidad de los datos y el reuso de la cache.

El planificador esta diseñado para empezar a partir del código secuencial (C, C++, Fortran), en el que se hacen llamadas a las tareas de cálculo. Conforme progresa la ejecución secuencial, las tareas se van añadiendo (mediante *QUARK_Insert_Task*) en orden al planificador, y entonces mediante los argumentos, el planificador puede deducir las dependencias entre las tareas [15].

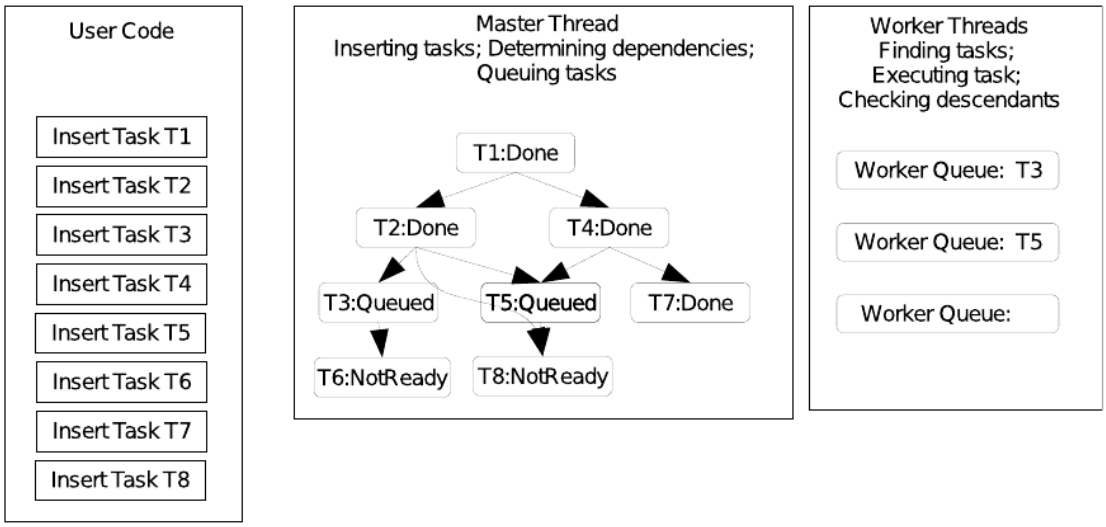


Figura 5. Visión idealizadas de Quark

El hilo que se encarga de la inserción de tareas se le conoce como el hilo maestro, bajo ciertas circunstancias el hilo maestro también ejecutará tareas de calculo.

La ejecución de las tareas disponibles la hace un conjunto de hilos esclavos, que simplemente esperan a que haya tareas disponibles y las ejecutan usando una combinación de asignación de tareas por defecto y *work-stealing*.

Las dependencias entre las tareas crean un DAG implícito, sin embargo, este DAG nunca se crea de forma explicita en el planificador. Por lo contrario se mantiene una estructura de manera que las tareas se meten en una cola basada en los datos que necesitan para continuar, esperando para el acceso adecuado a esos datos.

Las tareas se insertan en el planificador, que las almacena hasta el momento de ser ejecutadas, que será cuando se hayan cumplido todas las dependencias. Es decir, una tarea esta disponible para ser ejecutada cuando todas sus tareas padre han sido completadas.

La intención es facilitar a los diseñadores de algoritmos la experimentación con algoritmos y el diseño de nuevos algoritmos.

Si desplegásemos y mantuviéramos un DAG de tareas completo de un problema de gran tamaño en memoria, seríamos capaces de hacer un análisis detallado respecto a la planificación del DAG y al camino critico.

Sin embargo, el tamaño de las estructuras de datos crecería tanto de manera tan rápida que sería imposible de manejar. La solución propuesta es mantener un DAG sobre conjunto limitado de tareas.

Se recorre el DAG limitado compuesto por un conjunto de tareas (de tamaño variable), este conjunto debería ser lo suficientemente grande para garantizar que todos los *cores* se mantienen ocupados. Cuando se alcanza el tamaño máximo de tareas, el *core* encargado de insertar las tareas no acepta ninguna tarea más hasta que no se complete alguna tarea.

El uso de un rango de tareas tiene implicaciones en la manera en la que se despliegan los bucles de la aplicación y cuantos trozos del grafo tiene disponible el planificador, siendo incapaz de ver posible paralelismo en niveles inferiores del árbol si su DAG esta completo.

Se ha demostrado en el pasado que la reutilización de las memorias cache puede llevar a una mejora substancial de rendimiento en el tiempo de ejecución. Debido a que trabajamos con estructuras de datos que deberían caber en la memoria cache local de cada *core*, existe una opción que nos da la habilidad para indicarle al planificador el funcionamiento de la localidad de la cache.

Un parámetro en la llamada puede ser decorado con el flag de *LOCALITY* para decirle al planificador que el dato (ese parámetro) debería mantenerse en cache si es posible.

Después de que un *core* ejecuta esa tarea, el planificador asignará por defecto cualquier tarea que use ese dato en el futuro al mismo *core*. Se debe tener en cuenta que *work-stealing* puede interferir con la asignación por defecto de tareas a *cores*.

3.2.2 XKAAPI

La mayoría de frameworks(StarPU, Quark), que también utilizan grafos de flujo de datos como método para representar y planificar de forma dinámica las tareas cumpliendo sus dependencias, no permiten la creación recursiva de tareas, ni mezclar bucles de tareas con bucles paralelos, aunque estos últimos sean importantes para aplicaciones científicas [7].

X-KAAPI, es una librería que permite usar un grafo de flujo de datos y bucles paralelos. A partir de código secuencial, este se anota con funciones identidad que serán transformadas en tareas, y que se ejecuten de forma paralela [10].

Cada función que se vaya a convertir en una tarea, debe ser anotada para especificar el modo de acceso (*read, write, reduction, exclusive*) que hacen sus parámetros a la memoria.

El compilador insertará las tareas y el programa detectará las dependencias y planificará las tareas a los *cores*. Los bucles paralelos se procesan de la misma manera, el usuario anota un bucle para que sea paralelizado, tal y como se hace en *OpenMP*, y el programa en tiempo de ejecución crea las tareas y los asigna a *cores* disponibles.

Para compilar las directivas es necesario tener instalado el entorno *ROSE*, pero también se pueden utilizar las API, de *C/C++/Fortran* y si no se desea trabajar con directivas.

Además cabe destacar que para mantener la portabilidad de los programas *XKAAPI* ha implementado la API de *Quark*, por lo que aplicaciones que estén programadas para ser usadas con Quark, pueden utilizar XKAAPI de forma transparente al programador, tan sólo debe tenerse en cuenta que se enlace con la librería de *Quark*, creada en la carpeta de *XKAAPI*.

3.2.2.1 Región paralela

X-KAAPI define el concepto de región paralela. Una región paralela es un ámbito dinámico donde varios threads cooperan para ejecutar las tareas creadas. Aunque el termino sea similar a la región paralela de *OpenMP*, la región paralela de *XKAAPI* no limita el número de hilos que pueden ejecutar tareas de forma concurrente.

Una región paralela es anotada por la directiva *#pragma kaapi parallel*. La directiva debe preceder a una instrucción o a un bloque de instrucciones. Al final de una región paralela existe un punto de sincronización que espera a que terminen todas las tareas previamente creadas.

Las regiones paralelas pueden incluir otras regiones paralelas. Las regiones anidadas comparten los mismos recursos computacionales. Fuera de la región más externa, solo el hilo maestro ejecuta el programa. El *X-KAAPI* selecciona el número de hilos dependiendo de variables de entorno o del número de *cores* disponibles.

3.2.2.2 Tareas

3.2.2.2.1 Creación de tareas

Una tarea es una llamada a una función que no devuelve nada, excepto a través las variables de memoria compartida o la lista de sus parámetros.

De la resolución de dependencias entre tareas y los intercambios de datos se encarga la librería. La única responsabilidad que recae en el programador es la de anotar el código, por lo tanto la granularidad de las tareas sigue siendo trabajo del programador.

La creación de la tarea ocurre mediante la directiva *#pragma kaapi task*, o mediante una llamada a la API, por ejemplo, `kaapi_spawn()`.

El usuario tiene la responsabilidad de indicar el modo de acceso de las tareas a memoria. *READ, VALUE, WRITE*.

Dada la declaración, dependiendo de cada parámetro y su modo de acceso, *XKAAPI* es capaz de analizar si dos tareas tienen dependencia de flujo de datos en alguna región de memoria.

La creación de tareas en una operación no bloqueante, el receptor no se espera a que la tarea se termine. Esto implica que una variable escrita por una tarea se puede leer, ya sea pasando la variable a una tarea que la anote como lectura, o después de un punto de sincronización, usando la directiva *#pragma kaapi sync*, se sincronizan los datos de la tareas.

Como ejemplo, se muestra a continuación el algoritmo mergesort, creado mediante tareas.

```
30.  void mergesort(int n,double *v) {
31.    int m, r;
32.    if( n>1 ) {
33.      m = n/2; r = n%2;
34.     kaapic_spawn( 2 , mergesort ,
35.         KAAPIC_MODE_V, m+r, 1 , KAAPIC_TYPE_INT,
36.         KAAPIC_MODE_W, v , 1 , KAAPIC_TYPE_PTR
37.     );
38.      kaapic_spawn( 2 , mergesort ,
39.         KAAPIC_MODE_V, m, 1 , KAAPIC_TYPE_INT,
40.         KAAPIC_MODE_W, &v[m+r] , 1 , KAAPIC_TYPE_PTR
41.      );
42.      kaapic_sync();
43.      merge( n, v );
44.    }
45.  }
```

3.2.2.3 Bucles paralelos

Como en *OpenMP*, en *X-KAAPI* es posible anotar un bucle paralelo, en el cual todas sus iteraciones son independientes. La anotación es *#pragma kaapi loop.*

Esta directiva de bucle no existen en otros software basados en control de flujo de datos (*StarPU, Quark*). En *X-KAAPI*, un bucle paralelo es una tarea que puede ser partida en tiempo de ejecución para generar paralelismo, se debe especificar el modo de acceso para las variables que se quieran acceder.

El bucle paralelo es una operación no bloqueante. Por defecto, el software añade una sincronización implícita al final del bucle paralelo. El usuario puede de forma explicita especificar la clausula *nowait* para evitar la sincronización.

3.2.2.4 Ejecución

Cuando se inicializa un programa *XKAAPI* empieza, se crean un número fijo de hilos, y el hilo principal continúa ejecutando el programa. Los otros hilos se llaman “hilos trabajadores”. A partir de las anotaciones o mediante llamadas a la API, el compilador de XKAAPI inserta el código para crear tareas. En tiempo de ejecución, cada tarea creada, se guarda dentro de una pila de tareas su hilo.

Las tareas que estén listas para ser ejecutadas en una pila, pueden cogerse y ejecutarse en otros hilos que estuvieran desocupados.

Por defecto, las dependencias se resuelven de forma “vaga” por un hilo desocupado, cuando este intenta coger una de las tareas listas de una de las pilas de otro hilo que se encuentra ocupado. Este es el principio de *work-stealing*.

En conclusión si comparamos otras librerías con *X-KAAPI*, se pueden apreciar las ventajas de usar este software. Por ejemplo, *OpenMP* no tiene dependencias entre datos y *QUARK/StarPU* no aceptan la creación recursiva de tareas, ni tienen directivas de creación de bucles paralelos. *X-KAAPI* permite crear tareas recursivas con dependencias de datos, además permite mezclar dependencias de datos entre tareas y bucles paralelos.

3.2.3 STARPU

En el campo de la supercomputación, la tendencia actual es diseñar arquitecturas con multiprocesadores con tecnologías heterogéneas como por ejemplo aceleradores de datos paralelos (es decir, *GPUs*) [14].

Si se intenta alcanzar el máximo rendimiento teórico con arquitecturas diferentes se verá que es un problema complicado. De hecho, los principales esfuerzos se han dedicados a descargar partes del calculo. Sin embargo, diseñar un modelo de ejecución que unifique todas las unidades de cálculo y sus respectivas memorias es un reto muy importante.

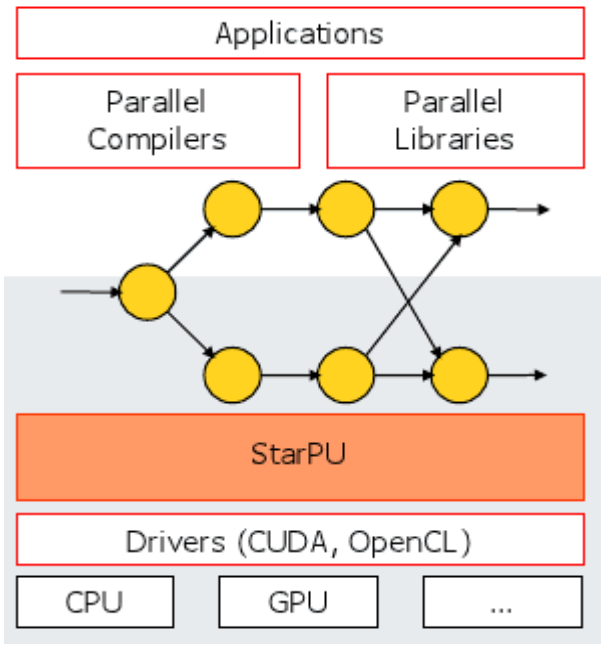


Figura 6. Modelo StarPU

El objetivo de *StarPU*, es permitir a los programadores aprovechar todo el poder de cálculo disponible en las *CPUs* y *GPUs* sin que tengan que adaptar su programas especialmente para determinas máquinas o unidades de procesamiento.

Con este propósito *StarPU* suministra una interfaz y un modelo de programación unificado sobre el cual los programadores pueden crear sus algoritmos sin tener que directamente usar librería de bajo nivel para enviar datos a *GPU's*.

La librería *StarPU* y sus extensiones dan soporte a un modelo de programación basado en tareas. La aplicación secuencial crea las tareas y si se ha implementado previamente las funciones para una arquitectura determinada, *StarPU* planificará estas tareas y se encargará de los intercambios de datos entre las *CPUs* o *GPUs* disponibles, y su posterior ejecución en las diferentes arquitecturas.

Los datos que una tarea manipula se transfieren automáticamente entre aceleradores y memoria central, para que los programadores no tengan que preocuparse de problemas de planificación o detalles técnicos asociados con la transferencia de datos.

Las ejecuciones de tareas, se basa en un par de estructuras de datos, el *codelet* y la estructura de tarea *task*.

El codelet describe un kernel computacional que puede ser implementado mediante múltiples arquitecturas tales como CPU o GPU.

Otra estructura de datos importante es la tarea, que se rellenará con los datos necesarios para su ejecución. Para ejecutar una tarea StarPU se debe aplicar un codelet a un conjunto de datos, en una de las arquitecturas que el codelet haya implementado.

Una tarea por tanto necesita la siguiente información el codelet que va usar, los datos a los que va a acceder, como deben de ser accedidos estos datos y las dependencias de datos que tienen los parámetros (lectura y/o escritura).

Las tareas StarPU son asíncronas, crear una tarea es una operación no bloqueante. Además, en la estructura *task* se puede especificar una serie de parámetros extra, como, por ejemplo, llamadas callback, o especificar el tipo de política de scheduling que se debe aplicar a esta tarea.

Como ejemplo creamos una tarea de “Hola mundo”.

La definición de estructuras necesarias sería

```
1.  struct params{
2.          int i;
3.          float f;
4.  };
5.  void cpu_func(void *buffers[], void *cl_arg){
6.      struct params *params = (struct params *) cl_arg;
7.     FPRINTF(stdout, "Hello world (params = {%i, %f} )\n", params->i, params->f);
8.  }
9.  struct starpu_codelet cl = {};
```

Y la creación de la tarea, comenzaría con el siguiente código.

```
1.          ret = starpu_init(NULL);
2.
3.          task = starpu_task_create();
4.          cl.where = STARPU_CPU;
5.          cl.cpu_funcs[0] = cpu_func;
6.          task->cl = &cl;
7.          task->cl_arg = &params;
8.          task->cl_arg_size = sizeof(params);
9.          task->synchronous = 1;  //0 asíncrono
10.         ret = starpu_task_submit(task);
11.
12.         starpu_shutdown();
```

3.2.4 DAGUE

En los últimos años el número de *CPUs* y el uso de aceleradores ha aumentado, y se espera que esta tendencia continúe, con proveedores anunciando chips con 80 *cores*, *multiGPUs* y una mayor integración de aceleradores y procesadores.

Desde el puro punto de vista del rendimiento se intenta extraer el mejor resultado mediante un modelo de programación híbrido MPI/threads.

Con procesos MPI funcionando sobre los nodos y múltiples threads dentro de cada nodo. Pero la programación híbrida es compleja y es fácil cometer errores. Lo que a menudo lleva a que el programador deje de concentrarse en resolver el problema y se centre en resolver la distribución de datos y el balanceo de carga, y cómo hacer que la comunicación y la computación ocurran al mismo tiempo, para evitarse los costes temporales que supone enviar datos.
Con *DAGuE* se pretende tener una herramienta de memoria distribuida que funcione como un compilador, que automáticamente analice el código secuencial en C, y genere un grafo de flujo de datos simbólico e independiente del tamaño del problema [1].

A través de análisis poliédrico, el compilador convierte el grafo creado por código de entrada como expresiones simbólicas parametrizadas. Estas expresiones permiten a cada tarea, ejecutarse de forma independiente en tiempo de ejecución.

Ahora pasaré a explicar las herramientas usadas para hacer esta traducción.

La librería *DAGuE* incluye un entorno formado un compilador, un planificador dinámico, y un motor de comunicaciones, y un motor de dependencias de datos.

La representación de un DAG puede ser muy grande, por lo que *StarPU* y *Quark* restringían la exploración del DAG un rango limitado, y en consecuencia, no pueden hacer un uso de un posible paralelismo con alguna de las siguientes tareas, lo que produce secuencialidad en la ejecución.

Esto no presenta un problema para *DAGuE*, ya que el motor no explora un DAG, sino que en cambio usa una representación simbólica, como muestra la figura de abajo, para explorar los sucesores y generar tareas y planificarlas [8].

```
1   DGEQRT(k) executes on A(k, k)
2    k = 0..SIZE-1
3
4    V <- (k==0)       ? A(0,0) : C2 DSSMQR(k-1,k,k)
5      -> (k==SIZE-1) ? A(k,k) : R DTSQRT(k,k+1)    [U]
6      -> (k!=SIZE-1) ? V DORMQR(k, k+1..SIZE-1)    [L]
7      -> A(k,k)                                    [L]
8    T -> T DORMQR(k, k+1..SIZE-1)                  [T]
9      -> T(k,k)                                    [T]
```

Figura 7. JDF

Por lo tanto el DAG nunca se completa en memoria, sólo existen colas de tareas listas para ejecutarse en los sistemas locales, es más, cada nodo resuelve sólo las tareas sucesoras a las que se hayan ejecutado localmente. Si un hilo, no tiene suficientes tareas en su cola, recurrirá a quitar tareas a otros hilos.

Este conocimiento es suficiente para implementar un planificador, basado sólo en la información local. Cuando un nodo descubre que una tarea remota ha terminado, el nodo ahora puede resolver ciertas tareas locales que acaban de ponerse en estado disponibles al haber terminado su dependencia.

El formato de entrada para DAGuE puede ser un pseudocódigo secuencial basado en bucles anidados o su representación en JDF. Con la herramienta H2J podemos convertir pseudocódigo al formato JDF.

3.2.4.1 JDF

El formato JDF es una representación textual del algoritmo, cada tarea dividida en dos partes: la representación del flujo de datos, y un cuerpo de código. El cuerpo del código es la traducción de la llamada en C, y puede ser personalizada por el programador [2].

Las dependencias DAG se presentan como condiciones algebraicas en las entradas y las salidas de cada tarea.

En la figura 7, si el parámetro $k=0$, el valor de V se lee de la matriz local A, de lo contrario viene de la salida de $C2$ en la función *DSSMQR*. El programador puede alterar un JDF en caso de querer afinar la aplicación para sacar mejor rendimiento.

Una vez tenemos la representación JDF, la podemos pre-compilar como un código en C, y se nos devuelve un fichero.

El precompilador genera un función C visible para el usuario, que toma todas las variables globales como argumentos, además se incluye el código insertado por el usuario como cuerpo. Se añaden también una serie de *stubs* para ejecutar los datos en local y para descubrir todas las tareas sucesoras.

DAGuE también es el responsable de mover los datos entre procesadores. La planificación está totalmente distribuida, todos los nodos tienen un planificador, cada proceso usa el JDF para identificar las tareas de las que pueda necesitar información, sin necesidad de un planificador centralizado.

DAGuE crea un hilo y se lo asigna a un *core*. Cada hilo alterna entre ejecutar funciones computacionales y la ejecución de *stubs*, que son llamadas al planificador.

Cada hilo mantiene una lista de tareas listas para ser ejecutadas, y cuando un hilo termina su tarea y este ejecuta un *stub,* extraído del JDF, que determina que tareas pueden haber sido activadas. Iterando en las dependencias de salida de la tarea, el hilo marca cuales son las dependencias de entrada que han sido activadas.

Para mejorar la localidad de datos en arquitecturas *NUMA*, se utiliza *HWLOC (*API que nos da información del procesador, de la memoria, topología del ordenador*)*, y con la información del JDF, si una tarea requiere datos se intentará ejecutar las tareas en el mismo core, o en el *core* más cercano.

3.2.4.2 Comunicaciones asíncronas

En *DAGuE*, las comunicaciones surgen a partir de una dependencia de datos entre tareas. El JDF no tienen información sobre el particionado de los datos, sólo tiene información respecto la asignación de tareas.

La tarea de distribución de datos recae sobre las capas superiores, la asignación de datos cae en cuenta del programador, pero asignaciones típicas como 1D o 2D bloque cíclicas se encuentran disponibles en el *framework*.

El *runtime* asigna tareas a *cores* dependiendo del particionamiento de datos, las tareas pueden cambiar de *cores*, pero no de nodos. Para alcanzar el máximo solapamiento posible entre comunicación y cálculo, las comunicaciones se ejecutan en un nodo separado, que recibe ordenes de los hilos que están ejecutando las tareas.

Las activaciones de tareas locales, se manejan de forma local, mientras que en los casos de dependencias remotas se debe enviar un mensaje de activación. Este mensaje contiene información respecto a los parámetros de salida, para que los planificadores locales puedan saber si existen tareas locales que ahora tengan sus dependencias de datos resueltas. Si es necesario las tareas recibirán los datos necesarios para realizar sus función.

3.2.4.3 GPU y soporte a aceleradores

Desde el punto de vista de la memoria, la GPU funciona como un recurso remoto, los datos se han de mover explícitamente dentro y fuera. Sin embargo, desde el punto de vista de la ejecución, un acelerador depende de una CPU que es la que se encarga de crear las tareas a ejecutar.

Debido a que existen tareas que se ejecutan con mejor rendimiento en una GPU que en una CPU, el *runtime* esta preparado para manejar esta heterogeneidad, ya que en el cuerpo se pueden incluir múltiples versiones de una implementación. De manera similar a *StarPU*, se sigue el modelo *codelet*. Si un JDF tiene varias implementaciones en su *BODY*, el planificador seleccionará cual de todas se adaptará mejor al hardware existente.

Cuando se invoca un planificador por medio de *stubs*, si el acelerador está inactivo y la tarea puede ejecutarse en una *GPU*, se cambia el modo de planificación, y ahora la *GPU* está al cargo de crear las tareas y recoger los datos. Un hilo mantendrá este rol hasta que todas las tareas de *GPU* hayan terminado

Cuando un planificador se encuentra en modo *GPU*, se intercalan las diferentes operaciones de forma asíncrona usando varios *streams* para permitir el solapamiento entre *I/O* y cálculo para así aumentar el rendimiento de la *GPU*. Se crean varias tareas en la GPU a la vez para alcanzar un uso más efectivo de todos los hilo/hebras GPU. Para minimizar el tráfico entre GPU y host, los resultados intermedios no se envían al host, exceptuando cuando una tarea que este en CPU los necesite, y las tareas que dependen en los datos ya cargados se ejecutarán de forma preferible en esa misma GPU.

4 Algoritmos matriciales con almacenamiento denso

4.1 Almacenamiento denso

Por matrices densas se conoce a las matrices en las que muchos de sus elementos son distintos de cero. Tradicionalmente la forma de almacenar este tipo de matrices en memoria ha sido mediante un vector unidimensional. Es decir, se guardan en memoria todos los elementos que tiene la matriz uno detrás de otro.

Ahora bien, dependiendo del algoritmo, y pensando en la jerarquía de memoria podemos deducir que si almacenamos los elementos de la matriz por bloques, los bloques podrían cargarse de forma eficaz en la memoria cache.

Con el almacenamiento por bloques se pueden evitar fallos de caché, fallos *TLB*, *false sharing* (situación que ocurre cuando dos variables diferentes se ubican en el mismo bloque de cache y entonces la memoria cache tiene sustituir bloques enteros) y a la vez maximizar la carga de los datos. Ahora *false sharing*, sólo puede ocurrir al principio o al final de un bloque.

Con la distribución estándar (*column-major*), los accesos a cada columna tienen muchas más probabilidades de causar un fallo de cache, por eso cuando se realizan operaciones con matrices densas empleando esta distribución se debe estar alerta, porque su mal uso podría llevar a una caída el rendimiento muy importante.

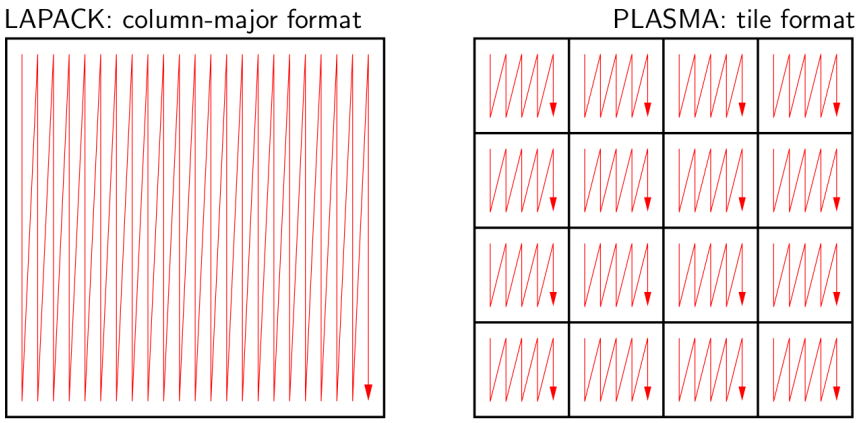
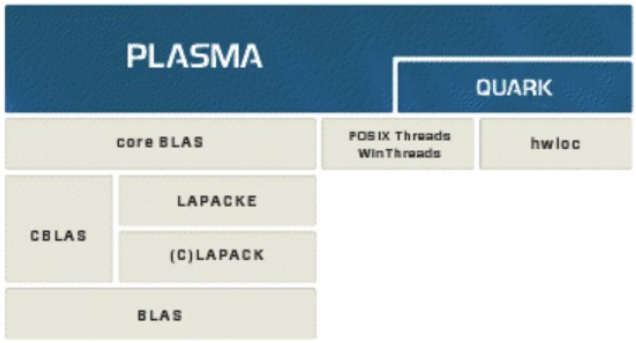


Figura 8. Comparación de almacenamiento por columnas y por bloques

4.2 PLASMA

PLASMA es una librería numérica que introduce nuevos algoritmos rediseñados para trabajar en bloques, los cuales maximizan la reutilización de los datos en varios niveles de cache de los sistemas *multicore* [11].



La distribución de la matriz se hace por bloques, los bloques se cargan en cache y se procesan completamente, antes de ser devueltos a memoria principal. Las operaciones en pequeños bloques crean un paralelismo de grano fino que suministra suficiente trabajo para mantener a un gran número de procesadores ocupados.

4.2.1 Planificación dinámica

PLASMA utiliza la planificación de tareas paralelas en tiempo de ejecución. Esta planificación se basa en la idea de asignar/ejecutar trabajo dependiendo de si los datos se encuentran disponibles.

Este concepto esta íntimamente relacionado con la idea de expresar la computación a través de un grafo de tareas, al que nos podemos referir como DAG (grafo acíclico dirigido), y también a la idea de flexibilizar el descubrimiento del DAG en tiempo de ejecución.

Esto es muy diferente de lo que ocurría en la planificación *fork-join*, donde puntos de sincronización externos delimitan las partes paralelas y secuenciales del código, causando que haya múltiples *cores* desocupados mientras se ejecuta la parte secuencial.

4.3 Funciones de matrices

4.3.1 Introducción

La necesidad de evaluar una función $f(A) \in C^{n \times n}$ de una matriz $A \in C^{n \times n}$ aparece en muchas aplicaciones desde soluciones numéricas a ecuaciones diferenciales a medidas de complejidad de redes [19].

A continuación explicaré que las funciones de matrices $f(A)$ pueden definirse de alguna de las siguientes maneras:

- Forma canónica de Jordan:

A partir de una matriz $A \in C^{n \times n}$, se puede obtener su forma canónica de Jordan de esta manera $Z^{-1}AZ = J_A = diag(J_1(\lambda_1), J_2(\lambda_2), \ldots, J_p(\lambda_p))$, donde Z no es singular,

$$J_k(\lambda_k) = \begin{bmatrix} \lambda_k & 1 & & \\ & \lambda_k & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_k \end{bmatrix} \in \mathbb{C}^{m_k \times m_k}$$

y $m_1 + m_2 + \cdots + m_p = n$. Ahora definimos lo siguiente:

$$f(A) := Z f(J_A) Z^{-1} = Z \operatorname{diag}(f(J_k(\lambda_k)))Z^{-1}$$

donde

$$f(J_k(\lambda_k)) := \begin{bmatrix} f(\lambda_k) & f'(\lambda_k) & \dots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ & f(\lambda_k) & \ddots & \vdots \\ & & \ddots & f'(\lambda_k) \\ & & & f(\lambda_k) \end{bmatrix}$$

- Interpolación polinómica:

Siendo $\lambda_1,\ldots,\lambda_s$ los distintos valores propios de $A \in C^{n\times n}$ y siendo n_i el índice de λ_i ,es decir, la dimensión de bloque de Jordan más grande asociado al valor propio λ_i . Dado $f(A):=r(A),$ donde r es el único polinomio interpolador de Hermite de grado menor que $\sum_{i=1}^{s} n_i$ donde se satisfacen las condiciones de interpolación.

$$r^{(j)}(\lambda_i) = f^{(j)}(\lambda_i), \qquad j = 0\colon n_i - 1, \quad i = 1\colon s.$$

- Convergencia en la serie de Taylor

Supongamos que f tiene una suma de series

$$f(z) = \sum_{k=0}^{\infty} a_k (z-\alpha)^k \qquad \left(a_k = \frac{f^{(k)}(\alpha)}{k!}\right)$$

con un radio de convergencia r. Para una $A \in C^{nxn}$ entonces se define $f(A)$ como:

$$f(A) = \sum_{k=0}^{\infty} a_k (A-\alpha I)^k$$

Sólo en el caso de que cada uno los distintos valores propios $\lambda_{1,}....\lambda_s$ cumplan con una de las siguientes condiciones.

(a) $|\lambda_i - \alpha| < r$,

(b) $|\lambda_i - \alpha| = r$ y la serie para $f^{(n_i-1)}(\lambda)$ (donde n_i es el índice de λ_i) converge en el punto $\lambda = \lambda_i$, $i=1\colon s.$

4.3.2 Transformaciones de similitud

El objetivo es elegir una X de manera que f se puede evaluar de forma más fácil en la matriz $B = XAX^{-1}$ que en la matriz A .

Cuando A es diagonalizable, se puede asumir que B es diagonal, entonces la evaluación de $f(B)$ es trivial.

En la mayoría de los casos de A, si la X esta restringida a ser unitaria, la máxima reducción posible de A sería a la forma Schur: $A=QTQ^*$, donde Q es unitaria y T es triangular superior.

Esta descomposición se realiza mediante el algoritmo QR. El problema se reduce ahora a evaluar f en una matriz triangular.

- Funciones para matrices triangulares

Si $T \in C^{n\times n}$ es triangular superior y f está definida en el espectro de T. Entonces $F=f(T)$ es triangular superior, y $f_{ii}=f(t_{ii})$, de donde se deduce que:

$$f_{ij} = \sum_{(s_0,\ldots,s_k)\in S_{ij}} t_{s_0,s_1} t_{s_1,s_2} \ldots t_{s_{k-1},s_k} f[\lambda_{s_0},\ldots,\lambda_{s_k}].$$

donde $\lambda_i=t_{ii}$, S_{ij} es el conjunto de todos los enteros crecientes que empiezan en i y terminan en j y donde $f[\lambda_{s0},\ldots,\lambda_{sk}]$ es la diferencia dividida del k-ésimo orden de f en $\lambda_{s0}, \ldots, \lambda_{sk}$.

Ecuación de Sylvester:
La ecuación de Sylvester es no singular precisamente cuando T_{ii} y T_{jj} no tienen ningún valor propio en común.

$$T_{ii}F_{ij} - F_{ij}T_{jj} = F_{ii}T_{ij} - T_{ij}F_{jj} + \sum_{k=i+1}^{j-1} (F_{ik}T_{kj} - T_{ik}F_{kj}), \qquad i < j.$$

El algoritmo de Schur–Parlett: Dada una $A \in C^{n\times n}$, una función f que tiene una serie de Taylor con un radio infinito de convergencia, y la habilidad de evaluar derivadas de f, este es el algoritmo que resuelve $F=f(A)$.

```
1.  Calcular la factorización de Schur A = QT Q∗ .
2.  Si T es diagonal, F = f (T ), ir a la línea 10, fin
3.  Reordenar y particionar T y actualizar Q para satisfacer las condiciones de arriba
4.  conδ = 0.1.
5.  % Ahora A = QT Q∗ es nuestra factorización de Schur reordenada
6.  % con T de tamaño de bloque  m×m .
7.  paraj = 1: m
8.     Usar algoritmo de evaluación de bloque atómico para evaluar    F_ii=f(T_ii)
9.     para i = j − 1: −1: 1
1.          Resolver    F_ij    la ecuación de Sylvester .
10.     fin
11. fin
12. F = QF Q∗
```

Este es el mejor método disponible para funciones generales f , generalmente es estable, el error está limitado por un múltiplo de $cond(f,A)u,$ pero puede ser inestable en grandes matrices .

4.3.3 Polinomios y aproximaciones racionales

Método general para aproximar $f(A)$:

```
1.  (1) Elegimos una aproximación racional adecuada r y una función de transformación g y
        hacer A ← g(A).
2.  (2) Calculo X = r(A) mediante algún esquema de aproximación.
3.  (3) Aplicar las transformaciones a X para deshacer el efecto de la transformación inicial de A
```

El paso 1, sirve para transformar A de manera que $f(A)\approx r(A)$ sea una buena aproximación.

En el paso dos, para una aproximación polinómica, si f tiene una serie de Taylor, entonces se trunca con la serie:

$$T_k(A) = \sum_{i=0}^{k} a_i A^i$$

En caso que tengamos una aproximación racional, las aproximaciones de Padé son las adecuadas. Se debe tener en cuenta que $r_{km}(x)=p_{km}(x)/q_{km}(x)$ es un aproximante de Padé $[k/m]$ de f si p_{km} y q_{km} son polinomios de grado como máximo k y m de forma respectiva, con $q_{km}(0)=1$ y

$$f(x) - r_{km}(x) = O(x^{k+m+1})$$

Es decir que un aproximante de Padé produce tantos términos como sea posible de la serie de Taylor respecto al origen. Si una aproximante de Padé $[k/m]$ existe entonces es único.

- Algoritmo exponencial:

Para matrices generales usar la aproximación de Padé resulta útil, porque ya se conocen los aproximantes de Padé $[k/m]$ de la función exponencial para todos los k y m .

$$p_{km}(x) = \sum_{j=0}^{k} \frac{(k+m-j)!\,k!}{(k+m)!\,(k-j)!} \frac{x^j}{j!}, \quad q_{km}(x) = \sum_{j=0}^{m} \frac{(k+m-j)!\,m!}{(k+m)!\,(m-j)!} \frac{(-x)^j}{j!}$$

La forma estándar de usar estas aproximaciones es con el algoritmo de “scaling and squaring”

1. para cada m = [3 5 7 9]
2. si $\|A\|_1 \leq \theta_m$, evaluar $X = r_m(A)$, fin
3. fin
4. $A \leftarrow A/2^s$ con s≥0 un entero mínimo tal que $\|A/2^s\|_1 \leqslant \Theta_{13}$
5. (es decir., s = ⌈ $\log_2$ ($\|A\|_1/\theta_{13}$)⌉).
6. Evalúa $X = r_{13}(A)$.
7. X ← X^{2^s} usando la técnica “repeated squaring.”

- Logaritmo de una matriz:

Para el logaritmo tenemos la expansión:

$$\log(I+X) = X - \frac{X^2}{2} + \frac{X^3}{3} - \frac{X^4}{4} + \cdots, \quad \rho(X) < 1.$$

Para poder usar el método general para aproximar $f(A)$, descrito anteriormente, se necesita tener una transformación inicial g para que A sea cercana a I. Esto se consigue mediante una sucesión de raíces cuadradas hasta que $A^{1/2^k}$ se acerca lo suficiente a I.

Entonces se aplica $\log(A^\alpha) = \alpha \log(A), \alpha \in [-1,1]$, para terminar teniendo una aproximación general de $2^k r(A^{1/2^k} - I)$.

- Funciones trigonométricas

Se debe usar el método general para aproximar $f(A)$, descrito arriba, escalando $A \leftarrow 2^{-s} A$ en el primer paso, y usando la formula adecuada del doble ángulo en el paso 3.

No se conoce si el aproximante de Padé r_{km} al coseno o al seno existe para todos los grados k y m, pero se pueden determinar simbólicamente para un rango a efectos prácticos.

4.3.4 Iteraciones de Matrices

Las raíces y la función signo se pueden tratar mediante iteraciones de forma $X_{k+1} = g(X_k)$, donde g no es lineal, normalmente es una función racional. La matriz inicial es casi siempre A, excepto cuando g es independiente de A.

- Método de Newton para la función signo matricial

$$X_{k+1} = \frac{1}{2}(X_k + X_k^{-1}), \qquad X_0 = A$$

siendo $X^2 = I$

- Método de Newton para la raíz cuadrada

$$X_{k+1} = \frac{1}{2}(X_k + X_k^{-1}A), \qquad X_0 = A.$$

siendo $X^2 = A$

- Método de Newton para la raíz p-ésima

$$X_{k+1} = \frac{1}{p}[(p-1)X_k + X_k^{1-p}A], \qquad X_0 = I.$$

4.4 Cálculo de cos(A) mediante aproximantes de Padé

En el apartado anterior hemos introducido el concepto de aproximaciones racionales de funciones de matrices. La función $f(x)$ será aproximada en una pequeña parte de su dominio [5]. En este caso trataremos $f(x)=\cos(x)$

$$\cos(A) = I - \frac{A^2}{2!} + \cdots + \frac{(-1)^k}{(2k)!}A^{2k} + \cdots$$

No se conoce si existen los aproximantes de Padé $[k/m]$ para $f(x)$, es decir, $R_{KM}(x)$ del $\cos(x)$ para todas las m . Puesto que el coseno es una función par, sólo tenemos que considerar los grados $2\mathrm{m}$.

Por ejemplo para calcular las dos primeras aproximantes de Padé:

$$r_2(x) = \frac{1 - \frac{5}{12}x^2}{1 + \frac{1}{12}x^2}, \qquad r_4(x) = \frac{1 - \frac{115}{252}x^2 + \frac{313}{15120}x^4}{1 + \frac{11}{252}x^2 + \frac{13}{15120}x^4}$$

Enseguida se aprecia que los numeradores y los denominadores de los coeficientes crecen rápidamente, en tamaño:

$$r_8(x) = \frac{1 - \frac{260735}{545628}x^2 + \frac{4375409}{141863280}x^4 - \frac{7696415}{13108167072}x^6 + \frac{80737373}{23594700729600}x^8}{1 + \frac{12079}{545628}x^2 + \frac{34709}{141863280}x^4 + \frac{109247}{65540835360}x^6 + \frac{11321}{1814976979200}x^8}$$

No existen una fracción continua conveniente o una fracción parcial para r_m :

Para evaluar r_m , tenemos que evaluar de forma explícita $p_{2\mathrm{m}}=\sum_{i=0}^{m} a_{2\mathrm{i}}x^{2\mathrm{i}}$ y $q_{2\mathrm{m}}=\sum_{i=0}^{m} b_{2\mathrm{i}}x^{2\mathrm{i}}$ y entonces resolver el sistema de ecuaciones múltiple :

$$q_{2\mathrm{m}}r_{2\mathrm{m}}=p_{2\mathrm{m.}}$$

Aquí podemos ver, los pasos para sacar los parámetros.

$$A_2 = A^2, \quad A_4 = A_2^2, \quad A_6 = A_2 A_4,$$
$$p_{12} = a_0 I + a_2 A_2 + a_4 A_4 + a_6 A_6 + A_6(a_8 A_2 + a_{10} A_4 + a_{12} A_6).$$
$$q_{12} = b_0 I + b_2 A_2 + b_4 A_4 + b_6 A_6 + A_6(b_8 A_2 + b_{10} A_4 + b_{12} A_6).$$

Para finalizar la siguiente figura muestra el coste de evaluar p_{2m} y q_{2m} para 2m desde grado 2 hasta 30. Midiéndose en el número de multiplicaciones p_{2m} y q_{2m} .

$2m$	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
π_{2m}	1	2	3	4	5	5	6	6	7	7	8	8	9	9	9

Figura 10. Costes para calcular el cos(A)

Ahora mostraré un ejemplo representativo de código MATLAB, para la resolución de una aproximación de Padé diagonal de grado 20.

```
1.  cosm_pade(A,m,sq, C)
2.  if sq == 1
3.     A2 = A;
4.  else
5.     A2 = A^2;
6.  end
7.  if m == 20
8.    X1 = A2; X2 = X1*X1; X3 = X2*X1; X4 = X2*X2; X5 = X4*X1;
9.     p = [1 ,,,]; %%Matrices omitidas en el ejemplo
10.    q = [1 ,,,];
11.    P = X5*((p(11)*eye(n))*X5+(p(6)*eye(n)+p(7)*X1+p(8)*X2+ p(9)*X3+p(10)*X4)*eye(n))
           +p(1)*eye(n)+p(2)*X1 + p(3)*X2+p(4)*X3+p(5)*X4;
12.    Q = X5*((q(11)*eye(n))*X5+(q(6)*eye(n)+ q(7)*X1+q(8)*X2+ q(9)*X3+q(10)*X4)*eye(n))
           +(q(1)*eye(n)+q(2)*X1+q(3)*X2+ q(4)*X3+q(5)*X4);
13. end
14. C = Q\P;
```

4.5 Implementación

4.5.1 Versión paralela básica

A partir de la versión secuencial, basada en código de la página anterior, se decide usar la librería Quark para paralelizar la ejecución de las rutinas matemáticas.

Se sustituyen las llamadas a BLAS/LAPACK por llamadas al kernel de Quark:

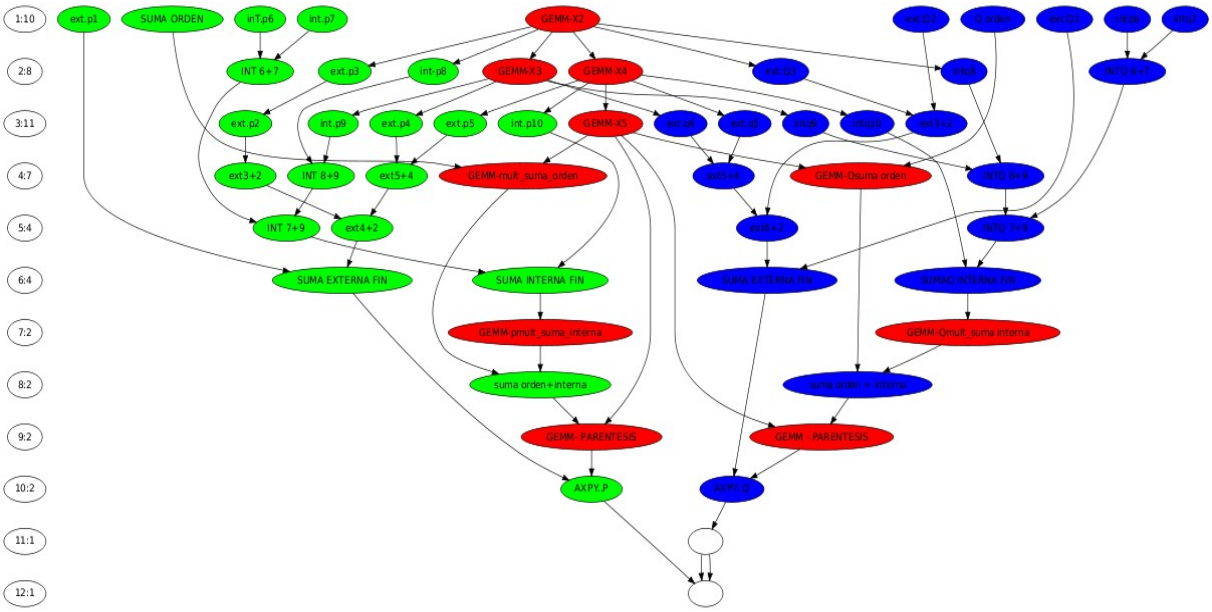
```
1.  cblas_dgemm (CblasColMajor, CblasNoTrans, CblasNoTrans,
2.                  N, N, N, alpha, X1, N, X1, N, beta, X2, N);
3.  cblas_dgemm (CblasColMajor, CblasNoTrans, CblasNoTrans,
4.                  N, N, N, alpha, X2, N, X1, N, beta, X3, N);
```

Se convierte en:

```
1.      QUARK_Insert_Task( quark, QUARK_gemm_core, NULL,
2.                  sizeof(int), &N, VALUE,
3.                  sizeof(double), &alpha, VALUE,
4.                  sizeof(double)*N*N, X1, INPUT,
5.                  sizeof(double)*N*N, X1, INPUT,
6.                  sizeof(double), &beta, VALUE,
7.                  sizeof(double)*N*N, X2, OUTPUT,
8.                  0 );
9.
10.     QUARK_Insert_Task( quark, QUARK_gemm_core, NULL,
11.                 sizeof(int), &N, VALUE,
12.                 sizeof(double), &alpha, VALUE,
13.                 sizeof(double)*N*N, X2, INPUT,
14.                 sizeof(double)*N*N, X1, INPUT,
15.                 sizeof(double), &beta, VALUE,
16.                 sizeof(double)*N*N, X3, OUTPUT,
17.                 0 );
```

Quark nos proporciona una utilidad para examinar como queda el DAG generado por nuestro código. En la Figura 11 (siguiente página), podemos distinguir cuatro colores: verde, rojo, azul y blanco. El color verde simboliza las operaciones AXPY realizadas por el vector P, el rojo representa las operaciones GEMM, el color azul son las operaciones AXPY correspondientes a Q, y los nodos blancos representan a DGESV(DGETRF Y DGETRS).

Varios nodos en el mismo nivel representan las operaciones que se pueden realizar de forma concurrente, como se puede apreciar tenemos sólo dos operaciones GEMM en paralelo que son las que van determinar el coste del algoritmo.



4.5.2 Paralelización con OpenMP

Sólo se ha paralelizado la versión secuencial con OpenMP, se ha hecho mediante la directiva *task*. Puesto que el trabajo de buscar a mano las funciones concurrentes y reordenar el código para evitar puntos de sincronización recae sobre el programador se ha decidido no usar OpenMP en la versión por bloques.

Aquí adjunto un código de ejemplo:

```
18.  #pragma omp task
19.  {
20.        cblas_dgemm (CblasColMajor, CblasNoTrans, CblasNoTrans,
21.              N, N, N, alpha, X1, N, X1, N, beta, X2, N);
22.        #pragma omp task
23.        cblas_dgemm (CblasColMajor, CblasNoTrans, CblasNoTrans,
24.              N, N, N, alpha, X2, N, X1, N, beta, X3, N);
25.        #pragma omp task
26.        cblas_dgemm (CblasColMajor, CblasNoTrans, CblasNoTrans,
27.              N, N, N, alpha, X2, N, X2, N, beta, X4, N);
28.        #pragma omp taskwait
29.
30.  }
31.  #pragma omp taskwait
```

La versión paralela en OpenMP tenía peores resultados que el algoritmo básico sobre Quark para matrices pequeñas.

4.5.3 Versión por bloques

Para intentar mejorar el rendimiento del algoritmo, se ha realizado una versión por bloques.

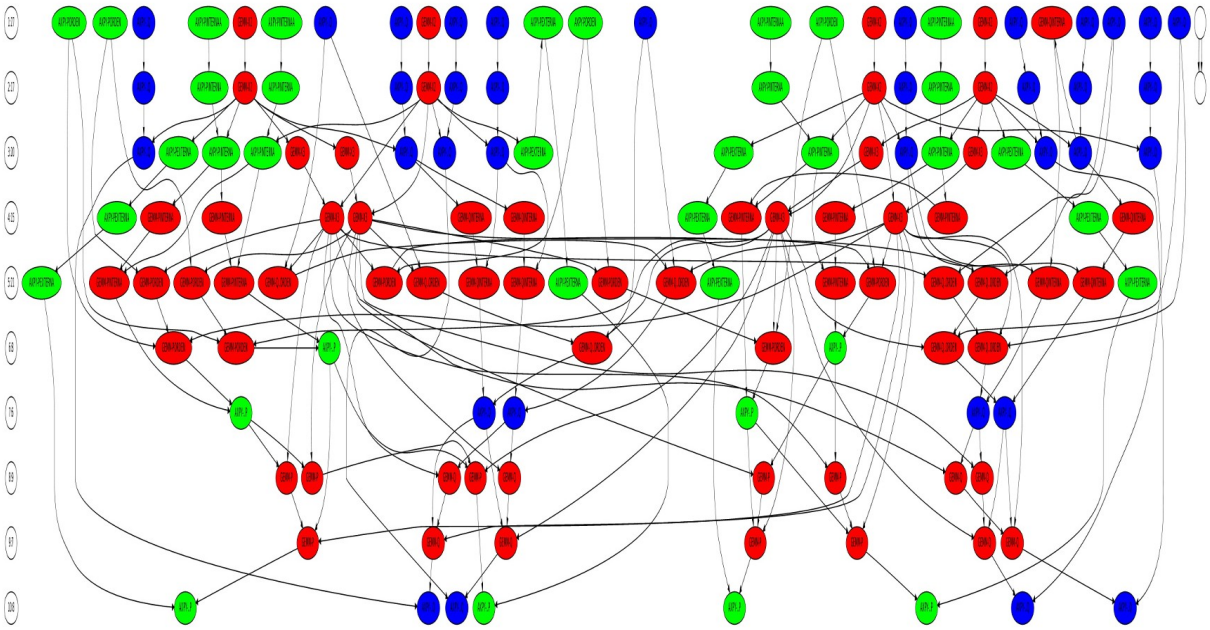


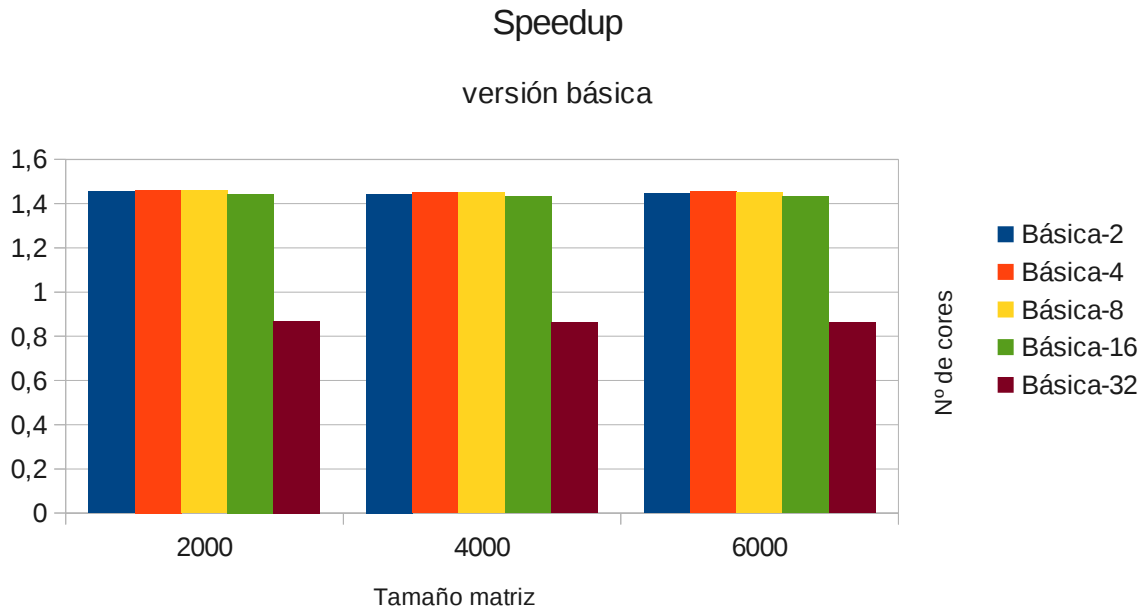
Figura 12. Versión Quark por bloques

4.6 Resultados

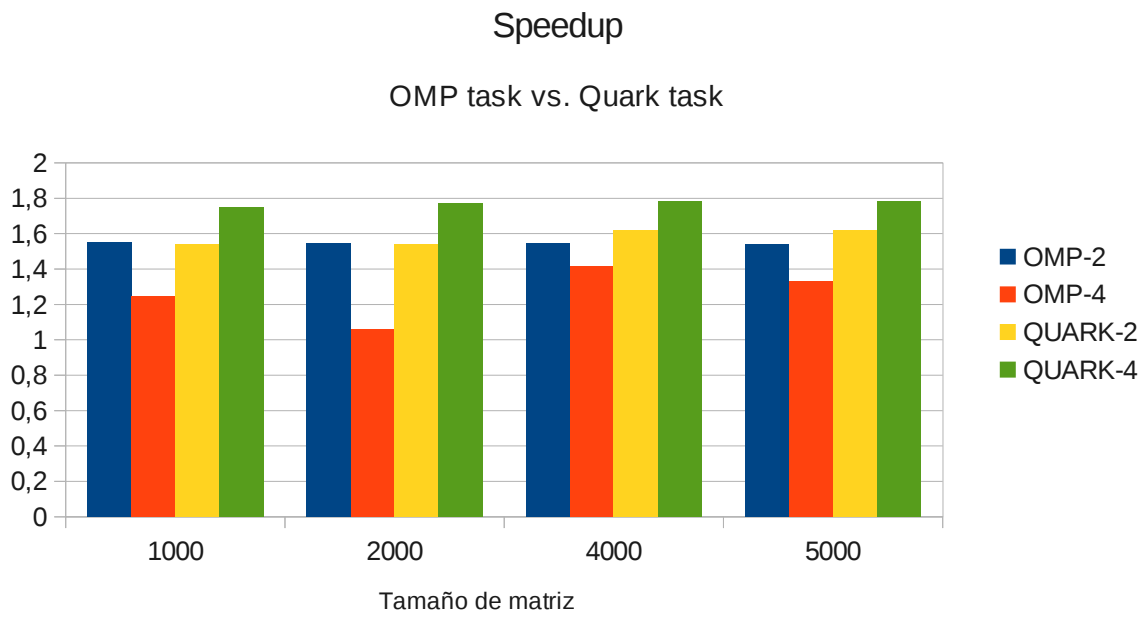
Si comenzamos por los resultados de la versión básica se observa que existe un paralelismo indicado por la naturaleza del algoritmo que nos permite realizar varias operaciones de suma de matrices (realizadas mediante la operación axpy), y multiplicación de matrices(operación GEMM).

Puesto que la multiplicación de matrices tiene mayor coste que el resto, n^3 , será su paralelización la que mejore el resultado del algoritmo.

En este caso apreciamos que sólo se pueden realizar de forma concurrente dos operaciones GEMM. Por lo que se aprecia una ligera mejora del rendimiento, pero si aumentamos el número de procesadores a más de dos, estos estarían desocupados durante gran parte del cálculo del algoritmo.
Esta prueba se ha realizado en el cluster kahan.

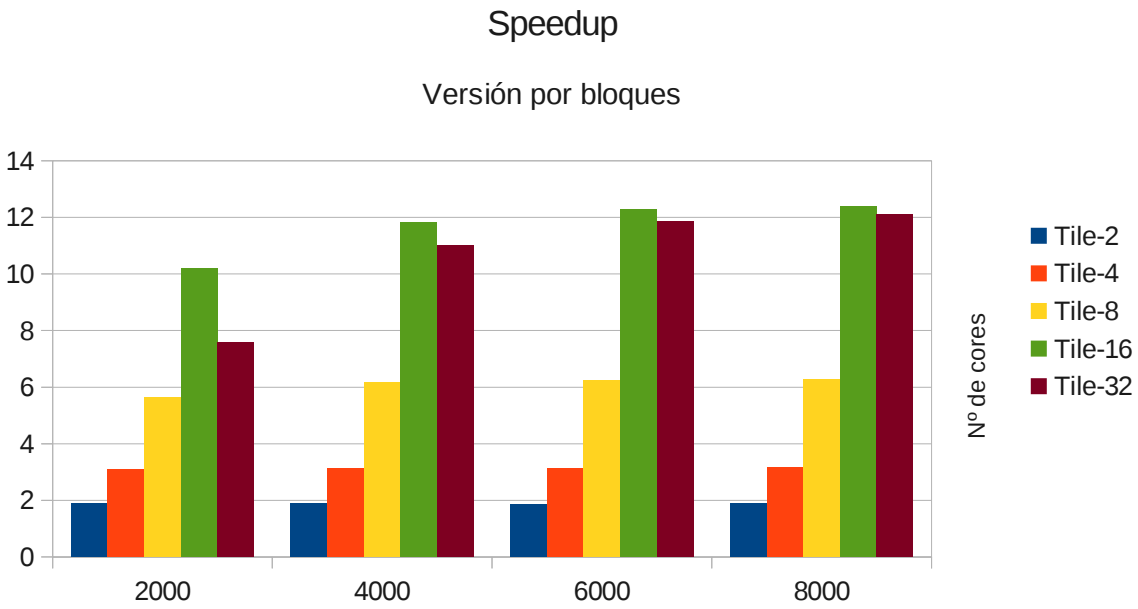


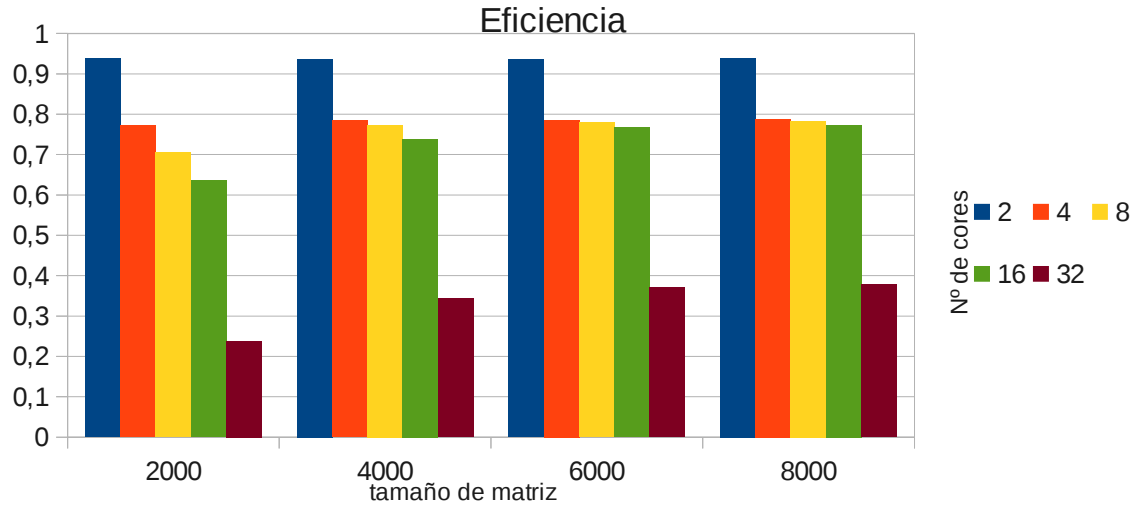
Para compararla he usado una versión con tareas OpenMP en la máquina athor.



Vemos que puesto que el número de tareas no crece, conforme aumenta el tamaño de la matriz, los resultados se mantienen estables, ya que no estamos usando ninguna librería matemática paralelizada, resultaría interesante, que cambiáramos la forma de utilizar las matrices.

El speedup y la eficiencia de la versión por bloques, en el cluster kahan:





La estructura de matriz por bloques nos permite mantener ocupados los procesadores, ya que cada *core* opera sobre un bloque. En las dos versiones se han ido apreciando se ha ido mejorando el tiempo de ejecución de manera incremental.

Por ello, con la idea de ganar escalabilidad se presenta un el algoritmo con la matriz almacenada por bloques y por lo tanto ahora pasamos a realizar todas las llamadas a funciones por bloques.

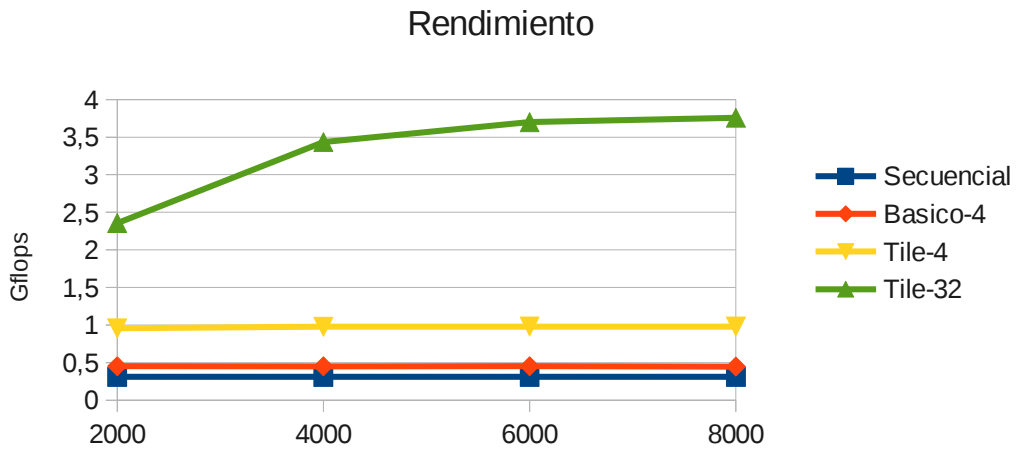
Con esto hemos conseguido aumentar el número de operaciones costosas(GEMM) que se realizan en paralelo, y como efecto secundario de tener una granularidad más fina en este algoritmo,se consigue también aumentar el número de operaciones axpy que se realizan de forma simultanea.

Examinando detenidamente el algoritmo, se aprecia que todavía existen varias partes secuenciales en el código, principalmente cuando se debe resolver un sistema de ecuaciones al final del algoritmo, operación DGESV.

Hasta este momento, se han utilizado dos softwares, Quark y LAPACK, puesto que la función DGESV estaba implementada en LAPACK, esto requería hacer una conversión de matrices por bloque a matrices por columna, o buscar un algoritmo para resolver un sistema de ecuaciones por bloque. La segunda opción parece la más adecuada.

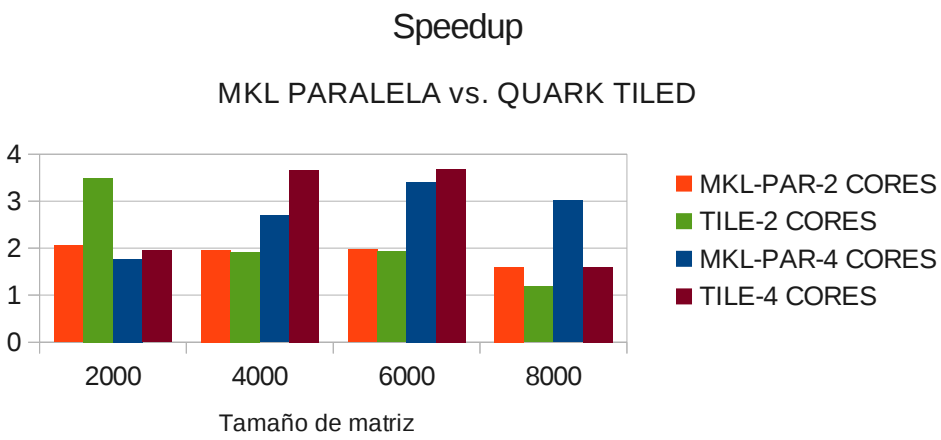
Entonces aprovechando que la librería PLASMA ofrece rutinas por bloque para mejorar la utilización de la jerarquía de memoria, he decidido usar las rutinas de PLASMA_DGESV por bloques en vez de usar LAPACK. Con el resultado de evitarnos el tiempo de conversión, o complicar la lógica del algoritmo teniendo que usar llamadas LAPACK.
Si presentamos el rendimiento de los algoritmos:

En esta comparación vemos la diferencia de rendimiento entre algunos de los algoritmos usados en este estudio en el cluster kahan.



Puesto que la librería MKL paralela es conocida por su buen rendimiento, he comparado el algoritmo por bloques que usa la MKL secuencial con una implementación del algoritmo que usa

la MKL paralela en la máquina “athor”. En ciertos casos, la implementación de quark junto con la MKL secuencial funciona más rápido que el algoritmo que utiliza MKL paralela.



5 Algoritmos matriciales con almacenamiento jerárquico

5.1 Matrices jerárquicas

Las matrices jerárquicas o H-matrix, ofrecen una aproximación de tipo disperso a matrices densas a través del intercambio de determinados bloques de la matriz por matrices de bajo rango que pueden ser almacenadas de forma muy eficiente [17][18]. En este trabajo se usará la librería HLIB.

El formato H-matrix se basa en una estructura de árbol jerárquica, llamada *block cluster tree*, que se obtiene haciendo un particionado jerárquico de un conjunto de índices en sub-bloques, al que llamaremos *cluster tree*.

Cuando estamos construyendo un árbol jerárquico, en cada nivel de particionado los sub-bloques deben satisfacer una condición de admisibilidad.

Según el concepto de admisibilidad, en el último nivel de la jerarquía del árbol se encuentran las hojas, que son los elementos que no pueden ser particionados más veces. Por ello es necesario crear unos criterios, que definan si un elemento será hoja o se podrá particionar en más elementos.

,

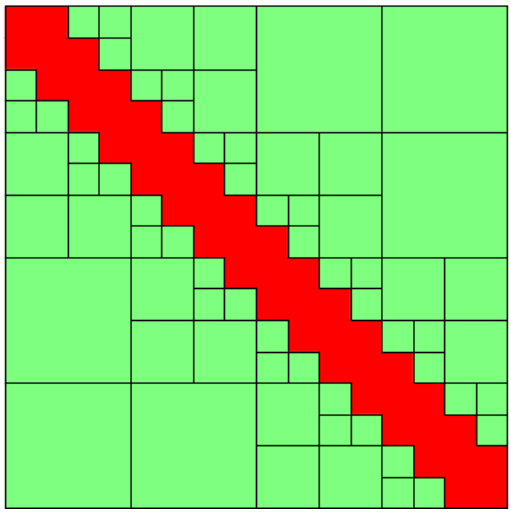
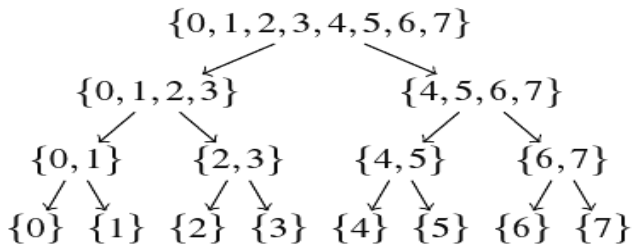


Figura 13. Matriz jerárquica

Los bloques verdes representan las aproximaciones creadas por matrices de bajo rango. Los bloques rojos representan matrices densas.

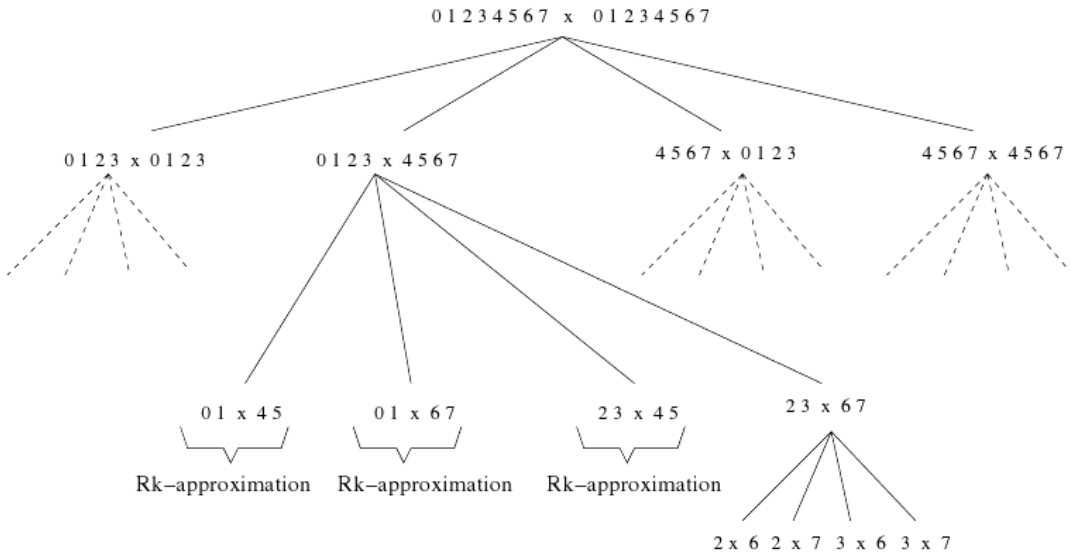
Los bloques que crearán las representaciones de bajo rango salen de una jerarquía de particiones llamada *cluster tree*.

Pongamos un ejemplo, digamos que tenemos un índice que queremos particionar, en este caso el conjunto I = {1,2,3,4,5,6,7}. Como se aprecia en la figura hemos ido dividiendo los índices entre dos conjuntos separados hasta que hemos llegado a las hojas.



El conjunto I se particiona como T_i, creándose el árbol Cluster tree, mientras que si quisiéramos particionar una matriz se debería usar un conjunto T_{ixi}.

A continuación se particiona una matriz. La partición se considera mediante un criterio de admisibilidad. Se considera admisible si dos elementos s y t están conectados en el grafo. Por ejemplo según la figura inferior {2,3} y {4,5} no están relacionados en el grafo.



Si queremos ver la matriz resultado, sería algo similar a esto:

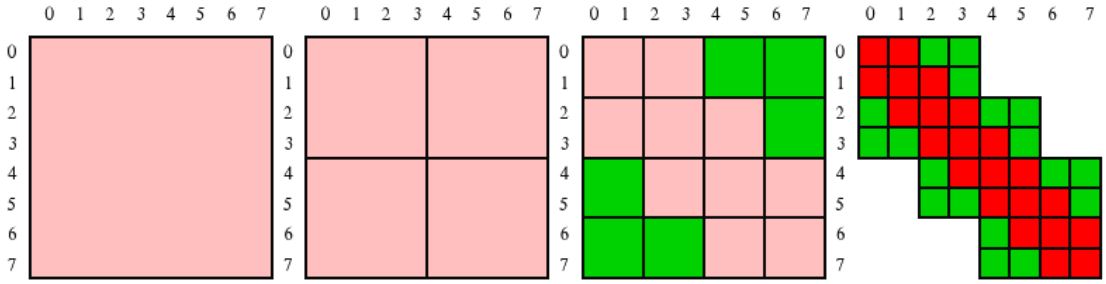
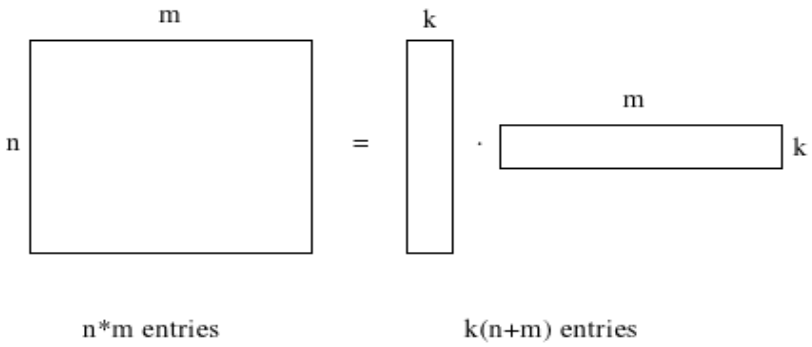


Figura 16. Representación de H-matrix

De los cuatro niveles del *block cluster tree* de T_{ixi} , los nodos que pueden ser refinado están en color rosa, las hojas inadmisible son de color rojo y las hojas admisibles son de color verde. Los bloques verdes al ser admisibles se pueden representar eficientemente mediante matrices de bajo rango.

Las matrices de bajo rango, rk-matrix, tienen como ventaja que su almacenamiento, formado por dos matrices.

$$\underset{n \times m}{A} \approx \underset{n \times k}{B} \; \underset{k \times m}{C}$$



Aquí se presentan las principales diferencias entre matrices densas y matrices rk.

	Densa	Matriz RK
Almacenamiento	**mxn**	**K (m + n)**
Producto Matriz-vector	**O(mn)**	**O(k(m + n))**

5.2 Métodos iterativos

Los métodos iterativos nos sirven para encontrar una solución mediante aproximaciones sucesivas a la solución, empezando desde una estimación inicial. Su principal ventaja es la de ser capaces de resolver matrices dispersas de gran tamaño de una forma más sencilla [4].

Los métodos de Krylov extraen aproximaciones a los vectores propios de una matriz cuadrada A de orden n a partir del denominado subespacio de Krylov, definido como:

$$\mathcal{K}_m(A,v) \equiv \left\langle v, Av, A^2v, \ldots, A^{m-1}v \right\rangle$$

donde v es un vector arbitrario y $m \leq n$ es la dimensión máxima del subespacio.

Si se realizan n pasos del método de Arnoldi se obtiene una reducción ortogonal a la forma de Hessenberg, AV = V H , donde H es una matriz Hessenberg superior de orden m , con los elementos de la primera subdiagonal positivos, y V es una matriz unitaria cuyas columnas forman una base del subespacio de Krylov.

Dado un vector v_1 de norma unidad
para $j = 1, 2, \ldots, m$
  $v_{j+1} = Av_j$
  para $i = 1, 2, \ldots, j$
    $h_{i,j} = v_i^* v_{j+1}$
    $v_{j+1} = v_{j+1} - h_{i,j} v_i$
  fin
  $h_{j+1,j} = \|v_{j+1}\|_2$
  si $h_{j+1,j} = 0$ **parar**
  $v_{j+1} = v_{j+1} / h_{j+1,j}$
fin

Las matrices V y H están determinadas de forma única (salvo cambios de signo) por la primera columna de V.

Para algunos posibles vectores iniciales, el método de Arnoldi se tiene que parar después de m pasos porque $\|v_{j+1}\|_2$ se hace cero tras la ortogonalización, debido a que v_{j+1} es una combinación lineal de los vectores anteriores. En ese caso el algoritmo produce una matriz V_m de dimensión $n \times m$, con columnas ortogonales, y una matriz H_m Hessenberg su-

perior de orden m con sus elementos no nulos h_{ij} definidos por el algoritmo, que satisface $AV_m - V_m H_m = 0$.

Sin embargo, $\|v_{j+1}\|_2$ nunca llega a ser exactamente cero en la implementación del algoritmo con coma flotante, y puede resultar problemático detectar esta condición.

En el caso general, donde $\|v_{j+1}\|_2$ no se anula después de m pasos, se cumple la siguiente relación $AV_m - V_m H_m = h_{m+1,m} v_{m+1} e_m^*$.
Donde al producto $h_{m+1,m}(v_{m+1})$ se le suele llamar el residuo de la factorización de Arnoldi de orden m [4].

Los vectores de la base son las columnas de V_m , que se denominan vectores de Arnoldi. Por construcción se tiene que $V_m^* v_{m+1} = 0,$ se obtiene multiplicando por la izquierda la ecuación $AV_m - V_m H_m = h_{m+1,m} v_{m+1} e_m^*$ por V_m^* se obtiene:

$$V_m^* A V_m = H_m$$

es decir, la matriz H_m representa la proyección ortogonal de A en el subespacio de Krylov, lo que nos permite calcular m aproximaciones de Rayleigh-Ritz a los valores propios de A,

$$H_m y_i = \lambda_i y_i$$

donde los valores λ_i son aproximaciones a los valores propios de A y $x_i = V_m y_i$ a sus vectores propios asociados. Para determinar cuáles de estas m aproximaciones son suficientemente precisas se utiliza la norma del residuo, que se puede calcular con un coste pequeño aplicando la ecuación $AV_m - V_m H_m = h_{m+1,m} v_{m+1} e_m^*$.
El residuo.

$$\|Ax_i - \lambda_i x_i\|_2 = \|AV_m y_i - \lambda_i V_m y_i\|_2 = \|(AV_m - V_m H_m) y_i\|_2 = h_{m+1,m} |e_m^* y_i|$$

5.3 SLEPC

SLEPC (Scalable Library for Eigenvalue Problem Computation) es una librería orientada a resolver problemas de valores propios y valores singulares grandes y dispersos en entornos paralelos. Soporta problemas estándares y generalizados, tanto hermitianos como no hermitianos, y aritmética real o compleja. Puede ser usado desde código en C, C++ y Fortran [13].

SLEPc está creado para actuar como una capa superior a PETSc (Portable, Extensible Toolkit for Scientific Computation,), una librería para la solución de problemas de ecuaciones diferenciales parciales, y que usa principalmente estructuras de datos básicas como vectores y matrices.

SLEPc ofrece una colección de herramienta para averiguar los valores propios, la mayoría están basados en el paradigma de proyección de subespacios. En particular incluye un implementación muy eficiente del método de Krylov-Schur. Además incluye varias funciones para resolver los métodos de Jacobi-Davidson y Davidson generalizado, con la posibilidad de calcular el vector de corrección.

En estas funciones, el usuario puede fácilmente seleccionar que precondicionador desea usar.

También se ofrece transformaciones de espectro como la técnica shift-and-invert, donde el usuario puede calcular valores propios interiores con la ayuda de solvers lineales y precondicionadores incluidos en PETSc

Las funciones en PETSc y en SLEPc tienen una implementación genérica respecto a la estructura de datos usada. Es decir, el cálculo se puede llevar a cabo con diferentes formatos de almacenamiento de matriz, incluso si la matriz no estuviera almacenada explícitamente.

Por defecto una matriz en PETSc se almacena en un formato disperso con filas comprimidas paralelas, en el cual cada procesador tiene un conjunto de filas. Para ser capaz de implementar una función genérica(shell matrix), el programador de la aplicación tiene que crear una matriz de ese tipo y definir sus operaciones, enlazándolas con una rutina creada por el usuario para cada operación.

Sólo las operaciones necesarias para el cálculo han de ser implementadas, por lo que en el caso más simple con definir el producto matriz vector sería suficiente.

5.4 Estudio de las atmósferas estelares(albedo)

Si los valores propios no están bien separados, y se quiere obtener los valores propios interiores. Es útil reemplazar A por una función shift-and-invert

$$C = (A - \sigma I)^{-1}$$

para un determinado shift σ, por ejemplo, en el intervalo $\alpha \leq \sigma \leq \beta$, donde deseamos conocer los valores propios. La función shift-and-invert C tiene los valores propios

$$\theta_i = \frac{1}{\lambda_i - \sigma} \quad \text{or} \quad \lambda_i = \sigma + \frac{1}{\theta_i},$$

Y ahora los valores propios θ_i que corresponden con los valores propios λ_i cercanos al valor de shift σ estarán en los extremos del espectro y estarán separados del resto claramente.

En la función shift-and-invert, el algoritmo comienza usando algún esquema de eliminación Gaussiano.

$$LDL^* = P^T(A - \sigma I)P,$$

P es una permutación y L es una matriz triangular inferior. Si existen valores propios λ en ambos lados del shift σ, no podemos utilizar una diagonal escalar D, sino que tenemos que calcular una factorización indefinida simétrica. Aquí D es un bloque diagonal de 1x1 y 2x2 bloques, ahora podemos conseguir la inercia $A - \sigma I$ como resultado.

Ahora se puede contar el número de valores propios en un intervalo sabiendo la inercia de $(A - \sigma I)$ para dos valores σ en los bordes del intervalo. Ahora que ya tenemos el número de valores, podemos garantizar que no nos hemos dejado ningún valor propio múltiple.

Durante la iteración, se usan los factores P, L, y D, para calcular

$$r = P(L^{-*}(D^{-1}(L^{-1}(P^T v_j)))) ,$$

Si consideramos el problema de los valores propios desde la formulación integral de un problema de transferencia en las atmósferas estelares, con los operadores

T : X → X, **X = L1 ([0, τ])**, definidos por:

$$(Tx)(\tau) := \int_0^{\tau^\star} g(\tau,\sigma)x(\sigma)d\sigma, \quad \tau \in [0,\tau^\star] ,$$

donde τ es finita, y dada la función

$$g(\tau,\sigma) := \frac{\varpi}{2} E_1(|\tau-\sigma|)$$

La función es una ecuación integral débilmente singular que se define a través de la primera función exponencial-integral, la primera de una serie de funciones es

$$E_\nu(\tau) := \int_1^\infty \frac{\exp(-\tau\mu)}{\mu^\nu} d\mu, \quad \tau > 0, \nu \geq 0$$

que depende de albedo, $\varpi \in [0,1]$ el cual consideraremos constante.
Para resolver el problema de los valores propios,

$$T\varphi = \lambda\varphi$$

con $\lambda \neq 0$ y $\varphi \neq 0$, $\varphi \in X$, consideramos una clase de aproximaciones en las cuales el rango es un subespacio dimensional finito de X .

Se crea una aproximación de rango finito de T mediante grids en [0, τ] [20].

Para cada $x \in X$ establecemos

$$\langle x, e^*_{n,j}\rangle := \frac{1}{\tau_{n,j} - \tau_{n,j-1}} \int_{\tau_{n,j-1}}^{\tau_{n,j}} x(\sigma)d\sigma,$$

donde $e_{n,j} = 1$ si $\tau \in]\tau_{n,j-1}, \tau n, j[$ y sino es $e_{n,j} = 0$.

Una proyección limitada de rango n en el subespacio $X_n = span\{e_{n,j} : j = 1, \ldots, n\}$ viene definidia por estas dos ecuaciones:

$$\pi_n x := \sum_{j=1}^{n} \langle x, e^*_{n,j} \rangle e_{n,j},$$

$$T_n x = \pi_n T x.$$

Para obtener la representación de una matriz de $T_n \varphi_n = \theta_n \varphi_n$:

El problema espectral para T_n se reduce a un problema de valores propios de una matriz $n \times n$.

$$A_n x_n = \theta_n x_n,$$

donde $\mathrm{A}_n(i, j) := \langle \mathrm{T}\, e_{n,j}, e^*_{n,i} \rangle$

5.5 Implementación

SLEPc implementa varios métodos iterativos para resolver valores propios. En albedo, se ha utilizado el método de Krylov-Schur.

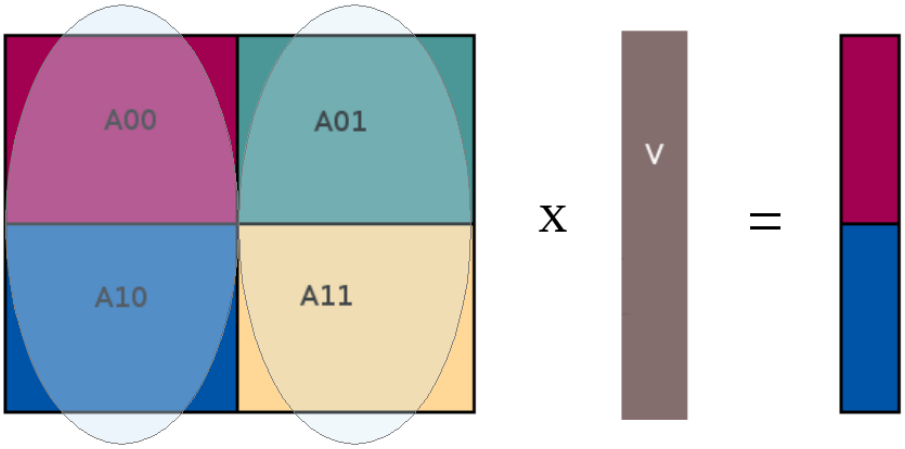
HLIB trae implementado una serie de funciones de creación de matrices jerárquicas, en albedo se ha utilizado la estructura de datos supermatrix. Además la librería ofrece las operaciones matemáticas más comunes para matrices jerárquicas (suma, multiplicación, factorización, etc...), y utiliza BLAS/LAPACK para resolverlas.

En este caso de estudio se ha usado SLEPC para crear una matriz shell que invoca al producto matriz-vector de la librería HLIB.

La paralelización se basa en paralelizar el recorrido de la estructura de datos, en este caso se trata de un árbol no balanceado.

```
1. mult_hmatrix_vec(H, v , r ) {
2.         for each child  H_i  de H :
3.                 mult_hmatrix_vec(  H_i  , v , r ) ;
4.         if H es Rk−matrix
5.                 mult_RKmatrix_vec(H, v, r) ;
6.         if H es  full−matrix
7.                 mult_FULLmatrix_vec(H, v, r) ;
8. }
```

Entonces, la paralelización propuesta sería la siguiente siguiente:
Puesto que existe una dependencia entre A00 y A01, sólo puedo realizar en paralelo, los bloques A00 y A10. Por lo tanto debo añadir sincronización. Otra solución sería añadir vectores resultados intermedios y lo hacer una suma entre ellos para sacar el resultado final.



Puesto que como máximo estoy paralelizando la mitad del código, puedo intuir por la ley de Amdahl, que la aceleración que obtendré no mayor que 2.

5.5.1 Paralelización con OpenMP

La paralelización se realiza cambiando la recursión por una creación recursiva de tareas, donde las tareas se hacen en paralelo(recursivamente) y por eso hace falta poner puntos de sincronización.

Por lo tanto cuando comienza la exploración de nodos, sólo se podrán paralelizar dos tareas hijas por cada nodo, de las cuatro posibles. Es decir, dos tareas hijas $(A00, A10)$, comienzan su exploración mientras que el padre queda suspendido hasta que se completen las dos primeras tareas, y puedan iniciarse las dos siguientes tareas hijas $(A01, A11)$.

```
1.  fastaddscaleeval_supermatrix(pcsupermatrix s, double alpha, const double *v, double *w) {
2.    int i,j;
3.    int vindex;
4.    int windex;
5.    int block_rows,block_cols;
6.    psupermatrix *s_el;
7.
8.    assert(s != 0x0);
```

```
  assert(v != 0x0);
  assert(w != 0x0);

  block_rows = s->block_rows;
  block_cols = s->block_cols;
  s_el = s->s;

  if(s->s) {
    vindex = 0;
    for(j=0; j<block_cols; j++) {
      windex = 0;
      for(i=0; i<block_rows; i++) {
        #pragma omp task
        fastaddscaleeval_supermatrix(s_el[i+j*block_rows], alpha,
                                     v+vindex, w+windex);
        windex += s_el[i+j*block_rows]->rows;
      }
      assert(windex == s->rows);
      vindex += s_el[j*block_rows]->cols;
      #pragma omp taskwait
    }
    assert(vindex == s->cols);
  }

  if(s->u)
    fastaddscaleeval_uniformmatrix(s->u, alpha);

  if(s->r)
    addscaleeval_rkmatrix(s->r, alpha, v, w);

  if(s->f)
    addscaleeval_fullmatrix(s->f, alpha, v, w);
}
```

5.5.2 Paralización con X-KAAPI

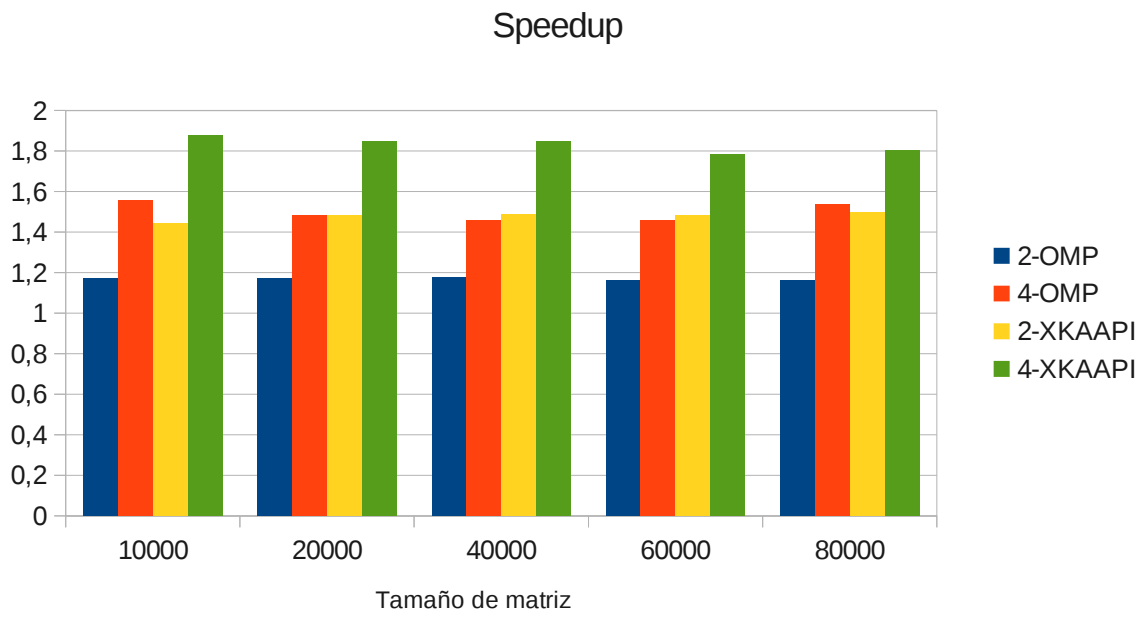
La paralelización se realizar cambiando la recursión por una creación de tarea. X-KAAPI es capaz de resolver dependencias por eso no necesito puntos de sincronización explícitos.

```
mxv_supermatrix(pcsupermatrix s, const double *v, double *w) {
  int i,j;
  int vindex;
  int windex;
  int block_rows,block_cols;
  psupermatrix *s_el;
  assert(s != 0x0);
  assert(v != 0x0);
  assert(w != 0x0);

  block_rows = s->block_rows;
  block_cols = s->block_cols;
  s_el = s->s;

```

```
15.   if(s->s) {
16.     vindex = 0;
17.     for(j=0; j<block_cols; j++) {
18.       windex = 0;
19.       for(i=0; i<block_rows; i++) {
20.        kaapic_spawn(3,  mxv_supermatrix,
21.           KAAPIC_MODE_R, s_el[i+j*block_rows], 1, KAAPIC_TYPE_PTR ,
22.           KAAPIC_MODE_R, v+vindex, s_el[i+j*block_rows]->block_cols , KAAPIC_TYPE_DOUBLE,
23.           KAAPIC_MODE_RW, w+windex,  s_el[i+j*block_rows]->block_rows,
              KAAPIC_TYPE_DOUBLE
24.        );
25.
26.         windex += s_el[i+j*block_rows]->rows;
27.             }
28.       assert(windex == s->rows);
29.       vindex += s_el[j*block_rows]->cols;
30.     }
31.     assert(vindex == s->cols);
32.    }
33.
34.   if(s->r){
35.     addscaleeval_rkmatrix(s->r, 1.0, v, w);
36.
37.   }
38.    if(s->f){
39.     addscaleeval_fullmatrix(s->f, 1.0, v, w);
40.
41.   }}
```

5.6 Resultados

Estos datos se han tomado en el cluster kahan.



La paralelización de multiplicación de una matriz jerárquica por un vector, está compuesta de dos operaciones internas, una para una matriz rk, y otra para una matriz densa.

En Quark, si usamos las dos funciones de addscaleeval_rkmatrix y addscaleeval_fullmatrix como funciones para tareas, debido a la restricción INOUT y a que exploración del árbol la realiza sólo el hilo maestro, se termina haciendo de forma secuencial, con peores resultados que la versión secuencial.

Debido a la estructura recursiva del algoritmo sobre un árbol no balanceado, y puesto que las operaciones de cálculo se encuentran en las hojas, y se hacía un a exploración mediante DFS, decidí intentar mejorar el tiempo que tardaba en el recorrer el árbol.

Para paralelizar exploración del árbol sustituí la llamada recursiva, por una llamada de creación de tarea.

Puesto que se está paralelizando la exploración del árbol sólo mediante dos nodos, y debo mantener la coherencia de los datos mediante puntos de sincronización, se consiguen unos mejoras muy pobres.

Otra factor importante ha sido que gran parte de la matriz esta formada por matrices de bajo rango, y cuando se hacen productos matrices-vector con una matriz de tamaño mxk , donde la k es un factor muy pequeño, se puede concluir que se hacen pocas operaciones en relación al movimiento de datos.

Basándome en esta versión, y debido al hecho que Quark no soporta la creación recursiva de tareas, decido usar otra librería, en este caso XKAAPI, puesto que XKAAPI también soporta funcionalidad de análisis de dependencias cuando va explorando sus tareas, puedo conseguir evitar algunos puntos de sincronía que tenía con OpenMP.

Pero debido al hecho de que X-KAAPI no despliega todo el grafo, este no es capaz de extraer el paralelismo necesario para ejecutar las operaciones en las hojas de forma concurrente. De todas formas se consigue una mejora de prestaciones con respecto al algoritmo anterior en OpenMP.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

En este trabajo, se había propuesto examinar las diferencias que se podían encontrar al resolver algoritmos mediante librerías que resolvieran las dependencias de datos mediante grafos respecto a las herramientas tradicionales como OpenMP, donde es el programador el responsable de mantener la coherencia de los datos.

La implementación se ha desarrollado bajo los objetivos de la computación de altas prestaciones, pero también se ha hecho especial hincapié en la facilidad y comodidad del programador.
Se han estudiado dos tipos de problemas, uno sobre matrices densas y otro sobre matrices jerárquicas, por un lado se puede decir que se ha demostrado satisfactoriamente el uso de estas herramientas de planificación dinámica en ambos casos, ya que en ambos casos las librerías con resolución de dependencias mediante DAG han tenido mejores resultados que los algoritmos que usaban OpenMP dependiendo totalmente del programador.

Por otra parte, se debe destacar que el problema de la resolución de un producto matriz-vector en matrices jerárquicas propuesto en este estudio no es adecuado para aplicar estas técnicas de paralelización.

Para finalizar, es necesario resaltar que las herramientas de resolución de dependencias y la planificación de tareas de forma dinámica, aumentan la productividad del programador y proporcionan un muy buen rendimiento.

6.2 Trabajo futuro

Intentar implementar albedo para que use DaGUE, y comprobar la diferencia entre la exploración de DAG varias librerías y su formato para resolver dependencias.
De la misma manera se podría implementar diferentes funciones de matrices con Quark y medir sus resultados, como por ejemplo la matriz exponencial. Por otro lado se podría implementar el algoritmo del cálculo del coseno mediante aproximantes Padé con StarPU o X-KAAPI, y ver sus diferencias con Quark.

7 Apéndice

7.1 Entorno de pruebas

Se han realizado pruebas en dos máquinas diferentes:

- Cluster de computación “kahan”.
- Máquina multicore “athor”.

El cluster de computación 6 nodos biprocesador conectados mediante una red InfiniBand.

Cada nodo consta de:

- 2 procesadores AMD Opteron 16 Core 6272, 2.1GHz, 16MB
- 32GB de memoria DDR3 1600
- Disco 500GB, SATA 6 GB/s
- Controladora InfiniBand QDR 4X (40Gbps, tasa efectiva de 32Gbps)

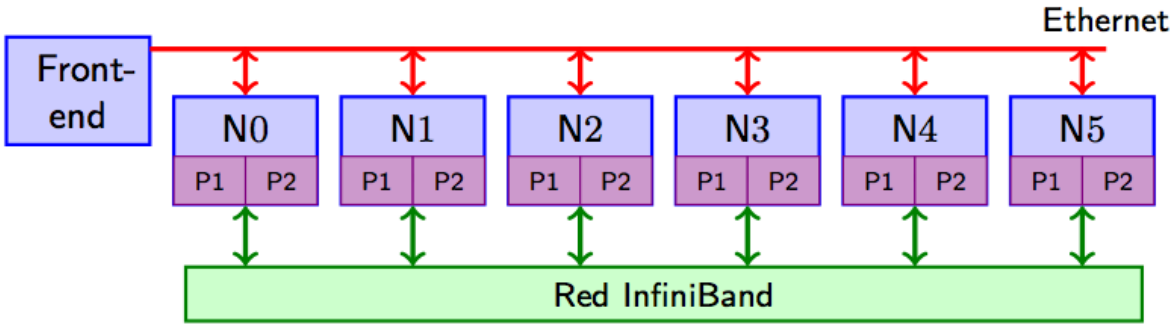


Figura 20. Cluster kahan

Sean usado las siguientes librerías software: Atlas para operaciones BLAS/LAPACK, Quark 0.9 y X-KAAPI 1.0.1, PETSC 3.3, SLEPC 3.3 y HLIB 1.3.

Los resultados en kahan se han compilado con gcc 4.4.6.

La máquina multicore athor:

- consta de 4 cores Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz, 8192 KB
- 8 GB de RAM

Se han usado las librerías de Intel MKL para operaciones BLAS/LAPACK, Quark 0.9 y X-KAAPI 1.0.1, PETSC 3.3, SLEPC 3.3 y HLIB 1.3.

Los resultados se han compilado con icc.

8 Bibliografía

[1] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., & Dongarra, J. (2011). DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*.

[2] Bosilca, George, et al. (2012): From Serial Loops to Parallel Execution on Distributed Systems, *Euro-Par 2012 Parallel Processing*, 246-257.

[3] Eduard Ayguadé, Nawal Copty, Alejandro Duran, et al.(2009): The Design of OpenMP tasks. IEEE Transactions on Parallel and Distributed Systems:404-418.

[4] Andrés Enrique Tomás Domínguez, (2009): Implementación paralela de métodos de Krylov con reinicio para problemas de valores propios y singulares, URL: www.dsic.upv.es/docs/bib-dig/tesis/etd-12152008-114615/TesisAndresTomas.pdf [Consulta 04/02/2013].

[5] Nicholas Higham (2008): Functions of Matrices: Theory and Computation, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA: 287-290.

[6] Ltaief, H., & Yokota, R. (2012). Data-driven execution of fast multipole methods. *arXiv preprint arXiv:1203.0889*.

[7] Gautier Thierry, Lementec Fabien, et al.(2012):X-Kaapi: a Multi Paradigm Runtime for Multicore Architectures, URL:hal.archives-ouvertes.fr/docs/00/72/78/27/PDF/RR-8058.pdf [Consulta 04/02/2013].

[8] Anthony Danalis, et al.(2012):Task data flow analysis and extraction from serial input code, URL:web.eecs.utk.edu/~dongarra/CCGSC-2012/talk23-Danalis.pdf [Consulta 04/02/2013].

[9] Jack Dongarra, A. J. van der Steen.(2012): High-performance computing systems: Status and outlook. Acta Numerica, 21:379–474. URL:http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpc-acti.pdf [Consulta 04/02/2013].

[10] Thierry Gautier, Fabien Lementec, Vincent Faucher, Bruno Raffin(2012): X-K AAPI: a MultiParadigm Runtime for Multicore Architectures, URL:hal.archives-ouvertes.fr/docs/00/72/78/27/PDF/RR-8058.pdf [Consulta 04/02/2013].

[11] Jack Dongarra, et al.(2012): Multithreading in the PLASMA Library, URL:http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunilstyle.pdf [Consulta 04/02/2013].

[12] A. Grama, Gupta, et al.(2003): An Introduction to Parallel Programming, Pearson Education:279-331

[13] Eloy Romero (2009): Implementación paralela del método de minimización de la traza para el problema de valores propios generalizado simétrico URL: http://riunet.upv.es/handle/10251/12260 [Consulta 04/02/2013].

[14],Cédric Augonnet(2009): StarPU: a unified platform for task scheduling onheterogeneous multicore architectures URL: www.par.univie.ac.at/project/peppher/publications/Published/cc-pe10.pdf [Consulta 04/02/2013].

[15] Jakub Kurzak, et al.(2009): QUARK Queuing And Runtime for Kernels URL: web.eecs.utk.edu/~kurzak/tutorials/ISC12/03_Kurzak_QUARK.pdf[Consulta 04/02/2013].

[16] Asim Yarkhan, Jakub Kurzak, and Jack Dongarra. (2011): QUARK Users' Guide. Tech-nical Report April, Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tenessee URL: icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_gui-de.pdf : [Consulta 04/02/2013].

[17] Hackbusch, W., & Grasedyck, L. (2002). An introduction to hierarchical matrices. *Mathematica Bohemica*, *127*(2), 229-241. URL: http://dml.cz/bitstream/handle/10338.dmlcz/134156/Math-Bohem_127-2002-2_11.pdf : [Consulta 04/02/2013].

[18] Börm, S., Grasedyck, L., & Hackbusch, W. (2003). Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements*, *27*(5), 405-422. URL: http://cite-seerx.ist.psu.edu/viewdoc/download?doi=10.1.1.126.9181&rep=rep1&type=pdf : [Consulta 04/02/2013].

[19] Higham, N. J., & Al-Mohy, A. H. (2010). Computing matrix functions. *Acta Numer*, *19*, 159-208. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.154.4605&rep=rep1&type=pdf [Consulta 04/02/2013].

[20] Roman, J. E., Vasconcelos, P. B., & Nunes, A. L. (2012). Eigenvalue computations in the context of data-sparse approximations of integral operators. *Journal of Computational and Applied Mathematics*. URL: users.dsic.upv.es/~jroman/preprints/datasparse.pdf [Consulta 04/02/2013].