UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Departamento de Sistemas
Informáticos y Computación

MÁSTER EN COMPUTACIÓN PARALELA Y DISTRIBUIDA

TESIS DE MÁSTER

Planificación consciente del consumo en algoritmos de álgebra lineal densa sobre procesadores multinúcleo

VALENCIA, A 10 DE DICIEMBRE DE 2010

TESIS DE MÁSTER PRESENTADA POR: MANUEL FRANCISCO DOLZ ZARAGOZÁ

DIRIGIDA POR: ENRIQUE S. QUINTANA ORTÍ
PEDRO ALONSO JORDÁ

Resumen

Desde años, el principal objetivo de la computación de altas prestaciones ha sido la optimización de algoritmos aplicados a la resolución de problemas complejos que, constantemente, aparecen en un amplio abanico de aplicaciones de casi todas las áreas científicas y tecnológicas. Esta optimización hace referencia directa a la reducción de los tiempos de ejecución. En este sentido, la mayoría de los logros conseguidos sobre optimización de prestaciones no hubieran sido posibles sin el constante avance tecnológico de los componentes de los computadores. El incremento de la frecuencia de los procesadores así como el número de los núcleos ha sido un factor clave para conseguir mejores prestaciones, aunque también ha servido para aumentar considerablemente su consumo. Por este motivo, la sociedad científico-técnica actual muestra especial interés en el desarrollo de herramientas y técnicas que permitan ahorrar energía en una variedad de componentes, particularmente en procesadores, a través técnicas que permiten reducir la frecuencia en momentos de baja demanda.

La temática de este trabajo se encuentra en la intersección de estas dos líneas de investigación: por una parte, los algoritmos paralelos de álgebra lineal densa aplicados a problemas de ingeniería y, por otra, los métodos y técnicas de ahorro de energía disponibles en procesadores mutinúcleo actuales. Concretamente, el objetivo del trabajo es realizar una planificación de tareas consciente del consumo en algoritmos de álgebra lineal densa sobre mutiprocesadores y procesadores multinúcleo, para ahorrar energía a través de la reducción de la frecuencia de los procesadores que ejecuten las tareas no críticas de los algoritmos afectando mínimamente a su rendimiento.

Como resultado de la primera parte del trabajo, se han implementado métodos para detectar prioridades entre tareas de algoritmos de álgebra lineal y procesar el grafo de dependencias para reducir las holguras entre las tareas no críticas. El algoritmo de reducción de holguras permite conocer a priori en qué porcentaje de tiempo una tarea puede ralentizarse, de modo que pueda reducirse, en tiempo de ejecución, la frecuencia del procesador que la ejecutará. La segunda parte del trabajo aborda la evaluación de resultados generados a partir de simulaciones de planificación de tareas de algoritmos de álgebra lineal densa sobre diversos tipos de configuraciones de procesadores. Concretamente se analizan algoritmos para la factorización de matrices densas, tales como Cholesky y QR. Finalmente, se obtienen conclusiones sobre los resultados de las simulaciones y se evalúa el factor de ahorro producido gracias al empleo de esta técnica.

En un futuro esta herramienta pretende incorporarse en entornos de ejecución (*runtimes*) de librerías de álgebra lineal tan conocidas como libflame o SuperMatrix, de modo que permita minimizar el consumo energético de sus rutinas.

Palabras clave

Álgebra lineal densa, ahorro de energía, DVFS, planificación, teoría de grafos,
grafo acíclico dirigido, computación de alto rendimiento, procesadores multinúcleo.

Agradecimientos

Hay que dar un sincero agradecimiento a las personas que han prestado su ayuda para que este trabajo haya podido hacerse realidad:

- Enrique S. Quintana-Ortí
- Pedro Alonso Jordá
- Rafael Mayo Gual
- Juan Carlos Fernández Fernández

Sin olvidar a mi familia que, en estos meses de intenso trabajo me han ayudado y mostrado su apoyo incondicional.

Gracias a todos.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción general

En este primer capítulo de presentación de la memoria, se introduce al lector en la temática de la tesis de máster, empezando con la sección de motivación, en la que se describe el interés del tema. En la segunda sección, estado del arte, se muestra el contexto científico-técnico en el que se encuentran los avances en materia relacionada con el ahorro de energía en álgebra lineal densa. Finalmente, se presentan los objetivos específicos del trabajo.

1.1. Motivación

Desde décadas, la computación de altas prestaciones ha concentrado sus esfuerzos en la optimización de algoritmos aplicados a la resolución de problemas complejos que aparecen en un amplio abanico de aplicaciones de casi todas las áreas científicas y tecnológicas. En particular, problemas de sistemas de ecuaciones lineales o problemas de mínimos cuadrados aparecen frecuentemente durante el análisis y el estudio del campo gravitatorio terrestre, la simulación del comportamiento de componentes estructurales en aviación o en la detección de enfermedades a partir de resonancias magnéticas. Para todos estos casos, la resolución de estos problemas supone la parte computacionalmente más costosa para la obtención de resultados.

Por esa razón, el principal objetivo de la computación de altas prestaciones es la optimización mediante en el uso de herramientas y técnicas, tales como la computación paralela, de problemas en ingeniería. En este contexto, el término optimización hace referencia a la reducción tiempo de ejecución, aunque también al espacio necesario para su cómputo. La mayoría de estos logros han sido posibles gracias al avance tecnológico de los componentes de los computadores impulsado principalmente por los fabricantes de hardware. Concretamente, las unidades centrales de procesamiento (CPU) o procesadores vienen doblando la velocidad y el número de transistores cada 18-24 meses en las últimas décadas. Desde 2004, la triple barrera del consumo energético, las limitaciones en el paralelismo de instrucción y la elevada latencia a memoria, ha provocado que el diseño de procesadores multinúcleo se haya convertido en la única vía para transformar el creciente número de transistores en un aumento del rendimiento.

En este sentido, el incremento de la velocidad de los procesadores así como el aumento del número de los núcleos en éstos ha sido un factor clave para el conseguir mejores prestaciones. No obstante, el aumento de la frecuencia de los procesadores junto con el uso de la tecnología multinúcleo ha implicado que el consumo energético necesario para su funcionamiento también haya crecido y se haya convertido, hoy en día, en un factor muy importante a tener en cuenta. La búsqueda de soluciones verdes o fuentes de energía alternativas que permitan reducir las emisiones de CO_2 a la atmósfera demuestran la creciente preocupación por el medio ambiente. En el ámbito de las tecnologías de la información y, más concretamente, en la computación de altas prestaciones, la comunidad cientifico-técnica actualmente muestra especial interés en el desarrollo de componentes, herramientas y técnicas que permitan minimizar el consumo energético. Medidas como FLOPS/Watt [1], *Energy-To-Solution* [2] o FTTSE [3] están empezando a tomar importancia cuando se evalúan las prestaciones de algoritmos y computadores: de hecho, se ha creado un *ranking* como el Green500 [4], análogo al Top500, que ya utiliza este tipo de métricas para comparar y clasificar los supercomputadores en el mundo.

Algunas herramientas de ahorro, basadas en la transición de la computadora a estados de bajo consumo o en la reducción de la frecuencia y el voltaje de forma dinámica (DVFS) en los procesadores, ofrecen la posibilidad de generar aplicaciones conscientes del consumo. Por ejemplo, sistemas autónomos de ahorro energético para grandes plataformas (clusters de computadores y supercomputadores) y en máquinas de sobremesa están empezándose a implantar con el objetivo de limitar el consumo y reducir los costes económicos generados tanto por las propias máquinas como por los sistemas de refrigeración.

En resumen, los algoritmos paralelos de álgebra lineal densa aplicados a problemas de ingeniería y las técnicas de ahorro de energía disponibles en procesadores mutinúcleo actuales son las dos vertientes que se aúnan en este trabajo. Concretamente, el objetivo del trabajo es realizar un estudio preliminar de un *planificador consciente del consumo en algoritmos de álgebra lineal densa sobre procesadores multinúcleo* mediante el uso de DVFS.

1.2. Estado del arte

La temática sobre el ahorro de energía está tomando, cada vez más, especial relevancia en la computación de altas prestaciones. Nuevas técnicas, herramientas, componentes y multitud de algoritmos intentan, de algún modo, reducir la energía consumida. En este sentido los problemas de álgebra lineal densa, necesitan, cada vez más, de potentes plataformas para aumentar su rendimiento y prestaciones. Estas plataformas, compuestas por una gran cantidad procesadores multinúcleo que operan a altas frecuencias, provocan grandes consumos energéticos lo que las convierte en opciones no deseadas a nivel económico en muchos casos. Por esta razón, centros de investigación, universidades y empresas dedican gran parte de sus esfuerzos a buscar nuevas soluciones y alternativas para desarrollar aplicaciones conscientes del consumo energético. En esta sección, se explican algunas de las técnicas, métodos y algoritmos utilizados en planificadores, en segundo lugar, se da a conocer la técnica de reducción de energía disponible en procesadores, basada en el escalado dinámico de la frecuencia y el voltaje (DVFS) y finalmente se comentan algunos trabajos desarrollados sobre planificación consciente del consumo.

1.2.1. Planificadores

El estudio de tareas paralelas y distribuidas se ha abordado en detalle con el objetivo de que los procesadores optimicen el tiempo y respeten las dependencias entre tareas cuando se ejecuta una aplicación.

Normalmente los algoritmos de planificación suelen clasificarse en dos categorías: estáticos y dinámicos. En los estáticos, la planificación y la asignación de recursos a las tareas se realiza antes de que las aplicaciones se ejecuten, realizándose el supuesto de partida de que se conoce a priori el coste de cada una de las tareas y las comunicaciones entre ellas. Además, asumen que las tareas ocupan el procesador asignado hasta que finalizan [5, 6]. Por otra parte, los algoritmos dinámicos planifican las tareas en tiempo de ejecución, aplicando técnicas de equilibrado de la carga.

Los planificadores por listas son los planificadores estáticos más conocidos. Estos poseen listas donde se ubican las tareas a ser ejecutadas, ordenadas a través de ciertas prioridades [7, 8]. En este trabajo se abordará un planificador estático que aplicará una determinada política, donde se conoce a priori la duración de las tareas obtenidas a partir del orden de coste teórico de las mismas.

1.2.2. Ahorro de energía: escalado dinámico de la frecuencia y el voltaje

El escalado dinámico de la frecuencia y el voltaje se ha convertido, hoy en día, en una característica prácticamente presente en todas de las nuevas generaciones de procesadores multinúcleo [9, 10]. La reducción de la frecuencia de reloj del procesador, y la consecuente reducción del voltaje necesario durante periodos ociosos o de baja demanda, dan como resultado final una importante reducción del consumo requerido. No obstante, hay que ser conscientes de que la reducción de la frecuencia tiene asociada un

aumento de los tiempos de ejecución. En [11] se definen los clusters DVFS, que son capaces de reducir la frecuencia del reloj en periodos de baja actividad. Actualmente existen numerosas técnicas de DVFS que pueden aplicarse en un amplio abanico de posibilidades, dentro del marco de la computación de altas prestaciones. Por ejemplo, en grandes centros de datos de alta producción y disponibilidad, para reducir el consumo del conjunto en total [12, 13].

En este ámbito, existen diferentes métodos que emplean como herramienta de ahorro de energía el escalado dinámico de la frecuencia y el voltaje, tales como:

- Análisis del grafo de dependencias (*Acyclic Directed Graph*, DAG) de aplicaciones científicas, donde se identifica el camino crítico, siendo posible reducir el consumo de aquellas tareas no críticas [14].
- Otras aplicaciones [15] dedicadas a trabajar en conjunto con el planificador del sistema operativo, para ajustar de forma dinámica en tiempo real la frecuencia de los procesadores.
- Técnicas de reducción de la frecuencia en aplicaciones paralelas durante los periodos de comunicación, como por ejemplo MPI [16, 17].
- Además de las aplicaciones paralelas, los planificadores de máquinas virtuales también tienen la posibilidad de utilizar DVFS [11].

En este trabajo se emplea DVFS con el propósito de realizar una planificación eficiente, a partir del grafo dirigido de dependencias, sobre aplicaciones y algoritmos asociados al álgebra lineal densa.

1.2.3. Planificación consciente del consumo

Si se busca el trabajo desarrollado relacionado con la planificación y el empleo de DVFS como herramienta de reducción de consumo, podremos observar que existen numerosas investigaciones y artículos sobre esta temática. Por ejemplo, en [18] se modela un planificador para clusters capaz de asignar tareas en tiempo real y capaz de regular la frecuencia de los procesadores en función de la carga de trabajos en un determinado momento; es decir, asignando tareas en periodos de no utilización, y constrastando finalmente los ahorros energéticos producidos.

En [19, 20] se discute cómo planificar tareas independientes con DVFS en un monoprocesador; en [21, 22] se emplea DVFS para planificar tareas con dependencias en múltiples procesadores; y en [23, 24, 25] se citan algunos algoritmos de planificación a tiempo real para tareas dependientes. En [26] se presenta una plataforma que integra la asignación de tareas a tiempo real, utilizando DVFS para minimizar el consumo en tareas con dependencias obteniendo resultados sobre problemas de programación lineal entera. LPHM [27] es un planificador dinámico que intenta maximizar el tiempo de las tareas no críticas mediante DVFS.

En [28] se proponen heurísticas para un planificador consciente del consumo de tareas paralelas en entornos de clusters heterogéneos. En [29] se emplea una estrategia, aplicada sobre clusters, basada en la reducción de las holguras de las tareas no críticas.

La intención de este trabajo es realizar una planificación consciente del consumo a través del uso de DVFS, basada en la misma idea que se comenta en párrafos anteriores. La reducción de las holguras entre tareas no críticas del grafo de dependencias puede conseguir importantes ahorros de energía, sin perjudicar el rendimiento de los algoritmos ejecutados. El objetivo, por tanto, es determinar hasta qué punto una tarea no crítica puede ralentizarse, y plasmar el resultado de esta investigación en forma de un *algoritmo de extensión de tiempos de tareas*. Como segundo objetivo, el algoritmo de extensión de tareas que se diseña se evaluará mediante un simulador, desarrollado también en el marco de esta tesis. En nuestro caso, el estudio hará especial hincapié en algoritmos comúnmente utilizados en álgebra lineal densa, tales como la descomposición en factores de la matriz densa ligada a un sistema de ecuaciones lineales a través de métodos de Cholesky, LU, QR o LDL^T.

1.3. Objetivos

La intención de nuestro trabajo es realizar una planificación consciente del consumo a través del uso de DVFS en los procesadores donde se ubiquen las tareas no críticas del grafo de dependencias entre tareas que representa las tareas en las que se subdivide un algoritmo paralelo de álgebra lineal densa. Por lo tanto, los objetivos concretos del trabajo son los siguientes:

- Búsqueda de información, técnicas y métodos e investigación sobre trabajos relacionados que puedan servir como referencia para nuestro objetivo. Palabras clave para la búsqueda: algoritmos de álgebra lineal densa, técnicas de ahorro de energía en procesadores multinúcleo, teoría de grafos y métodos de administración y planificación de tareas.

- Diseño y elección de un sistema de representación de grafos flexible, cómodo y fácil de tratar y visualizar. La elección de este sistema repercutirá en un futuro en la forma de manejar las dependencias entre tareas, por lo que la decisión tomada repercutirá en la implementación realizada del algoritmo de extensión de holguras y simulador.

- Búsqueda de algoritmos de álgebra lineal a bloques que puedan descomponerse en subtareas. Al mismo tiempo se intentarán elaborar métodos de generación automática de grafos a partir de algoritmos básicos. Este banco de pruebas permitirá, una vez implementado el planificador consciente del consumo, evaluar su rendimiento.

- Diseño e implementación de un algoritmo que permita, a través del grafo de dependencias de un algoritmo de álgebra lineal densa, analizarlo, detectar las dependencias entre tareas y determinar para cuales de ellas puede extenderse su duración en función de un rango de frecuencias discreto como parámetro de entrada. Este método deberá devolver, en forma de grafo, la frecuencia mínima a la que debe ejecutarse cada tarea para que el algoritmo de entrada no pierda prestaciones cuando sea ejecutado.

- Diseño e implementación de un planificador a modo de simulador que permita planificar las tareas del algoritmo anterior, permitiendo diferentes modos de planificación y configuración del número de procesadores multinúcleo que se emplearán. Finalmente esta traza de simulación deberá devolver estadísticas y porcentajes de tiempo, para cada procesador, a los que ha estado trabajando en cada frecuencia. Estos resultados servirán para verificar y evaluar el comportamiento de la herramienta implementada.

- Evaluación del planificador implementado mediante diferentes algoritmos de álgebra lineal densa. Principalmente se realizarán experimentos con algoritmos por bloques de descomposiciones de matrices cuadradas y densas. Estos algoritmos son:

  - Cholesky por bloques.
  - QR por bloques.
  - QR por bloques de columnas (tareas del mismo tipo con diferente coste).

- Conclusiones sobre los resultados de ahorro obtenidos con los algoritmos de álgebra lineal densa escogidos y discusión de trabajos futuros relacionados, que darán pie a la tesis doctoral en esta linea de investigación.

Esta tesis de máster, como ya se ha comentado, servirá como base para futuros trabajos orientados a las líneas de computación de altas prestaciones y al ahorro de energía sobre procesadores multinúcleo. Al mismo tiempo, la tesis introducirá la temática de la tesis doctoral que se pretende llevar a cabo en los próximos años.

1.3.1. Limitaciones del trabajo

A primera vista, la implementación de un simulador de planificación de tareas consciente del consumo parece sencilla. Sin embargo, basta consultar en la temática y teoría de planificación de tareas para comprobar que una planificación óptima de tareas, salvo en condiciones muy particulares, es un problema NP-completo. Si además se añade la posibilidad de que cada una de estas tareas puede ejecutarse a diferentes frecuencias y, por lo tanto, cambiar su duración, la complejidad del trabajo aumenta aún más.

En esta tesis, al abordarse una aproximación teórica, que servirá como base hacia la futura tesis doctoral, se ha decidido acotar el estudio a una serie de casos particulares. Estas limitaciones son las siguientes:

- Uno de los objetivos principales de este trabajo es implantar un planificador consciente del consumo en algún runtime de librerías de computación numérica, como libflame o SuperMatrix. En este trabajo sólo se implementará un planificador a modo de simulador que permita obtener resultados y estadísticas del ahorro producido, sin llegar a se incorporado en núcleos de estas librerías.
- El coste de realizar un cambio de frecuencia del procesador aplicando DVFS no es despreciable sino que depende del cambio de frecuencia realizado, es decir, de la frecuencia actual y la frecuencia destino. En este trabajo se asume un coste constante de cambio de frecuencia (actual y destino) para el rango de frecuencias de procesador aceptadas.
- Cuando se cambia la frecuencia del procesador, se asume que la frecuencia únicamente puede cambiarse a nivel de procesador o socket y no a nivel de núcleo. Por este motivo, cuando se cambie la frecuencia en un procesador, todos los núcleos asociados a éste adoptarán la nueva frecuencia después del tiempo de cambio de frecuencia.
- El algoritmo de expansión de tiempos de tareas asume que los recursos en la plataforma donde se simulará la planificación son infinitos.
- Con el objetivo de limitar el número de combinaciones posibles de planificación se asume que únicamente se puede cambiar la frecuencia de procesador cuando todos los núcleos del mismo están libres. Es decir, no se podrá cambiar la frecuencia del procesador durante la ejecución de una tarea en cualquiera de los núcleos.
- En una situación de simulación real, se recogen datos de tiempo reales de la ejecución de las tareas en cada una de las frecuencias permitidas por el procesador. En este trabajo se asume que el coste temporal del las tareas crece de forma inversamente proporcional a la frecuencia del procesador.

1.4. Organización de la memoria

La memoria de la tesis de máster se estructura en los siguientes capítulos:

- **Aproximación teórica**

  En este Capítulo 2 se revisan los fundamentos teóricos de algunos de los métodos y técnicas de revisión y evaluación de programas aplicados a la planificación de proyectos de ingeniería. A continuación, se evalúa el uso de esta técnica en algoritmos de álgebra lineal densa. Finalmente se definen una serie de restricciones y simplificaciones que se adoptan para limitar el número de casos posibles de estudio en esta tesis de máster.

- **Implementación**

  En el Capítulo 3 se describen los módulos y el planificador implementado para simular las ejecuciones de las tareas a partir del algoritmo de ajuste de holguras explicado en el capítulo anterior. A continuación, se explicarán las políticas que se han tenido en cuenta y se probará, con ejemplos y a modo explicativo, para analizar cómo se ejecuta la simulación.

- **Evaluación de algoritmos**

  En el Capítulo de 4 se evaluarán, mediante los grafos de dependencias asociados, diversos algoritmos de álgebra lineal densa, tales como Cholesky, QR y QR por bloques de columnas. Al mismo tiempo se presentan las conclusiones sobre los ahorros de energía producidos y, a través de estadísticas, se demuestra la factibilidad del algoritmo de ajuste de holguras implementado.

- **Conclusiones**

  En el Capítulo 5 se hace una recopilación de las ideas destacadas, tratando de enfatizar los aspectos más interesantes, los conceptos aprendidos y la aportación en la temática de ahorro de energía al álgebra lineal densa.

- **Futuras extensiones**

  En el Capítulo 6 se exponen las posibles mejoras o extensiones que pueden aplicarse al simulador. En particular, se comentan las intenciones y las posibles aplicaciones en un futuro no lejano de la viabilidad en la aplicación de esta técnica en librerías de cálculo de altas prestaciones.

Capítulo 2

Aproximación teórica

En este capítulo se abordan los fundamentos teóricos del algoritmo de reducción de holguras implementado en este trabajo. Fundamentalmente se hace hincapié en las técnicas de análisis y planificación de proyectos. En la segunda parte se ilustra, a través de ejemplos, cómo estos conceptos pueden aplicarse a algoritmos de álgebra lineal densa.

2.1. Introducción a la técnica PERT/CPM

Los métodos de PERT (*Program Evaluation and Review Technique*) y CPM (*Critical Path Method*), en el ámbito de administración y planificación de proyectos, son técnicas utilizadas en aquellos proyectos que necesitan un período prologando para su ejecución, que están compuestos por un conjunto de actividades elementales que deben realizarse en un determinado orden, y cuya fecha de inicio y tiempo de ejecución condiciona el inicio de otra actividad [30, 31]. Sus objetivos fundamentales son determinar la duración del proyecto y tratar de establecer las actividades denominadas *críticas*, es decir, aquellas que han de estar sujetas a un mayor control, pues si se retrasan provocan aumento del tiempo total de ejecución del proyecto.

En concreto, para el caso de estudio de este trabajo se utiliza el método CPM, puesto que éste asume que los tiempos de las tareas se conocen a priori, lo cual se adapta a las restricciones definidas en los objetivos. Es decir, se definen aproximadamente y a priori los tiempos de las tareas de los algoritmos de álgebra lineal densa a partir de orden del coste. El método PERT por contra, utiliza tiempos probabilísticos, por lo que no se adapta a las especificaciones.

2.1.1. Construcción del grafo

Elementos del grafo

Los grafos CPM sirven para representar tanto la duración de las tareas como las precedencias entre ellas. Estos grafos se caracterizan por ser ponderados, acíclicos y dirigidos. Los elementos principales son los siguientes:

- **Nodos:** Representan sucesos temporales o inicios o finales de actividades (el final de la actividad precedente y el inicio de la siguiente actividad). Por este motivo, un nodo no consume tiempo ni recursos. En estos nodos se inscribe la etiqueta correspondiente.

- **Arcos:** Representan la ejecución real de las actividades, por lo que sí consumen tiempo y recursos. Se representan mediante una arista dirigida. Sus etiquetas representan el nombre de la actividad o tarea y habitualmente tienen su coste representado entre paréntesis.

- **Camino:** Conjunto de actividades sucesivas conectadas mediante sucesos, con sendos sucesos únicos de inicio y final. Entre estos caminos, existen caminos alternativos o no críticos, y caminos críticos.

En la Figura 2.1 se representa un grafo de dependencias empleando el método CPM y en la Tabla 2.1 la representación a modo de tabla del mismo.

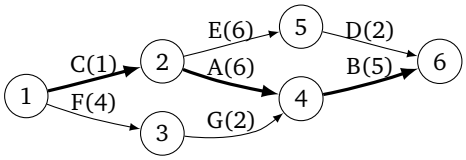


Figura 2.1: Ejemplo de grafo de dependencias empleando el método CPM.

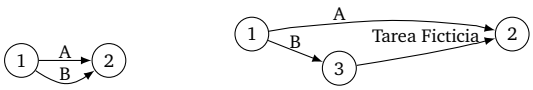
Actividades	Act. precedentes	Coste
A	C	6
B	A,G	5
C	-	1
D	E	2
F	-	4
G	F	2

Tabla 2.1: Tabla de precedencias y pesos del grafo del ejemplo de la Figura 2.1.

Principios de la construcción del grafo

Durante la construcción del grafo CPM cabe tener presentes una serie de reglas que permitirán la corrección del mismo. Estas de reglas son:

- **Designación sucesiva:** Los nodos se numeran ordenadamente, de manera que no se numere un nodo si no se han numerado antes todos los nodos de los que salen arcos hacia él.
- **Unidad del estado inicial y final:** Sólo puede existir un nodo inicial y otro final.
- **Designación unívoca:** No pueden existir dos o más aristas que tengan el mismo nodo de origen y tengan también el mismo nodo destino. Cada arista o arco presenta una única actividad, y los nodos inicial y final siempre son diferentes. Si sucediera lo contrario, se debería recurrir a las denominadas **actividades ficticias** que, al fin y al cabo, son actividades nulas, es decir con duración cero. Véase la Figura 2.2.



(a) Incorrecto (b) Correcto

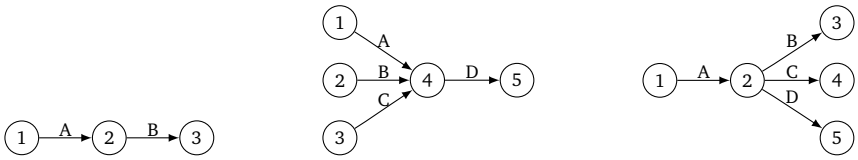
Figura 2.2: Designación unívoca de tareas.

Tipos de relaciones de precedencia

Cuando se construye el grafo a partir de la tabla de precedencias, es posible encontrar algunos casos especiales que es necesario tener en cuenta. Especialmente existen tres tipos de relaciones de precedencia:

- **Relaciones lineales:** Es decir, para iniciar la siguiente actividad tiene que haber finalizado la actividad precedente anterior. En el grafo de la Figura 2.3(a), para iniciar B, A tiene que haber finalizado. El suceso 2 es el suceso final de A y suceso inicio de B.
- **Relaciones convergentes:** Estas relaciones son las que se generan cuando la siguiente actividad depende de que haya finalizado más de una actividad. En el grafo de la Figura 2.3(b), para iniciar D, es necesario haber finalizado las actividades A, B y C.

- **Relaciones divergentes:** Estas relaciones son las que se generan cuando más de una actividad depende de que la actividad precedente haya finalizado. En el grafo de la Figura 2.3(c), para poder iniciar cualquiera de las actividades B, C o D, es necesario haber finalizado A.



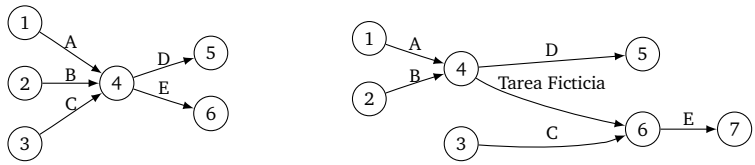
(a) Lineal (b) Convergente (c) Divergente

Figura 2.3: Ejemplos de tipos de relaciones de precedencia.

Ejemplo de conflicto y solución

Cuando se construye el grafo es posible encontrar situaciones de conflicto, que deben solucionarse mediante las denominadas actividades **nulas** o **ficticias**. Para ilustrar la situación, supongamos las siguientes relaciones:

1. Las actividades A y B preceden a la actividad D.
2. Las actividades A, B y C preceden a la actividad E.



(a) Situación de conflicto (b) Solución del conflicto

Figura 2.4: Conflicto y solución al conflicto en un grafo CPM.

En 2.4(a) se puede ver que la segunda relación se cumple pero no la primera puesto que, según el grafo, es necesario finalizar C para comenzar D, cuando no es así. Como solución al conflicto, en la Figura 2.4(b) se propone el grafo a través de la inserción de una tarea ficticia.

2.1.2. Tiempos

Para aplicar la técnica de CPM, es necesario planificar previamente las actividades que integran el proyecto y estimar sus duraciones. La duración total del proyecto no coincidirá con la suma de los tiempos del conjunto de las actividades, ya que algunas de ellas podrán ejecutarse en paralelo. El tiempo de realización total viene fijado por el tiempo que transcurre entre la ejecución del suceso inicial o nodo inicial y el nodo final. Es el tiempo que supone necesariamente la realización de todas las actividades intermedias.

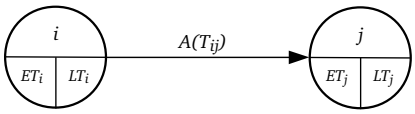


Figura 2.5: Nomenclatura de tiempos CPM.

Para conocer la duración de un proyecto es necesario calcular los denominados tiempos de comienzo tempranos de los nodos que forman el grafo CPM, siendo representados como se

- T_{ij}: duración de la actividad que comienza en el suceso i y finaliza en el j.
- ET_i: tiempo más temprano para comenzar A, donde:

$$ET_0 = 0 \quad \text{para el suceso inicial 0.} \tag{2.1}$$

- ET_j: tiempo más temprano para finalizar A, donde:

$$ET_j = \text{máx}(ET_i + T_{ij}) \quad : \quad \forall i \in \{\text{sucesos } i \text{ con tareas que llegan al suceso } j\}. \tag{2.2}$$

- LT_i: tiempo más tardío para comenzar A, donde:

$$LT_i = \text{mín}(LT_j - T_{ij}) \quad : \quad \forall j \in \{\text{sucesos } j \text{ con tareas que parten del suceso } i\}. \tag{2.3}$$

- LT_j: tiempo más tardío para finalizar A. En este caso,

$$LT_f = ET_f \quad \text{para el suceso final } f. \tag{2.4}$$

Algoritmo de CPM para el cálculo de tiempos tempranos y tardíos

Para calcular los tiempos ET y LT se deben seguir los siguientes pasos:

1. Los tiempos ET se asignan mediante la ecuación (2.2) realizando un recorrido hacia delante, o desde el nodo inicial al final en el grafo de dependencias. El tiempo temprano de comienzo del nodo raíz es siempre 0.
2. Los tiempos LT se asignan mediante la ecuación (2.3) realizando un recorrido hacia atrás, o desde el nodo final al inicial en el grafo de dependencias. El tiempo tardío de comienzo del nodo final será igual al valor de su ET.

2.1.3. Holguras

La holgura es la diferencia entre el tiempo temprano y el tardío, y puede indicar un posible exceso de recursos. Cuando la holgura es cero el suceso se denomina crítico. Por esta razón los nodos iniciales y finales siempre son críticos.

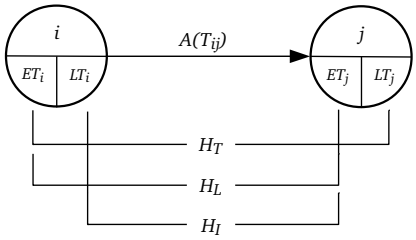


Figura 2.6: Tipos de holguras en CPM.

Siguiendo la Figura 2.6 existen los siguientes tipos de holguras:

- **Holgura del suceso i (H_i):** Es el tiempo que se puede retrasar su ejecución de forma que no se aumente la duración total. Cuando $H_i = 0$, entonces el suceso i es crítico. Se obtiene como:

$$H_i = LT_i - ET_i \tag{2.5}$$

- **Holgura total (H_T):** Es el tiempo máximo que puede retrasarse la ejecución de la actividad sin que esto afecte al tiempo de finalización, o también es el margen de tiempo excedente, suponiendo que a la situación representada por el nodo origen i se llegue lo más pronto posible y al destino j lo más tarde posible:

$$H_T = LT_j - ET_i - T_{ij} \tag{2.6}$$

  El camino crítico es la secuencia de actividades con holgura total cero, y puede que exista más de uno. La holgura total es menos restrictiva que la holgura libre. La holgura total sólo hace referencia al retraso del proyecto, pero no a la posibilidad de retrasar el inicio de una tarea sucesora.

- **Holgura libre (H_L):** Es el margen de tiempo excedente, suponiendo que al nodo de origen se llega lo más pronto posible y al de destino también lo más pronto posible. En otras palabras, sólo puede aprovecharse en las actividades precedentes o en la propia actividad:

$$H_L = ET_j - ET_i - T_{ij} \tag{2.7}$$

- **Holgura independiente (H_I):** Es el margen de tiempo excedente suponiendo que al nodo de origen se llega lo más tarde que es admisible y que al destino se llega lo más pronto posible. En otras palabras, si no se alarga la propia actividad, la holgura no se aprovecha:

$$H_I = ET_j - LT_i - T_{ij} \tag{2.8}$$

  La holgura independiente puede llegar a ser negativa, reflejando en este caso la falta de tiempo.

Para todos los casos, se puede comprobar que:

$$H_I \leq H_L \leq H_T. \tag{2.9}$$

Camino crítico

El camino crítico, como ya se ha comentado en secciones anteriores, es el que se forma al unir todas las actividades o tareas críticas, y que va desde el suceso inicial al suceso final. Aplicando el concepto de holgura, se pueden definir como actividades críticas aquellas cuyas holguras total H_T, libre H_L e independiente H_I son cero. Cualquier retraso o exceso de tiempo que sufra alguna de las tareas del camino crítico, implica un exceso en la duración del proyecto o, en el caso de este trabajo, un mayor tiempo de ejecución del algoritmo de álgebra lineal densa.

2.2. Aplicación al álgebra lineal densa

En esta sección se analiza la viabilidad sobre la aplicación de estos métodos a los grafos acíclicos de dependencias generados por los algoritmos de álgebra lineal densa. La estrategia de detectar holguras para minimizarlas lo máximo posible mediante la aplicación de DVFS tiene como objetivo generar ahorros de energía.

Para verificar y modelar un algoritmo capaz de extender las holguras de las tareas no críticas de un algoritmo de álgebra lineal densa se emplea la descomposición de Cholesky por bloques.

2.2.1. Caso de estudio: Descomposición de Cholesky

Fundamentos teóricos

En álgebra lineal, la descomposición de Cholesky es una factorización que descompone una matriz simétrica y definida positiva en el producto de una matriz triangular inferior por la traspuesta de ésta. El factor triangular inferior se denomina triángulo de Cholesky de la matriz original positiva definida. La

factorización de Cholesky se emplea habitualmente para resolver sistemas lineales de ecuaciones cuando la matriz de coeficientes es simétrica y positiva definida (s.p.d.) [32].

En general, si A es una matriz de $n \times n$ s.p.d., entonces A puede ser descompuesta como

$$A = LL^T,$$

donde L es una matriz de tamaño $n \times n$ triangular inferior con entradas diagonales estrictamente positivas, y L^T representa la traspuesta de L.

La descomposición de Cholesky es única: dada una matriz s.p.d. A, existe una única matriz triangular inferior L con entradas diagonales estrictamente positivas tales que $A = LL^T$.

Algoritmo básico

El algoritmo de Cholesky puede considerarse como una variante de la descomposición LU. En particular, se considera A y L como:

$$A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right), \quad L = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right),$$

donde α_{11} y λ_{11} son escalares. Como resultado, a_{21} y l_{21} son vectores de longitud $n-1$ y A_{22} y L_{22} son matrices de tamaño $(n-1) \times (n-1)$. El símbolo $\star$ indica la parte simétrica de A, que no se referencia durante el algoritmo. Entonces se define:

$$A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right) \left(\begin{array}{c|c} \lambda_{11} & l_{21}^T \\ \hline 0 & L_{22}^T \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11}^2 & \star \\ \hline \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{array}\right)$$

A partir de la igualdad anterior se obtienen las siguientes ecuaciones:

$$\lambda_{11} = \sqrt{\alpha 11}, \tag{2.10}$$

$$l_{21} = a_{21}/\lambda_{11}, \tag{2.11}$$

$$A_{22} - l_{21} l_{21}^T = L_{22} L_{22}^T. \tag{2.12}$$

Por lo tanto, el algoritmo de Cholesky puede definirse del siguiente modo:

1. Particionar $A = \left(\begin{array}{c|c} \alpha_{11} & \star \\ \hline a_{21} & A_{22} \end{array}\right)$.
2. $\alpha_{11} \leftarrow \lambda_{11} = \sqrt{\alpha 11}$.
3. $a_{21} \leftarrow l_{21} = a_{21}/\lambda_{11}$.
4. $A_{22} \leftarrow A_{22} - l_{21} l_{21}^T = L_{22} L_{22}^T$.
5. Continuar recursivamente con el cómputo de A_{22}.

El algoritmo básico se muestra en el Código 2.1. En el código, $A(r,s)$ referencia al elemento (r,s) de la matriz A. Se considera que la matriz está en compuesta de $n \times n$ elementos.

```
function Chokesky(A)
  for( k = 1; k<=n; k++ ) {
    A (k,k) = sqrt (A (k,k)) ; % POTRF (Cholesky)
    for( i = k+1; i<=n; i++ ) {
      A (i,k) = A (i,k) / A (k,k) ; % TRSM
    }
    for( i = k+1; i<=n; i++ ) {
      for( j = k+1; j<=i-1; j++ ) {
        A (i,j) = A (i,j) - A (i,k) * A (j,k)' ; % GEMM
      }
      A(i,i) = A(i,i) - A(i,k) * A(i,k)' ; % SYRK
    }
  }
```

Código 2.1: Algoritmo para la factorización de Cholesky.

El coste del algoritmo básico en una matriz simétrica de rango n es de $O(n^3/3)$ operaciones aritméticas y requiere un espacio de $O(n^2)$ datos.

Algoritmo por bloques

Los denominados *algoritmos por bloques* son algoritmos dividen la matriz en bloques (submatrices) para operar con éstos con el objetivo de agrupar los accesos a memoria, incrementando la probabilidad de que los datos se encuentren en los niveles de memoria más cercanos al procesador, y por tanto resultar más rápidos. En definitiva, los algoritmos por bloques extraen un mayor beneficio de la arquitectura jerarquizada del sistema de memoria. Como detalle adicional, el tamaño óptimo de los bloques en este tipo de algoritmos es dependiente de la arquitectura, especialmente del tamaño de caché.

Para el caso de la factorización de Cholesky, si se consideran A y L como:

$$A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right), \quad L = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right),$$

donde A_{11} y L_{11} son matrices de tamaño $b \times b$, se puede establecer la siguiente correspondencia:

$$A = \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} L_{11}^2 & \star \\ \hline L_{11}L_{21} & L_{21}L_{21}^T + L_{22}L_{22}^T \end{array}\right).$$

A partir de la igualdad anterior se obtienen las siguientes ecuaciones:

$$A_{11} = L_{11}L_{11}^T, \tag{2.13}$$

$$L_{21}L_{11}^T = A_{21}, \tag{2.14}$$

$$A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T. \tag{2.15}$$

Por lo tanto, el algoritmo por bloques de Cholesky se define del siguiente modo:

1. Particionar $\left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right)$ donde A_{11} es una matriz de tamaño $b \times b$.
2. $A_{11} \leftarrow L_{11} = Chol(A_{11})$.
3. $A_{21} \leftarrow L_{21} = A_{21}L_{11}^{-T}$.
4. $A_{22} \leftarrow A_{22} - L_{21}L_{21}^T$.
5. continuar recursivamente con el cómputo de A_{22}.

Implementación a bloques

El algoritmo básico de Cholesky por bloques se muestra en el Código 2.2. En el código, $A(r,s)$ referencia al bloque (r,s) de la matriz A. Se considera que la matriz está en compuesta de $b \times b$ bloques, numerados del 1 al b.

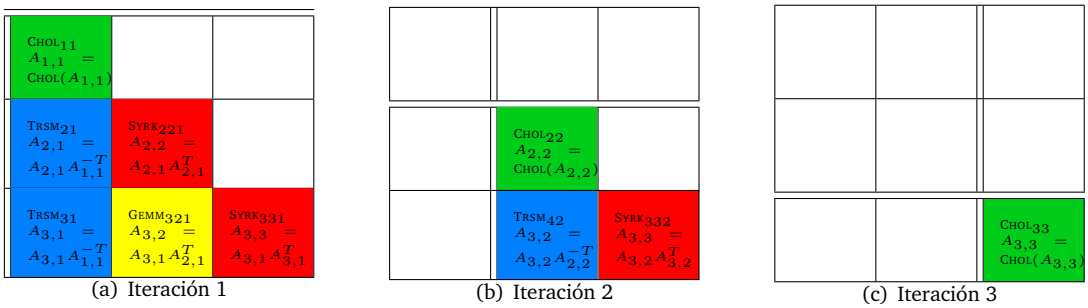
```
1  function CholeskyBlocks(A)
2    for( k = 1; k<=b; k++ ) {
3      A (k,k) = chol (A (k,k)) ;% POTRF (Cholesky)
4      for( i = k+1; i<=b; i++ ) {
5        A (i,k) = A (i,k) / A (k,k) ; % TRSM
6      }
7      for( i = k+1; i<=b; i++ ) {
8        for( j = k+1; j<=i-1; j++ ) {
9          A (i,j) = A (i,j) - A (i,k) * A (j,k)' ; % GEMM
10       }
11       A(i,i) = A(i,i) - A(i,k) * A(i,k)' ; % SYRK
12     }
13   }
```

Código 2.2: Algoritmo por bloques para la factorización de Cholesky.

El algoritmo tiene cuatro tipo de tareas que pueden realizarse utilizando rutinas de las librerías de álgebra lineal densa BLAS y LAPACK:

- `POTRF`: Es la rutina de LAPACK encargada de calcular la factorización de Cholesky de una matriz, en el caso de estudio, de una submatriz o bloque.
- `TRSM`: Es la rutina de BLAS encargada de resolver el siguiente sistema lineal:
  `op( A )*X = alpha*B` o `X*op( A ) = alpha*B`,
  donde `A` es la matriz triangular de coeficientes y `B` contiene los términos independientes.
- `GEMM`: Es la rutina de BLAS encargada de calcular el producto de dos submatrices:
  `C := alpha*op( A )*op( B ) + beta*C`.
- `SYRK`: Es la rutina de BLAS encargada de calcular el producto de una matriz por su traspuesta:
  `C := alpha*A*A' + beta*C`;
  actualizando únicamente la mitad triangular inferior o superior de `C` en el resultado.

Hay que destacar que el número de bloques en la implementación anterior viene dado por b. Un ejemplo de ejecución a bloques para $b = 3$ del algoritmo anterior se muestra en la Figura 2.7 [33]:



(a) Iteración 1 (b) Iteración 2 (c) Iteración 3

Figura 2.7: Ejemplo de ejecución de algoritmo de Cholesky de 3×3 bloques.

A partir de la Figura 2.7 se obtiene el grafo de dependencias mostrado en la Figura 2.8.

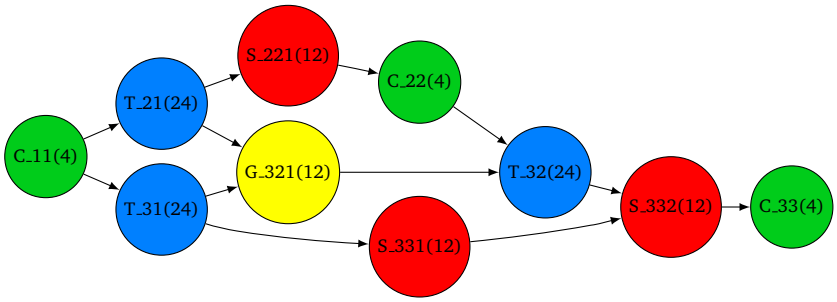


Figura 2.8: Grafo de dependencias entre tareas de la factorización de Cholesky de 3×3 bloques.

La nomenclatura de los nodos es la siguiente: `TAREA_ID(Coste)`. Como ejemplo, la tarea `C_11(4)` representa el cómputo de Cholesky del bloque (1,1) y tiene coste 4. En las tareas con tres números en el identificador, los dos primeros números representan el identificador del bloque mientras que el tercero identifica el número de iteración.

En lo que resta del trabajo, se asume que el coste de un mismo tipo de tarea es el mismo en todos los casos y está indicado en función del orden de coste temporal del algoritmo que se computa en cada una de estas tareas. Los tiempos se medirán a partir de ahora mediante unidades temporales (u.t.). En

una situación realista estos costes deberían reemplazarse por medidas de tiempo reales. Para el caso de ejemplo hemos definido los costes de las tareas como se indica en la Tabla 2.2.

Tarea	Orden de coste	Coste (u.t.)
CHOL	$O(n^3/3)$	4
TRSM	$O(n^3)$	12
SYRK	$O(n^3)$	12
GEMM	$O(2n^3)$	24

Tabla 2.2: Tabla de orden de costes y costes en u.t. de las tareas de la factorización de Cholesky.

2.2.2. Análisis del grafo mediante CPM

El interés de este apartado es poder analizar el grafo de dependencias entre tareas obtenido a partir de la descomposición de Cholesky por bloques expuesta como ejemplo en apartados anteriores.

Conversión del grafo: de nodos-tarea a arcos-tarea

Como se observa, el grafo de Cholesky de la Figura 2.8 presenta las tareas ligadas a los nodos. Sin embargo, para emplear la técnica de planificación CPM, es necesario convertir el grafo original de la factorización con las tareas ubicadas en los nodos al grafo destino con las tareas ubicadas en los arcos. Esta conversión puede realizarse aplicando el siguiente algoritmo:

```
paracada nodo en grafo
      si arcos_incidentes != 1
      {
         Crear un nuevo nodo
         Reasignar los arcos incidentes del antiguo nodo al nuevo
         Crear un nuevo arco desde el nuevo nodo al nodo antiguo
         Asignar coste 0 al nuevo nodo
      }
      finsi
fin paracada

paracada arcos en graph
         asignar el nombre de la tarea y coste del nodo destino al arco
fin paracada

renombrar nodos
```

El grafo obtenido tras realizar la conversión del recogido en la Figura 2.8 se refleja en la Figura 2.9.

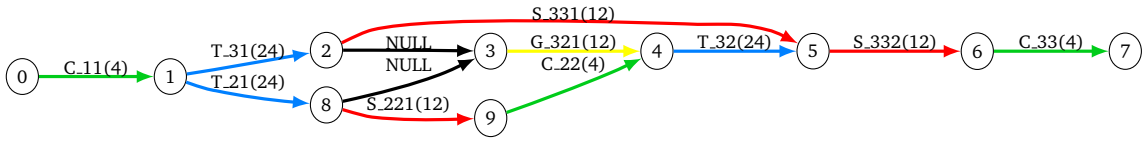


Figura 2.9: Grafo de dependencias con arcos-tarea de la factorización de Cholesky de 3×3 bloques.

Las tareas con etiqueta *NULL* son tareas ficticias, y se introducen para realizar la conversión del grafo de nodos-tarea a arcos-tarea.

Aplicación de la técnica CPM al grafo de dependencias de Cholesky

Una vez obtenido el grafo de dependencias basado en arcos-tarea, se puede aplicar la técnica de CPM para detectar las holguras entre tareas. En este sentido, se intentará obtener cuáles son las tareas no críticas que pertenecen a un mismo camino para extender su duración y eliminar, en la medida de lo posible, la holgura entre ellas.

En la Tabla 2.3 se muestran los resultados de aplicar la técnica CPM en el caso de ejemplo estudiado en esta sección: la factorización de Cholesky de 3×3 bloques. Las filas en gris representan las tareas críticas para las que la holgura del nodo, la holgura total, libre e independiente son nulas. Es decir, $H_i = H_T = H_L = H_I = 0$.

Tarea	i-j	ET_i	LT_j	$T_{i,j}$	H_i	H_j	H_T	H_L	H_I
C_11	0-1	0	4	4	0	0	0	0	0
T_21	1-8	4	28	24	0	0	0	0	0
T_31	1-2	4	32	24	0	4	4	0	0
NULL	2-3	28	32	0	4	4	4	0	-4
S_331	2-5	28	68	12	4	0	28	28	24
G_321	3-4	28	44	12	4	0	4	4	0
T_32	4-5	44	68	24	0	0	0	0	0
S_332	5-6	68	80	12	0	0	0	0	0
C_33	6-7	80	84	4	0	0	0	0	0
S_221	8-9	28	40	12	0	0	0	0	0
NULL	8-3	28	32	0	0	4	4	0	0
C_22	9-4	40	44	4	0	0	0	0	0

Tabla 2.3: Tabla de holguras del grafo de dependencias asociado a la factorización de Cholesky de 3×3 bloques.

Por tanto, el camino crítico está formado por la sucesión de nodos $\{0, 1, 8, 9, 4, 5, 6, 7\}$ o tareas {C_11, T_21, S_221, C_22, T_32, S_332, C_33} y tiene un coste de 84 u.t. Es lógico, por tanto, que el resto de tareas no críticas puedan ser alargadas en tiempo aplicando la técnica DVFS para reducir energía.

Como conclusión de este estudio previo, puede destacarse que la técnica CPM puede ser aplicada a grafos de dependencias en algoritmos de álgebra lineal densa. Esta técnica permitirá aplicar políticas de ahorro a través de la reducción de las holguras mediante el uso de DVFS. En el Capítulo 3 se presenta la explicación del *algoritmo expansión de tiempos de tareas* como objetivo de este trabajo. Posteriormente, se realizará una simulación de planificación a través de la cual se evalúan los resultados.

Capítulo 3

Descripción del simulador

ESTE capítulo abordará la descripción en detalle de los módulos implementados y fases que forman el planificador de tareas consciente del consumo. Como primer paso, se ofrecerá una visión general de la aplicación desarrollada, y en segundo lugar se explicarán los elementos que la componen.

3.1. Visión general

En esta sección se presenta una visión general de los componentes en los que se basa el simulador implementado, que serán los que se ilustren en detalle a lo largo de este capítulo. Un esquema general del simulador puede observarse en la Figura 3.1.

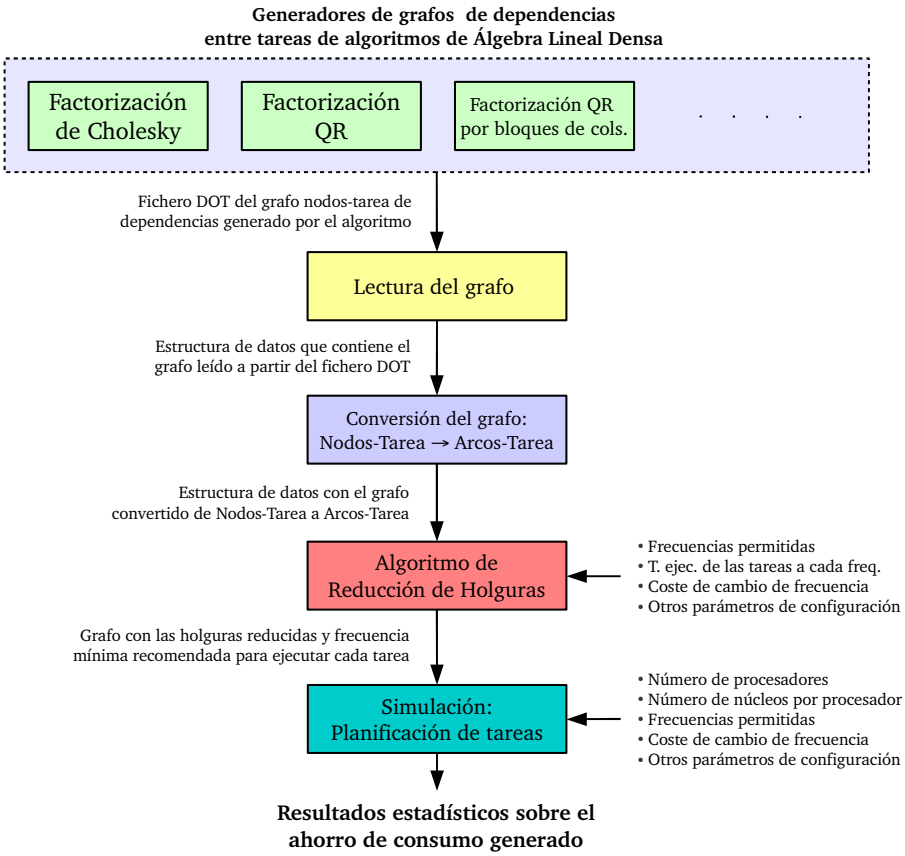


Figura 3.1: Esquema general del simulador implementado.

El proceso de simulación se realiza en cinco fases principales:

1. **Generador de grafos:** A partir de los algoritmos originales de álgebra lineal densa que se deseen estudiar se implementan programas análogos capaces de crear de forma automática el grafo de dependencias entre tareas en formato DOT.

2. **Lectura del grafo en formato DOT:** Emplea una de las funciones de lectura de la librería Networkx de Python para leer el fichero del grafo generado en el paso anterior.

3. **Conversión del grafo:** Realiza una conversión del grafo nodos-tarea obtenido a arcos-tarea, a partir de un módulo implementado, de modo que se puedan aplicar las técnicas estudiadas en el Capítulo 2.

4. **Reducción de holguras:** Aplica el algoritmo de reducción de holguras diseñado para esta tesis de máster. La aplicación de este algoritmo sobre el grafo de dependencias arcos-tarea, junto con las frecuencias de procesador permitidas, los costes de cambio de frecuencia, los tiempos de ejecución de cada tipo de tarea a cada frecuencia (y otros parámetros), generará otro grafo idéntico al cual se le han añadido atributos a los arcos. Entre estos atributos destacan la frecuencia mínima permitida a la que se debe ejecutar cada una de las tareas para que el tiempo de ejecución total se vea afectado lo menos posible.

5. **Simulación de planificación de tareas:** Ejecuta el simulador de planificación implementado en el trabajo. Este simulador, a partir del grafo de dependencias con los nuevos atributos obtenido en el paso anterior e información relativa a la plataforma (número de procesadores, núcleos por procesador, etc.), genera una planificación de las tareas consciente de las dependencias y ajusta la frecuencia de los procesadores a la demandada por las tareas. Al mismo tiempo, reduce al máximo la frecuencia cuando existen procesadores que no están ejecutando ninguna tarea. Finalmente, obtiene resultados estadísticos del ahorro generado al aplicar esta política.

3.1.1. Herramientas

Para realizar tanto la implementación del algoritmo de ajuste de tareas y del planificador se han empleado las siguientes herramientas software:

- **Python**: Se ha empleado como lenguaje de programación [34] para llevar a cabo la implementación de las aplicaciones y módulos estudiados en este trabajo. Se trata de un lenguaje dinámico y orientado a objetos, que puede ser utilizado para diferentes tipos de desarrollo de software. Una de sus principales ventajas es que, al tratarse de un lenguaje interpretado, ahorra un tiempo considerable en el desarrollo de los programas, pues no es necesario compilar ni enlazar los códigos.

- **DOT**: Este formato, proporcionado por la librería Graphviz [35], se emplea como lenguaje para representar los grafos de dependencias de tareas de los algoritmos. DOT permite representar grafos, guardarlos en ficheros y tratarlos mediante una interfaz de lectura de la librería Networkx [36] de Python.

3.2. Algoritmos de álgebra lineal como generadores de grafos

En la álgebra lineal densa existen numerosos algoritmos que pueden descomponerse en operaciones simples sobre bloques de tareas. En el caso de descomposiciones de matrices, tales como Cholesky, LU o QR las tareas deben ejecutarse en un determinado orden, es decir, existen dependencias entre ellas.El objetivo es diseñar un método sencillo para generar automáticamente grafos que puedan ser almacenados de forma legible y puedan ser visualizados cuando sea requerido.

Para implementar los programas que realizan la generación, se sustituyen las llamadas a las rutinas que generan las tareas y reemplazarlas por sentencias de escritura (función `printf` en el caso de emplear C) que imprimen en formato DOT las dependencias entre tareas. Un ejemplo para el caso de la descomposición de Cholesky puede ser el mostrado en el Código 3.1.

```
for( k = 1; k<=n; k++ ) {
  A (k,k) = sqrt (A (k,k)) ; % POTRF (Cholesky)
  for( i = k+1; i<=n; i++ ) {
    A (i,k) = A (i,k) / A (k,k); % TRSM
  }
  for( i = k+1; i<=n; i++ ) {
    for( j = k+1; j<=i-1; j++ ) {
      A (i,j) = A (i,j) - A (i,k) * A (j,k)'; % GEMM
    }
    A(i,i) = A(i,i) - A(i,k) * A(i,k)'; % SYRK
  }
}
```

Código 3.1: Factorización de Cholesky por bloques

```
   // A(i,i) = A(i,i) - A(i,k) * A(i,k)' ; % syrk
   printf("␣␣\"T_%d%d(%d)\"␣->␣\"S_%d%d%d(%d)\";\n",i,k,TRSM_WEIGHT,i,i,k,SYRK_WEIGHT);
   if( k>1 ) printf("␣␣\"S_%d%d%d(%d)\"␣->␣\"S_%d%d%d(%d)\";\n",i,i,(k-1),SYRK_WEIGHT,i,i,k,
       SYRK_WEIGHT);
```

Código 3.2: Ejemplo de sustitución de la tarea SYRK por `printf` correspondiente

En el Código 3.2, las variables que finalizan con `_WEIGHT` contienen el valor del coste asociado a dicha tarea. Este coste se representa como un número entero proporcional al orden de coste asociado a dicha tarea y se mide, a lo largo del trabajo, en unidades de tiempo o u.t. El algoritmo generador puede emplearse del siguiente modo:

```
$ ./generador-cholesky 3 > cholesky3x3.dot
```

Una vez se ejecuta este programa en C, se genera el siguiente código DOT de ejemplo para representar el grafo de dependencias entre tareas de la descomposición de Cholesky de 3×3 bloques.

```
digraph cholesky3x3 {
 // Grafo Cholesky para 3 bloques

 Graph [rankdir=LR];
 Node [shape=circle];

 // Nodos
 "C_11(4)"  [weight=4,style=filled,fillcolor="forestgreen"];
 "T_21(24)"  [weight=24,style=filled,fillcolor="dodgerblue"];
 ...
 "S_332(12)" [weight=12,style=filled,fillcolor="red"];
 "C_33(4)"  [weight=4,style=filled,fillcolor="forestgreen"];

 // Arcos
 "C_11(4)" -> "T_21(24)";
 "C_11(4)" -> "T_31(24)";
 ...
 "S_331(12)" -> "S_332(12)";
 "S_332(12)" -> "C_33(4)"
}
```

Este código se almacenará en un fichero `.dot` que leerá la aplicación diseñada en este trabajo. Al mismo tiempo se puede visualizar el contenido ejecutando el comando

```
$ dot -Tpdf cholesky3x3.dot > cholesky3x3.pdf
$ xpdf cholesky3x3.pdf
```

El resultado del grafo generado es el siguiente:

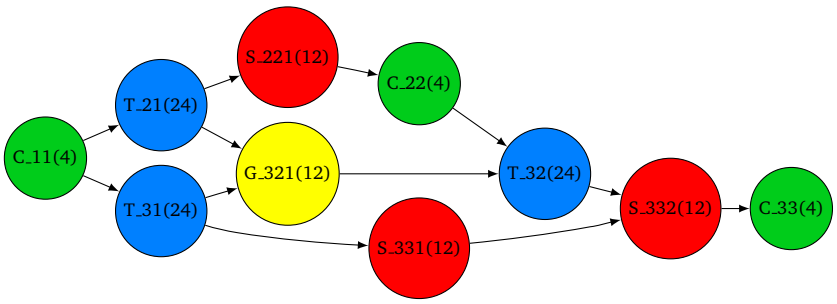


Figura 3.2: Grafo de dependencias entre tareas de la factorización de Cholesky de 3×3 bloques.

Una vez implementado el generador, es sencillo obtener grafos de la factorización para un tamaño de bloque determinado. Por otra parte, hay que ser conscientes de que este proceso resulta a primera vista costoso, por ese motivo, se tiene la intención de automatizar en un futuro, dando lugar a un procesador automático, que sea capaz de introducir automáticamente las sentencias de escritura o, como alternativa, que sea capaz de interpretar el algoritmo para detectar dependencias y cargue una estructura de grafo en memoria.

3.3. Conversión del grafo

Como se explica en el Capítulo 2, para aplicar la técnica CPM, y por lo tanto, para analizar holguras es necesario transformar el grafo obtenido donde los nodos representan tareas a otro donde sean las aristas las que representen las tareas. Por esa razón se ha implementado un conversor que realiza esta tarea automáticamente.

Algoritmo 3.1 Conversor de grafo: Nodos-Tarea a Arcos-Tarea

```
 1: para nodo en grafo hacer
 2:         si arcos_incidentes(nodo) != 1 entonces
 3:                 Crear un nuevo nodo
 4:                 Reasignar los arcos incidentes del antiguo nodo al nuevo
 5:                 Crear un nuevo arco desde el nuevo nodo al nodo antiguo
 6:                 Asignar coste 0 al nuevo nodo
 7:         fin si
 8: fin para
 9: para arco en grafo hacer
10:         Asignar el nombre de la tarea y coste del nodo destino al arco
11: fin para
12: Renombrar los nodos
```

Para realizar esta conversión se aplica el Algoritmo 3.1. De este modo, el grafo obtenido en la fase anterior se convierte en un grafo equivalente con las tareas situadas en los arcos. Si se observa el algoritmo, para cumplir las precedencias, es imprescindible insertar una serie de arcos nulos o actividades ficticias. En el caso de ejemplo, el grafo de la Figura 3.2 pasa a ser el de la Figura 3.3.

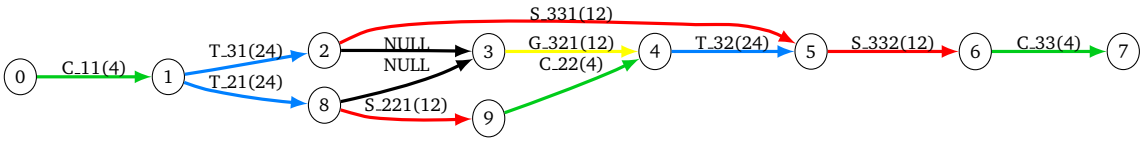


Figura 3.3: Grafo de dependencias con arcos-tarea de la factorización de Cholesky de 3×3 bloques.

A partir de este grafo se aplica el algoritmo de reducción de holguras, que. Este algoritmo se explica en detalle en la siguiente sección.

3.4. Reducción de holguras

En esta fase se explica en detalle el algoritmo de reducción de holguras. El objetivo del mismo es sencillo: alargar la duración de las tareas empleando el rango de frecuencias disponibles en el procesador para realizar, finalmente, la operación DVFS y ahorrar energía.

3.4.1. Aproximación teórica al algoritmo

Definiciones

Para comprender con mayor exactitud el algoritmo, es necesario introducir algunos aspectos teóricos previos. Si se parte de la definición matemática, un grafo de dependencias es un *digrafo ponderado* definido como $G = (V, E)$, donde:

- V es un conjunto de vértices o nodos y
- E es un conjunto de arcos o aristas, que relacionan estos nodos.

A partir de esta suposición, **definimos camino como una tupla de nodos conectados entre sí mediante aristas dirigidas**. En general, un camino tiene la siguiente forma:

$$C = \{V_1, V_2, \ldots, V_n\}, \tag{3.1}$$

donde entre cada par de nodos existe una arista dirigida que los une. Es decir, existe una arista que va del nodo V_i al V_{i+1}. Por ejemplo, en el camino C, formado por la tupla $\{8, 3, 4\}$, parte una arista del nodo 8 hacia el nodo 3, y de éste último existe otra arista hacia el nodo 4. Diremos también que el número de nodos de los que se compone un camino se representa mediante $|C|$ y supondremos que sólo existe un nodo inicial y otro final en el grafo.

A continuación se definen dos conjuntos de caminos en los que **primer nodo de los caminos es el inicial y el último es el final del grafo**:

- **Caminos críticos** (CP): conjunto formado por los caminos críticos (sucesiones de las tareas críticas):

$$CP = \bigcup_{i=1}^{n} CP_i = \{CP_1, \ldots, CP_n\}. \tag{3.2}$$

  Este conjunto estará compuesto por n caminos críticos. Si $i \in \mathbb{N}$ y $1 \leq i \leq n$ entonces el camino CP_i es el camino crítico i-ésimo del conjunto:

$$\begin{gathered} CP_i = \{V_1, \ldots, V_j\}, \text{ donde } H_T(V_k, V_{k+1}) = 0, \ \forall k : 1 \leq k \leq |CP_i| - 1 \wedge \\ V_1 = nodo\_inicial \ \wedge \ V_j = nodo\_final. \end{gathered} \tag{3.3}$$

- **Caminos alternativos** (CA): conjunto formado por caminos los alternativos (sucesiones de criticas o no críticas):

$$CA = \bigcup_{i=1}^{n} CA_i = \{CA_1, \ldots, CA_n\}. \tag{3.4}$$

  Este conjunto estará compuesto por n caminos alternativos. Si $i \in \mathbb{N}$ y $1 \leq i \leq n$ entonces el camino CA_i es el camino alternativo i-ésimo del conjunto:

$$CA_i = \{V_1, \ldots, V_j\}, \text{ donde } V_1 = nodo\_inicial \ \wedge V_j = nodo\_final. \tag{3.5}$$

Entonces, es fácil suponer que el conjunto de caminos críticos está incluido en el conjunto de caminos alternativos ($CP \subseteq CA$). Para la explicación posterior también es necesario definir los siguientes conceptos:

- **Subcaminos crítico** (SCP): conjunto formado por los caminos críticos obtenidos en el método explicado en la Sección 3.4.2 del algoritmo de reducción de holguras.
- **Subcamino alternativo** (SCA): conjunto formado por subcaminos de caminos alternativos pertenecientes a un camino alternativos del conjunto CA.

Aproximación teórica

Si se computa el conjunto de caminos alternativos CA del grafo,el coste de cada uno de ellos y, finalmente, éstos se ordenan de forma decreciente por el coste el camino o caminos con máximo coste, serán los críticos. Para el ejemplo de Cholesky de 3×3 bloques del grafo ilustrado en la Figura 3.3 tendremos caminos alternativos ordenados por coste descencedemente como se recoge en la Tabla 3.1.

CA_i	Camino	Coste
CA_0	$\{0, 1, 8, 9, 4, 5, 6, 7\}$	84
CA_1	$\{0, 1, 8, 3, 4, 5, 6, 7\}$	80
CA_2	$\{0, 1, 2, 3, 4, 5, 6, 7\}$	80
CA_3	$\{0, 1, 2, 5, 6, 7\}$	56

Tabla 3.1: Caminos alternativos de la factorización de Cholesky de 3×3 bloques.

Las fila resaltada en la Tabla 3.1 corresponde al camino crítico. Por lo tanto, el objetivo es **extender la longitud del resto de caminos sin que ningún otro camino alternativo supere la duración del camino o duración máxima del algoritmo**.

Por ejemplo, la tarea `G_321(12)` del grafo de la Figura 3.3, incluida en el subcamino alternativo $SCA_i = \{V_1, V_2, V_3\} = \{8, 3, 4\}$) del camino alternativo CA_1 puede ser alargada para reducir su holgura total. El coste original del camino formado por este subcamino es de:

$$coste\_original = \sum_{k=1}^{|SCA_i|-1} T_{V_k,V_{k+1}} = T_{V_1,V_2} + T_{V_2,V_3} = T_{8,3} + T_{3,4} = 0 + 12 = 12\, u.t. \quad (3.6)$$

Donde $T_{i,j}$ representa el coste de la arista que parte del nodo i y se dirige al nodo j. Si se centra la atención en el camino crítico CA_0, se puede observar que éste pasa por los nodos inicial y final del subcamino estudiado $\{8, 3, 4\}$, es decir, por los nodos 8 y 4. Por lo tanto, se puede concluir que, como el coste del subcamino alternativo $SCA_j = \{W_1, W_2, W_3\} = \{8, 9, 4\}$ del camino crítico es mayor que la del subcamino alternativo SCA_i $\{8, 3, 4\}$, sus tareas pueden ser ralentizadas. El coste máximo que podrá tener el subcamino alternativo SCA_i $\{8, 3, 4\}$, podrá ser de:

$$coste\_maximo = \sum_{k=1}^{|SCA_j|-1} T_{W_k,W_{k+1}} = T_{V_1,V_2} + T_{V_2,V_3} = T_{8,9} + T_{9,4} = 12 + 14 = 16\, u.t. \quad (3.7)$$

Como la diferencia entre el $coste\_maximo$ y el $coste\_original$ es de 4 u.t el coste del subcamino estudiado $\{8, 3, 4\}$ puede aumentarse en 4 u.t. (siempre que no afecte al coste de otro camino alternativo, que haga aumentar el coste total del algortimo). A partir de aquí, se puede definir el ratio de ralentización por tarea con la fórmula:

$$ratio\_de\_ralent = \frac{coste\_maximo}{coste\_original} = \frac{16}{12} = 1,33 \quad (3.8)$$

Si se aplica este ratio de ralentización a todas las tareas del subcamino $\{8, 3, 4\}$, la holgura total de las tareas que lo componen llegará a será nula:

$$\begin{aligned} coste &= \sum_{k=1}^{|SCA_i|-1} T_{V_k,V_{k+1}} \cdot ratio\_de\_ralent = \\ &= T_{V_1,V_2} \cdot ratio\_de\_ralent + T_{V_2,V_3} \cdot ratio\_de\_ralent = \\ &= T_{8,3} \cdot ratio\_de\_ralent + T_{3,4} \cdot ratio\_de\_ralent = 0 \cdot 1,33 + 12 \cdot 1,33 = 16\ u.t. \end{aligned}$$

Por lo tanto, el nuevo coste de las tareas del subcamino SCA_i $\{8, 3, 4\}$ será 0 u.t. para la tarea nula (`NULL`) y 16. u.t. la tarea `G_321`, que se escribirá como `G_321(16)`. Si se calcula el coste del subcamino alternativo SCA_i $\{8, 3, 4\}$, se verá que es igual al coste del subcamino alternativo SCA_j $\{8, 9, 4\}$:

$$\sum_{k=1}^{|SCA_i|-1} T_{V_k,V_{k+1}} = \sum_{k=1}^{|SCA_j|-1} T_{W_k,W_{k+1}} = 16\ u.t. \tag{3.9}$$

Este cambio en las holguras produce que el coste del camino alternativo CA_1, del que pertenece el subcamino alternativo SCA_i, sea el mismo que la del camino alternativo CA_0, del que pertenece el camino SCA_j. A modo general, ésta es la estrategia empleada por el algoritmo y la que se emplea para cada uno de los caminos alternativos. No obstante, la fórmula empleada por el algoritmo real para obtener el coste máximo de un subcamino es la siguiente:

$$coste\_maximo = coste\_maximo(nodo\_inicial, SCA_i(|SCA_i|)) - coste\_maximo(nodo\_inicial, SCA_i(1)) \tag{3.10}$$

En la ecuación (3.10) la función $coste\_maximo(V_i, W_j)$ devuelve el coste máximo para llegar del nodo V_i al nodo W_j, la variable $nodo\_raiz$ representa el nodo inicial del grafo y las expresiones $SCA_i(1)$ y $SCA_i(|SCA_i|)$ devuelven, respectivamente, el primer y último nodo del subcamino alternativo SCA_i. En en el siguiente apartado se explica en detalle el funcionamiento del algoritmo brevemente esbozado en esta sección.

3.4.2. Descripción del algoritmo

Para explicar con detalle su funcionamiento, a continuación se describe paso a paso el algoritmo junto con un ejemplo de su ejecución; concretamente, la descomposición Cholesky de 3×3 bloques. Para estos ejemplos se utilizan los rangos de frecuencia junto con los tiempos de ejecución para cada una de las frecuencias de las tareas del algoritmo mostrados en la Tabla 3.2. Los tiempos de ejecución a la máxima frecuencia se han obtenido a partir del orden de coste las tareas mostrados en la Tabla 2.2 escalados a un factor de 12. Por ejemplo, si el orden de coste de la tarea que computa factorización de Cholesky sobre un bloque es $O(\frac{n^3}{3})$, siendo n el tamaño del bloque, entonces se ha escogido como coste de ejecución a la máxima frecuencia $\frac{1^3}{3}12 = 4$ u.t. Esta suposición se repite para todas las tareas, aplicando en cada caso, la función de orden de coste correspondiente (véase la Tabla 2.2).

Frecuencia (GHz)	CHOL (u.t.)	TRSM (u.t.)	GEMM (u.t.)	SYRK (u.t.)
3,33	4,00	24,00	12,00	12,00
3,00	4,44	26.64	13,32	13,32
2,67	4,98	29,93	14,96	14,96
2,33	5,71	34,30	17,15	17,15
2.00	6,66	39,96	19,98	19,98

Tabla 3.2: Tabla de frecuencias de procesador con tiempos de ejecución estimados para cada tarea y frecuencia.

Por otra parte, se considera que el cambio de frecuencia tiene un coste significativo. Un cambio de frecuencia consumirá 1 u.t. Además se considera la frecuencia nula o 0,00 GHz cuando el procesador no está ejecutando ninguna tarea. Para el caso del algoritmo se considera la frecuencia 0,00 GHz como una alternativa, no obstante, no será necesaria su aplicación hasta que no se describa el simulador de planificación implementado en esta tesis de máster.

En los siguientes apartados se describen en detalle los pasos que sigue el algoritmo de reducción de holguras para su funcionamiento.

Asignación de frecuencias inicial

En el primer paso del algoritmo el objetivo es ordenar ascendentemente las frecuencias posibles del procesador y asignar a las tareas del algoritmo su ejecución a la máxima frecuencia. Este proceso se muestra en el Algoritmo 3.2.

Algoritmo 3.2 Asignar frecuencias a las tareas del grafo

```
1:  ordenar(rango_frecuencias)
2:  frecuencia_maxima= maximo(rango_frecuencias)
3:  para arco en grafo hacer
4:          peso = devuelve_peso_por_tarea_freq(arco.tipotarea, maxfreq)
5:          arco.peso_original = arco.peso = peso
6:          si arco.tarea = NULL entonces
7:                  arco.freq = maxfreq
8:          sino
9:                  arco.freq = 0
10:         fin si
11: fin para
```

Por defecto, el atributo de los arcos asociados a la frecuencia no viene establecido por los algoritmos de las fases anteriores; por este motivo, en este paso se asigna a todas las tareas, excepto a las nulas, la máxima frecuencia de procesador permitida. Dejando de este modo el grafo de dependencias no se aplicaría la técnica DVFS en ningún caso y no se producirían ahorros energéticos.

Obtención de subcaminos críticos

El objetivo en este paso del algoritmo es extraer o descomponer el grafo en subcaminos críticos. En primer lugar, se extrae el camino crítico principal. A continuación, este camino crítico es eliminado del grafo, y se vuelve a buscar el camino crítico en el grafo resultante. Si se repiten estos pasos hasta que el grafo quede vacío, se encuentra la secuencia de subcaminos críticos fundamentales. Este método es el que se encuentra implementando en en Algoritmo 3.3.

Cabe destacar que la rutina ***caminosAlternativos*** del Algoritmo 3.4 devuelve todos los caminos posibles del grafo, ordenados descendentemente, para llegar desde un nodo inicial a un nodo final pasados como argumentos. Esta rutina emplea otras dos subrutinas elementales que se muestran en los Algoritmos 3.5 y 3.6.

La rutina ***buscaCamAltRec*** del Algoritmo 3.5 recupera todos los caminos posibles entre los nodos inicio y final sobre el grafo pasado como argumento. Por otra parte la rutina ***ordenarPorCoste*** del Algoritmo 3.6 ordena la lista de caminos críticos por peso. Por este motivo, el primer camino de la lista será el crítico para todos los casos. En la Tabla 3.3 se muestra cómo el Algoritmo 3.3 obtiene los caminos críticos según el modo explicado en el Apartado 3.4.2, a partir del ejemplo del algoritmo de Cholesky de 3×3 bloques.

En la Tabla 3.4 se muestra cómo el Algoritmo 3.3 obtiene los caminos alternativos a partir del ejemplo del algoritmo de Cholesky de 3×3 bloques.

Algoritmo 3.3 Búsqueda de caminos críticos y alternativos

```
grafo_aux= copiar(grafo)
caminos_criticos = []
mientras existan nodos en grafo_aux hacer
    caminos = []
    raices = buscaRaices(grafo_aux)
    hojas = buscaHojas(grafo_aux)
    para raiz en raices hacer
        para hoja en hojas hacer
            caminos.añadir(caminosAlternativos(grafo_aux, raiz, hoja, costeDVFS))
        fin para
    fin para
    si contar(caminos) > 0 entonces
        ordenarPorCoste(grafo_aux, caminos, costeDVFS)
        camino = caminos.extraerprimero()
        caminos_criticos.añadir(camino)
        quitarCaminoDelGrafo(grafo_aux, camino)
    fin si
fin mientras
coste_total = caculcularCoste(grafo, caminos_criticos, costeDVFS)
c_alternativos = caminosAlternativos(grafo, costeDVFS)
```

Algoritmo 3.4 ***caminosAlternativos***(grafo, nodo_inicial, nodo_final, costeDVFS)

```
caminos= buscaCamAltRec(grafo, nodo_inicial, nodo_final)
caminos= ordenarPorCoste(grafo, caminos, costeDVFS)
```

Algoritmo 3.5 buscaCamAltRec(grafo, nodo_inicial, nodo_final, camino=[])

```
caminos = []
si nodo_inicial existe en grafo entonces
    camino.añadir(nodo_inicial)
    si nodo_inicial = nodo_final entonces
        devolver [camino]
    fin si
    si nodo_inicial no existe en el grafo entonces
        devolver []
    fin si
    para nodo alcanzables desde nodo_incial hacer
        si nodo no existe en grafo entonces
            nuevos_caminos = BuscaCamAltRec(grafo, nodo_inicial, nodo_final, camino)
            para nuevo_camino en nuevos_caminos hacer
                camino.añadir(nuevo_camino)
            fin para
        fin si
    fin para
    devolver caminos
fin si
```

Algoritmo 3.6 ordenarPorCoste(grafo, caminos, costeDVFS)

```
caminos_peso= []
para c en caminos hacer
    coste = costeCamino(grafo, c, c[0], c[longitud(c)-1], costeDVFS)
    caminos_peso.añadir([c, coste])
fin para
ordenar_desc(caminos_peso)
devolver caminos_peso
```

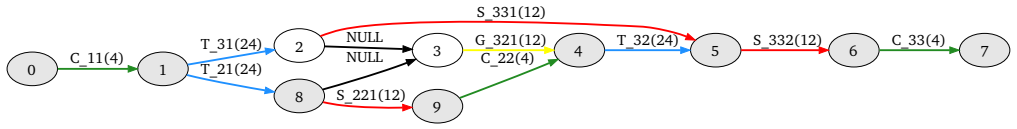
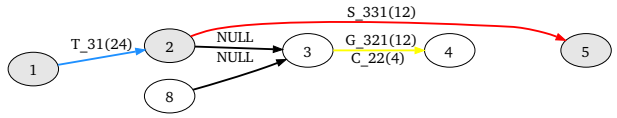
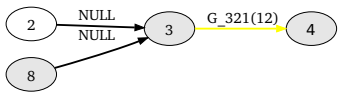
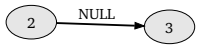
SCP	C	Grafo
{0, 1, 8, 9, 4, 5, 6, 7}	84	
{1, 2, 5}	36	
{8, 3, 4}	13	
{2, 3}	0	

Tabla 3.3: Secuencia de obtención de subcaminos críticos: secuencia, coste y grafo asociado.

Reducción de holguras

El siguiente paso del algoritmo es uno de los más importantes del algoritmo total, ya que se realizan las operaciones de ralentización de tareas en función del rango de frecuencias y tiempos posibles de ejecución de las tareas mostradas en la Tabla 3.2.

En primer lugar, se itera sobre los caminos alternativos, y se calcula cuál es la duración máxima que pueden llegar a tener. Esta duración se obtiene calculando la distancia desde el nodo raíz original al nodo final del camino crítico sobre el que se itera y sobre el grafo original (al que van actualizándose los pesos y las frecuencias de las tareas). Por otra parte, se calcula la distancia del nodo raíz original al nodo inicial del camino crítico. La diferencia de estas dos cantidades es la duración máxima que puede llegar a tener el camino crítico sobre el que itera el bucle. En otras palabras, se aplica la ecuación (3.10). Estas acciones son las que ralentizan las tareas en el Algoritmo 3.7. Tal y como puede comprobarse, este algoritmo llama a la rutina ***ajustaTiempos*** que, a partir del grafo, el camino crítico, el coste actual y el coste máximo que éste puede llegar a tener, ajusta la duración de las tareas (y la frecuencia), de modo que la duración final se ajuste lo máximo posible a la duración máxima permitida, y no sea superada por un umbral establecido. Definiremos a partir de ahora como *ratio de exceso* a este ajuste sobre la duración de las tareas.

Este *ratio de exceso* se establece por el usuario, que indicará el factor en el cual el coste máximo del algoritmo puede superarse. Este factor debe definirse como un valor flotante; por ejemplo, el valor 1 indica que la duración no puede excederse, mientras que el valor $1,5$ indica que la duración total puede excederse en la mitad de la duración total. En otras palabras, este ratio modifica el coste máximo (coste del camino crítico) del algoritmo del siguiente modo:

$$coste\_maximo\_permitido = coste\_camino\_critico \cdot ratio\_de\_exceso \tag{3.11}$$

Este parámetro es opcional y puede ayudar a obtener mejores ajustes de tiempos y a reducir el número de cambios de frecuencia. Por defecto, el algoritmo reduce la frecuencia en el mismo factor en las tareas de aquellos caminos donde se puede extender su duración; es decir, produce secuencias de tareas donde la frecuencia es constante. No obstante, si encuentra que la reducción de la frecuencia en una determinada

CA	C	Grafo
{0, 1, 8, 9, 4, 5, 6, 7}	84	0 —C_11(4)→ 1 —T_21(24)→ 8 —S_221(12)→ 9 —C_22(4)→ 4 —T_32(24)→ 5 —S_332(12)→ 6 —C_33(4)→ 7
{0, 1, 8, 3, 4, 5, 6, 7}	80	0 —C_11(4)→ 1 —T_21(24)→ 8 —NULL→ 3 —G_321(12)→ 4 —T_32(24)→ 5 —S_332(12)→ 6 —C_33(4)→ 7
{0, 1, 2, 3, 4, 5, 6, 7}	80	0 —C_11(4)→ 1 —T_31(24)→ 2 —NULL→ 3 —G_321(12)→ 4 —T_32(24)→ 5 —S_332(12)→ 6 —C_33(4)→ 7
{0, 1, 2, 5, 6, 7}	56	0 —C_11(4)→ 1 —T_31(24)→ 2 —S_331(12)→ 5 —S_332(12)→ 6 —C_33(4)→ 7

Tabla 3.4: Caminos alternativos: secuencia, coste y grafo asociado.

tarea provoca que un camino alternativo "ajeno" aumente el coste máximo permitido y, por tanto, el coste total del algoritmo, deberá aumentar dicha frecuencia hasta que el coste del camino que produjo el exceso esté por debajo de la coste máximo permitido.

Algoritmo 3.7 Ajuste de tiempos de tareas

```
1:  para cp en caminos_criticos hacer
2:          c_inicio = caminosAlternativos(grafo, nodo_inicial, cp[0], costeDVFS)
3:          c_final = caminosAlternativos(grafo, nodo_inicial, cp[longitud(cp)-1], costeDVFS)
4:          coste= costeCamino(grafo, cp, caminos[c][longitud(cp)-1], caminos[c+1][longitud(cp)-1], costeDVFS)
5:          si coste ¿ 0 entonces
6:                  s = costeCamino(grafo, cp, c_inicio[0], c_inicio[longitud(c_inicio)-1], costDVFS)
7:                  e = costeCamino(grafo, cp, c_final[0], c_final[longitud(c_final)-1], costDVFS)
8:                  si s > e entonces
9:                          max_coste= e - s
10:                 sino
11:                         max_coste= 0
12:                 fin si
13:                 ajustaTiempos(grafo, cp, max_coste, coste, coste_total, c_alternativos, costDVFS)
14:         fin si
15: fin para
```

A partir de un camino crítico de los calculados en pasos anteriores, la rutina presentada en el Algoritmo 3.8 ajusta la duración total de éste en función de un umbral máximo pasado como argumento. En este caso, el algoritmo calcula un ratio de ralentización que es aplicado, en un principio, a todas las tareas por igual. A medida que va actualizando la nueva frecuencia de las tareas, comprueba si la duración de alguno de los caminos alternativos se ve afectada; en tal caso, aumentará la frecuencia y, por tanto, reducirá el tiempo de la tarea hasta que ningún camino alternativo exceda la duración total del algoritmo.

Algoritmo 3.8 ***ajustaTiempos***(grafo, cp, max_coste, coste, coste_total, c_alternativos, costDVFS)

```
1:  si coste > 0 entonces
2:          ratio=max_coste/coste
3:  sino si max_coste < cost entonces
4:          ratio= 1
5:  sino
6:          ratio= 0
7:  fin si
8:  pesos= []
9:  para p en {0,...,longitud(cp)-1} hacer
10:         id_tarea= grafo.arco(cp[p],cp[p+1]).id_tarea
11:         peso_orig= grafo.arco(cp[p],cp[p+1]).peso_orig
12:         fr, tm= seleccFreq(id_tarea, peso_orig * ratio, peso_orig)
13:         si peso_orig = 0 entonces
14:                 fijo= Verdadero
15:         sino
16:                 fijo= Falso
17:         fin si
18:         pesos.añadir([tm, fr, fijo])
19: fin para
20: p= 0
21: mientras p < longitud(cp)-1 hacer
22:         grafo.arco(cp[p],cp[p+1]).tmp = pesos[p].tm
23:         grafo.arco(cp[p],cp[p+1]).frec = pesos[p].fr
24:         si pesos[p].fijo = Falso entonces
25:                 maxdiff = 0
26:                 diff = 0
27:                 error = Falso
28:                 para a en alt hacer
29:                         cost= costeCamino(grafo, a, a[0], a[longitud(a)-1], costeDVFS)
30:                         maxdiff= máx(maxdiff, cost - coste_total)
31:                         diff= máx(diff, coste - coste_total * ratio_de_exceso)
32:                 fin para
33:                 si diff > 0 entonces
34:                         nuevo_peso = pesos[p].peso - diff
35:                         id_tarea= grafo.arco(cp[p],cp[p+1]).id_tarea
36:                         peso_orig = grafo.arco(cp[p],cp[p+1]).peso_orig
37:                         si nuevo_peso < peso_orig entonces
38:                                 nuevo_peso= peso_orig
39:                         fin si
40:                         fr, tmp= seleccFreq(id_tarea, nuevo_peso, peso_orig)
41:                         pesos[p] = [tmp, fr, Verdadero]
42:                         para i en {p+1,...,longitudpesos} hacer
43:                                 si pesos[i].tmp > 0 Y pesos[i].fijo = Verdadero entonces
44:                                         pesos[i].fijo= Falso
45:                                 fin si
46:                         fin para
47:                         arcos_no_fijos = 0
48:                         para i en {0,...,longitud(pesos)} hacer
49:                                 si pesos[i].fijo = Falso entonces
50:                                         arcos_no_fijos+= 1
51:                                 fin si
52:                         fin para
53:                         si arcos_no_fijos > 0 entonces
54:                                 reparto= diff / arcos_no_fijos
55:                         sino
56:                                 reparto = diff
57:                         fin si
58:                         para pr en {0,...,longitud(cp)-1} hacer
59:                                 si pesos[pr].fijo = Falso entonces
60:                                         nuevo_peso= pesos[pr][0] + reparto
61:                                         id_tarea= grafo.arco(cp[pr],cp[pr+1]).id_tarea
62:                                         peso_orig = grafo.arco(cp[pr],cp[pr+1]).peso_orig
63:                                         fr, tmp= seleccFreq(id_tarea, nuevo_peso, peso_orig)
64:                                         pesos[pr] = [tmp, fr, Falso]
65:                                         grafo.arco(cp[pr],cp[pr+1]).peso = pesos[pr].tm
66:                                         grafo.arco(cp[pr],cp[pr+1]).frec = pesos[pr].fr
67:                                 fin si
68:                         fin para
69:                         p = 0
70:                 sino
71:                         p += 1
72:                 fin si
73:         sino
74:                 p += 1
75:         fin si
76: fin mientras
```

Ejemplo de funcionamiento

Para ilustrar el comportamiento del algoritmo de extensión de holguras, en este apartado se realiza una traza mediante el ejemplo de la factorización de Cholesky de 3×3 bloques. La función o algoritmo principal itera sobre los caminos subcríticos de la factorización de Cholesky obtenidos mediante el Algoritmo 3.3 recogidos en la Tabla 3.5.

SCP_i	Camino	Coste
SCP_0	$\{0, 1, 8, 9, 4, 5, 6, 7\}$	84
SCP_1	$\{1, 2, 5\}$	36
SCP_2	$\{8, 3, 4\}$	12
SCP_3	$\{2, 5\}$	0

Tabla 3.5: Caminos subcríticos de la factorización de Cholesky de 3×3 bloques.

La primera iteración del algoritmo, ilustrada en la Figura 3.4, ajusta el coste del subcamino crítico SCP_0 (nodos sombreados en gris) de duración 84 u.t. En esta iteración, se aplica la ecuación (3.10) para obtener la duración máxima (84 u.t.). En este caso, el subcamino crítico SCP_0 coincide con el camino crítico del grafo, por lo que el ratio de ralentización devuelto por la ecuación (3.8) es 1. Por este motivo las tareas conservan su coste y frecuencia.

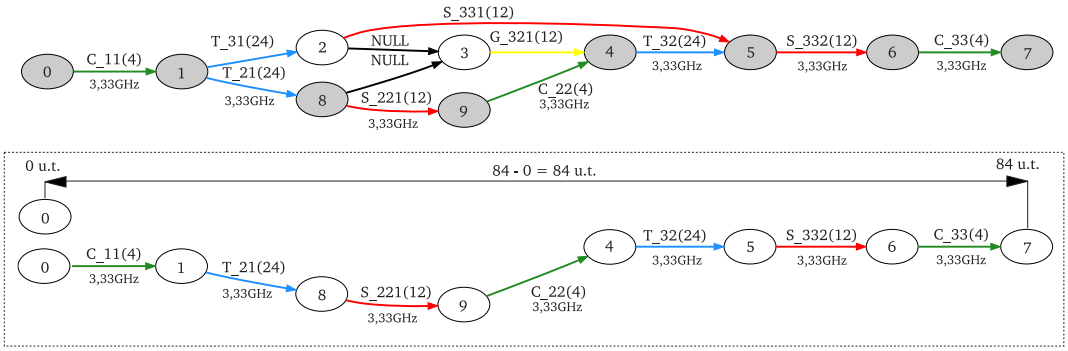


Figura 3.4: Iteración 1 del algoritmo de reducción de holguras.

La segunda iteración del algoritmo, ilustrada en la Figura 3.5, ajusta el coste del subcamino crítico SCP_1 (nodos sombreados en gris) de duración 36 u.t. En este caso la ecuación (3.10) devuelve una duración máxima para el subcamino de 64 u.t:

$$\begin{aligned} coste\_maximo &= coste\_maximo(0, SCA_i(3)) - coste\_maximo(0, SCA_i(1)) \\ &= coste\_maximo(0, 5) - coste\_maximo(0, 1) = 68 - 4 = 64\ u.t. \end{aligned}$$

Cuando se aplica la ecuación (3.8) y se ralentizan las tareas `T_31` y `S_331` del subcamino crítico SCP_1 se produce un efecto colateral en el grafo, haciendo que el camino alternativo sombreado aumente su duración en 97,69 u.t., superando el coste máximo permitido.

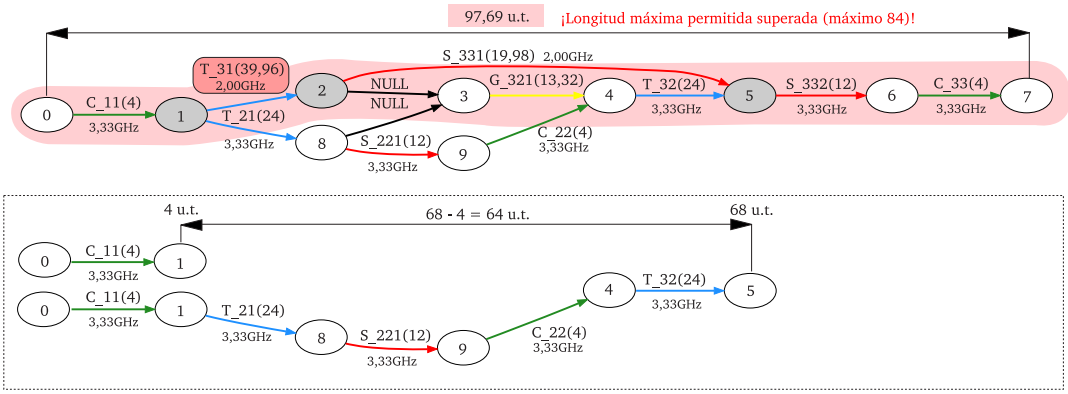


Figura 3.5: Iteración 2 del algoritmo de reducción de holguras y error detectado.

Cuando el algoritmo detecta el exceso, reduce la duración de la tarea `T_31` para que el camino alternativo sombreado, y cualquier otro camino, no exceda el coste máximo permitido de 84 u.t. El caso de corrección del error es el que se muestra en la Figura 3.6, al realizar esta corrección el camino alternativo sombreado pasa a tener una duración de 83,32 u.t., menor que 84.u.t.

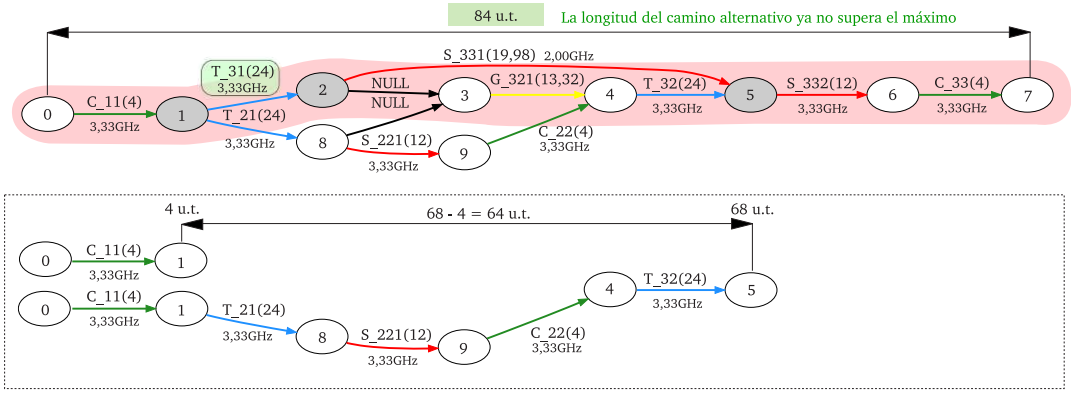


Figura 3.6: Iteración 2 del algoritmo de reducción de holguras y solución al error detectado.

La tercera iteración del algoritmo, ilustrada en la Figura 3.7, ajusta el coste del subcamino crítico SCP_2 (nodos sombreados en gris) de duración 12 u.t. En este caso la ecuación (3.10) devuelve una duración máxima para el subcamino de 16 u.t. Cuando se aplica la ecuación (3.8) y se ralentiza la tarea `G_321`, el camino alternativo sombreado aumenta su duración a 84,96 u.t. y supera el coste máximo permitido de 84 u.t.

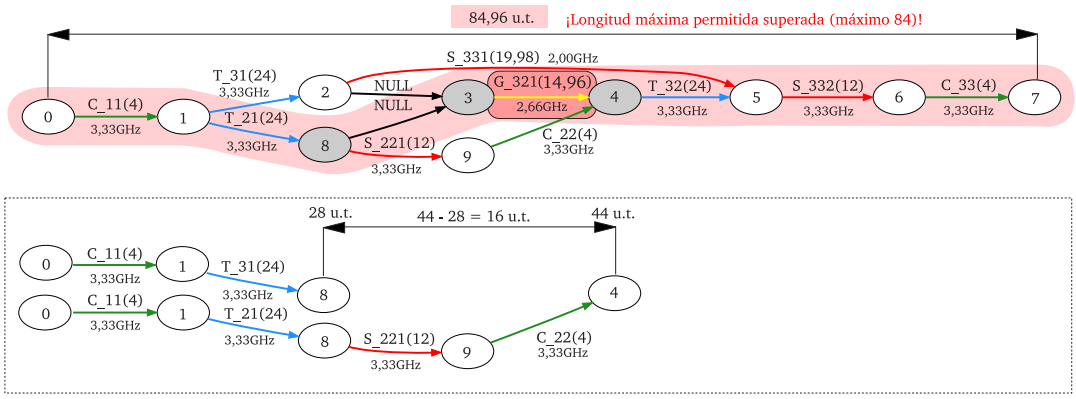


Figura 3.7: Iteración 3 del algoritmo de reducción de holguras y error detectado.

El algoritmo sigue iterando y sigue comprobado si existe algún camino alternativo más que exceda la duración máxima permitida. Como se ilustra en la Figura 3.8, el camino alternativo en sombreado también excede la duración máxima permitida: 84,96 u.t. Finalmente el algoritmo detecta cuál es la diferencia máxima entre la duración de los caminos alternativos que exceden la duración y la duración máxima permitida. Esta diferencia es, en este caso, de 0,96 u.t.

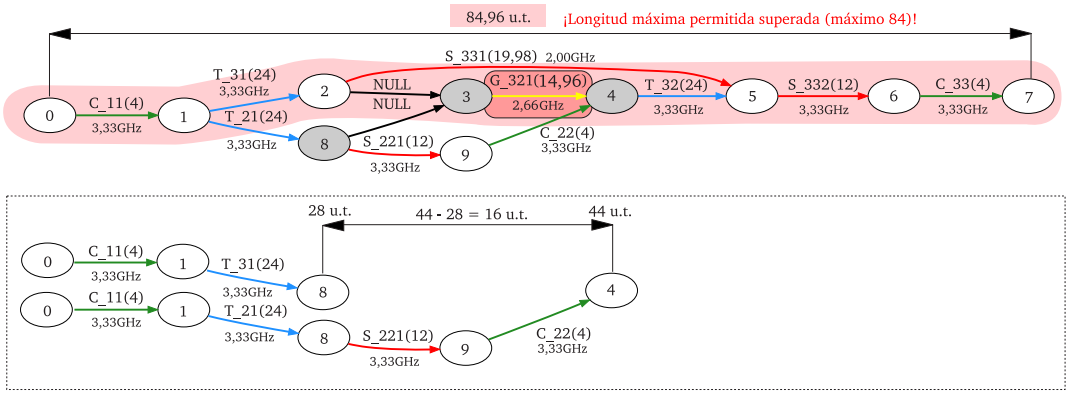


Figura 3.8: Iteración 3 del algoritmo de reducción de holguras y error detectado.

Finalmente ajusta y reduce el tiempo de la tarea `G_321` para que la duración de los caminos alternativos en sombreado estén por debajo del máximo permitido, en este caso 84 u.t.

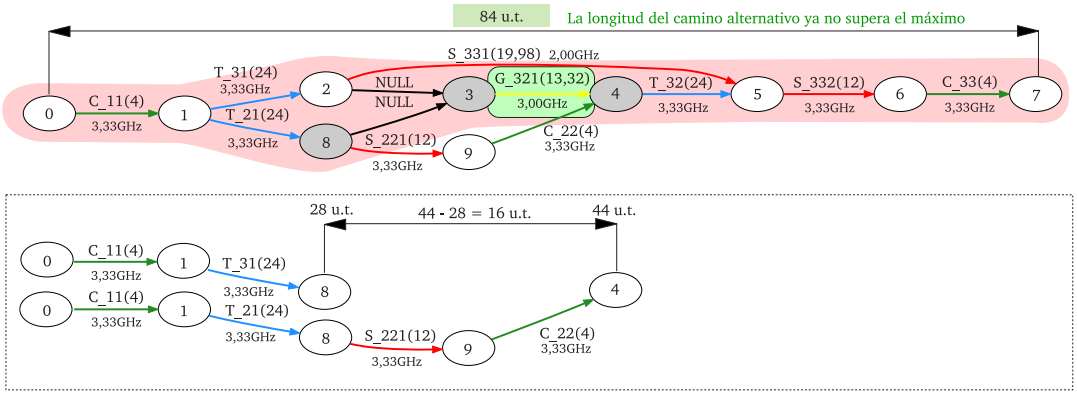


Figura 3.9: Iteración 3 del algoritmo de reducción de holguras y solución al error detectado

Finalmente, después de haber reducido la frecuencia de la tarea `G_321` y hacer que ningún camino alternativo provoque un exceso en el coste máximo permitido, se da por finalizada la iteración, tal y como muestra la Figura 3.9.

Una vez el algoritmo ha ralentizado el tiempo de las tareas pertenecientes a los **subcaminos críticos no nulos** de la Tabla 3.5, el algoritmo termina. El grafo resultante contiene, para cada una de las tareas, la duración y la frecuencia recomendadas para su ejecución.

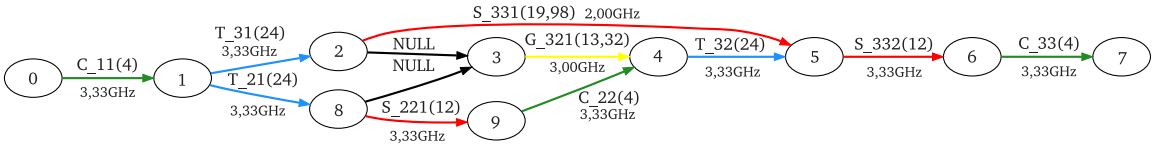


Figura 3.10: Grafo obtenido tras aplicar el algoritmo de reducción de holguras para la factorización de Cholesky de 3×3 bloques.

A modo general, en la Tabla 3.6 se ofrecen ciertos resultados comparativos sobre el ajuste realizado en el grafo. En una situación ideal, todos los caminos alternativos del grafo deberían tener la misma duración. Al tener un rango de frecuencias discreto, considerar costes por cambios de frecuencia y tener dependencias entre tareas que imposibilitan, en algunos casos, un ajuste óptimo, todos los caminos alternativos representados en la tabla no llegan a tener una duración de 84 u.t., aunque sí se aproximan.

CA_i	Camino	Coste original	Coste final
CA_0	$\{0, 1, 8, 9, 4, 5, 6, 7\}$	$84,00$	$84,00$
CA_1	$\{0, 1, 8, 3, 4, 5, 6, 7\}$	$80,00$	$83,32$
CA_2	$\{0, 1, 2, 3, 4, 5, 6, 7\}$	$80,00$	$83,32$
CA_3	$\{0, 1, 2, 5, 6, 7\}$	$56,00$	$65,98$
	Ratio de ajuste	**89 %**	**94 %**

Tabla 3.6: Comparativa de duración de caminos alternativos de la factorización de Cholesky de 3×3 bloques tras aplicar el algoritmo de reducción de holguras.

El parámetro ratio de ajuste mostrado en la tabla se calcula mediante la siguiente fórmula:

$$ratio\_ajuste = \frac{\sum_{i=1}^{n} coste(CA_i)}{n} \cdot 100 \tag{3.12}$$

Como se puede observar, para el grafo de dependencias entre tareas del algoritmo de Cholesky original es de un 89 %. Cuando se aplica el algoritmo de reducción de holguras, el ajuste es mayor, siendo de un 94 %. En caso de que se tuvieran condiciones ideales, es decir, que hubiera un número infinito de frecuencias posibles, no hubieran costes entre cambios de frecuencia y no existieran dependencias que limitaran los ajustes, el ratio de ajuste sería del 100 %.

3.5. Simulador para la planificación de tareas

El planificador consciente del consumo implementado en esta sección es la última fase del simulador implementado en este trabajo. Una vez obtenido el grafo de dependencias de un algoritmo de álgebra lineal densa donde las tareas contienen un atributo con la frecuencia recomendada para ser ejecutadas establecidas por el *algoritmo de reducción de holguras*, el grafo se pasa a al planificador consciente del consumo, donde se realiza una simulación de la ejecución de las tareas.

Concretamente, se desea planificar de forma simulada las tareas de un algoritmo de álgebra lineal, teniendo en cuenta que los procesadores pueden trabajar a una menor frecuencia, y por lo tanto pueden ahorrar energía. Este simulador obtiene una traza en forma de gráficas que permite ver en qué instante una tarea ha empezado o ha finalizado su ejecución, además de los cambios de frecuencia llevados a cabo. Finalmente, también es capaz de obtener estadísticas sobre el ahorro de energía producido; concretamente, una vez ha terminado la simulación, se obtienen los porcentajes de tiempo en los que cada socket ha permanecido en cada una de las frecuencias.

3.5.1. Parámetros de entrada

El planificador implementado recibe una serie de parámetros de entrada que cabe destacar:

- Grafo de dependencias entre tareas generado por el *algoritmo de reducción de holguras*.
- Especificación de la arquitectura donde se lleva a cabo la simulación:
  - *Número de sockets:* Representa el número de procesadores multinúcleo que puede tener una determinada plataforma.
  - *Número de núcleos por socket:* Representa el número de núcleos que tiene cada procesador multinúcleo o socket.
- Estructura de datos con la información de las tareas y tiempo de ejecución en cada una de las frecuencias de procesador aceptadas.
- Frecuencias de procesador aceptadas.
- Tiempo entre cambio de frecuencia o tiempo para realizar DVFS.

Todos estos parámetros servirán al simulador para realizar una planificación consciente de la energía, reduciendo, en el caso de que sea posible, la frecuencia del procesador.

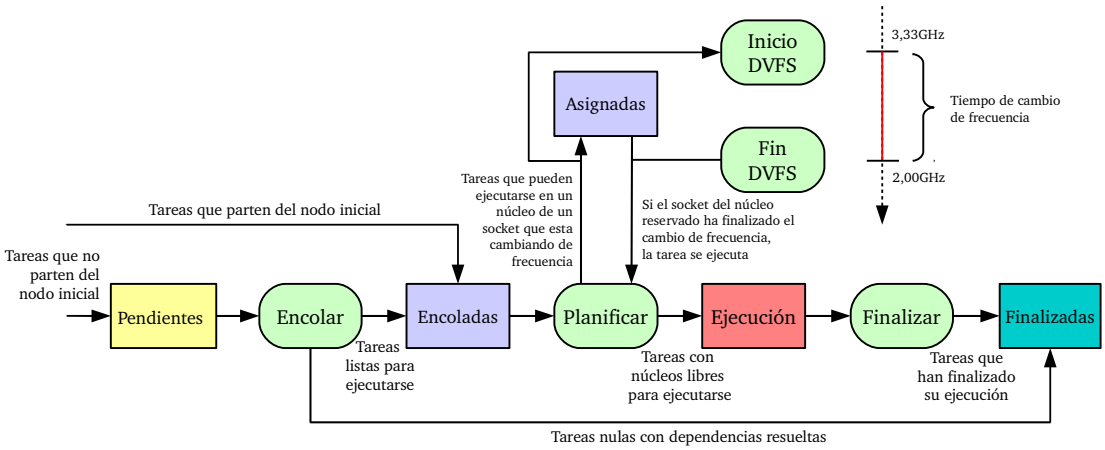


Figura 3.11: Estructura del planificador de tareas implementado.

3.5.2. Implementación

Algunas restricciones de este simulador, tal y como se explica en la Sección 1.3.1 del Capítulo 1, es que se ajusta la frecuencia de los procesadores a nivel de socket y no de núcleo, y además, no es capaz de realizar un cambio de frecuencia mientras existen tareas ejecutándose en los núcleos. Un esquema del planificador se muestra en la Figura 3.11.

Para realizar la implementación, se ha optado por utilizar el modelo de planificador *estático por listas de prioridad* por dos razones; en primer lugar, se conoce a priori la duración aproximada de las tareas (proporcionadas por el grafo) y, en segundo lugar, se necesita que ciertas tareas tengan prioridad en la cola, pues aquellas que requieran mayor frecuencia y pertenezcan a un subcamino crítico de mayor coste, deberán estar primero en la cola.

Estados de las tareas

Los planificadores por listas emplean una serie de estados por los que transcurren cada una de las tareas. Los estados considerados son los siguientes:

- **Pendiente:** Estado inicial de todas las tareas del grafo excepto las tareas que parten del suceso o nodo inicial del grafo, que estarán inicialmente encoladas. A medida que las tareas pasan a estar encoladas, se eliminan de esta lista.
- **Encolada:** Una tarea pasa a estar encolada cuando todas sus tareas de las que depende han finalizado su ejecución. Esta cola de tareas pendientes para ejecutarse ordena las tareas en función de dos parámetros. En primer lugar, por la frecuencia mínima recomendada para ser ejecutadas de forma descendente y, en segundo lugar, por el identificador de subcamino crítico al que pertenece (este identificador es el obtenido cuando se realiza la descomposición en subcaminos críticos explicada en la Sección 3.4.2). El proceso que se aplica en este caso es el que se muestra en en el Algoritmo 3.9.
- **Asignada:** Una tarea pasa a estar asignada cuando existe algún socket libre que no está funcionando a la frecuencia requerida por la tarea. Además, se reserva el núcleo del socket a dicha tarea, de modo que, cuando finalice el cambio de frecuencia, pueda ser asignada a este núcleo. Por otra parte una tarea también puede pasar a la lista de tareas asignadas cuando exista algún socket en proceso de cambio de frecuencia que tenga núcleos que no estén reservados. En este proceso uno de esos núcleos se reserva de modo que cuando finaliza el proceso de cambio de frecuencia la tarea se ejecuta en dicho núcleo. El proceso que se aplica en este caso es el que se muestra en el Algoritmo 3.9.

Algoritmo 3.9 Transición de tareas de pendientes a encoladas/finalizadas (**encolarTareas**)

```
nuevas_finalizadas = []
para tarea en pendientes hacer
    si tarea = NULL Y dependenciasResueltas(tarea) = Verdadero entonces
        nuevas_finalizadas.añadir(tarea)
    fin si
fin para
para tarea en nuevas_finalizadas hacer
    pendientes.eliminar(tarea)
    finalizadas.añadir(tarea)
fin para
nuevas_encoladas = []
para tarea en pendientes hacer
    si tarea != NULL Y dependenciasResueltas(tarea) = Verdadero entonces
        nuevas_encoladas.añadir(tarea)
    fin si
fin para
para tarea en nuevas_encoladas hacer
    pendientes.eliminar(tarea)
    encoladas.añadir(tarea)
fin para
encoladas.ordenar(frecuencia:descendente, subcaminocritico:ascendente)
```

- **Ejecución:** Una tarea pasa a ejecución cuando existe un núcleo libre en algún núcleo de un socket que funciona una frecuencia mayor o igual a la requerida por la tarea. Como ya se ha comentado anteriormente, la frecuencia requerida por la tarea es la asignada por el algoritmo de reducción de holguras. El proceso que se aplica en este caso es el que se muestra en el Algoritmo 3.10.

Algoritmo 3.10 Transición de tareas de encoladas/asignadas a ejecución (**ejecutarTareas**)

```
 1: nuevas_ejecutandose = []
 2: para tarea en asignadas hacer
 3:     si cambioFrecuenciaFinalizado(tarea.socket_reservado) = Verdadero entonces
 4:         ejecutar(tarea, tarea.socket_reservado, tarea.nucleo_reservado)
 5:         nuevas_ejecutandose.añadir(tarea)
 6:     fin si
 7: fin para
 8: para tarea en nuevas_ejecuntadose hacer
 9:     encoladas.eliminar(tarea)
10:     ejecutandose.añadir(tarea)
11: fin para
12: nuevas_ejecutandose = []
13: nuevas_asignadas = []
14: para tarea en encoladas hacer
15:     planificada, asignada, socket, nucleo = politicaAsignacion(tarea)
16:     si asignada = Verdadero entonces
17:         asignadas.añadir(tarea)
18:     sino si planificada = Verdadero entonces
19:         ejecutar(tarea, socket, nucleo)
20:         nuevas_ejecutandose.añadir(tarea)
21:     fin si
22: fin para
23: para tarea en nuevas_encoladas hacer
24:     encoladas.eliminar(tarea)
25:     ejecutandose.añadir(tarea)
26: fin para
```

- **Finalizada:** Una tarea pasa a estar finalizada cuando ha completado su ejecución. El proceso que se aplica en este caso es el que se muestra en el Algoritmo 3.11.

Algoritmo 3.11 Transición de tareas de ejecución a finalizadas (**finalizarTareas**)

```
 1: nuevas_finalizadas = []
 2: para socket en sockets hacer
 3:     para nucleo en socket.nucleos hacer
 4:         si nucleo.libre = Falso Y nucleo.tarea.tiempofin ≤ tiempo_actual entonces
 5:             nucleo.libre = Verdadero
 6:             nuevas_finalizadas.añadir(nucleo.tarea)
 7:             ejecutandose.eliminar(tarea)
 8:             finalizadas.añadir(tarea)
 9:         fin si
10:     fin para
11: fin para
```

El Algoritmo 3.12 muestra el planificador implementado, encargado de llamar a las funciones de cambio de transición anteriores y de iterar hasta el próximo evento.

Algoritmo 3.12 Algoritmo de transición de tareas ejecutándose a finalizadas

```
 1: tiempo= 0
 2: pendientes = encoladas = asignadas = ejecutandose = finalizadas = []
 3: para tarea en grafo hacer
 4:     si dependenciasResueltas(tarea) = Verdadero entonces
 5:         encoladas.añadir(tarea)
 6:     sino
 7:         pendientes.añadir(tarea)
 8:     fin si
 9: fin para
10: mientras existan tareas en pendientes hacer
11:     finalizarCambiosFrecuencia()
12:     finalizarTareas(tiempo)
13:     encolar_asignarTareas()
14:     ejecutarTareas()
15:     reducirFrecuenciaSocketsNoUsados()
16:     tiempo = obtenerTiempoProximoEvento()
17: fin mientras
```

Dicho planificador funciona por eventos, es decir, se incrementa la variable que representa el tiempo hasta el próximo evento (función **obtenerTiempoProximoEvento** en el Algoritmo 3.12). Estos eventos pueden ser el hecho de que una tarea ha finalizado su ejecución o bien el hecho de que un socket ha finalizado su cambio de frecuencia.

Política de asignación de tareas a núcleos

El proceso de asignación de tareas a ejecución se realiza en el siguiente orden:

1. Si existe alguna tarea en la cola de tareas *asignadas*, y además se da el caso de que el socket y el núcleo reservado haya finalizado el proceso de cambio de frecuencia y esté corriendo a la frecuencia establecida por la tarea, ésta pasará a ejecutarse directamente.

2. La cola de tareas *encoladas* irá recorriéndose y para buscar el primer núcleo que cumpla alguna de las siguientes condiciones:

   a) Si existe algún socket que no esté en proceso de cambio de frecuencia, esté funcionando a la misma frecuencia requerida por la tarea y además tenga algún núcleo libre, se asignará dicho núcleo a la tarea encolada. En caso contrario se comprobará el siguiente condición.

   b) Si existe algún socket que esté en proceso de cambio de frecuencia, la frecuencia a la que cambiará próximamente sea la requerida por la tarea y además tenga algún núcleo no reservado, se reservará dicho núcleo a la tarea encolada. En caso contrario se comprobará el siguiente condición.

   c) Si existe algún socket que no esté en proceso de cambio de frecuencia, esté funcionando a una frecuencia mayor que la frecuencia requerida por la tarea y además tenga algún núcleo libre, se asignará dicho núcleo a la tarea encolada (la tarea finalizará antes de lo previsto). En caso contrario se comprobará el siguiente condición.

   d) Si existe algún socket que esté en proceso de cambio de frecuencia, la frecuencia a la que cambiará próximamente será mayor a la requerida por la tarea y además tenga algún núcleo no reservado, se reservará dicho núcleo a la tarea encolada (la tarea finalizará antes de lo previsto). En caso contrario se comprobará el siguiente condición.

   e) Si existe algún socket que tenga todos sus núcleos libres y esté funcionando a una frecuencia diferente a la requerida por la tarea, se iniciará un proceso de cambio de frecuencia, y al mismo tiempo se reservará un núcleo de dicho socket para que cuando finalice el proceso se ejecute la tarea.

   f) Si no se cumple ninguna de las condiciones anteriores, la tarea no podrá pasar a ejecutarse, y permanecerá en cola hasta que se cumpla algunas de las condiciones anteriores en siguientes iteraciones del planificador.

Política de ahorro de energía

Como política de ahorro de energía, además de la producida cuando una tarea en cola genera un cambio de frecuencia por el propio diseño de la política de asignación y reserva de núcleos a tareas tal y como se comenta en la Sección 3.5.2, se tiene en cuenta cuándo un socket está libre y puede reducir su frecuencia al mínimo para reducir el consumo.

Si un socket está libre, o en otras palabras, tiene todos sus núcleos libres y está funcionando a una frecuencia mayor a la mínima permitida después de haber finalizado un paso de planificación, deberá reducirse su frecuencia a la mínima permitida para ahorrar energía.

3.5.3. Ejemplos de funcionamiento

En esta sección, tal y como se ha realizado a lo largo del trabajo, se verán algunos ejemplos sobre la ejecución del planificador sobre el algoritmo de Cholesky de 3×3 bloques. Para ello, se ejecuta el simulador con diferentes parámetros: número de sockets y número de núcleos por socket. Los parámetros comunes para todas las ejecuciones son los mostrados en la Tabla 3.7

Algoritmo	Factorización de Cholesky de 3×3 bloques
Rango de frecuencias de proc.	3,33Ghz, 3,00Ghz, 2,67Ghz, 2,33Ghz, 2,00Ghz, 0,00Ghz
Coste de cambio de frecuencia	1 u.t.

Tabla 3.7: Parámetros comunes para las ejecuciones de ejemplo.

A modo recordatorio en la Figura 3.12 se muestra el grafo de dependencias entre tareas del algoritmo de Cholesky de 3×3 bloques después de haber aplicado el *algoritmo de reducción de holguras*.

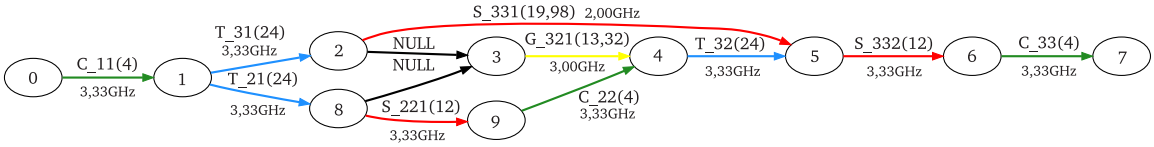


Figura 3.12: Grafo de dependencias de la factorización de Cholesky de 3×3 bloques con las holguras reducidas y las frecuencias recomendadas.

A continuación se pasan a realizar los experimentos:

- **1 socket de 4 núcleos**: En este caso se ha configurado el simulador para que tenga únicamente 1 socket de 4 núcleos, o en otras palabras, un quadcore. Al poderse regular la frecuencia únicamente a nivel de socket, éste permanece permanentemente a la máxima frecuencia, tal y como puede verse en la Figura 3.13. La escala de tiempo empleada para el diagrama es la siguiente: entre dos lineas verticales hay representado 1 u.t.

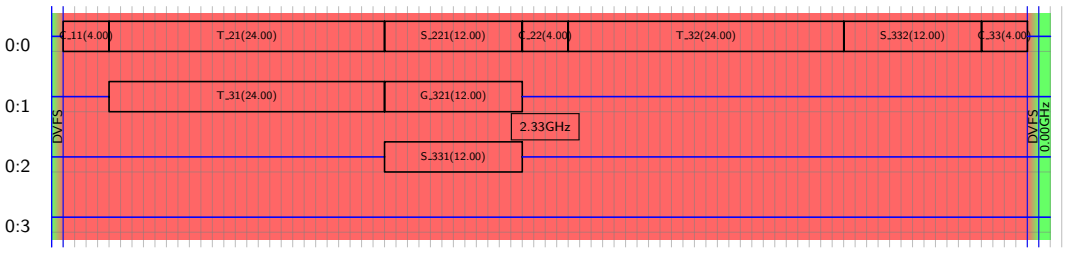


Figura 3.13: Cronograma de ejecución de la factorización de Cholesky de 3×3 bloques con 1 socket de 4 nucleos.

Frecuencia	Socket 0	Media
3,33 GHz	97,67 %	97,67 %
3,00 GHz	0,00 %	0,00 %
2,67 GHz	0,00 %	0,00 %
2,33 GHz	0,00 %	0,00 %
2,00 GHz	0,00 %	0,00 %
0,00 GHz	0,00 %	0,00 %
Total	97,67 %	97,67 %
Cambios de frecuencia	2	2
Porcentaje de tiempo	2,33 %	2,33 %

Tabla 3.8: Porcentajes de tiempo por frecuencia, media total y cambios de frecuencia de la Figura 3.13.

Los porcentajes de tiempo mostrados en la Tabla 3.8 muestran la información de la Figura 3.13. El socket ha permanecido ejecutándose a la máxima frecuencia durante toda la ejecución.

- **2 sockets de 2 núcleos**: En este caso se ha configurado el simulador para que tenga 2 sockets de 2 núcleos, o en otras palabras, un biprocesador de núcleo dual. En esta situación, se regula la frecuencia del segundo socket (ya que no ejecuta tareas críticas), tal y como puede verse en la Figura 3.14.

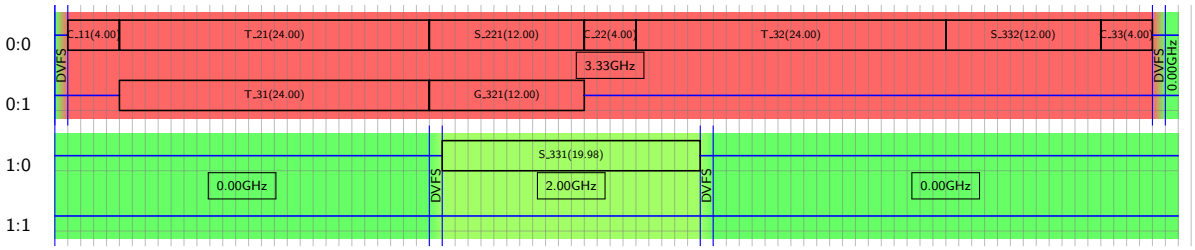


Figura 3.14: Cronograma de ejecución de la factorización de Cholesky de 3×3 bloques con 2 sockets de 2 nucleos.

Frecuencia	Socket 0	Socket 1	Media
3,33 GHz	97,67 %	0,00 %	48,84 %
3,00 GHz	0,00 %	0,00 %	0,00 %
2,67 GHz	0,00 %	0,00 %	0,00 %
2,33 GHz	0,00 %	0,00 %	0,00 %
2,00 GHz	0,00 %	23,23 %	11,62 %
0,00 GHz	0,00 %	74,44 %	37,22 %
Total	97,67 %	97,67 %	97,67 %
Cambios de frecuencia	2	2	2
Porcentaje de tiempo	2,33 %	2,33 %	2,33 %

Tabla 3.9: Porcentajes de tiempo por frecuencia, media total y cambios de frecuencia de la Figura 3.14.

Los porcentajes de tiempo mostrados en la Tabla 3.9 muestran la información de la Figura 3.14. El socket 0 ha permanecido ejecutándose a la máxima frecuencia durante toda la ejecución del algoritmo, mientras que el socket 1 ha permanecido ejecutándose a 2,00 GHz durante únicamente el 23,23 % del tiempo total.

- **4 Sockets de 1 núcleo**: En este caso se ha configurado el simulador para que tenga 4 sockets de 1 núcleos, o en otras palabras, un tetraprocesador de núcleo único. En esta situación, se regula la frecuencia de todos sockets que ejecutan tareas, tal y como puede verse en la Figura 3.15.

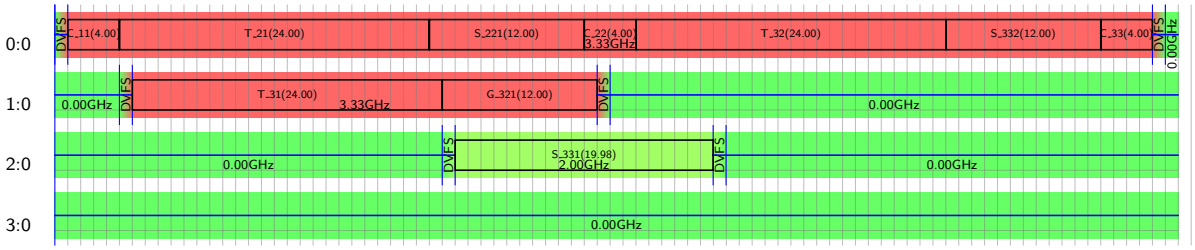


Figura 3.15: Cronograma de ejecución de la factorización de Cholesky de 3×3 bloques con 4 sockets de 1 nucleo.

Frecuencia	Socket 0	Socket 1	Socket 2	Socket 3	Media
3,33 GHz	97,67 %	41,86 %	0,00 %	0,00 %	34,88 %
3,00 GHz	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %
2,67 GHz	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %
2,33 GHz	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %
2,00 GHz	0,00 %	0,00 %	23,23 %	0,00 %	5,81 %
0,00 GHz	0,00 %	55,81 %	74,44 %	100,00 %	57,56 %
Total	97,67 %	97,67 %	97,67 %	100,00 %	98,26 %
Cambios de frecuencia	2	2	2	0	1,5
Porcentaje de tiempo	2,33 %	2,33 %	2,33 %	0,00 %	1,74 %

Tabla 3.10: Porcentajes de tiempo por frecuencia, media total y cambios de frecuencia de la Figura 3.15.

Los porcentajes de tiempo mostrados en la Tabla 3.10 muestran la información de la Figura 3.15. El socket 0 ha permanecido ejecutándose a la máxima frecuencia durante toda la ejecución del algoritmo, mientras que el socket 1 ha permanecido ejecutándose a la máxima frecuencia durante 41,86 % del tiempo total, el socket 2 se ha ejecutado durante el 23,23 % del tiempo total y el socket 3 no se ha empleado, por lo que ha permanecido apagado durante toda la ejecución del algoritmo.

En el Capítulo 4 se evaluarán el comportamiento del *algoritmo de reducción de holguras* y se verificará cómo el simulador que planifica las tareas de los algoritmos de álgebra lineal densa regula la frecuencia de los sockets para ahorrar la mayor energía posible. Concretamente se realizarán experimentos con la descomposición de Cholesky, la descomposición QR y QR por bloques de columnas. Este último algoritmo es un caso particular, pues el coste de cada tarea varía en función de la iteración en la que se encuentra, es por esto que, un mismo tipo de tarea, tiene diferentes costes en diferentes partes del algoritmo.

Capítulo 4

Evaluación de algoritmos

ESTE capítulo abordará la evaluación de los algoritmos y métodos revisados en el Capítulo 3 sobre algoritmos de álgebra lineal densa. Concretamente se realizarán experimentos empleando algoritmos por bloques de descomposiciones en factores de matrices densas tales como Cholesky, QR, y QR por bloques de columnas evaluando a través del simulador de planificador implementado, cuál es el ahorro energético e incremento de tiempo que supone emplear el *algoritmo de reducción de holguras* frente a ejecutar el algoritmo sin ningún tipo de ajuste en las holguras de las tareas, manteniendo a la máxima frecuencia todos los procesadores.

4.1. Descripción de los experimentos

4.1.1. Algoritmos de álgebra lineal densa de entrada

Como algoritmos de entrada al simulador consciente del consumo se emplean los siguientes algoritmos de álgebra lineal densa:

- *Descomposición de Cholesky por bloques*
- *Factorización QR por bloques*
- *Factorización QR por bloques de columnas*

Cada uno de estos algoritmos se evalúa variando los siguientes parámetros:

- **Tamaño de bloque**: Se considerarán los tamaños de bloque 192 y 256.
- **Tamaño de la matriz y número de bloques**: Dado el creciente tamaño del grafo de dependencias cuando el número de bloques aumenta, se ha decido acotar en consecuencia el número de bloques y tamaño de la matriz utilizada de las siguientes formas:
  - *Descomposición de Cholesky y factorización QR por bloques*: En este caso se emplearán matrices cuadradas con tamaños múltiplos de los tamaños de bloque definidos en el ítem anterior. Los tamaños de las matrices variarán entre 3×3 y 8×8 bloques de 192 y 256 elementos. En otras palabras, matrices desde 576×576 hasta 1536×1536 elementos para tamaño de bloque 192 y 768×768 hasta 2048×2048 elementos para tamaño de bloque 256.
  - *Factorización QR por bloques de columnas*: Se emplearán matrices cuadradas con tamaños múltiplos de los tamaños de bloque definidos en el ítem anterior. Los tamaños de las matrices variarán entre 3×3 y 18×18 bloques de 192 y 256 elementos. En otras palabras, matrices desde 576×576 hasta 3456×3456 elementos para tamaño de bloque 192 y 768×768 hasta 4608×4608 elementos para tamaño de bloque 256.

Por otra parte, es interesante destacar que los costes o tiempos de ejecución (u.t.) son proporcionales a los costes teóricos u ordenes de coste de las tareas. En cada algoritmo se especifica cuáles son los órdenes de coste de las tareas, y los tiempos de ejecución en u.t. elegidas en cada caso.

4.1.2. Parámetros del planificador consciente del consumo

- **Tiempo entre cambio de frecuencia:** Se considera que los cambios de frecuencia realizados sobre los sockets tienen un coste de 0.1 u.t.
- **Ratio de exceso:** El parámetro de entrada al algoritmo de reducción de holguras, utilizado en la ecuación (3.11), se emplea durante los experimentos empleando los valores 1 y 1,5. En otras palabras, el valor 1 hace que no se provoquen excesos de tiempo con respecto al algoritmo original, mientras que el valor 1,5, permite al algoritmo realizar excesos del 150 % del tiempo original.
- **Arquitectura simulada**: Para realizar la planificación de tareas la simulación se realiza en plataformas con diferente números de sockets y núcleos, concretamente se emplearán sockets de cuatro núcleos, evaluando resultados con simulaciones que cuentan con 1, 2 y 4 sockets de cuatro núcleos (*quadcores*).
- **Rango de frecuencias**: Se ha supuesto que cada uno de los sockets de los cuales se compone la arquitectura simulada tiene el rango de frecuencias posibles a las cuales pueden trabajar mostrado en la Tabla 4.1.

Frecuencias (GHz)	3,33	3,00	2,67	2,33	2,00	0,00

Tabla 4.1: Frecuencias de procesador consideradas.

4.1.3. Métricas

Durante la obtención de resultados experimentales y la comparación entre ellos, ha sido necesario definir una serie de métricas en forma de parámetros que se explican a continuación:

- **Tiempo de ejecución (T_{ejec}):** Este parámetro representa el tiempo total de ejecución de un algoritmo obtenido tras la simulación, representado mediante unidades de tiempo (u.t.). Este parámetro no debe confundirse con el coste del camino crítico del grafo de dependencias asociado al algoritmo, pues cuando el número de recursos es limitado, en este caso el número de núcleos, el coste del algoritmo se ve afectado, aumentando su tiempo de ejecución.
- **Porcentaje de tiempo de ejecución ($\%T_{ejec}$):** Este parámetro se ha definido con la intención de visualizar, en forma de porcentaje, el incremento de tiempo que supone emplear el algoritmo de reducción de holguras ($a.r.h$) explicado en el Capítulo 3 frente a ejecutar el algoritmo a la máxima frecuencia durante toda su ejecución. Este parámetro se obtiene mediante la siguiente fórmula:

$$\%T_{ejec} = \frac{T_{ejec}^{a.r.h}}{T_{ejec}^{original}} \cdot 100. \tag{4.1}$$

  Por ejemplo, si el algoritmo original se ejecuta en 100 u.t y aplicando el algoritmo de reducción de holguras el tiempo se incrementa a 120 u.t., se dirá que su $\%T_{ejec}$ es del 120 %.
- **Consumo (C):** Este parámetro representa el consumo para la ejecución de un algoritmo obtenido tras la simulación, representado mediante unidades de consumo (u.c.). Sabiendo que el consumo realizado es proporcional a la frecuencia al cubo, es fácil obtener un parámetro de indique el consumo en función del tiempo al cual el procesador ha permanecido trabajando a una frecuencia dada. Si se dispone de un rango de frecuencias discreto $F = \{f_1, \ldots, f_n\}$ a las que pueden trabajar

los procesadores (sockets) y se supone que entre cambios de frecuencia el consumo del procesador es procional a la máxima frecuencia al cubo, el consumo de ejecución se calcula mediante:

$$C = \sum_{i=1}^{n} f_i^3 \, T_{med}(f_i) + f_n^3 \, T_{med}(cambio\_frec). \tag{4.2}$$

En la ecuación (4.2), f_i representa la frecuencia i-ésima posible del procesador, $T_{med}(f_i)$ representa el tiempo medio entre sockets funcionando a frecuencia f_i y $T_{med}(cambio\_frec)$ representa el tiempo medio entre sockets utilizado para realizar cambios de frecuencia.

En caso de que no se emplee el algoritmo de reducción de holguras y, por tanto, el algoritmo se ejecute todo el tiempo a la máxima frecuencia, el consumo se calculará mediante:

$$C_{original} = f_n^3 \, T_{med}(f_n). \tag{4.3}$$

- **Porcentaje de consumo ($\%C$):** Este parámetro se ha definido con la intención de visualizar, en forma de porcentaje, la reducción del consumo que supone emplear el algoritmo de reducción de holguras ($a.r.h$) explicado en el Capítulo 3 frente a ejecutar el algoritmo a la máxima frecuencia durante toda su ejecución sin ningún tipo de herramienta de ahorro de energía.

$$\%C = \frac{C_{a.r.h}}{C_{original}} \cdot 100. \tag{4.4}$$

Por ejemplo, si el algoritmo original consume 100 u.c. y aplicando el algoritmo de reducción de holguras el consumo se reduce a 60 u.c., se dirá que su $\%C$ es del 60 %.

La obtención de resultados se realiza cambiado los parámetros de los algoritmos de álgebra lineal densa y los parámetros del simulador. De esta manera, se podrá evaluar el comportamiento del algoritmo y del planificador consciente del consumo implementado con respecto al ahorro energético generado y el incremento de tiempo que supone su utilización en algunos casos.

4.2. Descomposición de Cholesky por bloques

4.2.1. Algoritmo básico

El algoritmo básico de la descomposición de Cholesky por bloques se muestra en el Código 4.1. En el código, $A(r, s)$ referencia al bloque (r, s) de la matriz A. Se considera que la matriz está en compuesta de $b \times b$ bloques, numerados del 1 al b.

```
1  function CholeskyBlocks(A)
2    for( k = 1; k<=b; k++ ) {
3      chol( A(k,k) );
4      for( i = k+1; i<=b; i++ ) {
5        trsm( A(i,k), A(k,k) );
6      }
7      for( i = k+1; i<=b; i++ ) {
8        for( j = k+1; j<=i-1; j++ ) {
9          gemm( A(i,j), A(i,k), A(j,k)' );
10       }
11       syrk( A(i,i), A(i,k), A(i,k)' );
12     }
13   }
```

Código 4.1: Algoritmo por bloques de la factorización de Cholesky.

A modo de ejemplo, se muestra el grafo de dependencias para un tamaño de bloque de 3× 3 en la Figura 4.1.

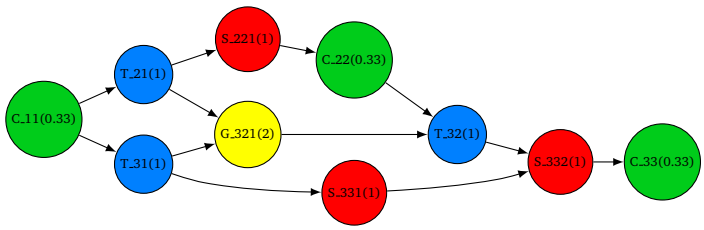


Figura 4.1: Grafo de dependencias entre tareas de la factorización de Cholesky de 3×3 bloques de 192 elementos.

Tarea	Orden de coste	Coste (u.t.) (t.b. 192)	Coste (u.t.) (t.b. 256)
CHOL	$O(b^3/3)$	0,33	0,79
TRSM	$O(b^3)$	1	2,37
SYRK	$O(b^3)$	1	2,37
GEMM	$O(2b^3)$	2	4,74

Tabla 4.2: Tabla de orden de costes y costes en u.t. de las tareas de la factorización de Cholesky.

Los ordenes de coste junto con los tiempos de ejecución (u.t.) en función del tamaño de bloque tomados como parámetros se muestran en la Tabla 4.2. Es interesante resaltar que la variable b en los ordenes de coste representa el tamaño del bloque (192 y 256).

4.2.2. Resultados

Simulación con 1 socket de 4 núcleos

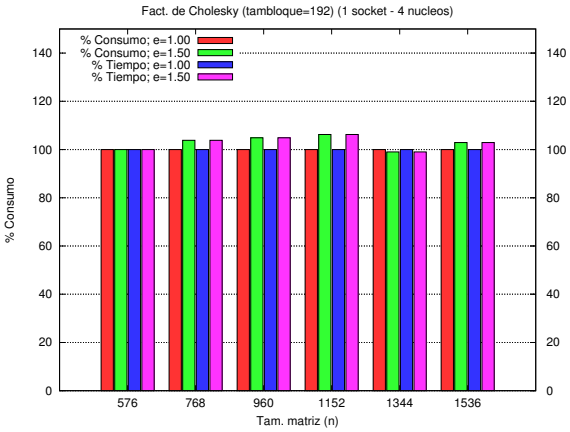


Figura 4.2: Cholesky: tb=192, 1 socket y 4 cores.

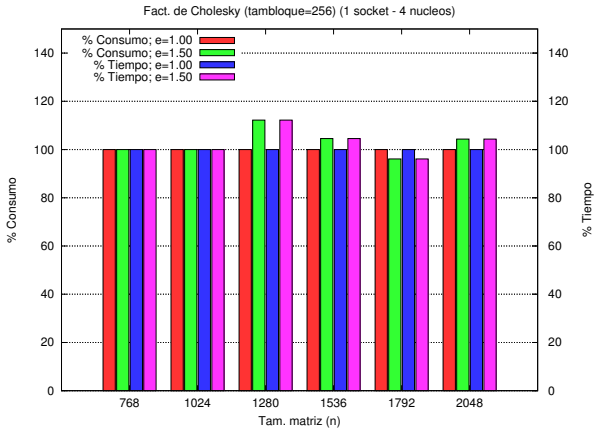


Figura 4.3: Cholesky: tb=256, 1 socket y 4 cores.

Las gráficas de barras obtenidas muestran en el eje X los tamaños de matriz o número de bloques ($n \times n$), en el eje Y izquierdo el porcentaje de consumo (% Consumo) y en el eje Y derecho el porcentaje de tiempo (% Tiempo). En el caso del consumo, las barras con valor de 100 % significan que el consumo aplicando el algoritmo de reducción de holguras es el mismo que si no se aplica el algoritmo. Del mismo modo, si la barra tiene un valor inferior, informará del porcentaje de energía consumida con respecto al algoritmo original. En otras palabras, la barra muestra el porcentaje obtenido a través de la ecuación (4.4). Para el caso del tiempo, una barra con valor de 100 % significa que el tiempo necesario para la ejecución del algoritmo aplicando el algoritmo de reducción de holguras es el mismo que si no se aplica dicho algoritmo. En el caso de que esta barra tenga un valor superior, indicará el sobrecoste que implica el uso de esta técnica de ahorro en algunos casos. El cálculo de estos valores se realiza aplicando la ecuación (4.1).

Los resultados obtenidos cuando se simula la planificación del algoritmo de Cholesky con 1 socket de 4 núcleos son los que se muestran en las gráficas de las Figuras 4.2 y 4.3 para tamaños de bloque de 192 y 256 respectivamente. Para este caso concreto, cuando se utiliza únicamente un socket, se puede observar que el algoritmo de reducción de holguras no aporta apenas beneficios. Estos resultados negativos se deben, por una parte, a que la reducción de frecuencias sólo puede aplicarse a nivel de socket y, por otra, a que el algoritmo evaluado genera un número de tareas que pueden ser ejecutadas en paralelo superior al número de núcleos totales de la plataforma. Por este motivo, el algoritmo de reducción de holguras decide no reducir la frecuencia de las tareas para no sacrificar el tiempo total de ejecución, lo que produce que no se generen ahorros energéticos.

Simulación con 2 sockets de 4 núcleos

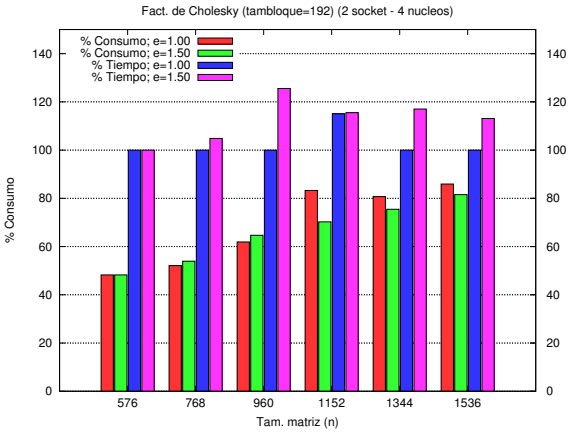


Figura 4.4: Cholesky: tb=192, 2 sockets y 4 cores.

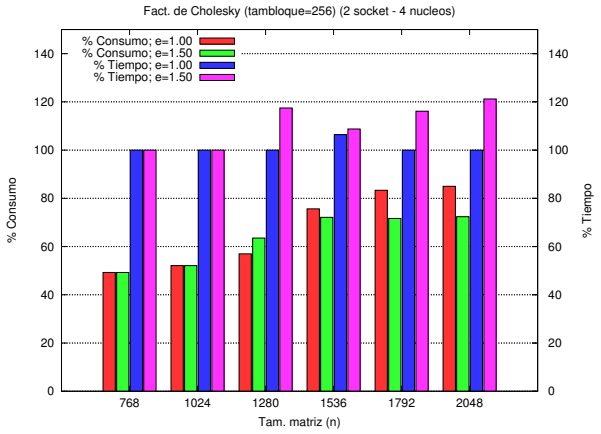


Figura 4.5: Cholesky: tb=256, 2 sockets y 4 cores.

Los resultados obtenidos cuando se simula la planificación del algoritmo de Cholesky con 2 sockets de 4 núcleos son los que se muestran en las gráficas de las Figuras 4.4 y 4.5 para tamaños de bloque de 192 y 256 respectivamente. La simulación con esta configuración de plataforma cambia la perspectiva con respecto al experimento anterior. En este caso, se observa cómo sí se producen considerables ahorros energéticos con apenas un incremento del respecto al tiempo de ejecución original. Las barras roja y verde muestran el porcentaje de consumo con ratios de ajuste de 1 y 1,5 respectivamente. Se puede ver, para los ambos tamaños de bloque que, el ahorro para tamaños pequeños de matriz es de prácticamente el 50 %. Cuando el tamaño de la matriz aumenta (y también el número de bloques), el consumo también lo hace, pues se requieren ejecutar más tareas en paralelo y quedan menos sockets libres a los cuales se les puede reducir la frecuencia.

Por otra parte, el incremento de tiempo generado no supera en ningún caso el 130 % del tiempo original, lo que significa que se puede extender en un pequeño porcentaje el tiempo de ejecución para generar importantes ahorros cuando el tamaño de la matriz es pequeño, y ahorros más reducidos entorno al 20 % cuando el tamaño de la matriz crece. Hay que destacar que, cuando se emplea un ratio de exceso de 1,5, los ahorros generados son mayores pero también lo son los incrementos de tiempo. Será en cada caso particular (según la plataforma y la configuración del algoritmo de entrada) donde se evaluará cuál es el mejor ratio de exceso.

Simulación con 4 sockets de 4 núcleos

Los resultados obtenidos cuando se simula la planificación del algoritmo de Cholesky con 4 sockets de 4 núcleos son los que se muestran en las gráficas de las Figuras 4.6 y 4.7 para tamaños de bloque de 192 y 256 respectivamente. La simulación con esta configuración de plataforma genera ahorros energéticos mayores e incrementos de tiempo menores. Es fundamental fijarse en el número de tareas de la des-

composición de Cholesky que pueden ejecutarse en paralelo. Si este número es mayor que el número de núcleos totales de la plataforma, los resultados de ahorro serán apenas notables. Sin embargo, si el número de tareas que pueden ejecutarse en paralelo es mucho menor que el número de núcleos totales, tanto el consumo como los incrementos de tiempo se verán reducidos, y por tanto favorecerán el comportamiento del algoritmo.

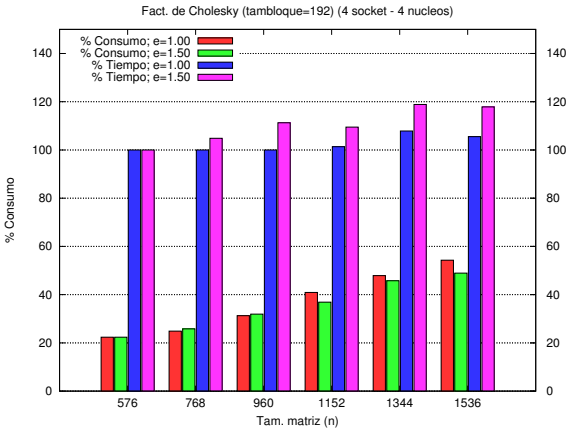


Figura 4.6: Cholesky: tb=192, 4 sockets y 4 cores.

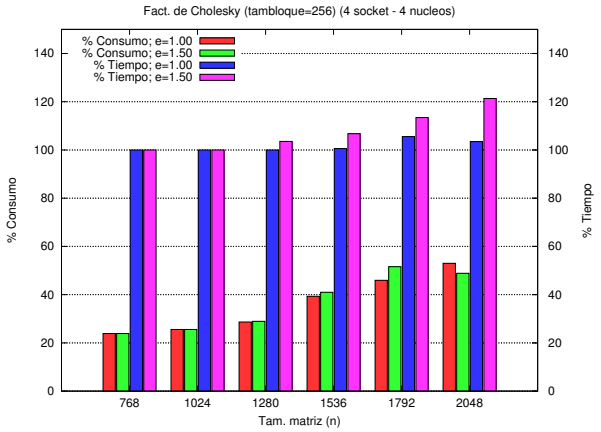


Figura 4.7: Cholesky: tb=256, 4 sockets y 4 cores.

En este caso, se puede ver que para tamaños pequeños de matriz el consumo generado, tanto para tamaños de bloque de 192 y 256, es únicamente del 25 % con respecto a una ejecución sin emplear el algoritmo de reducción de holguras. El máximo consumo, para matrices de mayor tamaños, ronda el 50 %. Si se centra la atención en los porcentajes de tiempo, éstos empiezan a incrementarse cuando los tamaños de matriz crecen, debido a las limitaciones del algoritmo de reducción de holguras: existe un rango discreto de frecuencias por lo que la longitud de subcaminos alternativos no siempre se ajustan perfectamente a la de los subcaminos críticos de referencia. Además, el consumo generado cuando se emplea un ratio de exceso de 1,5 es el 50 % del consumo original para tamaños de matrices grandes, siendo el tiempo de ejecución sólo un 120 % mayor que el tiempo de ejecución del algoritmo sin emplear el algoritmo de reducción de holguras.

4.3. Descomposición de QR por bloques

4.3.1. Algoritmo básico

El algoritmo básico de la descomposición QR (*out-of-core*) por bloques se muestra en el Código 4.2. En el código, $A(r, s)$ referencia al bloque (r, s) de la matriz A. Se considera que la matriz está en compuesta de $b \times b$ bloques, numerados del 1 al b.

```
1  function QR_Blocks(A)
2    for( k = 1; k <= b; k++ ) {
3       qeqrf(A(k,k));
4       for( j = k+1; j <= b; j++ ) {
5           orgqr(A(k,k), A(k,j));
6       }
7       for( i = k+1; i <= b; i++ ) {
8           geqrf2x1(A(k,k), A(i,k));
9           for( j = k+1; j <= n; j++ ) {
10              orgqr2x1(A(i,k), A(k,j), A(i,j));
11          }
12      }
13   }
```

Código 4.2: Algoritmo para la factorización QR *out-of-core*.

A modo de ejemplo, se muestra el grafo de dependencias para un tamaño de bloque de 3× 3 en la Figura 4.8.

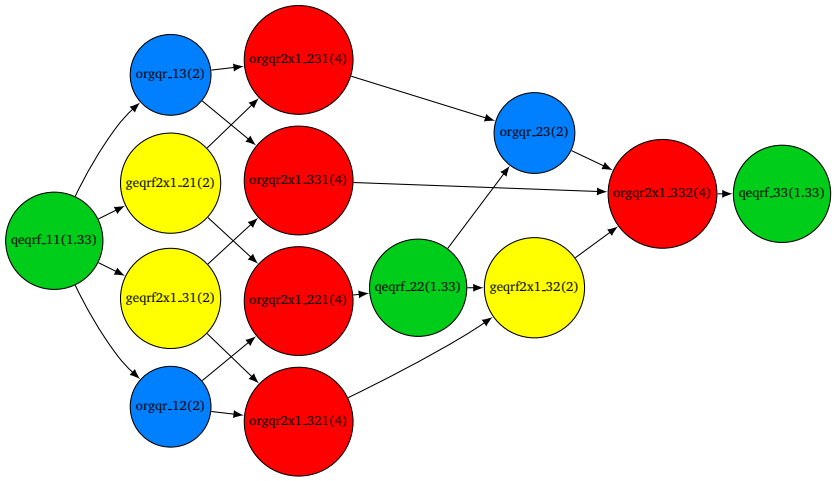


Figura 4.8: Grafo de dependencias entre tareas de la factorización QR de 3×3 bloques de 192 elementos.

Tarea	Orden de coste	Coste (u.t.) (t.b. 192)	Coste (u.t.) (t.b. 256)
QEQRF	$O(4b^3/3)$	1,33	3,16
ORGQR	$O(2b^3)$	2	4,74
GEQRF2X1	$O(2b^3)$	2	4,74
ORGQR2X1	$O(4b^3)$	4	9.48

Tabla 4.3: Tabla de orden de costes y costes en u.t. de las tareas de la factorización QR.

Los ordenes de coste junto con los tiempos de ejecución (u.t.) tomados como parámetros se muestran en la Tabla 4.3 Es interesante resaltar que la variable b en los ordenes de coste representa el tamaño del bloque (192 y 256).

4.3.2. Resultados

Simulación con 1 socket de 4 núcleos

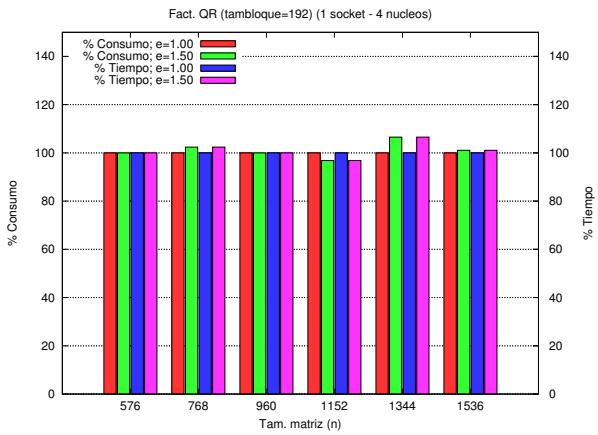


Figura 4.9: QR: tb=192, 1 socket y 4 cores.

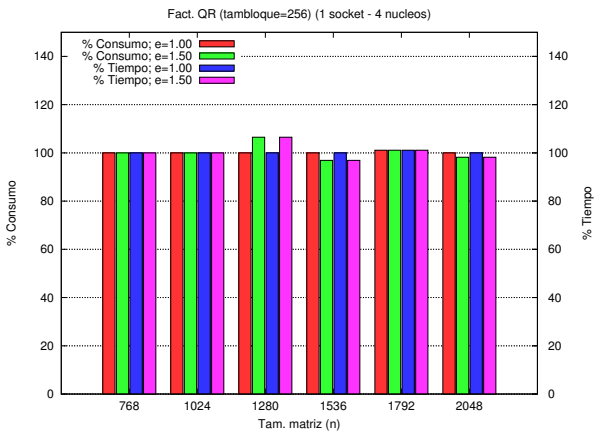


Figura 4.10: QR: tb=256, 1 socket y 4 cores.

Los resultados obtenidos cuando se simula la planificación de tareas del algoritmo QR *out-of-core* con 1 socket de 4 núcleos son los que se muestran en las gráficas de las Figuras 4.9 y 4.10 para tamaños de bloque de 192 y 256 respectivamente. A simple vista, los porcentajes de consumo con respecto a una ejecución sin emplear al algoritmo de reducción de holguras no se ven reducidos, ni tampoco incrementa, en un factor representativo el tiempo de ejecución. Como ya se ha comentado con los resultados de la descomposición de Cholesky estos resultados negativos se producen cuando el número tareas en paralelo que es capaz de ejecutar el algoritmos es mayor que el número total de núcleos de la plataforma. Además, es muy importante resaltar que, tal y como se indicó en la Sección 1.3.1, el algoritmo de reducción de holguras asume que existen recursos infinitos cuando ajusta las frecuencias de las tareas. Por esta razón, cuando se acota el número de recursos disponibles la reducción de consumo que realiza el algoritmo de reducción de holguras no es la óptima.

Simulación con 2 sockets de 4 núcleos

Los resultados obtenidos cuando se simula la planificación de tareas del algoritmo QR *out-of-core* con 2 sockets de 4 núcleos son los que se muestran en las gráficas de las Figuras 4.11 y 4.12 para tamaños de bloque de 192 y 256 respectivamente. Al emplear 2 sockets de 4 núcleos y tener un total de 8 núcleos, la situación es diferente con respecto al caso anterior. Para el caso de matrices de 576×576 los ahorros generados son del 50 %, pues existen recursos suficientes, o en otras palabras, existen más núcleos que tareas del algoritmo QR pueden ser ejecutadas en paralelo. Por esta razón existen sockets que no están siendo utilizados y pueden trabajar a la mínima frecuencia, y además, se reducen las holguras de las tareas que no forman parte de los caminos críticos del grafo de dependencias del algoritmo QR. En estos casos, el algoritmo de reducción de holguras trabaja bajo los supuestos citados en la sección de limitaciones (Sección 1.3.1), es decir, existen suficientes recursos de modo que la reducción de holguras realizada es óptima.

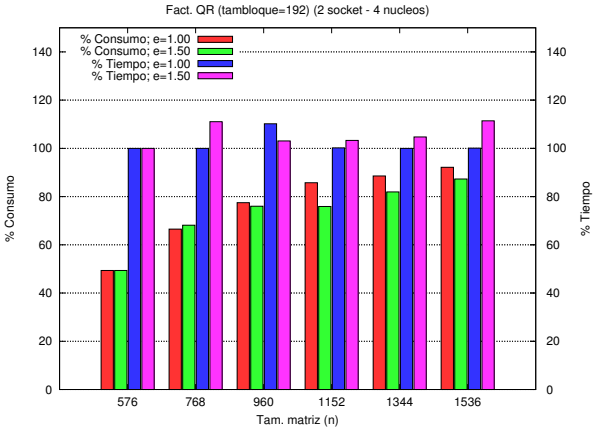


Figura 4.11: QR: tb=192, 2 sockets y 4 cores.

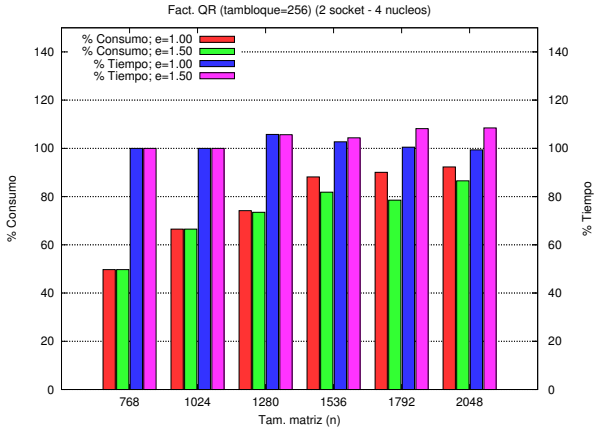


Figura 4.12: QR: tb=256, 2 sockets y 4 cores.

Cuando los tamaños de matrices crecen, tanto para tamaños de bloque de 192 y 256, el porcentaje de consumo con respecto al algoritmo ejecutado a la máxima frecuencia aumenta y no llega a superar en ningún caso el consumo producido por el algoritmo original. Además, cuando se emplea un ratio de exceso de 1,5 los consumos energéticos aún son menores (50 % menos para tamaños de matriz pequeños y 10 % menos para tamaños de matrices grandes). Por otra parte los porcentajes de tiempo únicamente son mayores del 100 % cuando se emplea un ratio de exceso del 1,5, lo que muestra un comportamiento mejor de lo esperado del algoritmo de reducción de holguras, pues el mayor exceso de tiempo producido es aproximadamente del 110 %.

Simulación con 4 sockets de 4 núcleos

Los resultados obtenidos cuando se simula la planificación de tareas del algoritmo QR *out-of-core* con 4 sockets de 4 núcleos son los que se muestran en las gráficas de las Figuras 4.13 y 4.14 para tamaños de bloque de 192 y 256 respectivamente. En este caso los consumos de energía, tanto para tamaños de bloque 192 y 256, aún son menores; del 25 % para tamaños de matriz pequeños y del 50 % para tamaños de matriz grandes con respecto a una ejecución del algoritmo QR ejecutado a la máxima frecuencia. Como ya se ha comentado anteriormente los resultados se ven favorecidos cuando el número de tareas que se pueden ejecutar en paralelo es menor que el número de núcleos totales de la plataforma. En este caso dicha proposición se cumple para tamaños de matriz pequeños, con lo que el algoritmo de reducción de holguras trabaja bajo los supuestos ideales citados en la Sección 1.3.1.

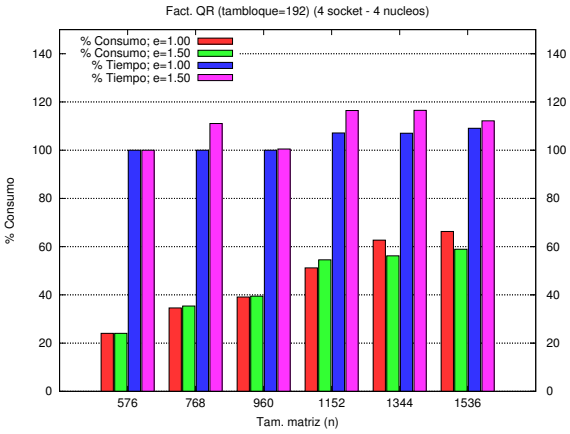


Figura 4.13: QR: tb=192, 4 sockets y 4 cores.

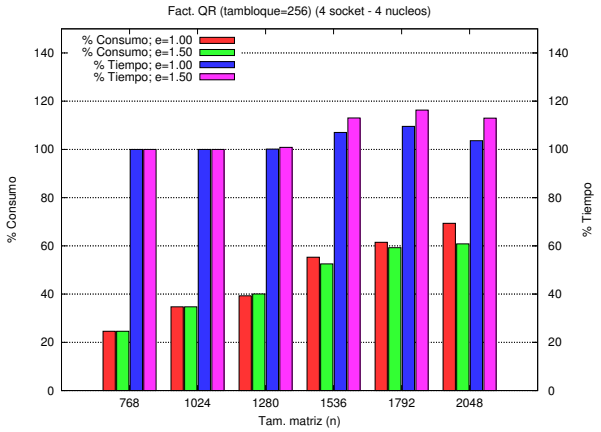


Figura 4.14: QR: tb=256, 4 sockets y 4 cores.

Los porcentajes de tiempo, no obstante, se ven incrementados en un factor no superior al 120 % cuando crece el tamaño de la matriz. Para un ratio de exceso 1,5 los incrementos son mayores que cuando se emplea un ratio de exceso 1. En conclusión, los resultados producidos en estas simulaciones son favorables, pues el consumo se ve reducido a la mitad y el tiempo de ejecución no supera el 120 % con respecto al algoritmo QR ejecutado en la misma plataforma con los sockets trabajando a la máxima frecuencia.

4.4. Descomposición de QR por bloques de columnas

4.4.1. Algoritmo básico

El algoritmo básico de la descomposición QR por bloques de columnas se muestra en el Código 4.3 implementado en Matlab. En este caso n representa el tamaño de la matriz y nb el tamaño de bloque.

```
1  function CholeskyColumnBlocks_Blocks(A, T)
2    k = n;
3    for( i = 1; i<=n; i++ ) {
4      dgeqr2( m-i+1, nb, A(i:m,i:i+nb-1) ) % coste_dgeqr2 = 2*k*(nb*nb*nb-nb*nb);
5      dlarft( m-i+1, nb, T(i:m,1:nb) ) % coste_dlarft = ((3*k-2)*nb*nb*nb-3*nb*nb)/4;
6      for( j = i+1; j<=n; j++ ) {
7          dlarfb( m-i+1, nb, nb, A( i:m, j:j+nb-1 ) ) % coste_dlarfb = (2*k-1)*(nb*nb*nb);
8      }
9      k--;
10   }
```

Código 4.3: Algoritmo para la factorización QR por bloques de columnas.

A modo de ejemplo, se muestra el grafo de dependencias para un tamaño de bloque de 3× 3 en la Figura 4.15.

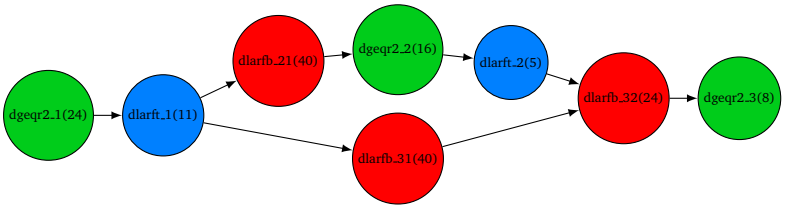


Figura 4.15: Grafo de dependencias entre tareas de la factorización QR por bloques de columnas de 3×3 bloques de 192 elementos.

Tarea	Orden de coste
DGEQR2	$O(2kb^3 - b^2))$
DLARFT	$O(((3k-2)b^3 - 3b^2)/4)$
DLARFB	$O((2k-1)(b^3))$

Tabla 4.4: Tabla de orden de costes y costes en u.t. de las tareas de la factorización de QR por bloques de columnas.

Los ordenes de coste junto con los tiempos de ejecución (u.t.) tomados como parámetros se muestran en la Tabla 4.4 En este caso los costes para tamaños de bloque de 192 y 256 no han sido indicados tal y como se ha realizado con los algoritmos de Chokesly y QR *out-of-core*, puesto que el coste de una tarea depende de la variable b que representa el tamaño de bloque y de la variable k que representa la iteración del algoritmo. Por este motivo, según en la iteración en que se encuentre el algoritmo, una misma tarea podrá tener diferentes costes.

No obstante, el algoritmo de reducción de holguras es capaz de contemplar este caso y reduce, en el mayor factor posible, las holguras de las tareas no críticas del grafo de dependencias del algoritmo QR por bloques de columnas estudiado en esta sección.

4.4.2. Resultados

Simulación con 1 socket de 4 núcleos

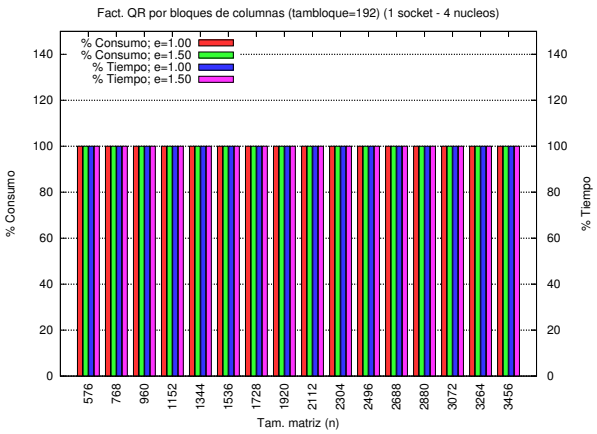


Figura 4.16: QR bloques de columnas: tb=192, 1 socket y 4 cores.

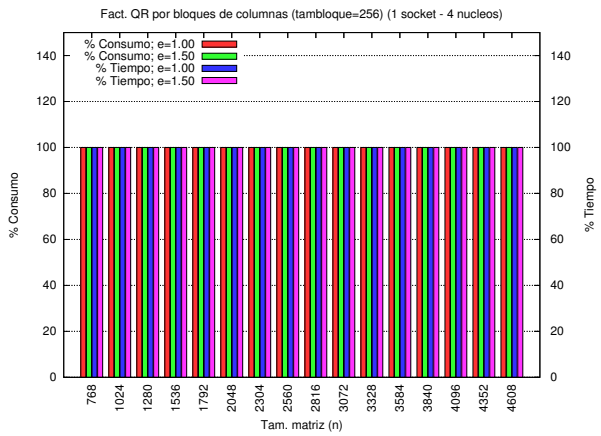


Figura 4.17: QR bloques de columnas: tb=256, 1 socket y 4 cores.

Para este algoritmo, se han realizado simulaciones para tamaños de matriz mayores que con respecto a los algoritmos anteriores (Cholesky y QR *out-of-core*). Los resultados obtenidos cuando se simula la planificación de tareas del algoritmo QR por bloques de columnas con 1 socket de 4 núcleos son los que se muestran en las gráficas de las Figuras 4.16 y 4.17 para tamaños de bloque de 192 y 256 respectivamente. Tal y como puede observarse, el algoritmo de reducción de holguras, tanto para tamaños de bloque de 192 y 256, produce consumos y tiempos de ejecución idénticos a una ejecución del mismo algoritmo sobre la misma plataforma haciendo que los sockets funcionen a la máxima frecuencia.

Estos resultados son debidos a que todos los núcleos de los cores permanecen funcionando a la máxima frecuencia (el algoritmo de reducción de holguras no reduce en ningún caso la frecuencia) y por tanto, tanto el consumo producido como el tiempo de ejecución son iguales que los originales.

Simulación con 2 sockets de 4 núcleos

Los resultados obtenidos cuando se simula la planificación del algoritmo QR por bloques de columnas con 2 sockets de 4 núcleos son los que se muestran en las gráficas de las Figuras 4.18 y 4.19 para tamaños de bloque de 192 y 256 respectivamente. Cuando se emplea esta configuración de la plataforma se tiene un total de 8 cores, con lo que para tamaños de matriz pequeños con tamaños de bloque 192 y 256, los consumos son un 50 % menores y los tiempos de ejecución se conservan con respecto a una ejecución original. Cuando el tamaño de la matriz aumenta, los consumos y el tiempo total de ejecución también lo hacen. La razón de estos excesos es la misma que se comenta en los apartados anteriores, es decir, el algoritmo de reducción de holguras no trabaja bajo las condiciones ideales.

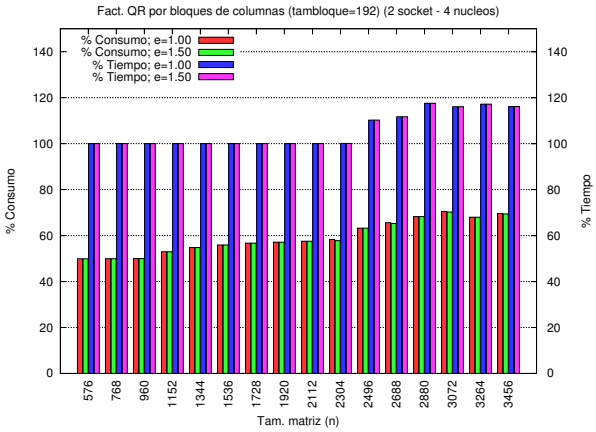


Figura 4.18: QR bloques de columnas: tb=192, 2 sockets y 4 cores.

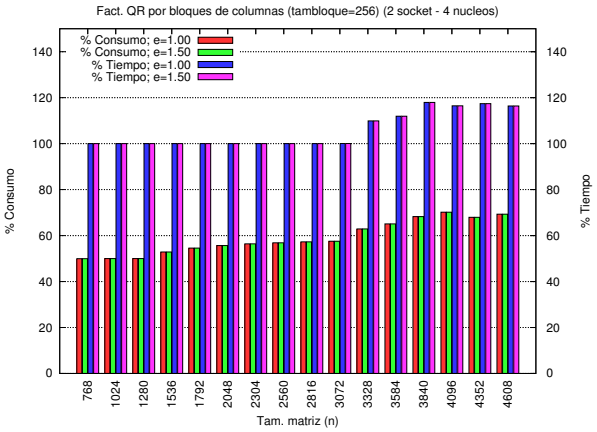


Figura 4.19: QR bloques de columnas: tb=256, 2 sockets y 4 cores.

No obstante, es interesante remarcar que, para un caso concreto, donde sea posible incrementar el tiempo de ejecución, los ahorros de energía producidos pueden resultar más que interesantes para determinadas aplicaciones.

Simulación con 4 sockets de 4 núcleos

Los resultados obtenidos cuando se simula la planificación del algoritmo QR por bloques de columnas con 4 sockets de 4 núcleos son los que se muestran en las gráficas de las Figuras 4.20 y 4.21 para tamaños de bloque de 192 y 256 respectivamente. Con esta configuración de plataforma se pasa a tener 16 núcleos. Esta cantidad de núcleos, mayor que el número de tareas que pueden ser ejecutadas en paralelo en el algoritmo QR por bloques de columnas, produce que el algoritmo de reducción de holguras trabaje bajo condiciones ideales y los ahorros de energéticos sean los máximos posibles. Del mismo modo, hace que no se produzcan incrementos en el tiempo de ejecución.

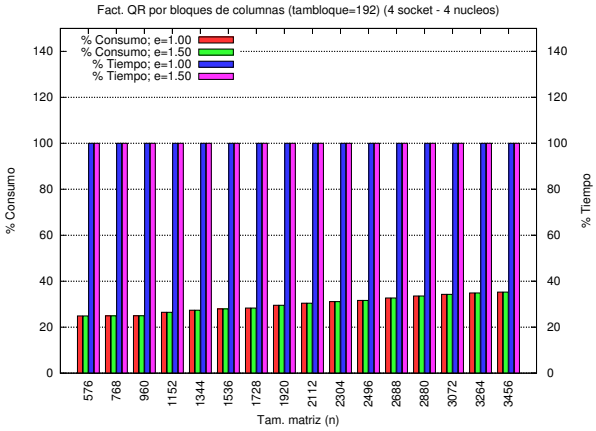


Figura 4.20: QR bloques de columnas: tb=192, 4 sockets y 4 cores.

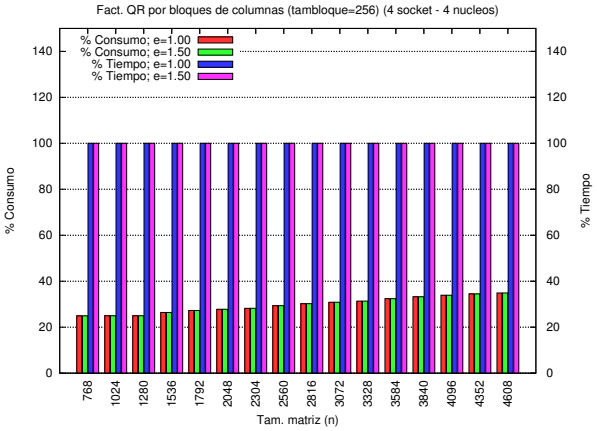


Figura 4.21: QR bloques de columnas: tb=256, 4 sockets y 4 cores.

Sin embargo, hay que se conscientes de las limitaciones que puede sufrir la plataforma donde se ejecute el algoritmo QR por bloques final, con lo que es interesante que, en futuras versiones, el algoritmo de reducción de holguras tenga en cuenta este caso. Es decir, ha de tener en cuenta que pueden existir recursos limitados y las reducciones de holguras para obtener una reducción óptima de la energía consumida pueden cambiar con respecto al número de núcleos totales disponibles.

Otro aspecto a resaltar en este algoritmo, a diferencia del resto de los estudiados, es que, tanto el porcentaje de tiempo consumido para su ejecución como el consumo, es independiente del tamaño de bloque.

Capítulo 5

Conclusiones

En esta tesis de máster se han descrito los conceptos teóricos fundamentales sobre los que se basa el algoritmo de reducción de holguras para tareas de grafos de dependencias en algoritmos de álgebra lineal densa. A continuación se realiza un breve recorrido sobre los aspectos y conceptos tratados en esta memoria.

En el Capítulo 1, se resumen cuales son los avances y el estado del arte en la temática situada en la intersección de la computación de altas prestaciones y el ahorro de energía. Se ponen en relieve las tecnologías, métodos y algoritmos actuales de reducción de consumo energético sobre procesadores multinúcleo y, finalmente, se exponen los objetivos que se pretenden llevar a cabo en este trabajo: un planificador consciente del consumo en algoritmos de álgebra lineal densa. En el Capítulo 2, se explican las metodologías básicas y conceptos fundamentales en los que se basa el algoritmo y planificador consciente del consumo. Se hace especial hincapié en el uso de conceptos y metodologías, provenientes de la teoría de planificación de proyectos, como herramienta para ahorrar energía mediante la expansión de tiempos y la reducción de la frecuencia de los procesadores destinados a ejecutar las tareas no críticas de los algoritmos de entrada. En este caso, se explican conceptos para la construcción del grafos de dependencias entre tareas como vía para confirmar la factibilidad de la aplicación esta metodología en algoritmos de álgebra lineal densa.

En el Capítulo 3, se explica detalladamente cuales han sido los módulos desarrollados y las fases de las cuales se compone el planificador de tareas consciente el consumo. A priori, se dan a conocer algunos conceptos teóricos y se resaltan las ecuaciones básicas que se aplican en el algoritmo implementado. A continuación, se describen las fases principales del algoritmo de reducción de holguras: creación y lectura del grafo, conversión del mismo, aplicación del algoritmo de reducción de holguras y, finalmente, se plantea el funcionamiento del simulador que realiza la planificación en procesadores ficticios de las tareas del grafo. En el Capítulo 4, se evalúan algunos algoritmos comúnmente utilizados en la resolución de sistemas de ecuaciones lineales, concretamente algoritmos encargados de descomponer en factores matrices densas ligadas a dicho sistemas de ecuaciones.

Los algoritmos de Cholesky, QR y QR por bloques de columnas, sirven para verificar la factibilidad del algoritmo de reducción de holguras y planificador implementado. Los resultados son positivos en todos los casos y el ahorro de energía producido es considerable cuando se reducen al máximo posible las holguras entre las tareas. Los algoritmos de Cholesky y QR ofrecen un mayor grado de paralelismo y por tanto prestaciones y ahorros energéticos. En cambio, el algoritmo QR por bloques de columnas tiende a ser más lento, puesto que explota en un menor grado el paralelismo. Este motivo lo convierte en una opción menos deseada, pues obtiene menos rendimiento y el consumo tiende a ser mayor. Por ello, opción interesante de investigación futura es evaluar en qué medida se ahorra energía con respecto a la pérdida de prestaciones. Sin embargo, resulta ser una cuestión abierta dado que, para obtener conclusiones, es necesario realizar experimentos reales en lugar de basarse únicamente en resultados obtenidos en simulaciones.

Por su parte, tal como se comenta en la sección correspondiente, el algoritmo de reducción de holguras, desarrollado en este trabajo, obtiene mejores resultados cuando el nivel de paralelismo es menor

o igual que el número de recursos disponibles en la plataforma. Sin embargo, la técnica no deja de ser válida, siendo una metodología que, a modo de introducción a la temática, es fundamental e imprescindible para continuar la investigación en este contexto. Como trabajo futuro, se plantea continuar en esta línea de investigación con el objetivo de emplear los conceptos aprendidos en este trabajo para transportarlos a planificadores reales, habitualmente incluidos en librerías de computación numérica de altas prestaciones.

Tanto la investigación realizada como los resultados obtenidos resultan ser opciones muy atractivas de cara a la sociedad que requiere, cada día más, de recursos y herramientas que sean capaces de ahorrar energía. Concretamente, en un amplio abanico de aplicaciones en el campo de la computación de altas prestaciones, métodos y algoritmos conscientes del consumo que permitan reducir las emisiones de CO_2 emitidas a la atmósfera son bienvenidos como alternativas hacia el respeto del medio ambiente.

Las impresiones, después de haber llevado a cabo este trabajo, han sido positivas. Por una parte, la satisfacción de haber conseguido los objetivos propuestos, y por otra, la posibilidad de abrir nuevas líneas de investigación que permitan, en un futuro no lejano, la continuidad del trabajo desarrollado en esta tesis de máster para conformar la temática y el contenido de la tesis doctoral que se pretende realizar en los próximos años.

Capítulo 6

Trabajo futuro

En este capítulo, se realiza un repaso al trabajo realizado durante esta tesis del máster y se intenta recopilar, a modo de lista de tareas, las mejoras posibles en el algoritmo de reducción de holguras y simulador implementado. Estas futuras tareas abrirán lineas de investigación que seguirán la temática establecida en este trabajo. Al tratarse de una tesis de máster orientada a la investigación hacía la temática de la tesis doctoral, las mejoras sobre los algoritmos propuestos, nuevas ideas y propuestas citadas en este capítulo de trabajo futuro se tendrán en cuenta cuando se empiece el trabajo orientado a realizar la tesis doctoral.

6.1. Extensiones y mejoras generales

Como se ha dicho, el trabajo realizado en esta tesis de máster ha consistido en una aproximación teórica a la planificación de tareas consciente del consumo. Al ser meramente una aproximación teórica, se ha acotado el número de casos de estudio, pues una planificación de tareas óptima y que, además, tenga en cuenta la energía consumida es un problema NP-completo que solamente puede resolverse a través del diseño de heurísticas que requieren un complejo y costoso estudio previo.

Este trabajo, tal y como se ha citado en la sección de limitaciones (Sección 1.3.1) asume una serie de restricciones que, de cara al futuro, sería interesante incorporarlas como funcionalidades adicionales al planificador consciente del consumo durante la ejecución de aplicaciones y algoritmos de altas prestaciones en situaciones reales.

Algunas de las posibles mejoras a las que se refiere el párrafo anterior son las siguientes:

- El coste de realizar un cambio de frecuencia ha sido tenido en cuenta por el algoritmo de reducción de holguras y por el simulador que planifica las tareas. Este coste ha sido considerado como constante en todos los casos, cuando no es así. Para una mayor exactitud en los resultados obtenidos, hay que tener en cuenta que, el coste de aumentar la frecuencia no es el mismo que de reducirla. Del mismo modo el coste no es el mismo si únicamente se aumenta la frecuencia en 0,33 GHz que si aumenta en 2,00GHz.

  Por esta razón se propone establecer una tabla o *matriz de costes entre cambios de frecuencia* donde la fila i-ésima represente la frecuencia actual y la columna j-ésima la frecuencia destino, siendo el coste de cambio de frecuencia el valor situado en la posición (i, j) de la matriz de cambio de frecuencias. La Tabla 6.1 muestra un posible ejemplo de matriz. En la tabla Tabla 6.1 la posición (i, j) de la matriz, o $C_{i \to j}$ representa el coste de cambiar de la frecuencia i a la frecuencia j.

  Para insertar valores realistas de coste de cambio de frecuencia en dicha tabla, es conveniente tomar tiempos sobre la plataforma que se quiere simular. Es muy probable que, dependiendo de la arquitectura o plataforma que se quiera simular en el simulador, o en un entorno real, se tomen datos reales, de modo que, el algoritmo de reducción de holguras trabaje sobre datos realistas y tome las decisiones más adecuadas posibles.

Frecuencia origen \ Frecuencia destino	0,00 GHz	2,00 GHz	2,33 GHz	2,67 GHz	3,00 GHz	3,33 GHz
0,00 GHz	0	$C_{0\to1}$	$C_{0\to2}$	$C_{0\to3}$	$C_{0\to4}$	$C_{0\to5}$
2,00 GHz	$C_{1\to0}$	0	$C_{1\to2}$	$C_{1\to3}$	$C_{1\to4}$	$C_{1\to5}$
2,33 GHz	$C_{2\to0}$	$C_{2\to1}$	0	$C_{2\to3}$	$C_{2\to4}$	$C_{2\to5}$
2,67 GHz	$C_{3\to0}$	$C_{3\to1}$	$C_{3\to2}$	0	$C_{3\to4}$	$C_{3\to5}$
3,00 GHz	$C_{4\to0}$	$C_{4\to1}$	$C_{4\to2}$	$C_{4\to3}$	0	$C_{4\to5}$
3,33 GHz	$C_{5\to0}$	$C_{5\to1}$	$C_{5\to2}$	$C_{5\to3}$	$C_{5\to4}$	0

Tabla 6.1: Tabla de costes entre cambios de frecuencia.

- Otra posible mejora, ligada a las limitaciones del hardware actual, es considerar que la frecuencia puede cambiarse a nivel de núcleo y no únicamente a nivel de socket, tal como se ha tenido en cuenta durante la realización de este trabajo. Actualmente los procesadores modernos únicamente incorporan la funcionalidad de cambio de frecuencia o DVFS a nivel de socket, no obstante, existen procesadores en fase de investigación, como el *Single-chip Cloud Computer* (SSC) [37] de Intel con 48 núcleos, que permiten ajustar la frecuencia a nivel de núcleo.

  Por esta razón se considera como una posible mejora, que haría que el rendimiento de las aplicaciones se conservara de mejor manera y, al mismo tiempo, se generasen mayores ahorros energéticos.

- El algoritmo principal de este trabajo, la implementación de un algoritmo capaz de reducir las holguras de las tareas, y que al mismo tiempo sea capaz de reducir la frecuencia en consecuencia para reducir el consumo y que el rendimiento de la aplicación se vea afectado lo menor posible, es el objetivo principal de este trabajo.

  No obstante, este algoritmo asume que, cuando se realiza la simulación, los recursos son infinitos, cuando en ocasiones no es así. Esta limitación se produce por el hecho de analizar el grafo de dependencias entre tareas generado directamente por el propio algoritmo, sin conocer a priori cuantos sockets y núcleos se disponen.

  La mejora que se propone en este ítem es sencilla: adaptar el algoritmo de reducción de holguras para que acepte como parámetros la configuración de la plataforma y que, en función de estos datos, sea capaz de ajustar las holguras con los recursos que finalmente se disponen. Para realizar esta mejora, el primer paso a llevar a cabo es obtener un nuevo grafo de dependencias en función de la simulación. Es decir, un grafo obtenido a partir de una simulación previa de las tareas del grafo sobre la plataforma, de modo que, se conserven las dependencias entre tareas originales del algoritmo estudiado y que, además, tenga en cuenta dependencias estructurales impuestas por la plataforma. En esta plataforma es donde finalmente dicho algoritmo se ejecutará y se planificará teniendo como objetivos el ahorro energético y la conservación del rendimiento.

- El simulador asume, en principio, que únicamente se puede cambiar la frecuencia del socket cuando todos los núcleos asociados a éste están libres y no ejecutan ninguna tarea. Esta limitación se ha asumido para reducir el número de casos posibles de planificación. Sin embargo, es posible cambiar la frecuencia de un procesador aunque exista alguna tarea ejecutándose en algún núcleo.

  En este caso se debería tener en cuenta dicha posibilidad, y por lo tanto, ser conscientes que el tiempo de finalización de la tarea que permanecía ejecutándose antes de realizar el cambio de frecuencia se verá, respectivamente, retrasado o adelantado por el hecho de haber reducido o aumentado la frecuencia.

  Además de tener en cuenta esta posibilidad en el simulador planificador consiente de tareas, el algoritmo de reducción de holguras también debe ser capaz de obtener datos acerca de cuándo se debe cambiar la frecuencia del procesador y cómo afecta la duración de las tareas que permanecen ejecutándose en sus núcleos.

- En el trabajo realizado se ha asumido que el coste de las tareas que componen los algoritmos de álgebra lineal densa viene dado por el orden de coste de las tareas. Este aspecto, en la realidad,

no siempre es así. Por esta razón, para obtener simulaciones y resultados más realistas se deben obtener tiempos de cada una de las tareas que componen los algoritmos de álgebra lineal densa ejecutadas a cada una de las posibles frecuencias de procesador, en la plataforma destino, donde posteriormente se realizaría la ejecución del algoritmo.

De este modo, los datos y tiempos obtenidos en una máquina real, serían leídos por el algoritmo de reducción de holguras, obteniendo así, resultados realistas durante la simulación.

6.2. Trabajo futuro

Una de las extensiones más ambiciosos de este trabajo es adaptar los planificadores de *runtimes* de librerías de computación numérica para que hagan uso del algoritmo de reducción de holguras, y que sus planificadores, realicen las funciones de reducción y aumento selectivo de las frecuencias de los procesadores. Esta propuesta, como trabajo futuro, es perfectamente viable en librerías como *libflame* o SuperMatrix, donde se conoce a priori el grafo de dependencias de las tareas del algoritmo además de los datos acerca de la plataforma donde se ejecutan dichos algoritmos.

Esta extensión, al tratarse de una de las más interesantes, una vez realizadas las mejoras comentadas anteriormente, formará uno de los objetivos a llevar a cabo durante el trabajo que se pretende realizar en los próximos años, cuando se realice la tesis doctoral orientada en estas lineas de investigación.

Como se ha comentado a lo largo de toda la memoria, esta tesis de máster conforma una introducción a la temática de la tesis doctoral, por este motivo, únicamente se ha realizado una aproximación teórica a la planificación consciente del consumo de tareas en algoritmos de álgebra lineal densa. Más adelante, este trabajo pretende expandirse y abarcar un mayor número de casos, donde el algoritmo de reducción de holguras, y futuros algoritmos y planificadores conscientes del consumo, puedan aplicarse en un abanico más amplio en el marco de la computación numérica de altas prestaciones.

Bibliografía

[1] Wikipedia. Performance per watt, 2010.

[2] Timo Minartz, Julian M. Kunkel, and Thomas Ludwig 0002. Simulation of power consumption of energy efficient cluster hardware. *Computer Science - R&D*, 25(3-4):165–175, 2010.

[3] Constantine Bekas and Alessandro Curioni. A new energy aware performance metric. *Computer Science - R&D*, 25(3-4):187–195, 2010.

[4] W. Feng and T. Scogland. The green500 list: Year one. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–7. IEEE, 2009.

[5] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. Comput.*, 37:1384–1397, November 1988.

[6] V. Sarkar. *Partitioning and scheduling parallel programs for execution on multiprocessors*. PhD thesis, Stanford, CA, USA, 1987. UMI Order No. GAX87-23080.

[7] Rongheng Li and Huei-Chuen Huang. List scheduling for jobs with arbitrary release times and similar lengths. *J. of Scheduling*, 10:365–373, December 2007.

[8] Abdellatif Mtibaa, Bouraoui Ouni, and Mohamed Abid. An efficient list scheduling algorithm for time placement problem. *Comput. Electr. Eng.*, 33:285–298, July 2007.

[9] C. Hsu and W. Feng. A feasibility analysis of power awareness in commodity-based high-performance clusters. *Cluster 2005*, 2005.

[10] C. Hsu and W. Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 1. IEEE Computer Society, 2005.

[11] Gregor von Laszewski, Lizhe Wang, Andrew J. Younge, and Xi He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *CLUSTER*, pages 1–10, 2009.

[12] Ian Gorton, Paul Greenfield, Alexander S. Szalay, and Roy Williams. Data-intensive computing in the 21st century. *IEEE Computer*, 41(4):30–32, 2008.

[13] W. Feng, A. Ching, and C.H. Hsu. Green Supercomputing in a Desktop Box. In *2007 IEEE International Parallel and Distributed Processing Symposium*, page 352. IEEE, 2007.

[14] G. Chen, K. Malkowski, M. Kandemir, and P. Raghavan. Reducing power with performance constraints for parallel sparse applications. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 11 - Volume 12*, IPDPS '05, pages 231.1–, Washington, DC, USA, 2005. IEEE Computer Society.

[15] R. Ge, X. Feng, W. Feng, and K.W. Cameron. CPU MISER: A performance-directed, run-time system for power-aware clusters. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, page 18. IEEE, 2007.

[16] V.W. Freeh and D.K. Lowenthal. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 164–173. ACM, 2005.

[17] B. Roundtree, DK Lowenthal, SH Funk, VW Freeh, BR de Supinski, and M. Schulz. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. *ACM/IEEE Supercomputing 2007 (SC07)*, 2007.

[18] Maja Etinski, Julita Corbalán, Jesús Labarta, and Mateo Valero. Utilization driven power-aware parallel job scheduling. *Computer Science - R&D*, 25(3-4):207–216, 2010.

[19] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 374–, Washington, DC, USA, 1995. IEEE Computer Society.

[20] A. Manzak and C. Chakrabarti. Variable voltage task scheduling for minimizing energy or minimizing power. In *Proceedings of the Acoustics, Speech, and Signal Processing, 2000. on IEEE International Conference - Volume 06*, pages 3239–3242, Washington, DC, USA, 2000. IEEE Computer Society.

[21] Gu yeon Wei, Jaeha Kim, Dean Liu, Stefanos Sidiropoulos, and Mark A. Horowitz. A variable-frequency parallel i/o interface with adaptive power-supply regulation. *IEEE J. Solid-State Circuits*, 35:1600–1610, 2000.

[22] Flavius Gruian and Krzysztof Kuchcinski. Lenes: task scheduling for low-energy systems using variable supply voltage processors. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 449–455, New York, NY, USA, 2001. ACM.

[23] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 721–725, New York, NY, USA, 2002. ACM.

[24] Jiong Luo and Niraj K. Jha. Power-efficient scheduling for heterogeneous distributed real-time embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(6):1161–1170, 2007.

[25] Jiong Luo, Li-Shiuan Peh, and Niraj Jha. Simultaneous dynamic voltage scaling of processors and communication links in real-time distributed embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 11150–, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Yumin Zhang, Xiaobo Sharon Hu, and Danny Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 183–188, New York, NY, USA, 2002. ACM.

[27] Yves Robert, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors. *High Performance Computing - HiPC 2006, 13th International Conference, Bangalore, India, December 18-21, 2006, Proceedings*, volume 4297 of *Lecture Notes in Computer Science*. Springer, 2006.

[28] Y.C. Lee and A.Y. Zomaya. Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid-Volume 00*, pages 92–99. IEEE Computer Society, 2009.

[29] H. Kimura, M. Sato, Y. Hotta, T. Boku, and D. Takahashi. Emprical study on reducing energy of parallel programs using slack reclamation by DVFS in a power-scalable high performance cluster. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10. IEEE, 2007.

[30] Laura Guitart Tarres. *Problemas de economía de la empresa*. Ana Núñez Carballosa and Universitat de Barcelona. Departament d'Economia i Organització, 2006.

[31] Georgina Tazón Maigre. *Técnica de PERT*. Estadística y Telemática Universidad Rey Juan Carlos Departamento de Informática, 2010.

[32] John Gunnels, Greg Morrow, Beatrice Riviere, and Robert Van De Geijny. Plapack: High performance through high level abstraction. In *In Proceedings of ICPP98*, 1998.

[33] Ernie Chan. *Application of Dependence Analysis and Runtime Data Flow Graph Scheduling to Matrix Computations*. PhD thesis, Department of Computer Science, The University of Texas at Austin, August 2010.

[34] Python Programming Language Official Website. *Python*. http://www.python.org/.

[35] Graph Visualization Software. *Graphviz*. http://www.graphviz.org/.

[36] High productivity software for complex networks. *NetworkX*. http://networkx.lanl.gov/.

[37] Intel Corporation. *Single-Chip Cloud Computer*. http://techresearch.intel.com/ProjectDetails.aspx?Id=1.