Universitat Autònoma de Barcelona

escola d'enginyeria

3911

DISEÑO Y EVALUACIÓN DE UN ALGORITMO PARALELO PARA LA ELIMINACIÓN GAUSIANA EN PROCESADORES MUTI-CORE

Memoria del Proyecto Final de Carrera de Ingeniería en Informática realizado por Sergi Almazán Sánchez y dirigido por Juan Carlos Moure

Bellaterra, 2 de febrero de 2011

El abajo firmante, **Juan Carlos Moure López**

Profesor de la Escuela Técnica Superior de Ingeniería de la UAB,

CERTIFICA:

Que el trabajo que corresponde a esta memoria ha sido realizado bajo su dirección por **Sergi Almazán Sánchez**.

Y para que conste firma la presente,

Firmado: Juan Carlos Moure López

Bellaterra, 1 de febrero de 2011.

Índice

1 Introducción

Dada la necesidad de computar cada vez problemas más grandes en un tiempo razonable, los desarrolladores de sistemas de computación comenzaron a unir sus máquinas existentes para trabajar como una sola. Este fue el inicio de la computación paralela. La computación paralela es la computación que enfatiza el tratamiento concurrente de un conjunto de datos para más de un procesador con el objetivo de resolver un único problema.

La programación paralela es una técnica de programación basada en la ejecución simultánea de varias tareas. Durante la última década se han aplicado sólo a la computación de altas prestaciones (HPC) pero últimamente el interés ha crecido debido a las restricciones físicas que impiden el escalado en frecuencia. A continuación podemos ver en la Figura 1, un diagrama donde se muestra la subdivisión de un programa en tareas para ejecutarlas de forma paralela en diferentes hilos de ejecución y como éstos se sincronizan después de acabar todas las tareas.

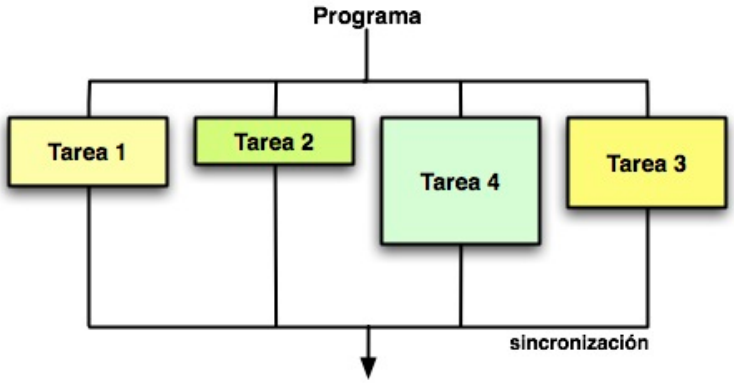

Figura 1. Ejecución de un programa paralelo

Hasta el momento el desarrollo tecnológico había permitido la integración de más transistores en un mismo procesador. Este aumento de la integración, el escalado en frecuencia, el aumento de las operaciones ejecutadas simultáneamente (procesadores superescalares) y la segmentación han otorgado esta capacidad necesaria de cómputo sin cambiar la interficie de programación secuencial.

Las mejoras del rendimiento mediante estas estrategias, empiezan a tocar techo. Este hecho ha provocado que la industria opte por aumentar el número de núcleos de procesamiento secuencial en el mismo chip para obtener futuros incrementos del rendimiento, en vez de diseñar un procesador con una única unidad de proceso más rápida y con mayor capacidad de cómputo. Debido a este cambio en el diseño es necesario un cambio de interficie de programación. Por este motivo nace el término "*multinúcleo*".

1.1 Descripción del trabajo y objetivos

A continuación , se detallan cronológicamente las tareas previstas para la realización del proyecto.

Búsqueda de la información necesaria para la comprensión de los trabajos a realizar.

Aprendizaje y estudio

Estudio y comprensión del hardware

Estudio y comprensión del algoritmo estudiado

Estudio y comprensión de la programación paralela

Estudio y comprensión de las herramientas de extracción de datos

Estudio y comprensión de la programación paralela mediante OpenMP

Instalación, familiarización y pruebas

Codificación

Paralelización y optimización de la aplicación

Codificación de la versión secuencial
Optimización de la versión secuencial
Paralelización de la versión optimizada

Obtención y estudio de resultados

Generación de scripts para la extracción de resultados de la aplicación serie

Generación de scripts para la extracción de resultados de la aplicación paralela

Obtención y estudios de resultados en el sistema A

Obtención y estudios de resultados en el sistema B

Obtención y estudios de resultados en el sistema C

Generación de excels para la generación de gráficas para el estudio

Obtención de resultados extras para corroborar conclusiones

La metodología seguida para llevar a cabo la versión paralela del algoritmo escogido es la siguiente:

- **Obtención Algoritmo secuencial base**: Tras un estudio en profundidad del algoritmo se ha realizado una codificación inicial.

- **Optimización de la versión secuencial**: Después de la verificación del algoritmo base, se ha realizado un estudio del algoritmo serie en busca de los puntos débiles de la aplicación y zonas críticas para aumentar su rendimiento.

- **Estudio de la aplicación serie**: Una vez generadas las diferentes versiones de la aplicación serie con las optimizaciones propuestas. Se ha realizado un estudio del rendimiento de la aplicación ejecutada en los tres procesadores. Finalizado el estudio, se ha escogido la mejor versión para realizar la aplicación paralela.

- **Obtención de la versión paralela**: Después del estudio de la ejecución serie, y escogida la mejor versión, se realiza la paralización de aplicación mediante OpenMP.

- **Estudio de la aplicación paralela**: Con la versión paralela generada, se realiza un estudio en profundidad de la ejecución paralela multihilo en los diferentes procesadores seleccionados para extraer las conclusiones finales.

El objetivo de este proyecto, es el aprendizaje de una metodología de diseño y evaluación de aplicaciones paralelas multithread/multihilo para computadores multicore/multinúcleo. Para conseguir este objetivo, se ha realizado el estudio, optimización y paralelización de un importante algoritmo con infinidad de aplicaciones en el campo de la ingeniería como es el algoritmo de la Eliminación Gaussiana. La Eliminación Gaussiana es un eficiente algoritmo de resolución de sistemas de ecuaciones lineales. Para evaluar la ejecución multinúcleo se ha efectuado un estudio del comportamiento del algoritmo en tres procesadores totalmente diferentes. Dos procesadores fuera de orden con una configuración multicore diferente y un procesador en orden.

1.2 Esquema de la memoria

La memoria está compuesta por ocho capítulos, que se detallan brevemente a continuación.

Capítulo 1: En este capítulo se realiza una breve introducción, se muestran los objetivos del proyecto, la descripción de los trabajos a realizar y la metodología que se seguirá.

Capítulo 2: En este capítulo se definen los conceptos necesarios para la comprensión del proyecto. Se introducen los conceptos tales como la evolución del procesador single-thread, los procesadores multicore multithread, los factores que limitan el rendimiento de los sistemas multicore/multithread, la programación OpenMP y la extracción de datos de los contadores hardware mediante PAPI.

Capítulo 3: En este capítulo se describe el problema de la Eliminación Gaussiana.

Capítulo 4: En este capítulo se realiza el análisis y evaluación del algoritmo secuencial de la Eliminación Gaussiana con pivotaje parcial por filas. Se realiza una primera codificación del algoritmo serie y tras el análisis computacional**,** se realiza una serie de optimizaciones que acaban generando 4 nuevas versiones de la aplicación. Cada una de estas versiones es ejecutada en los sistemas considerados y se extraen los resultados del rendimiento. Como resultado de este capítulo, se identifica la versión secuencial con un mejor rendimiento que se usará como base en el capítulo siguiente para la versión paralela.

Capítulo 5: En este capítulo se realiza el análisis y diseño del algoritmo paralelo. Primeramente se realiza un estudio de la versión serie y su división en tareas independientes para ser ejecutadas simultáneamente y se diseña el algoritmo paralelo. A continuación se efectúa un análisis computacional del algoritmo independiente del procesador. Por último se ejecuta la versión paralela en los diferentes sistemas considerados para la extracción de resultados del rendimiento. Tras la muestra de los resultados se extraen las conclusiones del estudio.

Capítulo 6: En este capítulo se describe, de forma cronológica, los diferentes problemas encontrados a lo largo del proyecto y lo que se ha aprendido de ellos.

Capítulo 7: En este capítulo se incluyen las conclusiones finales del proyecto, las líneas futuras de investigación y ampliaciones.

Capítulo 8: Bibliografía.

Capítulo 9: Anexo I.

2 Fundamentos teóricos

En este capítulo se describen los conceptos básicos se emplearan durante el desarrollo del proyecto. Se explicará la evolución del procesador desde single-thread hasta llegar al procesador multicore. Finalmente se realizará una breve introducción a la programación mediante OpenMP y la extracción de prestaciones mediantes contadores hardware con PAPI.

2.1 Evolución del procesador single-thread

En un procesador clásico podemos dividir el proceso de ejecución de una instrucción en cuatro etapas:

1. Búsqueda de la instrucción (prefetch y fetch)
2. Decodificación de la instrucción
3. Búsqueda de os operandos
4. Ejecución de la instrucción y escritura de los resultados.

Para cada etapa se emplea un tiempo denominado un ciclo. En la figura 2 se muestra las diferentes ciclos que emplea un procesador clásico en ejecutar dos instrucciones. Se puede apreciar que hasta que la primera instrucción no finaliza, no se puede realizar la búsqueda de la segunda instrucción.

Ciclos	1	2	3	4	5	6	7	8
Búsqueda Instrucción	INST 1				INST 2			
Decodificación Instrucción		INST 1				INST 2		
Búsqueda operandos			INST 1				INST 2	
Ejecución Instrucción				INST 1				INST 2

Figura 2. Etapas empleadas para la ejecución de dos instrucciones en un procesador clásico.

El tiempo de ejecución de un programa (T_e) se puede expresar con la siguiente ecuación de rendimiento:

$$T = N \times CPI \times t$$

donde **N** es el número total de instrucciones ejecutadas por el programa, **CPI** es el número de ciclos por instrucción capaz de ejecutar el procesador y **t** el tiempo de un ciclo de reloj del procesador.

Para minimizar el tiempo de ejecución existen dos alternativas que son, reducir el número de instrucciones y/o reducir el CPI.

2.1.1 Procesamiento segmentado

En el procesamiento segmentado se adopta una nueva estrategia con el objetivo de disminuir el tiempo medio de ejecución por instrucción de una aplicación. Se divide internamente el computador en segmentos individuales, cada uno especializado en una de las etapas.

A diferencia del procesador clásico donde todas las etapas tenían que completarse antes de buscar la instrucción siguiente, ahora la existencia de segmentos especializados permite el solapamiento en la ejecución de las instrucciones. Así, un segmento puede empezar a trabajar con una nueva instrucción sin la necesidad de que la instrucción anterior haya finalizado todas las etapas.

El resultado es un aumento del numero de instrucciones ejecutadas por ciclo. Como se observa en la Figura 3, con la ejecución segmentada de instrucciones, un procesador segmentado de K etapas en un caso ideal pasa de ejecutar una instrucción cada K ciclos a una instrucción por ciclo. No obstante, los saltos y las dependencias de datos limitan esta ganancia. En el caso de los saltos, las instrucciones de bifurcación condicional pueden derivar en diferentes caminos a seguir por lo que no es posible conocer la siguiente instrucción a ejecutar hasta que no se haya ejecutado completamente la instrucción de salto. En el caso de dependencia de datos una instrucción puede quedar detenida en una etapa a la espera del dato que otra instrucción anterior debe calcular.

Ciclos	1	2	3	4	5	6	7	8
Búsqueda Instrucción	INST 1	INST 2	INST 3	INST 4	INST 5			
Decodificación Instrucción		INST 1	INST 2	INST 3	INST 4	INST 5		
Búsqueda operandos			INST 1	INST 2	INST 3	INST 4	INST 5	
Ejecución Instrucción				INST 1	INST 2	INST 3	INST 4	INST 5

Figura 3. Ejecución segmentada de varias instrucciones.

2.1.2 Memoria Cache

La memoria cache es una memoria de alta velocidad y menor tamaño que la memoria principal, situada entre el procesador y la memoria principal. Su misión es almacenar temporalmente los contenidos de la memoria principal que estén siendo utilizados por el procesador. De esta forma se reduce el número de operaciones de acceso a memoria principal, disminuyendo el número de ciclos dedicados a acceso a memoria, y reduciéndose por tanto el CPI.

La posibilidad de aumentar las prestaciones incorporando memoria cache se debe a la localidad temporal y a la localidad espacial. Estas son características del comportamiento de las aplicaciones. En el caso de la localidad temporal existe una alta probabilidad de acceder a posiciones de memoria a las que ya se ha accedido anteriormente. En el caso de la localidad espacial existe una alta probabilidad que de una vez accedida a una posición de memoria, se acceda a las posiciones de memoria cercanas, en un futuro inmediato.

[1]

2.1.3 El procesador superescalar

Un procesador superescalar es capaz de ejecutar más de una instrucción en cada etapa del pipeline del procesador. El número máximo de instrucciones en cada etapa depende del número y del tipo de las unidades funcionales de que disponga el procesador. Sin embargo, un procesador superescalar sólo es capaz de ejecutar más de una instrucción simultáneamente si las instrucciones no presentan ningún tipo de dependencia. Las dependencias que pueden aparecer son:

- **Dependencias estructurales**: cuando dos instrucciones requieren el mismo tipo de unidad funcional pero su número no es suficiente.

- **Dependencias de datos**:
  - Lectura después de Escritura: cuando una instrucción necesita el resultado de otra para ejecutarse.
  - Escritura después de Lectura o Escritura: cuando una instrucción necesita escribir en un registro sobre el que otra instrucción previamente debe leer o escribir.

- **Dependencias de control**: cuando existe una instrucción de salto que puede variar la ejecución de la aplicación

Podemos distinguir diferentes tipos de procesadores por la forma de actuar ante una dependencia estructural o de datos. En un procesador con ejecución **en orden** las instrucciones quedarán paradas a la espera de que se resuelva la dependencia. Mientras que en un procesador con ejecución **fuera de orden** las instrucciones dependientes quedarán paradas pero será posible solapar parte de la espera con la ejecución de otras instrucciones independientes que vayan detrás.

En el caso de las dependencias de control, se conoce como **ejecución especulativa** de instrucciones a la ejecución de instrucciones posteriores a la instrucción de salto (antes de que el contador del programa (PC) llegue a la instrucción de salto).

[2-3]

2.2 Limitaciones del procesador single-thread

A continuación describiremos una serie de factores que limitan el rendimiento de la ejecución de una aplicación en un procesador single-thread.

2.2.1 Problema de la memoria

La diferencia de velocidad entre procesador y memoria, limita el rendimiento del procesador. Las operaciones de memoria son lentas comparadas con la velocidad del procesador. Los accesos a memoria, por ejemplo en un fallo de cache, pueden consumir de 100 a 1000 ciclos de reloj, durante los cuales el procesador debe esperar a que el acceso a memoria finalice.

Por tanto, un aumento de la frecuencia de reloj del procesador sin incrementar la velocidad de la memoria solamente mejoraría el rendimiento en un pequeño porcentaje. Los ciclos de cómputo se realizarían más rápido pero el tiempo de acceso a memoria continuaría siendo el mismo. Esto se puede apreciar en la figura 4 que representan las fases de ejecución de un programa single-thread en dos procesadores con distinta frecuencia de reloj. En la figura, el procesador 2 tiene el doble de frecuencia de reloj que el procesador 1 y consume los mismos ciclos en el acceso a memoria que el procesador 1.

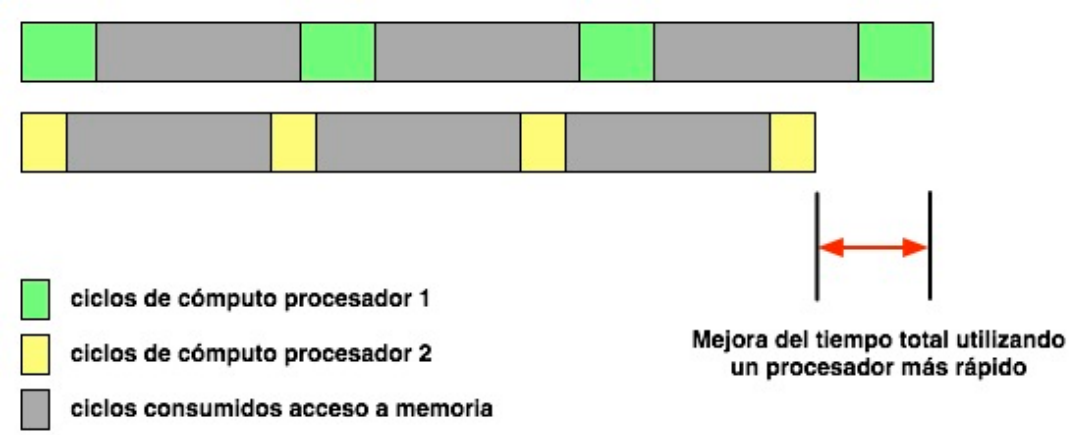

Figura 4. Fases de ejecución de un programa single-thread en dos procesadores con distinta frecuencia de reloj.

Además hay que tener en cuenta que a los cientos de ciclos que se consumen en cada acceso a memoria hay que sumarle decenas de ciclos extra, por cada acceso a nuevos niveles de cache (provocados por fallos en el nivel anterior).

La solución para aprovechar los ciclos en los que el procesador esta esperando a que finalice la operación de memoria es el "*multithreading*" por hardware. El "*multithreading*" por hardware es una propiedad que permite al procesador alternar de un hilo a otro hilo cuando el hilo que esta ocupando el procesador queda parado. Esta solución se analizará en profundidad más adelante.

2.2.2 Calor y coste asociado

El incremento de la frecuencia de reloj del procesador implica un aumento de la potencia consumida y del calor generado. En la actualidad los altos valores de frecuencia de reloj de los procesadores suponen un problema, tanto económico por el consumo eléctrico, y gasto dedicado a la disipación del calor y refrigeración del procesador, como tecnológico por la dificultad para disipar la gran cantidad de calor generado, de la pequeña superficie de un procesador. Por estos motivos, se ha abandonado la idea de aumentar la frecuencia de reloj del procesador para aumentar el rendimiento, y se ha optado continuar aumentando el nivel de integración sin aumentar la frecuencia de reloj y en el mismo espacio añadir más procesadores en un mismo chip. Con esta solución el calor se incrementa de forma lineal y no exponencial como ocurre con el aumento de frecuencia de reloj.

2.3 El procesador Multi-thread

Para entender conceptos como el *multithreading* por software, el *multithreading* por hardware y las ventajas de los procesadores multi-thread/multi-hilo es necesario conocer la diferencia entre hilo y proceso.

2.3.1 Thread/Hilo y Proceso

Un proceso es secuencia de código ejecutable en ejecución. Cada proceso posee un espacio de direccionamiento propio para almacenar sus estructuras de datos asociados. Un proceso esta formado por uno o más threads o hilos de ejecución. Un hilo es la unidad mínima de procesamiento. Los hilos existen dentro de un proceso y comparten recursos como el espacio de memoria, la pila de ejecución y el estado de la CPU. Un proceso con múltiples hilos tiene tantos flujos de control como hilos. Cada hilo se ejecuta con su propia secuencia de instrucciones de forma concurrente e independiente.

2.3.2 Multi-threading por Software

Debemos distinguir entre multi-programación y aplicaciones paralelas.

La multi-programación es una técnica de planificación que permite tener varios hilos (o procesos) en estado de ejecución. Los hilos comparten los recursos del sistema como la memoria principal y el procesador. Existe la falsa apariencia de que los hilos se están ejecutando simultáneamente, esto es debido a que el sistema operativo es de tiempo compartido. En un escenario de tiempo compartido cada hilo se ejecuta durante un breve intervalo de tiempo. El cambio de contexto se realiza lo suficientemente rápido como para simular la ejecución de varios hilos simultáneamente.

En la figura 5 se observa como el Hilo 1, se para al producirse una excepción que requiere esperar un tiempo. Inmediatamente el Sistema Operativo cambia de contexto y ejecuta otro hilo. Cada columna de la figura representa una instrucción de las cuatro que se pueden llegar a ejecutar en cada ciclo (ejecución superescalar de grado 4).

Figura 5: Cambio de Hilo del SO en un entorno multiprogramado.

El *multithreading* por software posibilita la realización de aplicaciones paralelas. Es un nuevo modelo de programación que permite a múltiples hilos existir dentro de un proceso. Los hilos comparten los recursos del proceso pero se ejecutan de forma independiente. El hecho de que sean independientes permite la concurrencia (es decir su ejecución simultanea), y si el procesador lo soporta se podrán ejecutar en paralelo.

Las ventajas de realizar la concurrencia a nivel de hilo en lugar de a nivel de proceso son varias. Los hilos se encuentran todos dentro de un mismo proceso y por lo tanto pueden compartir los datos globales. Además, una petición bloqueante de un hilo no parará la ejecución de otro hilo. Por último, si el procesador lo soporta, los diferentes hilos están asociados a diferentes conjuntos de registros, por lo que el cambio de contexto del procesador podrá realizase de forma eficiente.

2.3.3 Multithreading por Hardware

El *multithreading* por Hardware es una técnica que incrementa la utilización de los recursos del procesador. A continuación se analizarán diferentes tipos de multi-threading por Hardware.

Large-Grain Multithreading

En el modelo *large-grain multithreading* el procesador ejecuta el hilo de forma habitual y solamente realiza un cambio de contexto cuando ocurre un evento de larga duración (como un fallo de caché). Para que el cambio de contexto sea eficiente es necesario que exista una copia del estado de la arquitectura (PC, registros visibles) para cada hilo. Este método tiene la ventaja de ser sencillo de implementar.

En la Figura 6 se muestra la ejecución de las los diferentes hilos en la que el procesador cambia de hilo cuando se produce un evento que requiere un tiempo de espera por ejemplo un fallo de cache.

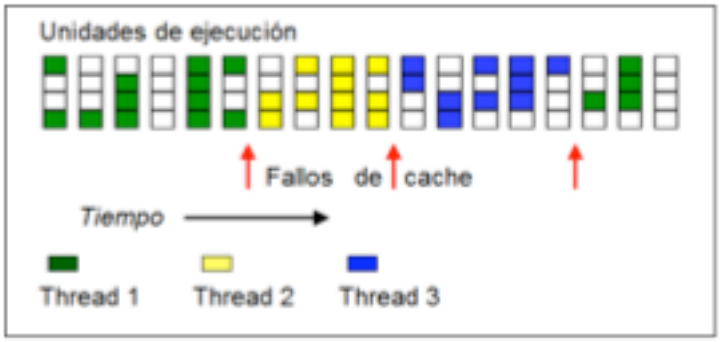

Figura 6: Ejecución de hilos siguiendo el modelo Large-Grain Multithreading.

Fine-grained Multithreading

El *fine-grained multithreading* se basa en un cambio "rápido" entre hilos, ejecutando en cada ciclo un hilo diferente. Es un mecanismo que tiene como base una planificación de la ejecución de las instrucciones en orden. Con el fin de evitar largas latencias por hilos bloqueados, se ejecutan instrucciones de diferentes hilos. Este enfoque tiene la ventaja de eliminar las dependencias de datos que paran el procesador. Al pertenecer las instrucciones a diferentes hilos, las dependencias de datos y de control desaparecen.

En la figura 7 se muestra como en cada ciclo se ejecutan instrucciones de hilos diferentes. El procesador no espera a que se produzca una interrupción (fallo de caché,....) para saltar al siguiente hilo.

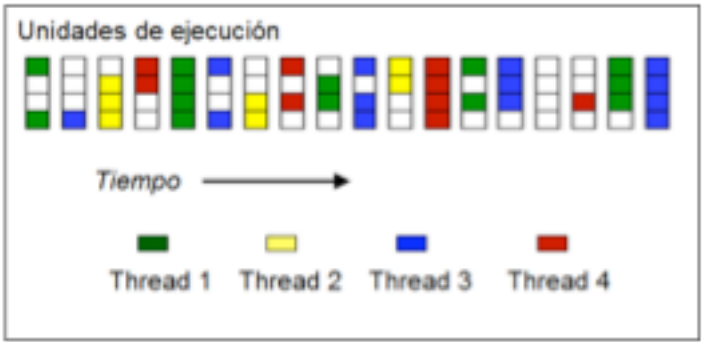

Figura 7: Ejecución de hilos siguiendo el modelo Fine-grained Multithreading.

Simultaneous Multithreading

La filosofía de *simultaneous multithreading* (SMT) consiste en poder ejecutar instrucciones de diferentes hilos, en cualquier momento y en cualquier unidad de ejecución. Desarrollar esta tecnología requiere un hardware adicional para toda la lógica. Como consecuencia, su realización para un gran número de hilos aumentaría la complejidad y, por tanto el coste. Por este motivo en las implementaciones SMT se opta por reducir el número de hilos.

Simultaneous multithreading muestra a un procesador físico como dos o más procesadores lógicos. Los recursos físicos son compartidos y el estado de la arquitectura es copiada para cada uno de los dos procesadores lógicos. El estado de la arquitectura está formado por un conjunto de registros: registros de propósito general, registros de control, registros del controlador de interrupciones (APIC) y registros de estado.

Los programas verán a los procesadores lógicos como si se tratara de dos o más procesadores físicos diferentes. Sin embargo, desde el punto de vista de la microarquitectura, las instrucciones de los procesadores lógicos se ejecutarán simultáneamente compartiendo los recursos físicos.

2.4 El procesador Multi-core

Los procesadores multi-core o multi-núcleo combinan dos o más procesadores, a los que nos referiremos como núcleos o cores, en un mismo chip. Estos procesadores mejoran el rendimiento de las aplicaciones paralelas. Las aplicaciones paralelas están compuestas por múltiples hilos independientes, de forma que es posible la concurrencia. Es decir, los hilos se pueden ejecutar al mismo tiempo y en paralelo.

Como consecuencia el rendimiento de las aplicaciones paralelas puede teóricamente escalar linealmente con el número de procesadores. En la práctica existen factores que lo impiden, como los overheads por creación/eliminación de hilos, las comunicaciones entre las memorias de los procesadores, y el posible desbalanceo en las aplicaciones del volumen de cómputo por hilo (hilos esperando a que otros hilos finalicen). Describiremos estos factores con más detalle, más adelante. Otra de las ventajas es poder aplicar el diseño de un núcleo ya probado a los demás núcleos del procesador. Así se evita el coste que supondría el diseño de nuevos núcleo. Como inconveniente, las aplicaciones han de ser correctamente paralelizadas para aprovechar todo el potencial de los procesadores multi-núcleo.

2.4.1 Arquitectura de los procesadores multi-core

Los procesadores de una arquitectura multi-núcleo comparten la memoria principal. Existen dos alternativas para compartir la memoria principal, el acceso uniforme a memoria y el acceso no uniforme a memoria.

En la primera los procesadores del sistema tienen el mismo tiempo de acceso a memoria, a esta arquitectura se la conoce como arquitectura UMA (Acceso uniforme a memoria). En la segunda el acceso parcial o total a la memoria es controlado por un único procesador, lo que provoca que este procesador tenga un tiempo de acceso menor, a la memoria controlada por él, que el resto de procesadores. El resto de procesadores debe interactuar con el procesador que controla la memoria para acceder a ella, a esta arquitectura se la conoce como arquitectura NUMA. (Acceso no uniforme a memoria).

En cuanto a la organización de las memorias caché de un multi-núcleo existen varias opciones. En algunas arquitecturas se opta por mantener todos los niveles de cache privados a cada núcleo, mientras que en otras arquitecturas se comparte el último nivel de cache.

Las figura 8 muestra el esquema de un procesador dual-core, en el que las memorias cache de nivel 1 de instrucciones y de datos son privadas, la memoria cache de nivel 2 es compartida, y el acceso a memoria principal es de tipo UMA. En la misma figura, se muestra el esquema de un procesador quad-core, en el que las memorias cache de nivel 1 y de nivel 2 son privadas, la memoria cache de nivel 3 es compartida, y el acceso a memoria principal es de tipo UMA.

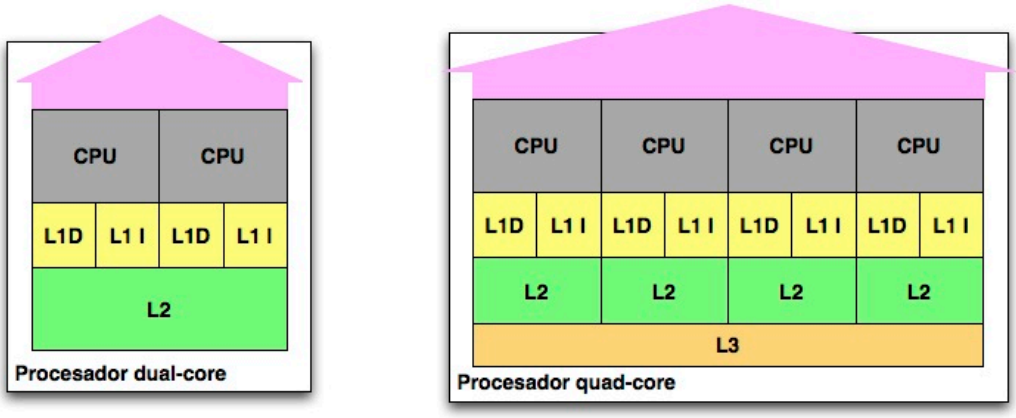

Figura 8. Esquema de los procesadores de doble y cuádruple núcleo

La figura 9 muestra el esquema de una arquitectura NUMA, formada por dos procesadores dual-core. Cada uno de los procesadores accede directamente a su memoria principal y para acceder a la memoria principal del otro procesador tiene que interactuar con él. Como en la figura 8, los núcleos de cada uno de los procesadores tienen memorias cache de nivel 1 privadas y comparten la memoria cache de nivel 2 y la memoria principal que corresponde al procesador.

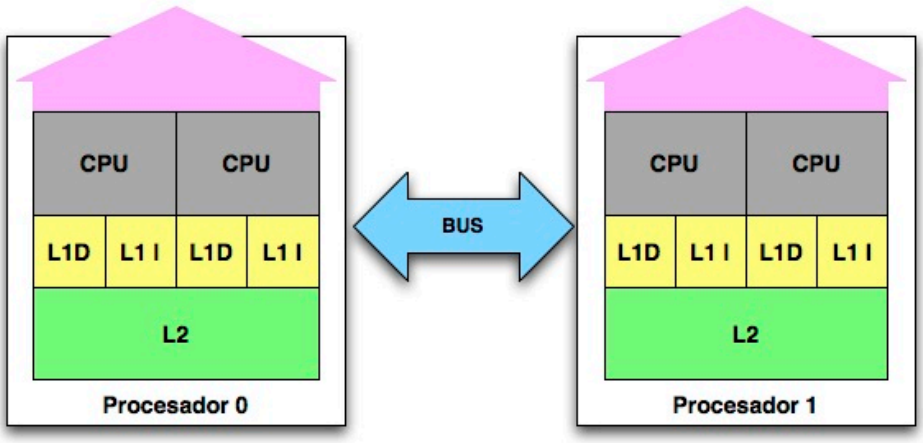

Figura 9. Esquema de la arquitectura NUMA con dos procesadores dual-core

Las principales ventajas de compartir la memoria cache son las siguientes:

1. **Uso eficiente del último nivel de cache:** Si un núcleo está inactivo el otro núcleo puede utilizar toda la cache compartida. Si los dos núcleos trabajan en paralelo, la memoria cache se reparte proporcionalmente en función de la frecuencia de solicitudes de cada núcleo. Incrementa la utilización de los recursos.

2. **Reducción del almacenamiento de los datos:** Los mismos datos utilizados por dos núcleos que comparten memoria cache, serán almacenados una única vez.

3. **Reducción de la complejidad de la coherencia en la memoria cache:**
   - Reduce el problema de *false-sharing*
   - Menor carga de trabajo para mantener la coherencia, comparado con la arquitectura de memoria cache privada.

[1]

2.4.2 Procesador SUN UltraSPARC T2

Siguiendo esta línea SUN ha diseñado un conjunto de procesadores multi-núcleo y multi-hilo. De este conjunto estudiaremos el denominado proyecto Niágara II: El UltraSPARC T2. A continuación nos encontramos en la Figura 10, el esquema principal de este procesador multi-núcleo y sus unidades funcionales.

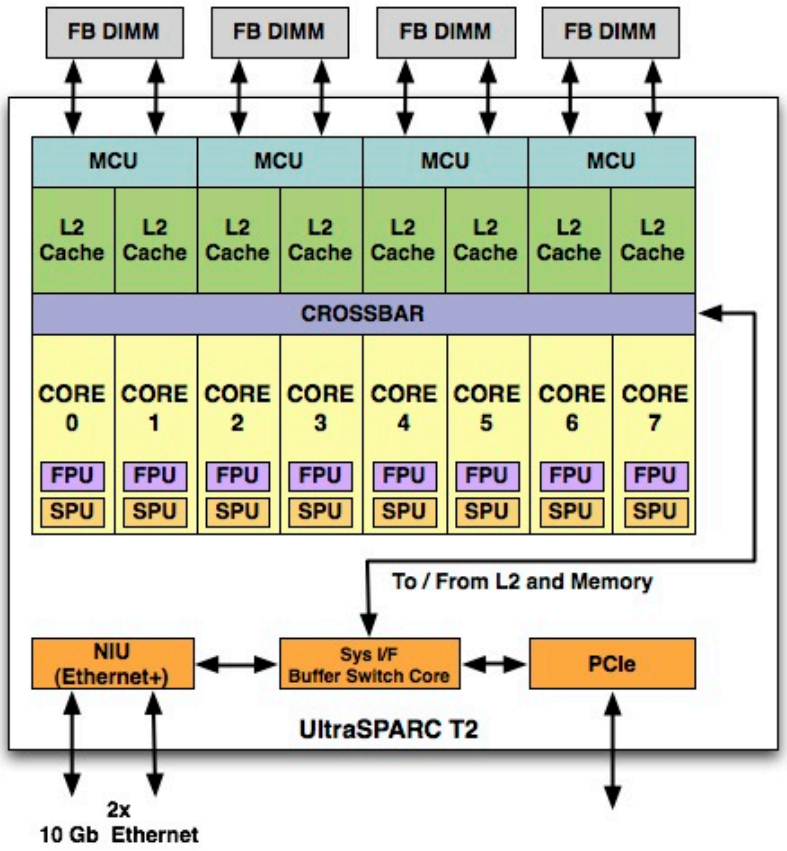

Figura 10. Diagrama de componentes del processador Multi-núcleo UltraSPARC T2 (8 núcleos)

El procesador UltraSPARC T2 dispone de 4 a 8 núcleos de procesamiento capaces de manipular hasta 8 hilos de ejecución cada uno. Cada núcleo dispone a su vez de dos memorias cache de primer nivel (L1) de 16KB para instrucciones y 8KB para datos, una unidad para operaciones en punto flotante y una unidad de cifrado.

La jerarquía de memoria del T2 es de dos niveles. Todos los núcleos disponen de 4MB de memoria cache de segundo nivel organizada en 8 bancos asociativos de 16 vías compartida a través de una red de interconexión de tipo "crossbar" donde todos los núcleos puede acceder a todos los bancos simultáneamente (todas las parejas no conflictivas pueden acceder al mismo tiempo).

Además el UltraSPARC T2 incorpora 4 controladores de memoria de doble canal denominados FB-DIMM integrados para bajar la latencia hacia la memoria principal y dos interficies de red accesibles a través del crossbar, dos puertos 10GB Ethernet y 8 puertos de entrada/salida del tipo PCIe integrados también.

En el diseño del UltraSPARC T2 se ha buscado la máxima eficiencia energética. Una de las grandes características que tiene este procesador es su reducido consumo, tan sólo 95w. Además tiene la posibilidad de encender/apagar los hilos por software o reducir la frecuencia de los núcleos para minimizar el consumo. [4]

2.4.3 Procesador AMD Dual-Core

Siguiendo esta línea AMD ha diseñado un conjunto de procesadores multi-núcleo y multi-hilo. De este conjunto estudiaremos AMD Athlon 64 X2 Dual-core. A continuación nos encontramos en la Figura 11, el esquema principal de este procesador multi-núcleo y sus unidades funcionales.

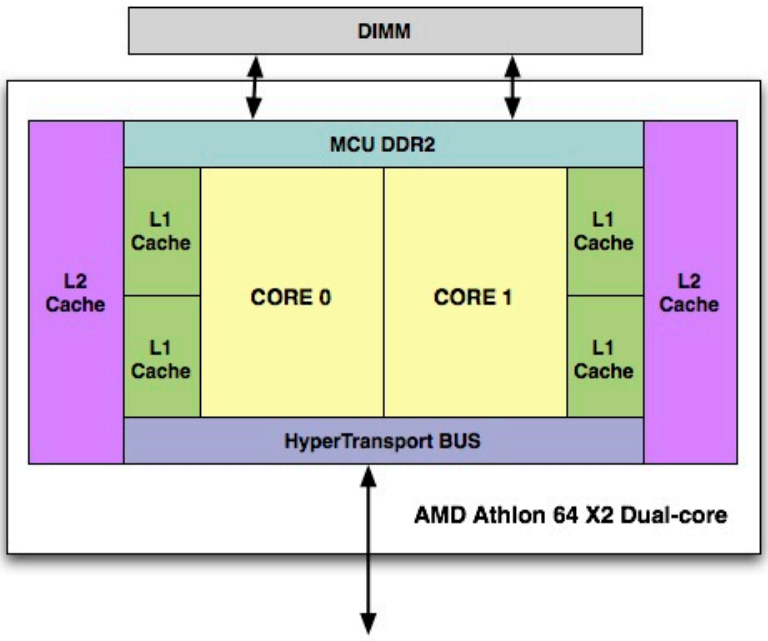

Figura 11. Diagrama de componentes del processador Multi-núcleo AMD Athlon 64 X2 Dual-core

El procesador AMD Athlon 64 dispone de 2 núcleos con tecnología *AMD64* que proporciona soporte a plena velocidad para x86.

La jerarquía de memoria del AMD Athlon 64 es de dos niveles. Cada núcleo dispone a su vez de dos memorias cache de primer nivel (L1) de 64KB para instrucciones y 64KB para datos. Cada núcleo disponen de 1MB de memoria cache de segundo nivel (L2).

Además el AMD Athlon 64 incorpora 1 controlador de memoria de doble canal denominados DIMM integrados para bajar la latencia hacia la memoria principal y un bus con tecnología *HyperTransport* de 8GB/s, para lograr unas comunicaciones de entrada/salida de alta velocidad.

En el diseño del AMD Athlon 64 se ha buscado la máxima eficiencia energética. Una de las grandes características que tiene este procesador es su reducido consumo, tan sólo 89w. Además posee la tecnología *Cool'N'Quiet* con lo que es capaz de reducir la velocidad y el voltaje de los núcleos para minimizar el consumo reduciendo el consumo a tan sólo 22w.

2.4.4 Procesador Intel Core2 Quad

Siguiendo esta línea Intel ha diseñado un conjunto de procesadores multi-núcleo y multi-hilo. De este conjunto estudiaremos Intel Core2 Quad. A continuación nos encontramos en la Figura 12, el esquema principal de este procesador multi-núcleo y sus unidades funcionales.

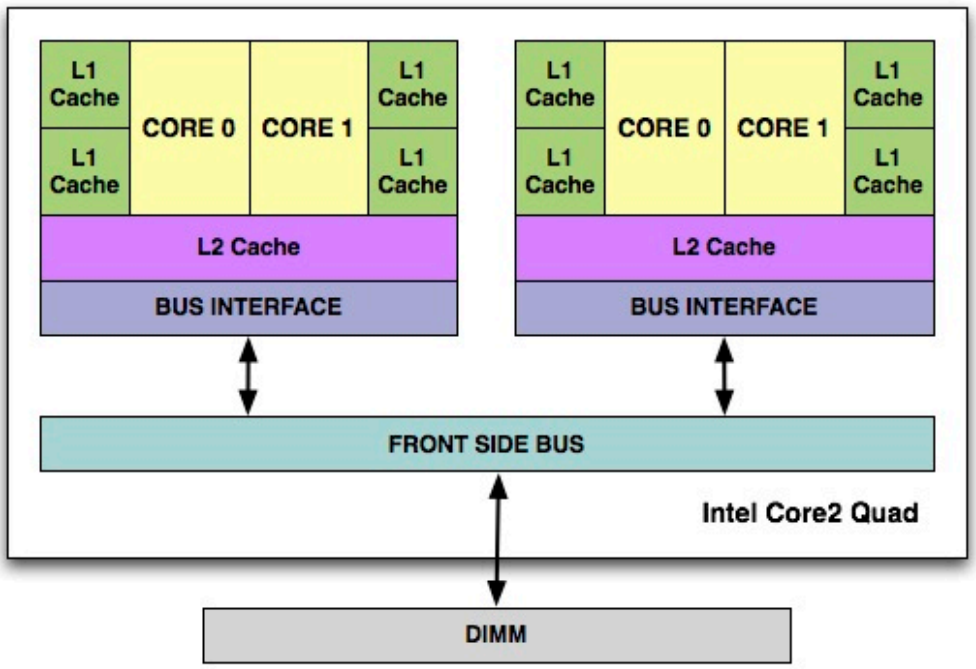

Figura 11. Diagrama de componentes del processador Multi-núcleo Intel Core2 Quad

El procesador Intel Core2 Quad está formado por dos procesadores Intel Core2 Duo cada uno con de 2 núcleos de procesamiento empaquetados en un mismo chip. Los dos procesadores de doble núcleo están interconectados mediante un bus frontal y la memoria principal.

La jerarquía de memoria del Intel Core2 Quad es de dos niveles. Cada núcleo dispone a su vez de dos memorias cache de primer nivel (L1) de 32KB para instrucciones y 32KB para datos. Cada uno de los procesadores de doble núcleo que hay dentro del chip dispone de una memoria cache de segundo nivel (L2) de 3MB.

Además el Intel Core 2 Quad incorpora un bus frontal (*Front Side Bus*), que interconecta los dos procesadores de doble núcleo entre ellos y con la memoria principal, capaz de ofrecer un ancho de banda de 12.4GB/s, para lograr unas comunicaciones de entrada/salida de alta velocidad. [5-6]

2.5 Factores que impiden un alto rendimiento en sistemas multi-Core/multi-Thread

Como hemos visto hasta el momento podría parecer que el aumento paulatino del número de núcleos de un procesador parece la solución a la necesidad de aumentar la capacidad de procesamiento en los procesadores, pero no es así. El rendimiento de las aplicaciones en entornos multi-núcleo/multi-hilo no escala linealmente, y al aumentar el número de hilos con los que se ejecuta la aplicación no se reduce linealmente el tiempo de cómputo de manera transparente al programador. A continuación enunciaremos los principales factores que impiden esta escalabilidad lineal en los entornos de programación multi-núcleo/multi-hilo.

2.5.1 Overheads por creación/eliminación de hilos

Uno de los factores que impiden la escalabilidad lineal en sistemas multi-núcleo/multi-hilo, es el proceso de creación y eliminación de hilos que trabajan en dicho sistema paralelo, que supone un coste de tiempo extra ("*overhead"*) sobre el total del tiempo de ejecución.

Este "*overhead"* ha de ser mucho menor al tiempo total de ejecución para que sea práctico paralelizar una aplicación. La importancia de este "*overhead"* dependerá de la relación entre el tiempo de ejecución de la zona paralela por un hilo entre el número total de hilos y el tiempo empleado en la creación, eliminación de hilos, el reparto del cómputo y la obtención del resultado final. A continuación en la figura 13 mostramos de manera gráfica este "*overhead"*.

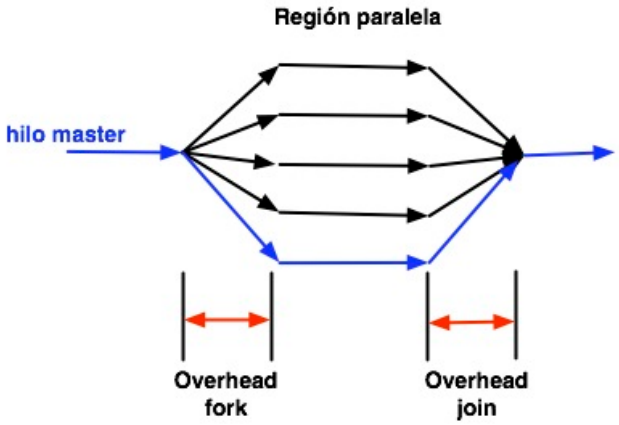

Figura 13. Paradigma fork-join en la computación paralela

2.5.2 Balanceo incorrecto del volumen de cómputo

Otro de los factores que impiden la escalabilidad lineal en sistemas multi-núcleo/multi-hilo es la incorrecta distribución del volumen de cómputo por hilo, que implica que algunos hilos finalicen sus tareas antes que el resto y por tanto tendrán que esperar a que el resto acabe para proseguir la ejecución.

Esta espera supone un coste en ciclos de reloj del procesador desaprovechados y por lo tanto un *overhead*. En el diseño de aplicaciones paralelas es muy importante una óptima asignación del trabajo a realizar a cada uno de los hilos. A la hora de programar una aplicación paralela uno de los puntos a tener en cuenta es como de bien se reparte el cómputo entre los diferentes hilos (*computation load balance*).

Este *overhead* se puede reducir aunque no eliminar totalmente, asignando el trabajo dinámicamente entre los diferentes hilos que participan en la ejecución. Por contrapartida se genera otro pequeño *overhead* asociado al cómputo necesario para gestionar la asignación dinámica.

2.5.3 Las comunicaciones entre hilos de ejecución

Otro de los factores que impiden la escalabilidad lineal en sistemas multi-núcleo/multi-hilo es la comunicación entre los núcleos. En una región paralela los hilos de una ejecución multi-hilo trabajan de manera independiente y con datos independientes, pero por las características de las aplicaciones, en algún momento necesitarán intercambiar estos datos entre ellos. Este intercambio de datos se realiza de manera transparente al hilo ya que éste únicamente accederá a unas posiciones de memoria que previamente otro hilo habrá modificado. Esto aunque es transparente para el hilo no esta libre de coste en tiempo. Los datos que hayan sido modificados en la cache de un hilo tendrán que ser copiados a la cache del hilo que los necesita en ese momento.

Hay que tener en cuenta que el coste de comunicar datos modificados por hilos que se ejecutan dentro de un mismo procesador es muy inferior al coste de comunicar datos entre hilos que se ejecutan en núcleos de diferentes procesadores.

Estos costes suponen un *overhead* a considerar a la hora de diseñar una aplicación multi-hilo. Habrá que prestar especial atención a la localidad temporal y espacial de la aplicación y a la descomposición por dominio, intentando primero minimizar las comunicaciones entre hilos que se ejecutan en distintos procesadores y segundo minimizar las comunicaciones entre hilos que se ejecutan en el mismo procesador.

2.6 OpenMP

A continuación en este apartado se describirá OpenMP y se darán algunos ejemplos de uso, únicamente para que el lector tenga una visión general. Este apartado es una breve introducción, y en ningún momento pretende ser un manual.

OpenMP es un modelo de programación paralela para procesadores de memoria compartida y distribuida. OpenMP es el estándar de "facto" para aplicaciones paralelas donde los diferentes hilos de ejecución colaboran para realizar una misma tarea en paralelo. En el modelo de memoria compartida que mostramos en la figura 14 los diferentes hilos de ejecución comparten un único espacio de direcciones, se comunican y se sincronizan entre ellos mediante variables compartidas.

En el modelo de variables compartidas la visión estándar del paralelismo es el fork/join. Cuando un programa empieza la ejecución, un único hilo, llamado "master" esta activo. El master ejecuta toda la parte secuencial del algoritmo. En el momento que el programador especifica que pueden hacer operaciones en paralelo, el master hace un "fork", (crea o despierta) un conjunto hilos de ejecución adicionales. El hilo master y el resto de hilos creados trabajan concurrentemente durante la sección paralela definida y una vez acabada esta sección, los hilos creados mueren o se suspenden, i el flujo de control vuelve hacia un único hilo, el master. A esta última acción se le llama "join".

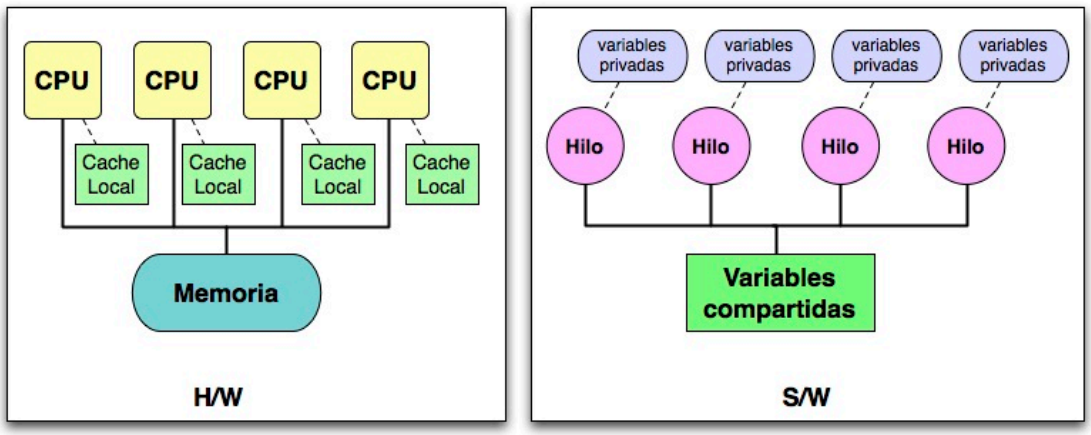

Figura 14. Modelo de programación paralela de memoria compartida

OpenMP no es un nuevo lenguaje de programación, consiste en un conjunto de directivas para el compilador, un conjunto de funciones de soporte y variables de entorno para el tratamiento de hilos de ejecución (comunicación y sincronización de hilos). Estas directivas se añaden al código fuente y describen el paralelismo de éste para que las interprete el compilador. Actualmente estas directivas están definidas para los lenguajes Fortran, C y C++.

La sintaxis básica de OpenMP es: **# pragma omp <directiva> [clausula, …]** Podemos ver en la Figura 15, un ejemplo de código en C donde se marcan a estas directivas de OpenMP en azul. Este código define 3 vectores de 10.000 posiciones. Inicializa el primer vector con la suma de los valores del segundo y el tercer vector. Podemos ver justamente antes de la instrucción *for* del bucle, la directiva OpenMP *"#pragma omp for*". Con esta directiva explicitamos la paralelización del bucle de manera que se dividirán las iteraciones del bucle entre los diferentes hilos de ejecución de forma equitativa.

```
int  a[10000],  b[10000], c[10000] ;
#pragma omp parallel
#pragma omp for
for (i = 0; i < 10000; i++) {
        a[i] = b[i] + c[i];
}
```

Figura 15. Codigo C con las directivas OpenMP para la paralelitzación de un bucle for

A continuación citaremos las directivas, cláusulas y funciones de OpenMP más usadas en el desarrollo de aplicaciones paralelas y adjuntaremos una breve explicación.

2.6.1 Directivas

A continuación en este apartado citamos las directivas OpenMP más usadas en el desarrollo de aplicaciones paralelas.

- **parallel**: Indica la parte del código que se podrá ejecutar por varios hilos
- **for**: Indica que a cada hilo se le asigna una parte del for a ejecutar (paralelismo de datos).
- **sections**: Indica que cada sección será ejecutada por un hilo (paralelismo funcional).
- **single**: Indica que esta sección será ejecutada por un único hilo.
- **master**: Indica que esta sección será ejecutada por el master-hilo.

- **parallel for**: Combinación de parallel (fork) + for (repartición del for entre los hilos).

- **parallel sections**: Combinación de parallel (fork) + sections (asignación de cada sección a un hilo).
- **critical**: Indica que la sección será de exclusión mutua.

- **barrier**: Indica la necesidad de una sincronización de los hilos en este punto.

2.6.2 Cláusulas

En este apartado citamos las cláusulas OpenMP más usadas en el desarrollo de aplicaciones paralelas.

- s**chedule** (static | dynamic | guided | runtime [, chunk]): Determina de que forma se realizará la asignación del trabajo a los hilos.

- **private** (variable [,variable, …]): Indica que las variables que aparecen en la cláusula serán privadas, cada hilo tendrá una copia independiente.

2.6.3 Funciones

En este apartado citamos las cláusulas OpenMP más usadas en el desarrollo de aplicaciones paralelas.

- **omp_set_num_thread:** Fija el número de hilos que se crearán en los forks de las regiones paralelas de código.
- **omp_get_num_threads:** Devuelve el número de hilos activos en la región paralela actual.

- **omp_get_thread_num:** Devuelve el identificador del hilo que lo ejecuta.

- **omp_get_num_procs:** Devuelve el número de cores del multi-core en el que se esta ejecutando el hilo.

[7]

2.7 PAPI (Performance Application Programming Interface)

De la misma forma que en el apartado anterior, se realizará una breve descripción, sin que ésta pretenda ser un manual. El objetivo es únicamente que el lector tenga una visión general de PAPI.

PAPI es una API para acceder a los contadores hardware de rendimiento, disponibles en la mayoría de los procesadores actuales. Estos contadores son un conjunto de registros que cuentan la ocurrencia de determinados eventos en el procesador y nos ayudan a conocer de manera más fidedigna el comportamiento de nuestra aplicación al ejecutarse en el sistema.

El kernel del sistema operativo puede necesitar algunas modificaciones para permitir acceder a los contadores hardware mediante el uso de PAPI. Dependiendo del procesador y del sistema operativo puede llegar a ser necesario aplicar un parche (PerfCtr o perfmon2) y recompilar el kernel.

Una aplicación compilada en un sistema cuyo kernel tiene PAPI instalado, puede ser ejecutada en cualquier sistema con el parche (PerfCtr o perfmon2), siempre que ambos sistemas tengan una arquitectura y sistema operativo compatibles.

Al incorporar PAPI a una sección de código, es posible pasar (como parámetro) a la aplicación, los nombres de los contadores hardware de los que se quiere obtener información. La extracción de esta información de los contadores hardware nos ayudará posteriormente al análisis del rendimiento de nuestras aplicaciones en los diferentes sistemas paralelos ayudándonos a localizar la causa real de los diferentes overheads que aparezcan. En la figura 16 mostramos algunos de los contadores hardware a los que se puede acceder mediante la utilización de PAPI.

[8]

Nombre de contador	Descripción
PAPI_L1_DCM	Level 1 data cache
PAPI_L1_ICM	Level 1 instruction cache
PAPI_L2_DCM	Level 2 data cache
PAPI_L2_ICM	Level 2 instruction cache
PAPI_L3_DCM	Level 3 data cache
PAPI_L3_LDM	Level 3 load misses
PAPI_L1_LDM	Level 1 load misses
PAPI_L2_LDM	Level 2 load misses
PAPI_TOT_INS	Instructions completed
PAPI_INT_INS	Integer instructions
PAPI_FP_INS	Floating point instructions
PAPI_BR_INS	Branch instructions
PAPI_TOT_CYC	Total cycles
PAPI_FML_INS	Floating point multiply instructions
PAPI_FAD_INS	Floating point add instructions

3 Descripción del problema: Eliminación Gaussiana

La resolución de sistemas lineales es uno de los problemas más antiguos de la matemática. Muchos de los problemas actuales de la ingeniería y la ciencia pueden modelarse como un complejo sistema de ecuaciones lineales.

La resolución de sistemas de ecuaciones lineales posee una infinidad de aplicaciones: análisis estructural, estimación, predicción y más generalmente en la aproximación de problemas no lineales de análisis numérico. Las aplicaciones de la programación Lineal son muy extensas entre ellas destacan los problemas de flujos de redes y problemas de flujos de mercancías, problemas relacionados con la economía (gestión de inventarios, la cartera, planificación de campañas de publicidad, reducir gastos y maximizar ingresos…). El análisis numérico se encarga de modelar a través de reglas matemáticas simples procesos matemáticos más complejos utilizados en la vida real por físicos e ingenieros para todo tipo de resolución y aproximación de problemas reales.

Es por este motivo que se dedican muchos esfuerzos en desarrollar algoritmos eficientes que resuelvan estos problemas en un lapso de tiempo menor con una mayor precisión para obtener la solución más adecuada.

La Eliminación Gaussiana es un eficiente algoritmo para la resolución de sistemas de ecuaciones lineales Ax=b. Este algoritmo esta dividido en dos fases: La Eliminación hacia delante y la sustitución inversa. En el apartado siguiente (1.1) se definirán una serie de conceptos que se necesitarán más adelante. Y en el apartado 1.2 se realizará una descripción general del algoritmo y en el 1.3 se describirán las diferentes estrategias a seguir para generar el código serie del algoritmo.

3.1 Resolución de sistemas lineales

Una ecuación es la expresión matemática de una condición de igualdad. Un sistema de "k" ecuaciones es la expresión matemática de un conjunto de "k" condiciones de igualdad. Una ecuación lineal de "n" variables x_0, x_1, ..., x_{n-1} es una ecuación que puede ser representada de la forma $a_0x_0 + a_1x_1 + \cdots + a_{n-1}\, x_{n-1} = b$, donde a_0, a_1, … ,a_{n-1} y b son constantes.

Por tanto llamamos sistema de ecuaciones lineal o sistema lineal a un conjunto de ecuaciones lineales finito. Y llamamos solución (S_0, S_1, …, S_{n-1}) de un sistema de ecuaciones al conjunto de valores que al realizar las sustituciones $x_0 = S_0$, $x_1 = S_1$, … , $x_{n-1} = S_{n-1}$ satisface todas las ecuaciones del sistema lineal.

Un sistema lineal de "m" ecuaciones y "n" variables puede representarse de la siguiente forma:

$$
\begin{aligned}
&a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} = b_0 \\
&a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} = b_1 \\
&a_{2,0}x_0 + a_{2,1}x_1 + \cdots + a_{2,n-1}x_{n-1} = b_2 \\
&\vdots \\
&a_{m-1,0}x_0 + a_{m-1,1}x_1 + \cdots + a_{m-1,n-1}x_{n-1} = b_m
\end{aligned}
$$

Figura 17. Sistema de ecuaciones

y se suele representar de la forma Ax = b donde A es una matriz $n \times m$, x es el vector columna de incógnitas de longitud n y b es el vector de columna términos independientes de longitud m.

$$
\begin{pmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{pmatrix} \bullet \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ \vdots \\ b_{m-1} \end{pmatrix}
$$

Figura 18. Sistema lineal de ecuaciones

El sistema lineal asociado también puede representarse a través de su matriz ampliada, que no es más que la matriz asociada al sistema de coeficientes añadiéndole la columna de términos independientes. En la figura 19 se muestra un ejemplo de una matriz ampliada en la que aparece la columna final con los términos independientes de cada ecuación del sistema.

$$
\left[\begin{array}{cccc|c}
4 & 6 & 2 & -2 & 8 \\
2 & 0 & 5 & -2 & 4 \\
-4 & -3 & -5 & 4 & 1 \\
8 & 18 & -2 & 3 & 40
\end{array}\right]
$$

Figura 19. Ejemplo de una matriz ampliada.

Una matriz A, es una matriz banda de anchura w si $: i - j > w \rightarrow a_{ij} = 0$ y $j - i > w \rightarrow a_{ij} = 0$. Es decir si todos los elementos no nulos de A se encuentran en la diagonal cuyo rango lo marca la constante w.

$$\begin{bmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{bmatrix}$$

Figura 20. Representación de una matriz banda con $W = 1$

La matriz será triangular superior si : $i > j \rightarrow a_{ij} = 0$.

$$\begin{bmatrix} a_{00} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 & 0 \\ a_{20} & a_{21} & a_{22} & 0 & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} & a \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Figura 21. Representación de una matriz triangular superior

La matriz será triangular inferior si: $i < j \rightarrow a_{ij} = 0$.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ 0 & a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & 0 & a_{44} \end{bmatrix}$$

Figura 22. Representación de una matriz triangular inferior

La matriz será estrictamente diagonal dominante si $\left|a_{i,j}\right| > \left|\sum_{i \neq j}^{n} \left|a_{i,j}\right|\right|$; $0 \leq i \leq n$.

$$\begin{bmatrix} 5 & 2 & 1 & -1 & 0 \\ 1 & 9 & 0 & -2 & 3 \\ 2 & -1 & -6 & 0 & 2 \\ 1 & 0 & 2 & 5 & -1 \\ 3 & 2 & -1 & 1 & -9 \end{bmatrix} \quad \begin{array}{r} |5| > (2 + 1 + |-1| + 0) \\ |9| > (1 + 0 + |-2| + 3) \\ |-6| > (|-1| + 2 + 0 + 2) \\ |5| > (0 + 2 + 1 + |-1|) \\ |-9| > (2 + |-1| + 1 + 3) \end{array}$$

Figura 23. Representación de una matriz diagonal dominante

3.2 Descripción general del algoritmo de Eliminación Gaussiana

La eliminación Gaussiana es un eficiente algoritmo para la resolución de sistemas lineales Ax=b. Este algoritmo consta de dos fases. La primera fase llamada "Forward elimination" (o Eliminación hacia delante**)** reduce el sistema lineal mediante transformaciones elementales transformándolo en un sistema triangular superior o genera una ecuación sin solución indicando que el sistema lineal no tiene solución. En la segunda fase se emplea el algoritmo llamado "back substitution" (o sustitución inversa), para encontrar la solución del sistema Ax=b, cuando la matriz A es triangular superior.

La Eliminación Gaussiana es *numéricamente estable*[1] para matrices diagonal dominantes y para matrices *definidas positivas*[2]. Para las matrices en general, la Eliminación Gaussiana suele ser estable si se usa el pivotaje total o parcial que describiremos más adelante en el apartado 3.2, aunque existen ejemplos para estos métodos que son inestables.

3.2.1 Primera Fase: Eliminación (Forward Elimination)

Durante la primera fase del algoritmo las transformaciones elementales nos ayudan a simplificar el sistema sin variar su solución y son las siguientes:

- Multiplicar cada término de una ecuación por una constante no nula.
- Intercambiar dos ecuaciones.
- Añadir un múltiple de una ecuación a otra ecuación del sistema.

A continuación mostramos un ejemplo gráfico de la aplicación del algoritmo de resolución de la Eliminación Gaussiana sin pivotaje sobre un sistema de ecuaciones lineales, representados mediante su matriz ampliada.

[1] *En el análisis numérico, la estabilidad numérica es una propiedad de los algoritmos numéricos. Describe cómo los errores en los datos de entrada se propagan a través del algoritmo. En un método estable, los errores debidos a las aproximaciones se atenúan a medida que la computación procede. En un método inestable, cualquier error en el procesamiento se magnifica conforme el cálculo procede. Métodos inestables generan rápidamente basura y son inútiles para el procesamiento numérico.*

[2] *Una matriz definida positiva es una matriz cuadrada de elementos complejos que tiene la característica de ser igual a su propia traspuesta conjugada.*

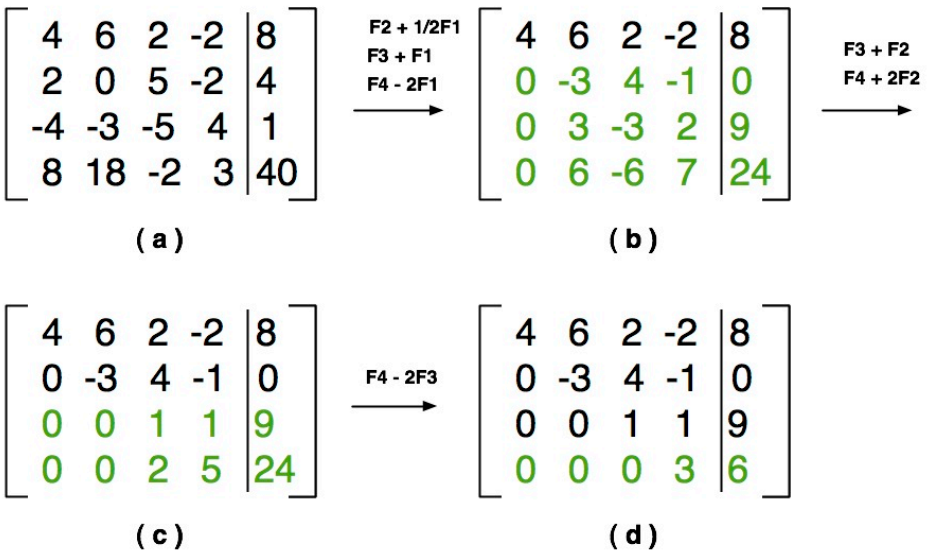

Figura 24. Ejemplo del algoritmo de Eliminación Gaussiana.

La figura 24 muestra el proceso de la primera fase del algoritmo la "eliminación hacia delante". La matriz (a) representa el sistema original de ecuaciones lineales. El primer paso transforma la matriz (a) en la matriz (b) usando el primer elemento de la primera fila de la matriz 4 y con este valor como referencia transforma la matriz para convertir en ceros todos los elementos que están por debajo de éste. A la fila dos le resta un medio de la fila 1($f_2 - \frac{1}{2} f_1$). A la fila 3 le suma la fila ( ). Y a la fila cuatro le resta dos veces la primera fila ( ).

El segundo paso transforma la matriz (b) en la matriz (c) usando el siguiente elemento de la diagonal -3 para aplicar las trasformaciones oportunas para triangular la matriz hasta llegar a la matriz (d). Una vez ha aplicado todas las transformaciones necesarias para convertir el sistema de ecuaciones lineal en una matriz triangular superior es cuando podemos aplicar el algoritmo sustitución inversa para resolver el sistema.

3.2.2 Segunda Fase: Sustitución inversa (Back substitution)

El algoritmo de sustitución inversa es un algoritmo que resuelve sistemas lineales Ax=b, cuando la matriz A es triangular superior. En la figura 25 mostramos el pseudocódigo de la sustitución inversa que aplicaremos seguidamente en el siguiente ejemplo.

```
A[0..n-1,0..n-1] //matriz de coeficientes
b[0..n-1] //vector de constantes
x[0..n-1] //vector de soluciones

for i = n-1 down to 1 do
    x[i] = b[i]/A[i][i]
    for j = 0 t i-1 do
        b[j] =  b[j] - x[i]*A[i][j]
        A[i][j] = 0 //línea opcional
    end for
end for
```

Figura 25. Pseudo código del algoritmo de sustitución inversa.

Supongamos que tenemos el siguiente sistema de ecuaciones lineal triangulado previamente en la primera fase del algoritmo de la Eliminación Gaussiana, eliminación hacia delante siguiente:

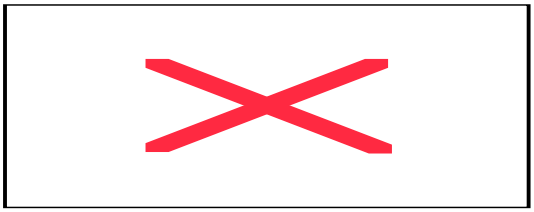

Figura 26. Sistema de ecuaciones

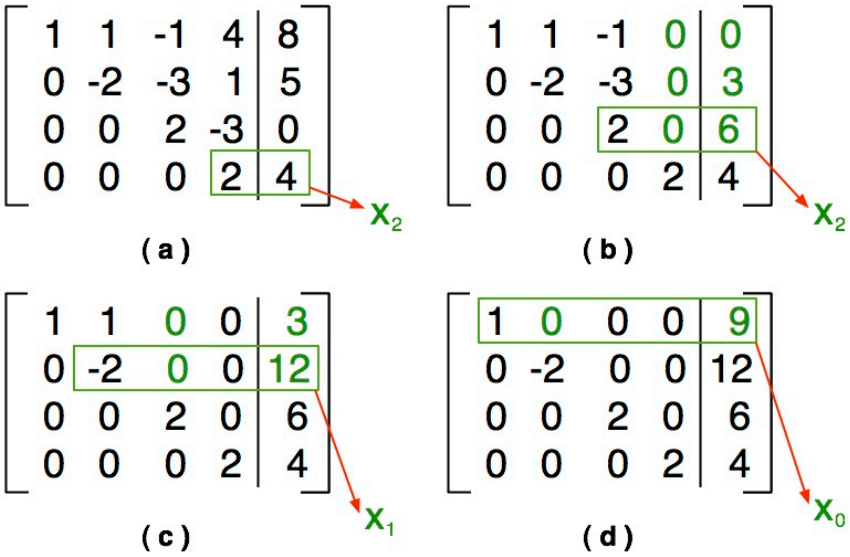

Figura 27. Ejemplo de la fase de sustitución inversa del algoritmo de Eliminación Gaussiana.

La figura 27 muestra el proceso de la segunda fase del algoritmo la "sustitución inversa" del sistema anterior. La matriz (a) representa el sistema original de ecuaciones lineales. La primera iteración transforma la matriz (a) en la matriz (b) resolviendo la última ecuación de forma directa ya que sólo tenemos una incógnita.
De la última ecuación obtenemos la solución de la variable x_3. Con el valor de ésta variable  se simplifican el resto de ecuaciones sustituyendo el valor de la variable y sumándoselo al termino independiente cada ecuación.

La segunda iteración transforma la matriz (b) en la matriz (c) resolviendo esta vez, la tercera ecuación que tras la primera iteración se ha sustituido el valor de la variable x_3 y ahora sólo posee una incógnita, de forma directa. De esta tercera ecuación obtenemos el valor de la variable x_2. Con el valor de ésta última variable obtenida se vuelven a actualizar el resto de ecuaciones que se encuentran por encima de la tercera ecuación, simplificándolas con el valor [ ] del mismo modo que en la iteración anterior.

La tercera y última iteración transforma la matriz (c) en la matriz (d) resolviendo con en las iteraciones anteriores la segunda ecuación, obteniendo así de esta forma el valor de la variable x_1. Con el valor de [ ] se sustituye la primera ecuación dejándola con una sola incógnita. De este último paso obtenemos directamente el valor de la variable [ ], de manera que ya hemos resuelto el sistema.

3.3 Pivotaje: estrategia para aumentar la precisión y estabilidad

Como ya comentábamos con anterioridad, la Eliminación Gaussiana es numéricamente estable para matrices diagonal dominantes y definidas positivas, ya que los pivotes se encuentran en la diagonal. Cuando estos valores de la diagonal tienen un valor pequeño se pueden producir grandes errores de redondeo y divisiones por cero provocando una inestabilidad del algoritmo e imprecisando el resultado final. Esta inestabilidad se resuelve empleando la estrategia del pivotaje total o parcial.

En la figura 28 podemos observar una representación gráfica simplificada de la estrategia del pivotaje. El pivotaje esta formado por tres fases bien diferenciadas. La búsqueda del pivote, dónde se selecciona el valor absoluto más grande. El intercambio de filas i/o columnas para colocar el nuevo pivote en la diagonal de la matriz y la actualización de filas, dónde se aplica la Eliminación hacía delante escalonando la matriz.

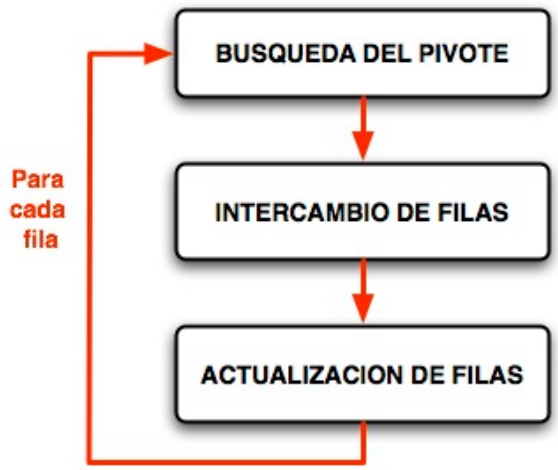

Figura 28. Representación simplificada del algoritmo de Eliminación Gaussiana con pivotaje por filas.

El pivotaje total selecciona la posición de la fila o columna con el valor absoluto más grande e intercambia las filas y columnas situando el elemento escogido en la diagonal de la matriz, la posición del pivote.

El pivotaje parcial se puede realizar por filas o por columnas. Con el pivotaje parcial, el algoritmo selecciona la posición con el valor absoluto más grande de la columna intercambiando las filas de la matriz para colocar este valor en la diagonal de la matriz, si nos referimos al pivotaje parcial por filas o la posición con el valor absoluto más grande de la fila e intercambiado las columnas, si nos referimos al pivotaje parcial por columnas.

El pivotaje parcial se diferencia del pivotaje total en que el pivotaje total intercambia las filas y columnas para colocar los valores absolutos más grandes en la diagonal de la matriz mientras que el parcial tan sólo selecciona los valores absolutos e intercambia valores para filas o columnas según de que pivotaje parcial estemos hablando.

En la práctica el pivotaje total puede consumir mucho tiempo computacionalmente hablando ya que para cada iteración se debe buscar el máximo entre todas las posiciones de las filas y columnas de la matriz sin escalonar. El pivotaje parcial es generalmente suficiente para reducir de manera adecuada el error del algoritmo.

Seguidamente mostramos dos ejemplos gráficos de la aplicación sobre un sistema de ecuaciones lineales, representados mediante la matriz ampliada de éste, del algoritmo de resolución de la Eliminación Gaussiana con pivotaje parcial. En el primer ejemplo se muestra la resolución mediante el pivotaje por filas y en el segundo se muestra la resolución mediante el pivotaje por columnas.

En la figura 29 se muestra un ejemplo del algoritmo de Eliminación Gaussiana con pivotaje parcial por filas.

La matriz (a) representa el sistema original de ecuaciones lineal. El primer paso de la primera fase del algoritmo es la búsqueda del pivote, el elemento con el valor máximo de la primera columna, en este caso el $|-4|$.

Este valor pasará a ser el pivote. El segundo paso de la primera fase del algoritmo es el intercambio de filas, en este caso, de la primera fila por la fila del pivote para colocar de esta forma el pivote en la diagonal de la matriz para mejorar la estabilidad del algoritmo, este paso lo podemos observar en la matriz (b). El tercer y último paso de la primera fase del algoritmo es la actualización de las filas posteriores a la fila pivote, con éste. Como se observa en la matriz (c) se han eliminando todos los valores de la primera columna con el pivote.

En la siguiente iteración se selecciona el máximo valor de la siguiente columna entre las filas restantes como se observa en la matriz (d). Esta vez en nuevo pivote se encuentra en la cuarta fila de la matriz, con el valor $|2|$. De nuevo se intercambian la segunda fila con la cuarta para colocar el nuevo pivote en la diagonal de la matriz como se observa en la matriz (c) y se procede a actualizar las filas restantes obteniendo la matriz (f). En este caso sólo se actualiza la tercera fila, ya que el valor de la cuarta fila ya es un 0.

Una vez actualizadas las filas se procede con la tercera iteración del algoritmo. En esta

iteración el pivote seleccionado, $|-15/4|$, se encuentra en la tercera fila por lo que no es necesario intercambiar filas, ya que el pivote se encuentra en la diagonal. Con el nuevo pivote seleccionado se procede a actualizar la última fila de la matriz, dejando por tanto la matriz finalmente escalonada como se observa en la matriz (g).

La matriz resultado de la primera fase del algoritmo la eliminación hacia delante, matriz (h), es una matriz triangular superior, en la que la diagonal está formada por todos los pivotes calculados en las diferentes iteraciones**.** Una vez aplicadas todas las transformaciones elementales de la primera fase del algoritmo, en la segunda fase, se le aplica el algoritmo de sustitución inversa a la matriz diagonal superior (h) obtenida para resolver el sistema.

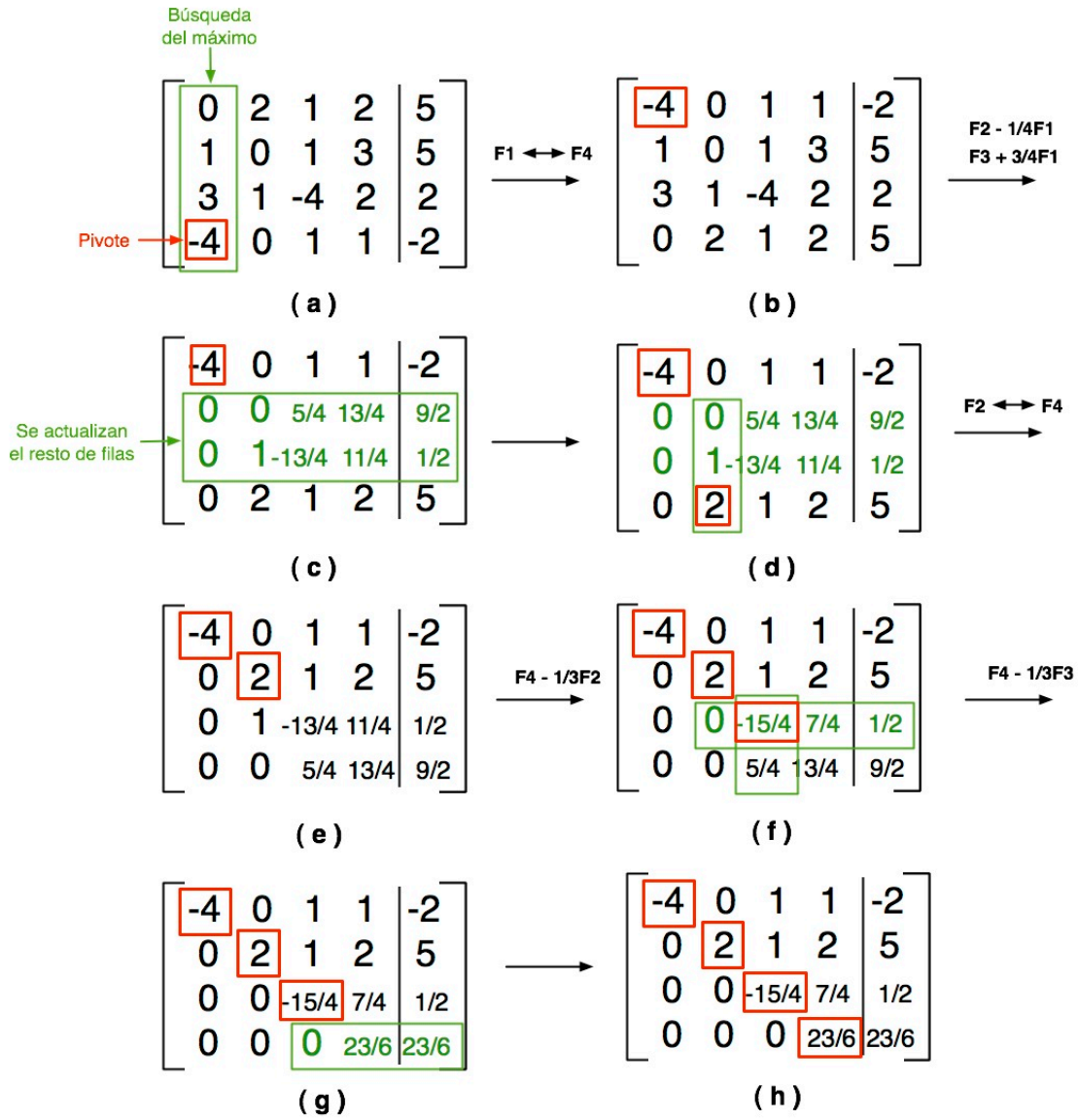

Figura 29. Ejemplo del algoritmo de Eliminación Gaussiana con pivotaje por filas

A continuación se muestra paso a paso como se aplica esta segunda fase del algoritmo a la matriz obtenida en la primera fase (h) en la figura 30.

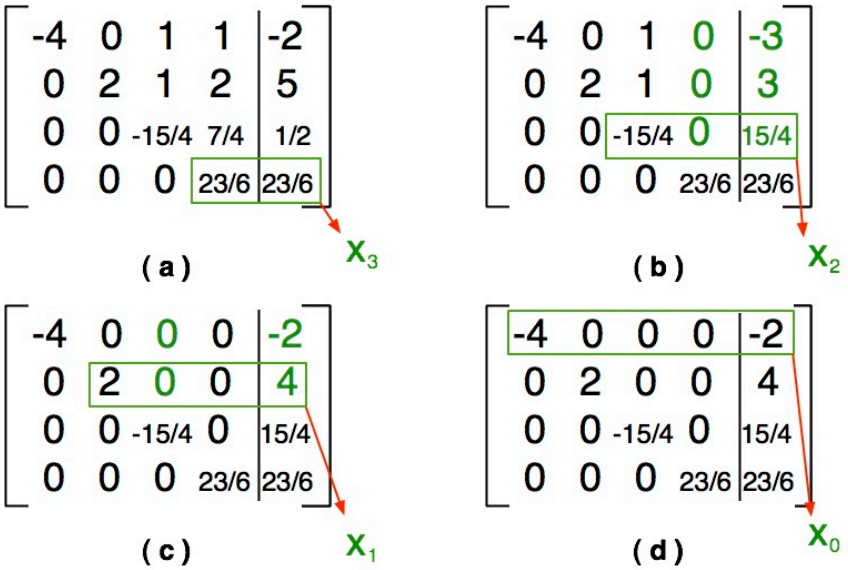

Figura 30. Ejemplo de la fase de sustitución inversa del algoritmo de Eliminación Gaussiana.

La figura 30 muestra el proceso de la segunda fase del algoritmo la "sustitución inversa" del sistema anterior.

La matriz (a) representa la matriz ampliada del sistema original de ecuaciones lineales reducida mediante la primera fase del algoritmo de la Eliminación Gaussiana con pivotaje parcial por filas. En la primera iteración de la sustitución inversa, transforma la matriz (a) en la matriz (b) resolviendo la última ecuación de forma directa ya que sólo tenemos una incógnita. De la última ecuación obtenemos la solución de la variable x_3. Con el valor de ésta variable $x_3 = 1$ se simplifican el resto de ecuaciones sustituyendo el valor de la variable x_3 y sumándoselo al termino independiente cada ecuación.

La segunda iteración transforma la matriz (b) en la matriz (c). Tras la primera iteración, la tercera ecuación sólo posee una incógnita al el valor de la variable x_3 se resuelve de forma directa. De esta tercera ecuación obtenemos el valor de la variable x_2. Con el valor de ésta última variable obtenida se vuelven a actualizar el resto de ecuaciones que se encuentran por encima de la tercera ecuación, simplificándolas con el valor [ ] del mismo modo que en la iteración anterior.

La tercera iteración transforma la matriz (en la matriz (d) resolviendo con en las iteraciones anteriores la segunda ecuación, obteniendo así de esta forma el valor de la variable x_1. Con el valor de $x_1 = 2$ se sustituye la primera ecuación dejándola con una sola incógnita, por lo que no es necesario una iteración más. De este último paso obtenemos directamente el valor de la variable [ ], de manera que ya se ha obtenido la solución al sistema de ecuaciones lineal.

La figura 31 muestra un ejemplo del algoritmo de Eliminación Gaussiana con pivotaje parcial por columnas donde la matriz (a) representa el sistema original de ecuaciones lineal.

En la primera iteración se localiza el valor máximo de la primera fila, $|6|$. Este pasará a ser el nuevo pivote. En la matriz (b) se observa el intercambio de la primera columna por la columna del pivote para colocar de esta forma el pivote en la diagonal de la matriz  . Una vez está colocado el pivote se aplica la actualización de las filas posteriores a la fila pivote eliminando todos los valores de la primera columna con el pivote, escalonando así la matriz. El resultado se observa en la matriz (c).

En la siguiente iteración se selecciona el máximo valor de la siguiente fila $|5|$,entre las columnas restantes como se observa en la matriz (d). Esta primera fase del algoritmo de Eliminación Gaussiana con pivotaje parcial por columnas se repite para todas las filas hasta obtener la matriz (h), triangular superior, en la que la diagonal está formada por todos los pivotes calculados anteriormente**.**

Una vez finalizada la primera fase del algoritmo, se promediaría a aplicar la segunda fase del algoritmo, "la sustitución inversa", a la matriz diagonal superior obtenida (h) para encontrar la solución al sistema de ecuaciones lineal.

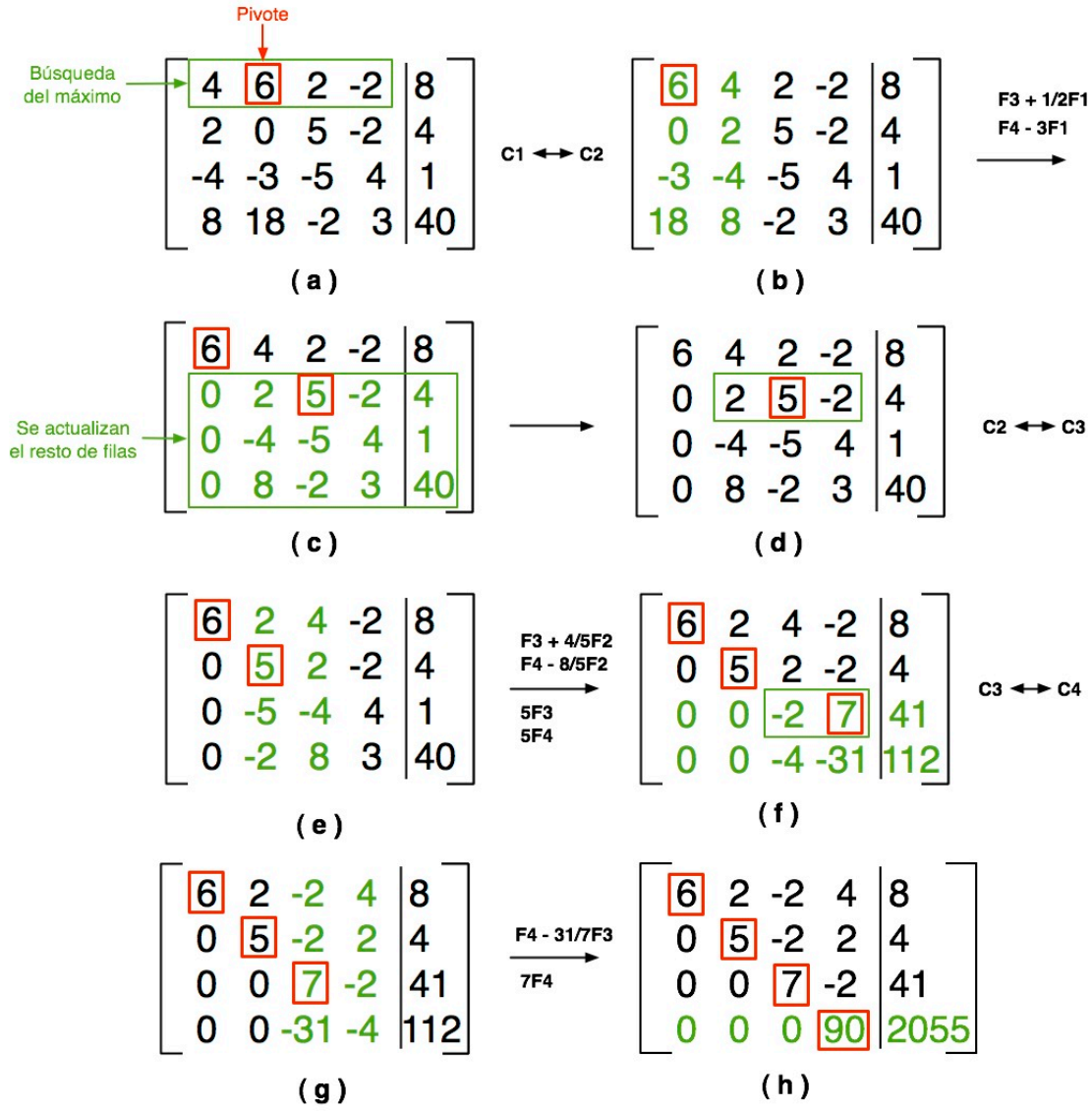

Figura 31. Ejemplo del algoritmo de Eliminación Gaussiana con pivotaje por columnas.

En resumen, muchos de los problemas actuales de la ingeniería y la ciencia pueden modelarse como un complejo sistema de ecuaciones lineales, por ello la resolución de sistemas de ecuaciones lineales posee una infinidad de aplicaciones y es por este motivo que se existe mucho interés en el desarrollo de aplicaciones eficientes que resuelvan estos complejos sistemas lineales en un lapso de tiempo menor con una mayor precisión para obtener la solución más adecuada.

La Eliminación Gaussiana es un eficiente algoritmo para la resolución de sistemas de ecuaciones lineales dividido en dos fases la primera llamada "Eliminación hacia delante" en la que se reduce la matriz ampliada del sistema a una matriz diagonal superior y la segunda fase llamada "sustitución inversa" en la que partiendo de la matriz diagonal superior, se obtienen los valores de las diferentes incógnitas resolviendo así el sistema lineal. La Eliminación Gaussiana, es más estable en las matrices diagonal dominantes, es por este motivo por el cual se aplica la estrategia del pivotaje total o parcial para precisar el resultado.

Una vez descrito el problema, en el capítulo siguiente, se realizará un estudio en profundidad del algoritmo secuencial de la Eliminación Gaussiana con pivotaje parcial independiente del procesador. Se codificará el algoritmo serie y se propondrán una serie de posibles optimizaciones a realizar para ser más tarde evaluadas en diferentes microprocesadores.

[2,9]

4 Análisis y Evaluación del algoritmo secuencial

En este capítulo se realizará el estudio del algoritmo secuencial para la Eliminación Gaussiana. En el apartado 4.1 efectuará el diseño del algoritmo serie de la aplicación. Mostraremos el pseudocódigo del algoritmo que más tarde se implementará en código C y las mejoras introducidas para realizar la versión base de la aplicación. Para ello se realizará en el apartado 4.2 un análisis computacional del algoritmo independiente del procesador, para determinar los parámetros que determinan el problema, las zonas de mayor peso computacional, para ser optimizadas. A continuación en el apartado 4.3 se realizarán, a partir de los datos obtenidos en el estudio anterior, una serie de propuestas de optimizaciones. Estas optimizaciones darán lugar a diferentes versiones de la aplicación, que se ejecutarán en cada uno de los procesadores considerados, tomando medidas del tiempo de ejecución y de valores de ciertos contadores hardware del procesador.

En este capítulo se considerará la ejecución sobre un único núcleo de procesamiento. Será en el capítulo siguiente donde se estudiará la ejecución en un procesador multi-núcleo del algoritmo de Eliminación Gaussiana. Para ello, habrá que dividir la ejecución en múltiples hilos de procesamiento.

4.1 Diseño del algoritmo serie

Tras la descripción del algoritmo y los ejemplos de la Eliminación Gaussiana, en el capítulo anterior, en la figura 32 se muestra el diagrama de flujo de la Eliminación Gaussiana con pivotaje parcial por filas. Tomaremos este diagrama como base para modelar el algoritmo serie en código C sobre el cual se aplicará el estudio más adelante para optimizarlo y paralelizarlo. Tomaremos el algoritmo de pivotaje parcial ya que es menos costoso computacionalmente hablando que el pivotaje total y el pivotaje parcial es suficiente para conseguir la estabilidad del algoritmo.

Observamos que se debe buscar el pivote parcial por cada columna, e intercambiar la posición de la fila actual por la del nuevo pivote. Una vez intercambiadas las posiciones con la nueva fila pivote, se deben actualizar una a una todas las filas restantes hasta acabar escalonando la matriz por completo.

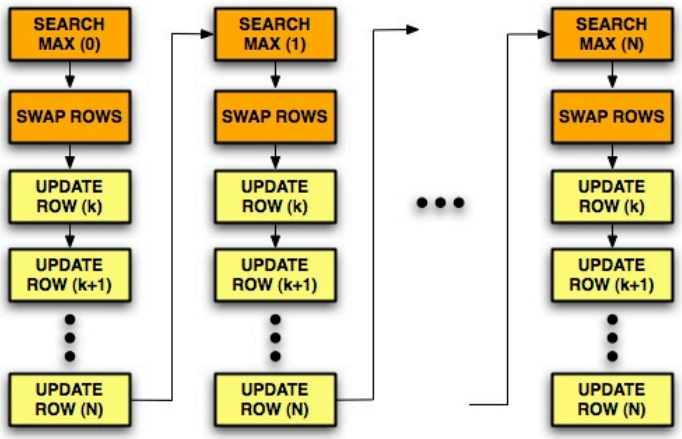

Figura 32. Diagrama de flujo del algoritmo serie de la Eliminación Gaussiana con pivotaje parcial por filas.

El primer paso para modelar el algoritmo serie, es generar el pseudocódigo. Este paso intermedio entre el diagrama y la codificación en un lenguaje de programación formal, permite representar, de forma más sencilla, las operaciones repetitivas complejas así como facilita la observación y estudio de los niveles de anidación en la estructura del programa y su complejidad. El pseudocódigo también nos permitirá en el siguiente apartado poder realizar el análisis computacional del algoritmo para más tarde codificarlo en cualquier lenguaje de programación formal, en nuestro caso el lenguaje C. El pseudocódigo generado a partir del diagrama de flujo de la figura 32 lo podemos encontrar en la figura 33.[2]

```
  { inicializar el vector de índices "fila" }
for i = 0 to n-1
  fila[i]=i
end for

for i = 0 to n-1
  { encontrar el pivote }

  max = 0
  for j = i to n-1
     if | A[fila[j]][i] | > max then
        max = | A[fila[j]][i] |
        pivote = j
     end if
  end for

  tmp = fila[i]
  fila[i] = fila[j]
  fila[j] = tmp

  { actualizar filas con el pivote }
  for j = i+1 to n-1
     t = a[fila[j]][i] / a[fila[i]][i]
     for k = i+1 to n+1
        a[fila[j]][k] = a[fila[i]][k] - a[fila[i]][k]*t
     end for
  end for
end for
```

Figura 33. Pseudocódigo de la Eliminación Gaussiana con pivotaje parcial por filas.

Substituir vector de índices por vector de punteros: Versión 0

Para el diseño de la versión base del algoritmo serie, se procedió a codificar el pseudocódigo de la Eliminación Gaussiana con pivotaje parcial por filas que se muestra en la figura 33. Sobre este algoritmo se realizaron una serie de optimizaciones. Se redujo una iteración en la búsqueda del máximo realizando una primera inicialización de las variables fuera del bucle. También se eliminaron las llamadas a una función para el cálculo del valor absoluto. Y se procedió a eliminar el vector de índices (fila) que se emplea para realizar los accesos indirectos a la matriz por un vector de punteros. Este cambio implica un cambio en la estructura de datos, una optimización que el compilador difícilmente es capaz de realizar y que implica una gran mejora en el rendimiento de la aplicación.

A continuación, en la figura 34, se muestra de manera gráfica los cambios realizados en el algoritmo de la versión base para su posterior análisis computacional. El código que se aprecia en la parte izquierda de la figura muestra el código antes de las modificaciones. A la derecha se muestra el código resultante tras las pequeñas optimizaciones realizadas.

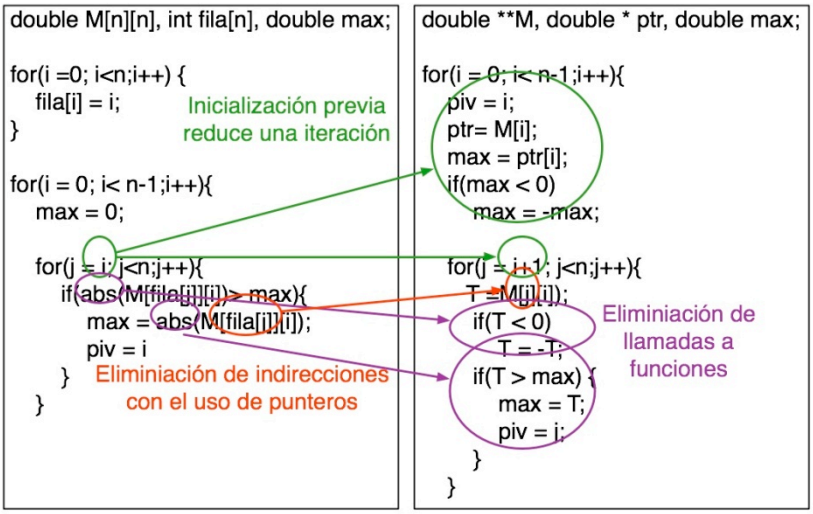

Figura 34. Mejoras realizadas en el pseudo-código de la versión serie al codificar la versión base en C.

El código C, de la versión base del algoritmo se adjunta en el anexo A.

4.2 Análisis Computacional del Algoritmo

El primer paso para modelar el algoritmo es identificar los parámetros que determinan el tamaño del problema. Para la Eliminación Gaussiana el tamaño del problema viene determinada por la cantidad de incógnitas (**n**) y ecuaciones (**n**) que definen el sistema de ecuaciones lineal a resolver. Por tanto, el parámetro que determina el tamaño del problema es **n**, la cantidad de filas/columnas que posee la matriz asociada al sistema de ecuaciones.

El segundo paso para modelar el algoritmo consiste en calcular la complejidad del algoritmo en función del tamaño del problema. La Eliminación Gaussiana sin pivotaje para resolver sistemas de ecuaciones n x n requiere un total aproximado de n^3 operaciones. Por tanto, la complejidad de la Eliminación Gaussiana es $O(n^3)$. A continuación calcularemos de forma explícita la complejidad del algoritmo de Eliminación Gaussiana con pivotaje por filas, y comprobaremos que es la misma que sin pivotaje.

En el algoritmo que se observa en la figura 35, el bucle más externo se ejecuta n veces. Dentro de este bucle hay dos bucles, el primero (1.1) que calcula el pivote de la fila y el segundo (1.2) que actualiza el resto de filas con el pivote. tanto el primer bucle como el segundo se ejecutan (n-1)(n-2)/2 veces. Dentro del segundo bucle hay otro bucle (1.2.1) que se ejecuta $(n-1)^2(n-2)/4$ veces.

En la primera fase del algoritmo, en la búsqueda del pivote, se realizan $2n^2-6n+4$ lecturas a memoria para calcular las indirecciones y para acceder a los valores de la matriz y $2n^2-6n+4$ saltos condicionales de ellos la mitad tienen un comportamiento irregular ya que son dependientes de los datos por lo que son difícilmente predecibles y la otra mitad tienen un comportamiento regular fáciles de predecir. En la actualización de las filas, en el bucle 1.2, se realizan $4n^2-12n+8$ lecturas a memoria para calcular las indirecciones y para acceder a los valores de la matriz y n^2-3n+2 saltos condicionales de comportamiento regular. En el bucle principal ,1.2.1, se realizan $4n^3-8n^2-4n+8$ lecturas a memoria para calcular las indirecciones y para acceder a los valores de la matriz, n^3-2n^2-n+2 escrituras a memoria para actualizar los valores, n^3-2n^2-n+2 restas, multiplicaciones y saltos de comportamiento regular. Por tanto la complejidad del algoritmo es $O(n^3)$, la misma complejidad que la del algoritmo de Eliminación Gaussiana sin pivotaje.

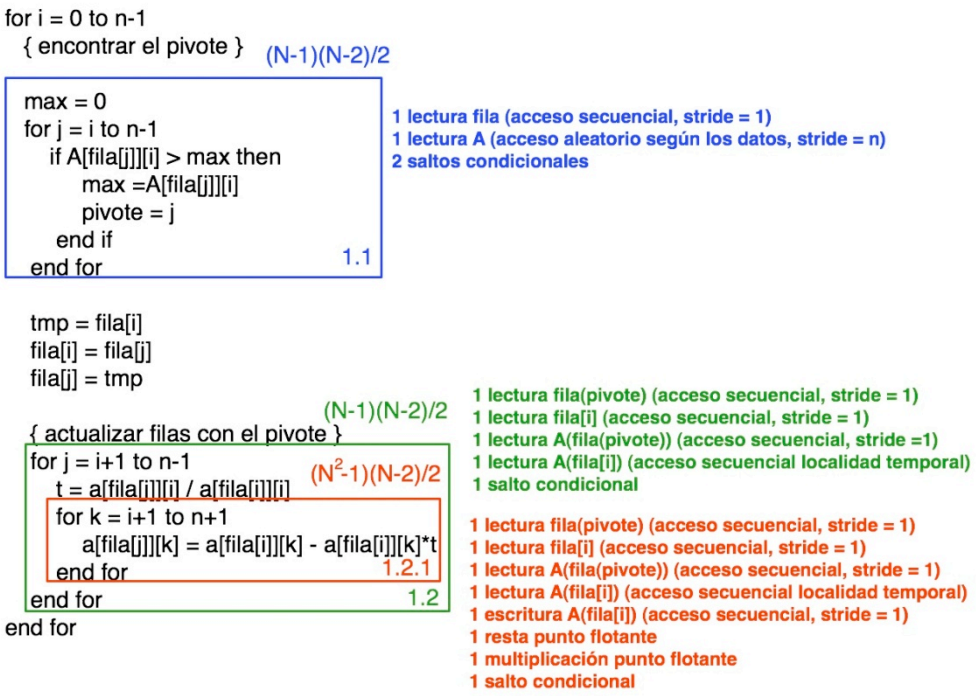

Figura 35. Estudio del pseudocódigo de la Eliminación Gaussiana con pivotaje por filas.

El tercer paso para modelar el algoritmo consiste en identificar los patrones de acceso a memoria en función del tamaño del problema.
Los requerimientos de almacenamiento del algoritmo crecen de forma cuadrática (n^2). La matriz asociada al sistema consta de $n \times n$ *doubles* (8 Bytes).

En la Eliminación Gaussiana con pivotaje por filas, como se observa en la Figura 35 en la primera fase del algoritmo, en la búsqueda del pivote, se realizan dos accesos de lectura regulares secuenciales (stride = 1) recorriendo el vector de indirecciones a las filas una para la fila pivote y otra para la fila a actualizar y se realiza un acceso regular aleatorio (stride = n) a las posiciones candidatas a ser el pivote de la matriz de datos.

En la fase de actualización de las filas con la fila pivote, se realizan dos accesos de lectura regulares secuenciales recorriendo el vector de indirecciones una para la fila pivote y otra para la fila a actualizar, un acceso de lectura regular secuencial a la posición de la matriz a actualizar y un acceso de escritura secuencial para actualizar la posición de la matriz. En esta fase existe *localidad temporal* [3] con la fila pivote ya que se accede en cada iteración del bucle principal (1.2) para actualizar la fila correspondiente y mucha *localidad espacial* [4] ya que en cada iteración del bucle se hace referencia a posiciones secuenciales de la matriz que corresponden a posiciones cercanas en la memoria.

4.3 Propuestas de Optimizaciones

Una vez estudiado el algoritmo la Eliminación Gaussiana con pivotaje por filas, observamos que el bucle 1.2.1 es la zona que posee una mayor complejidad $O(n^3)$. Este bucle 1.2.1 es la zona de código en la que se realizan más iteraciones y por tanto la parte que más peso tiene en tiempo total de ejecución. Por tanto es la zona de código dónde deberíamos aplicar las optimizaciones para que sean realmente eficientes.

---

[3] **Localidad Temporal:** si en un momento una posición de memoria particular es referenciada, entonces es muy probable que la misma ubicación vuelva a ser referenciada en un futuro cercano. Existe proximidad temporal entre las referencias adyacentes a la misma posición de memoria. En este caso es común almacenar una copia de los datos referenciados en caché para lograr un acceso más rápido a ellos.

[4] **Localidad Espacial:** si una localización de memoria es referenciada en un momento concreto, es probable que las localizaciones cercanas a ella sean también referenciadas pronto. Existe localidad espacial entre las posiciones de memoria que son referenciadas en momentos cercanos. En este caso es común estimar las posiciones cercanas para que estas tengan un acceso más rápido.

Las optimizaciones que a priori se proponen son las siguientes:

1. Substituir instrucciones if-then-else por expresiones condicionales que son evaluadas en tiempo de ejecución y no provocan ningún salto en el flujo de la aplicación.

2. Fusionar los bucles 1.1 y 1.2 para aumentar la concurrencia de operaciones, la reusabilidad de los datos, y mejora el uso de la caché.

3. Desplegar el bucle 1.2 para reducir el número de saltos condicionales, facilitar el reordenamiento de instrucciones exponiendo más paralelismo a nivel de instrucción, y reducir el número de instrucciones y, por tanto, el tiempo de CPU.

4. Fusionar los bucles 1.1 y 1.2 para aumentar la concurrencia de operaciones y la reusabilidad de datos y desplegar esta fusión para reducir el número de saltos condicionales y reducir el numero de instrucciones ejecutadas.

4.3.1 Substituir instrucciones if-then-else por expresiones condicionales: Versión 1

En la versión 1 se optimizó la versión 0 para evitar los saltos condicionales. Estos saltos condicionales se modificaron por expresiones condicionales que son evaluadas en tiempo de ejecución y no provocan ningún salto en el flujo de la aplicación. A continuación, en la figura 36, se muestra de manera gráfica los cambios realizados en el algoritmo de la versión base para la primera optimización. El código que se aprecia en la parte izquierda de la figura muestra el código antes de las modificaciones. A la derecha se muestra el código resultante tras la optimización.

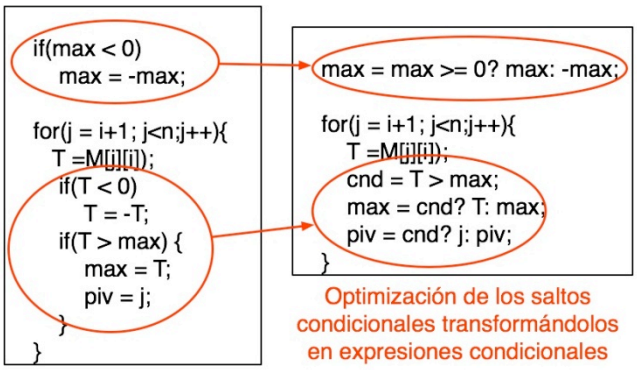

Figura 36. Cambios realizados en la versión 1 del algoritmo serie (optimización de instrucciones de salto por expresiones condicionales).

4.3.2 Fusión de Bucles: Versión 2

En la versión 2 se optimizó la versión 1 fusionando la búsqueda del pivote con la actualización de las filas (*loop fusion*/*fuse loops*[5]). Como se muestra en la figura 37, en esta versión el primer cálculo del pivote se realiza en un bucle fuera del programa. Las sucesivas búsquedas de los pivotes se realizan simultáneamente con el intercambio de filas del ciclo anterior, para reducir considerablemente las instrucciones a ejecutar, permitir la reutilización de datos y mejorar el uso de la caché.

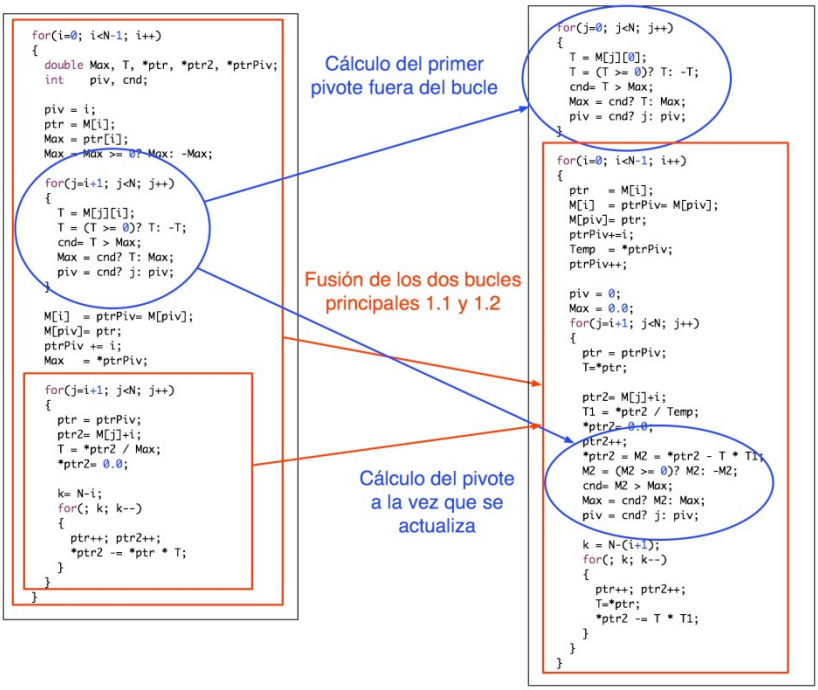

Figura 37. Cambios realizados en la versión 2 del algoritmo serie(*fuse loops)*.

[5] **Fusión de Bucles (fuse loops):** La fusión de bucles es una optimización convencional del compilador que transforma dos o más bucles adyacentes en uno solo. Así mismo permite la re-usabilidad de los datos (que están en los registros del CPU) y una mejora en el uso del caché (si el bucle es grande). El uso de las pruebas de dependencias de datos permite la fusión de bucles, tanto como sean posibles.

4.3.3 Desenrollar Bucles: Versión 3

En la versión 3 se optimizó la versión 1 a realizar un *loop unrolling*[6] desplegando los dos bucles principales para aumentar la concurrencia de las operaciones de éstos. La búsqueda del pivote se realiza de manera independiente y una vez encontrado el nuevo pivote se procede la actualización de las filas. El segundo bucle, la actualización de las filas con el nuevo pivote, se despliega realizando la actualización simultánea de dos filas. También se añade una iteración más para la actualización de las filas impares. En la figura 38, observamos como queda el bucle principal una vez desplegado.

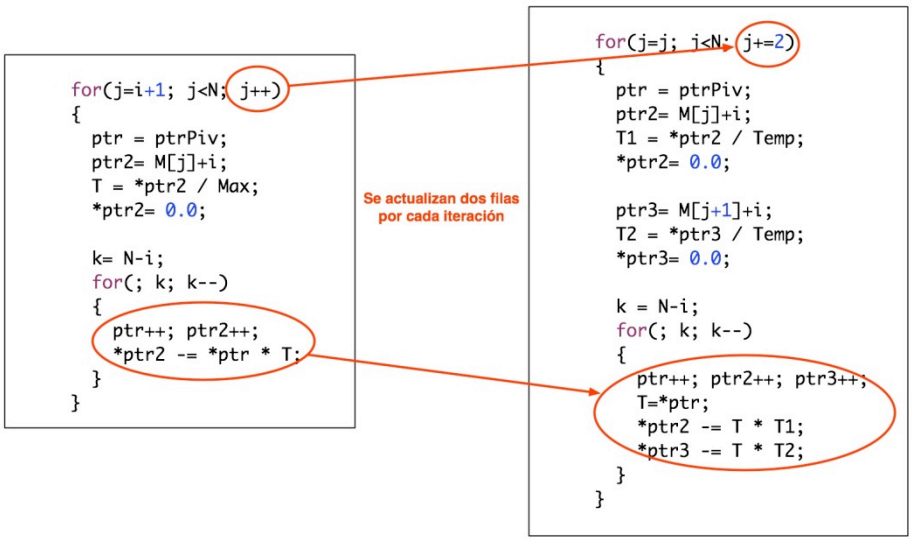

Figura 38. Cambios realizados en la versión 1 del algoritmo serie (*Loop unrolling*).

4.3.4 Fusión y Desenrollar Bucles: Versión 4

En la versión 4 se fusionaron las optimizaciones realizas en las versiones 2 y 3 de manera que la primera búsqueda del pivote se realiza fuera del bucle principal, las sucesivas búsquedas del pivote se realizan simultáneamente mientras se actualizan las filas, se despliegan los dos bucles principales para aumentar la concurrencia de las operaciones y se despliega el segundo bucle realizando la actualización simultanea de dos filas, añadiendo una iteración para las filas impares.

[6] **Desenrollado de Bucles (Loop Unrolling):** Esta técnica de optimización consiste en desenrollar el cuerpo del bucles dos o más veces incrementando los saltos del bucle, con el fin de mejorar el reuso de registros, minimizar recurrencias y de exponer más paralelismo a nivel instrucción.

4.4 Análisis del código Ensamblador

Una vez codificadas las diferentes optimizaciones del algoritmo procederemos a analizar el código ensamblador generado por los diferentes procesadores. De esta manera podremos evaluar la efectividad de las optimizaciones realizadas. Mediante el análisis del código ensamblador podemos observar la cantidad de instrucciones generadas por el compilador por cada operación del alto nivel. Con el recuento de instrucciones podemos apreciar si las optimizaciones ya las realizaba el compilador o por el contrario si realmente son efectivas.

4.4.1 Versión 0

A continuación en la figura 39 mostramos el recuento de operaciones dividido en grupos del código ensamblador generado en el procesador SUN UltraSPARC T2. Este recuento recoge sólo la parte del bucle 1.2.1 que se observa en la figura 35. Seguidamente mostraremos el mismo recuento para cada una de las versiones generadas de la aplicación cada una asociada a una mejora realizada para evaluar su efectividad.

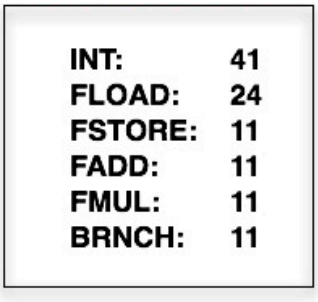

Figura 39. Cuadro resumen de las instrucciones ensamblador generadas por el compilador para la versión base de la aplicación.

4.4.2 Substituir instrucciones if-then-else por expresiones condicionales: Versión 1

En la figura 40 mostramos la cantidad de operaciones dividida en grupos que se han generado en código ensamblador. Observamos que no existe ninguna mejora respecto a la versión anterior. Esto se debe a que en la zona estudiada no existe ninguna instrucción/expresión condicional y por tanto no se observa ninguna mejora.

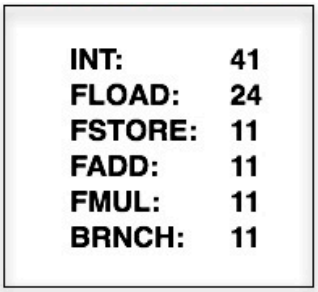

Figura 40. Cuadro resumen de las instrucciones ensamblador generadas por el compilador para la versión 1 de la aplicación.

4.4.3 Fusión de Bucles: Versión 2

En la figura 41 mostramos la cantidad de operaciones dividida en grupos que se han generado en código ensamblador en las diferentes máquinas. Observamos que tras la fusión de bucles el compilador reduce ligeramente la cantidad de operaciones enteras. Aunque la mejora no es sustancial, esta pequeña mejora debería notarse al realizarse en la parte del código con mayor complejidad. De esta manera podemos afirmar que la optimización de la versión 2 ha sido efectiva aunque no tanto como nos esperábamos a priori. Esta fusión es una tarea que el compilador difícilmente podría realizar y por este motivo los resultados, que observaremos más adelante, deberían mostrar esa mejora respecto a la versión base de la aplicación.

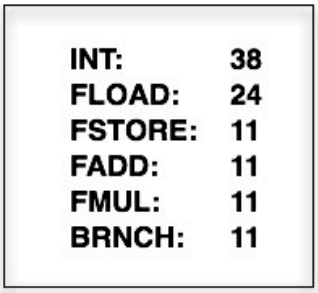

Figura 41. Cuadro resumen de las instrucciones ensamblador generadas por el compilador para la versión 2 de la aplicación.

4.4.4 Desenrollar Bucles: Versión 3

En este caso, observamos que el recuento de instrucciones aumenta considerablemente respecto a la versión base y las versiones anteriores de la aplicación. Esto se debe a que esta versión realiza el despliegue del bucle con más complejidad realizando simultáneamente dos iteraciones. Observamos pues, que cuando antes se necesitaban ejecutar en el procesador UltraSPARC $41 \times 2 = 82$ operaciones enteras con la versión 3 tan sólo se ejecutan 67, y lo mismo ocurre con las operaciones de carga en punto flotante (FLOAD) $24 \times 2 = 48$ operaciones en este caso se ejecutan 26. Podemos concluir de esta manera que esta mejora resulta realmente efectiva ya que se reduce considerablemente el número de operaciones ejecutadas y por tanto observaremos más adelante un aumento sustancial del rendimiento de la aplicación para esta versión

INT:	67
FLOAD:	26
FSTORE:	16
FADD:	16
FMUL:	16
BRNCH:	10

Figura 42. Cuadro resumen de las instrucciones ensamblador generadas por el compilador de los diferentes procesadores para la versión 3 de la aplicación.

En la figura 43, mostramos una gráfica de las instrucciones ejecutadas normalizadas con la complejidad del algoritmo de la Eliminación Gaussiana para visualizar de manera gráfica y sencilla el comportamiento de la aplicación. Estos datos se han extraído de los contadores hardware del procesador AMD mediante PAPI. La figura muestra de manera gráfica la cantidad de instrucciones ejecutadas a lo largo de los diferentes experimentos realizados que describirán con más detalle en los apartados (4.5.1 y 4.5.2). En esta gráfica podemos observar en azul la cantidad de instrucciones ejecutadas por la versión base del algoritmo para los valores de N escogidos y en rojo las ejecutadas por la optimización 3 .Esta figura no hace más que corroborar el descenso de instrucciones ejecutadas observado en el código ensamblador generado y la mejora que posteriormente se verá reflejada en el rendimiento de la versión.

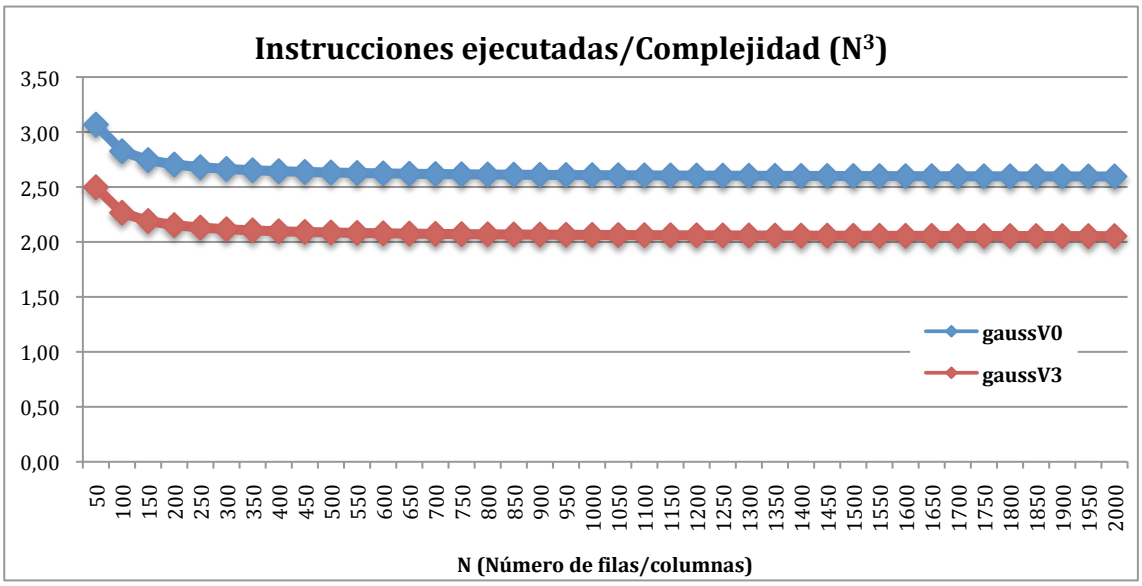

Figura 43. Comparación de la cantidad de instrucciones ejecutadas entre la versión base y la versión 3 (AMD Dual Core).

4.4.5 Fusión y Desenrollar Bucles: Versión 4

Por último en la versión 4, observamos que de la misma forma que en la versión anterior aumenta considerablemente la cantidad de operaciones en el bucle fusionado y desplegado. Observamos también que aumenta ligeramente el número de operaciones enteras, lecturas en punto flotante y saltos incondicionales respecto a la versión 3. De todos modos sigue siendo una mejora sustancial respecto a las 82 instrucciones de la versión base que se ejecutarían para dos iteraciones, ya que en este caso se ejecutan 75.

Estas optimizaciones realizadas en esta versión difícilmente las podría realizar el compilador. De esta manera podemos afirmar que las optimizaciones realizadas en la versión 4, la fusión y el despliegue del bucle fusionado son realmente efectivas.

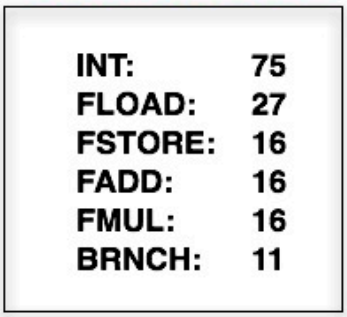

Figura 44. Cuadro resumen de las instrucciones ensamblador generadas por el compilador para la versión 4 de la aplicación.

En la figura 45, observamos cómo ya hicimos en la versión anterior, la gráfica de la cantidad de instrucciones ejecutadas normalizadas por la complejidad durante los diferentes experimentos realizados. Los datos se han extraído de los contadores hardware del procesador AMD mediante PAPI. Esta gráfica corrobora la mejora observada en el código ensamblador generado que comentábamos anteriormente y se asemeja a la curva descrita por la versión 3 por lo que estas dos versiones son las candidatas a priori a ser las dos versiones con un mejor rendimiento.

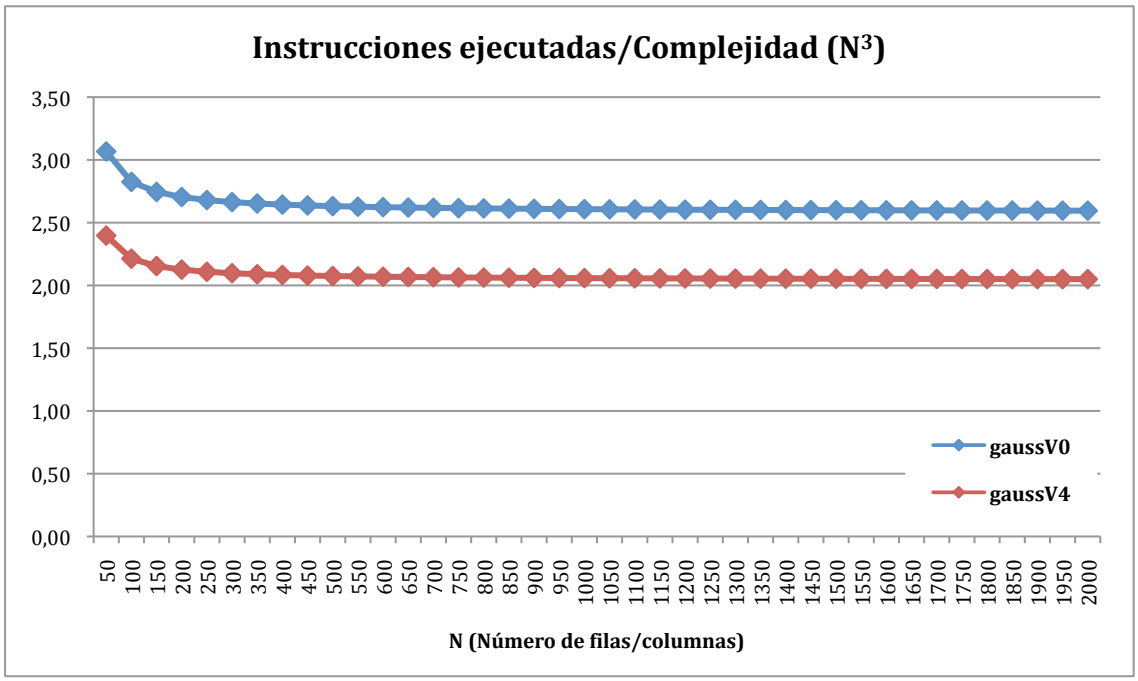

Figura 45. Comparación de la cantidad de instrucciones ejecutadas entre la versión base y la versión 4 (AMD Dual Core).

4.5 Evaluación de las optimizaciones realizadas

En este apartado se mostrarán los resultados obtenidos tras la ejecución de la aplicación paralela. En el apartado 4.5.1 se mostrarán las principales características de los procesadores utilizados en los experimentos. En el apartado 4.5.2 se mostrará la metodología experimental seguida para la obtención de los resultados. En el apartado 4.5.3 se realizará el estudio de los resultados obtenidos del programa base para que se pueda comparar con los resultados conseguidos por las diferentes optimizaciones realizadas. El estudio de los resultados obtenidos por las diferentes optimizaciones se mostrará en el apartado 4.5.4 . Finalmente en el aparatado 4.5.5 se realizará un breve resumen de los resultados obtenidos y se procederá a seleccionar la mejor versión. Esta versión servirá de base para el siguiente capítulo en el que se realizará la paralelización de la aplicación.

4.5.1 Procesadores utilizados

Para realizar las pruebas, se han seleccionado tres configuraciones de núcleos de procesamiento completamente diferentes, de esta manera podremos estudiar el comportamiento del algoritmo paralelo, dividiendo la ejecución en varios hilos. Aunque en este apartado estudiaremos la ejecución serie, Los procesadores escogidos son los siguientes: El **AMD Athlon 64 Dual Core**, un procesador de dos núcleos de procesamiento. El **Intel Core2 Quad**, un procesador de última generación con cuatro núcleos de procesamiento. El **Sun UltraSPARC T2**, un procesador de ejecución en orden de 4 núcleos de procesamiento. A continuación en la figura 46 se muestran las características principales de cada procesador.

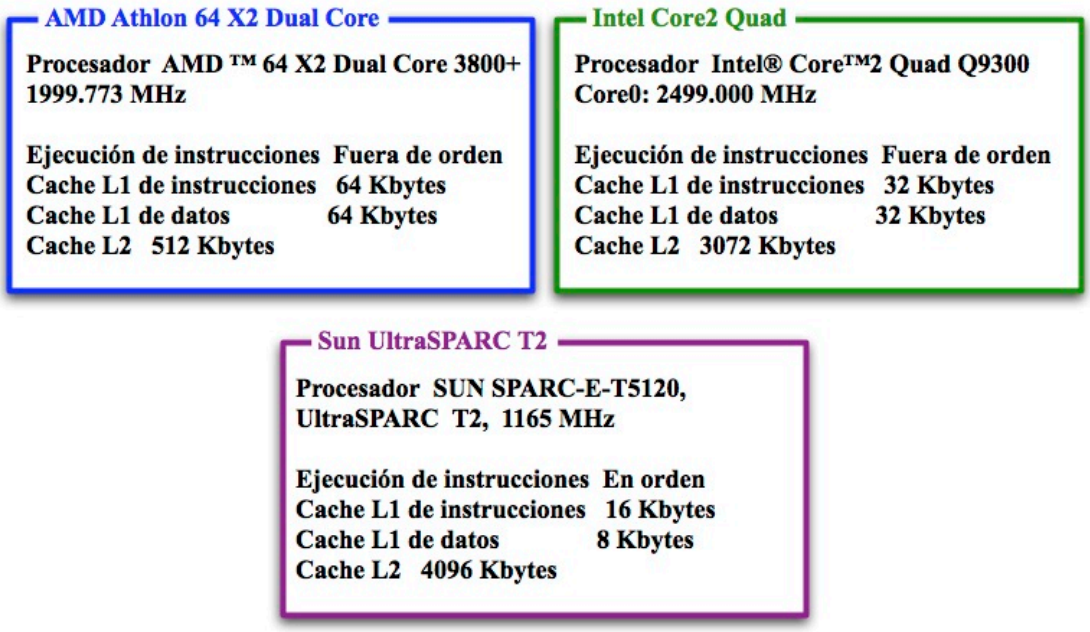

Figura 46. Características principales de los procesadores escogidos para el estudio.

4.5.2 Metodología experimental

La métrica fundamental que se ha medido en las ejecuciones ha sido el tiempo de ejecución del programa, excluyendo la parte de entrada/salida , de inicialización de datos de entrada y de verificación del resultado. Aunque la métrica obtenida en los experimentos es el tiempo de ejecución total del algoritmo, para facilitar el estudio y comprensión de los resultados, en las gráficas mostradas utilizaremos el tiempo de ejecución dividido por la complejidad del algoritmo (N^3). De ahora en adelante, al tiempo de ejecución dividido por la complejidad, le llamaremos "*tiempo normalizado*" (*ns* /n^3). En las ejecuciones se ha variado el número de Filas/Columnas, N, entre 50 y 2000 de 50 en 50.

Tal como hemos evaluado en apartados anteriores la complejidad espacial del algoritmo de Eliminación Gaussiana es de $8n^2$ Bytes, lo que supone que con n=50 se necesitan 20KB de memoria, y con n= 2000 se necesitan 32MB de memoria. Con los procesadores que se han considerado, este valor de **N** es suficiente para analizar el rendimiento cuando los datos no caben en la memoria caché de los procesadores, y hay que obtenerlos de la memoria principal. Se necesitan valores mucho más grandes de n=2000 para forzar al sistema a tener que utilizar el almacenamiento secundario. Como la complejidad de cómputo es $O(n^3)$, el tiempo de cómputo crecería demasiado, (4GB con n=22.500, el tiempo de ejecución en un procesador sería del excesivamente grande), es por este motivo que se han elegido estos valores para realizar los experimentos.

Las mediciones de tiempo y otros eventos ligados al rendimiento para cada uno de los threads las realizamos con llamadas a unas funciones especiales, que se incluyen al inicio y al final de la parte del código que queremos medir. Estas funciones se enlazan a la aplicación en tiempo de compilación, y se puede elegir dos versiones: una que hace únicamente medidas de tiempo de ejecución en microsegundos (utilizando las funciones de tiempo del sistema operativo), y otra que accede a los contadores hardware del procesador. Esta última versión utiliza "PAPI" (Performance Application Programming Interface), en concreto las versiones 3.4.9 y 3.6.2 (http://icl.cs.utk.edu/ ").

En los sistemas en los que se ha usado PAPI, las medidas que se han tomado han sido la cantidad de instrucciones máquina ejecutadas (PAPI_TOT_INS), la cantidad de fallos (PAPI_L1/2_DCM) y accesos (PAPI_L1/2_DCA) en L1 y L2, la cantidad de operaciones suma (PAPI_FAD_INS) y multiplicación (PAPI_FML_INS) en punto flotante y la cantidad total de instrucciones de salto ejecutadas (PAPI_BR_INS).

Para cada una de las pruebas se realizan 7 ejecuciones de las cuales se descartan 4 con el objetivo de filtrar las perturbaciones y valores atípicos (*outliers*) provocados por procesos del sistema operativo, o por la ejecución de procesos de otros usuarios del sistema. Con los valores de las 3 ejecuciones restantes se calcula la media aritmética.

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

Para determinar la variabilidad de la muestra obtenida se calcula la *desviación estándar*[7], también conocida como desviación típica.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(X_i - \bar{X})^2}{n-1}}$$

[7] **La desviación estándar:** es la medida de la magnitud en que se desvían las diversas puntuaciones obtenidas de su valor medio. Si las puntuaciones se agrupan estrechamente en torno a la media, la desviación estándar será relativamente pequeña; si se extienden en todas direcciones, la desviación estándar será relativamente grande.

4.5.3 Resultados de Rendimiento del programa base

En este apartado se mostrarán los resultados obtenidos tras ejecutar la versión del programa base en los diferentes procesadores mediante la metodología experimental explicada en el apartado anterior. De esta forma obtendremos la gráfica de resultados del tiempo de ejecución de nuestra aplicación en los diferentes procesadores. Mediante la extracción de datos con PAPI de los contadores hardware se interpretarán los datos obtenidos para averiguar el causante de los puntos flacos de la aplicación que merman su rendimiento. Finalmente se compararán los datos de los diferentes procesadores para extraer conclusiones.

El primer procesador que estudiaremos será el AMD Athlon 64. En la figura 47 se muestra la gráfica del tiempo de ejecución *normalizado* de la versión base para los diferentes valores de N escogidos. Observamos el tiempo de ejecución se mantiene por debajo de 1 para valores más pequeños de N=300, salvo para N=50 donde el tiempo de ejecución es mayor frente al poco trabajo que debe realizar. A partir de N=300 observamos un aumento sustancial en el tiempo de ejecución hasta llegar a N=550 donde se estanca. Una vez obtenidos estos valores, procederemos al estudio de los contadores hardware mediante PAPI para averiguar a que se debe este comportamiento de la aplicación base.

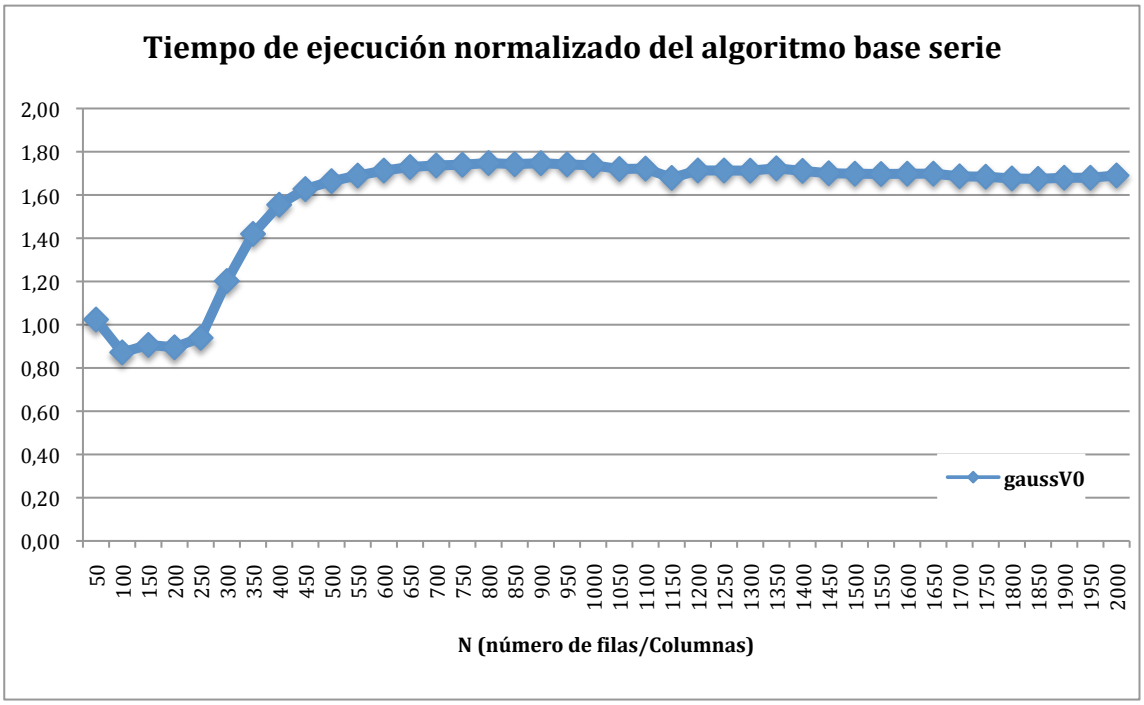

Figura 47. Tiempo de ejecución de la versió base del algoritmo serie (AMD Dual Core).

La gráfica “Ciclos de reloj/complejidad (n^3)” en la Figura 48, muestra la cantidad de ciclos de reloj utilizados durante la ejecución de la Eliminación Gaussiana en el procesador AMD. En esta curva podemos identificar claramente tres zonas bien diferenciadas. La primera zona delimitada entre N=50 y N=150, donde podemos observar que la matriz cabe en la memoria cache L1. La segunda zona comprendida entre N=150 y N=300, donde observamos que la matriz ya no cabe en la memoria cache L1 pero si en L2. Y la tercera zona a partir de N=300, en la que observamos que la matriz ya no cabe en las memorias cache.

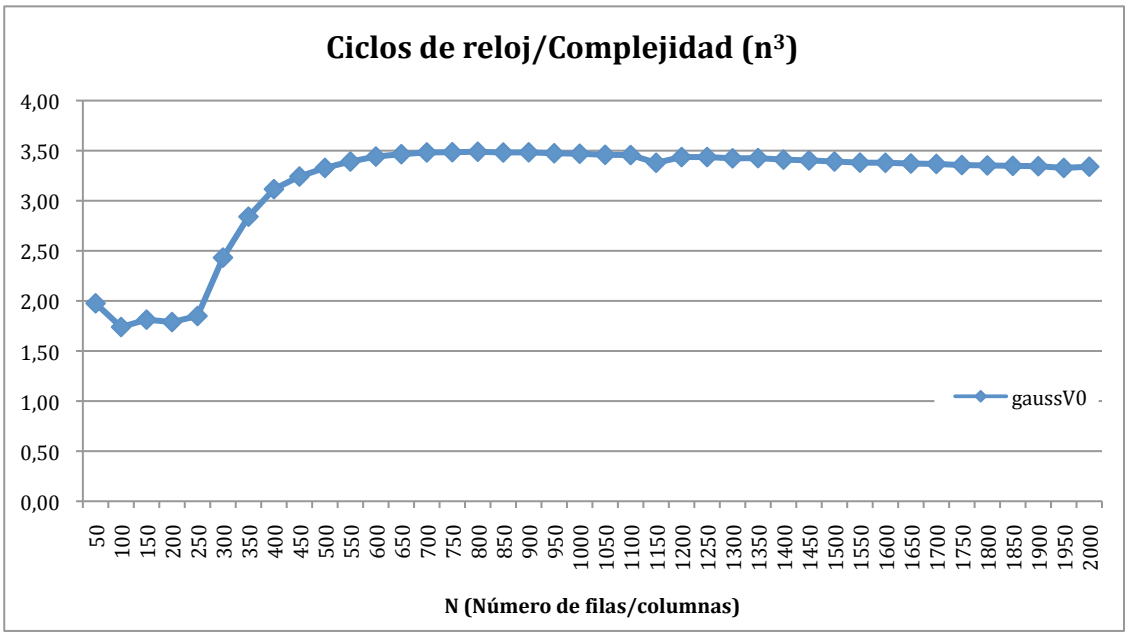

Figura 48. Gráfica de la versión base de ciclos de reloj dividida entre la complejidad del algoritmo en el procesador AMD.

En la figura 49, se muestra la gráfica de los ciclos por instrucciones ejecutados del algoritmo base de la versión serie en el procesador AMD Dual Core. Observamos que la curva es exactamente igual, a la de los ”ciclos de reloj / complejidad” y la del “tiempo de ejecución *normalizado*”. Estás tres gráficas muestran la misma información tomada mediante dos métodos diferentes. La figura 49 tomada mediante los contadores hardware a través de PAPI valida la medida del tiempo de ejecución tomada en nanosegundos y que la complejidad aproxima bien del número total de instrucciones ejecutadas, para valores de N grandes.

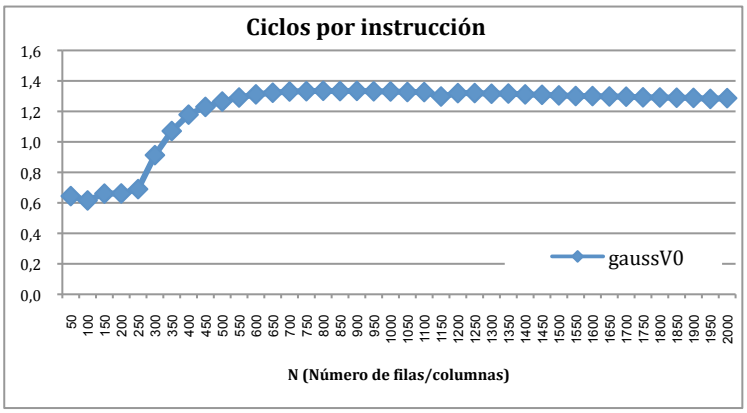

Figura 49. Ciclos por instrucción de la versión base (AMD Dual Core).

En la figura 50, en la primera zona, (ZONA 1), como podemos observar en la gráfica de "fallos en memoria cache L1", no existen fallos para N=50 ya que la matriz cabe totalmente en la memoria cache de primer nivel. Es a partir de N=100 cuando empieza a fallar dicha memoria pero estos fallos no afectan prácticamente al rendimiento de la aplicación.

En la segunda zona, (ZONA 2), como podemos observar en la gráfica de "fallos en memoria cache L1" y en la gráfica de "fallos en memoria cache L2", detectamos un aumento considerable de fallos en la memoria cache de primer nivel hasta llegar a un valor estable. Este comportamiento suele deberse al aumento del conjunto de los datos que experimentan una localidad temporal, que se produce al aumentar el tamaño del problema a partir de un cierto valor, en nuestro caso N= 150. Por el contrario, en esta segunda zona, si acierta la memoria cache de segundo nivel. Esta conducta se debe a que para este rango de valores de N la matriz no cabe totalmente en la cache de primer nivel pero si en la de segundo nivel. Este aumento de fallos considerable en la memoria de cache de primer nivel afecta ligeramente al rendimiento del programa al tener, éste, que acceder a la memoria de segundo nivel para resolver el fallo.

A partir de este momento en la tercera zona o (ZONA 3), apreciamos en la gráfica de "fallos en memoria cache L2", un rápido incremento del volumen de fallos en la memoria cache de segundo nivel hasta alcanzar un valor estable también provocado por el aumento del tamaño del problema y por consiguiente del conjunto de datos que experimentan localidad temporal. Este hecho se produce debido a que en esta zona la matriz ya no cabe en la memoria cache de segundo nivel y provoca que el programa deba buscar en la memoria principal para resolver los fallos. Este rápido incremento del volumen de fallos en la memoria cache de segundo nivel causa que aumente considerablemente el tiempo de ejecución de la aplicación.

Una vez que se estabilizan los fallos en la cache, también lo hace el tiempo de ejecución. Podemos afirmar por tanto que el problema de rendimiento de la aplicación se debe al incremento de los fallos en la cache de segundo nivel.

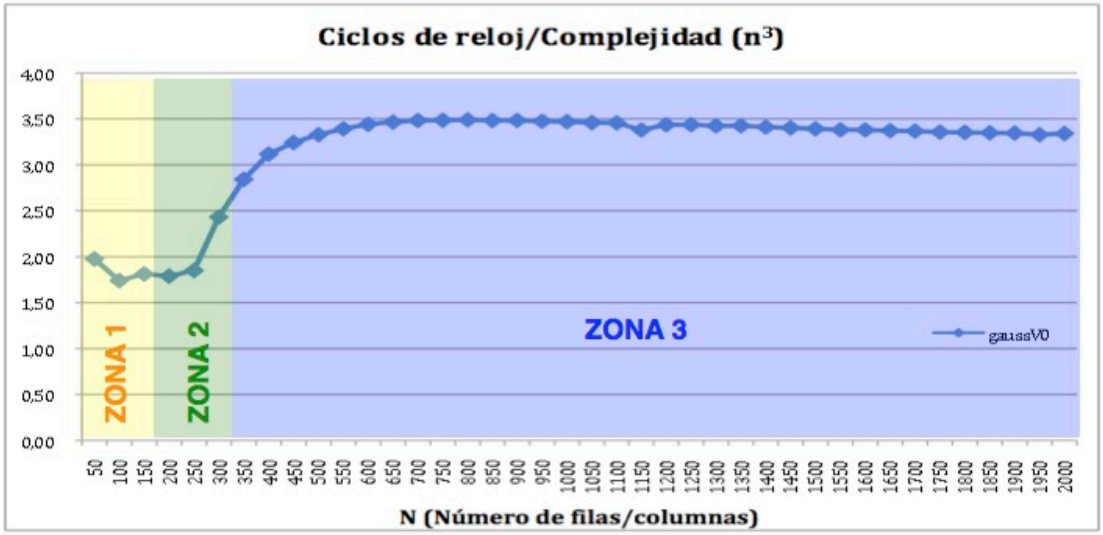

Figura 50. Ciclos de reloj de la versión base del algoritmo serie mostrando las 3 zonas.

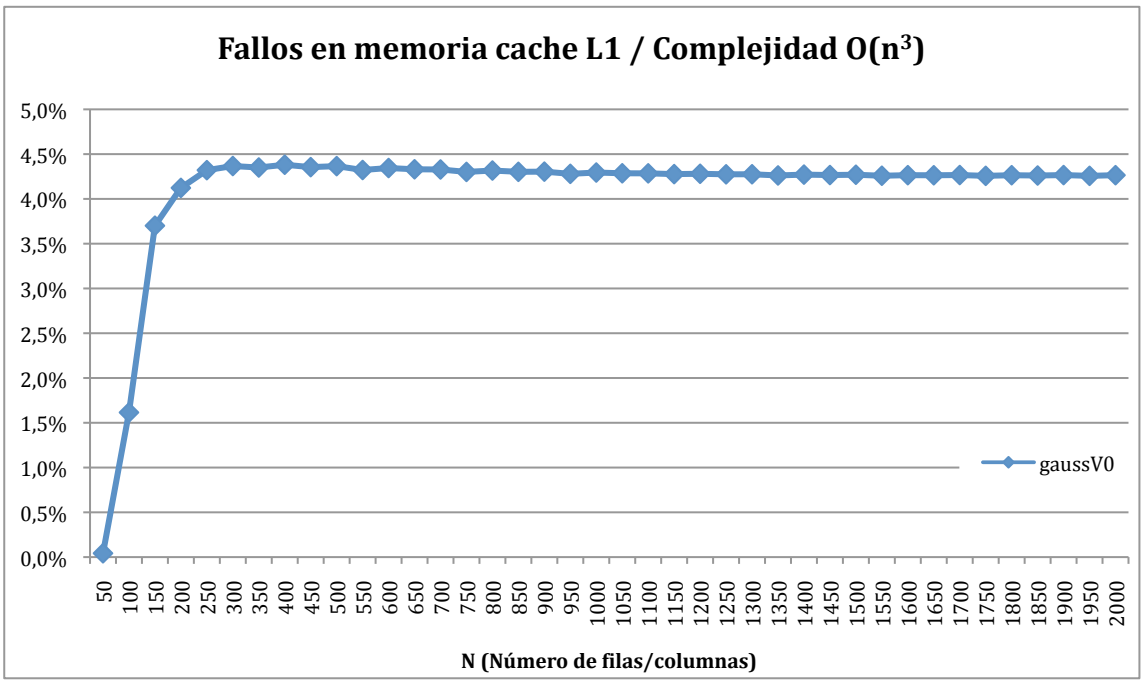

Figura 51. Fallos en la memoria cache de primer nivel (L1) de la versión base del algoritmo serie (AMD Dual Core)

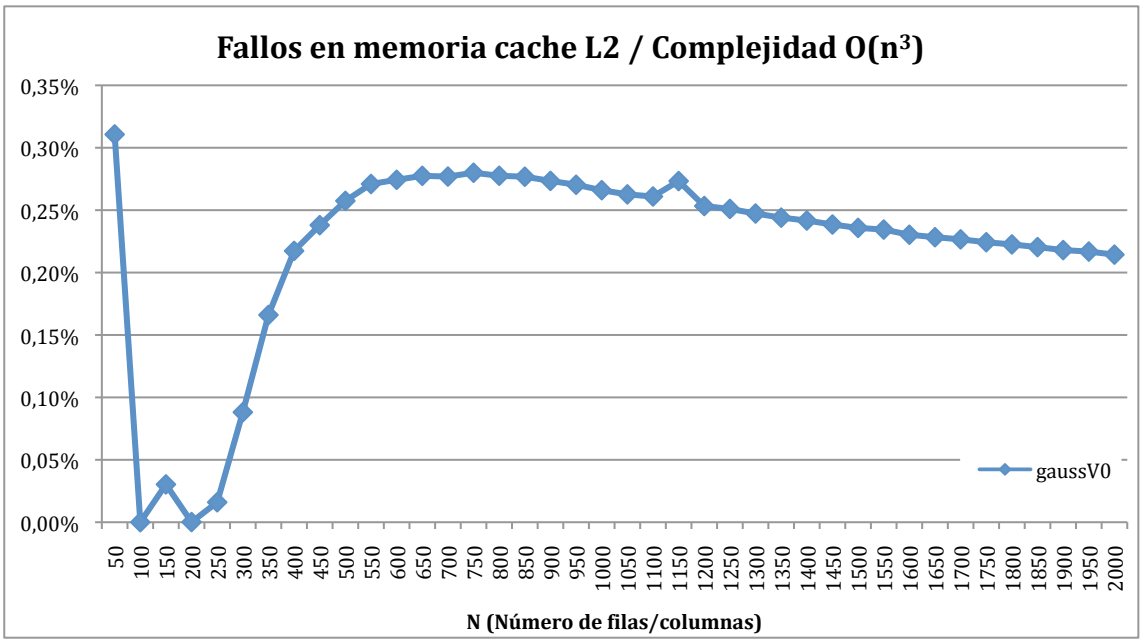

Figura 52. Fallos en la memoria cache de segundo nivel (L2) de la versión base del algoritmo serie (AMD Dual Core).

4.5.4 Resultados de Rendimiento

En este apartado se mostrarán los resultados obtenidos en los diferentes procesadores de las diferentes versiones generadas a partir de las optimizaciones realizadas como se explicada anteriormente. Estos resultados se compararán con los mostrados en el apartado anterior para extraer conclusiones. Finalmente se seleccionará la versión con mayor rendimiento como versión base para el desarrollo de la aplicación paralela.

En la figura 53 vemos la comparativa entre la versión base de la aplicación y las dos última optimizaciones realizadas, versión 3 y versión 4 respectivamente. Observamos que la versión 3 empeora los resultados obtenidos por la versión base hasta el valor de N=300, mejorándolos a partir de este valor considerablemente. Por otro lado, la versión 4 mejora los tiempos de la versión base ligeramente hasta el valor N=300 y es a partir de este valor, tal y como lo hace la versión 4 mejora formidablemente los tiempo incluso se mantiene por debajo de la versión 3 en todo momento. Seguidamente estudiaremos los resultados obtenidos en los contadores hardware mediante PAPI, para averiguar donde se han producido las mejoras que han hecho aumentar el rendimiento de la aplicación base.

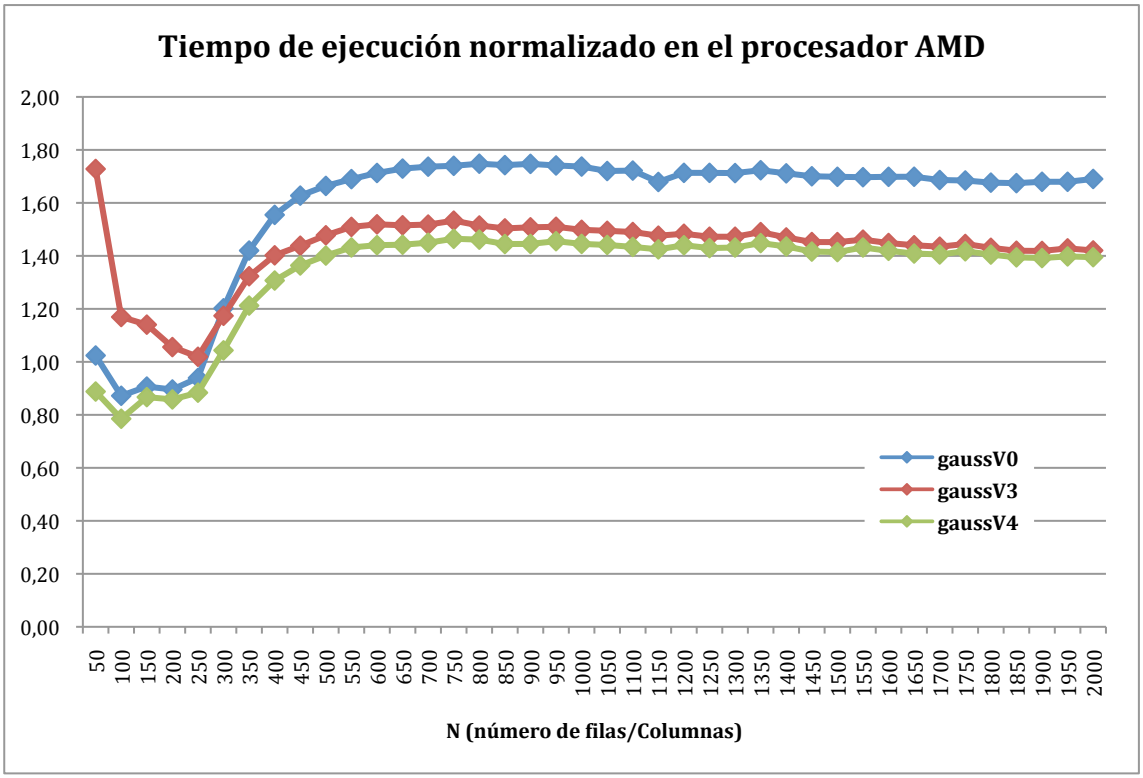

Figura 53. Tiempo de ejecución normalizado de la versión base del algoritmo serie de las diferentes optimizaciones realizadas (AMD Dual Core).

En la figura 54 podemos observar como las diferentes versiones siguen el mismo patrón que la versión base y como consecuencia podemos diferenciar las 3 fases que detallamos en el apartado anterior. Una primera zona, de N=50 hasta N=150, donde la matriz se consigue almacenar en la L1, una segunda zona, de N=150 hasta N=300, en la que la matriz es demasiado grande para ser almacenada en la L1 provocando un aumento sustancial de los fallos de caché en este primer nivel pero no tan grande como para no caber en la L2 y una tercera zona, a partir de N=300, en la cuál la matriz, no puede almacenarse en la L2 y se produce un rápido incremento de fallos en la caché de segundo nivel, L2 que merma en gran parte el rendimiento de la aplicación. Esta información la podemos contrastar en las figuras 55 y 56, donde aparecen las gráficas de los fallos en los dos niveles de memoria cache.

Observamos en la figura 55, la gráfica de fallos en el primer nivel de la memoria cache (L1), prácticamente las tres versiones tienen el mismo porcentaje de fallos. También en la figura 56, la gráfica de fallos en el segundo nivel de memoria cache (L2), que la versión 3 posee un porcentaje de fallos mayor que la versión base y la versión 4, a partir de N=300. Observamos también que la versión 4 reduce ligeramente el porcentaje tanto respecto a la versión base como a la versión 3. Como podemos ver en la figura 54, la gráfica de ciclos de reloj, las optimizaciones realizadas en la versiones 3 y 4 reduciendo el número de instrucciones ejecutadas provocan que el número de ciclos ejecutados disminuya considerablemente para las dos versiones. Estas optimizaciones junto con la reducción del porcentaje de fallos en la memoria cache de segundo nivel hacen que la versión 4 logre ser la que obtiene el mejor rendimiento de todas.

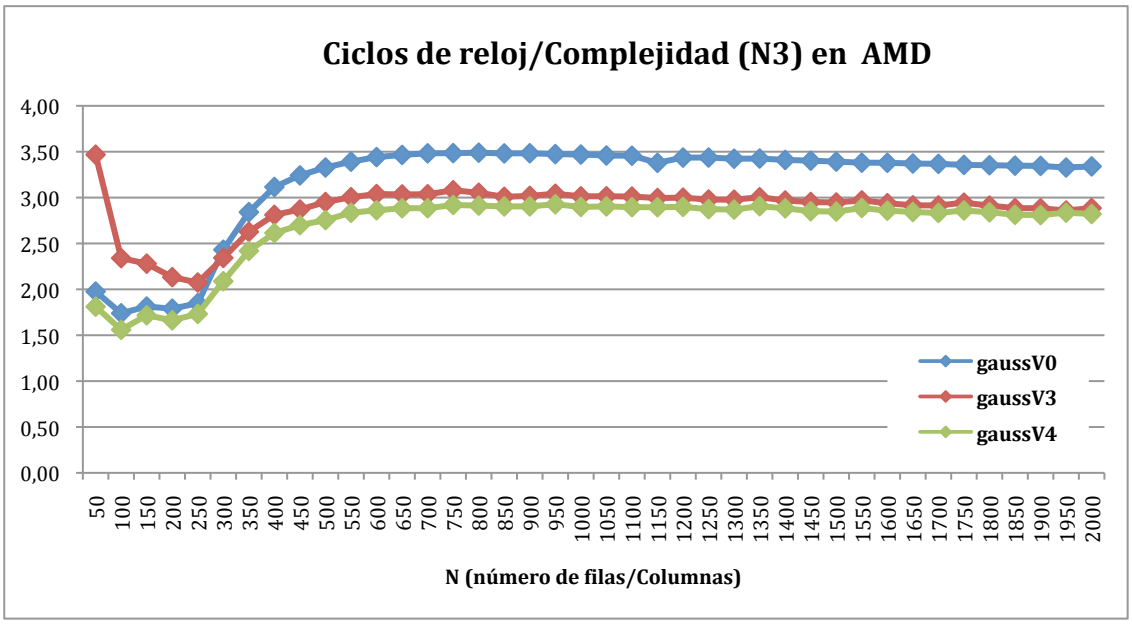

Figura 54. Ciclos de reloj de las diferentes optimizaciones realizadas (AMD Dual Core).

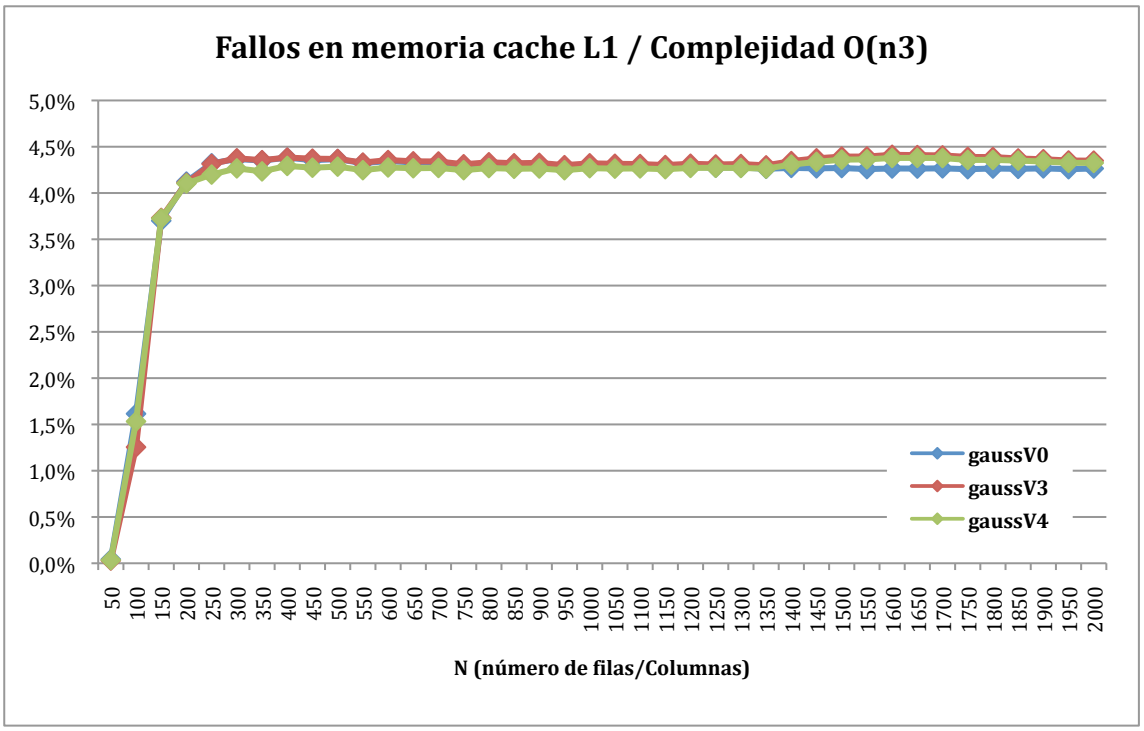

Figura 55. Fallos en la memoria cache de primer nivel (L1) de las diferentes optimizaciones realizadas (AMD Dual Core).

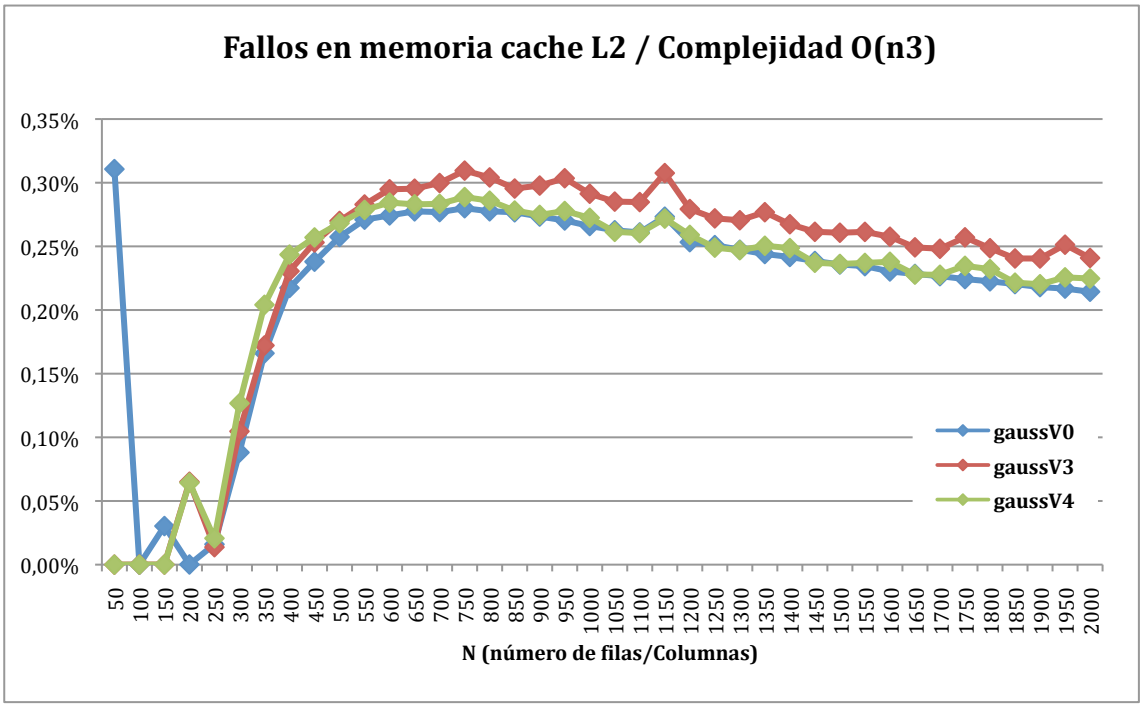

Figura 56. Fallos en la memoria cache de primer nivel (L1) de las diferentes optimizaciones realizadas (AMD Dual Core).

Una vez mostrados los resultados para el procesador AMD Athlon procederemos a estudiar los resultados para el procesadores Intel Core2 Quad. En la figura 57 observamos el tiempo de ejecución en nanosegundos divididos por la complejidad del algoritmo de la Eliminación Gaussiana para el procesador Intel Core2 Quad. Como vimos en el punto anterior en la figura 54, tanto las medidas en nanosegundos o ciclos de reloj/complejidad son equivalentes, por tanto el estudio del rendimiento de las aplicaciones en el procesador Intel Core2 Quad se realizará a partir de la curva del tiempo de ejecución.

En la figura 57 se muestra la gráfica del tiempo de ejecución normalizado de la versión base del algoritmo de la eliminación Gaussiana con pivotaje parcial por filas ejecutado en el procesador Intel Core2 Quad. Observamos que en este caso, el procesador Intel mejora sustancialmente los resultados obtenidos por el procesador AMD. Mientras que los tiempo del AMD se movían entre 0,8 ns y 1,7 ns, en el procesador Intel los valores se mueven entre 0,7 ns y 1,2 ns. Esta mejora en los tiempos de ejecución se debe a que el núcleo del procesador Intel posee más una capacidad de cómputo mayor que el del AMD debido a que tiene una frecuencia de reloj más alta. En la figura podemos identificar tres zonas bien diferenciadas.

La primera zona se esta comprendida entre N=50 y N=100, donde la matriz de datos cabe totalmente en la memoria cache de primer nivel (L1). La segunda zona se extiende entre N=150 y N=600. En esta zona la matriz no cabe en la memoria L1 pero si en la memoria cache de segundo nivel (L2). Podemos apreciar que esta segunda zona es más grande que la obtenida con el procesador AMD. Esto se debe a que el procesador Intel posee un tamaño doce veces más grande de memoria cache L2 que el AMD, 6MB. A partir de N=650 nos encontramos en la tercera zona. En esta última zona se aprecia un aumento del tiempo de ejecución sustancial. Este aumento se debe a que empieza a fallar la memoria cache L2 al no caber totalmente la matriz en esa memoria y los datos deben ir a buscarse a la memoria principal.

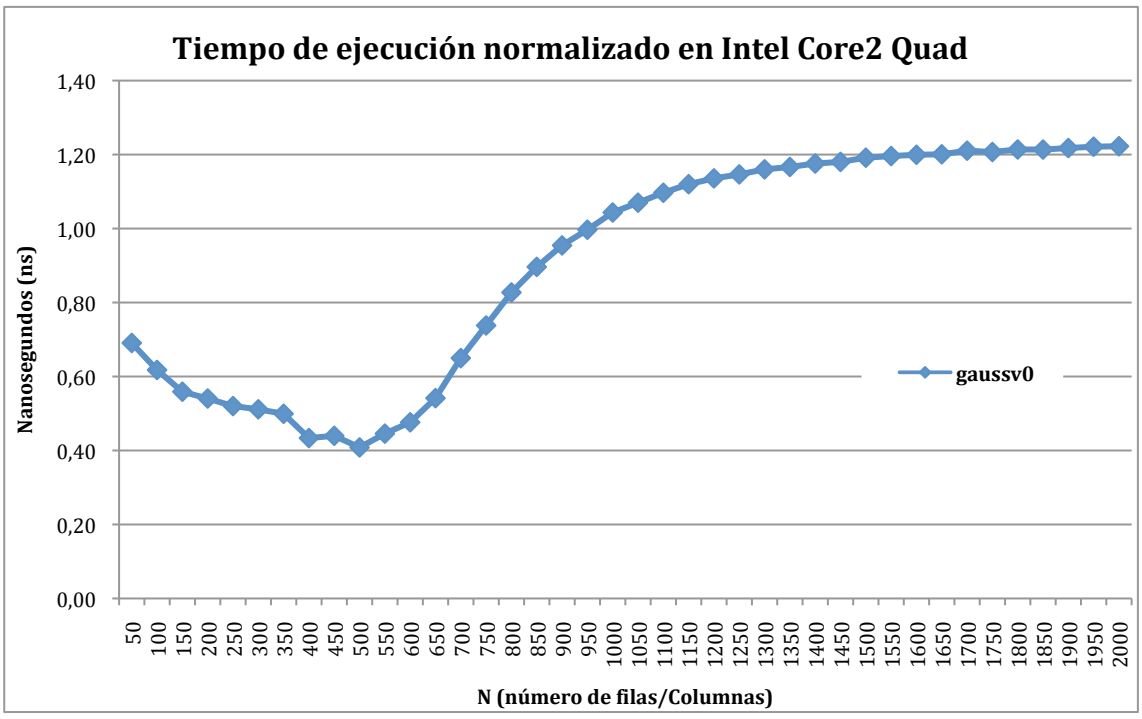

Figura 57. Tiempo de ejecución normalizado de la versión base del algoritmo serie (Intel Core2 Quad).

En la figura 58, se muestra el tiempo de ejecución normalizado del algoritmo de Eliminación Gaussiana con pivotaje parcial por filas ejecutado en el procesador Intel Core2 Quad de las diferentes versiones desarrolladas a partir de las optimizaciones realizadas. Comprobamos que las versiones 3 y 4 obtienen un rendimiento mayor que el resto de versiones para pequeños tamaños del problema. Para grandes tamaños todas las versiones obtienen resultados similares que rondan los 1,2 ns.

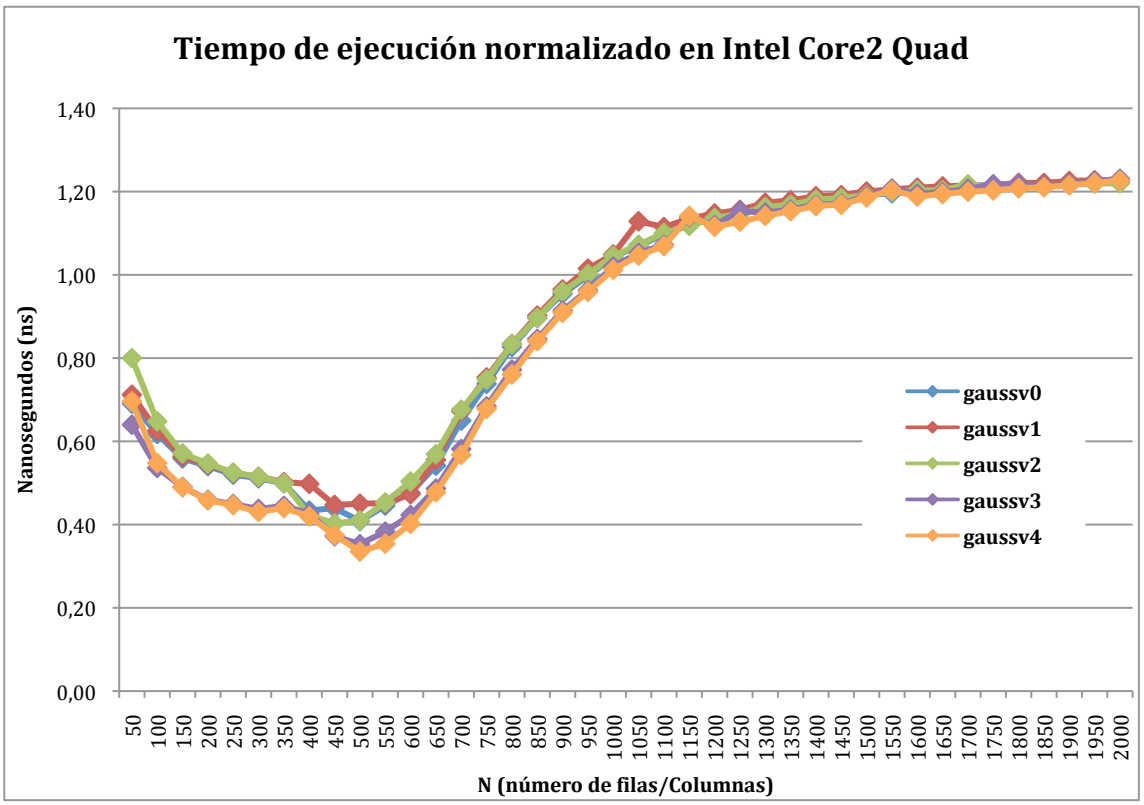

Figura 58. Tiempo de ejecución normalizado del algoritmo serie de las diferentes optimizaciones realizadas (Intel Core2 Quad).

A continuación mostraremos los resultados obtenidos para el procesador SUN UltraSPARC T2. En la figura 59 se muestra el tiempo de ejecución normalizado de la versión base del algoritmo de Eliminación Gaussiana con pivotaje parcial por filas ejecutado en el este procesador.

Podemos ver que esta curva es ligeramente diferente a la obtenida con el procesador AMD. Notamos que el tiempo de ejecución es más grande para los mismos valores de N, mientras que para el procesador AMD los tiempos se encontraban entre 0.8 y 2 ns, para el procesador UltraSPARC los tiempos de ejecución se mueven entre 8 y 10. Esto se debe a que la capacidad de cómputo del procesador de los núcleos de UltraSPARC es mucho menor que la de los núcleos del AMD. El procesador UltraSPARC dispone de una planificación en orden, la capacidad de ejecutar hasta dos instrucciones por ciclo y una frecuencia de reloj de 12GHz.

De igual modo que en la gráfica del AMD identificamos 3 zonas. La primera zona que se extiende entre N=50 y N=100 donde la matriz cabe en la memoria cache de primer nivel (L1). Una segunda zona entre N=100 y N=550 donde la memoria ya no cabe en la memoria de primer nivel pero si en la de segundo (L2). A partir de N=550 nos encontramos en la tercera zona, donde la matriz ya se puede almacenar por completo en la memoria cache de segundo nivel. Observamos que en este procesador la segunda zona se alarga más que en la del procesador AMD debido en gran parte a que el procesador UltraSPARC posee 8 veces más memoria cache de segundo nivel que el procesador AMD. Pero de igual forma que en el procesador anterior, el aumento sustancial del tiempo de ejecución es debido al aumento de fallos en la memoria de segundo nivel provocado por el aumento del tamaño del problema y por consiguiente del conjunto de datos que experimentan localidad temporal.

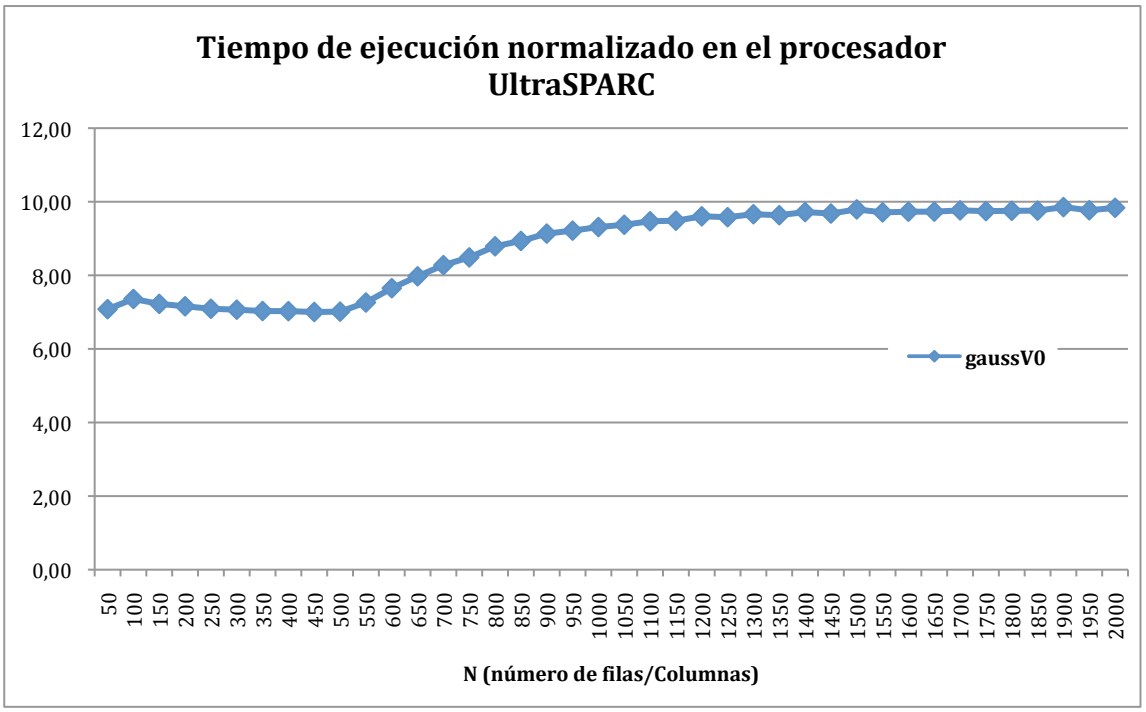

Figura 59. Tiempo de ejecución normalizado de la versión base del algoritmo serie (SUN UltraSPARC T2).

Observamos en la figura 60, que los resultados obtenidos en el procesador UltraSPARC para las versiones 3 y 4 son prácticamente idénticos y mejoran sustancialmente respecto al resto de versiones para grandes tamaños del problema mientras que para pequeños tamaños todas las versiones obtienen prácticamente el mismo rendimiento.

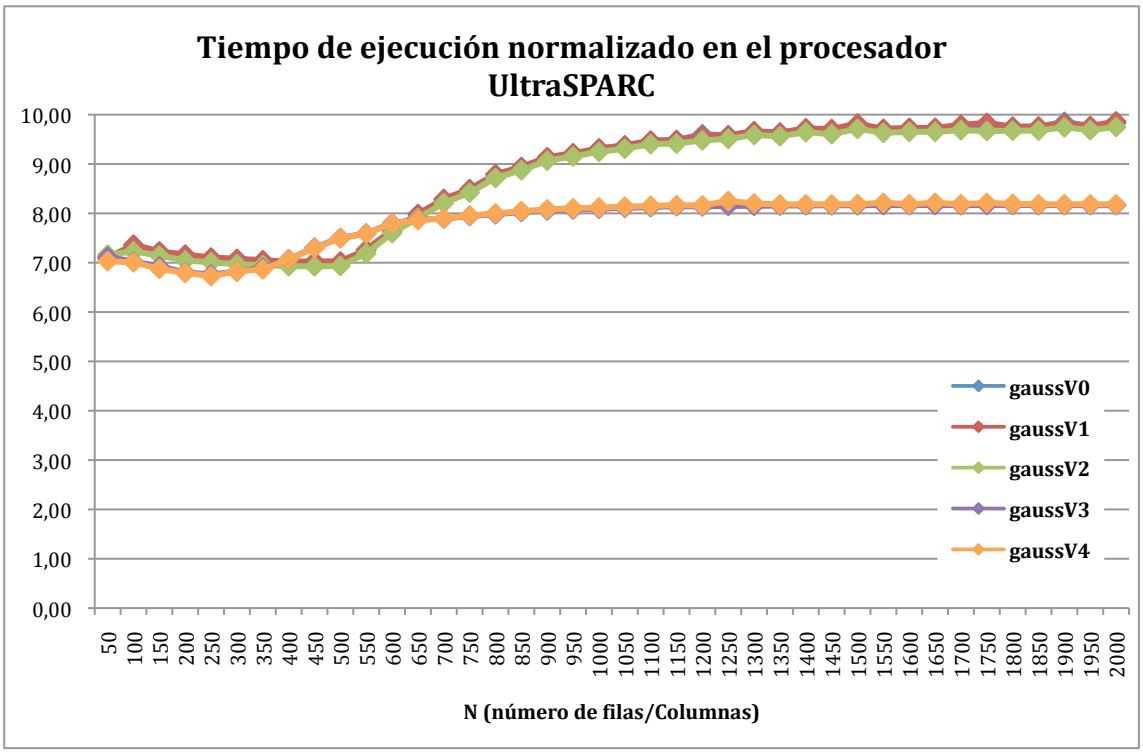

Figura 60. Tiempo de ejecución normalizado del algoritmo serie de las diferentes optimizaciones realizadas (SUN UltraSPARC T2).

4.5.5 Resumen

En resumen, en este capítulo se ha realizado el diseño del modelo serie de la aplicación del algoritmo de la Eliminación Gaussiana con pivotaje parcial por filas. Tras el análisis computacional del algoritmo y la codificación de la versión base. Hemos realizado la propuesta de varias optimizaciones que han derivado en un total de 4 versiones de la aplicación.

Estas versiones han sido ejecutadas en varios procesadores. De estos resultados se ha evaluado la repercusión de las diferentes optimizaciones para valorar si realmente han sido efectivas. De las ejecuciones se han extraído los resultados tanto de los tiempos de ejecución como de los contadores hardware para poder seleccionar la versión con mejor rendimiento como base para que en el próximo se realice la versión paralela del algoritmo.

Una vez realizado el estudio en profundidad de los resultados, hemos observados que las características de cada procesador condicionan el rendimiento de las aplicaciones y el problema que reduce el rendimiento de la aplicación se encuentra en el elevado porcentaje de fallos en la memoria cache de segundo nivel (L2) provocado por el aumento del tamaño del problema y por consiguiente del conjunto de datos que experimentan localidad temporal. Las versiones con mayor rendimiento en todos los procesadores son las versiones 3 y 4, y por tanto serán las escogidas para realizar la versión paralela en el próximo capítulo.

5 Diseño y análisis del algoritmo paralelo

En este capítulo se realizará el estudio del algoritmo paralelo de la Eliminación Gaussiana con pivotaje parcial por filas. En el apartado 5.1 se efectuará el diseño del algoritmo paralelo a partir de la versión serie estudiada en el capítulo anterior. En el apartado 5.2 se realizará un análisis computacional del algoritmo paralelo independiente del procesador. Este estudio nos permitirá averiguar los factores que perjudican el rendimiento de la ejecución paralela de la aplicación. En el apartado 5.3 se mostrarán los resultados obtenidos tras la ejecución de la aplicación paralela en los diferentes sistemas para extraer conclusiones.

5.1 Diseño del algoritmo paralelo

En este apartado se estudiará la paralelización del algoritmo de la Eliminación Gaussiana con pivotaje por filas. Para crear un aplicación paralela logrando un buen rendimiento debemos tener en cuenta evitar las dependencias de datos, reducir las sincronizaciones y las regiones críticas, gestionar apropiadamente el balanceo de carga, así como reducir el volumen de comunicaciones entre los diferentes hilos de ejecución.

Como pudimos observar en el apartado 4, la optimización que obtenía mejores resultados era las versión gaussv4. Tras el estudio de los cambios a realizar para paralelizar este caso, se descarta la optimización gaussv4 ya que al calcular el máximo simultáneamente al actualizar las filas de dos en dos, se crea una gran dependencia de datos y un alto volumen de comunicaciones entre los diferentes hilos de ejecución que afecta al rendimiento paralelo de éste y dificulta en gran parte su paralelización. Se opta por paralelizar la versión anterior gaussv3, ya que, como también observábamos en el apartado 4, las mejoras introducidas en la versión 4 no eran tan significativas respecto a la versión 3. Por este motivo partiremos del algoritmo gaussv3 para diseñar la versión paralela.

5.1.1 División de algoritmo en tareas independientes y análisis de dependencias de datos

El primer paso para paralelizar una aplicación, consiste en dividir la en subtareas completamente independientes que puedan ser ejecutadas simultáneamente en diferentes núcleos de procesamiento. A continuación mostramos en la figura 61 el esquema serie del código base a partir del cuál generaremos la versión paralela. La aplicación se divide en cuatro grandes bloques independientes: La **búsqueda del pivote**, el **intercambio de filas**, la **actualización de la fila impar** y por último, el bucle principal, la **actualización del resto de las filas** con el pivote escogido.

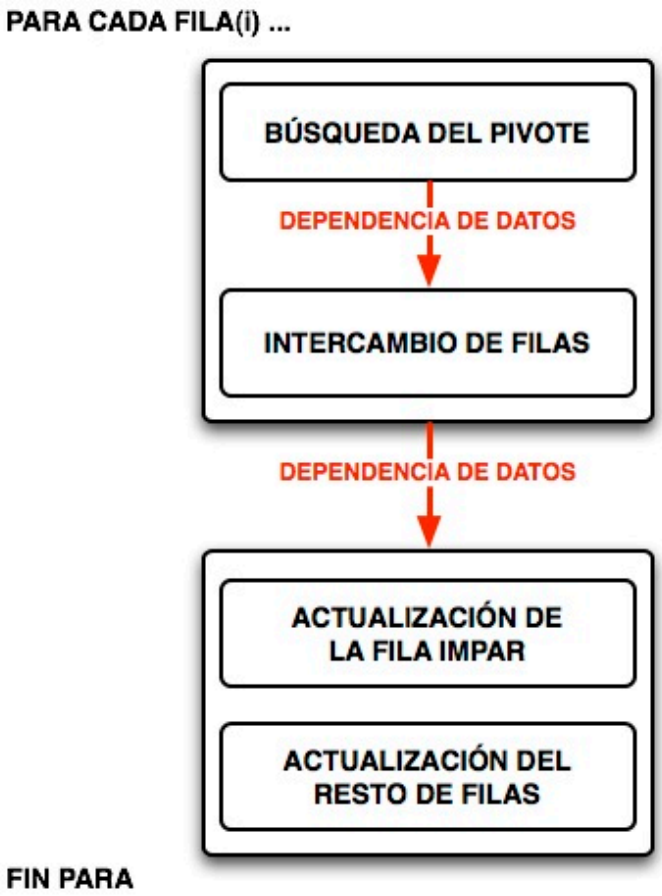

Figura 61. Representación simplificada del algoritmo de Eliminación Gaussiana con pivotaje por filas gaussv3.

- **La búsqueda del pivote** realiza una búsqueda secuencial de los elementos de la columna "*i*" con el que se actualizarán posteriormente las filas.

- **El intercambio de filas** reorganiza las filas para triangular la matriz según el pivote escogido en el bloque anterior.

- **La actualización de la fila impar** actualiza en las iteraciones en las que el número de filas es impar, esto se produce debido a que el conjunto de filas a actualizar se reduce en una fila en cada iteración y al actualizarse dos filas simultáneamente en el bucle principal se debe actualizar la fila impar por separado con el pivote escogido en el primer bloque.

- **La actualización del resto de las filas** con el pivote escogido, de la misma manera que se actualiza la fila impar, es en este bloque donde se concentra la mayor parte del cómputo y por tanto la zona más interesante a paralelizar ya que al ser la zona con más peso en todo el cómputo total del programa se obtendrían mayores beneficios.

Como se observa en la figura 61, las flechas rojas indican que los cuatro bloques no son del todo independientes entre si. El cálculo del pivote y el intercambio de filas están fuertemente ligados y no se pueden realizar de forma simultánea. Para la actualización de las filas es necesario conocer la fila pivote y haber realizado el intercambio de filas. Sin embargo una vez conocido el pivote, la actualización de cada una de las filas siguientes es una tarea repetitiva que pueden realizarse de forma simultánea para obtener un mayor rendimiento de la aplicación. En el apartado siguiente estudiaremos la dependencia de datos que existe en estos dos bloques para valorar su posible paralelización.

A la hora de ejecutar esta aplicación de forma paralela encontramos una fuerte dependencia de datos. Como se observa en la figura 62, para calcular el pivote de la fila "*i*+1" y poder actualizar con éste el resto de filas, se necesitan los valores obtenidos al calcular la actualización de la fila anterior "*i*". Para el cálculo del primer pivote se accede a la primera columna de la matriz y con el nuevo pivote se actualizan todas las filas por debajo del pivote, a partir de la primera iteración, para el cálculo del siguiente pivote, se necesitan los valores actualizados de la iteración anterior, esto dificulta la paralelización del cálculo del pivote para cada fila y su posterior actualización ya que es necesario una sincronización de todos los procesos después de actualizar las filas.

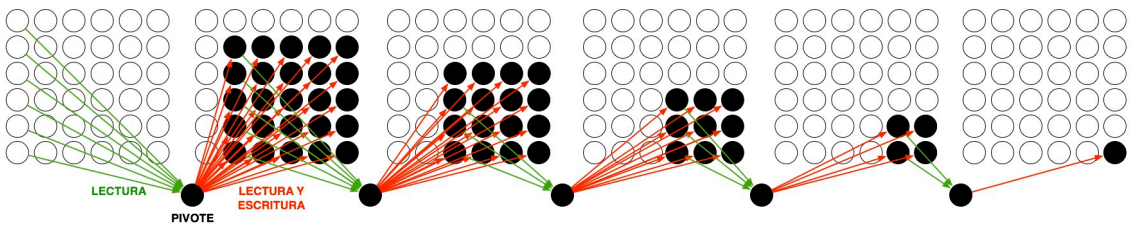

Figura 62. Diagrama de dependencias del algoritmo de la Eliminación Gaussiana con pivotaje por filas.

5.1.2 Diseño del algoritmo paralelo

La búsqueda del pivote se realiza una vez en cada iteración y realizarla de forma paralela provoca una sobrecarga/*"overhead"* en las comunicaciones sin obtener una mejora significativa. Por lo que la búsqueda del pivote y el intercambio de filas se realizan en serie por el hilo maestro (*master*). Todos los hilos deben esperar hasta que el master escoge el pivote para leer simultáneamente el valor y actualizar las filas con éste, deben sincronizarse.

En cada iteración el conjunto de filas a actualizar se reduce en una fila, al actualizarlas de dos en dos, en las iteraciones con un conjunto de filas impar se debe actualizar ésta por separado. Esta tarea la realiza un hilo cualquiera sin necesidad que el resto espere, y mientras el resto de hilos ejecutan, del conjunto de filas pares, su parte actualizando los valores. Una vez todos han acabado de actualizar sus filas, el master, prosigue la ejecución serie buscando el siguiente pivote.

A continuación, en la figura 63, mostramos de forma gráfica el diseño del algoritmo paralelo para la Eliminación Gaussiana por filas generado tras el estudio de la aplicación serie e inconvenientes encontrados durante la paralelización.

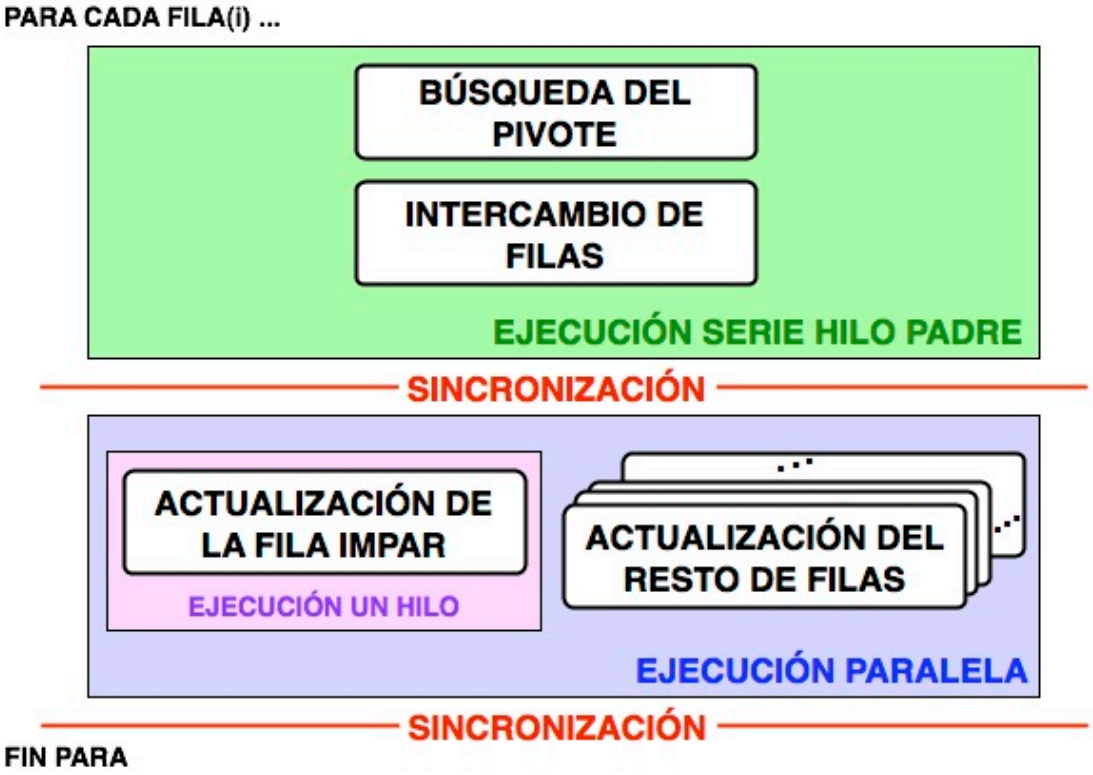

Figura 63. Esquema de la paralelización del algoritmo de Eliminación Gaussiana con pivotaje por filas.

Una vez diseñado el algoritmo se procederá a implementar la aplicación paralela. Para realizar la aplicación paralela se tomará como base el código de la aplicación paralela *gaussV3*. Para describir el paralelismo a nivel de hilo, se añadirán al código las directivas OpenMP. Las directivas empleadas son las siguientes:

- **Parallel** : Determina la sección de código que podrán ejecutar varios hilos.
- **Master**: Indicando las secciones de código que ejecutará el hilo maestro, en este caso el cálculo del pivote y el intercambio de filas en cada iteración.
- **Single nowait**: Indicando la sección de código que ejecutará un solo hilo sin que el resto de hilos deban permanecer a la espera, en este caso hablamos del cálculo de la actualización de la fila impar.
- **Barrier**: Para sincronizar todos los hilos de ejecución hasta que el hilo maestro calcule el pivote y realice el intercambio de filas.
- **For schedule(static):** Indicando que el bucle lo realizarán varios hilos repartiéndose de forma equitativa las iteraciones entre ellos, en este caso se trata de la actualización de las filas, donde cada iteración es una fila. Estas filas se repartirán de manera equitativa entre los diferente hilos de ejecución.

A continuación se muestra, en la figura 64, la representación esquematizada con las directivas OpenMP del diseño de la aplicación paralela.

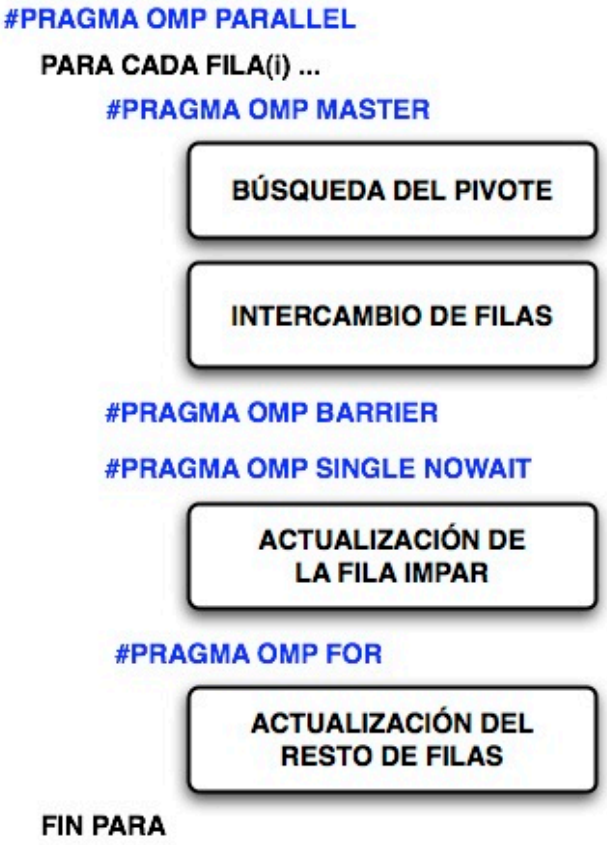

Figura 64. Representación simplificada del algoritmo de Eliminación Gaussiana con pivotaje por filas paralelo con directivas OpenMP.

5.2 Análisis computacional del algoritmo paralelo

Una vez realizado el diseño del apartado anterior, en este apartado se realizará el análisis computacional del algoritmo paralelo. Para realizar dicho análisis estudiaremos diferentes aspectos de la paralelización; que corresponden a las siguientes preguntas.

1. ¿Cómo se reparte el cómputo entre los procesadores?
2. ¿Qué cantidad de memoria necesita leer/escribir cada procesador?
3. ¿Cuánto aumenta el total de memoria utilizada por el programa?
4. ¿Qué cantidad de sincronizaciones se generan?
5. ¿Qué cantidad de comunicaciones se generan entre procesadores?

5.2.1 Reparto del cómputo entre los procesadores (computation load balance)

El volumen de cómputo en el algoritmo de la Eliminación Gaussiana con pivotaje por filas está desbalanceado. El master es quien se ocupa de realizar la parte serie del algoritmo. Una vez finaliza con estas tareas todos los hilos de ejecución se reparten equitativamente las filas a actualizar y en el caso que el conjunto de filas sea divisible entre el total de hilo de ejecución, un hilo asume la carga extra de actualizar las filas sobrantes. Por este motivo el desbalance es directamente proporcional al número de hilos de ejecución. Cabe destacar como se muestra en la figura 65, la parte secuencial es de menor orden que la parte paralela y por tanto en proporción es menor cuanto mayor es N.

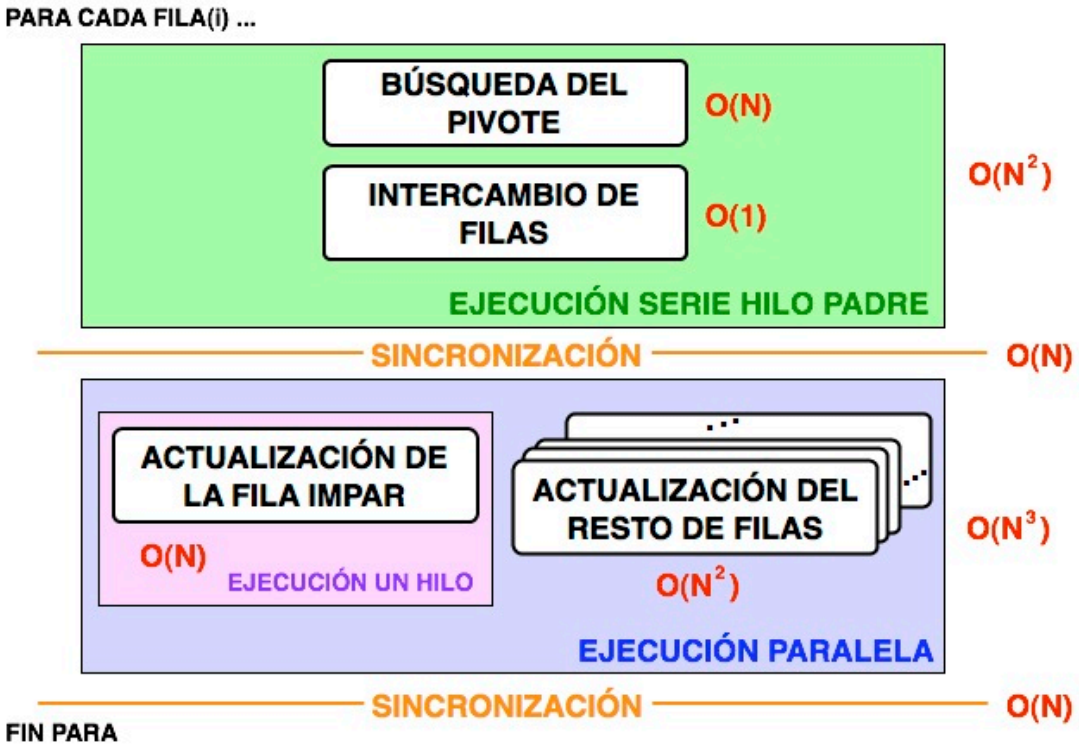

Figura 65. Esquema de la paralelización del algoritmo de Eliminación Gaussiana con pivotaje por filas.

5.2.2 Cantidad de memoria leída/escrita por cada procesador

Durante la ejecución paralela, los hilos modifican los valores de las variables. Ese valor lógico puede encontrarse en un registro físico de uno de los núcleos o en una caché privada. Si otro de los hilos necesita leer esos valores, el sistema debe asegurar que el valor leído esté actualizado correctamente. Por este motivo puede ser necesario mover los datos entre los núcleos y las memorias cache de éstos. Este movimiento de datos supone realizar comunicaciones y no es despreciable ya que suponen un importante "*overhead*". La cantidad de memoria leída/escrita por los hilos de ejecución es la misma cantidad que la versión serie del algoritmo repartida entre los diferentes hilos añadiendo la lectura de las variables añadidas para la sincronización de OpenMP.

5.2.3 Aumento del total de memoria utilizada por el programa

Los requerimientos de memoria crecen de forma lineal respecto a N el número de filas/columnas del problema y al número de hilos de ejecución en el algoritmo serie. No hay aumento de memoria del algoritmo paralelo frente a la versión serie. Existe pequeño un aumento de las variables. Cada hilo necesita total de 6 variables de tipo double (T, T1, T2, ptr, ptr2, ptr3) y dos de tipo entero (j, k), para realizar la actualización de las filas que se le han repartido. Al ser un valor tan pequeño, las variables necesarias se asignan a registros de cada núcleo de procesamiento.

5.2.4 Cantidad de sincronizaciones

El diagrama de flujo de la aplicación paralela resultante se muestra en la figura 66, donde podemos observar que para resolver un sistema de N variables, se realizan 2N sincronizaciones. Por cada fila de la matriz los hilos deben sincronizarse para obtener la fila pivote y así poder actualizar las filas que se le han asignado. Deben volver a sincronizarse tras actualizar todas sus filas para proseguir la ejecución, ya que el master debe leer los nuevos valores actualizados para el cálculo del siguiente pivote.

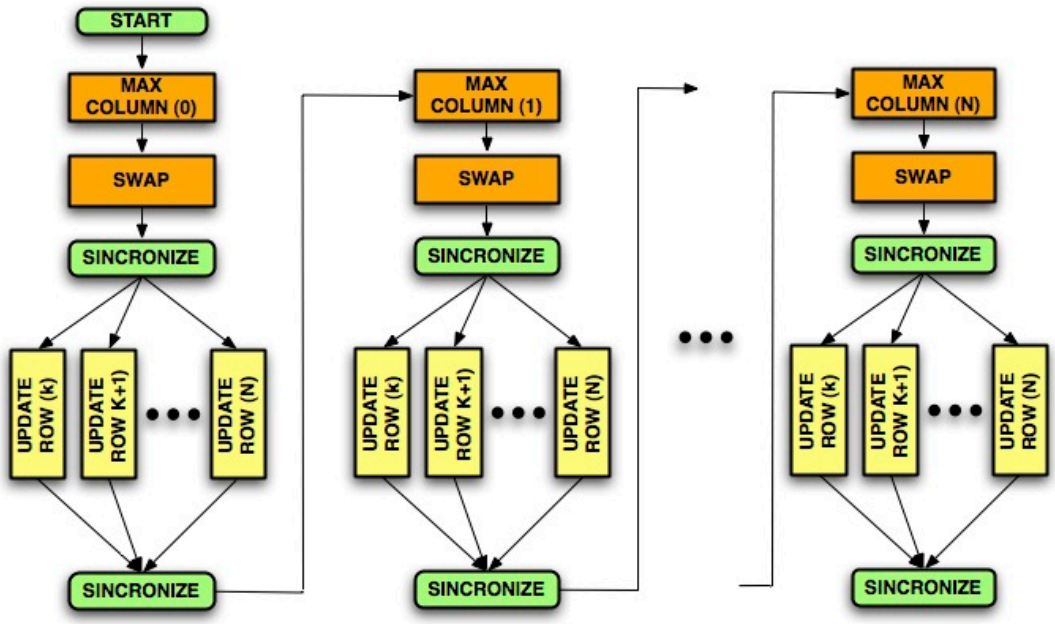

Figura 66. Diagrama de flujo del algoritmo paralelo de la Eliminación Gaussiana que muestra las sincronizaciones necesarias.

5.2.5 Cantidad de comunicaciones (movimiento de datos) entre procesadores

Las comunicaciones que se generan entre los procesadores son muy complejas. El hilo maestro en cada iteración es quien calcula el pivote. Una vez ha hecho el intercambio de filas, todos los hilos de ejecución que permanecían esperando recuperan el valor del pivote y con éste actualizan las filas que le corresponden. Una vez finalizada la primera iteración el hilo maestro debe leer los valores de las nuevas filas generadas y por tanto necesita acceder a los valores que se encuentran en las memorias caches de los diferentes hilos. El algoritmo en cada iteración reduce en una unidad las filas y columnas a computar. De esta manera en cada iteración el hilo debe acceder a las filas que han sido modificadas previamente por otros hilos. Este gran volumen de comunicaciones, que aseguran la coherencia en las memorias caches dependen totalmente del grado de paralelismo de la aplicación (número de hilos de ejecución).

En la figura 67, se muestra cómo tras realizar una escritura de un valor por un hilo de ejecución en un procesador, éste valor, debe propagarse hasta el resto de procesadores para mantener la coherencia de datos en cache.

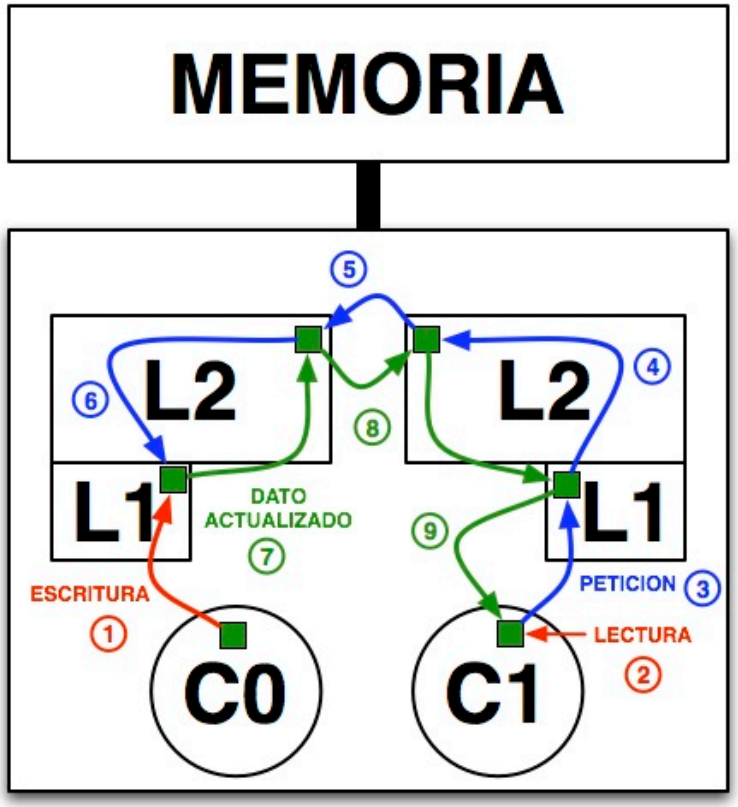

Figura 67. Propagación del valor de una variable entre dos procesadores tras una escritura para mantener la coherencia de cache.

5.3 Resultados experimentales

En este apartado se mostrarán los resultados obtenidos tras ejecutar la versión paralela del programa en los mismos procesadores y con la misma la metodología experimental explicada en el capítulo anterior. De esta forma obtendremos la gráfica de resultados del tiempo de ejecución de nuestra aplicación en los diferentes procesadores. Igual que en el capítulo anterior, las gráficas de resultados muestran el "*tiempo normalizado*", el tiempo de ejecución dividido por la complejidad del algoritmo (nanosegundos/n^3), para que sea más fácil interpretar los resultados. Los resultados se muestran en 3 apartados. En el apartado 5.3.1 se realizará el estudio del "*overhead*" que provoca en cada procesador la ejecución paralela de la aplicación. En el apartado 5.3.2 se evaluará la mejora que aporta la ejecución "*multicore"* al añadir más hilos y usar más núcleos de procesamiento. Finalmente en el apartado 5.3.3 se comparan los resultados obtenidos por los tres procesadores estudiados y se sacan conclusiones del estudio.

5.3.1 Medida del "overhead" de la ejecución paralela

El objetivo de este apartado es estudiar y cuantificar el "overhead" que provoca la paralelización de la aplicación. Para poder observar este overhead, se compararán los resultados obtenidos de la versión serie de la aplicación en el capítulo anterior con los resultados de la versión paralela con un solo hilo de ejecución. Tras comparar ambos resultados podremos averiguar cómo afecta la gestión de hilos y las sincronizaciones explicitadas en las directivas OpenMP a la aplicación, en función del tamaño del problema a resolver.

El primer procesador que estudiaremos será el **AMD Athlon Dual Core**. En la figura 68 se muestra la gráfica del tiempo de ejecución normalizado del algoritmo de Eliminación Gaussiana de las versión serie y paralela para los diferentes valores de N escogidos. Observamos que, para valores pequeños de N, el algoritmo paralelo tiene un peor rendimiento que el serie. Para pequeños valores de N, la parte secuencial tiene más peso sobre el total del tiempo de ejecución que la parte paralela a causa del desbalanceo del cómputo explicado en el apartado 5.2.1. Además aún ejecutando un solo hilo, se genera un "*overhead*" por el tratamiento y sincronización de los hilos. A medida que aumentamos el tamaño del problema, este "*overhead*" y la parte serie, dejan de tener tanto peso con respecto al tiempo dedicado a ejecutar la aplicación paralela, de manera que ambos algoritmos tardan prácticamente el mismo tiempo en ejecutarse.

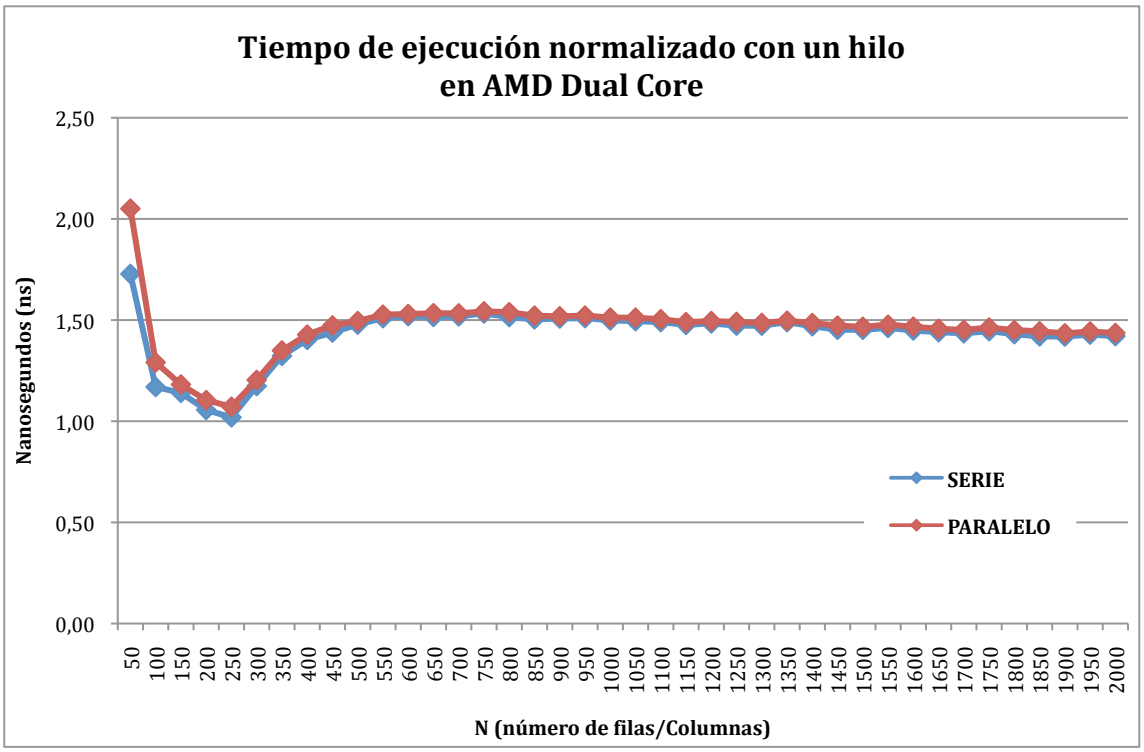

Figura 68. Comparación del tiempo de ejecución normalizado de la versión secuencial y la versión paralela ejecutadas con un hilo (AMD Athlon Dual Core).

En segundo lugar, continuaremos con el procesador **Intel Core2 Quad**. Observamos en la figura 69 la gráfica de comparación del tiempo de ejecución normalizado para los algoritmos serie y paralelo con un hilo de ejecución. De la misma forma que hemos ido observando en el anterior procesador, encontramos el "overhead" provocado por el tratamiento de los hilos y las sincronizaciones definidas en el algoritmo paralelo. Este "overhead" para valores pequeños de N tiene mucho peso en el cómputo total del tiempo de ejecución y provoca que el algoritmo paralelo tenga bastante menos rendimiento en estos casos. Conforme vamos elevando el tamaño del problema este "overhead" deja de tener tanto peso y es entonces cuando el algoritmo paralelo llega a igualar al algoritmo serie con un solo hilo de ejecución. Observamos una pequeña anomalía a partir de N=1000 donde la aplicación paralela supera la aplicación serie con un speedup de 1,07. Esto indica claramente que la versión serie es mejorable. Una explicación plausible puede ser, que el orden en el que accede a las variables en memoria la aplicación paralela provoca un mejor rendimiento.

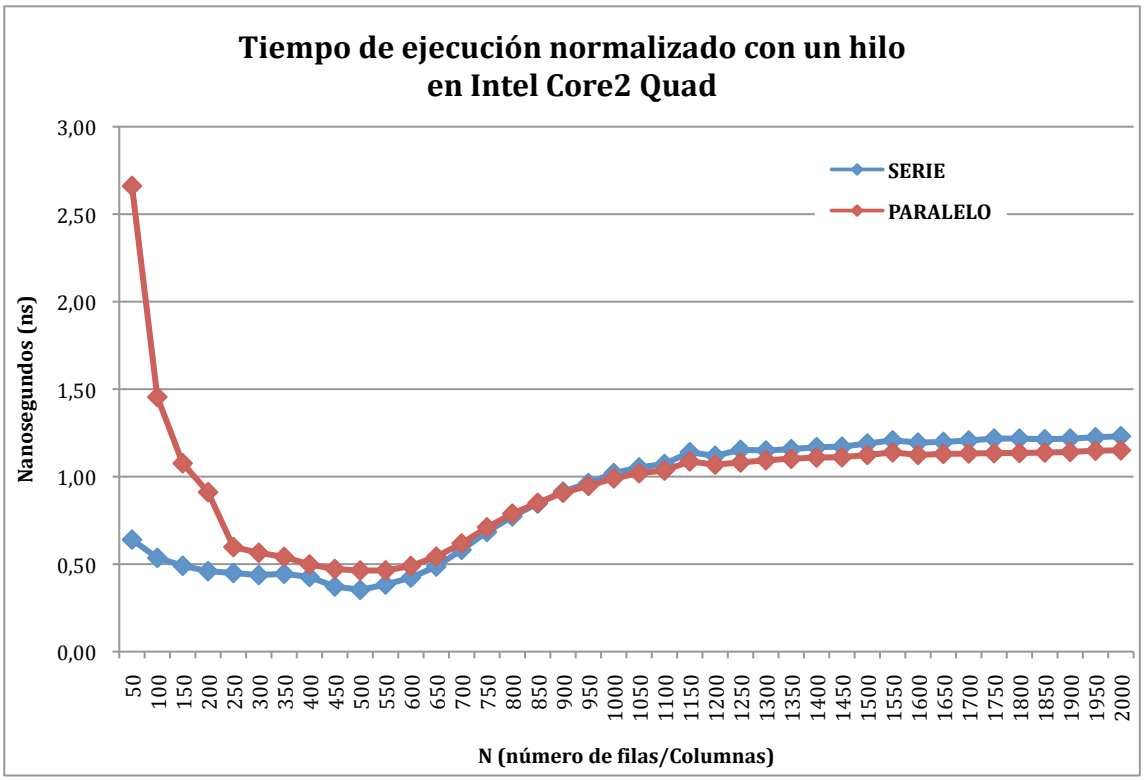

Figura 69. Comparación del tiempo de ejecución normalizado de la versión secuencial y la versión paralela con un hilo (Intel Core 2 Quad).

En tercer lugar, analizaremos el procesador **SUN UltraSPARC T2**. Observamos en la figura 70, la gráfica de comparación del tiempo de ejecución normalizado para los algoritmos serie y paralelo con un hilo de ejecución. En este caso observamos una anomalía que se produce para todos los valores de N estudiados. La ejecución paralela sufre un *"overhead"* que crece de manera constante a la complejidad del cómputo ($O(N^3)$). Como veremos más adelante este "*overhead"* no crece al añadir más hilos de ejecución. Por falta de tiempo no se ha llegado a hacer un análisis exhaustivo de esta anomalía negativa del 8%. Este estudio se deja como posible ampliación o línea futura de investigación.

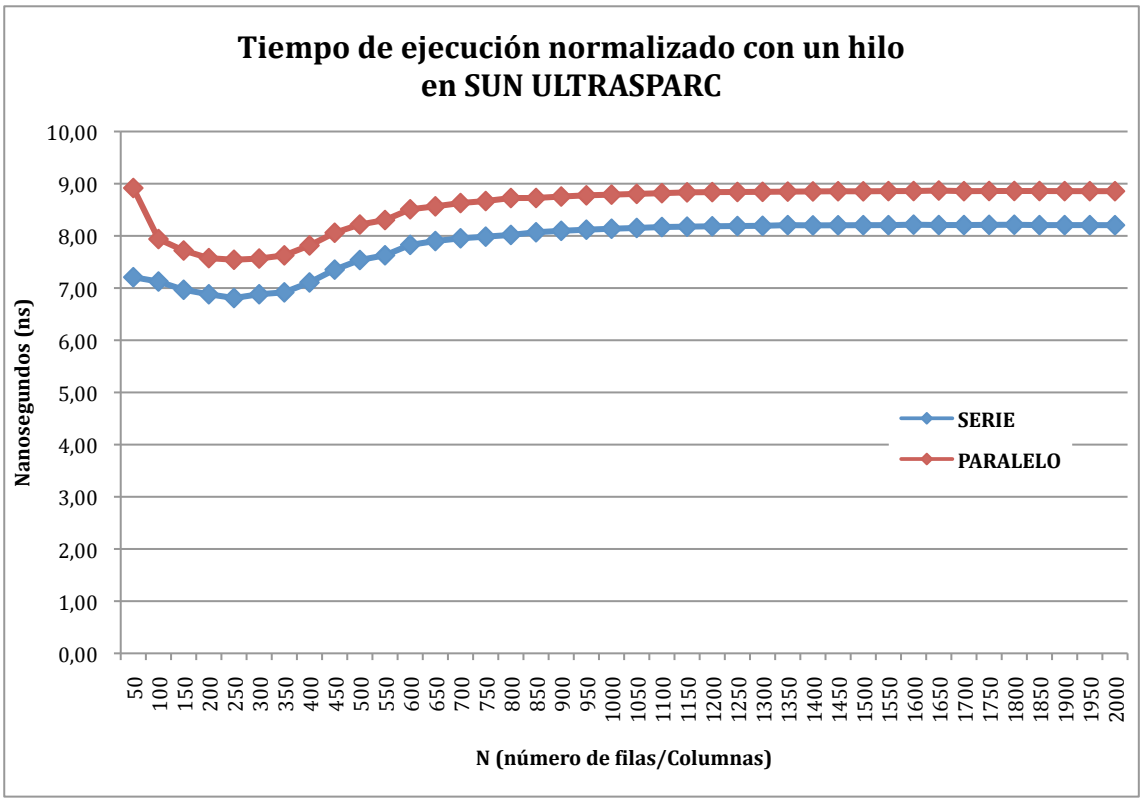

Figura 70. Comparación del tiempo de ejecución normalizado de la versión secuencial y la versión paralela con un hilo (SUN UltraSPARC T2).

Tras el análisis de los resultados en los tres procesadores, podemos concluir que exceptuando la anomalía encontrada en la ejecución del SUN UltraSPARC, la paralelización implica un "overhead" sobre el tiempo de ejecución serie provocado por la gestión, comunicación y sincronización de los hilos durante la ejecución. Este "overhead" causado por la paralelización de la aplicación es constante o de menos complejidad que el cómputo y es por este motivo, queda solapado por el cómputo a medida que aumentamos el tamaño del problema.

5.3.2 Evaluación de la ejecución Multicore.

Una vez comparados los algoritmos serie y paralelo en un núcleo de los diferentes procesadores escogidos, pasaremos a realizar un estudio de la escalabilidad del algoritmo paralelo en cada procesador. Tal y como se explicó en el capítulo anterior, los resultados se medirán en tres procesadores diferentes. Cada uno dispone de una configuración de núcleos de procesamiento diferentes y estudiaremos, como se comporta el algoritmo paralelo, dividiendo la ejecución en varios hilos para cada una de las configuraciones. El objetivo de este apartado es evaluar la mejora obtenida al usar varios núcleos de procesamiento.

A continuación, en la figura 71, se muestra la gráfica del tiempo de ejecución normalizado del algoritmo paralelo en el procesador **AMD Athlon Dual Core**. En la gráfica podemos contemplar la versión del algoritmo serie y la versión paralela de dos hilos, donde cada hilo es ejecutado en un núcleo. Podemos dividir la ejecución en tres zonas.

En la primera zona, comprendida entre N=50 y N=150, contemplamos que para pequeños valores del problema, la parte secuencial tiene un mayor peso que la parte paralela, como ya explicamos en el apartado 5.2.1. El cómputo de la aplicación está desbalanceado. Además, el "overhead" provocado por la gestión y sincronización de dos hilos es mayor. Este "overhead" es suficientemente grande para que, con un pequeño tamaño del problema, acapare un peso muy importante en el total del tiempo de ejecución de la aplicación paralela. Y por tanto empeora sustancialmente el rendimiento de ésta. Observamos también que conforme aumenta el tamaño del problema, este overhead, va perdiendo peso, hecho que provoca que mejore notablemente el rendimiento de la aplicación a partir del valor N=150.

En la segunda zona, comprendida entre N=200 y N=450, nos encontramos en la región con speedup ideal. En esta región el tamaño de los datos cabe en la memoria de segundo nivel (L2). Por tanto cada núcleo de ejecución trabaja con su memoria de primer y segundo nivel sin entorpecer al resto. Al añadir un núcleo de ejecución más se obtiene el rendimiento esperado en esta zona.

En la tercera zona, comprendida a partir de N=500, observamos un gran descenso del rendimiento de la aplicación, cayendo por debajo de nuestras expectativas el speedup de 2 a 1,2. En esta zona, el tamaño de los datos no cabe en la memoria cache de segundo nivel, por lo que los hilos deben ir a la memoria principal a buscar los datos. Una explicación plausible de este descenso del speedup es que el ancho total de banda a memoria principal no es capaz de satisfacer las necesidades de los dos hilos de ejecución cuando ambos van a buscar los datos a memoria. Por este motivo la ejecución con dos hilos no es capaz de mejorar notablemente el rendimiento de la ejecución con un hilo en esta región.

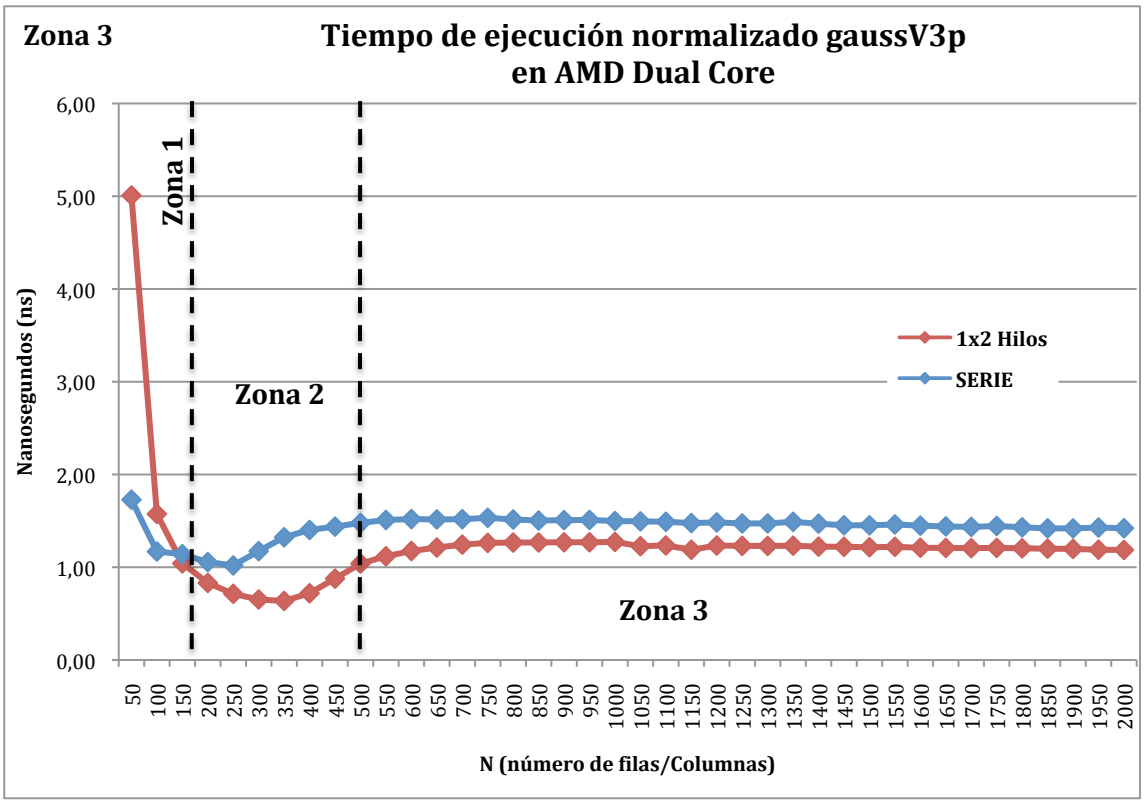

Figura 71. Comparación del tiempo de ejecución normalizado de la versión serie y la versión paralela con dos (AMD Athlon Dual Core).

En las figuras 72, se muestra la gráfica de comparación del tiempo de ejecución normalizado del algoritmo paralelo para el procesador **Intel Core2 Quad**. En esta gráfica se muestra la versión secuencial de la aplicación; la versión paralela KxC Hilos, donde C es el número de núcleos utilizados y K es el número de hilos por núcleo. De nuevo la ejecución se puede dividir en tres zonas diferenciadas:

En la primera zona, los datos caben en la memoria cache de primer nivel. Comprobamos que como ya vimos en el procesador anterior, el "overhead" provocado por el tratamiento y la sincronización de los hilos, aumenta fuertemente a medida que se añaden más hilos de ejecución. En la segunda zona, los datos caben en la memoria cache de segundo nivel. Observamos una gran mejora entre la versión de un hilo y la de dos hilos llegando el speedup a 2,6. No obstante, entre la versión de dos hilos y la versión de cuatro hilos no se aprecia una mejora sustancial (máximo speedup de 1.38) hecho que se repite entre la versión de cuatro hilos y la versión de ocho hilos (máximo speedup 1,2). Una explicación razonable seria que, al añadir dos núcleos trabajando con la L2 la aplicación consigue el speedup ideal. Al añadir cuatro núcleos que se disputan el espacio en la memoria L2 el porcentaje de fallos aumenta en proporción reduciendo el rendimiento de la aplicación. Al añadir 8 hilos de ejecución sobre 4 núcleos, los procesadores son capaces de aprovechar el tiempo de espera de los hilos para ejecutar las instrucciones del otro. De esta manera se obtiene la pequeña mejora de rendimiento con la versión de ocho hilos.

En la tercera zona, en la que los datos no caben en la memoria cache de segundo nivel, observamos que el speedup se reduce paulatinamente hasta llegar a 1,53 entre la versión de un hilo y el resto de versiones mostradas en la gráfica. Una explicación plausible de este descenso del speedup, seria que, el ancho total de banda a memoria principal no es capaz de satisfacer las necesidades de los hilos de ejecución cuando todos van a buscar los datos a memoria. Hay que sumar también el "*overhead*" generado por la sincronización y al desbalanceo del algoritmo que crece al aumentar el número de hilos. Por estos motivos la ejecución con más dos hilos no es capaz de mejorar el rendimiento de la ejecución con dos hilos en esta región.

Observamos que los núcleos de procesamiento al no tener tecnología *multithreading*, no son capaces de ejecutar simultáneamente más de un hilo y por tanto deben cambiar de contexto por cada hilo. Este cambio de contexto provoca un descenso del rendimiento directamente proporcional a los hilos que soporte cada núcleo. Por este motivo las versiones con dieciséis hilos y treinta y dos hilos obtienen un peor rendimiento que la versión de ocho hilos.

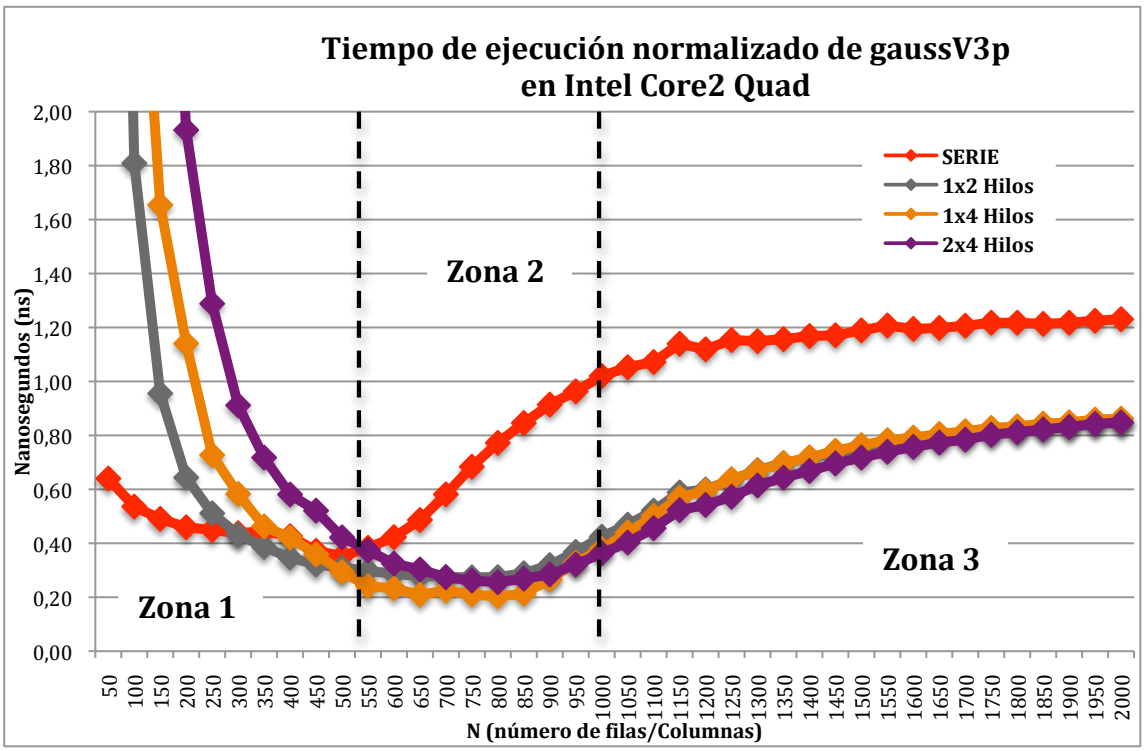

Figura 72. Comparación del tiempo de ejecución normalizado de la versión paralela (Intel Core2 Quad).

En la figura 73, tenemos la gráfica de comparación del tiempo de ejecución normalizado del algoritmo paralelo en el procesador SUN UltraSPARC T2. En la gráfica encontramos la versión paralela de la aplicación con KxC Hilos, donde C es el número de núcleos utilizados y K es el número de hilos por núcleo.

Comprobamos que el overhead, de la misma forma que en los procesadores anteriores en la zona 1, aumenta fuertemente a medida que añadimos más hilos de ejecución. Sin embargo, una vez que se desvanece el efecto del overhead sobre el cómputo, a diferencia de los sistemas anteriores, en el T2 la aplicación escala correctamente consiguiendo prácticamente el speedup ideal. Por tanto, el "*overhead*" de complejidad $O(N^3)$ que se observaba en la figura 70 con un hilo, no crece al añadir más hilos de ejecución.

En la zona 2, entre la versión de un hilo trabajando con un núcleo, y la versión de dos hilos, trabajando con dos núcleos, la aplicación consigue un speedup medio de 1,95. Entre la versión de dos hilos trabajando con dos núcleos, y la versión de cuatro hilos, trabajando con cuatro núcleos, la aplicación consigue un speedup medio de 1,94.

Entre la versión de cuatro hilos trabajando con cuatro núcleos, y la versión de ocho hilos, trabajando con cuatro núcleos (dos hilos por núcleo), la aplicación consigue un speedup medio de 1,83. Entre la versión de ocho hilos trabajando con cuatro núcleos, y la versión de dieciséis hilos, trabajando con cuatro núcleos (cuatro hilos por núcleo), la aplicación consigue un speedup medio de 1,5. Por tanto, el *multithread hardware*, mejora sustancialmente el rendimiento ya que permite que se aproveche al 100% la capacidad de cómputo de los núcleos de procesamiento.

Finalmente es entre la versión de dieciséis hilos trabajando con cuatro núcleos, y la versión de treinta y dos hilos, trabajando con cuatro núcleos (ocho hilos por núcleo), la aplicación no consigue una mejora sustancial obteniendo un speedup medio de 1,07. Vemos que en este caso, el efecto del *multithread hardware* es pequeño. Este fenómeno se puede deber a dos circunstancias. La primera es que con 8 hilos por núcleo se saturen los recursos de cómputo. La segunda posibilidad, es que con 32 hilos en ejecución solicitando datos a memoria principal se sature la memoria debido a que el acho de banda ofrecido no se capaz de abastecer todos los hilos.

La conclusión que podemos extraer es que, con dos hilos por núcleo no se aprovecha al 100% los recursos de cómputo sino que se necesitan al menos 4 hilos por núcleo. Además, la parte secuencial de la aplicación limita la mejora ya que aún añadiendo más hilos de ejecución éstos deben esperar que el hilo maestro realice los cálculos secuenciales. Es por este motivo que por más hilos que se añadan el tiempo no disminuye tan significativamente.

Si nos fijamos con más detalle, también podemos observar, que las versiones con dieciséis y treinta y dos hilos, sufren subidas puntuales del tiempo de ejecución para valores concretos de N. Estas bajadas de rendimiento para valores concretos de N, se deben posiblemente a problemas con los alineamientos de los datos en memoria.

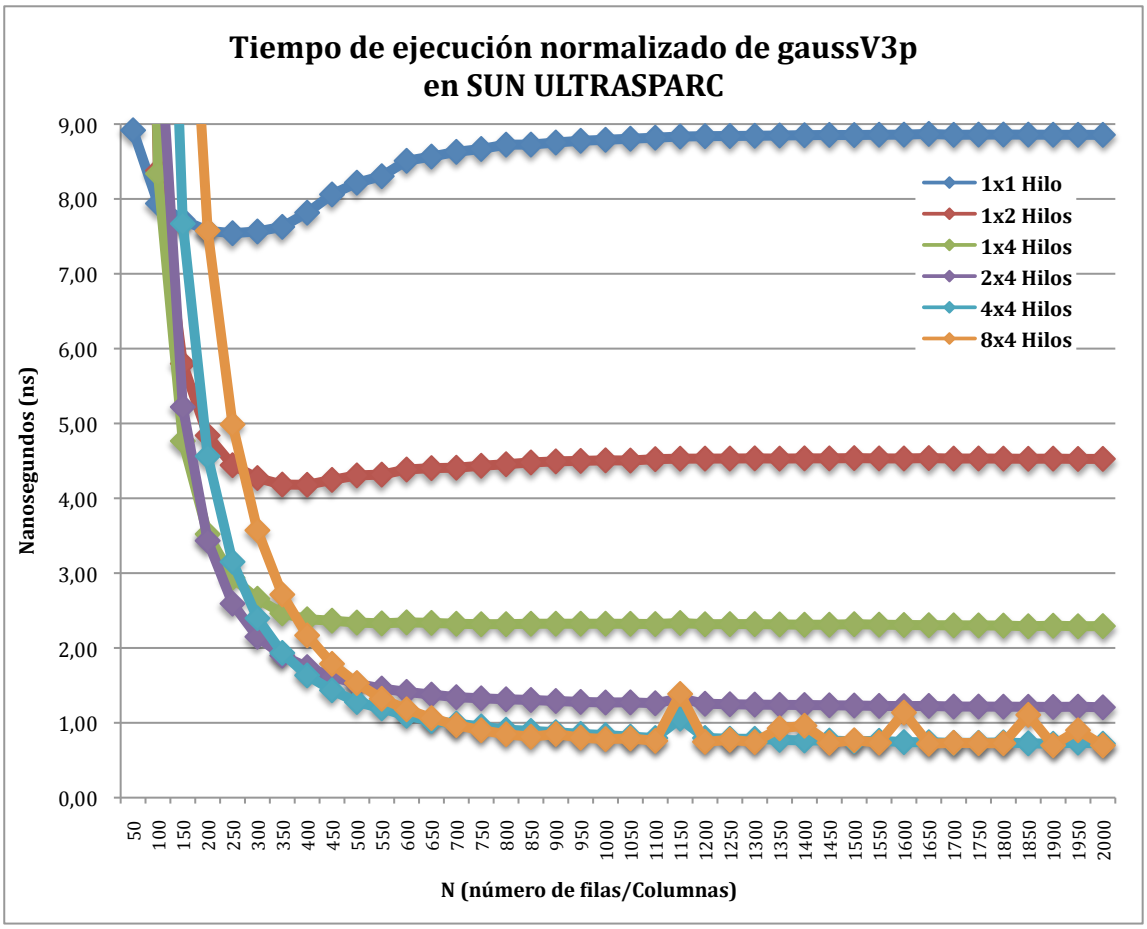

Figura 73. Comparación del tiempo de ejecución normalizado de la versión paralela (SUN UltraSPARC T2.)

Tras el análisis de los resultados en los tres procesadores, podemos concluir que la mejora de la aplicación en la ejecución "*multicore"* esta supeditada al ancho de banda a memoria que disponen los hilos de ejecución para recuperar los datos y a la capacidad de los recursos de cómputo disponibles. Para cada uno de los procesadores el límite que dificulta la escalabilidad de la aplicación añadiendo más hilos de ejecución puede variar o ser una combinación de muchas circunstancias. En los procesadores Intel y AMD, parece que el hándicap que encuentran es la saturación del ancho de banda a memoria, mientras que en al procesador SUN, una explicación plausible de el bajo speedup obtenido con la versión de 32 hilos en relación a la de 16 hilo, es la saturación de la capacidad de los recursos cómputo.

5.3.3 Comparativa entre procesadores

Una vez estudiados los resultados de la ejecución multi-núcleo en las diferentes configuraciones de procesador escogidos, pasaremos a comparar los diferentes procesadores para evaluar su idoneidad para nuestro problema. El objetivo de este apartado es evaluar la mejora obtenida al usar varios núcleos de procesamiento.

En la figura 74 se muestra la grafica comparativa del tiempo de ejecución normalizado de la aplicación paralela ejecutando un hilo sobre un núcleo de procesamiento en los tres procesadores.

Observamos, en este caso, que el mejor tiempo de ejecución lo obtiene el procesador Intel Core2 Quad, seguido de cerca por el AMD Dual Core mientras que el procesador SUN UltraSPARC T2 obtiene unos resultados mucho peores. Estos resultados son totalmente lógicos ya que como vimos en el capítulo anterior (figura 46), los núcleos del procesador Intel poseen una capacidad de cómputo mayor que los núcleos del procesador T2. El procesador Intel con la capacidad de ejecutar 3 instrucciones por ciclo, una planificación de ejecución fuera de orden y una frecuencia de 2,5GHz posee la mayor capacidad de cómputo de los tres procesadores. Le sigue de cerca el procesador AMD que tiene la capacidad de ejecutar 3 instrucciones por ciclo y una planificación de ejecución fuera de orden, como el procesador Intel pero dispone de una frecuencia menor, 2GHz. Por último el núcleo del procesador SUN es capaz de ejecutar hasta 2 instrucciones por ciclo a una frecuencia de 1,2GHz. La planificación de ejecución del SUN es fuera de orden por tanto, la ejecución se suspende hasta que se resuelven las dependencias de datos. También podemos observar la diferencia de tamaño de la memoria cache de segundo nivel (L2), comparando el tamaño de N por el cual empieza a bajar el rendimiento. Contemplamos que el procesador Intel con 6MB de memoria L2 obtiene un mayor rendimiento hasta N=600; el procesador T2 con 4MB de memoria L2 empieza a fallar con N=350 y por último el procesador AMD con sólo 512KB empieza a fallar con N=250.

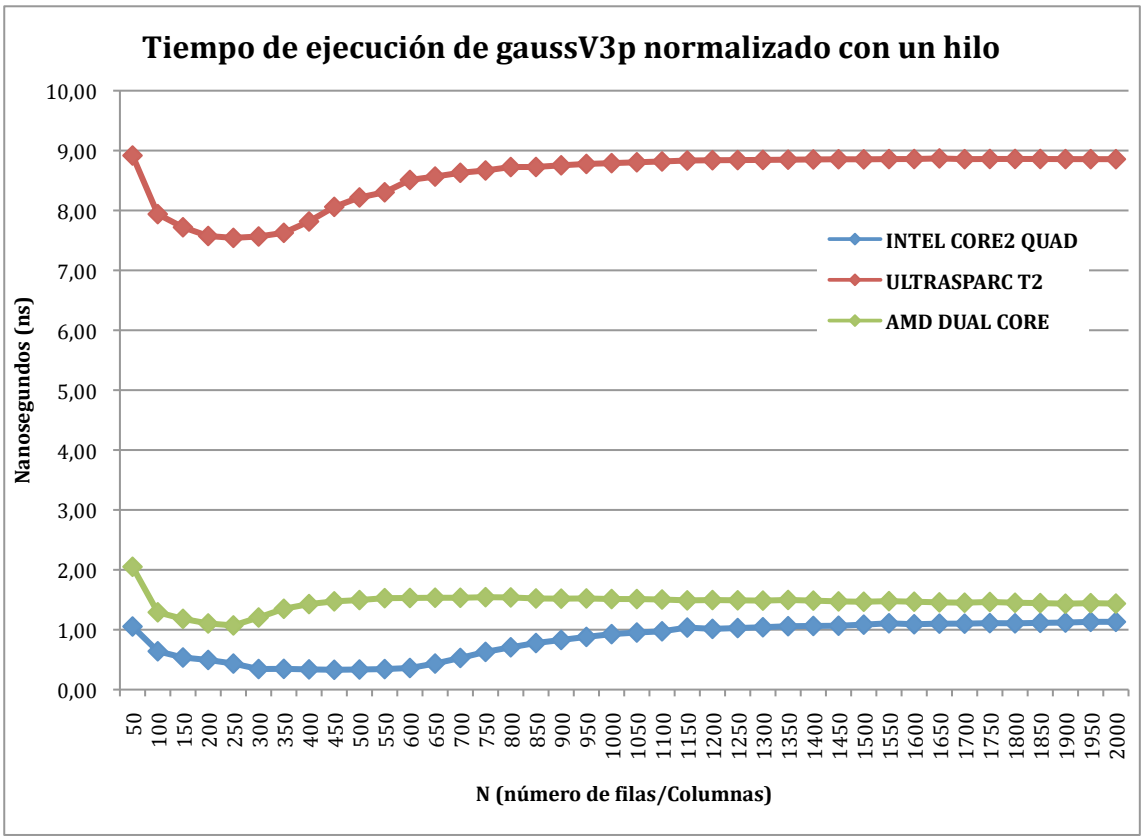

Figura 74. Comparación del tiempo de ejecución normalizado de la versión paralela ejecutadas con un hilo de ejecución en cada procesador.

En la figura 75, se muestra la gráfica comparativa del tiempo de ejecución normalizado de la aplicación paralela de la versión con KxC hilos que más rendimiento consigue cada uno de los procesadores. Dónde C es el núemor de núcleos de procesamiento utilizados y K es el número de hilos ejecutados por núcleo.

Observamos que para grandes tamaños del problema, el caso que nos interesa estudiar, es el procesador SUN UltraSPARC T2 el que alcanza un mayor rendimiento con un speedup de 1,21 sobre el procesador Intel Core2 Quad y de 1,6 respecto al AMD Dual Core. La elección del mejor procesador depende del tamaño del problema que se quiera resolver. Para tamaños pequeños (N<350) el procesadore que obtiene un mayor rendimiento es el procesador AMD Dual Core. En realidad, como y observábamos en el capítulo anterior, para pequeños valores del problema se obtiene mayor rendmiento con la versión serie del algoritmo ya que no sufre el "*overhead*" provocado por la gestión de los hilos. En esta última figura, se observa que para valores más pequeños que N=1600 el procesador que obtiene mayor rendimiento es el Intel. Para grandes volúmenes de trabajo a patir N=1600 es el procesador SUN el que obtiene un mayor rendimiento.

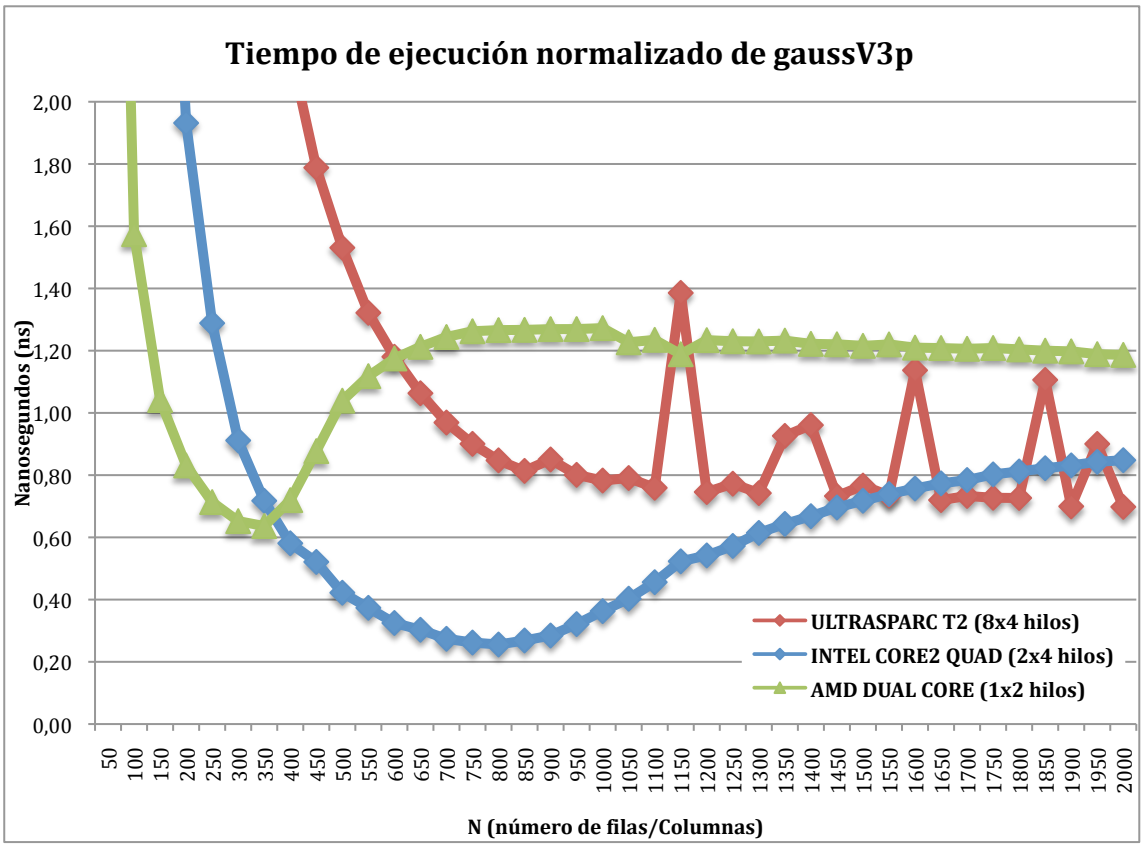

Figura 75. Comparación del tiempo de ejecución normalizado de la versión paralela con la mejor configuración de kxC hilos en cada procesador.

A continuación en la figura 76, se muestra para cada valor de N el mejor tiempo conseguido por las diferentes configuraciones de hilos estudiadas en el apartado anterior para cada uno de los procesadores.

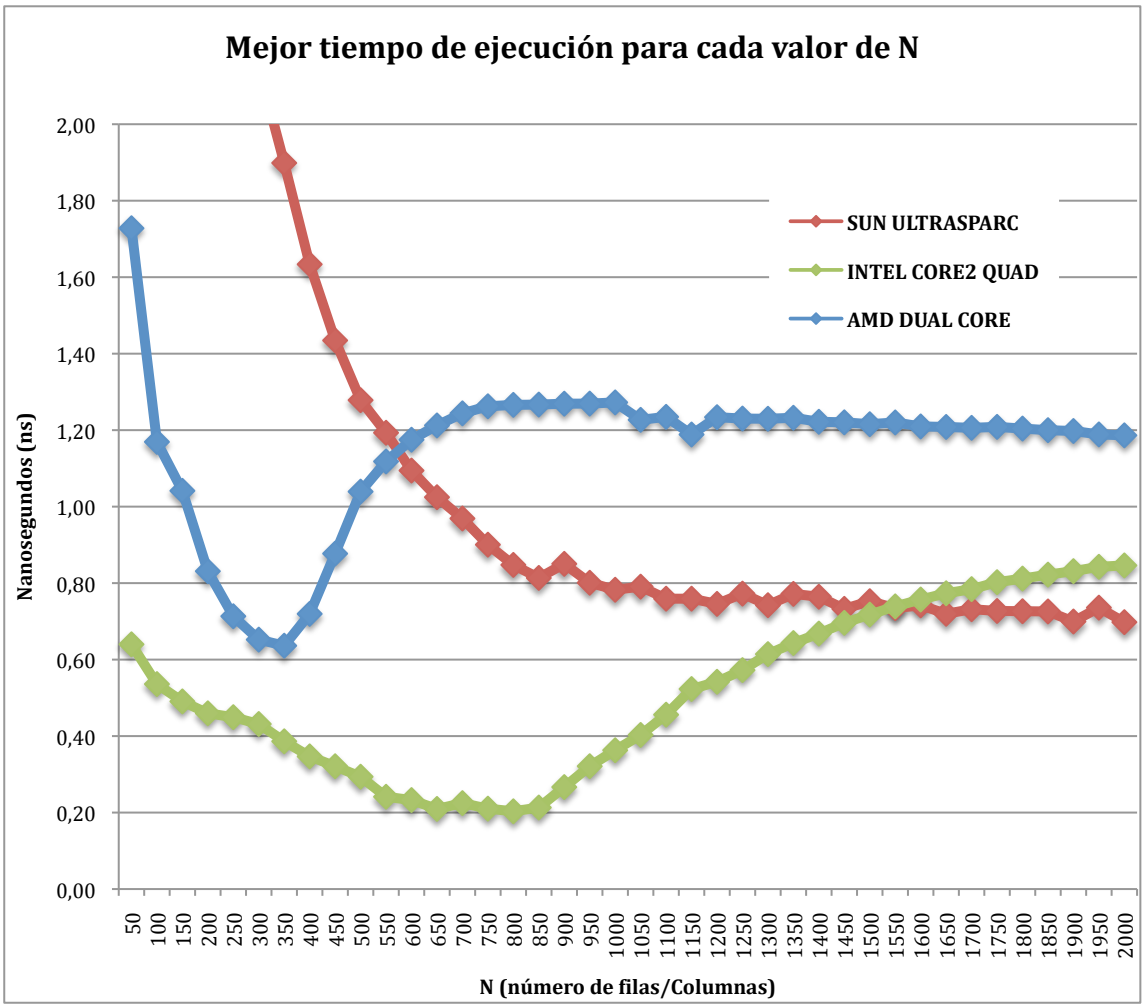

Figura 76. Comparación del mejor tiempo de ejecución normalizado para cada procesador.

5.3.4 Conclusiones finales

Existen dos medidas fundamentales en a la hora de calcular las prestaciones de un procesador: la latencia (tiempo comprendido entre el inicio y el final de una tarea) y el rendimiento (trabajo realizado por unidad de tiempo). Los diseñadores de procesadores calibran cuidadosamente el equilibrio entre la optimización de la latencia y el rendimiento ya que mejorando una puedes perjudicar a la otra. La conclusión que podemos extraer de los resultados mostrados en este apartado, es la existencia de dos tendencias diferentes en el diseño de procesadores. Los procesadores orientados a la latencia (*latency-oriented*) y los procesadores orientados al rendimiento (*throughput-oriented*)[10-13].

Los procesadores escalares están tradicionalmente orientados a reducir la latencia, asumiendo que el volumen de trabajo paralelo es pequeño. Esta tendencia es la que siguen los procesadores Intel Core2 Quad y AMD Dual Core. Observamos en la figura 74, que con la ejecución de un hilo obtienen resultados mucho mejores frente al procesador SUN UltraSPARC T2.

Por el contrario, los procesadores orientados al rendimiento, como el procesador SUN, sacrifican la ejecución serie (*single-thread*) alargando la latencia de la ejecución secuencial de una tarea para maximizar el rendimiento total de todas las múltiples tareas paralelas. Este aumento del rendimiento paralelo lo consiguen usando muchos procesadores simples, y por lo tanto pequeños, y dedicando mayor esfuerzo a proporcionar un elevado ancho de banda a memoria principal. Observamos en la figura 75, que el procesador SUN, aún teniendo núcleos de procesamiento más simples, es capaz de mejorar el rendimiento de los dos procesadores escalares Intel y AMD añadiendo más hilos de ejecución.

6 Problemas encontrados

En este capítulo se detallarán cronológicamente alguno de los problemas encontrados durante el desarrollo del proyecto.

6.1 Selección de la métrica adecuada

El primer problema que nos encontramos al realizar este proyecto, fue la comprensión e interpretación de los resultados obtenidos mediante la ejecución de la aplicación en el primer procesador. El tiempo de ejecución del algoritmo estudiado crece de manera cúbica (N^3). Como observamos en la gráfica 77, este crecimiento no es fácil de corroborarlo visualmente. Al presentarse el tiempo en una escala lineal, para tamaños pequeños de N los resultados se ven planos y para tamaños grandes de N los resultados son enormes. Presentando el tiempo en escala logarítmica facilitaría la lectura de estos valores pero de este modo sería aún más difícil visualizar el crecimiento cúbico. Por este motivo se optó por "normalizar" los resultados obtenidos dividiéndolos por la complejidad del algoritmo $O(N^3)$.

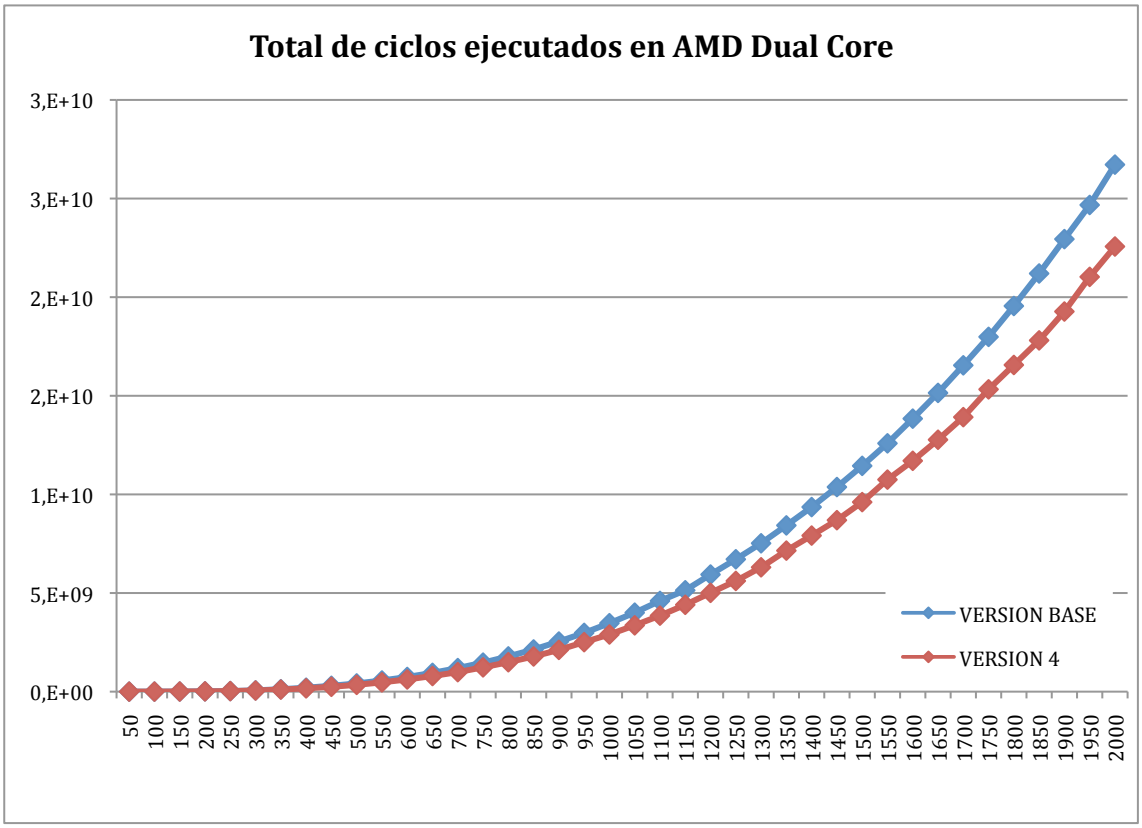

Figura 77. Total de ciclos ejecutados (AMD Dual Core).

En la figura 78 se muestran los mismos resultados mostrados en la figura 77 normalizados por la complejidad del algoritmo. En la grafica 78 podemos apreciar fácilmente como para pequeños valores de N, ambas versiones a partir de N=300 sufren un aumento repentino del total de ciclos ejecutados hasta estabilizarse en N= 500. Esta anomalía se percibía en la gráfica 77, por este motivo podemos afirmar que una correcta elección de la métrica adecuada al problema nos facilita enormemente el estudio del rendimiento de la aplicación.

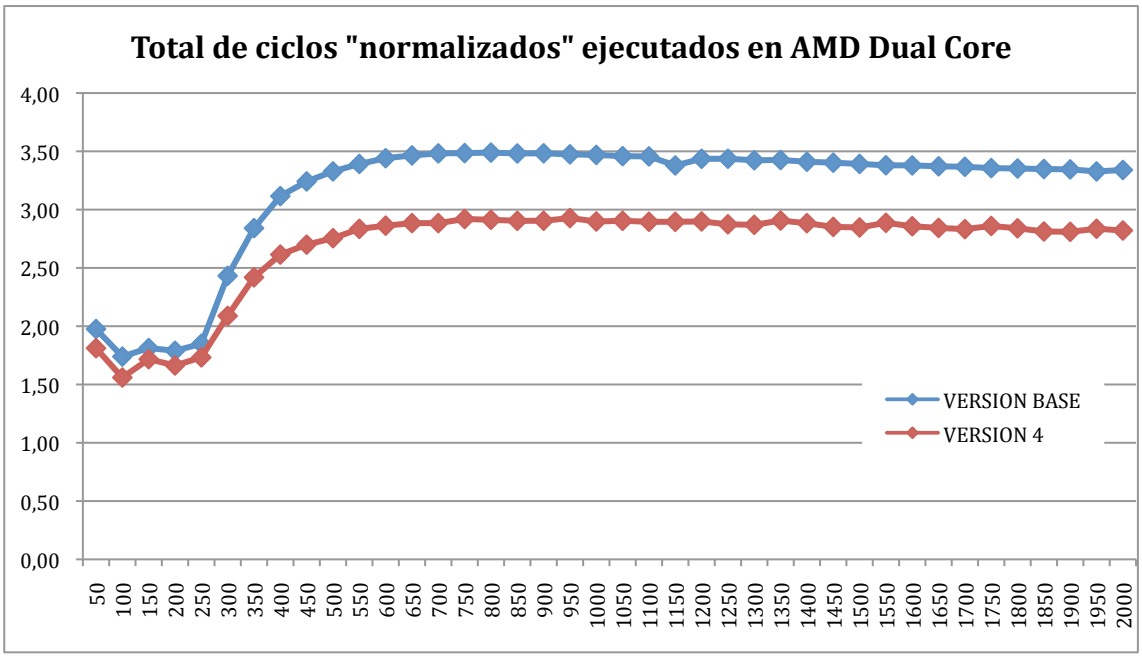

Figura 78. Total de ciclos ejecutados /complejidad $O(N^3)$ (AMD Dual Core).

Por tanto, para estudiar el rendimiento de una aplicación escoger la métrica correcta nos facilita la lectura de los resultados.

6.2 Problemas de paralelización

Durante el diseño del algoritmo serie de la aplicación, se propusieron una serie de optimizaciones que finalmente acabaron generando cuatro nuevas versiones del algoritmo base. Después del estudio en profundidad de los resultados, la versión que obtenía los mejores resultados en todos los sistemas escogidos fue la versión 4. Esta versión realizaba el cálculo del primer pivote fuera del bucle principal y las sucesivas búsquedas del pivote se realizaban simultáneamente mientras se actualizaban las filas restantes.

En primer lugar, se intentó llevar a cabo el diseño de la versión paralela partiendo de la versión 4 como versión base. Esta versión agresivamente optimizada, calcula el pivote de la iteración siguiente simultáneamente mientras se actualizan las filas de la iteración actual de dos en dos. Esta optimización, creaba una gran dependencia de datos y un elevado volumen de comunicaciones entre los diferentes hilos de ejecución que dificultaba en gran parte su paralelización.

Después de dedicar un gran tiempo con este primer intento sin obtener resultados satisfactorios, se optó por tomar como base para el algoritmo paralelo la anterior versión. Ya que como observábamos (figura 54), la mejora del rendimiento obtenida por la versión 4 respecto la versión 3, era muy pequeña y la dependencia de datos de la versión 3 era mucho menor.

El diseño de la versión paralela tomando como base la versión 3 del algoritmo serie, fue mucho más rápido y sencillo.

6.3 Anomalías de rendimiento

Una vez diseñado el algoritmo paralelo, se procedió a realizar los experimentos descritos en el apartado 4.5.2 con la versión paralela. Una vez obtenidos todos los resultados de la versión paralela en todos los sistemas estudiados se procedió a compararlos con los obtenidos por la versión serie.

Después de comparar los resultados se observó una anomalía de la versión paralela que se producía a partir de un determinado valor de N en el procesador Intel Core2 Quad. Esta anomalía se muestra en la figura 79. Para verificar que estos resultados no se debían a alteraciones puntuales por interferencias del sistema operativo u otros procesos, se repitió hasta 3 veces los experimentos. La anomalía se reproducía en todas las pruebas de la versión paralela pero, como se observa en la figura 79. Por este motivo se decidió volver a repetir los experimentos para la versión serie en el procesador Intel y así poder verificar que eran correctos.

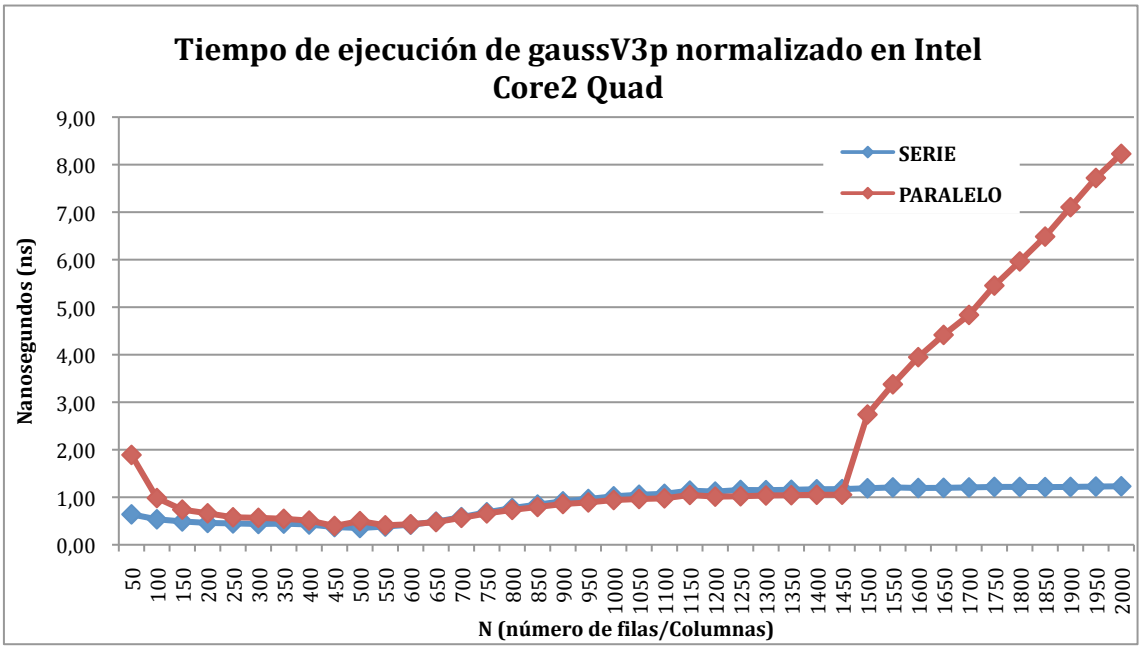

Figura 79. Tiempo de ejecución normalizado de la versión paralela (Intel Core2 Quad).

Tras verificar los resultados serie se observó que no se reproducía la anomalía en esta versión. De esta forma pudimos aislar la anomalía en la versión paralela. Para averiguar las posibles causas, se revisó detenidamente el código de la versión paralela hasta que se descubrió un error de codificación que fue corregido. El puntero que apuntaba a las filas que van a ser actualizadas se desplazaba por error una posición accediendo de esta manera a posiciones de memoria fuera de matriz de datos a tratar.

Después de corregir el error y verificar el correcto funcionamiento de la versión paralela se volvieron a realizar los experimentos paralelos en todos los sistemas para extraer de nuevo los resultados. De esta manera se pudimos observar que la anomalía ya no se producía y los resultados obtenidos eran coherentes.

En la figura 80 se muestran los resultados obtenidos una vez corregido el error de codificación.

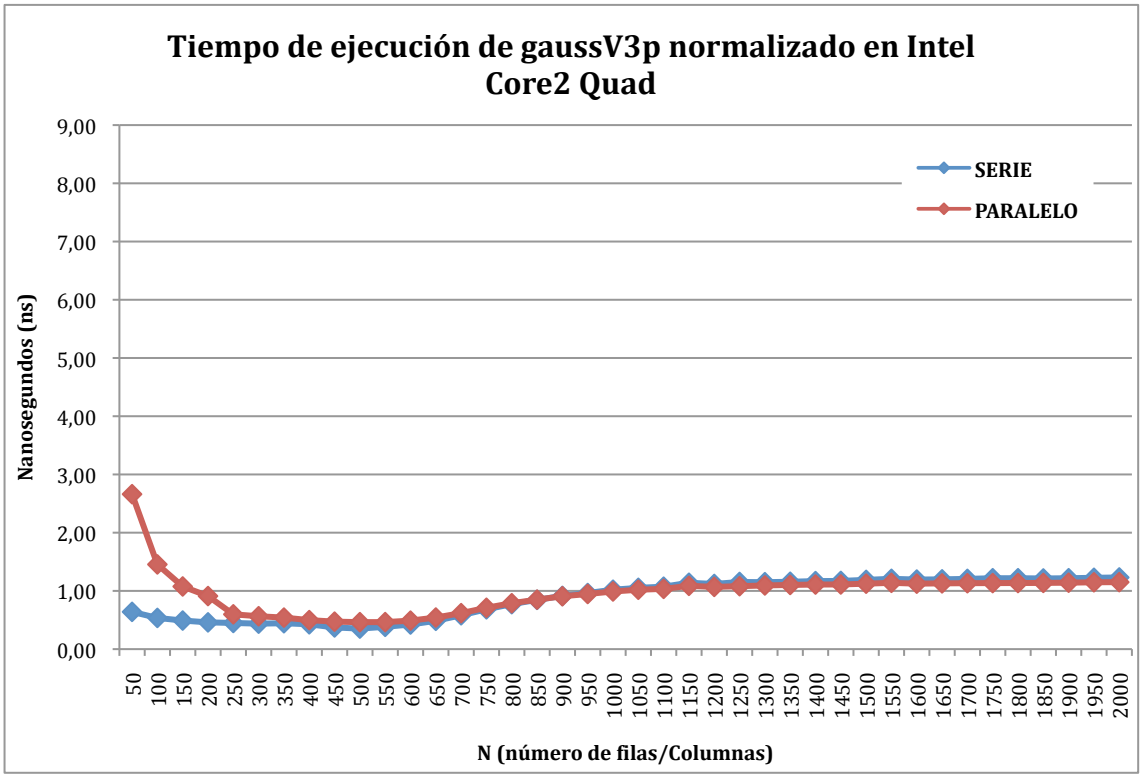

Figura 80. Tiempo de ejecución normalizado de la versión paralela (Intel Core2 Quad).

De esta manera podemos concluir, que el análisis exhaustivo del rendimiento puede ayudarnos a detectar "anomalías" del rendimiento provocadas por errores de funcionalidad.

7 Conclusiones, líneas futuras y ampliaciones

En este apartado de presentan describirán las conclusiones finales extraídas del estudio realizado sobre el algoritmo de la Eliminación Gaussiana en procesadores *mutlicore*, las líneas futuras y ampliaciones

7.1 conclusiones

Una de las conclusiones que podemos extraer del estudio realizado es que el peso de la paralelización recae sobre el programador. Para la generación de una aplicación paralela se debe modificar el código. Como pudimos observar en el capítulo 6, la puede ser muy costoso paralelizar una aplicación optimizada de forma agresiva para obtener un gran rendimiento en entornos "*single-thread*". El programador debe codificar el algoritmo minimizando las dependencias de datos, causantes de los *overheads* provocados por las comunicaciones y sincronizaciones. Estas dependencias de datos son las que dificultan la escalabilidad de los algoritmos paralelos en entornos multihilo. Para realizar una apliación paralela es mejor tomar como punto de partida con el algoritmo más simple posible y no necesariamente de la versión más optimizada. Es más fácil después de haber realizado la versión paralela, aplicar las optimizaciones que se crean oportunas que tomar como punto de partida una versión agresivamente optimizada.

El conocimiento del sistema *multicore* por parte del programador es vital para conseguir un mayor rendimiento. Los resultados de la ejecución *multicore* de una aplicación paralela varían en función de sistema en el cual se ejecute. El conocimiento del potencial y de las limitaciones del sistema *multicore* facilitan la codificación eficiente de la aplicación paralela. Por ejemplo si se dispone de un sistema con limitaciones en el ancho de banda y una alta capacidad de procesamiento, el programador debe hacer hincapié en reducir el acceso a memoria. Sin embargo, si se posee de un sistema con tecnología *multithreading hardware*, como el SUN UltraSPARC T2, no debe juzgarse al sistema por su rendimiento *single-thread*, si no que el programador debe minimizar las comunicaciones para asegurar la escalabilidad del algoritmo para explotar al 100% la capacidad de cómputo.

Como hemos explicado anteriormente la paralelización de una aplicación supone un coste elevado para el programador y realmente es efectiva para grandes volúmenes de trabajo. El *overhead* provocado por el peso de la zona secuencial del algoritmo junto con el *overhead* generado por la sincronización, comunicación y tratamiento de los hilos en pequeños problemas provoca un fuerte aumento del tiempo de ejecución y por tanto un gran descenso del rendimiento de la aplicación. A medida que se aumenta el tamaño del problema, el *overhead* generado por la paralelización del algoritmo, deja de tener tanto peso en el tiempo total de ejecución y el rendimiento de la aplicación paralela puede llegar a ser mucho mayor que el de la versión secuencial si no existen limitaciones que dificulten la escalabilidad de la aplicación.

El análisis exhaustivo del rendimiento de la aplicación junto con el aprendizaje de una metodología de trabajo robusta puede ayudar a la detección de mejoras i/o anomalías funcionales de la aplicación. Para realizar el estudio de los resultados del análisis del rendimiento es muy importante escoger una métrica adecuada al problema. De esta manera es más fácil detectar problemas de rendimiento.

Existen dos tendencias a la hora de diseñar procesadores. Los procesadores "*latency-oriented*", orientados a reducir la latencia asumiendo que el volumen de trabajo paralelo es pequeño y los procesadores "*throughput-oriented*", orientados a maximizar el rendimiento alargando la latencia. Para el algoritmo que hemos estudiado ambas filosofías son acertadas, cada una para un tamaño del problema diferente (figura 76). Los procesadores orientados a la latencia son idóneos para pequeños volúmenes de trabajo y que requieran un elevado cómputo. Los procesadores orientados al rendimiento obtienen un mejor resultado para grandes volúmenes de trabajo, cuando aprovechan el 100% de su capacidad de cómputo añadiendo más hilos de procesamiento.

7.2 Líneas futuras y ampliaciones

A continuación se describen las líneas futuras y ampliaciones a corto y largo plazo.

Durante la confección de este proyecto se ha detectado un problema de rendimiento en el procesador SUN UltraSPARC T2 en la versión paralela. Este procesador presentaba un *overhead* de orden N^3 a lo largo de todas las ejecuciones realizadas. Este *overhead* no crecía al añadir más hilos de procesamiento. Una posible ampliación seria el estudio en profundidad de esta anomalía para averiguar su origen.

En esta línea, también se detectó un bajo speedup de la versión con 8x4 hilos en el procesador SUN respecto la versión de 4x4 hilos. Por falta de tiempo no se llegó a realizar un estudio más detallado para averiguar cual de las dos posibles causas explicadas es el origen de este bajo rendimiento, la saturación del ancho de banda a memoria principal o la saturación de la capacidad de cómputo de los núcleos de procesamiento. Se deja como línea futura o ampliación este estudio más detallado y la búsqueda de posibles soluciones.

Viendo los buenos resultados obtenidos por el procesador SUN para elevados tamaños de N, una línea futura podría ser el estudio del rendimiento paralelo de la aplicación en un procesador "agresivamente" orientado al rendimiento como són las Unidades de Procesamiento Gráfico (GPU).

Una posible configuranción que podría mejorar el rendimiento de la aplicación paralela, seria un sistema donde un procesador "*latency-oriented*" y un procesador "*throughput-oriented*" trabajaran conjuntamente para poder hacer frente a todo tipo de cargas de trabajo []. Una línea futura a largo plazo seria la recodificación del algoritmo paralelo para que se pudiera repartir la carga de trabajo entre dos unidades con filosofías diferentes. De esta manera se podría analizar los resultados de combinar una CPU, que se ocupara de ejecutar la zona serie del algoritmo, y una GPU, que se ocuparía de parte paralela, para evaluar la viabilidad de este sistema.

8 Bibliografía

8.1 Referencias

[1] William Stallings, "*Organización y Arquitectura de Computadores",* (2000).

[2] Michael J. Quinn, "*Parallel programming in C with MPI and OpenMP*", Oregon State University (2004).

[3] Emilio Luque, "*Más rápido, más rápido: el reto de la velocidad en la arquitectura de los ordenadores",* Bellaterra Materials de la UAB.

[4] OpenSPARC-T2 documents, http://www.opensparc.net/

[5] Ilya Gavrichenkov, "*The Youngest of Yorkfields: Intel Core 2 Quad Q9300 Processor Review*" (2008).

[6] Intel Core2 Quad http://ark.intel.com/Product.aspx?id=33922

[7] OpenMP Application Program Interface, http://www.openmp.org/

[8] Performance Application Programming Interface, http://icl.cs.utk.edu/papi/

[9] Enric, Nart, "*Notes d'àlgebra lineal*", Bellaterra Materials de la UAB (2006).

[10] Weber, W.D. y Gupta, A. (1989). "*Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results*", ACM New York, NY, Volume:17, Issue 3, 273 - 280

[11] Saavedra-Barrera, R.; Culler, D. y Von Eicken, T. (1990). "*Analysis of multithreaded architectures for parallel computing*", ACM New York, NY, 169 - 178

[12] Kongetira, P., Aingaran, K., and Olukotun, k. "*Niagara: A 32-way multithreaded Sparc processor.*" IEEE Micro 25, (2005) 21-29.

[13] Davis, J.D., Laudon, J., and Olukotun, K. "*Maximizing CMP throughput with mediocre cores.*" IEEE Computer Society, Washington, D.C. (2005), 51-62.

9 Anexo I

9.1 Código C versión base del algoritmo serie

```
// VERSION 0: use of pointers, reduce iterations of loops, branches

/*
compilar en windows con: mingw32-gcc-sjlj.exe -fopenmp -o * *.c
compilar en solaris con: cc -lm -xopenmp -xO3 -o * *.c
compilar en linux con  : gcc -lm -fopenmp -o * *.c
*/

#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <tgmath.h>
#include <stdlib.h>
#include <omp.h>
#include "sample.h"


void gaussian ( int N, double **M )
{
{
    int  NTHR, THR, i, j, k;

    NTHR= omp_get_num_threads();
    THR = Sample_PAR_install();
    NTHR = 2;

    Sample_Start(THR);

    for(i=0; i<N-1; i++)
    {
      double Max, T, *ptr, *ptr2, *ptrPiv;
      int    piv;

      piv = i;
      ptr = M[i];
      Max = ptr[i];
      if (Max < 0)
        Max = -Max;

      for(j=i+1; j<N; j++)
      {
        T = M[j][i];
        if (T<0)
          T=-T;
        if (T > Max) {
          Max = T;
          piv = j;
        }
      }

      M[i]  = ptrPiv= M[piv];  // swap pivote to row number i
      M[piv]= ptr;
      ptrPiv += i;
      Max   = *ptrPiv;

      for(j=i+1; j<N; j++)
      {
        ptr = ptrPiv;
        ptr2= M[j]+i;
        T = *ptr2 / Max;
        *ptr2= 0.0;

        k= N-i;
        for(; k; k--)
        {
          ptr++; ptr2++;
          *ptr2 -= *ptr * T;
        }

      } // for j
    }  // for i

    Sample_Stop(THR);
  } //pragma omp parallel
}
```

9.2 Código C versión 3 del algoritmo serie

```
// PARARELL ROW VERSION 3: + Unroll of Loops on J  (less memory accesses on innner loop)

#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <tgmath.h>
#include <stdlib.h>
#include <omp.h>
#include "sample.h"


void gaussian ( int N, double **M )
{

  {
    int THR, i, j, k, piv, cnd;
    double T, T1, T2, M2, *ptr, *ptrPiv, *ptr2, *ptr3, Max;

    THR = Sample_PAR_install();
    Sample_Start(THR);

    for(i=0; i<N-1; i++)
    {
       j=i+1;
       piv = i;
       ptr = M[i];
       Max = ptr[i];
       Max = (Max >= 0)? Max: -Max;

       for(k=i+1; k<N-1; k++) //find pivote
       {
         T = M[k][i];
         T = (T >= 0)? T: -T;
         cnd = T > Max;
         Max = cnd? T: Max;
         piv = cnd? k: piv;
       }

       M[i] = ptrPiv= M[piv];  // swap pivote with row i
       M[piv] = ptr;
       ptrPiv+=i;
       Max = *ptrPiv;

       if ((N-j) & 1) //iteracion impar
       {
         ptr = ptrPiv;
         ptr2= M[j]+i;
         T1 = *ptr2 / Max;
         *ptr2= 0.0;
        j++; //saltamos la iteracion impar
         k = N-i;
         for(; k; k--) {
            ptr++; ptr2++;
            *ptr2 -= *ptr * T1;
          }
         }

      for(; j<N; j+=2)
      {
        ptr = ptrPiv;
        ptr2= M[j]+i;
        T1 = *ptr2 / Max;
        *ptr2= 0.0;

        ptr3= M[j+1]+i;
        T2 = *ptr3 / Max;
        *ptr3= 0.0;

        k = N-i;
        for(; k; k--)
        {
          ptr++; ptr2++; ptr3++;
          T=*ptr;
          *ptr2 -= T * T1;
          *ptr3 -= T * T2;
        }
      } // for j
    }  // for i
    Sample_Stop(THR);
  }
}
```

9.3 Código C versión 3 del algoritmo paralelo

```
// PARARELL ROW VERSION 3: + Unroll of Loops on J  (less memory accesses on innner loop)

#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <tgmath.h>
#include <stdlib.h>
#include <omp.h>
#include "sample.h"

void gaussian ( int N, double **M )
{
       double Temp, *ptrPiv; //shared variables;

#pragma omp parallel
  {
    int THR, i, j, k, piv, cnd;
    double T, T1, T2, *ptr, *ptr2, *ptr3;

    THR = Sample_PAR_install();
    Sample_Start(THR);

    for(i=0; i<N-1; i++)
    {
       j=i+1;

#pragma omp master
      {
         piv = i;
         T1 = M[i][i];
         T1 = (T1 >= 0)? T1: -T1;

         for(k=j; k<N; k++) //find pivote
         {
           T = M[k][i];
           T = (T >= 0)? T: -T;
           cnd= T > T1;
           T1 = cnd? T: T1;
           piv= cnd? k: piv;
         }

         ptr   = M[i];
         M[i]  = ptrPiv= M[piv];  // swap pivote with row i
         M[piv]= ptr;
         ptrPiv+=i;
         Temp  = *ptrPiv;
       }

#pragma omp barrier  // aqui se ha calculado ptrPiv y Temp (y se ha hecho el swap)

#pragma omp single nowait
    {  // iteracion impar
       if ((N-j) & 1)
       {
         ptr = ptrPiv;
         ptr2= M[j]+i;
         T1 = *ptr2 / Temp;
         *ptr2= 0.0;
         k = N - i;
         for(; k; k--) {
           ptr++; ptr2++;
           T=*ptr;
           *ptr2 -= T * T1;
         }
       }
    }// single

    if ((N-j) & 1) j++;  // saltar la iteracion impar

#pragma omp for schedule(static)
      for(j=j; j<N; j+=2)
      {
        ptr = ptrPiv;
        ptr2= M[j]+i;
        T1 = *ptr2 / Temp;
        *ptr2= 0.0;

        ptr3= M[j+1]+i;
        T2 = *ptr3 / Temp;
        *ptr3= 0.0;

        k = N-i;
        for(; k; k--)
        {
          ptr++; ptr2++; ptr3++;
          T=*ptr;
          *ptr2 -= T * T1;
          *ptr3 -= T * T2;
        }
      } // for j
    }  // for i
    Sample_Stop(THR);
  } //pragma omp parallel
}
```

RESUMEN

Dada la necesidad de computar cada vez problemas más grandes en un tiempo razonable, la industria ha optado por aumentar el número de núcleos de procesamiento secuencial en el mismo chip para obtener futuros incrementos del rendimiento, en vez de diseñar un procesador con una única unidad de proceso más rápida y con mayor capacidad de cómputo. Este proyecto presenta un análisis de rendimiento de los resultados obtenidos por el algoritmo de la Eliminación Gaussiana en tres procesadores multi-core. Los resultados muestran las particularidades existentes entre las diferentes arquitecturas multi-core/multi-thread y la importancia del estudio del rendimiento en estos sistemas.

RESUM

Donada la necessitat de computar cada vegada problemes més grans en un temps raonable, la indústria ha optat per augmentar el nombre de nuclis de processament seqüencial al mateix xip per obtenir futurs increments del rendiment, en comptes de dissenyar un processador amb una única unitat de processament més ràpida i amb major capacitat de còmput. Aquest projecte presenta una anàlisi de rendiment dels resultats obtinguts per l'algorisme de l'Eliminació Gaussiana en tres processadors multi-core. Els resultats mostren les particularitats existents entre les diferents arquitectures multi-core/multi-thread i la importància de l'estudi del rendiment en aquests sistemes.

ABSTRACT

Due to the need of computing increasingly larger problems in a reasonable time, the industry has chosen to increase the number of simple sequential processing cores on a single chip for future increases in performance, rather than designing a processor with a single processing unit faster and with greater computing power. This project presents a performance analysis of the results obtained by the Gaussian Elimination algorithm on three multi-core processors. The results show the characteristics between the different architectures multi-core/multi-thread and the importance of studying the performance of these systems.