Universitat Autònoma de Barcelona

# Departamento de Arquitectura de Computadores y Sistemas Operativos
# Máster en Computación de Altas Prestaciones

## Análisis de Rendimiento de Aplicaciones Paralelas de Memoria Compartida: problema N-body.

Memoria del trabajo de investigación del "Máster en Computación de Altas Prestaciones", realizada por Héctor Manuel Doblado Ruesga, bajo la dirección de Juan Carlos Moure
Presentada en la Escuela Técnica Superior de Ingeniería (Departamento de Arquitectura de Computadores y Sistemas Operativos)

**2009**

ÍNDICE

## RESUMEN

Este trabajo analiza el rendimiento de cuatro nodos de cómputo multiprocesador de memoria compartida para resolver el problema N-body. Se paraleliza el algoritmo serie, y se codifica usando el lenguaje C extendido con OpenMP. El resultado son dos variantes que obedecen a dos criterios de optimización diferentes: minimizar los requisitos de memoria y minimizar el volumen de cómputo. Posteriormente, se realiza un proceso de análisis de las prestaciones del programa sobre los nodos de cómputo. Se modela el rendimiento de las variantes secuenciales y paralelas de la aplicación, y de los nodos de cómputo; se instrumentan y ejecutan los programas para obtener resultados en forma de varias métricas; finalmente se muestran e interpretan los resultados, proporcionando claves que explican ineficiencias y cuellos de botella en el rendimiento y posibles líneas de mejora. La experiencia de este estudio concreto ha permitido esbozar una incipiente metodología de análisis de rendimiento, identificación de problemas y sintonización de algoritmos a nodos de cómputo multiprocesador de memoria compartida.



***Palabras clave:*** *Multicore, multithread, rendimiento, OpenMP, N-body, PAPI*.



## RESUM

Aquest treball analitza el rendiment de quatre nodes de còmput multiprocessador de memòria compartida per resoldre el problema N-body. Es paral·lelitza l'algoritme sèrie, i es codifica utilitzant el llenguatge C estès amb OpenMP. El resultat són dues variants que obeeixen a dos criteris d'optimització diferents: minimitzar els requisits de memòria i minimitzar el volum de còmput. Posteriorment, es realitza un procés d'anàlisis de les prestacions del programa sobre els nodes de còmput. Es modela el rendiment de les variants seqüencials i paral·leles de l'aplicació, i dels nodes de còmput; s'instrumenten i s'executen els programes per obtenir resultats en forma de diverses mètriques; finalment es mostren i s'interpreten els resultats, proporcionant claus que expliquen ineficiències i colls d'ampolla en el rendiment i possibles línies de millora. L'experiència d'aquest estudi concret ha permès esbossar una incipient metodologia d'anàlisis de rendiment, identificació de problemes i sintonització d'algoritmes a nodes de còmput multiprocessador de memòria compartida.



***Paraules clau:*** *Multicore, multithread, rendiment, OpenMP, N-body, PAPI*.



## ABSTRACT

This research analyzes the performance of four, shared-memory, multiprocessor, computing nodes solving the N-body problem. The sequential algorithm is parallelized and coded using the C language extended by OpenMP. Two program variations are designed, obeying two different optimization goals: minimize memory requirements and minimize the amount of computation. Subsequently, we analyze the program's performance over the computation nodes. We model the performance of the serial and parallel applications and the performance of the computing nodes; the programs are implemented and executed to obtain results in form of several metrics; finally, results are displayed and interpreted, providing keys to explain the performance inefficiencies and bottlenecks, and showing possible areas for improvement. The experience of this study has made possible an incipient methodology to analyze performance, to identify problems, and to tune an algorithm on shared memory multiprocessor nodes.



***Keywords****: Multicore, multithread, performance, OpenMP, N-body, PAPI*.

# 1 Introducción

La evolución de los procesadores ha pasado por diversas etapas en el transcurso de los años, con el objetivo de dar el máximo de rendimiento posible. Sin entrar mucho en detalle, podemos distinguir tres etapas. En la primera, el aumento de rendimiento se obtuvo principalmente aumentando el nivel de integración del procesador. En la segunda etapa el rendimiento siguió incrementándose, aumentando la frecuencia de reloj del procesador, y en la tercera etapa, debido a problemas económicos y técnicos que dificultan seguir incrementando la frecuencia de reloj, se opta por integrar diversos núcleos (cores) en el mismo chip.

La incorporación de conceptos como el pipeline, la memoria cache y procesador superescalar mejoraron el rendimiento de los procesadores. El rendimiento máximo teórico de un procesador es también conocido como rendimiento Pico Teórico. Los procesadores han ido evolucionando par conseguir un mayor rendimiento Pico Teórico gracias a las mejoras comentadas, nivel de integración, frecuencia de reloj, arquitectura interna: pipeline, memoria-cache, superescalar, multi-core/multi-thread.

En la figura 1 se muestra conceptualmente la siguiente conjetura hipotética: La diferencia entre el Rendimiento Pico Teórico de los procesadores y el rendimiento que se pude obtener sin esfuerzo extra del programador, es decir simplemente recompilando.

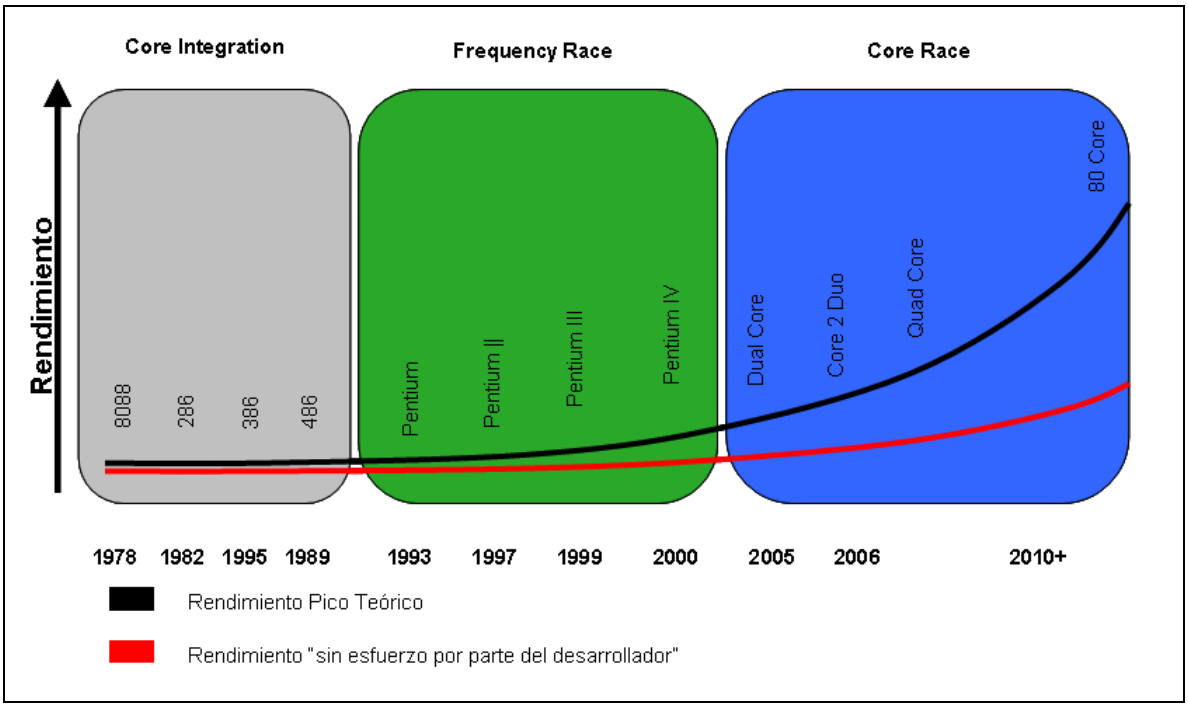



**Figura 1:** Conjetura hipotética de rendimiento pico teórico y de rendimiento “sin esfuerzo” (se utilizan como referencia para la cronología, los procesadores Intel).

Las mejoras aportadas por los procesadores, como pipeline, ejecución fuera de orden, ejecución especulativa, dificultan el trabajo del compilador y del programador. Esta dificultad añadida y la propia naturaleza de los problemas a implementar, obligan al programador a realizar un esfuerzo de análisis del procesador y del problema a tratar. Sin este esfuerzo extra del programador, el rendimiento obtenido no estará en concordancia con el rendimiento aportado por dichas mejoras.

Con la aparición de los multi-core/multi-thread, las aplicaciones mono-thread no pueden aprovecharse del rendimiento extra, que proporciona poder ejecutar multiples threads en paralelo. Es necesario utilizar paradigmas de programación paralela y utilizar, por ejemplo, pthreads, o OpenMP. Aparecen las aplicaciones multi-thread. Este cambio en el diseño y programación de aplicaciones dificulta el trabajo del programador. Por lo que el programador necesitará de una metodología para el diseño e implementación de aplicaciones paralelas. También hay que tener en cuenta que aunque las aplicaciones paralelas ejecutadas sobre procesadores multi-core/multi-thread pueden escalar casi linealmente en rendimiento cuando se ejecutan con un número bajo de cores+threads (2, 4, 8), a medida que aumentamos el número de cores+threads la escalabilidad va dejando de ser lineal. En el transcurso de este documento, explicaremos los factores que impiden un alto rendimiento de las aplicaciones paralelas ejecutadas en procesadores multi-core/multi-thread.

En el contexto de sistemas multi-core con memoria compartida y de aplicaciones paralelas observamos lo siguiente:

- Dado un algoritmo A podemos encontrarnos que con una codificación y sintonización específica obtengamos un alto rendimiento ejecutando sobre un sistema 1 pero que con estos ajustes no únicamente no obtengamos un alto rendimiento sobre un sistema 2, si no que incluso obtengamos un rendimiento menor que antes de realizar los ajustes. En la figura 2 se muestra un esquema que asocia las posibles codificaciones de dos algoritmos al sistema en el que dan un mayor rendimiento.

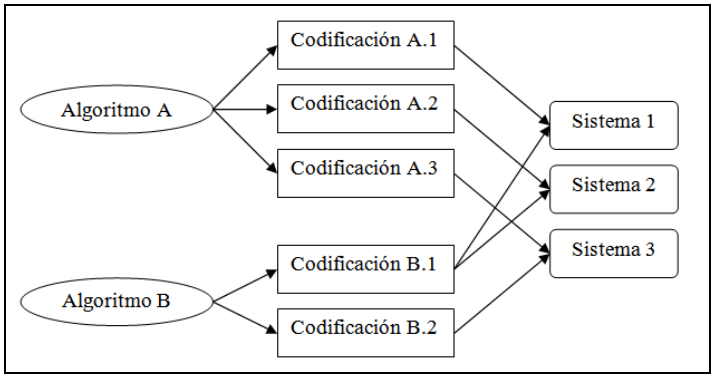



**Figura 2:** Las codificaciones y sintonizaciones de una aplicación no tienen por que proporcionar un alto rendimiento en cualquier sistema de memoria compartida.

- Es necesaria una codificación y sintonización específica de la aplicación para cada sistema.
- Obtener alto rendimiento requiere de un análisis detallado de la aplicación y del sistema.

Esto supone:

- Que desconocemos a priori si una codificación y sintonización concretas nos proporcionaran un alto rendimiento en un sistema.
- Que dada una aplicación, desconocemos a priori en que sistema obtendremos mejor rendimiento.
- Un alto coste en tiempo cada vez que debemos codificar y sintonizar una aplicación para un sistema concreto.

Pretendemos explicar el rendimiento de procesadores multi-core de memoria compartida de diferentes arquitecturas, mediante el análisis y la comparación de su comportamiento. Para explicar el rendimiento, se buscarán los factores que determinan el tiempo de ejecución y se cuantificará su impacto. Para aproximarse a este objetivo estudiaremos variantes de un algoritmo tipo y computadores paralelos con arquitecturas diferentes.

## 1.1. Descripción del trabajo y objetivos

La descripción del trabajo se divide en:

- Aprender el modelo de programación de memoria compartida y su codificación usando OpenMP.
- Comprender un problema paralelo clásico de la literatura: N-Body.
- Codificar el algoritmo en C y paralelizar el algoritmo con OpenMP.
- Ejecutar en cuatro nodos de cómputo de características muy diferentes usando una infraestructura API de acceso a contadores hardware, como por ejemplo PAPI.
- Interpretar los resultados de rendimiento para explicarlos y extraer conclusiones que permitan optimizar la codificación de cada algoritmo en cada nodo de cómputo.

Los objetivos de este trabajo son:

- Obtener los conocimientos necesarios para iniciar una investigación relacionada con el estudio de los nodos de cómputo, utilizando C, OpenMP, y API de acceso a contadores hardware.

- Iniciarse en el proceso de investigación:
  - Realizando una interpretación de los resultados de rendimiento en nodos de cómputo con características muy diferentes.
  - Extrayendo conclusiones a partir de la interpretación que permitan optimizar los algoritmos en cada nodo de cómputo.

- Aportar información y recomendaciones sobre como mejorar el rendimiento de las partes críticas de las aplicaciones paralelas en diferentes arquitecturas multiprocesador con memoria compartida, que ayude a los desarrolladores en su trabajo.

Obtendremos información sobre el rendimiento de partes críticas de aplicaciones paralelas en diferentes arquitecturas multiprocesador con memoria compartida. A partir de está información realizaremos recomendaciones que ayudarán a los desarrolladores a:

- Identificar posibles mejoras.
- Estimar los órdenes de magnitud del beneficio en rendimiento que se pueden obtener aplicando dichas mejoras.
- Ayudar a seleccionar la arquitectura adecuada para sus propósitos.

Hay que tener en cuenta el coste de un desarrollador y el coste de alquiler/adquisición del hardware. El ahorro en tiempo del desarrollador y el aumento obtenido en rendimiento de la aplicación, así como un mayor rendimiento obtenido al utilizar la arquitectura adecuada, repercutirán directamente en los costos y en la amortización de la inversión.

Futuras fases de síntesis que podrían aprovechar la información obtenida en este trabajo son:

- Automatizar parte del proceso de optimización de la paralelización de aplicaciones en sistemas multiprocesador de memoria compartida.
- Creación de modelos de predicción de comportamiento de los procesadores.

## 1.2. Método y plan de trabajo

El método que se pretende utilizar es el experimental, intentando establecer relaciones de causa y efecto entre las diferentes variantes de las aplicaciones, las diferentes características de los sistemas multi-core de memoria compartida y el rendimiento que se obtiene.

Método experimental:

Ejecutaremos las diferentes variantes de los algoritmos y micro-benchmarks que obtengamos, en diferentes sistemas multi-core de memoria compartida para analizar el rendimiento.

Estudios de casos:

- Se estudiará en profundidad el algoritmo: N-body.
- Cuatro sistemas multi-core de memoria compartida de fabricantes diferentes: Intel, AMD, y SUN Microsystems.

**Plan de trabajo:** La línea general de trabajo consistirá en:

- Documentarse sobre:
  - modelos de aplicación.
  - arquitectura y modelo de rendimiento de los nodos de cómputo.
  - instrumentos para medir el rendimiento.
  - metodología para aumentar el rendimiento.
- Estudio de la aplicación, desarrollo serie, paralelización usando OpenMP y definición de su modelo.
- Estudio del nodo de cómputo.
- Creación de instrumentos (subpartes críticas de la aplicación).
- Ejecución de los instrumentos (subpartes críticas de la aplicación) e interpretación de los resultados.
- Ajuste de los instrumentos, y del modelo de aplicación.
- Ejecución de la aplicación e interpretación de los resultados.
- Extracción de conclusiones para aumentar el rendimiento.

## 1.3. Esquema de la memoria

La memoria se compone de 7 capítulos, que serán descritos brevemente a continación.

**Capítulo 1**: Consta de la introducción, seguida del resumen del del trabajo y su justificación. Posteriormente se explica el método y plan de trabajo y se plantea el calendario. Finalmente se describen los diferentes capítulos de los que consta la memoria.

**Capítulo 2:** Marco teórico, en el que se describen y explican los conceptos necesarios para comprender el trabajo. Se introduce al lector en conceptos relacionados con la evolución de los procesadores, las limitaciones de los procesadores single-thread, los procesadores multi-core/multi-thread, los factores que limitan el rendimiento en sistemas multi-core/multi-thread. Tambien se expone una pequeña introducción a OpenMP y a PAPI (API de rendimiento).

**Capítulo 3:** En él se plantea y describe el problema de n-body en su método directo de resolución (partícula-partícula), con interacción de fuerzas de tipo gravitacional.

**Capítulo 4:** Se realiza el diseño del algoritmo serie para el problema n-body. Para ello se contemplan diferentes variantes del algoritmo. De entre estas variantes se seleccionan dos, a las que se llaman nbody-A y nbody-F. Para cada una de estas dos variantes, el trabajo se divide en dos partes. En primer lugar se realiza un análisis del algoritmo independiente del procesador. En segundo lugar se implementa en C, y se ejecuta en cada uno de los sistemas considerados, tomando medidas de rendimiento. Como resultado de este capítulo, se identifican las características de los programas y de los procesadores que más influyen en el rendimiento.

**Capítulo 5:** Se estudia la paralelización funcional de los algoritmos nbody-A y nbody-F obtenidos anteriormente y se analiza y se trata de mejorar su rendimiento teórico. Igual que en el capítulo 4, para cada uno de los algoritmos, el trabajo se divide en dos partes. En la primera se realiza un analisis del algoritmo independiente del procesador. En la segunda se implementa en C, utilizando directivas de OpenMP, y se ejecuta en cada uno de los sistemas considerados tomando medidas de rendimiento. Seguidamente, en este capítulo se incluye el estudio específico del overhead de la paralelización y de la sincronización.

**Capítulo 6:** Incluye las conclusiones, las líneas futuras de investigación, algunos problemas metodológicos a destacar, y finalmente una propuesta de metodología de diseño y análisis de aplicaciones paralelas.

**Capitulo 7:** Bibliografía

# 2 Marco Teórico

## 2.1. Evolución del procesador single-thread

Describiremos el procesador clásico y conceptos asociados, para explicar posteriormente los procesadores multi-thread y multi-core, así como los problemas para conseguir alto rendimiento en estos sistemas. Finalmente realizaremos una breve introducción a OpenMP y PAPI.

## 2.2. Procesador clásico

En el procesador clásico podíamos dividir el proceso de ejecución de una instrucción en cuatro etapas: búsqueda de la instrucción (prefetch y fetch), decodificación de la instrucción, búsqueda de operandos y ejecución de la instrucción (ejecución y escritura de los resultados). En la figura 3 se muestra la ejecución secuencial de dos instrucciones en un procesador clásico. Se puede apreciar que hasta que la primera instrucción no finaliza, no se puede comenzar la busqueda de la segunda instrucción.

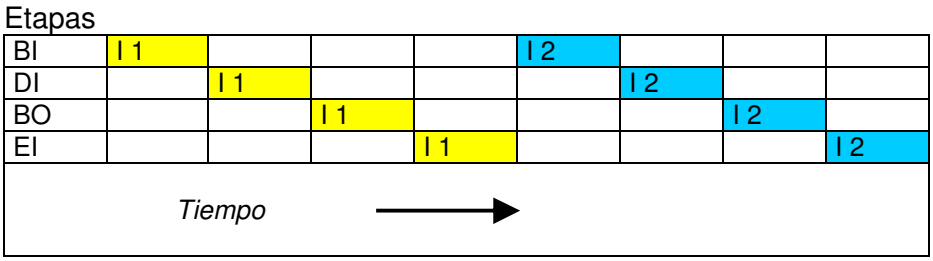



**Figura 3:** Ejecución secuencial de instrucciones en un procesador clásico.

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

$$T_{ej} = N \times CPI \times t$$

donde

N = número de instrucciones de un programa. Depende de la arquitectura del computador a través del Repertorio de Instrucciones.

CPI = número medio de Ciclos Por Instrucción.

t = tiempo de un ciclo de instrucción.

Para reducir el tiempo de ejecución en esta ecuación independientemente del nivel de integración, existen dos alternativas que son, reducir el número de instrucciones y/o reducir el CPI. Existen dos tipos de filosofías en las arquitecturas de procesadores, que atacan el problema desde distintos puntos de vista.

RISC (Computadoras con un conjunto de instrucciones reducido): Implementación H/W más simple, por tanto más rápida y eficiente. Se basa en disponer de un repertorio de instrucciones reducido, permitiendo su implementación por hardware. Los programas tendrán un número elevado de instrucciones pero prácticamente la totalidad de ellas se ejecutarán en un ciclo de reloj. En este tipo de arquitectura hay una fuerte dependencia del compilador en la generación eficiente de código máquina.

CISC (Computadoras con un conjunto de instrucciones complejo): Implementación H/W más compleja y por tanto más lenta e ineficiente. Se basa en disponer de un repertorio de instrucciones amplio y complejo. En ocasiones la ejecución de algunas instrucciones se

implementa de forma microprogramada, lo que significa que cada instrucción máquina es interpretada por un microprograma localizado en una memoria, en el circuito integrado del procesador. El número de instrucciones de un programa es menor que en el RISC, pero el CPI suele ser mayor.

## Procesamiento Segmentado (Pipeline)

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 número de instrucciones ejecutadas por ciclo. Como muestra la figura 4, con la ejecución segmentada de instrucciones, un procesador segmentado de cuatro etapas pasa de ejecutar una instrucción cada cuatro ciclos, a una instrucción por ciclo.

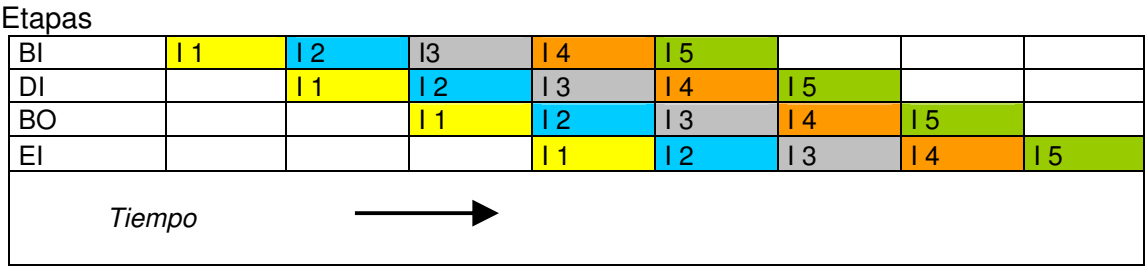



**Figura 4:** Ejecución segmentada de instrucciones.

No obstante, los saltos y las dependencias de datos limitan esta ganancia.

En el primer caso las instrucciones de bifurcación condicional pueden derivar en diferentes caminos. Por tanto no es posible conocer la siguiente instrucción hasta que no se haya ejecutado por completo la instrucción de bifurcación.

En el segundo caso una instrucción podría quedar detenida en una etapa a la espera del dato que otra instrucción anterior debe calcular.

## 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.

### 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:

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

- 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.

- 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 PC llegue a la instrucción de salto).

## 2.3. 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.

**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 las figuras 5 y 6 que representan las fases de ejecución de un programa single-thread en dos procesadores con distinta frecuencia de reloj.

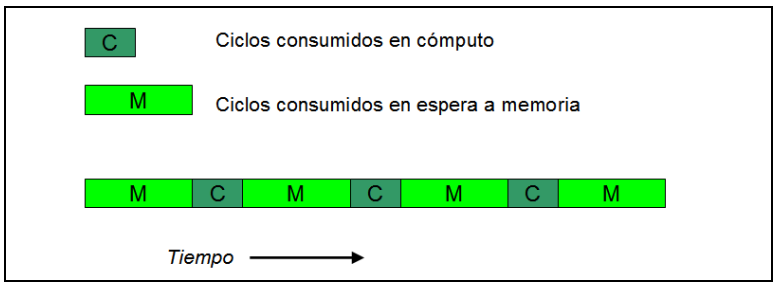



**Figura 5:** Fases de la ejecución de un programa single-thread.

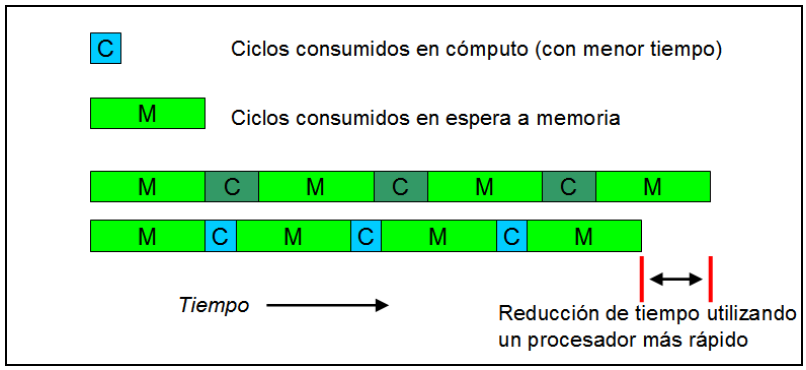



**Figura 6:** Fases de la ejecución de un programa single-thread con un procesador más rápido que el de la Figura 5.

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 thread a otro thread cuando el thread que esta ocupando el procesador queda parado. Esta solución se analizará en profundidad más adelante.

### 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 (consumo eléctrico, y gasto dedicado a la disipación del calor y refrigeración), como tecnológico (dificultad para disipar la gran cantidad de calor generado, de la pequeña superficie de un procesador)

Por estos motivos, se abandona la idea de aumentar la frecuencia de reloj del procesador para aumentar el rendimiento, y se opta por añadir más procesadores en el mismo chip. Con esta solución el calor se incrementa de forma lineal y no exponencial como ocurre con el aumento de frecuencia de reloj.

### El procesador Multi-thread y Multi-core

Para entender conceptos como el Multithreading por Software, el Multithreading por Hardware y las ventajas de los procesadores multi-core es necesario conocer la diferencia entre thread y proceso.

**Thread y Proceso**

Un thread es una secuencia de código ejecutable. Los threads 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 threads tiene tantos flujos de control como threads. Cada thread se ejecuta con su propia secuencia de instrucciones de forma concurrente e independiente.

Un proceso es una instancia de un programa formado por uno o más threads, con su propio espacio de direccionamiento.

**Multi-threading por Software**

Debemos distinguir entre multi-programación y aplicaciones paralelas.

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

En la figura 7 se observa como el Thread 1, se para al producirse una excepción que requiere esperar un tiempo. Inmediatamente el Sistema Operativo cambia de contexto y ejecuta otro thread. 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).

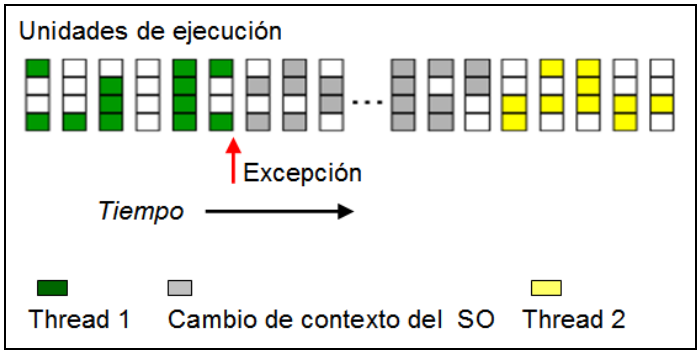



**Figura 7**: Cambio de Thread del SO en un entorno multiprogramado.

El multi-threading por Software posibilita la realización de aplicaciones paralelas. Es un nuevo modelo de programación que permite a múltiples threads existir dentro de un proceso. Los threads 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 thread en lugar de a nivel de proceso son varias. Los threads se encuentran todos dentro de un mismo proceso y por lo tanto pueden compartir los datos globales. Además, una petición bloqueante de un thread no parará la ejecución de otro thread. Por último, si el procesador lo soporta, los diferentes threads están asociados a diferentes conjuntos de registros, por lo que el cambio de contexto del procesador podrá realizase de forma eficiente.

**Multi-threading por Hardware**

El multi-threading 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 Multi-threading**

En el modelo Large-Grain Multithreading el procesador ejecuta el thread 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 thread. Este método tiene la ventaja de ser sencillo de implementar.

En la Figura 8 se muestra la ejecución de las los diferentes threads en la que el procesador cambia de thread cuando se produce un evento que requiere un tiempo de espera por ejemplo un fallo de cache.

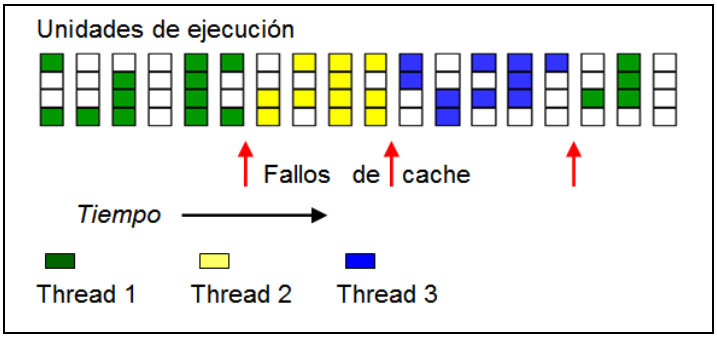



**Figura 8:** Ejecución de threads siguiendo el modelo Large-Grain Multithreading.

**Fine-grained Multithreading**

Fine-grained Multi-threading se basa en un cambio "rápido" entre threads, ejecutando en cada ciclo un thread 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 threads bloqueados, se ejecutan instrucciones de diferentes threads. Este enfoque tiene la ventaja de eliminar las dependencias de datos que paran el procesador. Al pertenecer las instrucciones a diferentes threads, las dependencias de datos y de control desaparecen.

En la figura 9 se muestra como en cada ciclo se ejecutan instrucciones de threads diferentes. El procesador no espera a que se produzca una interrupción (fallo de caché,....) para saltar al siguiente thread.

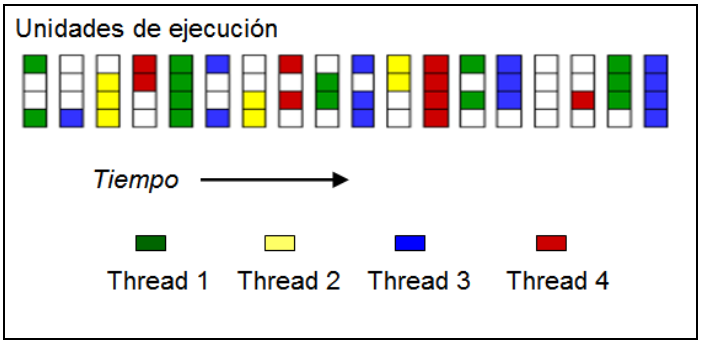



**Figura 9:** Ejecución de threads siguiendo el modelo Fine-grained Multithreading.

**Simultaneous Multi-threading**

La filosofía de Simultaneous Multi-threading (SMT) consiste en poder ejecutar instrucciones de diferentes threads, 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 threads aumentaría la complejidad y, por tanto el coste. Por este motivo en las implementaciones SMT se opta por reducir el número de threads.

Simultaneous Multi-threading 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. Multi-core

Los procesadores multi-core 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 threads independientes, de forma que es posible la concurrencia. Es decir, los threads 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 threads, las comunicaciones entre las memorias de los procesadores, y el posible desbalanceo (en las aplicaciones) de volumen de cómputo por thread (threads esperando a que otros threads finalicen). Describiremos estos factores con más detalle, más adelante. Experimentalmente con nbody (partícula-partícula) hemos comprobado que el rendimiento (en un procesador SPARC T2) escala prácticamente linealmente hasta 16 threads, perdiendo progresivamente hasta un 50% de eficacia por thread al ir incrementando el número de threads hasta 32.

Otra de las ventajas es poder aplicar el diseño de un core ya probado a los demás cores del procesador. Así se evita el coste que supondría el diseño de nuevos core.

Como inconveniente, las aplicaciones han de ser correctamente paralelizadas para aprovechar todo el potencial de los procesadores multi-core.

**Arquitecturas multi-procesador**

Los procesadores de una arquitectura multi-procesador comparten la memoria principal. Existen dos alternativas para compartir la memoria principal.

En la primera los procesadores del sistema tienen el mismo tiempo de acceso a memoria, a esta arquitectura se la conoce como arquitectura U.M.A (Acceso uniforme a memoria).

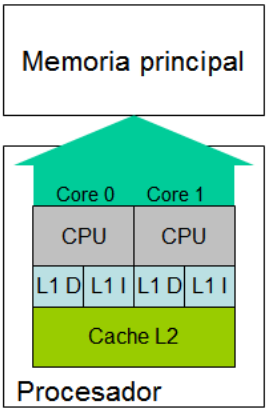
En la segunda el acceso parcial o total a la memoria es controlado por un único procesador, lo que provoca que este procesador tenga un tiempo de acceso menor, a la memoria controlada por él, que el resto de procesadores. El resto de procesadores debe interactuar con el procesador que controla la memoria para acceder a élla, a esta arquitectura se la conoce como arquitectura N.U.M.A. (Acceso no uniforme a memoria).

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

Las figura 10 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 U.M.A.

La figura 11 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 U.M.A.

La figura 12 muestra el esquema de una arquitectura N.U.M.A, 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 10, los cores de cada uno de los procesadores tienen caches de nivel 1 privadas y comparten la memoria cache de nivel 2 y la memoria principal que corresponde al procesador.



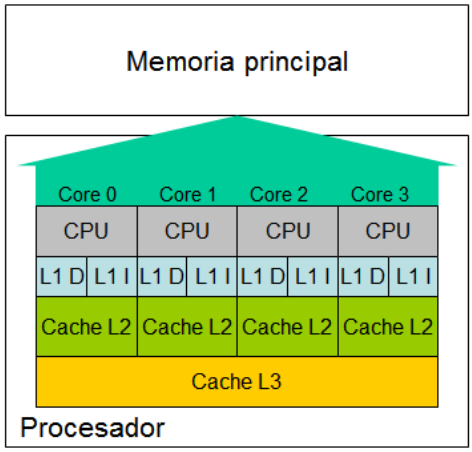




**Figura 10:** dual-core.

**Figura 11:** quad-core.

**Figura 10:** Esquema de procesador dual-core con cache de nivel 2 compartida.
**Figura 11:** Esquema de procesador quad-core con cache de nivel 3 compartida.

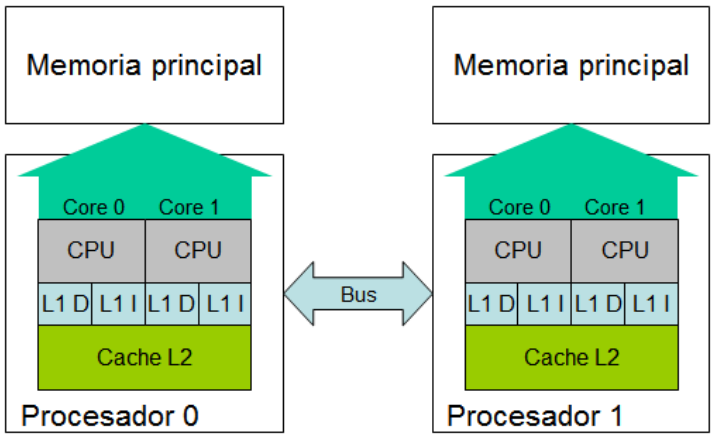



**Figura 12:** Arquitectura N.U.M.A con dos procesadores dual-core. Los cores de cada uno de los procesadores comparten la cache de nivel 2 y el acceso a la memoria principal del procesador.

Las ventajas de compartir la memoria cache son las siguientes:

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

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

- 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.

## 2.5. Factores que impiden un alto rendimiento en sistemas multiCore-multiThread

Por todo lo explicado hasta el momento podría parecer que la solución a la necesidad de seguir aumentando el rendimiento en los procesadores, simplemente se solucionaría aumentando el número de cores de un procesador y/o aumentando el número de procesadores ya sean estos UMA o NUMA.

Si el rendimiento de las aplicaciones en entornos multi-core/multi-thread escalara linealmente al aumentar el número de threads con los que se ejecuta la aplicación, el problema estaría resuelto. Pero lo cierto es que en este tipo de entornos, existen una serie de factores que impiden que esta escalabilidad sea lineal:

- Overheads por creación/eliminación de threads

  Como muestra la figura 13, la creación y posterior eliminación de los threads que trabajan en paralelo, tiene un coste en tiempo (overhead). La importancia de este overhead dependerá de la relación entre el (tiempo total de ejecución del bucle con un thread / nº de threads) y (tiempo que se tarda en crear los threads, repartir el trabajo, recoger el resultado y eliminar los threads)

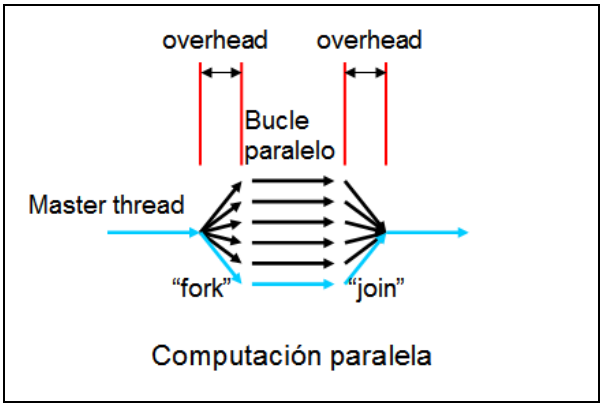



**Figura 13:** Paradigma fork-join con overheads por creación/eliminación de threads.

  El overhead ha de ser pequeño en comparación con (tiempo de ejecución / n threads). Estamos suponiendo que el volumen de cómputo se reparte de forma equitativa. En el apartado siguiente se analiza el caso en el que esto no es así.

- Desbalanceo (en las aplicaciones)

  Una incorrecta distribución del volumen de cómputo por thread implicará que algunos threads finalicen su trabajo antes que otros. Por lo que los threads que han finalizado tendrán que esperar. Esta espera supone un coste en ciclos de reloj del procesador desaprovechados y por lo tanto un overhead.

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

  En el diseño de aplicaciones paralelas es muy importante una óptima asignación del trabajo a realizar a cada uno de los threads.

- Las comunicaciones entre las memorias de los cores

  Los threads de una ejecución multi-thread pueden trabajar de manera independiente y con datos independientes. Pero por las características de las aplicaciones, en algún momento necesitarán intercambiar datos. Este intercambio de datos se realiza de manera transparente al thread ya que éste únicamente accederá a unas posiciones de memoria que previamente otro thread habrá modificado. Esto aunque es transparente para el thread no esta libre de coste en tiempo. Los datos que hayan sido modificados en la cache de un thread tendrán que ser copiados a la cache del thread que los necesita en ese momento.

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

  Estos costes suponen un overhead a considerar a la hora de diseñar una aplicación multithread. 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 threads que se ejecutan en distintos procesadores y segundo minimizar las comunicaciones entre threads que se ejecutan en el mismo procesador.

## 2.6. OpenMP

A continuación 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 una API que soporta programación paralela multi-thread en arquitecturas de memoria compartida. Se puede utilizar con Fortran (77, 90, 95), C, y C++.

Para situar cronológicamente OpenMP se a añadido la figura 14.

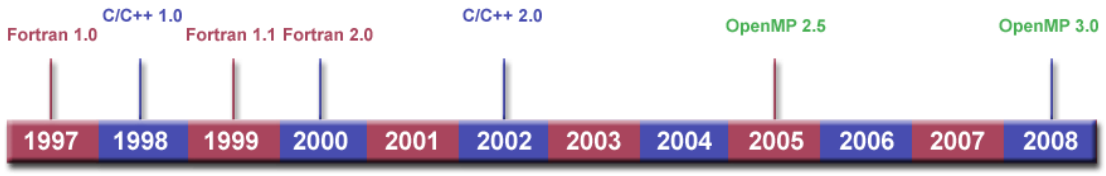



**Figura 14:** Cronología publicada en http://www.openmp.org.

El modelo de programación de OpenMP:

- Esta basado en paralelismo a nivel de thread utilizando memoria compartida.
- Utiliza el paradigma fork-join ( mostrado en la figura 15). Un programa OpenMP empieza con un proceso único (el master thread), que realiza una ejecución secuencial hasta que llega a la primera región paralela, donde creará un conjunto de threads (fork), que se ejecutarán en paralelo. Cuando el conjunto de threads finaliza el trabajo, se sincronizan y terminan (join), quedando únicamente master thread.

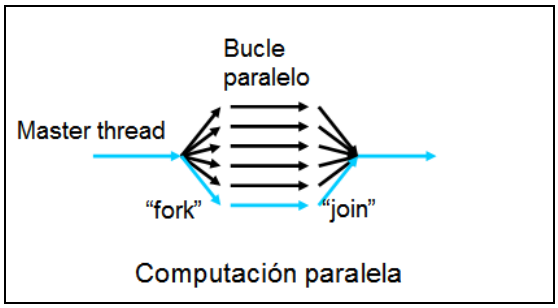



**Figura 15:** Paradigma fork-join.

- Es un modelo explicito en el que el programador tiene un control total sobre la paralelización.
- Basado en directivas de compilación que se incluyen en el código fuente.
- Soporta:
    - Paralelismo a nivel de bucle (paralelismo de datos).
    - Paralelismo funcional.
    - Regiones críticas (exclusión mutua).
    - Regiones single (ejecutadas por un único thread).
    - Variables públicas y privadas.
    - Sincronizaciones explícitas.
    - Repartición estática y dinámica del trabajo.

La sintaxis básica de OpenMP es:

**# pragma omp <directiva> [clausula, …]**

Directivas a destacar

- **parallel**: Indica la parte del código que se podrá ejecutar por varios threads
- **for**: Indica que a cada thread se le asigna una parte del for a ejecutar (paralelismo de datos).
- **sections**: Indica que cada sección será ejecutada por un thread (paralelismo funcional).
- **single**: Indica que esta sección será ejecutada por un único thread.
- **master**: Indica que esta sección será ejecutada por el master-thread.
- **parallel for**: Combinación de parallel (fork) + for (repartición del for entre los threads).
- **parallel sections**: Combinación de parallel (fork) + sections (asignación de cada sección a un thread).
- **critical**: Indica que la sección será de exclusión mutua.
- **barrier**: Indica la necesidad de una sincronización de los threads en este punto.

Cláusulas a destacar

- **schedule** (static | dynamic | guided | runtime [, chunk]): Determina de que forma se realizará la asignación del trabajo a los threads.
- **private** (variable [,variable, …]): Indica que las variables que aparecen en la clausula seran privadas, cada thread tendrá una copia independiente.

Funciones a destacar

- **omp_set_num_threads:** Fija el número de threads que se crearán en los forks de las regiones paralelas de código.
- **omp_get_num_threads:** Devuelve el número de threads activos en la región paralela actual.
- **omp_get_thread_num:** Devuelve el identificador del thread que lo ejecuta.
- **omp_get_num_procs:** Devuelve el número de cores del multi-core en el que se esta ejecutando el thread.

Ejemplos de uso:

- Paralelismo a nivel de for, con variables públicas y privadas.

```
int n,a, b; n=100;
#pragma omp parallel
{   a=0;b=0;
  #pragma omp for private (a)
  for(j=0; j<n; j++) {a++;b++;}
}
```

En este ejemplo la variable n es inicializada por el master thread, mientras que las variables a y b son inicializadas por cada uno de los threads que se crean en el fork (directiva #pragma omp parallel).
Cada thread tendrá una copia de la variable a, donde realizará sus escrituras.
La variable b será compartida y todos los threads escribirán sobre ella.

- Asignación inicial del volumen de cómputo a realizar por cada thread.

```
int j,n; n=100;
#pragma omp parallel
#pragma omp for schedule(static)
for(j=0; j<n; j++) { “cómputo” }
```

En este ejemplo a cada thread le corresponderá ejecutar el for para (n/nº de threads) valores consecutivos de j. En el caso de ejecutar con 4 threads una posible asignación inicial seria:

thread 0 ( 0 < j < 25 )
thread 1 (25 < j < 50 )
thread 2 (50 < j < 75 )
thread 3 (75 < j < 100)

- Asignación dinámica del volumen de cómputo a realizar por cada thread.

```
int j,n;  n=100;
#pragma omp parallel
#pragma omp for schedule(dynamic,10)
for(j=0; j<n; j++) { “cómputo” }
```

En este ejemplo a cada thread se le van asignando dinámicamente grupos de 10 valores consecutivos de j para ejecutar el for. En el caso de ejecutar con 4 threads una posible asignación inicial seria:

thread 0 ( 0 < j < 10 )
thread 1 (10 < j < 20 )
thread 2 (20 < j < 30 )
thread 3 (30 < j < 40)

A partir de este momento al primer thread que finalice su trabajo, se le asignarán (50 < j < 60) y al siguiente que quede libre (70 < j < 80) y así sucesivamente hasta que j = n.

## 2.7. PAPI (Performance Application Programming Interface)

Igual 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.

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.

Algunos de los contadores hardware a los que se puede acceder mediante la utilización de PAPI son los siguientes (presentamos esta lista, para que el lector se haga una ida de todo lo que se puede llegar a medir):

| 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_ICM | Level 3 instruction cache |
| PAPI_L1_TCM | Level 1 cache misses |
| PAPI_L2_TCM | Level 2 cache misses |
| PAPI_L3_TCM | Level 3 cache misses |
| PAPI_CA_SNP | Requests for a snoop |
| PAPI_CA_SHR | Requests for exclusive access |
| PAPI_CA_CLN | Requests for exclusive access |
| PAPI_CA_INV | Requests for cache line |
| PAPI_CA_ITV | Requests for cache line |
| PAPI_L3_LDM | Level 3 load misses |
| PAPI_L3_STM | Level 3 store misses |
| PAPI_BRU_IDL | Cycles branch units are |
| PAPI_FXU_IDL | Cycles integer units are |
| PAPI_FPU_IDL | Cycles floating point units |
| PAPI_LSU_IDL | Cycles load/store units are |
| PAPI_TLB_DM | Data translation lookaside buffer |
| PAPI_TLB_IM | Instruction translation lookaside buffer |
| PAPI_TLB_TL | Total translation lookaside buffer |
| PAPI_L1_LDM | Level 1 load misses |
| PAPI_L1_STM | Level 1 store misses |
| PAPI_L2_LDM | Level 2 load misses |
| PAPI_L2_STM | Level 2 store misses |
| PAPI_BTAC_M | Branch target address cache |
| PAPI_PRF_DM | Data prefetch cache misses |
| PAPI_L3_DCH | Level 3 data cache |
| PAPI_TLB_SD | Translation lookaside buffer shootdowns |
| PAPI_CSR_FAL | Failed store conditional instructions |
| PAPI_CSR_SUC | Successful store conditional instructions |
| PAPI_CSR_TOT | Total store conditional instructions |
| PAPI_MEM_SCY | Cycles Stalled Waiting for |
| PAPI_MEM_RCY | Cycles Stalled Waiting for |
| PAPI_MEM_WCY | Cycles Stalled Waiting for |
| PAPI_STL_ICY | Cycles with no instruction |
| PAPI_FUL_ICY | Cycles with maximum instruction |
| PAPI_STL_CCY | Cycles with no instructions |
| PAPI_FUL_CCY | Cycles with maximum instructions |
| PAPI_HW_INT | Hardware interrupts |
| PAPI_BR_UCN | Unconditional branch instructions |
| PAPI_BR_CN | Conditional branch instructions |
| PAPI_BR_TKN | Conditional branch instructions taken |
| PAPI_BR_NTK | Conditional branch instructions not |
| PAPI_BR_MSP | Conditional branch instructions mispredicted |
| PAPI_BR_PRC | Conditional branch instructions correctly |
| PAPI_FMA_INS | FMA instructions completed |
| PAPI_TOT_IIS | Instructions issued |
| PAPI_TOT_INS | Instructions completed |

| Nombre de contador | Descriptor |
|---|---|
| PAPI_INT_INS | Integer instructions |
| PAPI_FP_INS | Floating point instructions |
| PAPI_LD_INS | Load instructions |
| PAPI_SR_INS | Store instructions |
| PAPI_BR_INS | Branch instructions |
| PAPI_VEC_INS | Vector/SIMD instructions |
| PAPI_RES_STL | Cycles stalled on any |
| PAPI_FP_STAL | Cycles the FP unit(s) |
| PAPI_TOT_CYC | Total cycles |
| PAPI_LST_INS | Load/store instructions completed |
| PAPI_SYC_INS | Synchronization instructions completed |
| PAPI_L1_DCH | Level 1 data cache |
| PAPI_L2_DCH | Level 2 data cache |
| PAPI_L1_DCA | Level 1 data cache |
| PAPI_L2_DCA | Level 2 data cache |
| PAPI_L3_DCA | Level 3 data cache |
| PAPI_L1_DCR | Level 1 data cache |
| PAPI_L2_DCR | Level 2 data cache |
| PAPI_L3_DCR | Level 3 data cache |
| PAPI_L1_DCW | Level 1 data cache |
| PAPI_L2_DCW | Level 2 data cache |
| PAPI_L3_DCW | Level 3 data cache |
| PAPI_L1_ICH | Level 1 instruction cache |
| PAPI_L2_ICH | Level 2 instruction cache |
| PAPI_L3_ICH | Level 3 instruction cache |
| PAPI_L1_ICA | Level 1 instruction cache |
| PAPI_L2_ICA | Level 2 instruction cache |
| PAPI_L3_ICA | Level 3 instruction cache |
| PAPI_L1_ICR | Level 1 instruction cache |
| PAPI_L2_ICR | Level 2 instruction cache |
| PAPI_L3_ICR | Level 3 instruction cache |
| PAPI_L1_ICW | Level 1 instruction cache |
| PAPI_L2_ICW | Level 2 instruction cache |
| PAPI_L3_ICW | Level 3 instruction cache |
| PAPI_L1_TCH | Level 1 total cache |
| PAPI_L2_TCH | Level 2 total cache |
| PAPI_L3_TCH | Level 3 total cache |
| PAPI_L1_TCA | Level 1 total cache |
| PAPI_L2_TCA | Level 2 total cache |
| PAPI_L3_TCA | Level 3 total cache |
| PAPI_L1_TCR | Level 1 total cache |
| PAPI_L2_TCR | Level 2 total cache |
| PAPI_L3_TCR | Level 3 total cache |
| PAPI_L1_TCW | Level 1 total cache |
| PAPI_L2_TCW | Level 2 total cache |
| PAPI_L3_TCW | Level 3 total cache |
| PAPI_FML_INS | Floating point multiply instructions |
| PAPI_FAD_INS | Floating point add instructions |
| PAPI_FDV_INS | Floating point divide instructions |
| PAPI_FSQ_INS | Floating point square root |
| PAPI_FNV_INS | Floating point inverse instructions |

# 3 El Problema de los N cuerpos (N-body)

En este apartado se describirá el problema de los n cuerpos (*n-body*) para sistemas gravitacionales, utilizando el método de resolución conocido como partícula-partícula.

## 3.1. Descripción del problema general

Tenemos un sistema formado por un conjunto de cuerpos. El estado inicial está determinado por las posiciones y las velocidades de estos cuerpos. El problema n-body consiste en simular la evolución del sistema con el paso del tiempo.

Entre cada par de cuerpos interactúa una fuerza que dependerá del tipo de sistema; por ejemplo, astrofísico, plasma, o molecular. Cada cuerpo variará su posición y su velocidad con el paso del tiempo en función de la fuerza total que reciba en cada momento.

El problema n-body surge en diferentes ámbitos científicos, como por ejemplo en:

- Astrofísica: Los cuerpos son planetas, estrellas o galaxias y dependiendo de la escala de la simulación la fuerza que interacciona entre los cuerpos es la gravedad (Ley de Newton)

- Física de plasma: Los cuerpos son iones, electrones, etc. y la fuerza que interacciona entre los cuerpos es la eléctrica (Ley de Coulomb)

- Dinámica molecular: Los cuerpos son átomos o moléculas y la fuerza que interacciona entre los cuerpos es la electrostática (Ley de Coulomb)

Lo que diferencia la resolución del problema n-body en cada uno de estos ámbitos es la fórmula que determina la fuerza que interactúa entre los cuerpos que forman el sistema.

## 3.2. Descripción de n-body partícula-partícula

En este método de resolución, para obtener la siguiente posición y velocidad de cada cuerpo del sistema, es necesario calcular **todas** las fuerzas que interactúan entre **todos** los cuerpos que forman el sistema.

Para obtener las posiciones y velocidades de los cuerpos del sistema durante un periodo de tiempo *T* será necesario realizar una serie de *k* simulaciones. En estas k simulaciones se está calculando el estado del sistema en los instantes de tiempo $t_1$, $t_2$, $t_3$,….$t_k$, (el instante $t_0$ es el del estado inicial). Para cada una de estas *k* simulaciones se calcula la evolución del sistema durante un periodo de tiempo *dt,* donde dt es el tiempo que transcurre entre los instantes de tiempo consecutivos, y donde $k = T / dt$.

Conceptualmente, en cada una de las k simulaciones se realizan dos pasos:

1) Para cada cuerpo se calcula la fuerza que ejercen el resto de cuerpos sobre él, en el instante t, donde t se corresponderá con la simulación que se esta realizando.

2) Se calcula la velocidad y la posición en el instante t + dt de todos los cuerpos a partir de las fuerzas obtenidas en el paso anterior.

En la figura 16 se muestra una representación simplificada del algoritmo de simulación para el problema n-body.

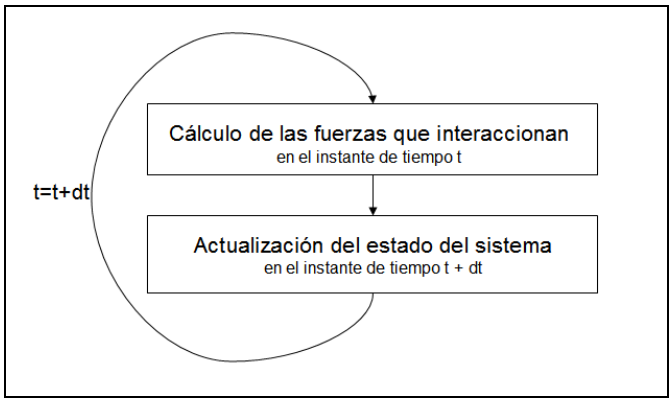



**Figura 16:** Representación simplificada del algoritmo n-body.

Este algoritmo partícula-partícula, también conocido como método directo o método básico, tiene una complejidad de $O(n^2)$ en la cantidad de operaciones que debe realizar para cada iteración y magnitud n en cuanto a necesidades de memoria, siendo n el número de partículas.

Hay otras alternativas matemáticas para resolver el problema n-body que, a diferencia del método anterior, realizan aproximaciones para reducir el orden de magnitud de la complejidad a cambio de asumir un cierto error controlado en los resultados. Algunos ejemplos son el método Barnes-Hut, que tiene complejidad O(*n* log *n*), y el método *Fast Multipole*, con complejidad O(*n*).

## 3.3. Descripción de n-body con interacción de fuerzas de tipo gravitacional

En este caso los cuerpos del sistema cuya evolución interesa simular son planetas y estrellas o galaxias. Cada uno de estos cuerpos tiene una masa y un estado inicial formado por su posición y velocidad iniciales, y entre los cuerpos interactúan fuerzas gravitacionales.

**Descripción matemática:**

Cada cuerpo i del sistema tiene una masa $m_i$, y en el instante t se encuentra ubicado en el espacio en la posición que determina un vector $P_i^t = (Px_i^t, Py_i^t, Pz_i^t)$ y mantiene una velocidad descrita por un vector de velocidad $V_i^t = (Vx_i^t, Vy_i^t, Vz_i^t)$. El estado inicial del sistema es conocido y está determinado por $P_i^{t_0}$, $V_i^{t_0}$ y $m_i$ para cada una de las n partículas.

A continuación se describen las variables utilizadas en el cálculo:

| | |
|---|---|
| $P_i^t = (Px_i^t, Py_i^t, Pz_i^t)$ | : vector de Posición del cuerpo i en el instante de tiempo t |
| $V_i^t = (Vx_i^t, Vy_i^t, Vz_i^t)$ | : vector de Velocidad del cuerpo i en el instante de tiempo t |
| $F_i^t = (Fx_i^t, Fy_i^t, Fz_i^t)$ | : vector de Fuerza que incide sobre el cuerpo i en el instante t |
| $m_i$ | : masa del cuerpo i |
| *dt* | : tiempo que simulamos que pasa en cada iteración |
| *k* | : número de pasos (iteraciones) a simular |
| *G* | : constante gravitacional |

La evolución del sistema (posición y velocidad de los cuerpos que lo forman) se simula durante un periodo de tiempo *T*, dividido en *k* pasos de tiempo *dt*, donde (*k* = *T* / *dt*). Considerando que las fuerzas que interactúan son de tipo gravitacional, la descripción matemática del problema para cada una de las k iteraciones es la siguiente:

$$Fx_i^t = \sum_{j=1, j\neq i}^{n} G \times \frac{m_i \times m_j}{\left(\sqrt{\left(Px_j^t - Px_i^t\right)^2 + \left(Py_j^t - Py_i^t\right)^2 + \left(Pz_j^t - Pz_i^t\right)^2}\right)^3} \times \left(Px_j^t - Px_i^t\right)$$

$$Fy_i^t = \sum_{j=1, j\neq i}^{n} G \times \frac{m_i \times m_j}{\left(\sqrt{\left(Px_j^t - Px_i^t\right)^2 + \left(Py_j^t - Py_j^t\right)^2 + \left(Pz_j^t - Pz_i^t\right)^2}\right)^3} \times \left(Py_j^t - Py_i^t\right)$$

$$Fz_i^t = \sum_{j=1, j\neq i}^{n} G \times \frac{m_i \times m_j}{\left(\sqrt{\left(Px_j^t - Px_i^t\right)^2 + \left(Py_j^t - Py_i^t\right)^2 + \left(Pz_j^t - Pz_i^t\right)^2}\right)^3} \times \left(Pz_j^t - Pz_i^t\right)$$

$$V_i^{t+1} = V_i^t + \frac{F_i^t}{m_i} \times dt$$

$$P_i^{t+1} = P_i^t + V_i^{t+1} \times dt$$

**Figura 17:** Descripción matemática del problema para una iteración.

Las fórmulas anteriores se aplican de forma consecutiva para calcular los valores sucesivos de $P_i^t$ y $V_i^t$, donde t = $t_1$, $t_2$, $t_3$,….$t_k$

Esta descripción matemática es el punto de partida para el desarrollo de los algoritmos que se presentan a continuación.

# 4 Diseño del algoritmo serie para el problema n-body

En este capítulo se acometerá el diseño del algoritmo serie para el problema de los n cuerpos. Se analizarán varios algoritmos (o variantes del algoritmo) para tratar de encontrar aquél que dé lugar a una implementación cuyo tiempo de ejecución en un cierto procesador sea mínimo. Para ello, se dividirá el análisis en dos partes. En la primera parte, los algoritmos se analizarán de forma independiente al procesador. En la segunda parte, los algoritmos se implementarán en lenguaje C y se compilarán y ejecutarán en cada uno de los procesadores considerados, tomando medidas de su rendimiento mediante la lectura de los contadores H/W del procesador. Tras este análisis se identificarán las características del programa y las características de los procesadores que más influyen el rendimiento.

El algoritmo que se tomará como punto de partida es el que se obtiene directamente de la expresión matemática mostrada en la figura 17 del capítulo anterior. El algoritmo resultante es muy ineficiente, y se le aplicarán una serie de optimizaciones clásicas, sencillas, que podrían ser realizadas por un compilador, para reducir la cantidad de operaciones a realizar (volúmen de cómputo y de accesos a memoria). El algoritmo final resultante dará lugar a un análisis en mayor profundidad, que llevará a dos líneas diferentes de actuación.

En la primera línea de análisis, el objetivo principal es minimizar los requerimientos de memoria. Al algoritmo resultante le denominamos Nbody-A (Aceleraciones), porque se basa en calcular las aceleraciones provocadas entre cada par de partículas, y usarlas directamente para actualizar las velocidades y posiciones de las partículas.

En la segunda línea de trabajo, el objetivo principal será minimizar el volumen de cómputo. Al algoritmo resultante le denominamos Nbody-F (Fuerzas), porque calcula las fuerzas ejercidas entre cada par de partículas, y luego las usa para actualizar las velocidades y posiciones de las partículas.

En este capítulo se considerará la ejecución sobre un único núcleo (*core*) del procesador. En el capítulo siguiente se paralelizarán las dos variantes algorítmicas consideradas (A y F), para dividir la ejecución en múltiples threads que puedan ser ejecutados en un computador paralelo. En la implementación se utilizará OpenMP y un modelo de paralelismo de tipo *fork-join* con variables compartidas. En este caso los algoritmos se ejecutarán en forma multi-thread sobre varios núcleos del procesador, y se medirá y analizará el rendimiento variando el número de threads y de núcleos utilizados durante la ejecución.

## 4.1. Análisis del algoritmo independiente del procesador

La primera parte del análisis de algoritmos que proponemos abstrae los detalles concretos que diferencian unos procesadores de otros. En este caso se necesita un modelo de rendimiento general y, suficientemente simple como para permitir comparar alternativas de forma rápida, pero a la vez suficientemente complejo para capturar los detalles más influyentes en el rendimiento.

**Procesador: Modelo Abstracto**

Para extraer características cuantitativas del algoritmo que sean significativas en el rendimiento de su ejecución en un procesador, es necesario considerar un cierto modelo de rendimiento abstracto de los procesadores. Básicamente, el tiempo de ejecución depende de:

- El volúmen de cómputo que genera el programa, que es el nº total de operaciones ejecutadas por el algoritmo, agrupadas por tipos.
- Los requerimientos de memoria necesarios para almacenar los datos del programa.

En una primera aproximación, un algoritmo será mejor cuanto menor sea el volúmen de cómputo y los requerimientos de memoria. Sin embargo, no todas las operaciones de cómputo son igual de costosas en tiempo. Se puede considerar que algunas operaciones no realizan cómputo, como las lecturas y escrituras a memoria, aunque este tipo de operaciones pueden llegar a ser muy costosas.

Para tener un modelo más preciso del algoritmo es necesario distiguir una serie de tipos de operaciones, en función del coste en tiempo de ejecución que suponen en la mayoría de los procesadores. Una posible clasificación de estas operaciones que denominamos *básicas* es la siguiente:

- **Punto-flotante:**
  - Suma/resta/comparación (SUMApf)
  - Multiplicación (MULTpf)
  - División (DIVpf)
  - Raíz cuadrada (RAIZpf)

- **Enteras:**
  - Suma/resta/comparación (SUMAent)
  - Multiplicación (MULTent)
  - División (DIVent)

- **Saltos:**
  - Condicional (SLTcnd)
  - Incondicional (SLTinc)

- **Accesos a memoria:**
  - Lecturas (LecMem)
  - Escrituras (EscMem)

Las operaciones se identifican en la descripción algorítmica (lenguaje C, en nuestro caso). En cada procesador y compilador concreto, la correspondencia entre operaciones *básicas* del algoritmo y operaciones *máquina* o *primitivas* puede ser diferente. Una operación básica se puede transformar en múltiples operaciones máquina (como la operación raíz cuadrada en el procesador SPARC T2). Alternativamente, varias operaciones básicas se pueden transformar en una única operación máquina (como la operación multiplicar-acumular, que esta disponible en muchos procesadores).

Además del volumen de operaciones básicas, clasificadas por tipos, hay al menos tres características adicionales que son muy importantes en el rendimiento:

- Los caminos de *dependencias de datos* entre operaciones básicas, que determinan el grado de paralelismo a nivel de instrucciones (ILP)
- El *patrón* de las operaciones de acceso a memoria:
  - Tamaño de la zona accedida por las instrucciones de acceso a memoria
  - Orden de acceso a los datos dentro de cada zona:
    - Acceso regular: Secuencial (stride=1), No secuencial (stride>1)
    - Acceso Irregular
- El *patrón* de comportamiento de las operaciones de salto condicional:
  - Comportamiento repetitivo regular de resultados Salta (S) y No Salta (NS)
  - Comportamiento irregular, dependiente de los datos

**Algoritmo: Tamaño del Problema y Complejidad**

El primer paso para modelar el algoritmo es identificar los parámetros que determinan el tamaño del problema. Para n-body, el tamaño del problema (*Problem Size*) se define por:

- ***n***: número de cuerpos que forman el sistema
- ***k***: número de pasos (o iteraciones) a simular, (***k*** = ***T*** / ***dt***)

El segundo paso para modelar el algoritmo consiste en calcular la complejidad del algoritmo en función del tamaño del problema. En los algoritmos n-body que se van a describir, basados en el método directo descrito en el capítulo anterior, la complejidad es siempre de $O(k \times n^2)$ operaciones de cómputo. Esta complejidad corresponde al cómputo que hay que realizar entre cada pareja de partículas (en total $n \times (n-1)$ parejas), repetido para cada uno de los ***k*** pasos de simulación.

Este resultado supone un modelo simple del comportamiento del algoritmo, ya que la complejidad es independiente de los datos concretos de entrada que se utilicen para definir las posiciones y velocidades iniciales de las ***n*** partículas.

Podemos considerar que el algoritmo realiza $k \times n^2$ "*operaciones abstractas*", cada una de ellas compuesta por un cierto número de operaciones básicas de cómputo y acceso a memoria. En este caso, una "operación abstracta *n-body*" se puede definir en términos del problema como: "el cálculo de la fuerza que provoca una partícula en otra partícula en un instante de tiempo determinado". Esta definición variará ligeramente al modificar alguno de los algoritmos (se calcularán aceleraciones en lugar de fuerzas).

La definición de una operación repetitiva, que contiene la esencia del algoritmo, permite analizarlo y comprender los resultados de la ejecución de una forma más simple y dirigida, tal como se mostrará más adelante. En ciertos algoritmos muy irregulares, con muchas operaciones condicionales, puede no ser posible identificar una operación abstracta simple que se ejecute repetidamente. En estos casos, el modelo de la aplicación que se propone a continuación debería variar.

Para nuestro estudio consideraremos valores de ***n*** entre 100 y 200.000 y valores de ***k*** igual o mayores a 1. Los requerimientos de almacenamiento del algoritmo crecen de forma lineal con el valor de ***n*** (como se verá más adelante). Con ***n***= 200.000, la cantidad de memoria necesaria para almacenar los datos es algo mayor de 10 MBytes. Con los procesadores que se han considerado, este valor es suficiente para analizar el rendimiento cuando los datos no caben en la memoria caché del procesador, y hay que obtenerlos de la memoria principal. Se necesita un valor mucho más grande de ***n*** para forzar al sistema a tener que utilizar de forma frecuente el almacenamiento secundario. Como los tiempos de ejecución crecen con $n^2$, se hace inviable realizar experimentos con valores tan grandes.

En cambio, los requerimientos de almacenamiento no dependen de ***k***, y el tiempo de ejecución crece de forma perfectamente lineal respecto a ***k***, tal y como se verá más adelante. No es necesario hacer ejecuciones con valores de ***k*** grandes, y aún así se puede estimar perfectamente el rendimiento con cualquier valor de ***k***.

## 4.2. Algoritmo serie inicial

A continuación se muestran las variables utilizadas para representar el problema n-body y el pseudo-código de la primera aproximación del algoritmo serie, basado directamente en la descripción matemática mostrada en la figura 17 del apartado 3.3. Como el algoritmo resultante es muy ineficiente, posteriormente se le aplicarán una serie de optimizaciones clásicas, sencillas, para reducir la cantidad de operaciones primitivas a realizar. En el pseudo-código se ha añadido a la derecha el recuento de operaciones básicas dentro de dos zonas diferentes (zona 1 y zona 2), que se comentarán a continuación**:**

**Variables:**

| | |
|---|---|
| Px [1..n], Py [1..n], Pz [1..n] | vectores de posición |
| Vx [1..n], Vy [1..n], Vz [1..n] | vectores de velocidad |
| Fx [1..n], Fy [1..n], Fz [1..n] | vectores de fuerzas |
| m [1..n] | vector de masas |
| dist | distancia euclídea |
| G | constante gravitacional |
| dt | tiempo transcurrido por iteración de la simulación |

**Para S=1..k** **(Para cada iteración de tiempo de paso dt)**

**{**

**Para i=1..n** **(Para cada cuerpo i)**

**{**

Fx [i] = Fy [i] = Fz [i] = 0 **Zona 2**

Para j=1..n (siendo i diferente de j) (Para cuerpos diferentes al cuerpo i)

{

$dist = \sqrt{ (Px[j] - Px[i])^2 + (Py[j] - Py[i])^2 + (Pz[j] - Pz[i])^2 }$

$Fx[i] = Fx[i] + (G \times m[i] \times m[j]) \times (Px[j] - Px[i]) / dist^3$

$Fy[i] = Fy[i] + (G \times m[i] \times m[j]) \times (Py[j] - Py[i]) / dist^3$

$Fz[i] = Fz[i] + (G \times m[i] \times m[j]) \times (Pz[j] - Pz[i]) / dist^3$

**Zona 1**

}

k×n×(n-1) veces

**}**

k×n veces

**Para i=1..n** **(Para cada cuerpo i)**

**{**

$Vx[i] = Vx[i] + dt \times Fx[i] / m[i]$ **Zona 2**

$Vy[i] = Vy[i] + dt \times Fy[i] / m[i]$

$Vz[i] = Vz[i] + dt \times Fz[i] / m[i]$

$Px[i] = Px[i] + dt \times Vx[i]$

$Py[i] = Py[i] + dt \times Vy[i]$

$Pz[i] = Pz[i] + dt \times Vz[i]$

}

k×n veces

**}**

| Zona 1: k×n×(n-1) | |
|---|---|
| SUMApf: | 11 |
| MULTpf: | 9 |
| DIVpf: | 3 |
| Elevar a 2: | 3 |
| Elevar a 3: | 3 |
| RAIZpf: | 1 |
| LecMem: | 21 |
| EscMem: | 3 |
| SLTcond: | 2 |

| Zona 2: k×n | |
|---|---|
| SUMApf: | 6 |
| MULTpf: | 6 |
| DIVpf: | 3 |
| LecMem: | 15 |
| EscMem: | 9 |
| SLTcond: | 2 |

Consideramos que las variables escalares (por ejemplo, las variables *dist* y *dt*) se corresponderán en la implementación del programa con registros de propósito general del procesador. Así, los accesos a estas variables supondremos que no requieren la ejecución de operaciones de acceso a memoria. Esto sólo será cierto si el número de registros del procesador es suficiente para contener todas las variables escalares que se están utilizando durante ciertos periodos de tiempo.

Por otro lado, se considera que los vectores (por ejemplo, *Px* y *m*) se almacenan en la memoria, y no en registros, y que por lo tanto todos los accesos a un vector suponen operaciones de acceso a memoria. Así, la necesidad de memoria es, básicamente, de $3 \times n + 3 \times n + 3 \times n + n = 10n$ valores de tipo *double* (3 vectores para la posición, 3 para la velocidad, 3 para la fuerza y uno para la masa). Como cada elemento ocupa 8 bytes de memoria, los requerimientos totales de memoria son de $80n$ bytes. Para $n$=200.000, los requerimientos son de 16 MBytes.

Las operaciones en el interior del bucle mostrado dentro del recuadro etiquetado como Zona 1 se realizan $k \times n \times (n-1)$ veces. En cambio, las operaciones dentro de los dos recuadros etiquetados como Zona 2 se realizan $k \times n$ veces. La complejidad del algoritmo corresponde con la del bucle de mayor complejidad, que tal como se indicó anteriormente es $\theta(k \times n^2)$, y corresponde con la ejecución de la Zona 1. Cada ejecución de las operaciones de la Zona 1 corresponde, además, con la "operación abstracta" que se definió en la sección anterior (cálculo de la fuerza que ejerce la partícula j sobre la partícula ).

Al contar el número de operaciones no se han tenido en cuenta las operaciones con enteros que se realizan en el cálculo de direcciones de memoria, ni las que se realizan para incrementar los contadores de los bucles. Esta cantidad es proporcionalmente pequeña, y podemos considerar que la ejecución de las operaciones enteras se solapa con la ejecución de operaciones en punto flotante y de acceso a memoria, que son las que determinan el tiempo de ejecución del programa.

A continuación se describe la serie de optimizaciones sencillas que se aplican para reducir la cantidad de operaciones a realizar por el algoritmo:

1. ***loop-invariant code motion:*** *consiste en mover instrucciones fuera del cuerpo del bucle sin afectar la semántica del programa. El código movido fuera del cuerpo del bucle (cálculo o acceso a memoria) se ejecuta con menos frecuencia, reduciendo el tiempo total de ejecución. Los valores calculados o leídos de memoria fuera del bucle, y que son invariantes dentro del bucle, se almacenan en variables locales, que potencialmente pueden ser asignadas a registros*

    a. **Pxi = Px[i]; Pyi = Py[i]; Pzi = Pz[i];** La posición del cuerpo i se lee una vez fuera del bucle y se guarda en variables locales (registros)

    b. **Fxi = Fyi = Fzi = 0;** El vector de fuerzas del cuerpo i se va acumulando usando variables locales (registros) dentro del bucle, y al final del bucle se guardan sus valores en el vector

    c. **MGi = m[i]×G;** Esta expresión se calcula fuera del bucle

2. ***Common subexpression elimination*** *(CSE): busca casos de expresiones idénticas (que se evalúen siempre al mismo valor y sin efectos colaterales) y se analiza si es conveniente reemplazarlas por una única variable que guarde el valor de la expresión calculado una única vez:*

    a. **dx = Px [j] – Pxi; dy= Py [j] – Pyi; dz= Pz [j] – Pzi;** Cada expresión se usa dos veces, así que vale la pena guardarla temporalmente

    b. **k = MGi × m [j] / dist$^3$;** La expresión se usa tres veces

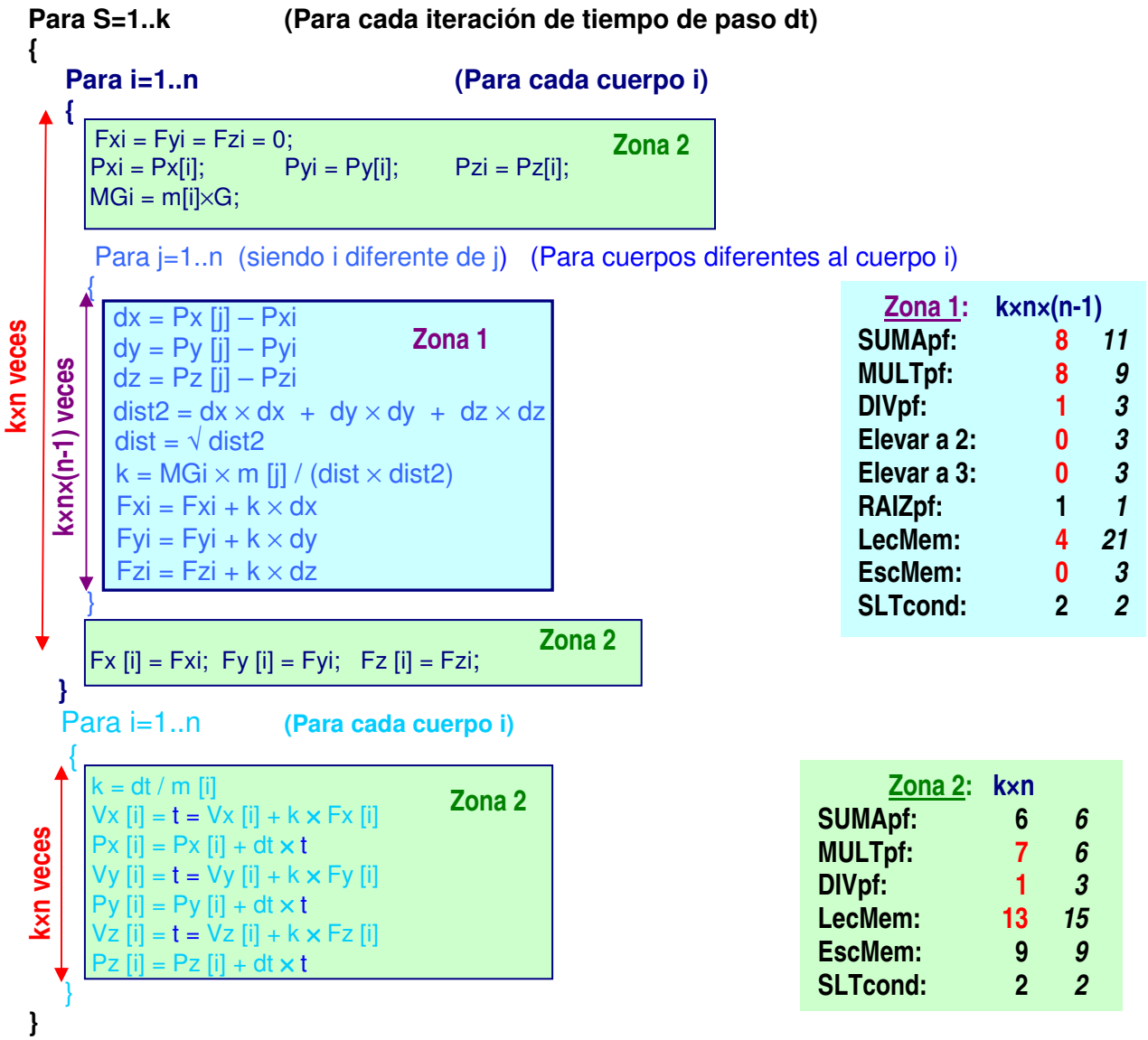
    c. **t = Vz [i] = ... ; Pz [i] = Pz [i] + dt × t;** Primero se escribe en Vz[i] y luego se lee: se guarda el valor en un registro t para no tener que volver a leer.

3. ***Strength reduction****: reemplaza una operación potencialmente costosa por una o varias equivalentes, pero menos costosas. Se utilizan identidades matemáticas, y se debe tener en cuenta las limitaciones de las operaciones en coma flotante, que no son asociativas:*

    a. **dx × dx; dy × dy; dz × dz;** En lugar de la operación de elevar al cuadrado

    b. como **dist = √dist2** se puede usar **dist × dist2** en lugar de **dist$^3$**

Seguidamente se muestra el pseudo-código final del algoritmo serie inicial, después de aplicar las diferentes técnicas de optimización descritas**:**




| Zona 1: | kxnx(n-1) | |
|---|---|---|
| SUMApf: | 8 | *11* |
| MULTpf: | 8 | *9* |
| DIVpf: | 1 | *3* |
| Elevar a 2: | 0 | *3* |
| Elevar a 3: | 0 | *3* |
| RAIZpf: | 1 | *1* |
| LecMem: | 4 | *21* |
| EscMem: | 0 | *3* |
| SLTcond: | 2 | *2* |

| Zona 2: | kxn | |
|---|---|---|
| SUMApf: | 6 | *6* |
| MULTpf: | 7 | *6* |
| DIVpf: | 1 | *3* |
| LecMem: | 13 | *15* |
| EscMem: | 9 | *9* |
| SLTcond: | 2 | *2* |

Las optimizaciones han reducido el volumen de operaciones de cómputo y de acceso a memoria de forma considerable. A cambio, se necesitan más registros disponibles de forma simultánea, para almacenar un mayor número de variables temporales y escalares (dx, dy, dz, dist2, k, Fxi, Fyi, Fzi, Pxi, Pyi, Pzi)

## 4.3. Algoritmo serie: con cálculo de aceleraciones (nbody-A)

La siguiente variante algorítmica incluye dos optimizaciones y se basa en el cálculo de aceleraciones. Además de reducir ligeramente el volumen de operaciones de cómputo y de acceso a memoria, reduce los requisitos de memoria. A cambio, se necesita un registro más.

En primer lugar, se avanza la actualización de las velocidades desde el 2º bucle al 1º. De este modo, la fuerza que recibe la partícula i, calculada en el 1er bucle, se utiliza inmediatamente para actualizar la velocidad de la partícula i. Así, **no es necesario** almacenar las fuerzas en un vector, y se reducen los requerimientos de memoria.

En el algoritmo original, al calcular las fuerzas que recibe la partícula i, se multiplican los factores por la masa de la partícula i. Después, al modificar la velocidad, la fuerza se divide por la masa de la partícula i. La segunda optimización propuesta en este apartado consiste en evitar esta multiplicación y luego división. En lugar de calcular la fuerza total que recibe la partícula i, se calculará lo que se denomina la *aceleración recíproca* total sobre la partícula i.

A continuación, se muestran las variables utilizadas para representar el problema n-body y el pseudo-código del algoritmo serie, después de aplicar estas dos optimizaciones.

**Variables:**

| | |
|---|---|
| Px [1..n], Py [1..n], Pz [1..n] | vectores de posición |
| Vx [1..n], Vy [1..n], Vz [1..n ] | vectores de velocidad |
| m [1..n] | vector de masas |
| dist, dist2 | distancia euclídea y su valor al cuadrado |
| ac_rec, Axi, Ayi, Azi, dx, dy, dz | variables de tipo coma flotante |
| Pxi, Pyi, Pzi | variables de tipo coma flotante |
| dt | tiempo transcurrido por iteración de la simulación |
| Gdt | constante gravitacional multiplicada por dt |

```
Gdt=G×dt;
Para t=1..k          (Para cada iteración de tiempo de paso dt)
{ Para i=1..n        (Para cada cuerpo i)
  {
    Axi = Ayi = Azi = 0                          Zona 2
    Pxi = Px[i];   Pyi = Py[i];   Pzi = Pz[i];
    Para j=1..n (siendo i diferente de j)   (Para cuerpos diferentes al cuerpo i)
    {
      dx = Px [j] – Pxi                          Zona 1
      dy = Py [j] – Pyi
      dz = Pz [j] – Pzi
      dist2 = dx × dx + dy × dy + dz × dz
      dist = √ dist2
      ac_rec = m [j] / (dist × dist2)
      Axi = Axi + ac_rec × dx
      Ayi = Ayi + ac_rec × dy
      Azi = Azi + ac_rec × dz
    }
    Vx [i] = Vx [i] + Gdt × Axi                  Zona 2
    Vy [i] = Vy [i] + Gdt × Ayi
    Vz [i] = Vz [i] + Gdt × Azi
  }
  Para i=1..n        (Para cada cuerpo i)
  {
    Px [i] = Px [i] + dt × Vx [i]                Zona 2
    Py [i] = Py [i] + dt × Vy [i]
    Pz [i] = Pz [i] + dt × Vz [i]
  }
}
```

kxn veces — kxnx(n-1) veces — kxn veces

| Zona 1: | kxnx(n-1) | |
|---|---|---|
| SUMApf: | 8 | *8* |
| MULTpf: | 7 | *8* |
| DIVpf: | 1 | *1* |
| RAIZpf: | 1 | *1* |
| LecMem: | 4 | *4* |
| EscMem: | 0 | *0* |
| SLTcond: | 2 | *2* |

| Zona 2: | kxn | |
|---|---|---|
| SUMpf: | 6 | *6* |
| MULTpf: | 6 | *7* |
| DIVpf: | 0 | *1* |
| LecMem: | 12 | *13* |
| EscMem: | 6 | *9* |
| SLTcond: | 2 | *2* |

La necesidad de memoria se reduce de ***10n*** valores de tipo *double* a ***7n*** valores (no son necesarios los 3 vectores para las fuerzas). A continuación en la figura 18, se muestra una tabla resumen de las necesidades de volumen de cómputo, clasificadas por tipo, del algoritmo nbody-A.

| | **Zona 1: cálculo de aceleraciones** | **Zona 2: Actualización de V y P** |
|---|---|---|
| **Complejidad:** | **k×n×(n-1)** | **k×n** |
| SUMAent * | **0 *** | **0 *** |
| SLTcond | **2** | **2** |
| LecMem | **4** | **12** |
| EscMem | **0** | **6** |
| SUMApf | **8** | **6** |
| MULTpf | **7** | **6** |
| DIVpf | **1** | **0** |
| RAIZpf | **1** | **0** |

**Figura 18:** Número de operaciones clasificadas por tipo para dos zonas del algoritmo nbody-A. Para cada zona se muestra la complejidad (número de veces que se ejecutan instrucciones en esa zona). Las operaciones marcadas con * no se han considerado.

**Dependencias de Datos**

Los procesadores actuales son capaces de ejecutar varias operaciones de forma simultánea, siempre y cuando no existan dependencias de datos entre las operaciones. Para evaluar la posibilidad de ejecutar múltiples operaciones en paralelo se deben analizar las dependencias de datos potencialmente críticas. Para afectar al rendimiento, las dependencias deben darse en la zona del algoritmo de mayor complejidad (la Zona 1 en este ejemplo), y deben involucrar una secuencia de dependencias de una latencia superior al tiempo necesario para ejecutar en paralelo todas las instrucciones. Como en este punto del análisis no disponemos de datos concretos de latencia, se deben analizar las dependencias que involucren más operaciones.

En la figura 19 de la página siguiente se muestran las dependencias de datos entre las instrucciones, resaltando en amarillo los accesos a memoria y en azul las operaciones y los accesos a variables. Las flechas indican el flujo de los datos. El cuadro azul más interno muestra la zona potencialmente más crítica.

Las cadenas de dependencias pueden mostrarse de forma textual tal y como se indica a continuación, para la cadena más larga dentro del bucle de la Zona 1:

{ LecMem, LecMem, LecMem } →
{ SUMApf, SUMApf, SUMApf } →
{ MULTpf, MULTpf } →
SUMApf →
SUMApf →
RAIZpf →
MULTpf →
DIVpf →
{ MULTpf, MULTpf, MULTpf } →
{ SUMApf, SUMApf, SUMApf }

Para determinar qué dependencias afectan al tiempo de ejecución es necesario conocer los valores de tiempo de las operaciones primitivas del procesador y la capacidad de ejecución de cada tipo de operación. No sólo eso: si el procesador ejecuta instrucciones fuera de orden, el tamaño de las colas de instrucciones y de los registros físicos de renombrado, y las políticas de planificación de instrucciones de bajo nivel pueden tener una influencia determinante en el rendimiento.

Los otros dos factores importantes que afectan al rendimiento son el comportamiento de las operaciones de acceso a memoria y de las operaciones de salto.

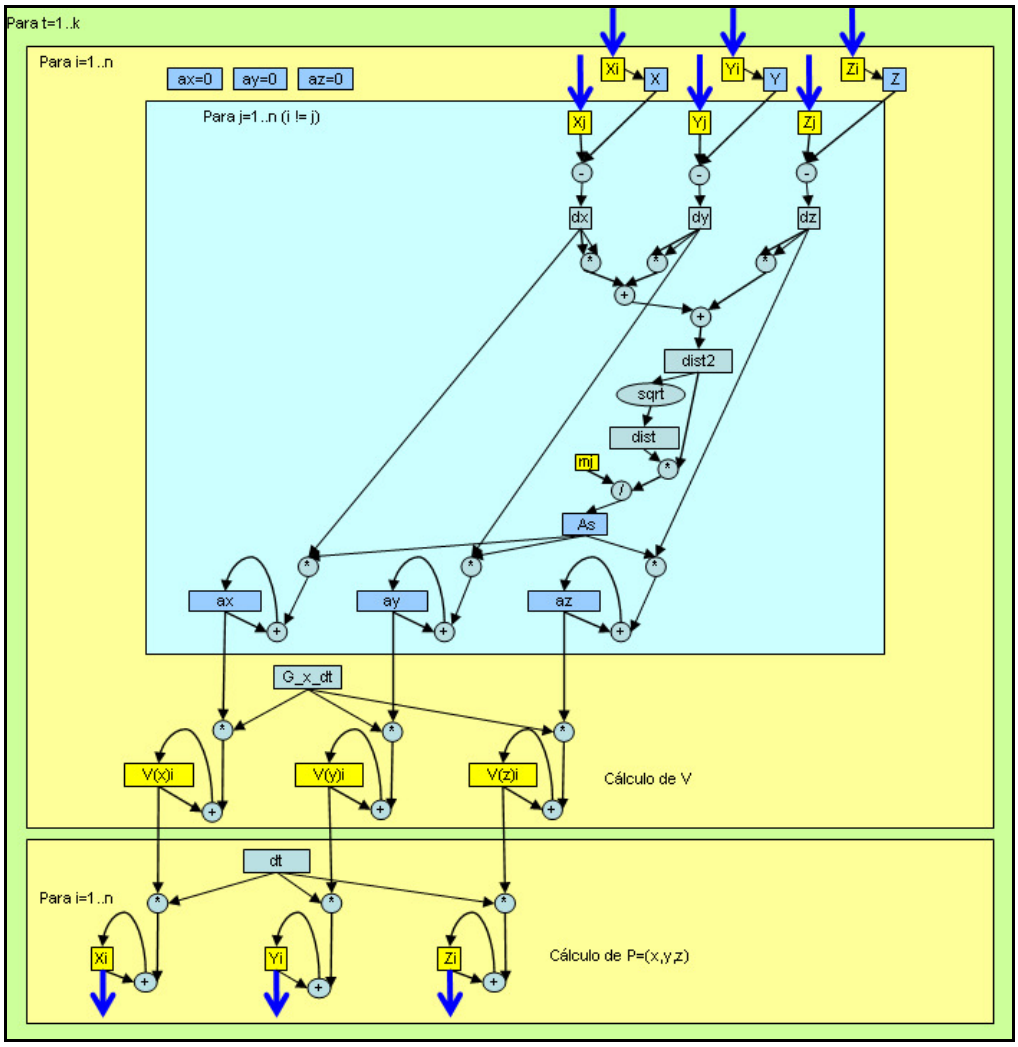



**Figura 19:** Caminos de dependencias de datos del algoritmo serie nbody-A.

### Operaciones de Acceso a Memoria

Si nos centramos en la Zona 1 del algoritmo, observamos que hay 4 operaciones de lectura (LecMem) y ninguna operación de escritura (EscMem). Hemos de identificar los patrones de acceso de las 4 instrucciones LecMem, que denominaremos *LecMem.Px, LecMem.Py, LecMem.Pz,* y *LecMem.m*.

Cada una de las 4 operaciones se ejecuta aproximadamente $\boldsymbol{k}\times\boldsymbol{n}^2$ veces. Además, en cada uno de los casos se accede ***n*** veces a ***n*** posiciones inmediatamente consecutivas (*stride*= 1). El patrón de todas las lecturas se puede definir de la siguiente forma:

***k*** × ***n*** repeticiones × { acceso secuencial, ***stride***=1, sobre ***n*** datos }

Como veremos más adelante, el patrón de acceso secuencial es el más favorable al rendimiento en los procesadores. En realidad, como se trata de un patrón de acceso muy común, los procesadores están optimizados para ser eficientes con este tipo de accesos.

### Operaciones de Salto Condicional

Hay dos saltos condicionales en la Zona 1 del algoritmo, que denominaremos *SLTcond.if* y *SLTcond.for*, ya que el primero determina si el índice *j* que controla el bucle interno es diferente del índice *i*, y el segundo controla que el índice *j* llegue al valor ***n***. Ambos saltos se ejecutan $\boldsymbol{k}\times\boldsymbol{n}^2$ veces. El salto *SLTcond.if* salta en casi todos los casos una vez de cada ***n***+1 ejecuciones, excepto en ***k***-1 ocasiones, que salta dos veces seguidas. El salto *SLTcond.for* repite el siguiente patrón de comportamiento: ***n***-1 saltos seguidos de un caso en que no salta,

esto repetido ***k*×*n*** veces. El patrón de saltos se puede definir de la siguiente forma (S = Saltar, N = No saltar):

***SLTcond.for:*** ***k*** × ***n*** veces × { S, S, S, *(n-1 veces) ...* , N }

***SLTcond.if:*** ***k*** veces × { S, ***n-1*** veces ×{ N, N, N, *(n veces) ...* , S } }

Como también se verá más adelante, un patrón de saltos en el que la mayor parte de las veces el resultado es el mismo (*SLTcond.for= S el (n-1)/n% de las veces; SLTcond.if= N el (n-1)/n% de las veces*) es muy favorable para el rendimiento de los procesadores que disponen de algún mecanismo básico de predicción de saltos. Como en el caso de los accesos secuenciales a memoria, los procesadores están optimizados para ser eficientes con este tipo de patrones de comportamiento de saltos.

## 4.4. Algoritmo serie: con cálculo optimizado de fuerzas (nbody-F)

La alternativa A mejora el algoritmo serie inicial minimizando el número de operaciones y de accesos a memoria, y los requisitos de memoria. Sin embargo, se puede reducir aún más el volumen de operaciones y accesos a memoria si se observa que los cálculos de las fuerzas o aceleraciones entre pares de partículas están "repetidos" dos veces.

La fuerza que ejerce la partícula i sobre la partícula j (suponiendo que i y j son diferentes) es la misma, pero de signo contrario, que la que ejerce la partícula j sobre la partícula i. De esta manera, sólo es necesario calcular la fuerza de i sobre j y utilizar el valor negado para obtener la fuerza de j sobre i. Es decir, la cantidad de operaciones y accesos a memoria se puede reducir aproximadamente a la mitad.

En la figura 20 se muestran las operaciones necesarias para calcular la fuerza que recibe un cuerpo, si únicamente se realizan la mitad de los cálculos. Por ejemplo, en verde se ha marcado que para calcular la fuerza que recibe la particula 3 (i=2) será necesario realizar las siguientes operaciones: $f_x(0,2)$, $f_x(1,2)$, $f_x(2,3)$, $f_x(2,4)$, en concreto el sumatorio se debería realizar de la siguiente forma: $-f_x(0,2) - f_x(1,2) + f_x(2,3) + f_x(2,4)$ que es igual a $f_x(2,0) + f_x(2,1) + f_x(2,3) + f_x(2,4)$

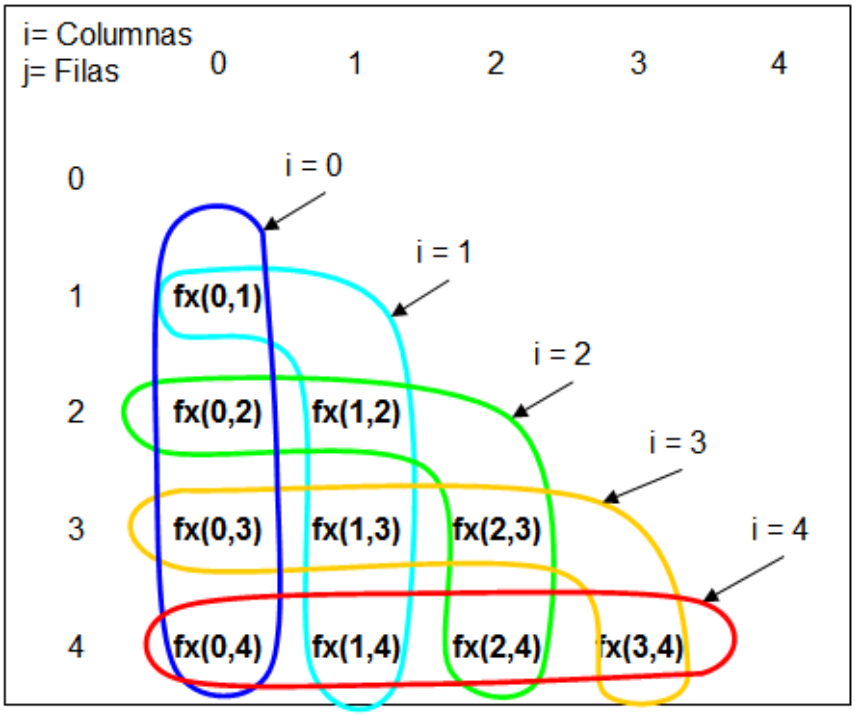



**Figura 20:** Agrupaciones de cálculos de $f_x(i,j)$ con i < j.

Almacenando los valores de las fuerzas en una matriz **n×n** para cada una de las coordenadas, y accediendo a ellos de la manera mostrada en la figura 20, se reducen las operaciones a la mitad, pero se necesita espacio de memoria para 3 **×** **n** **×** **n** valores de tipo *double*. En lugar de utilizar estas matrices, se pueden utilizar tres vectores ($ff_x$, $ff_y$, $ff_z$) de tamaño ***n*** en los que ir almacenando los valores parciales de los sumatorios de las fuerzas que inciden en cada uno de los ***n*** cuerpos. A continuación, se muestran las variables y el pseudo-código del algoritmo serie del algoritmo nbody-F.

## Variables:

| | |
|---|---|
| Px [1..n], Py [1..n], Pz [1..n] | vectores de posición |
| Vx [1..n], Vy [1..n], Vz [1..n] | vectores de velocidad |
| m [1..n] | vector de masas |
| ffx [1..n], ffy [1..n], ffz [1..n] | vectores de fuerzas |
| dist, dist2 | distancia euclídea y valor al cuadrado |
| mi, dx, dy, dz, Pxi, Pyi, Pzi | variables de tipo coma flotante |
| Fs, Fsdx, Fsdy, Fsdz, Gdtm | variables de tipo coma flotante |
| dt | tiempo transcurrido por iteración de la simulación |
| Gdt | constante gravitacional x dt |

```
Gdt=G×dt;
Para t=1..k          (Para cada iteración de tiempo de paso dt)
{
  Para i=1..n                                   (Para cada cuerpo i)
  {
    Pxi = Px[i];   Pyi = Py[i];   Pzi = Pz[i]
    fx = fy = fz = 0
    mi = m[i];                                  Zona 2

    Para j=i+1...n
    {
      dx = Px [j] – Pxi                         Zona 1
      dy = Py [j] – Pyi
      dz = Pz [j] – Pzi
      dist2 = dx × dx  +  dy × dy  +  dz × dz
      dist = √ dist2
      Fs =  m [j] × mi / (dist × dist2)
      Fsdx = Fs × dx
      Fsdy = Fs × dy
      Fsdz = Fs × dz
      ffx[j] = ffx[j] – Fsdx;    fx = fx + Fsdx
      ffy[j] = ffy[j] – Fsdy;    fy = fy + Fsdy
      ffz[j] = ffz[j] – Fsdz;    fz = fz + Fsdz
    }   // k×n×(n-1)/2 veces

    ffx[i] = ffx[i] + fx
    ffy[i] = ffy[i] + fy                        Zona 2
    ffz[i] = ffz[i] + fz
  }   // k×n veces

  Para i=0..n
  {
    Gdtm = Gdt / m[i]                           Zona 2
    vx[i] = t =vx[i] + ffx[i] x Gdtm
    px[i] = px[i] + t x dt
    vy[i] = t =vy[i] + ffy[i] x Gdtm
    py[i] = py[i] + t x dt
    vz[i] = t =vz[i] + ffz[i] x Gdtm
    pz[i] = pz[i] + t x dt
    ffx[i] = ffy[i]= ffz[i]=0
  }   // k×n veces
}
```

| Zona 1: | k×n×(n-1)/2 | |
|---|---|---|
| SUMApf: | 11 | *8* |
| MULTpf: | 8 | *8* |
| DIVpf: | 1 | *1* |
| RAIZpf: | 1 | *1* |
| LecMem: | 7 | *4* |
| EscMem: | 3 | *0* |
| SLTcond: | 1 | *2* |

| Zona 2: | k×n | |
|---|---|---|
| SUMpf: | 9 | *6* |
| MULTpf: | 6 | *7* |
| DIVpf: | 1 | *1* |
| LecMem: | 17 | *13* |
| EscMem: | 12 | *9* |
| SLTcond: | 2 | *2* |

Este algoritmo también tiene dos zonas diferentes: el cálculo de las fuerzas ejercidas entre cada par de partículas, y la actualización de las fuerzas totales, velocidades y posiciones de las partículas. En la figura 21, se muestra una tabla resumen de las necesidades de volumen de cómputo, clasificadas por tipo, del algoritmo nbody-F.

| | Zona 1: cálculo de fuerzas | Zona 2: Actualización de F, V y P |
|---|---|---|
| **Complejidad:** | **k×n×(n-1)/2** | **k×n** |
| SUMAent * | **0 *** | **0 *** |
| SLTcond | **1** | **2** |
| LecMem | **7** | **17** |
| EscMem | **3** | **12** |
| SUMApf | **11** | **9** |
| MULTpf | **8** | **6** |
| DIVpf | **1** | **1** |
| RAIZpf | **1** | **0** |

**Figura 21:** Número de operaciones clasificadas por tipo para dos zonas del algoritmo nbody-F. Para cada zona se muestra la complejidad (número de veces que se ejecutan instrucciones en esa zona). Las operaciones marcadas con * no se han considerado.

Comparando estos resultados con la tabla del algoritmo nbody-A se encuentran las siguientes diferencias:

- La cantidad de operaciones DIVpf, RAIZpf y SLTcond se divide a la mitad
- La cantidad de operaciones SUMApf se divide por 1,45, y la cantidad de operaciones MULTpf se divide por 1,75.
- La cantidad de lecturas a memoria disminuye un 14%, pero aparecen escrituras a memoria en el bucle principal. En total, la cantidad de accesos a memoria (lecturas o escrituras) aumenta un 25%.

Por tanto, considerando únicamente el volumen de cómputo y accesos a memoria no se puede determinar con seguridad qué algoritmo será más rápido. Aunque como se verá más adelante en los nodos de nuestro estudio, el algoritmo nbody-F es con el que se obtienen los mejores tiempos. El algoritmo nbody-A podría ser más rápido en sistemas en los que la diferencia entre el rendimiento de la memoria y el rendimiento en cómputo fuera mucho mayor.

### Dependencias de Datos

En la figura 22 de la página siguiente se muestran las dependencias de datos entre las instrucciones. La cadena de dependencias más larga dentro del bucle de la Zona 1 es la siguiente:

{ LecMem, LecMem, LecMem } →
{ SUMApf, SUMApf, SUMApf } →
{ MULTpf, MULTpf } →
SUMApf →
SUMApf →
RAIZpf →
{ MULTpf, MULTpf } →
DIVpf →
{ MULTpf, MULTpf, MULTpf } →
{ SUMApf, SUMApf, SUMApf, SUMApf, SUMApf, SUMApf }

Comparado con el algoritmo nbody-A, la cadena contiene alguna operación adicional que se puede ejecutar en paralelo con las operaciones anteriores. El camino no se hace más largo en latencia teórica, pero requiere más capacidad de ejecución simultánea de instrucciones.

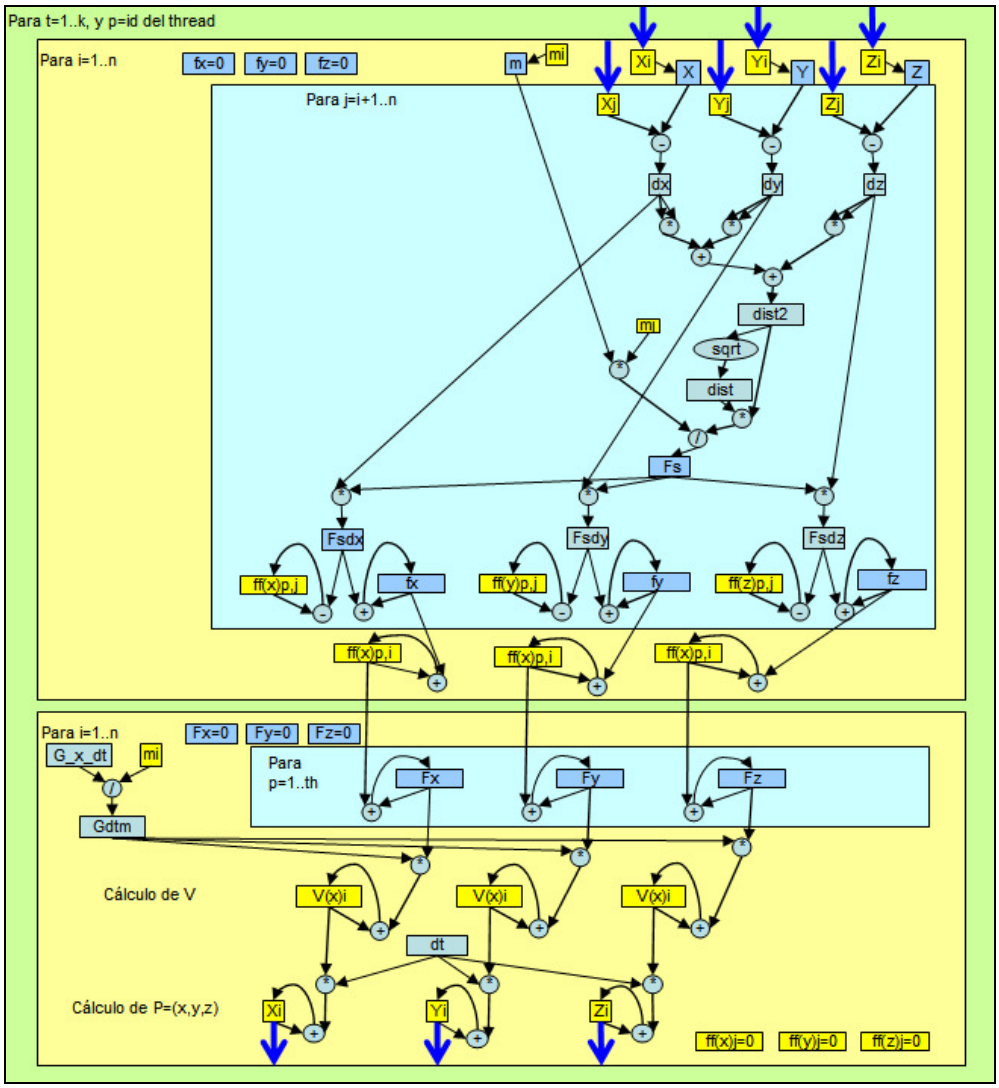



**Figura 22:** Caminos de dependencias de datos del algoritmo serie nbody-F.

### Operaciones de Acceso a Memoria

Si nos centramos en la Zona 1 del algoritmo, observamos que hay 7 operaciones de lectura (*LecMem.Px/Py/Pz/m/ffx/ffy/ffz*) y 3 operaciones de escritura (*EscMem.ffx/ffy/ffz*).

Cada una de las 10 operaciones se ejecuta aproximadamente $k \times n^2/2$ veces. En cada uno de los casos se accede ***n*** veces a posiciones inmediatamente consecutivas (stride = 1), primero a n-1 posiciones, luego a n-2, luego a n-3 … El patrón tanto de lecturas como de escrituras se puede definir de la siguiente forma:

***k*** × { { acceso secuencial, ***stride***=1, sobre ***n-1*** datos };
{ acceso secuencial, ***stride***=1, sobre ***n-2*** datos }; …{ acceso al último dato };

Los patrones de acceso secuencial son similares a los del algoritmo nbody-A, pero se siguen 7 patrones de forma simultánea, 3 de ellos de lectura/escritura.

### Operaciones de Salto Condicional

Hay un salto condicional en la Zona 1 del algoritmo, que denominaremos *SLTcond.for*, que se ejecuta $k \times n^2/2$ veces. El salto repite el siguiente patrón de comportamiento: ***n***-2 saltos seguidos de un no salto, luego ***n***-3 saltos seguidos de un no salto, etc. El patrón se puede definir de la siguiente forma:

***SLTcond.for:*** ***k*** × { { S, S, S, *(n-2 veces) ...* , N }; { S, S, S, *(n-3 veces) ...* , N } ... {N}}

El patrón de saltos también tiene una proporción de saltos elevada, del (n-1)/n% de las veces.

## 4.5. Modelo de rendimiento de los procesadores

En este apartado se describirán los procesadores utilizados en este trabajo. A continuación se presentará el método experimental para tomar medidas de rendimiento de cada procesador y proporcionar un modelo cuantitativo de cada uno de ellos. Finalmente se mostrarán los resultados obtenidos con estos experimentos, que definirán una parte del perfil de rendimiento de los procesadores.

### Descripción de los procesadores

Para cada procesador se muestra el nombre clave, el modelo de ejecución de instrucciones, los tamaños de cada nivel de memoria caché, y el sistema operativo, compilador y versión de PAPI utilizados.

**AMD Athlon(tm) 64 X2 Dual Core** (**AMDAthlonCore**)

| | |
|---|---|
| Procesador | AMD Athlon(tm) 64 X2 Dual Core 3800+ 1999.773MHz |
| Ejecución de instrucciones | Fuera de orden |
| Memoria principal : | 2048 MBytes |
| Cache L1 de instrucciones | 64 Kbytes |
| Cache L1 de datos | 64 Kbytes |
| Cache L2 | 512 Kbytes |
| Sistema Operativo : | Ubuntu 7.10 |
| Kernel | 2.6.21.1 |
| Compilador : | gcc version 4.1.2 (Ubuntu 4.1.2-0ubuntu4) |
| PAPI | 3.4.9 |

**Quad-Core AMD Opteron(tm)** (**AMDOpteronCore**)

| | |
|---|---|
| Procesadores | Quad-Core AMD Opteron(tm) 2352, 2111.021Mhz |
| Ejecución de instrucciones | Fuera de orden |
| Memoria principal : | 8192 MBytes |
| Cache L1 de instrucciones | 64 Kbytes |
| Cache L1 de datos | 64 Kbytes |
| Cache L2 | 512 Kbytes |
| Sistema Operativo : | Ubuntu 8.10 |
| Kernel | 2.6.24-16 |
| Compilador : | gcc version 4.3.2 (Ubuntu 4.3.2-1ubuntu12) |
| PAPI | ---------- |

**Intel(R) Core(TM)2 Duo** (**IntelCore**)

| | |
|---|---|
| Procesador | Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz |
| Ejecución de instrucciones | Fuera de orden |
| Memoria principal : | 1024 Mbytes |
| Cache L1 de instrucciones | 32 Kbytes |
| Cache L1 de datos | 32 Kbytes |
| Cache L2 | 2048 Kbytes |
| Sistema Operativo : | Fedora release 9 (Sulphur) |
| Kernel | 2.6.25.14 SMP |
| Compilador : | gcc version 4.3.0 20080428 (Red Hat 4.3.0-8) (GCC) |
| PAPI | 3.6.2 |

**Sun UltraSPARC T2** (SPARCT2core).

| | |
|---|---|
| Procesador | SUN SPARC-E-T5120, UltraSPARC T2, 1165 MHz |
| Ejecución de instrucciones | En orden |
| Memoria principal : | 8192 MBytes |
| Cache L1 de instrucciones | 16 Kbytes |
| Cache L1 de datos | 8 Kbytes |
| Cache L2 | 4096 Kbytes |
| Sistema Operativo : | SunOS 5.10 |
| Kernel | 2.6.24-16 |
| Compilador : | Sun C 5.9 SunOS_sparc |
| PAPI | ---------- |

## Metodología de Medición

Se han diseñado programas sintéticos de medida de rendimiento, específicos para cada tipo de operación básica considerada en el apartado 4.1. Los programas se han codificado en lenguaje C, pero se ha generado, analizado y editado el código ensamblador para verificar que las operaciones máquina ejecutadas corresponden con las operaciones básicas que se pretende evaluar. Hay que tener en cuenta que el compilador aplica optimizaciones al código que pueden suponer cambios determinantes en el programa ejecutable final, y que provoquen que la medida de tiempo final no refleje el resultado que se pretende obtener.

El siguiente código se usa como programa principal desde el que llamar a la función Benchmark, que ejecuta una determinada operación, una gran cantidad (repetida) de veces (Rep). El programa permite crear varios threads idénticos, realizando las mismas operaciones, y realiza la toma de medidas de tiempo y la lectura de contadores H/W de rendimiento mediante el interfaz implementado con las funciones sample_*. Para las operaciones de acceso a memoria se puede definir el tamaño del vector al que se realizan accesos secuenciales consecutivos.

```
double Benchmark (long Rep, void *V, long sz, int THR);
int main (int argc, char **argv)
{
  long Rep, SIZE;
  void *Vect;

  if (argc <= 2) {
    printf("Usage: Rep VectorSize [sample]\n"); return -1;
  }
  Rep  = (long) atof(argv[1]); argc--; argv++;
  SIZE = (long) atof(argv[1]); argc--; argv++;
  Sample_Init(argc, argv);
  Vect = (void *) malloc (SIZE*sizeof(double));

#pragma omp parallel
  {
    int THR = Sample_PAR_install();
    Sample_Start(THR);
    Benchmark(Rep, Vect, SIZE, THR);
    Sample_Stop(THR);
  }
  Sample_End();
}
```

El siguiente código se usa como patrón para la función Benchmark. Este código se ensambla y se edita para modificar el cuerpo del bucle principal (observar que contiene todas las operaciones básicas que se han considerado hasta ahora) y forzar a que se ejecute de forma repetitiva una operación básica concreta. En las dos tablas siguientes se muestra parte del código ensamblador original y la modificación realizada para generar el programa que mide el tiempo de la operación SUMApf cuando las operaciones se pueden hacer en paralelo (BW = *Bandwidth*, Ancho de Banda).

```
#include "math.h"
double Benchmark(long Rep, void *X, long sz, int THR)
{
  double v1, v2;
  double *init, *end;

  init = (double *) X;  end = init + sz;
  v1 = Rep;  v2 = -Rep;

  for (; Rep; Rep--) {
    v1 = *init;  v1 = v1 + v2; v1 = v1 * v2;
    v1 = v2 / v1; v2 = sqrt(v1); *init = v2;  init++;
    if (init >= end) init = end-sz;
  }
  return v1;
}
```

```
…
        movq    %rdi, %rax
        movq    %r13, %r12
        negq    %rax
        subq    %rdx, %r12
        cvtsi2sdq     %rax, %xmm1
        .p2align 4,,7
.L5:
        movsd   (%rbx), %xmm0
        movapd %xmm1, %xmm2
        addsd   %xmm1, %xmm0
        mulsd   %xmm1, %xmm0
        divsd   %xmm0, %xmm2
        sqrtsd %xmm2, %xmm0
        ucomisd       %xmm0, %xmm0
        jp      .L12
        je      .L6
.L12:   movapd %xmm2, %xmm0
        movsd   %xmm2, (%rsp)
        call    sqrt
        movsd   (%rsp), %xmm2
.L6:    movsd   %xmm0, (%rbx)
        addq    $8, %rbx
        cmpq    %rbx, %r13
        movapd %xmm0, %xmm1
        cmovbe %r12, %rbx
        subq    $1, %rbp
        jne     .L5
.L4:
        addq    $24, %rsp
        movapd %xmm2, %xmm0
        popq    %rbx
        popq    %rbp
        popq    %r12
        popq    %r13
        ret
…
```

**Assembler Original**

```
…       movq    %rdi, %rax
        movq    %r13, %r12
        negq    %rax
        subq    %rdx, %r12
        cvtsi2sdq     %rax, %xmm1
        movapd %xmm1, %xmm0
        movapd %xmm1, %xmm2
        movapd %xmm1, %xmm3
        movapd %xmm1, %xmm4
        movapd %xmm1, %xmm5
        movapd %xmm1, %xmm6
        movapd %xmm1, %xmm7
        movapd %xmm1, %xmm8
        movapd %xmm1, %xmm9
        .p2align 4,,7
.L5:    addsd %xmm1, %xmm1
        addsd %xmm0, %xmm0
        addsd %xmm2, %xmm2
        addsd %xmm3, %xmm3
        addsd %xmm4, %xmm4
        addsd %xmm5, %xmm5
        addsd %xmm6, %xmm6
        addsd %xmm7, %xmm7
        addsd %xmm8, %xmm8
        addsd %xmm9, %xmm9
        subq    $1, %rbp
        jne     .L5
.L4:
        addq    $24, %rsp
        movapd %xmm2, %xmm0
        popq    %rbx
        popq    %rbp
        popq    %r12
        popq    %r13
        ret
```

**Assembler Modificado (SUMApfBW)**

## Perfil de Rendimiento de los procesadores

Las siguientes tablas muestran parte del perfil de rendimiento de dos de los procesadores, obtenido a partir de la ejecución de los programas sintéticos descritos en el apartado anterior. Para cada operación hay dos medidas: Latencia y Ancho de Banda. La medida de latencia corresponde a la ejecución de la operación de forma repetitiva y **no solapada**: el resultado de una operación se usa para la operación inmediatamente siguiente. La medida de ancho de banda corresponde a la ejecución de la operación de forma repetitiva y **solapada**: hay 10 operaciones que se pueden ejecutar de forma independiente. Los resultados de cada una de estas 10 operaciones sirven para iniciar la ejecución de otro conjunto de 10 operaciones (ver la tabla anterior etiquetada Assembler Modificado – SUMApfBW).

En el caso de las operaciones de lectura y escritura, siempre se refiere a accesos secuenciales en memoria (*stride*=1, que corresponde al mejor caso). Cuando los accesos consecutivos van a parar a distintas líneas de caché, los tiempos de los accesos se disparan (no se muestran en las tablas).

En la tabla de la figura 23, se puede apreciar que, para todos los tipos de operación analizados, el procesador AMD Athlon ofrece un mejor ancho de banda de ejecución que lo que indica la latencia de cada operación (gracias a la segmentación de la ejecución y/o a la duplicación de unidades funcionales de ejecución). Para este procesador, por tanto, es ventajoso que el programa proporcione paralelismo a nivel de instrucción.

Otro detalle importante para resaltar es la reducción de rendimiento en las operaciones de lectura y escritura cuando los datos se leen o escriben en Memoria, respecto a escribirlos en la caché L2, o más aún respecto a hacerlo en la caché L1. Esta reducción, entre 4 y 10 veces, aumenta muchísimo si los accesos a memoria no son consecutivos.

| ***Métrica: ns por operación*** | | **AMDAthlonCore (1.9997 GHz)** | |
|---|---|---|---|
| **Operación básica** | **Operación Máquina** | **Latencia** | **Ancho de Banda** |
| **SUMAent** | **add** | **0,5** | **0,167** |
| **SLTcond** | **je** | **4,47 (si salta)** | **0,167 (si no salta)** |
| **LecMemEnt (stride=1)** | **movq (%rdi), %rax (8 bytes)** | **1,5 (L1 cache)**<br>**2,07 (L2 cache)**<br>**3,47 (Main Memory)** | **0,25 (L1 cache)**<br>**1,01 (L2 cache)**<br>**2,47 (Main Memory)** |
| **LecMemPf (stride=1)** | **movsd (%rdi), %xmm1 (8 bytes)** | **------** | **0,72 (L1 cache)**<br>**0,92 (L2 cache)**<br>**2,50 (Main Memory)** |
| **EscMemEnt (stride=1)** | **movq %rax, (%rdi) (8 bytes)** | **------** | **0,25 (L1 cache)**<br>**1,02 (L2 cache)**<br>**3,20 (Main Memory)** |
| **EscMemPf (stride=1)** | **movq %xmm1, (%rdi) (8 bytes)** | **------** | **0,5 (L1 cache)**<br>**1,02 (L2 cache)**<br>**3,20 (Main Memory)** |
| **SUMApf** | **addsd** | **2,0** | **0,5** |
| **MULTpf** | **mulsd** | **2,0** | **0,5** |
| **DIVpf** | **divsd** | **5,5** | **4,0** |
| **RAIZpf** | **sqrtsd** | **13,5** | **12,0** |

**Figura 23:** Perfil de Rendimiento del procesador AMD Athlon.

En la tabla de la figura 24, se muestran los resultados para el procesador SPARC T2. Como se trata de un procesador con capacidad de ejecución multi-thread, se han hecho experimentos usando 1, 2, 4 y 8 threads. Los resultados indican tiempo por operación ejecutada en el núcleo de ejecución: cuando se ejecutan varios threads en el mismo núcleo, se divide el tiempo total de ejecución de todos los threads entre la suma de todas las operaciones ejecutadas por estos threads.

En todos los casos la ejecución de múltiples threads mejora el resultado. Hay que tener en cuenta que el modo de planificación de la ejecución de instrucciones del procesador es en orden. Esta desventaja se ve compensada por la capacidad de ejecución de threads fuera de

orden, siempre que haya 2 o más threads disponibles para ejecutar. La disponibilidad de varios threads permite proporcionar al procesador varias operaciones independientes, aunque dentro del mismo thread la ejecución de las operaciones no se pueda solapar. Por esta razón, el tiempo de ejecución por operación en el modo "Latencia" va disminuyendo al aumentar el nº de threads que intercalan su ejecución.

Al contrario que en el procesador anterior, el procesador Sun SPARC T2 sólo ofrece un mejor ancho de banda de ejecución que lo que indica la latencia de cada operación en las operaciones SUMApf, MULTpf y RAIZpf. Para este procesador, por tanto, no es tan ventajoso que cada thread ofrezca un alto grado de paralelismo a nivel de instrucción, como el hecho de que se dispongan de varios threads independientes para ser ejecutados.

| ***Métrica: ns por operación*** | | **SPARCT2core (1.165 GHz)** | |
|---|---|---|---|
| **Operación básica** | **Operación Máquina** | **Latencia** | **Ancho de Banda** |
| **SUMAent** | **addx** | **1,01 / 0,54 / 0,45 / 0,45** | **1,01 / 0,55 / 0,46 / 0,45** |
| **MULTent** | **mulx** | **7,94 / 3,97 / 2,12 / 1,32** | **7,94 / 3,97 / 2,02 / 1,27** |
| **SLTcond** | **bne** | **6,86 / 3,59 / 1,93 / 1,48 (si salta)** | **2,57 / 1,29 / 0,9 / 0,66 (si no salta)** |
| **LecMemPf (stride=1)** | **ldd [%i1], %f0 (8 bytes)** | **------** | **1,56 / 0,99 / 0,91 / 0,88 (L1) 11 / 5,6 / 3,12 / 1,86 (L2) 27,8 / 18,7 / 13,7 / 7,0 (Main)** |
| **EscMemPf (stride=1)** | **std %f0,[%i1] (8 bytes)** | **------** | **4,6 / 2,6 / 1,8 / 1,71 (L1) 4,6 / 2,7 / 1,8 / 1,71 (L2) 25,6 / 12,9 / 6,50 / 3,34 (Main)** |
| **SUMApf** | **addd** | **5,15 / 2,58 / 1,43 / 1,29** | **1,46 / 0,95 / 0,875 / 0,8663** |
| **MULTpf** | **muld** | **5,15 / 2,58 / 1,43 / 1,29** | **1,46 / 0,95 / 0,875 / 0,8663** |
| **DIVpf** | **divd** | **31,8 / 29,5 / 29,0 / 29,9** | **31,8 / 29,5 / 29,0 / 29,9** |
| **RAIZpf** | **Call sqrt** | **153,0 / 77,3 / 41,5 / 31,1** | **123,4 / 62,5 / 34,25 / 31,1** |

**Figura 24:** Perfil de Rendimiento del procesador SPARCT2, cuando se ejecutan 1 / 2 / 4 / 8 threads en un núcleo (con capacidad multithread)

Los resultados en el IntelCore y el AMDOpteronCore (que no se muestran) son similares a los obtenidos en el procesador AMDAthlon, excepto que están escalados por la frecuencia del reloj.

Además de los datos anteriores, hay otras caracteristicas de los procesadores, que afectan de forma significativa a su rendimiento, y que no se han medido, como:

- Número total de registros de renombrado del procesador
- Capacidad máxima de planificación dinámica y ejecución fuera de orden.

## 4.6. 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 y de inicialización de datos de entrada y de verificación del resultado. Este tiempo de ejecución se ha dividido por la complejidad del algoritmo asociada al tamaño del problema ($\boldsymbol{k}\ x\ \boldsymbol{n}^2$), y en las gráficas se muestran valores medidos en ns / $\boldsymbol{k}\ x\ \boldsymbol{n}^2$. En las ejecuciones se ha variado el número de iteraciones, ***k***, entre 1 y 10, y el número de partículas, ***n***, entre 10 y 500.000.

Para instrumentar los programas cuya ejecución se quiere medir se ha utilizado una serie de funciones que se han creado a propósito (*sample_init, sample_PAR_install, sample_start, sample_stop y sample_end*), y que se pueden ver en el código presentado en el apartado anterior. Las mediciones de tiempo y otros eventos ligados al rendimiento para cada uno de los threads las realizamos incluyendo las llamadas *sample_start* y *sample_stop* al inicio y al final de la ejecución de cada thread.

Al generar el ejecutable final, se pueden 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 fallos en L1 y L2, y la cantidad de instrucciones máquina ejecutadas.

En cada procesador y para cada variable a estudiar se realizan 15 ejecuciones, de las que se descartan 4. Son descartadas las dos ejecuciones que contengan los dos valores más altos y los dos valores más bajos de la variable estudiada. El objetivo es filtrar las perturbaciones (*outliers*) provocadas por procesos de sistema operativo, o por la ejecución de procesos por otros usuarios del sistema. Con los valores de las 11 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, también conocida como desviación típica.

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}$$

Como parte del trabajo se ha preparado una infraestructura portable para la compilación y/o ejecución en infrastructuras x86 y x86_64 de aplicaciones codificadas en C, utilizando OpenMP y PAPI (para acceder a los contadores hardware del procesador). Esta infraestructura se basa en un sistema operativo GNU/Linux al que se le ha aplicado el parche perfctr y en el que se ha recompilado el kernel con las opciones necesarias para poder acceder a los contadores hardware de los procesadores x86 y x86_64 (utilizando “PAPI”). De este sistema operativo se ha creado una imagen y se ha procedimentado el como transferir esta imagen a la memoria flash de un pendrive. Permitiendo así que un equipo pueda arrancar del sistema operativo del pendrive. De esta manera se dispone de una infrastructura portable para poder realizar pruebas de rendimiento en cualquier plataforma x86 y x86_64 soportada por PAPI, sin la necesidad de modificar el sistema operativo del equipo en cuestión.

## 4.7. Resultados para nbody-A

En este apartado se mostrarán y discutirán los resultados de rendimiento obtenidos al ejecutar la implementación del algoritmo n-body-A en los cuatro procesadores descritos anteriormente. El objeto del análisis es extraer experiencia que permita generalizarlo a diferentes casos de aplicaciones y proporcionar una serie de reglas.

### Gráfica de Tiempo Total y Gráfica de Tiempo dividido por Complejidad

Aunque la métrica de medida obtenida en los experimentos es el tiempo de ejecución total del programa, a continuación mostraremos que es conveniente modificar la forma en que se muestra este tiempo. La figura 25 muestra simultáneamente el tiempo de ejecución en segundos y el tiempo dividido por la complejidad (en nanosegundos divides por $\boldsymbol{k} \times \boldsymbol{n}^2$), además de las desviaciones estándar de ambas métricas obtenidas según el proceso descrito en el apartado anterior. Se han tomado medidas para diferentes valores de $\boldsymbol{n}$, y con $\boldsymbol{k}$=1.

La curva del tiempo de ejecución (color azul) debería mostrar un crecimiento cuadrático con el número de partículas ($\boldsymbol{n}$), que se varía en el eje horizontal. Este crecimiento cuadrático no es fácil de corroborar visualmente. Además, como el tiempo se presenta en una escala lineal, los resultados para tamaños pequeños de $\boldsymbol{n}$ se hacen planos y son difíciles de leer, o los valores para tamaños grandes de $\boldsymbol{n}$ se hacen enormes, y difíciles de visualizar. Si el tiempo se presentara en una escala logarítmica, entonces se facilitaría la lectura de valores pero se haría aún más difícil corroborar el crecimiento cuadrático.

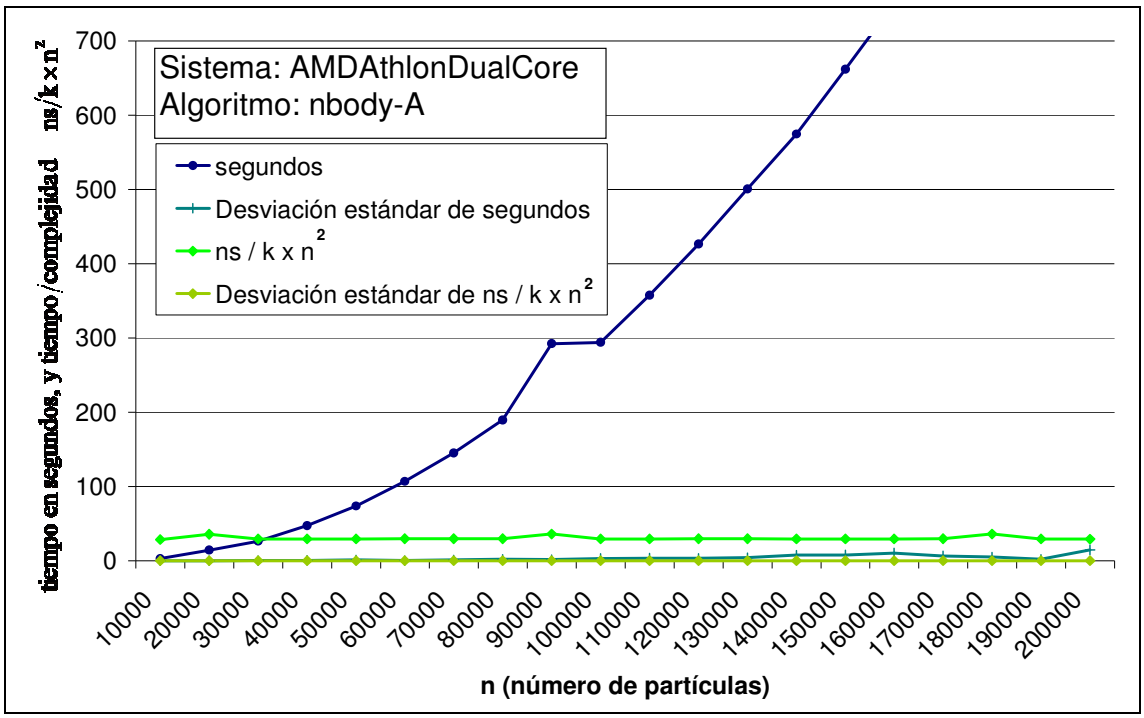



**Figura 25:** Tiempo de ejecución, tiempo de ejecución dividido por complejidad, y desviaciones estándar de ambas métricas, para la ejecución del algoritmo nbody-A en el procesador AMDAthlon, con k=1, y n entre 10.000 y 200.000.

En la curva del tiempo de ejecución se puede apreciar una irregularidad o anomalía en el punto ***n***=90.000. El tiempo de ejecución en este punto es demasiado alto, casi el mismo que para **n**=100.000, pero es difícil (sobre la curva) calcular el porcentaje de desviación respecto al tiempo "esperado".

En cambio, si observamos la curva del tiempo de ejecución dividido por la complejidad (color verde), se pueden leer fácilmente cuatro datos significativos:

- La curva corresponde a una recta (excepto en los casos anómalos que se comentarán a continuación), lo cual corrobora el crecimiento cuadrático del tiempo de ejecución. El valor concreto de la constante que aproxima a esta recta se puede leer fácilmente si la escala de la gráfica es la apropiada, tal como se ve en la figura 26 (el valor es casi 30 ns). Este valor constante representa el tiempo de ejecución promedio de cada operación abstracta, y se analizará posteriormente para ser explicado en términos del tiempo de ejecución de las operaciones básicas y será utilizado para descubrir los "cuellos de botella" del rendimiento.
- Para el valor anómalo en el punto ***n***=90.000 se puede hacer una estimación rápida de su porcentaje de desviación respecto al valor obtenido para el resto de puntos de la curva. En la figura 26 se observan unos 6 ns de incremento respecto a 30, es decir, un 20%.
- Se descubren otros puntos anómalos, que en la curva de tiempo total de ejecución quedaban "ocultos", como los punto ***n***=20.000 y ***n***=180.000.
- La desviación estándar de las muestras de tiempo obtenidas en las ejecuciones es muy baja, y se mantiene bastante constante a lo largo de los diferentes tamaños de problema considerados. Esto indica que no hay problemas de estabilidad en el rendimiento, incluso para los casos anómalos.

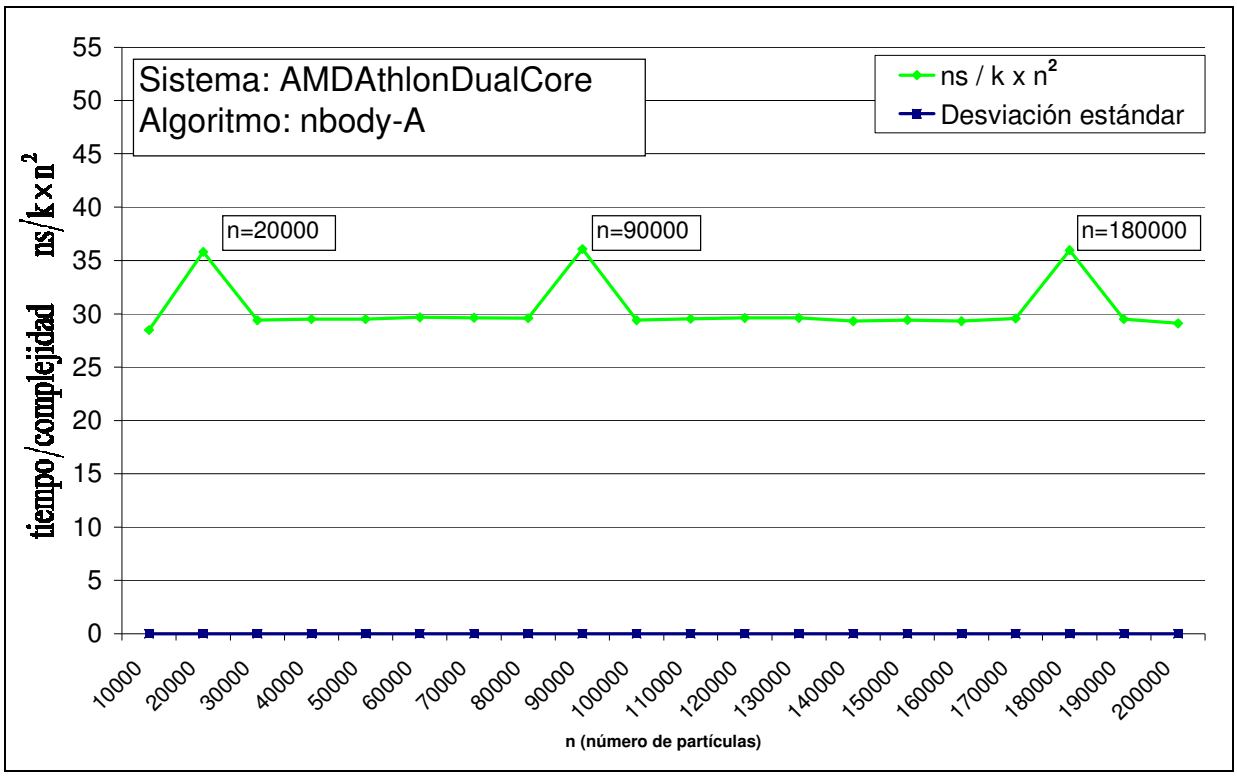



**Figura 26:** Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador AMDAthlon, con **k**=1, y **n** entre 10.000 y 200.000.

En resumen, para analizar el efecto en el tiempo de ejecución de variar el tamaño del problema, y descubrir ineficiencias de rendimiento, se divide el tiempo total por el volumen de operaciones abstractas ejecutadas (complejidad). Así se anula el efecto de crecimiento en el tiempo de ejecución debido simplemente al incremento del volumen de operaciones. Esta gráfica es sencilla de interpretar:

- Valores altos en la gráfica indican más tiempo, y por tanto peor rendimiento
- Si no hay ineficiencias, la curva tendrá un valor constante. Las ineficiencias se visualizan en las variaciones o anomalías de la gráfica
- Permite comparar resultados con distintos procesadores y/o compiladores

Si la complejidad del algoritmo es difícil de evaluar, y la cantidad de operaciones abstractas ejecutadas depende fuertemente del valor de los datos de entrada, entonces la gráfica puede ser poco útil para analizar el rendimiento. En este caso se podría utilizar el número total de operaciones básicas o el número total de instrucciones máquina (obteniendo tiempo por instrucción o el equivalente de ciclos por instrucción, conocido como CPI).

## Anomalía debida al alineamiento de los datos de los vectores

En la figura 26 se ha descubierto que, respecto a la mayoría de los casos, en los puntos ***n***=20.000, ***n***=90.000 y ***n***=180.000 existe aproximadamente un 20% de incremento en el tiempo de ejecución. Para analizar esta anomalía se han realizado experimentos con ***n*** incrementado de uno en uno en el entorno de ***n***=20000, y se ha comprobado que la anomalía aparece con 19965 > ***n*** > 20478 (ver figura 27).

También se ha analizado con detalle el comportamiento del rendimiento para valores de ***n*** inferiores a 3000 (ver figura 28). El problema aparece a intervalos mucho más estrechos: ***n***=110, 220, 330 ,440 y así sucesivamente hasta ***n***=2320.

El problema anterior se produce tanto en el procesador AMDAthlon que se está analizando, como en el AMDOpteron, pero no se produce ni en el Intel ni en el SPARCT2. Esto indica que alguna característica de la microarquitectura de los dos primeros procesadores, combinada con el comportamiento del algoritmo, es la causa de esta anomalía en el rendimiento.

En general, muchas de las anomalías de rendimiento se deben a la interacción con la jerarquía de memoria. En concreto, si el problema ocurre en zonas del tamaño del problema relativamente pequeñas, en muchas ocasiones es debido a la confluencia de ciertos alineamientos en los accesos a memoria. Una forma de verificar que efectivamente el problema es causado por la jerarquía de memoria es analizar los fallos de caché. En las figuras 29 y 30 se muestran los fallos de caché por instrucción máquina ejecutada tanto en L1 como en L2. Se puede verificar que en los puntos anómalos la tasa de fallos en L1 aumenta de forma considerable, reforzando el argumento de que la causa del problema está relacionada con el sistema de memoria.

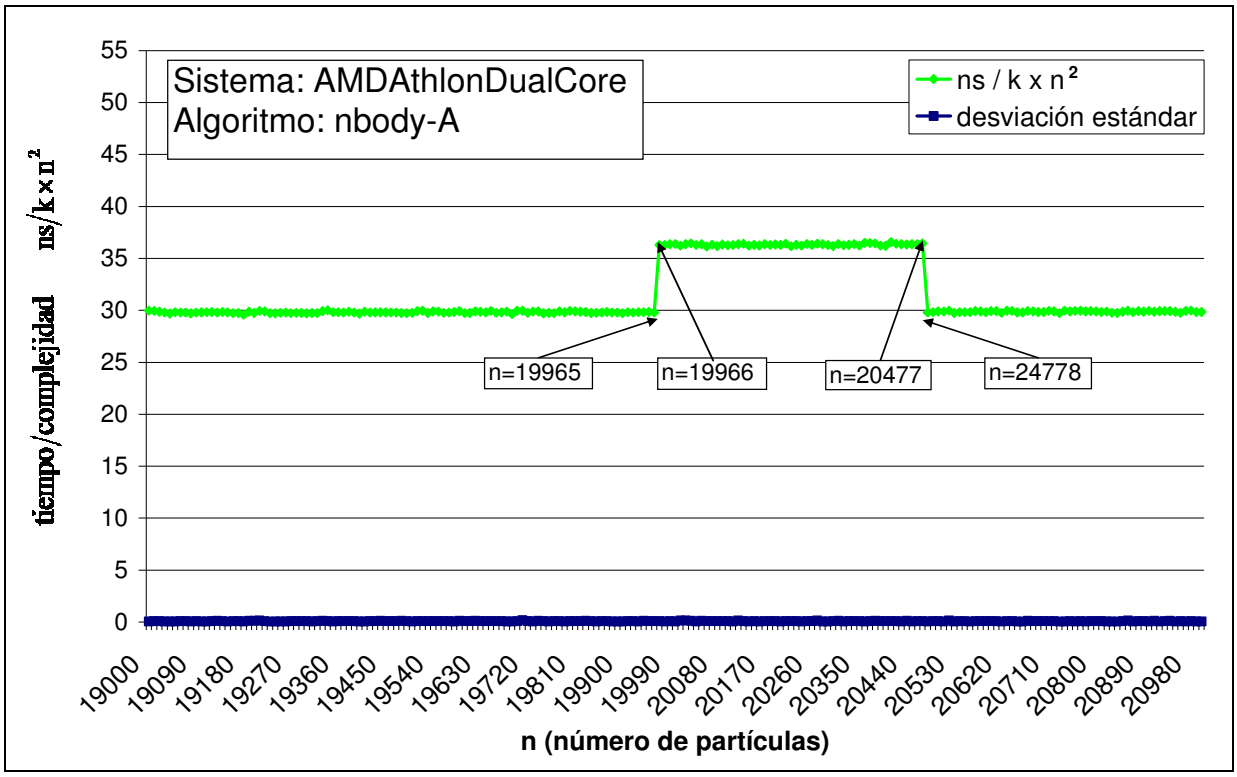



**Figura 27:** Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador AMDAthlon, con **k**=1, y **n** entre 19.000 y 21.000.

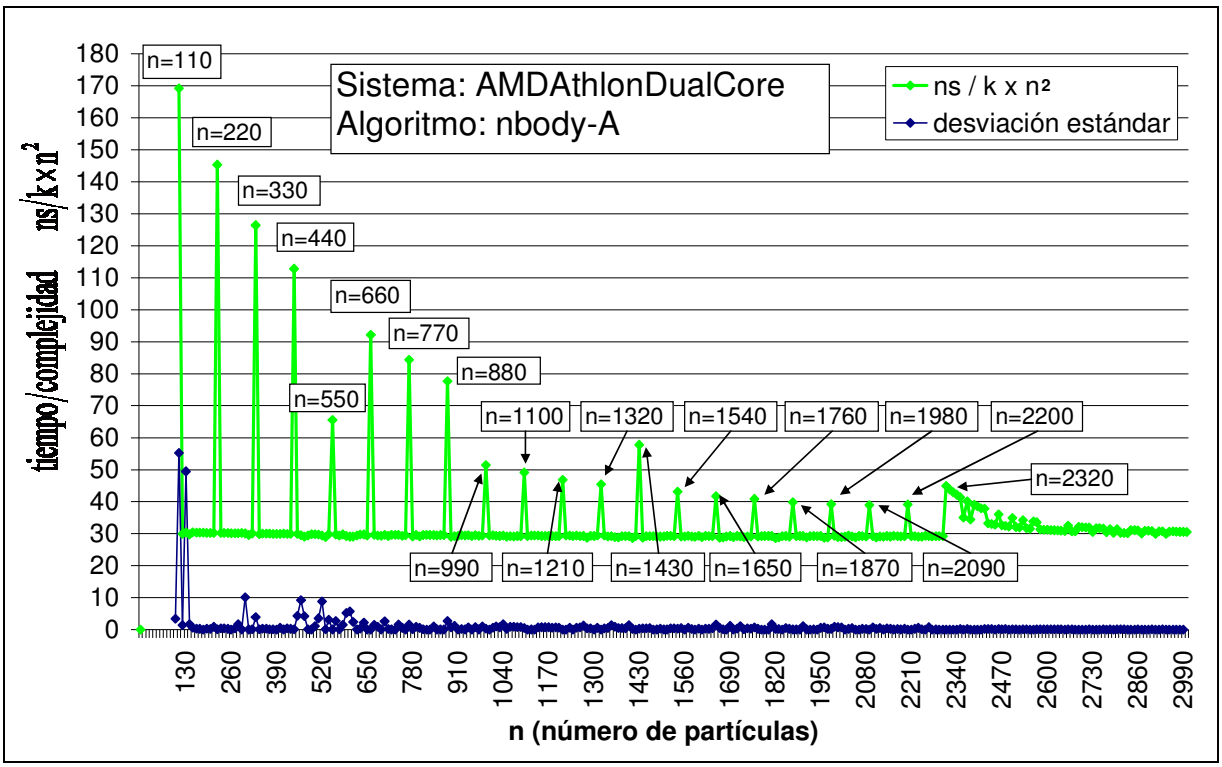



**Figura 28:** Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador AMDAthlon, con **k**=1, y **n** menor que 3.000.

Puede parecer curioso que en los mismos puntos anómalos, la tasa de fallos en L2 disminuya, lo cual en principio es beneficioso para el rendimiento. Sin duda se trata de un efecto positivo provocado por el mismo problema, pero que se ve superado por el efecto negativo del incremento en los fallos en la caché L1.

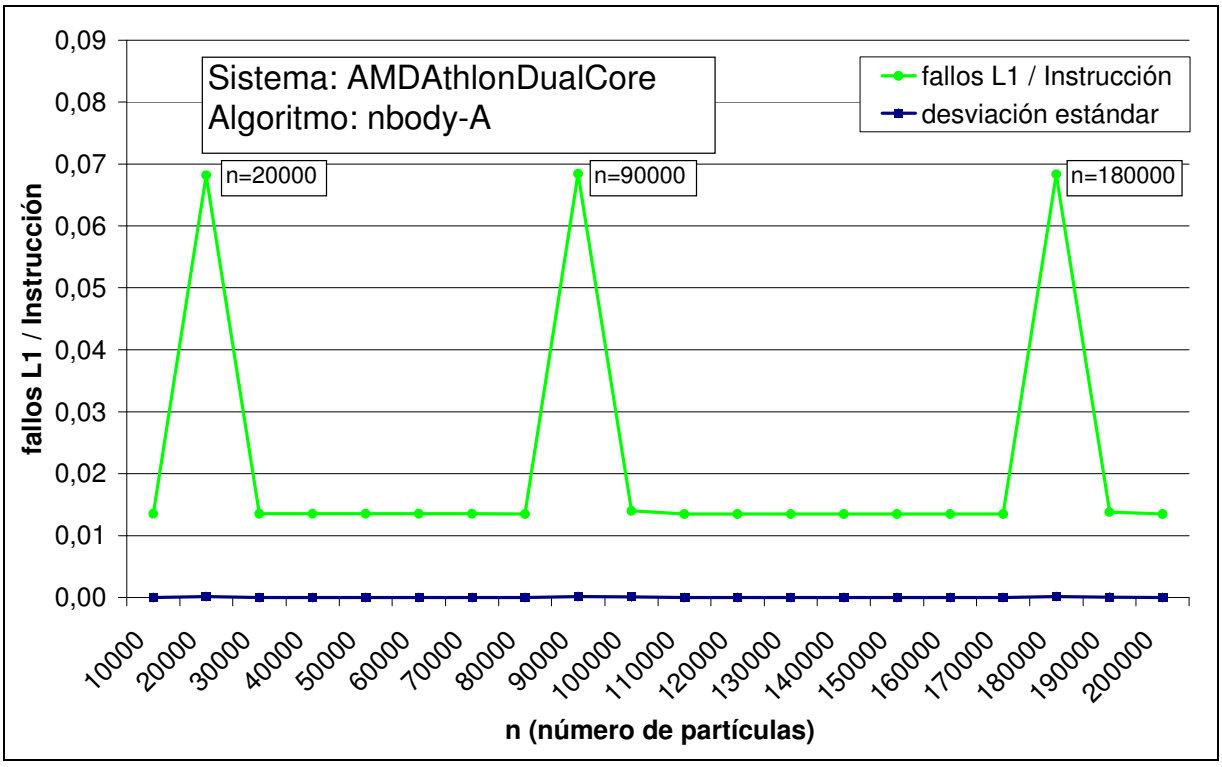



**Figura 29:** Fallos de caché en L1 divididos por el nº total de instrucciones ejecutadas y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador AMDAthlon, con **k**=1, y **n** entre 10.000 y 200.000.

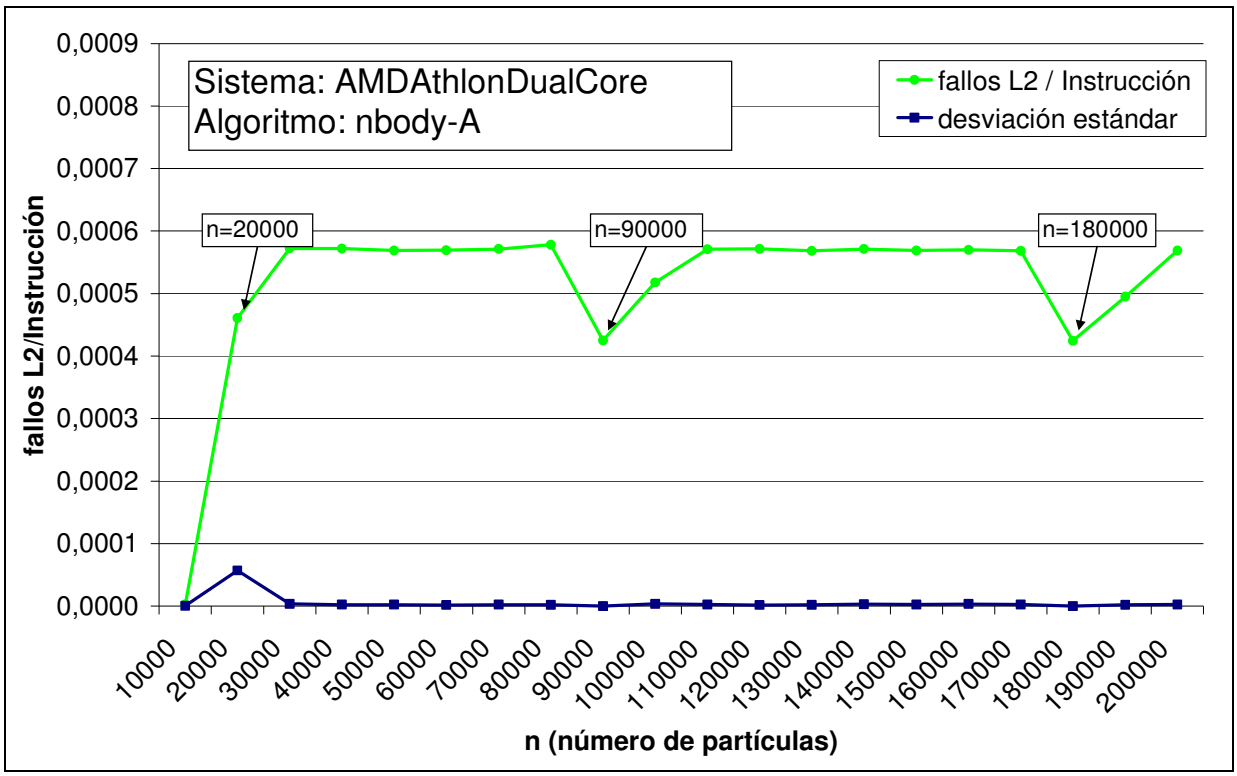



**Figura 30:** Fallos de caché en L2 divididos por el nº total de instrucciones ejecutadas y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador AMDAthlon, con **k**=1, y **n** entre 10.000 y 200.000.

Llegados a este punto decidimos aventurar que el problema es debido a un alineamiento desafortunado de los vectores de memoria a los que el programa está accediendo: para ciertas distancias entre los accesos a memoria se producen conflictos en el uso de la memoria caché que provocan "*thrashing*" (por falta de asociatividad en el uso de los bloques de caché, los datos que se traen provocan la salida de datos que se van a usar en un futuro muy próximo). Tiene lógica que el problema no aparezca (al menos en los puntos considerados) en los procesadores Intel y SPARCT2, porque el tamaño concreto de los vectores que provoca el problema es dependiente de la topología concreta de las memorias caché. También es consistente el hecho comprobado de que las anomalías sean en los mismos puntos para los dos procesadores AMD, que tienen una jerarquía de caché L1 similar.

Una vez se dispone de un posible diagnóstico, se prueba la solución de sobredimensionar el tamaño de los vectores usados en el programa (en concreto los vectores Px, Py, Pz y m, sospechosos de generar el problema) para los tamaños del problema en el entorno de ***n***=20.000, ***n***=90.000 y ***n***=180.000. Esta estrategia, denominada "padding", es efectiva y hace desaparecer el comportamiento anómalo del rendimiento en las zonas consideradas.

Aumentando un 5% los vectores Px, Py, y Px de posición en los puntos en los que se producía el problema, es suficiente para evitarlo. Esta solución ha servido para demostrar que el problema estaba relacionado con los alineamientos de memoria, y para obtener unas gráficas con las que comparar con mayor facilidad el rendimiento de los distintos nodos analizados. Se ha aplicado esta solución en los experimentos a partir de los cuales se han obtenido los datos con los que se han creado las gráficas resumen de rendimiento de nbody-A y nbody-F mostradas al final del capítulo 4.8 y las gráficas del estudio paralelo (capitulo 5).

Una solución más elegante al problema del alineamiento consiste en intercalar los elementos de los vectores Px, Py, Pz, para forzar que todos los accesos a memoria realizados en el bucle interno del programa se hagan de forma puramente secuencial. La estrategia ha sido probada y ha resultado ser efectiva en varios casos concretos, pero por falta de tiempo no se ha podido aplicar a todos los algoritmos.

## Estudio en profundidad de los fallos de caché

El análisis de los fallos de caché puede sacar a la luz problemas latentes de rendimiento, que pueden llegar a hacerse efectivos en procesadores con ciertas características. En el caso concreto de los procesadores considerados, el efecto de los fallos en L1 sobre el tiempo de ejecución es insignificante. La razón de ello es que la latencia de los fallos en L1 se puede solapar perfectamente con el cómputo del algoritmo. Los fallos en L2 explican la pequeña degradación en el rendimiento que se observa en la figura 26 al pasar de n=10.000 a n=30.000, inferior al 4%.

### Fallos en la caché L1

En la figura 31 se muestran los fallos de caché en L1 por instrucción máquina ejecutada para valores de ***n*** inferiores a 5.000. Notar que la escala vertical de valores es logarítmica. La curva de fallos de caché muestra un comportamiento de varias fases, muy frecuente en los algoritmos que contienen patrones de acceso secuencial a vectores de tamaño creciente. Cabe recordar en este momento los requerimientos de memoria y los patrones de acceso a memoria del algoritmo nbody-A, descrito en el apartado 4.3. Se necesita almacenar 7 vectores de ***n*** elementos, para un total de 7×8×***n*** = 56***n*** bytes. El bucle más interno contiene 4 instrucciones de acceso a memoria, *LecMem.Px, LecMem.Py, LecMem.Pz,* y *LecMem.m*, cada una con un patrón de acceso secuencial descrito así:

***k*** × ***n*** repeticiones × { acceso secuencial, ***stride***=1, sobre ***n*** datos }

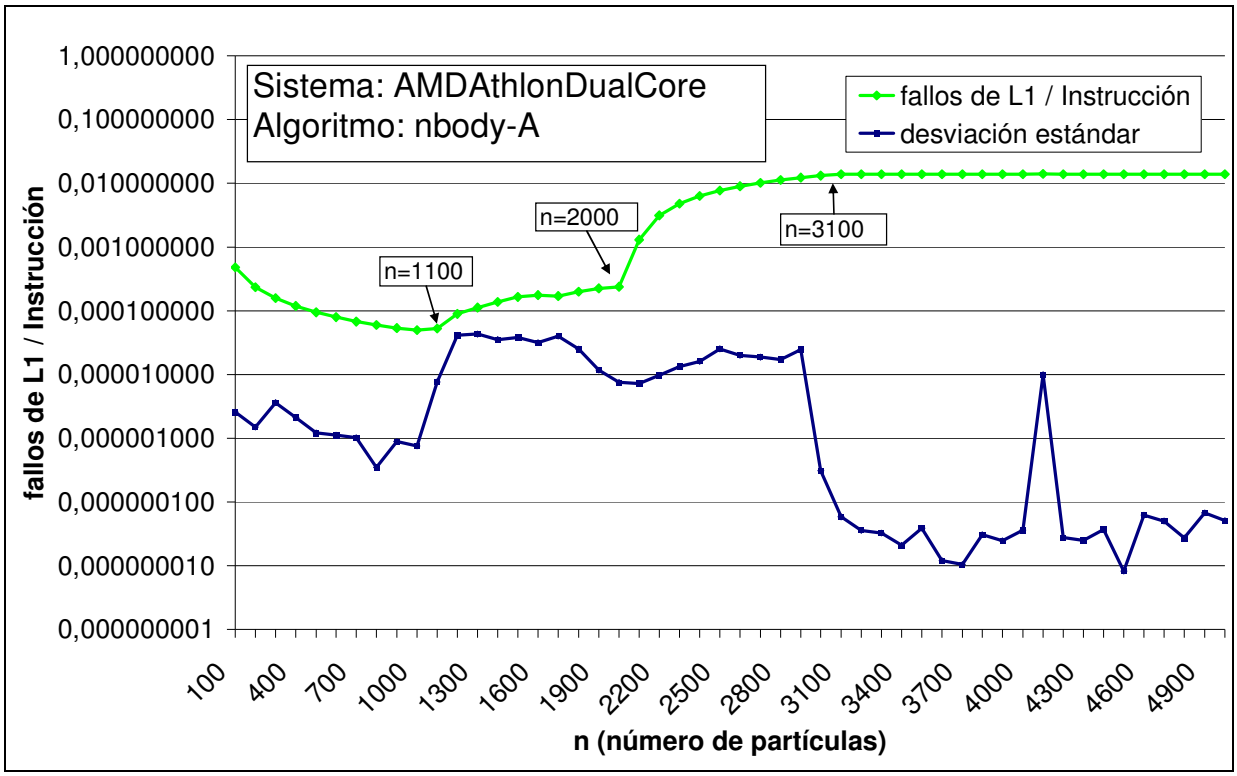



**Figura 31.** Fallos de caché en L1 divididos por el nº total de instrucciones ejecutadas y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador AMDAthlon, con **k**=1, y **n** menor de 5.000.

En este ejemplo se distinguen 4 fases, identificadas por tres puntos de la curva que muestran variaciones bruscas en la pendiente, indicados en la figura, y que se describen a continuación:

1. Los datos de los 7 vectores **ocupan menos del 95%** del tamaño de la caché L1 (***n***=1.100 supone 56×1.100= 61,6 KB). La proporción de fallos decrece al aumentar ***n*** hasta un valor del 0,007%. En esta fase sólo se producen fallos en frío, y como se hace una única iteración del algoritmo (***k***=1), el efecto de estos fallos en frío sobre el promedio de fallos no es despreciable. El promedio va disminuyendo al aumentar ***n*** porque los fallos en frío crecen linealmente con ***n*** mientras que las instrucciones ejecutadas crecen con $n^2$.
2. Los datos de los 7 vectores **ocupan más del 95%** del tamaño de la caché L1 pero los datos de los vectores Px, Py, Pz y m **ocupan menos del 100%** de la caché L1 (***n***=2.000 supone 8×4×2.000= 64 KB). La proporción de fallos va creciendo con ***n*** hasta un valor del 0,02%. En esta fase los datos accedidos dentro del bucle más interno (Px, Py, Pz y m) caben en la caché L1, pero los datos accedidos por el bucle más externo (Vx, Vy y Vz) no caben junto con los primeros. Cada vez que se accede a los datos Vx, Vy y Vz existe la posibilidad de fallo de capacidad, que además desaloje un dato útil de los vectores Px, Py, Pz y m. Esta posibilidad se hace más frecuente al crecer ***n***, y eso provoca que la tasa de fallos vaya creciendo.
3. Los datos de los vectores Px, Py, Pz y m **ocupan más del 100% pero menos del 150%** del tamaño de la caché L1 (***n***=3.100 supone 8×4×3.100=99KB). La proporción de fallos crece fuertemente hasta que se estabiliza en un valor del 1,3%. Al crecer ***n*** aumentan los fallos de capacidad de la caché y disminuye el aprovechamiento de la localidad temporal
4. Los datos de los vectores Px, Py, Pz y m **ocupan más del 150%** del tamaño de la caché L1. Se alcanza la situación en la que los fallos de capacidad son máximos y no se aprovecha en ningún caso la localidad temporal. Solamente se aprovecha la localidad espacial en los accesos a Px, Py, Pz y m.

**Fallos en la caché L2**

En la figura 30 se muestran los fallos en la caché L2. Si descontamos el efecto anómalo del alineamiento de accesos a memoria, la curva de fallos crece desde **n**=10.000 hasta **n**=30.000 y luego se estabiliza.

El valor **n**=10.000 supone unos requerimientos de memoria de unos 312 KB, suficientes para caber en la caché L2. El valor **n**=20.000 supone unos requerimientos de 625 KB, insuficientes para caber en la caché L2, pero no superan el doble del tamaño de la L2 (2x512=1024KB). En este caso, se producen algunos fallos de capacidad pero aún se está aprovechando parte de la localidad temporal. Finalmente, el valor **n**=30.000 supone unos requerimientos de 937KB, que prácticamente suponen el doble del tamaño de la L2, y provoca que casi no se aproveche nada de localidad temporal.

La figura 32 muestra los fallos en la caché L2 para el procesador Intel, que dispone de una caché L2 de un tamaño 4 veces superior al de la caché de los procesadores AMD. Tras el análisis anterior, es bastante previsible que los problemas de fallos en L2 se retrasen a valores de **n** cuatro veces superiores a los de antes.

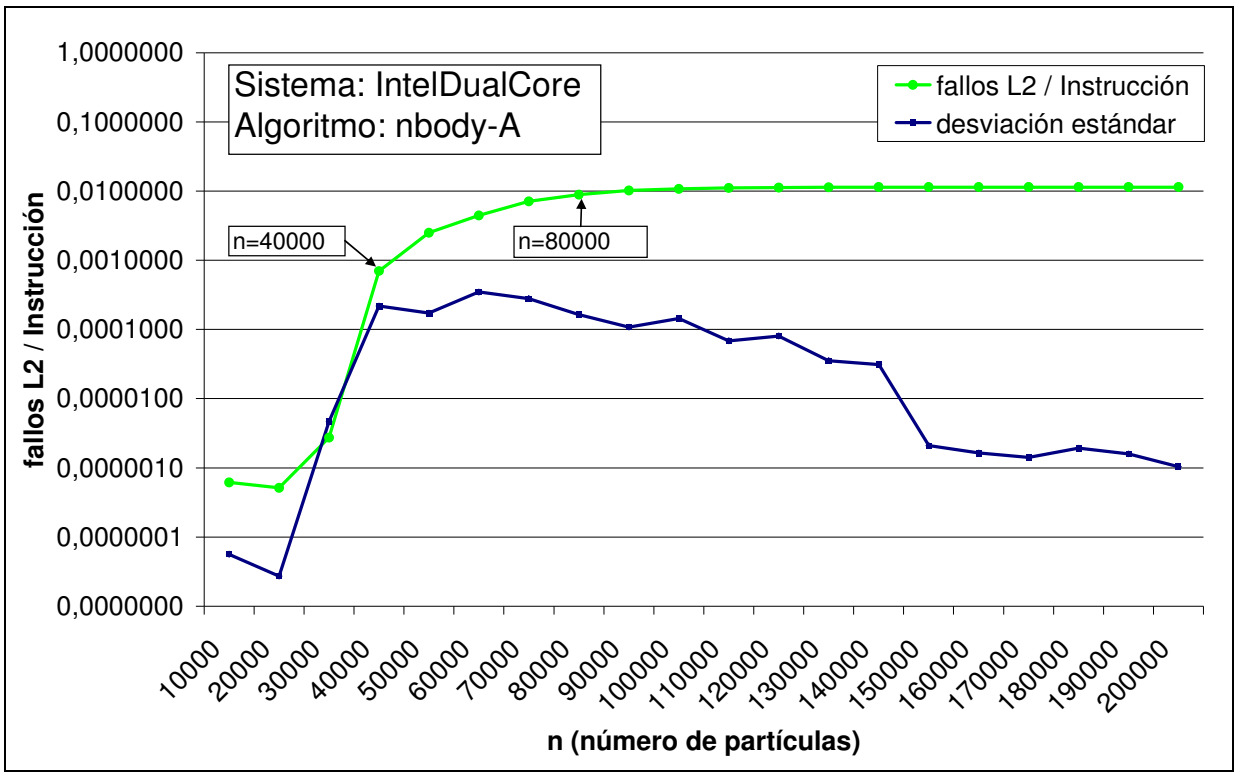



**Figura 32:** Fallos de caché en L2 divididos por el nº total de instrucciones ejecutadas y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador Intel, con **k**=1, y **n** menor de 200.000.

Antes de finalizar este apartado, notar que en la métrica (fallos de cache / intrucción) la escala al mostrar los resultados es importante. Si se utiliza escala lineal, los puntos en los que la pendiente de la gráfica cambia drásticamente, se pueden encontrar con facilidad, pero es prácticamente imposible discernir que ocurre en los tramos en los la pendiente de la gráfica cambia levemente. Si se utiliza escala logarítmica, se puden detectar con facilidad tanto los cambios drásticos de pendiente como los leves. Por contrapartida, al utilizar escala logarítmica la interpretación de la gráfica puede dar lugar a conclusiones erroneas. Especialmente si no se tiene muy presente el rango de valores en los que se encuentran los resultados. Por norma general, requiere un esfuerzo extra interpretar correctamente las gráficas logarítmicas.

### Resumen del Análisis de los fallos de caché

El tipo de problemas habituales con la jerarquía de memoria se suele manifestar de dos formas:

- Un incremento rápido en el volumen de fallos de caché hasta alcanzar un valor estable. Suele deberse al aumento del conjunto de los datos que experimentan localidad temporal, que se produce al aumentar el tamaño del problema a partir de un cierto valor. El tipo de estrategia para reducir el problema consiste en agrupar el acceso a los datos en bloques más pequeños.

- Un incremento puntual en el volumen de fallos de caché. Suele ser debido a conflictos entre varias secuencias de acceso a memoria que se encuentran a una cierta distancia crítica. Se puede reestructurar la posición de los datos, o bien intercalando los datos accedidos en secuencia o bien rellenando la memoria con huecos (“padding”).

## Tiempo dividido por Complejidad en el resto de Procesadores

A continuación se muestra el tiempo dividido por la complejidad para los otros tres procesadores, objeto de análisis en este trabajo. Los experimentos en el procesador SPARCT2 sólo se realizan hasta ***n***=100.000, debido al alto tiempo de ejecución que se alcanza con valores superiores.

Los resultados más significativos se pueden resumir así:

- En el procesador AMDOpteron se aprecian los mismos problemas de alineamiento que en el procesador AMDAthlon.
- En los procesadores Intel y en el SPARCT2 no se han detectado ineficiencias de rendimiento significativas.
- El rendimiento (de ejecución) de un núcleo es un 50% superior en los procesadores AMD (aproximadamente 30 ns por operación abstracta) que en el procesador Intel (unos 45 ns por operación abstracta), y 10 veces mejor que en el procesador SPARCT2 (300 ns)

En las figuras 33, 34, y 35 se muestran los (tiempos / complejidad) y la desviación estándar para la ejecución del algoritmo nbody-A, en los sistemas 2xAMDOpteronQuadCore, IntelDualCore, y SPARCT2, respectivamente.

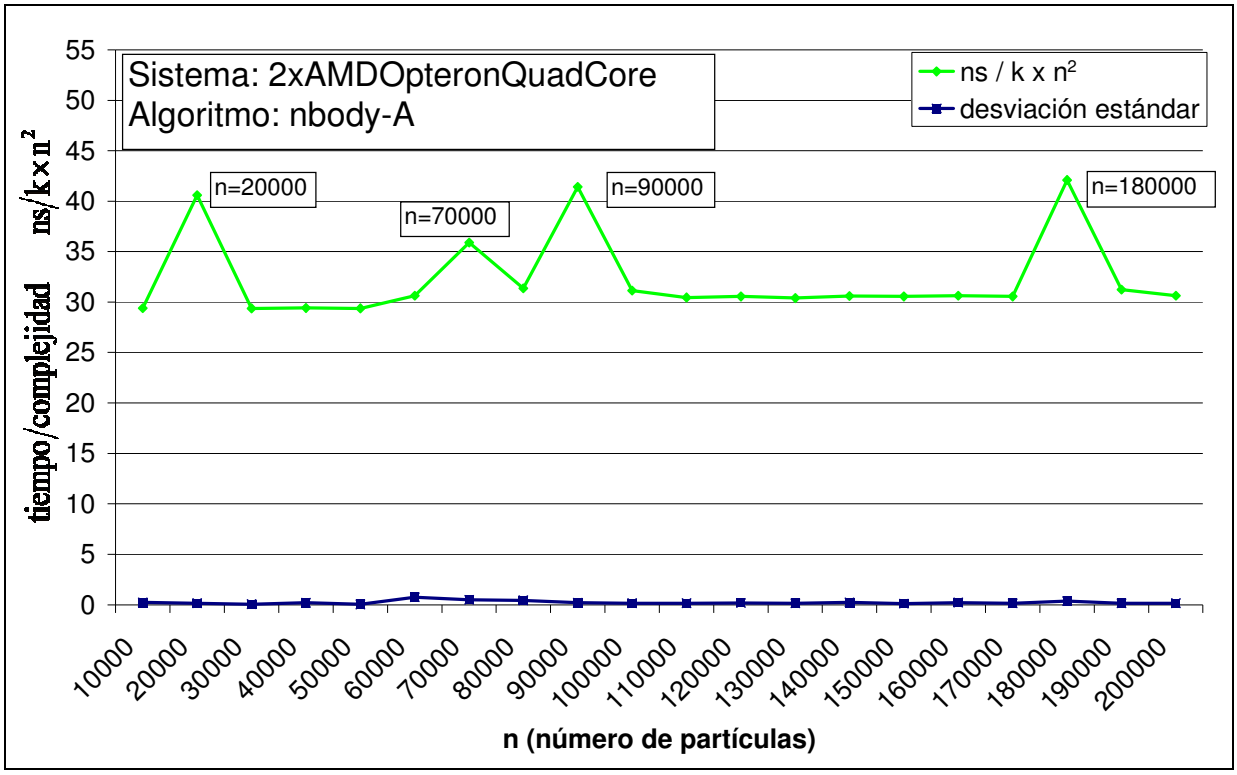



**Figura 33:** Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador AMDOpteron, con **k**=1, y **n** entre 10.000 y 200.000.

En la figura 33 se puede apreciar como en el 2xAMDOpteron aparecen problemas de rendimiento para los mismos tamaños de problema, que en el AMDAthlonDualCore.

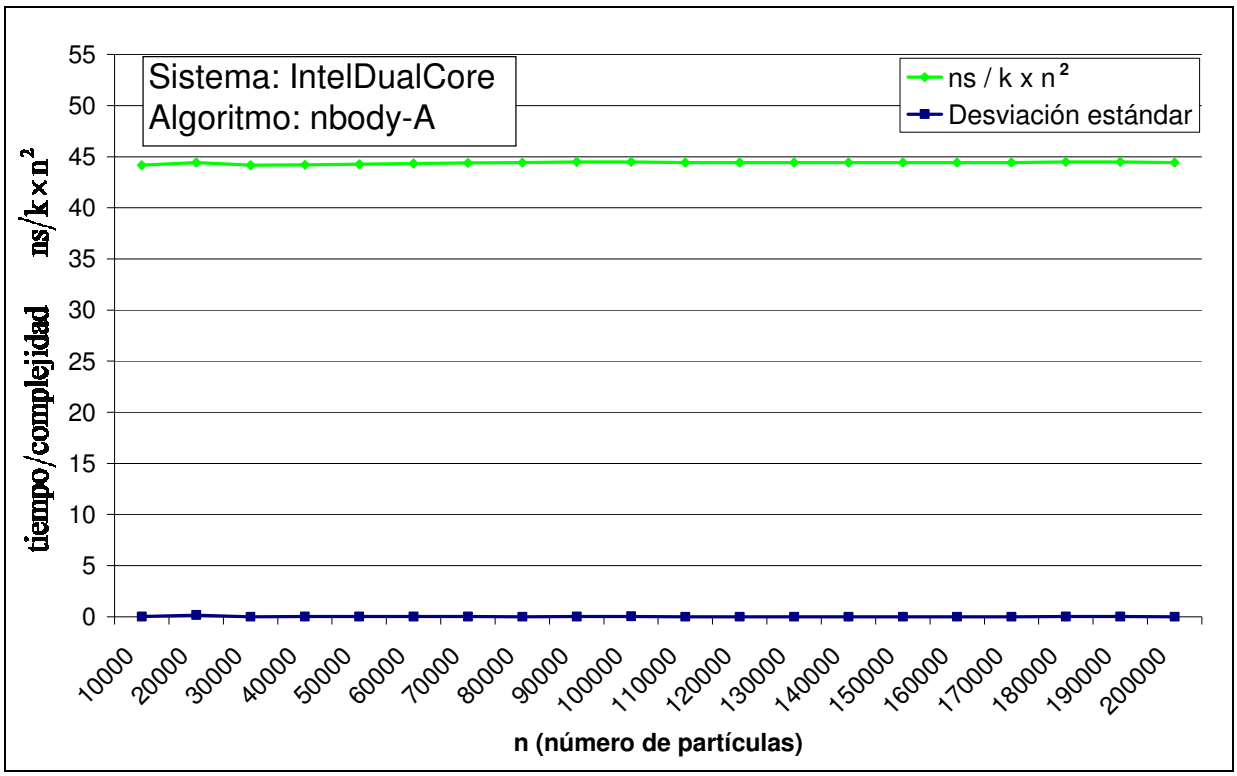



**Figura 34:** Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador Intel, con **k**=1, y **n** entre 10.000 y 200.000.

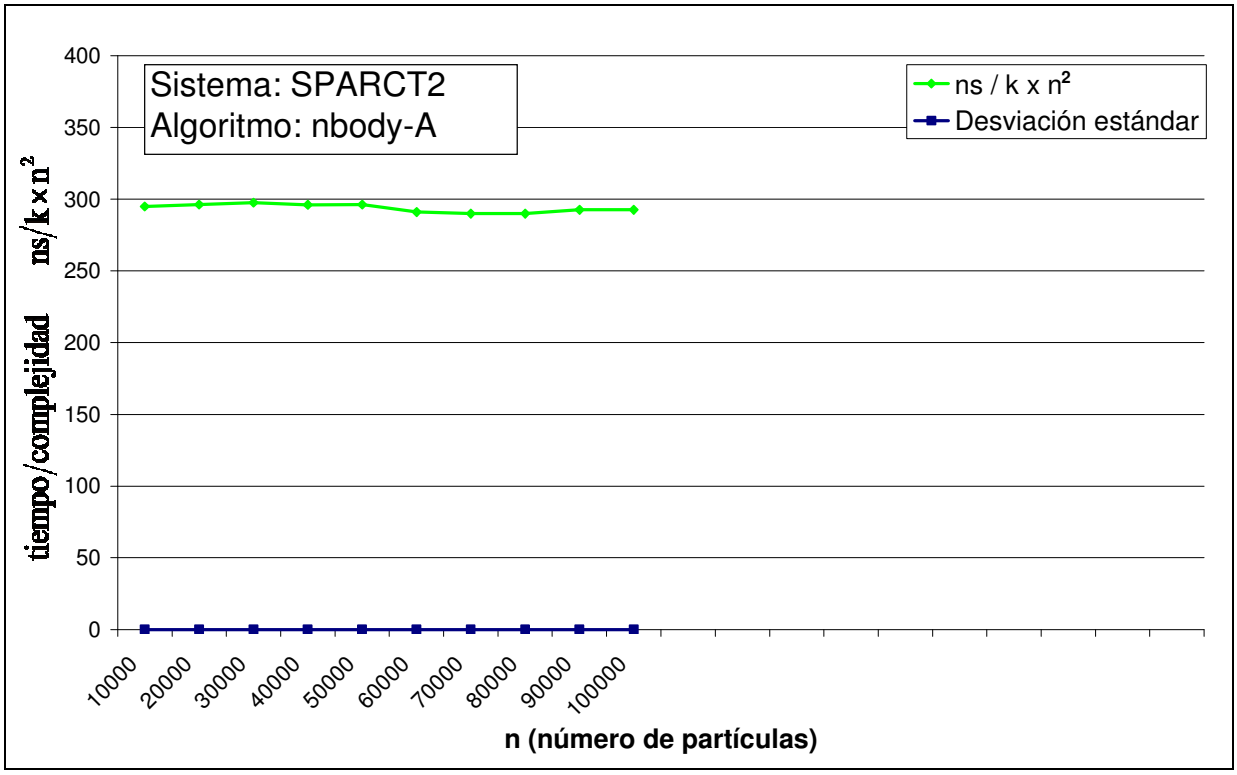



**Figura 35:** Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-A en el procesador SPARCT2, con **k**=1, y **n** entre 10.000 y 100.000.

## Análisis de los factores que determinan el rendimiento

Una métrica clásica para comprender los factores que determinan el tiempo de ejecución de una aplicación es el CPI (Ciclos por Instrucción ejecutada), o el valor similar de TPI (Tiempo por Instrucción ejecutada).

En los procesadores Intel y AMD, que disponen de acceso a los contadores H/W de rendimiento, se ha podido obtener el recuento del número total de instrucciones máquina ejecutadas. Dividiendo este total por la complejidad del algoritmo se determina la cantidad de instrucciones máquina que se ejecutan por operación abstracta del algoritmo. En los sistemas en los que no se dispone de acceso a contadores H/W de rendimiento, se ha analizado el código en ensamblador generado por el compilador y se han contado a mano las instrucciones dentro del bucle principal.

La cantidad de instrucciones ejecutadas por operación abstracta, se puede cruzar con el tiempo promedio por operación abstracta para obtener el tiempo por instrucción máquina (TPI), que a su vez se puede combinar con la frecuencia del reloj para obtener el CPI. Estos cálculos se encuentran en la tabla que se muestra en la figura 36.

| Procesador: | Frecuencia | ns / $kn^2$ | instr / $kn^2$ | ns / instr. (TPI) | ciclos / instr. (CPI) | CPI mínimo teórico |
|---|---|---|---|---|---|---|
| AMDAthlon | 2,0 Ghz | 31,4 | 36 | 0,873 | 1,747 | 0,33 |
| AMDOpteron | 2,11 Ghz | 29,8 | 36 | 0,828 | 1,747 | 0,33 |
| Intel | 2.4 Ghz | 44,2 | 42 | 1,053 | 2,526 | 0,33 |
| SPARCT2 | 1,165 Ghz | 295 | 96 | 3,071 | 3,579 | 0,5 |

**Figura 36:** Cálculo del Tiempo Por Instrucción (TPI) y de los ciclos por instrucción (CPI).

Por un lado, el análisis del recuento de instrucciones máquina ejecutadas nos muestra que la arquitectura AMD (junto al compilador gcc 4.3) es la más eficiente, mientras que el Intel (con el mismo compilador) ejecuta un 16% más instrucciones, y la arquitectura SUN (de filosofía RISC) ejecuta 2,6 veces más instrucciones (compilador cc).

Por otro lado, si consideramos el tiempo promedio por instrucción, el procesador Intel tarda alrededor de un 25% de tiempo más que los procesadores AMD, lo cual es bastante relevante, ya que la frecuencia de reloj del procesador Intel es un 20% superior. Por otro lado, el tiempo promedio por instrucción del procesador SPARCT2 es casi 4 veces mayor al del AMD, y la menor frecuencia de reloj del SPARCT2 sólo explica la mitad de la desventaja.

El análisis del CPI permite hacer comparaciones sin considerar la frecuencia de reloj. El AMD es el que menor CPI consigue, mientras que el Intel tiene un CPI que es un 44% mayor, y el SPARCT2 tiene un CPI de algo más del doble. Además, ninguno de los procesadores alcanza el límite teórico de CPI que se muestra en la última columna de la tabla (figura 36). En los cuatro casos, el cuello de botella al rendimiento no es la capacidad genérica de ejecución de instrucciones, sino la capacidad para ejecutar algún tipo concreto de operaciones, de entre las que contiene el bucle principal del programa.

Para determinar qué operaciones del algoritmo son responsables de consumir más tiempo de ejecución utilizamos dos estrategias. La primera analiza limitaciones en la capacidad de ejecución, y la segunda analiza las latencias del camino crítico de dependencias.

### Limitaciones de capacidad de ejecución

El perfil de rendimiento de los procesadores, obtenido en el apartado 4.5, nos proporciona datos cuantitativos de la capacidad máxima de ejecución para ciertos tipos de operaciones. En los apartados 4.3 y 4.4, se generó una tabla de necesidades de volumen de cómputo, clasificadas por tipo. Cruzando estos datos se pueden calcular los tiempos mínimos de ejecución impuestos por cada límite de tipo de operación.

La tabla que aparece en la figura 37, muestra en la segunda columna, el volumen de cómputo dividido por la complejidad (para el algoritmo nbody-A). En las siguientes columnas se muestran los tiempos por operación del procesador AMDAthlon, considerando tanto ejecución solapada como no solapada de operaciones, y el cálculo del tiempo mínimo (por operación abstracta de complejidad 1) que el procesador necesita para ejecutar las operaciones requeridas por el algoritmo. Se considera que las lecturas a memoria se obtienen de L2 o de Memoria, con un patrón de acceso secuencial. Como se verificó en apartados anteriores, esto ocurre para valores de n superiores a 4000 y a 80.000, respectivamente.

| | | AMDAthlonCore | | | |
|---|---|---|---|---|---|
| **Operación básica** | **Cómputo / $kn^2$** | **Latencia** | **Tiempo: 29,5 ns / $kn^2$** | **Ancho de Banda** | **Tiempo: 29,5 ns / $kn^2$** |
| SUMAent | 0 * | 0,5 | 0 | 0,167 | 0 |
| SLTcond | 2 | 0,167/ 4,47 | 1 × 0,167 + 1 × 4,47 = 4,64 | 0,167/ 4,47 | 1 × 0,167 + 1 × 4,47 = 4,64 |
| LecMemPf (stride=1) | 4 | 2,07 (L2)<br>3,47 (Mem) | 4 × 2,07 = 8,14<br>4 × 3,47 = 13,88 | 0,92 (L2)<br>2,50 (Mem) | 4 × 0,92 = 3,68<br>4 × 2,50 = 10,00 |
| SUMApf | 8 | 2,0 | 8 × 2,0 = 16,0 | 0,5 | 8 × 0,5 = 4,0 |
| MULTpf | 7 | 2,0 | 7 × 2,0 = 14,0 | 0,5 | 7 × 0,5 = 3,5 |
| DIVpf | 1 | 5,5 | 1 × 5,5 = 5,5 | 4,0 | 1 × 4,0 = 4,0 |
| RAIZpf | 1 | 13,5 | 1 × 13,5 = 13,5 | 12,0 | 1 × 12,0 = 12,0 |
| | | | Σ = 53,64 + 8,14 (L2) / 13,9 (M) | | Σ = 28,14 + 3,68 (L2) / 10,0 (M) |

**Figura 37:** Límites de tiempo de ejecución dividido por complejidad calculados a partir de los perfiles de rendimiento del procesador AMD Athlon.

Si consideramos las latencias de las operaciones (suponiendo que la ejecución de operaciones básicas del mismo tipo no se solapa) y las sumamos (suponiendo que la ejecución de operaciones básicas de diferente tipo tampoco se solapa), obtenemos un valor (en la última fila de la cuarta columna) que duplica el tiempo total de ejecución (siempre considerando tiempos divididos por la complejidad). Por tanto, hemos de concluir que existe solapamiento en la ejecución de las operaciones.

Por otro lado, si suponemos que la ejecución de operaciones básicas del mismo tipo sí se solapa, se han de utlizar los valores de tiempo etiquetados como de “ancho de banda”. Si también suponemos que la ejecución de operaciones básicas de diferente tipo también se solapa, entonces hemos de calcular el valor máximo de los obtenidos en la sexta columna, que sería de 12 ns. Este valor es claramente inferior al tiempo total de ejecución, lo que indica que no existe tanto grado de solapamiento. Por otro lado, si suponemos que la ejecución de operaciones básicas del mismo tipo se solapa y la de operaciones de diferente tipo no se solapa, se deben sumar los valores de la sexta columna, y se obtiene el valor mostrado en la última fila, que aproxima bastante el tiempo de ejecución si excluimos las operaciones de lectura de memoria (que son las que típicamente se solapan con el resto de operaciones en un procesador con ejecución fuera de orden).

La tabla de la figura 38 muestra la misma información que la tabla de la figura 37, pero con datos obtenidos para el procesador SPARCT2. Para los valores de ***n*** considerados, los accesos a memoria siempre se resuelven en la caché L2, y por tanto sólo se consideran las latencias de acceso a memoria para este caso. La tabla muestra que la suma de tiempos de las latencias de las operaciones básicas (316,5) es similar (algo superior) al tiempo de ejecución medido, mientras que el máximo o la suma de los anchos de banda de ejecución (123,4 y 230,7) son muy inferior o bastante inferior al tiempo de ejecución. Por tanto, en este procesador hay mucha menos proporción de solapamiento en la ejecución de las operaciones que en el procesador AMD, lo cual era previsible, dado el modelo estático y en orden de planificación de las instrucciones que implementa el SPARCT2.

| | | SPARCT2core | | | |
|---|---|---|---|---|---|
| Operación básica | Cómputo / $kn^2$ | Latencia | Tiempo: 295 ns / $kn^2$ | Ancho de Banda | Tiempo: 295 ns / $kn^2$ |
| SUMAent | 0 * | 1,01 | 0 | 1,01 | 0 |
| SLTcond | 2 | 2,57/ 6,86 | 1 × 2,57 + 1 × 6,86 = 9,43 | 2,57/ 6,86 | 1 × 2,57 + 1 × 6,86 = 9,43 |
| LecMemPf (stride=1) | 4 | 11,0 (L2) | 4 × 11,0 = 44,0 | 11,0 (L2) | 4 × 11,0 = 44,0 |
| SUMApf | 8 | 5,15 | 8 × 5,15 = 42,20 | 1,46 | 8 × 1,46 = 11,86 |
| MULTpf | 7 | 5,15 | 7 × 5,15 = 36,05 | 1,46 | 7 × 1,46 = 10,22 |
| DIVpf | 1 | 31,8 | 1 × 31,8 = 31,8 | 31,8 | 1 × 31,8 = 31,8 |
| RAIZpf | 1 | 153,0 | 1 × 153,0 = 153,0 | 123,4 | 1 × 123,4 = 123,4 |
| | | | Σ = 316,5 | | Σ = 230,7 |

**Figura 38:** Límites de tiempo de ejecución dividido por complejidad calculados a partir de los perfiles de rendimiento del procesador SPARCT2.

### Latencias del camino crítico de dependencias

El grado de solapamiento en la ejecución de las operaciones depende de la capacidad del procesador para solapar instrucciones (disponibilidad de múltiples unidades de ejecución, de capacidad de gestión de múltiples instrucciones, de ejecución fuera de orden, tamaño de las colas de espera, …) pero también del paralelismo que exista entre las operaciones básicas. Este paralelismo se refleja en el camino crítico que se analizó en los apartados anteriores y que se muestra a continuación para el algoritmo nbody-A:

{ LecMem, LecMem, LecMem } →
{ SUMApf, SUMApf, SUMApf } →
{ MULTpf, MULTpf } →
SUMApf →
SUMApf →
RAIZpf →
MULTpf →
DIVpf →
{ MULTpf, MULTpf, MULTpf } →
{ SUMApf, SUMApf, SUMApf }

La segunda estrategia de análisis de los factores que determinan el rendimiento consiste en modificar el programa original para eliminar del camino crítico ciertas operaciones seleccionadas, ejecutar la nueva versión y medir el nuevo tiempo de ejecución, y así calcular el coste en tiempo asociado a la instrucción. Supongamos que la versión original de la aplicación es A y la aplicación modificada para eliminar un tipo de operación o evento, cuyo impacto se pretende medir, es Am. Tenemos TA = Tiempo de ejecución de A y TAm = Tiempo de ejecución de Am. El % de tiempo consumido por nuestra aplicación provocado por el evento se puede calcular de la siguiente manera: (TA - TAm)*100 / TA.

Las Figuras 39, 40 y 41 muestran la reducción en el tiempo de ejecución (normalizado a complejidad 1) debida a la eliminación de las operaciones de SUMApf, MULTpf, DIVpf y RAIZpf, en tres procesadores: AMDAthlon, Intel y SPARCT2.

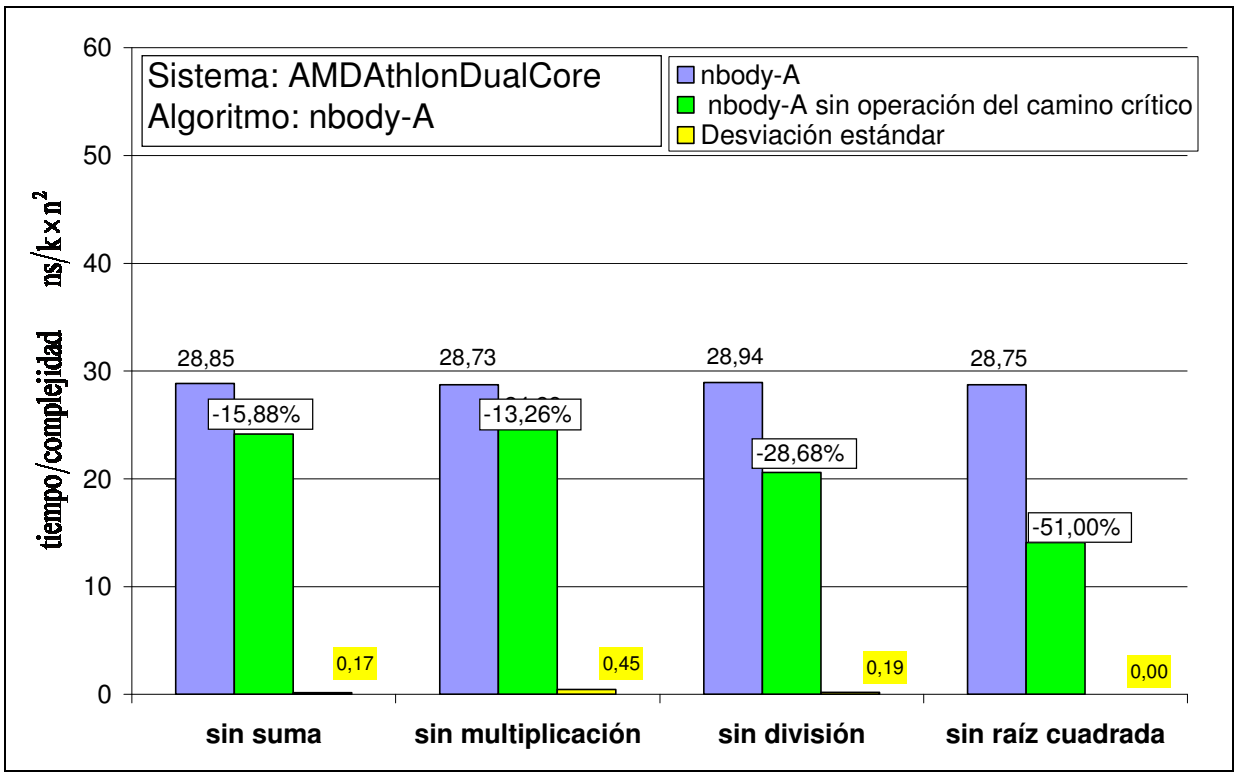



**Figura 39:** Tiempo de ejecución dividido por complejidad y desviación estándar, para nbody-A en el procesador AMDAthlon cuando se eliminan operaciones seleccionadas ( **k**=1, **n**=10.000).

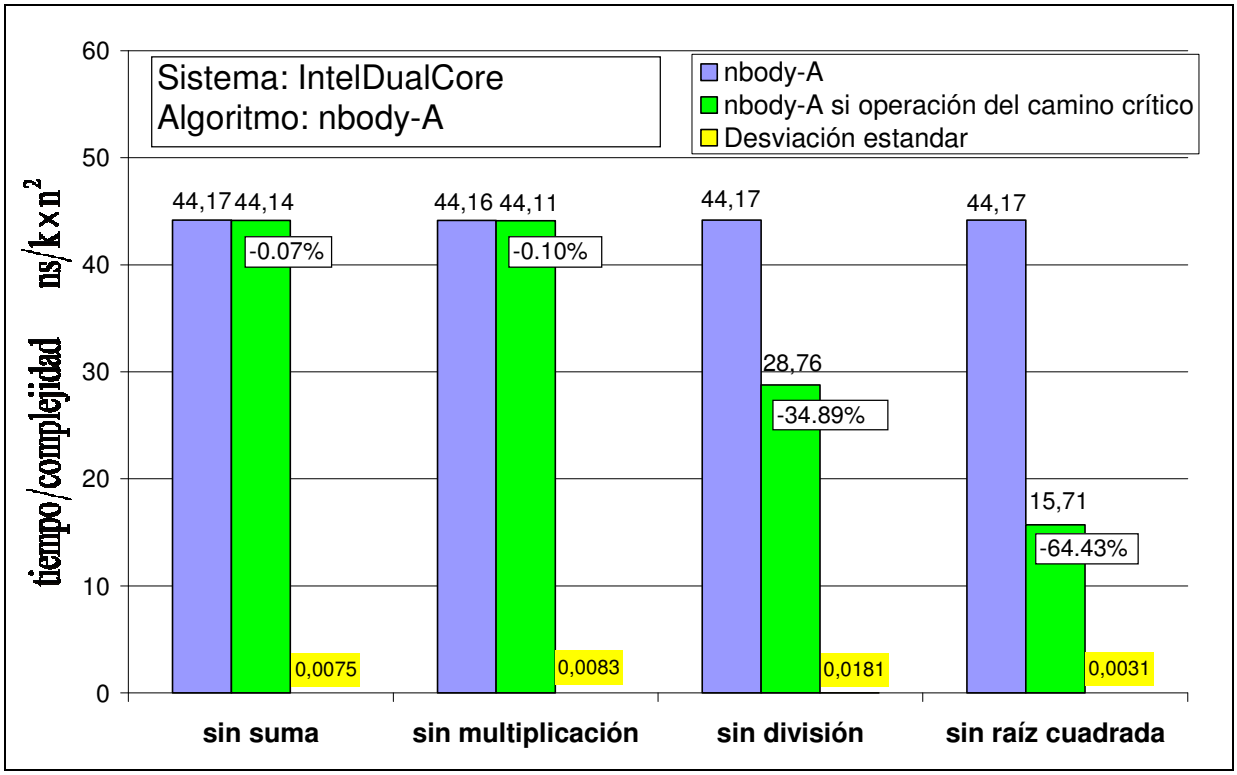



**Figura 40:** Tiempo de ejecución dividido por complejidad y desviación estándar, para nbody-A en el procesador Intel cuando se eliminan operaciones seleccionadas ( **k**=1, **n**= 10.000).

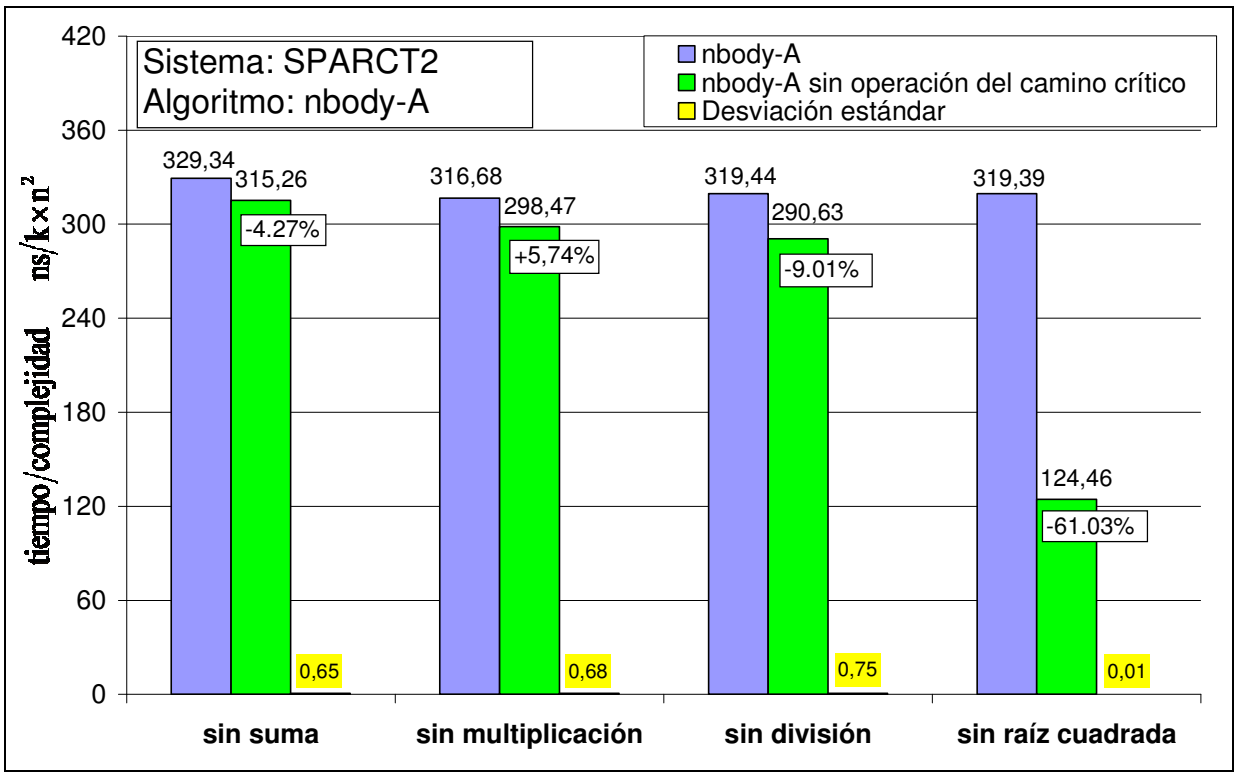



**Figura 41:** Tiempo de ejecución dividido por complejidad y desviación estándar, para nbody-A en el procesador SPARCT2 cuando se eliminan operaciones seleccionadas ( **k**=1, **n**=10.000).

En los tres procesadores la raíz cuadrada consume más del 50% (AMD 51%, Intel 64.43%, SUN 61.03%) del tiempo necesario para ejecutar la operación básica. El impacto de la división es mayor en el AMD y en el Intel, que en la SUN, (AMD 28.68%, Intel 34.89%, SUN 9.01% del tiempo necesario para ejecutar la operación básica). La explicación de que la raíz cuadrada y la división tengan un elevado impacto en el rendimiento es que, por un lado, son operaciones de alto coste y que, por otro lado, aparecen en el camino crítico sin la posibilidad de solaparse con otras operaciones.

## 4.8. Resultados para nbody-F

A continuación se mostrarán los resultados obtenidos para el algoritmo nbody-F. En este caso no se realiza el análisis de fallos de caché ni se estudian con detalle los factores que afectan al rendimiento, porque no aportan conocimiento adicional al ya obtenido hasta el momento.

Posteriormente se presentan unas gráficas resumen, de rendimientos obtenidos en los cuatro nodos, para los algoritmos nbody-A y nbody-F y se comparan y explican los resultados entre los diferentes procesadores y entre algoritmos.

En las figuras 42, 43, 44 y 45 se muestran los (tiempos / complejidad) y la desviación estándar para la ejecución del algoritmo nbody-F, en los sistemas AMDAthlonDualCore, 2xAMDOpteronQuadCore, IntelDualCore, y SPARCT2, respectivamente.

Los resultados se comentarán en la sección siguiente.

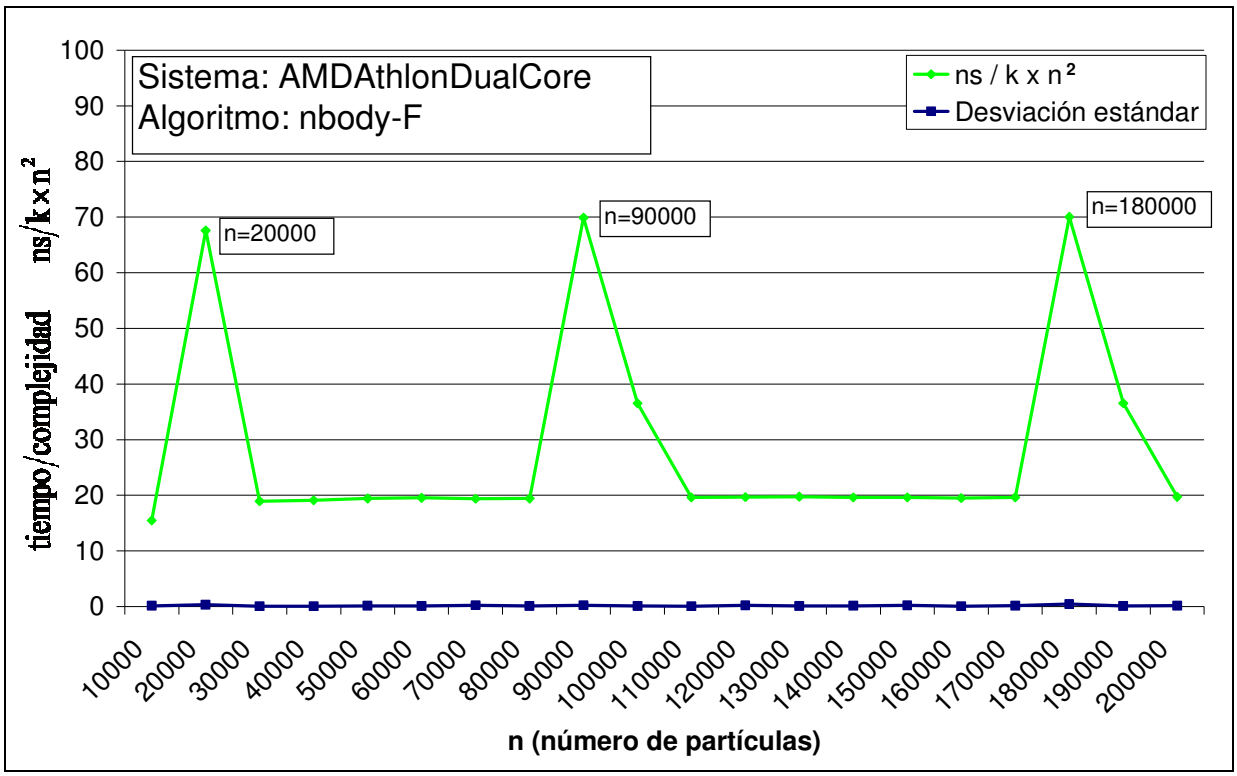



**Figura 42:** Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-F en el procesador AMDAthlon, con **k**=1, y **n** entre 10.000 y 200.000.

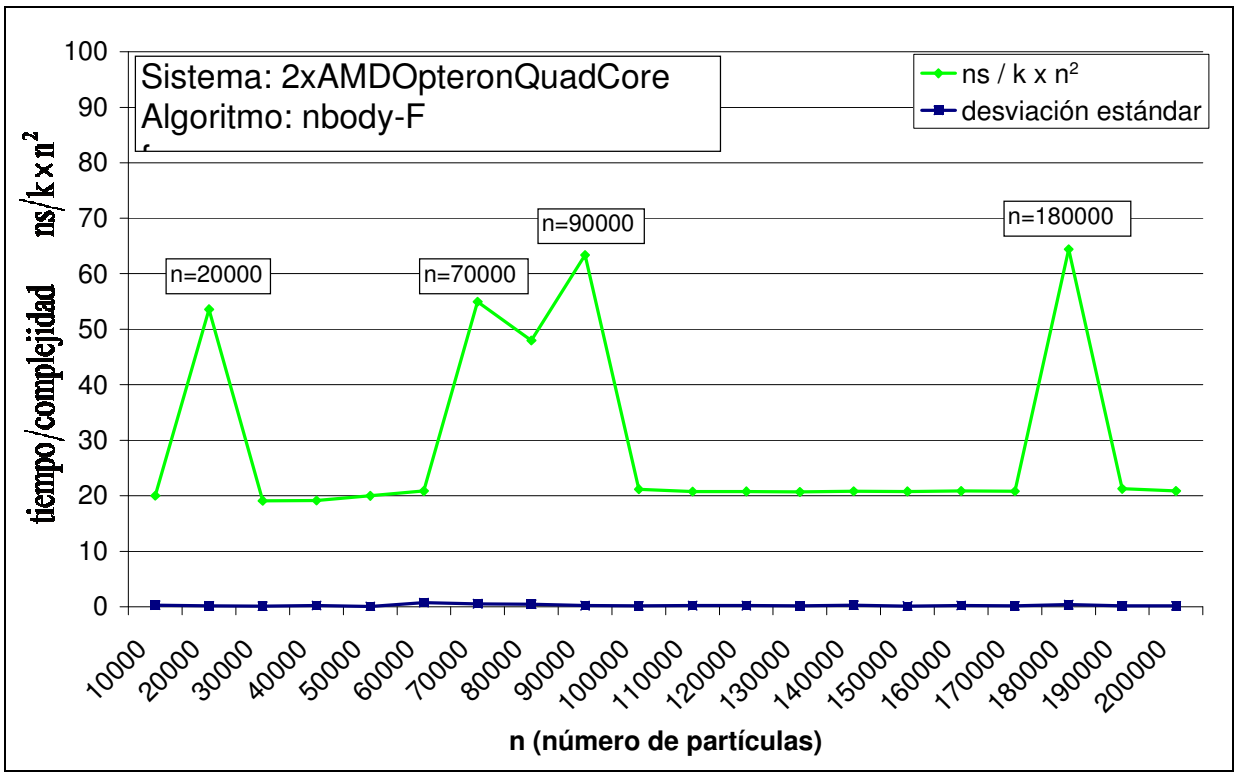



**Figura 43**: Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-F en el procesador AMDOpteron, con **k**=1, y **n** entre 10.000 y 200.000.

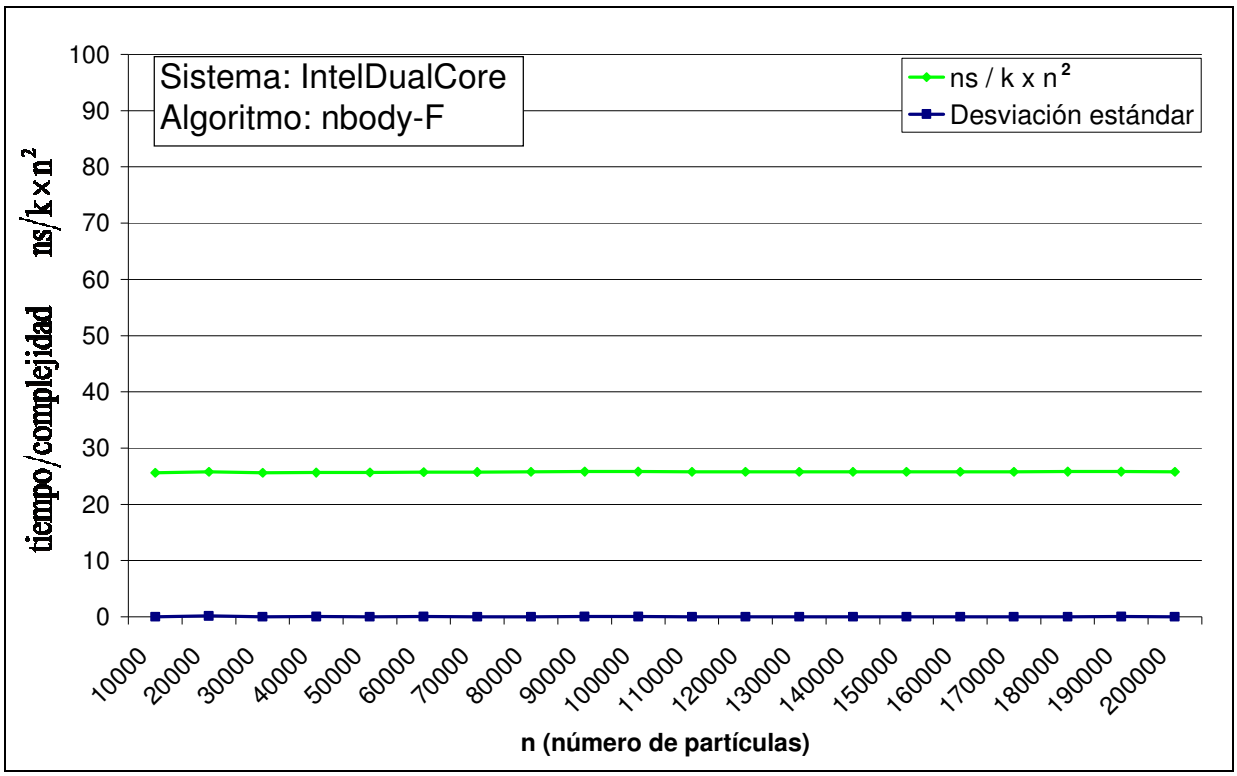



**Figura 44:** Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-F en el procesador Intel, con **k**=1, y **n** entre 10.000 y 200.000.

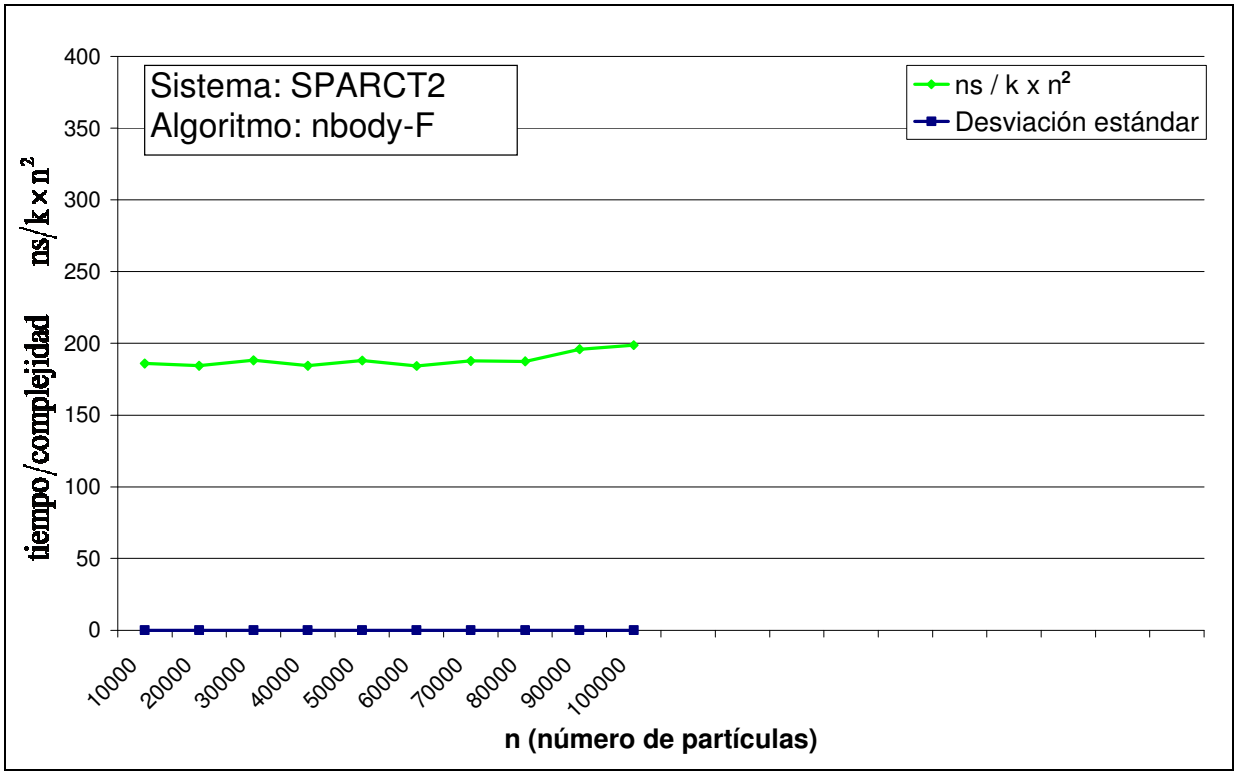



**Figura 45:** Tiempo de ejecución dividido por complejidad y desviación estándar, para la ejecución del algoritmo nbody-F en el procesador SPARCT2, con **k**=1, y **n** entre 10.000 y 100.000.

## Gráficas Resumen de Tiempo dividido por Complejidad

En este apartado presentamos los valores obtenidos para las métricas (tiempo / complejidad) para cada uno de los nodos estudiados. Se presentan dos gráficas, la primera corresponde con las ejecuciones del algoritmo nbody-A en los cuatro nodos y la segunda con las ejecuciones del algoritmo nbody-F. Posteriormente se comparan y explican los resultados entre los diferentes procesadores y entre algoritmos.

En las gráficas de las figuras 46 y 47 se muestra el (tiempo / complejidad) para las ejecuciones de las variantes nbody-A y nbody-F respectivamente, en los cuatro sistemas analizados.

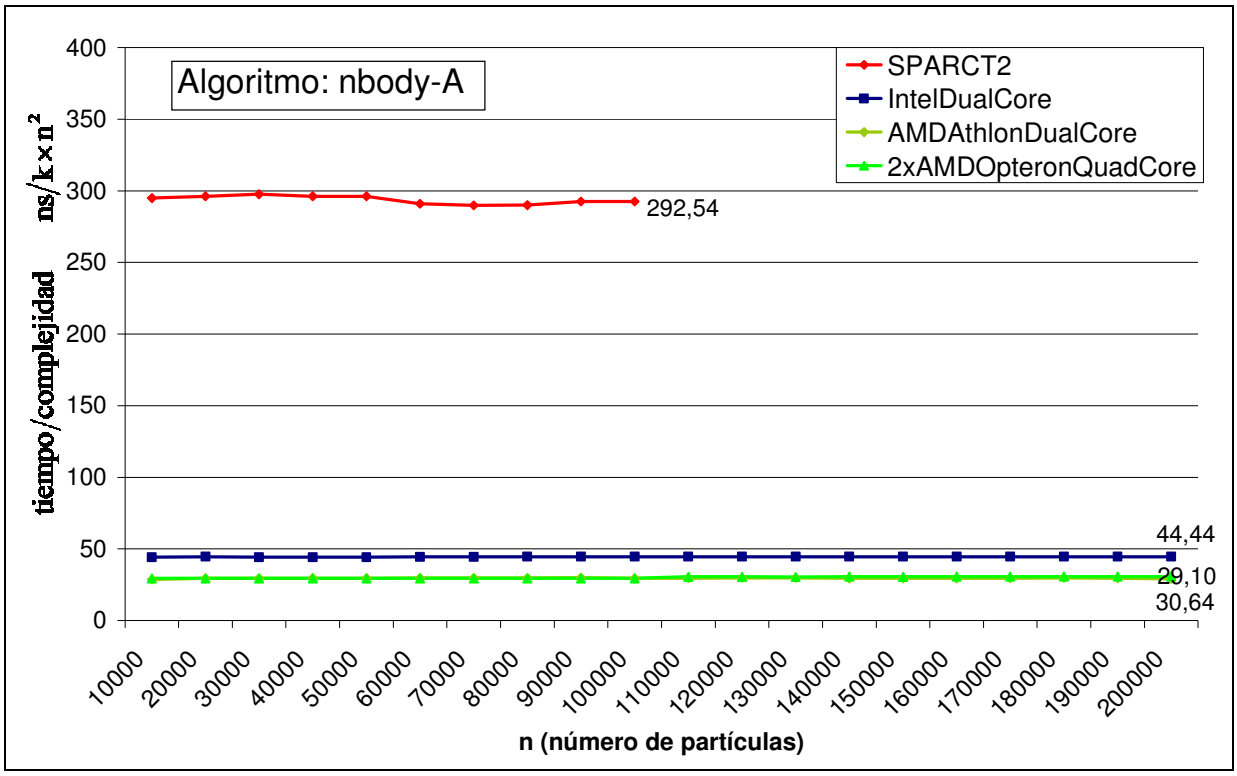



**Figura 46:** Tiempo de ejecución dividido por complejidad, para la ejecución del algoritmo nbody-A, en los procesadores SPARCT2, Intel, AMDAthlon, y 2xAMDOpteron, con **k**=1, y **n** entre 10.000 y 200.000.

A partir de las gráficas de las figuras 46 y 47, se puede apreciar que tanto utilizando el algoritmo nbody-A, como el nbody-F el rendimiento obtenido en el nodo SPARCT2 es aproximadamente 10 veces inferior al obtenido en los nodos AMD, y unas 7 veces inferior que el obtenido en el nodo IntelDualCore. Esta diferencia de rendimiento es debida principalmente a que el nodo SPARCT2 ejecuta las instrucciones en orden, mientras el resto de nodos es capaz de ejecutar instrucciones fuera de orden. Otros factores que también influyen en el bajo rendimiento del nodo SPARCT2, son una menor de frecuencia de reloj y un mayor número de instrucciones generadas por el compilador.

Los cuatro nodos son multi-core, pero en estos algoritmos secuenciales únicamente pueden aprovechar uno de sus núcleos (cores). Además en el SPARCT2, cada uno de sus cores es multi-thread, pudiendo ejecutar hasta 8 threads por core. En estas ejecuciones secuenciales, el SPARCT2, a parte de únicamente poder aprovechar un core, únicamente puede aprovechar en muchos de los casos la mitad de la capacidad de cómputo de que dispone el core, y no puede aprovechar la oportunidad de usar hasta 8 threads para solapar las esperas entre instrucciones de los threads.

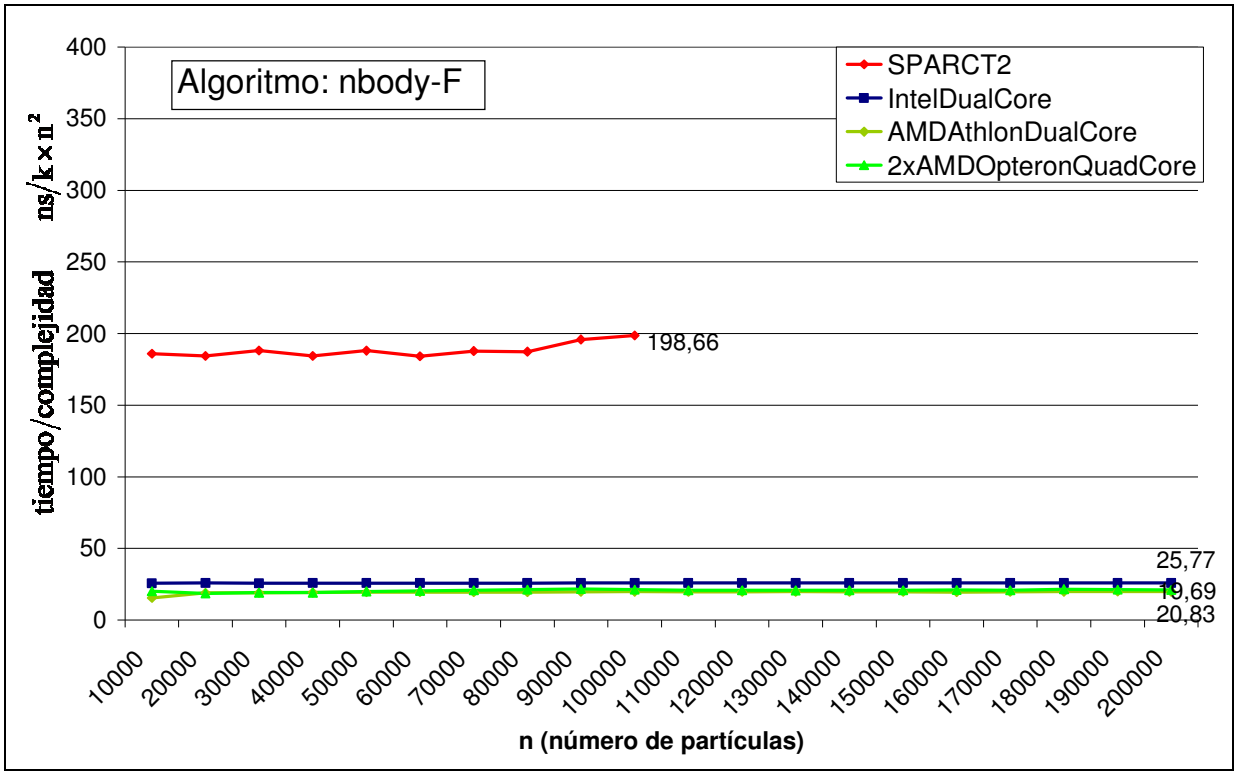



**Figura 47:** Tiempo de ejecución dividido por complejidad, para la ejecución del algoritmo nbody-F, en los procesadores SPARCT2, Intel, AMDAthlon, y 2xAMDOpteron, con **k**=1, y **n** entre 10.000 y 200.000.

El speedup obtenido al utilizar la variante nbody-F en lugar de la nbody-A es de 1,47 para los nodos SPARCT2, AMDAthlonDualCore, y 2xAMDOpteronQuadCore. En cambio en el nodo IntelDualCore, el speedup obtenido es de 1,76. Con el algoritmo nbody-F se obtienen mejores rendimientos porque se realizan la mitad de cálculos de fuerzas entre partículas, aunque para ello requiere de tres vectores de fuerzas ffx, ffy, ffz donde ir acumulando sumatorios parciales de fuerzas reciprocas.

Comparando los algoritmos nbody-A y nbody-F (ver figuras 18 y 21, y explicación de la página 38) se observa una reducción a casi la mitad de muchas de las operaciones de cómputo, sobre todo las más complejas de DIVpf y RAIZpf, pero las operaciones SUMApf y MULTpf se dividen sólo por 1,45 y 1,75. Pero lo que es más importante, las lecturas a memoria casi son las mismas, y aparecen escrituras a memoria, que provocan que el total de accesos sea un 25% mayor en nbody-F que en nbody-A. Esta podría ser la explicación a que el speeup del algoritmo nbody-F no sea tan cercano a 2 como podría esperarse.

En los nodos que estamos analizando, el tiempo de acceso a la memoria es muy alto en comparación con el tiempo que tarda el procesador en ejecutar una instrucción. Este es el motivo principal por el cual, con el algoritmo nbody-F no se consigue un speedup cercano a 2. Por lo que cuanto mayor sea el speedup obtenido, menor es la diferencia entre tiempo de acceso a memoria y tiempo de ejecución por instrucción del nodo.
Según los speedups obtenidos, el IntelDualCore es el nodo en el que esta diferencia entre tiempo de acceso a memoria y tiempo de ejecución por instrucción es menor.

# 5 Estudio del problema paralelo

Para lograr un buen rendimiento se trata principalmente de evitar las dependencias de datos innecesarias, de reducir el uso de regiones críticas, de reducir el número de operaciones de sincronización, de reducir el volumen de comunicaciones entre los threads, y finalmente de gestionar de forma apropiada el balanceo de carga.

En los apartados 5.1 y 5.2 se estudiará la paralelización funcional de los algoritmos nbody-A y nbody-F obtenidos anteriormente y se analizará y se tratará de mejorar su rendimiento teórico. Para asegurar una correcta funcionalidad hay que mantener las dependencias de datos durante la ejecución paralela. Para lograr un buen rendimiento se trata principalmente evitar las dependencias de datos innecesarias, de reducir el uso regiones críticas, de reducir el número de operaciones de sincronización, de rehuir el volumen de comunicaciones entre los threads, y finalmente de gestionar de forma apropiada el balanceo de carga.

A continuación en el apartado 5.3 se mostrarán y comentarán los resultados de la ejecución de ambas aplicaciones completas.

Finalmente en el apartado 5.4 se mostrarán los resultados de ejecuciones específicas. El objetivo de estas ejecuciones es explicar cuantitativamente los resultados de la ejecución de ambas aplicaciones completas.

## 5.1. Paralelización de nbody-A

A la hora de ejecutar esta aplicación en un computador paralelo encontramos las siguientes dependencias de datos:
(Siendo Ci : vector de coordenadas, y Vi : vector de velocidad)

Para t=1..k
  Para i=1..n
    Leer **$P_i$ (en t-1)** y **$V_i$ (en t-1)** → Escribir **$V_i$ (en t)**
  Para i=1..n
    Leer **$P_i$ (en t-1)** y **$V_i$ (en t)** → Escribir **$P_i$ (en t)**

Esto nos impone dos restricciones en el momento de repartir el trabajo:

- La primera y más restrictiva es que para cada paso t a simular en el que queremos obtener la nueva posición de cualquier cuerpo (**$P_i$** en t), es necesario conocer el resultado de la simulación anterior de las posiciones de todos los cuerpos (**$P_i$** en t-1), esto imposibilita la paralelización de las k simulaciones, ya que es necesaria una sincronización de todos los procesos después de actualizar **$P_i$** en t.

- La segunda es que para calcular el vector de velocidades **$V_i$** en t de cualquier cuerpo es necesario conocer la posición de todos los cuerpos **$P_i$** en t-1, por lo que no se pueden modificar los **$P_i$** obtenidos en t-1 hasta que no se haya calculado **$V_i$** en t , lo que implica que para calcular **$P_i$** en t es necesaria una sincronización de todos los procesos después de actualizar **$V_i$** en t

Por lo que si queremos simular k pasos tendremos que realizar 2 x k sincronizaciones
Estas sincronizaciones las podemos ver gráficamente en la figura 48 de ejecución en un computador paralelo de la alternativa A.

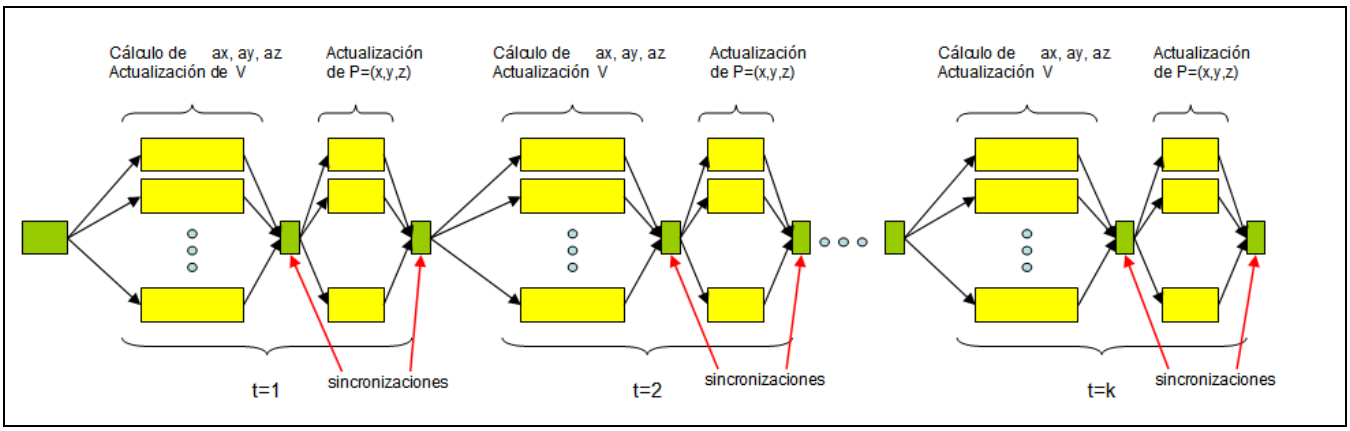



**Figura 48:** Ejecución en un computador paralelo de la alternativa A.

Es interesante eliminar las dependencias de datos que obligarían a sincronizar los diferentes procesos entre los que repartiríamos el trabajo o si esto no es posible, evitar las sincronizaciones. En nuestro algoritmo estas sincronizaciones están entre iteraciones y dentro de cada iteración, es decir tenemos $2 \times k$ sincronizaciones.

Para evitar la primera de las dos sincronizaciones se puede añadir un buffer auxiliar y unir las iteraciones por parejas, de esta manera queda una única sincronización por cada iteración. En la figura 49 se muestran las sincronizaciones utilizando un buffer auxiliar.

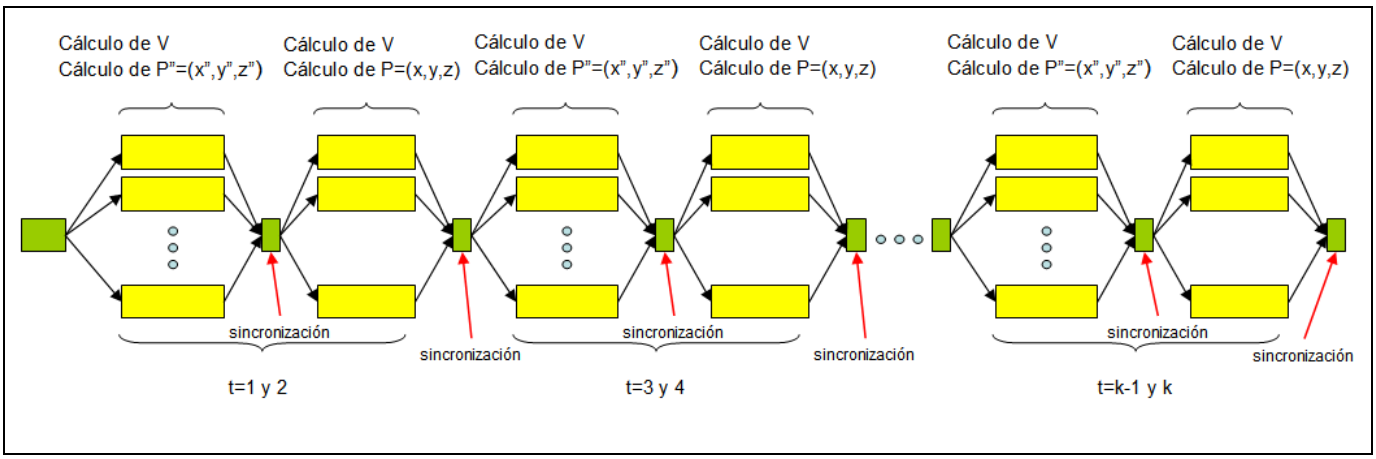



**Figura 49:** Ejecución en un computador paralelo del la alternativa A"paralela" que consiste en la alternativa A a la que se le ha añadido un buffer auxiliar y se han unido las iteraciones por parejas.

Esta solución, cuyo pseudo-código se muestra a continuación, tiene dos desventajas, la primera es que obliga a que k sea un número par, y la segunda es que requiere de un buffer auxiliar del mismo tamaño que P (Posiciones).
La principal ventaja es que evita la sincronización entre la actualización de V y de P.
De las dos restricciones que habíamos detectado, se ha resuelto la segunda, eliminado la necesidad de sincronización dentro de cada iteración, quedándo únicamente las sincronizaciones entre iteraciones (ahora se realizarán k en lugar de **2** x **k** sincronizaciones).
Respecto al número de operaciones se ha reducido en número de lecturas en memoria en **3** x **k** x **n**, a costa de aumentar el número de lecturas de variables en **3** x **k** x **n**.

**Para t=1..k , t=t+2**(Para cada dos iteraciones de tiempo de paso dt)
**{**
**EN PARALELO Para i=1..n (Para cada cuerpo i)**
**{ Ax = Ay = Az = 0**
**Pxi = Px[i]; Pyi = Py[i]; Pzi = Pz[i];**
Para j=1..n (siendo i diferente de j) (Para el resto de cuerpos diferentes al cuerpo i)
{
dx = Px [j] – Pxi
dy = Py [j] – Pyi
dz = Pz [j] – Pzi
dist2 = dx × dx + dy × dy + dz × dz
dist = √ dist2
ac_rec = m [j] / (dist × dist2)
Ax = Ax + ac_rec × dx
Ay = Ay + ac_rec × dy
Az = Az + ac_rec × dz
}
Vx [i] = Vx [i] + Gdt × Ax
Vy [i] = Vy [i] + Gdt × Ay
Vz [i] = Vz [i] + Gdt × Az
**}**
Para i=1..n **(Para cada cuerpo i)**
{
Px2 [i] = Px [i] + dt × Vx [i]
Py2 [i] = Py [i] + dt × Vy [i]
Pz2 [i] = Pz [i] + dt × Vz [i]
}
**EN PARALELO Para i=1..n (Para cada cuerpo i)**
**{ Ax = Ay = Az = 0**
**Pxi = Px2[i]; Pyi = Py2[i]; Pzi = Pz2[i]**
Para j=1..n (siendo i diferente de j) (Para el resto de cuerpos diferentes al cuerpo i)
{
dx = Px2 [j] – Pxi
dy = Py2 [j] – Pyi
dz = Pz2 [j] – Pzi
dist2 = dx × dx + dy × dy + dz × dz
dist = √ dist2
ac_rec = m [j] / (dist × dist2)
Ax = Ax + ac_rec × dx
Ay = Ay + ac_rec × dy
Az = Az + ac_rec × dz
}
Vx [i] = Vx [i] + Gdt × Ax
Vy [i] = Vy [i] + Gdt × Ay
Vz [i] = Vz [i] + Gdt × Az
**}**
Para i=1..n **(Para cada cuerpo i)**
{
Px [i] = Px2 [i] + dt × Vx [i]
Py [i] = Py2 [i] + dt × Vy [i]
Pz [i] = Pz2 [i] + dt × Vz [i]
}
**}**

## Modelo del algoritmo nbody-A paralelo

A continuación se analizarán los tres aspectos (sincronización, comunicación y balanceo del volumen de cómputo) descritos en apartado 5.1.

### Sincronización

En el algoritmo nbody-A paralelo, al utilizar un buffer auxiliar se evita la sincronización entre la actualización de Vx, Vy, Vz y la actualización de Px, Py, Pz, por lo que se realizarán k sincronizaciones.

### Comunicación

Cuando un thread de una región paralela escribe en una variable y posteriormente cualquiera de los otros threads de la región paralela necesita leer dicha variable, el sistema debe asegurar que el valor que se lee es el correcto. Las variables lógicas del programa se asignan a posiciones de memoria, que en un determinado momento del tiempo pueden corresponder con el contenido de un registro físico de uno de los procesadores, o de una caché privada de un procesador, o de una posición de la memoria principal compartida. Por lo tanto, para que un thread lea el resultado producido por otro thread, puede ser necesario mover los datos entre los procesadores y las memorias caché. A este movimiento de datos entre threads le denominamos comunicación entre threads, y es importante determinar su volumen.

En cuanto a las comunicaciones, un mismo thread tiene dos comportamientos. Los threads generan datos (escribiendo resultados en memoria) y consumen datos (leyendo datos de la memoria, que previamente han sido escritos por otros threads).

El volumen de datos escrito por cada thread y que posteriormente es leído por otros threads se corresponde con (k x n/th x 3 doubles). Cada thread actualiza 3 doubles (n/th) en cada una de las k iteraciones, que posteriormente serán leidos por otros threads.

El volumen de datos leídos por cada thread, que previamente han sido escritos por otros threads se corresponde con (k x n(th-1)/th x 3 doubles). En este caso cada thread lee 3 doubles (n x (th-1)/th) en cada una de las k iteraciones.

Los 3 doubles se corresponden a las coordenadas x, y, z del vector de posición P(x,y,z).

El patrón de comunicaciones entre threads es de todos con todos, es decir cada uno de los threads escribe datos en memoria que todos los threads leerán y cada thread lee datos escritos por cada uno de los threads restantes.

### Balanceo del volumen de cómputo

En esta variante del algoritmo la asignación del volumen de cómputo a cada thread es equitativa. El bucle que se reparte entre los threads esta compuesto de n x (n - 1) cálculos de aceleraciones entre partículas. Al repartir el bucle a cada thread le corresponden n / th x (n-1) cálculos de aceleraciones. Por tanto, el volumen de cómputo que le corresponde a cada thread será k x n x (n - 1) / th.

### Requerimientos de memoria

Los requerimientos de memoria crecen linealmente respecto a n. No dependen ni del número de threads, ni del número de iteraciones k. Concretamente se necesita un vector de masas m, un vector de coordenadas de velocidad V(x,y,z) y dos vectores de coordenadas de posición que son P(x,y,z) y el buffer auxiliar P”(x,y,z).
Estas necesidades se corresponden con 1 x n (masas) + 3 x n (Velocidades) + 3 x n (Posición) + 3 x n (Buffer auxiliar de posición) = 10n doubles.

## 5.2. Paralelización de nbody-F

En la alternativa B se utilizan tres vectores $ff_x$, $ff_y$, y $ff_z$ de tamaño n para almacenar los sumatorios parciales de las fuerzas que recibe cada cuerpo. Al intentar paralelizar esta alternativa. podría ocurrir que dos o más threads escribieran sobre la misma posición de estos vectores, al realizar las restas en cada iteración del bucle, o al realizar la suma de fx, fy, fz respectivamente al salir del bucle, por lo que es necesario gestionar estas dos posibles concurrencias. Utilizando OpenMP se pueden establecer regiones críticas o atómicas. En nuestros experimentos su implementación ha supuesto incrementos en el tiempo de ejecución de un 400% respecto a la misma versión de la aplicación sin regiones críticas o atómicas. Por esta razón es importante evitar este tipo de regiones.

En la siguiente figura 50 se muestran el cómputo que realizarían dos threads simultaneamente y la posición del vector ffx que sería accedida por cada uno de los threads para cada uno de los cálculos. Como se puede observar existe la posibilidad de que ambos threads accedan a las mismas posiciones de memoria a la vez, por lo que es necesaria una región crítica o atómica con el consecuente overhead.

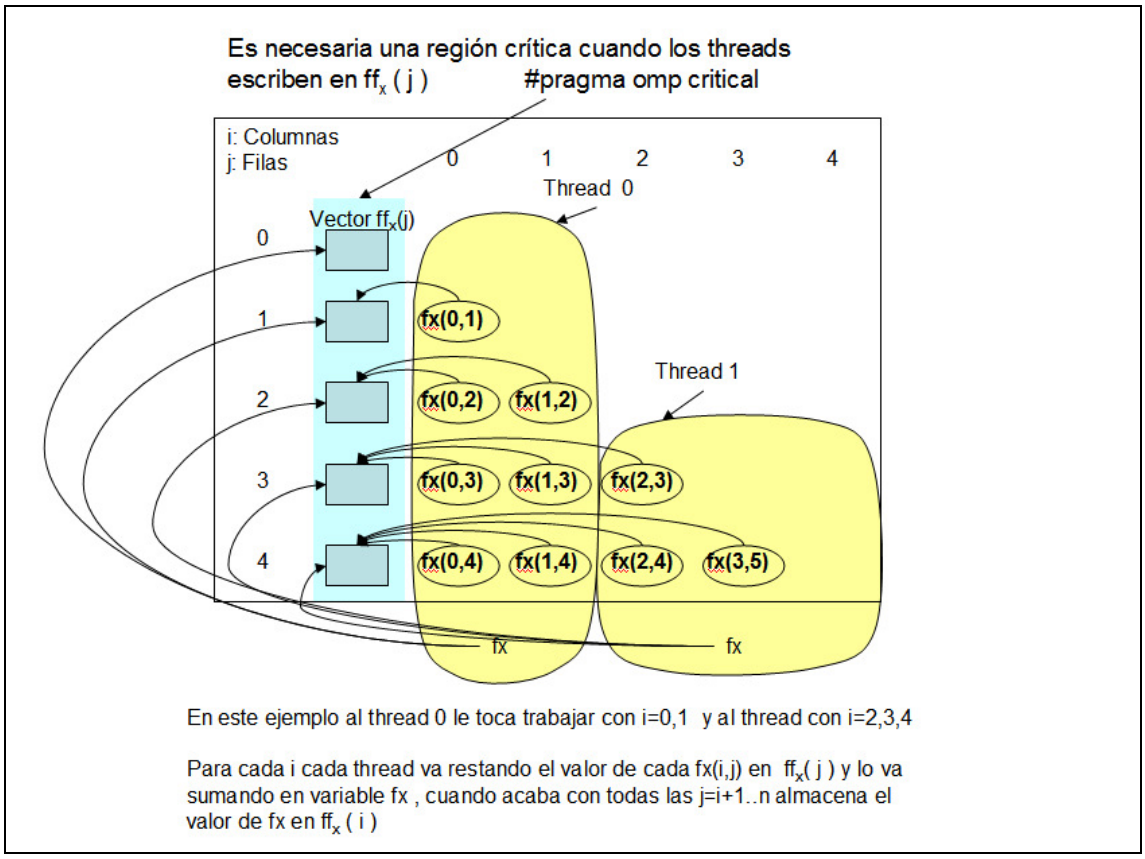



**Figura 50:** Accesos al vector $ff_x$ de tamaño n durante el cálculo de las fuerzas que inciden sobre un cuerpo para la coordenada X, utilizando un vector ffx.

En el ejemplo de la figura 50 podríamos intentar realizar las operaciones en lugar de con

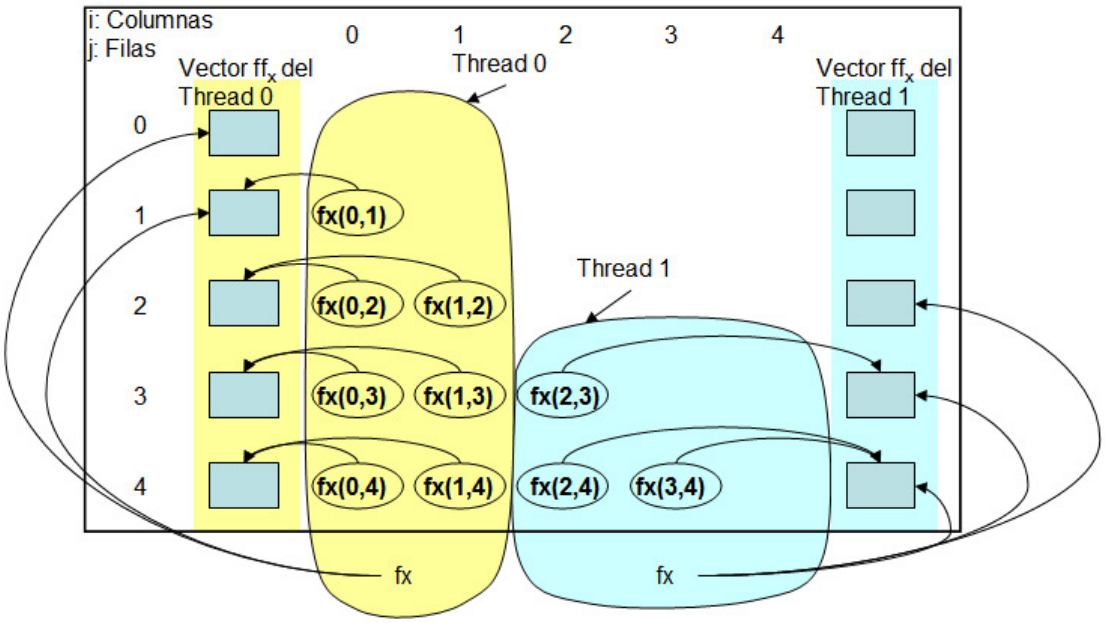
```
i=0..n
  j=i+1 ..n
```

con

```
j=1..n
  i=0..j
```

de esta forma eliminamos la región crítica en la escritura de $ff_x( j )$ pero la provocamos sobre la escritura de $ff_x( i )$

Para evitar estas regiones críticas utilizaremos tres vectores ffx, ffy, ffz de n posiciones cada uno, para cada thread, en la figura 51 se muestra esta idea con un ejemplo en el que n=5 y trabajan dos threads.




En este ejemplo al thread 0 le toca trabajar con i=0,1 y al thread 1 con i=2,3,4

Para cada i cada thread va restando el valor de cada fx(i,j) en $ff_x$( th )( j ) y lo va sumando en variable fx , cuando acaba con todas las j=i+1..n almacena el valor de fx en $ff_x$( th )( i ), donde th es el identificador del thread.

**Figura 51:** Accesos a los vectores $ff_x$ de tamaño n durante el cálculo de las fuerzas que inciden sobre un cuerpo para la coordenada X, utilizando un vector ffx por thread.

Como se puede observar, el volumen de cómputo a realizar por cada thread no es equitativo. Existe desbalanceo de carga. Concretamente en el ejemplo de la figura 51 en el que el trabajo se reparte entre dos threads, al thread 0 le corresponde más del doble de volumen de computo a realizar que al thread 1.

A continuación se muestra el código que incorpora la idea de utilizar tres vectores X, Y, Z de n posiciones cada uno, para cada thread.

```
TODOS A LA VEZ EN PARALELO
{
   Para tc = 1..k
   {
    REPARTIR EN PARALELO  Para i = 1..n
    {
      // inicializa variables Pxi, Pyi, …

      Para j = i+1..n
      {
           // Calcular Fsdx, Fsdy y Fsdz

           ffx[THREAD_ID][j] = ffx[my][j] - Fsdx
           ffy[my][j] = ffy[my][j] - Fsdy
           ffz[my][j] = ffz[my][j] - Fsdz
           fx = fx + Fsdx
           fy = fy + Fsdy
           fz = fz + Fsdz
         }
      ffx[THREAD_ID][i] = ffx[my][i] + fx
      ffy[my][i] = ffy[my][i] + fy
      ffz[my][i] = ffz[my][i] + fz
    }

    REPARTIR EN PARALELO   Para i = 0..n
        {
          fx = fy = fz=0;
          for(j=0; j<th; j++) {
               fx+=ffx[j][i];  fy+=ffy[j][i];   fz+=ffz[j][i];
          }

          // usar fx, fy y fz para actualizar vx, vy, vz, px, py y pz

          for (j=0;j<th;j++) {
               ffx[j][i]=0;      ffy[j][i]=0;        ffz[j][i]=0;
          }
        }
    }
  }
```

A medida que aumenta el valor de i disminuye el volumen de cómputo a realizar

## Modelo del algoritmo nbody-F paralelo

A continuación se analizarán los tres aspectos (sincronización, comunicación y balanceo del volumen de cómputo) descritos en apartado 5.2.

### Sincronización

En el algoritmo nbody-F paralelo es necesario que todos los threads finalicen el cálculo de los sumatorios parciales de las fuerzas reciprocas entre partículas que tienen asignados, antes de proseguir. Estos sumatorios parciales son necesarios para cálcular las coordenadas fx, fy, fz de fuerzas reciprocas, a partir de las cuales se realizará la actualización de las velocidades y las posiciones.
Esta necesidad de que todos los threads deban finalizar un trabajo antes de proseguir implica una sincronización extra en el algoritmo, por lo que el número de sincronizaciones entre threads es de (2 x k), siendo k el número de iteraciones.

### Comunicación

En el algoritmo nbody-F el scheduling es dinámico (en el algoritmo nbody-A el scheduling es estático). Por este motivo daremos una aproximación del volumen de datos transferido entre los threads y del patrón de comunicación.

El volumen de datos escrito por cada thread y que posteriormente es leído por otros threads se corresponde con (k x n/th x 3 doubles) posiciones + (k x n/th x 1/2 x 3 doubles) + (k x n/th x 3 doubles) fuerzas recíprocas.

En este algoritmo en cada una de las k iteraciones es necesario, primero acumular sumatorios parciales de fuerzas recíprocas, segundo sumar los sumatorios parciales calculados por cada uno de los threads en el paso anterior, y finalmente actualizar las posiciones, que serán leídas en la siguiente iteración.

El volumen de datos leídos por cada thread, que previamente han sido escritos por otros threads se corresponde con (k x n(th-1)/th x 3 doubles) posiciones + (k x n/th x 3 doubles) fuerzas reciprocas.

Para poder obtener las fuerzas recíprocas, cada thread necesita en primer lugar leer las posiciones cálculadas previamente en la iteración anterior, y en segundo lugar es necesario acumular los sumatorios parciales de las fuerzas recíprocas, calculadas en la iteración en curso.

Los 3 doubles se corresponden a las coordenadas x, y, z, ya sean del vector de posición o del vector de sumatorios parciales de fuerzas recíprocas.

El patrón de comunicaciones entre threads igual que en el caso del nbody-A es de todos con todos.

### Balanceo del volumen de cómputo

La carga de trabajo de los procesos en el algoritmo nbody-F paralelo esta desbalanceada. En la figura 51 se puede observar que en el caso de utilizar dos threads, al thread 0 le corresponde más del doble de volumen de cómputo que al thread 1. Esto es debido a que el bucle con i=0…n que se reparte entre los threads contiene un bucle interno, y en este bucle interno el volumen de cómputo a realizar se va reduciendo a medida que aumenta la i. Este comportamiento se ha destacado y comentado en el código del algoritmo mostrado anteriormente.

En nuestro caso el desbalanceo de carga se puede solucionar utilizando el parámetro schedule en la directiva #pragma omp for, por ejemplo:

#pragma omp for schedule(dynamic,10)

Otra manera de balancear la carga es manualmente repartiendo las i a procesar alternativamente entre los threads. Por ejemplo para dos threads, quedarían repartidas de la siguiente manera: al tread 0 le corresponderían i=0,2,4… y al thread 1 le corresponderían i=1,3,5 ….

Aunque por muy poco respecto al balanceo manual, los mejores resultados se obtienen simplemente dejando que el balanceo lo realice openmp mediante la directiva schedule (dynamic, [valor]). En nuestros experimentos utilizamos schedule (dynamic, [chunk]), variando chunk de 1 a 100, obteniendo los mejores resultados para chunk = 10.

**Requerimientos de memoria**

Los requerimientos de memoria crecen linealmente respecto a n y a th. En este algoritmo tampoco dependen del número de iteraciones k. Concretamente se necesita un vector de masas m, un vector de coordenadas de velocidad V(x,y,z) y un vectores de coordenadas de posición que son P(x,y,z) y un vector de coordenadas de fuerzas F(x,y,z) por thread.
Estas necesidades se corresponden con 1 x n (masas) + 3 x n (Velocidades) + 3 x n (Posición) + 3 x n x th (Sumatorios parciales de Fuerzas) = 7 x n + 3 x n + th doubles

Hay que destacar que aunque los requerimientos memoria globales aumentan con el número de threads, los requerimientos de memoria por thread únicamente dependen de n.

## 5.3. Análisis del rendimiento de la aplicación en un multiprocesador

Se han utilizado las dos variantes paralelizadas del problema n-body mostradas y descritas anteriormente para evaluar las diferencias de rendimiento entre los nodos multi-core de memoria compartida de diferente arquitectura.

Igual que en el caso del análisis de los procesadores, se podrían ejecutar baterías de instrumentos de medida para caracterizar y cuantificar el rendimiento de los nodos de cómputo, compuestos de múltiples procesadores. En este estudio no se han realizado. En cambio se han creado versiones modificadas del algoritmo en las que se evitan determinados eventos a estudiar. El objetivo de disponer del algoritmo modificado es poder realizar experimentos y comparar los resultados obtenidos del algoritmo modificado y del algoritmo sin modificar. De esta forma, por comparación, se puede medir el impacto del evento estudiado.

### Metodología experimental

A continuación se muestra información relativa a los nodos de cómputo en los que se realizan los experimentos. Las tablas que se muestran contienen la información que afecta a la interacción entre los threads ejecutándose en diferentes núcleos del nodo. En el apartado 4.2, del capítulo anterior, se ofrecía la información general de los cores (núcleos) que componen el nodo y del sistema operativo, versión del kernel, versión del compilador, y versión de la API de rendimiento utilizada.

**NODO IntelDualCore (2 x IntelCore)**

| | | |
|---|---|---|
| Procesadores | 1 | Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz |
| Cores por procesador | 2 | |
| Threads por core | 1 | |
| | | |
| Cache L1 | | Privada |
| Cache L2 | | Compartida |

**NODO AMDAthlonDualCore (2 x AMDAthlonCore)**

| | | |
|---|---|---|
| Procesadores | 1 | AMD Athlon(tm) 64 X2 Dual Core 3800+ 1999.773MHz |
| Cores por procesador | 2 | |
| Threads por core | 1 | |
| | | |
| Cache L1 | | Privada |
| Cache L2 | | Privada |

**NODO 2xAMDOpteronQuadCore 2 x (4 x AMDOpteronCore)**

| | | |
|---|---|---|
| Procesadores | 2 | Quad-Core AMD Opteron(tm) 2352, 2111.021Mhz |
| Cores por procesador | 4 | |
| Threads por core | 1 | |
| | | |
| Cache L1 | | Privada |
| Cache L2 | | Privada |

**NODO SPARCT2QuadCore (4 x SPARCT2core)**

| | | |
|---|---|---|
| Procesadores | 1 | SPARC-E-T5120, UltraSPARC T2, 1165 MHz |
| Cores por procesador | 4 | |
| Threads por core | 8 | |
| | | |
| Cache L1 | | Privada |
| Cache L2 | | Compartida |

Las mediciones las realizamos mediante librerías que nos permiten el acceso a contadores hardware de los procesadores, en concreto utilizamos “PAPI (Performance Application Programming Interface.) versiones : 3.4.9 y 3.6.2( http://icl.cs.utk.edu/ “ )

Las soluciones estudiadas de n-body tienen una estructura del tipo

Inicialización de los vectores
Inicio de la región paralela (pragma omp parallel)
Inicio fork-join (pragma omp for)
“Computo”
Fin fork-join
Inicio fork-join (pragma omp for)
“Computo”
Fin fork-join
fin de la región paralela

Las mediciones de la ocurrencia de los eventos para cada uno de los threads las realizamos incluyendo llamadas a la librería PAPI de la siguiente manera.

Inicialización de los vectores
Inicio de la región paralela (pragma omp parallel)
**Inicio del contador de eventos** (instrucciones, fallos de L1, L2, L3, u otros)
Inicio fork-join (pragma omp for)
"Computo"
Fin fork-join
Inicio fork-join (pragma omp for)
"Computo"
Fin fork-join
**Parada del contador de eventos**
fin de la región paralela

## Resultados del estudio de las implementaciones paralelas

Los resultados del estudio se dividen en dos apartados.
El primero se tratará a continuación y se centra en los resultados de tiempo en nanosegundos normalizados por $k \times n^2$. Estos tiempos se han obtenido para los algoritmos nbody-A paralelo y nbody-F paralelo en cada uno de los nodos a estudio. En cada uno de estos nodos se han ejecutado ambos algoritmos con diferente número de threads, con el objetivo estudiar en cada nodo la escalabilidad del algoritmo y detectar el overhead asociado a la paralelización.
El segundo apartado se tratará en el capítulo 5.4, y se dedica al análisis del overhead asociado a la paralelización. Para ello se realizan una serie de experimentos específicos, de los que se mostrarán y explicarán los resultados

### Resultados para el algoritmo nbody-A paralelo

En las figuras 52, 53, 54, y 55 se muestran tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-A paralelo, en los sistemas AMDAthlonDualCore, 2xAMDOpteronQuadCore, IntelDualCore y SPARCT2

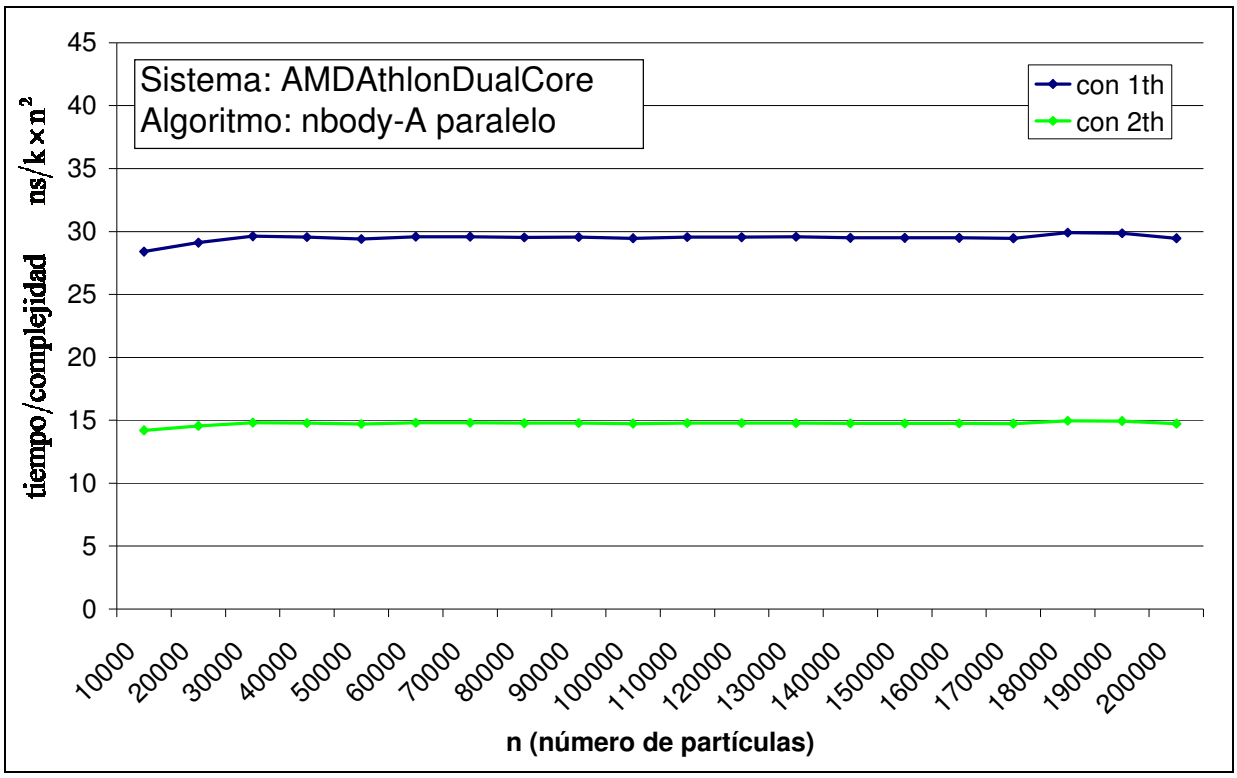



**Figura 52:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-A paralelo en el procesador AMDAthlonDualCore, con **k**=1, y **n** entre 10.000 y 200.000.

En el nodo AMDAthlonDualCore disponemos de un procesador con dos núcleos (cores). En cada uno de estos cores es capaz de ejecutar un thread. Cuando ejecutamos el algoritmo nbody-A paralelo con dos threads el tiempo de ejecución se reduce a la mitad. Obtenemos un speedup es de 2, por lo que la eficiencia por thread es del 100%

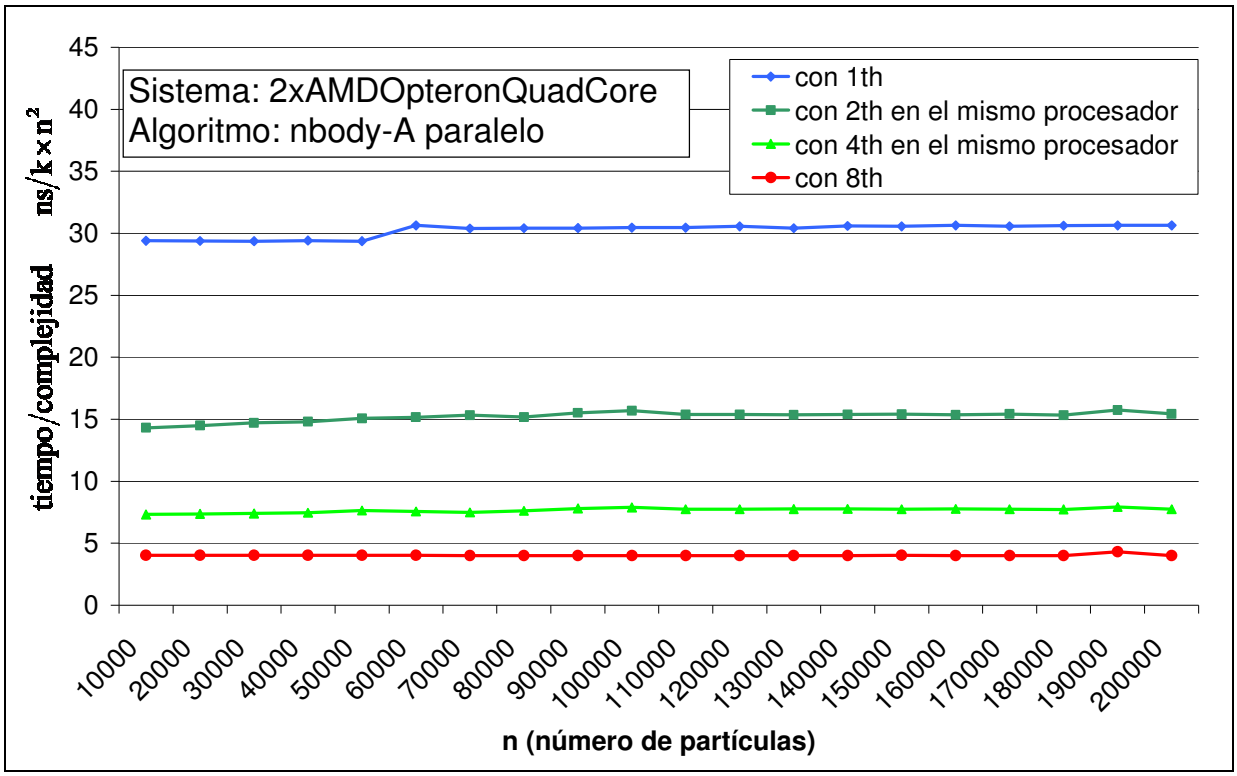



**Figura 53:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-A paralelo en el procesador AMDOpteronQuadCore, con k=1, y n entre 10.000 y 200.000.

En el nodo 2xAMDOpteronQuadCore, disponemos de dos procesadores con cuatro núcleos (cores) cada uno. Cada uno de estos cores es capaz de ejecutar un thread. Al ejecutar el algoritmo nbody-A parelelo con dos threads el tiempo de ejecución se reduce a la mitad. Obtenemos un speedup de 2 y una eficiencia por thread es de 100%. Al ejecutar con cuatro threads el tiempo se reduce a la cuarta parte. Obtenemos un speedup de 4 y la eficiencia por thread se sigue siendo del 100%. Pero al ejecutar con ocho threads, el tiempo ya no se reduce a una octava parte. Se empieza a notar el overhead por paralelización. Obtenemos un speedup de 7.64, y por lo tanto la eficiencia por thread es de un 95.57%. El paso de 4 a 8 threads supone utilizar los dos procesadores QuadCore del sistema, por lo que algunas comunicaciones entre threads deben realizarse a través de la placa base, lo que provoca la (pequeña) reducción en la eficiencia.

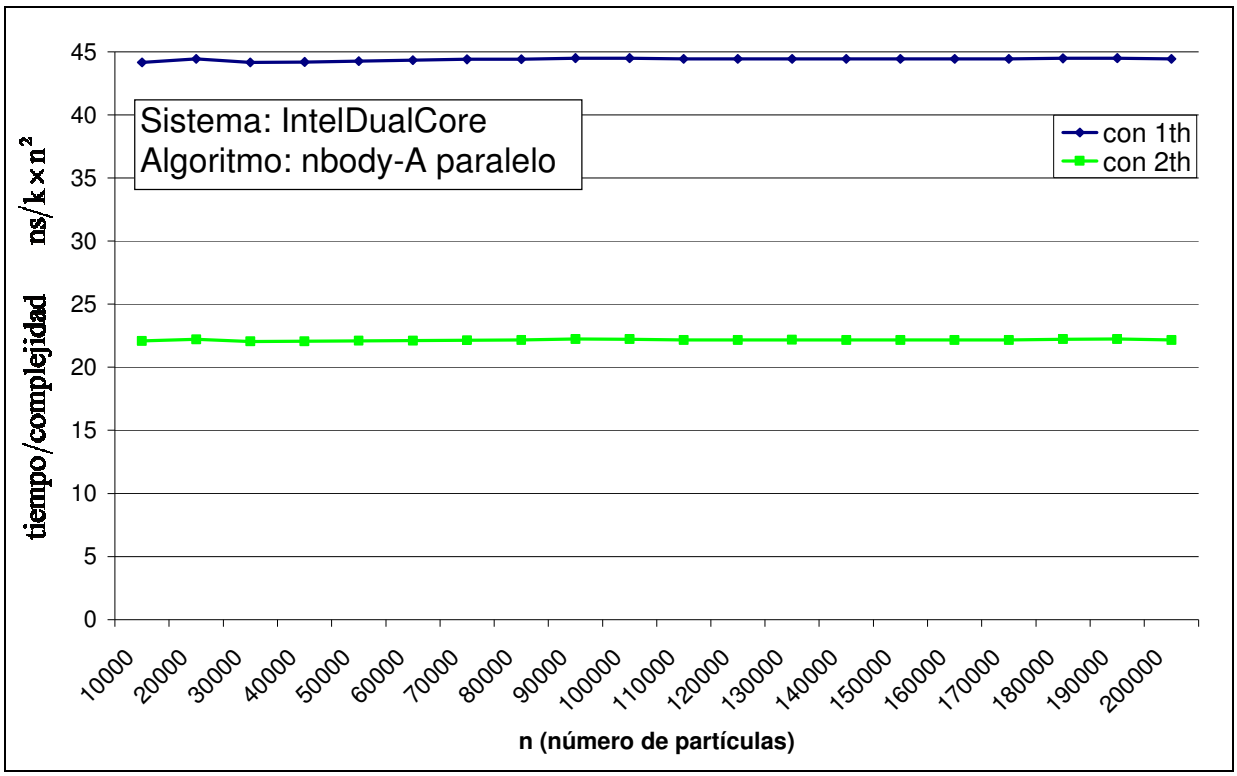



**Figura 54:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-A paralelo en el procesador IntelDualCore, con **k**=1, y **n** entre 10.000 y 200.000.

En el nodo IntelDualCore, disponemos de un procesador con dos núcleos (cores). Cada uno de estos cores es capaz de ejecutar un thread. Al ejecutar el algoritmo nbody-A paralelo con dos threads el tiempo de ejecución se reduce a la mitad. Obtenemos un speedup de 2, por lo que la eficiencia por thread es del 100%

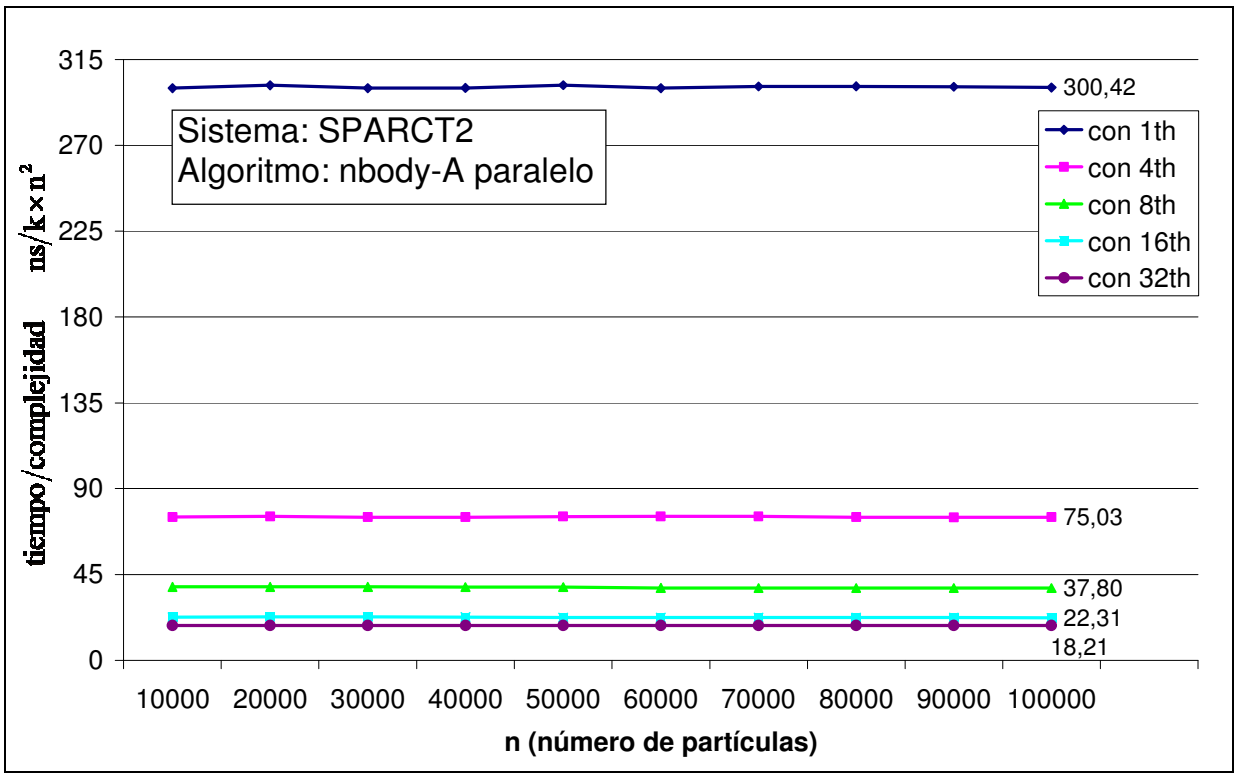



**Figura 55:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-A paralelo en el procesador SPARCT2, con k=1, y n entre 10.000 y 100.000.

En el nodo SPARCT2 disponemos de un procesador con cuatro núcleos (cores). A diferencia de los nodos anteriores, cada uno de estos cores puede ejecutar hasta 8 threads en paralelo. Al aumentar el número de threads con el que ejecutamos la aplicación nbody-A paralelo, se va perdiendo eficiencia por thread. En concreto, con hasta cuatro threads, se ejecuta un thread por core, por lo que la eficiencia por thread es prácticamente del 100%. Al ir añadiendo threads por core, se va perdiendo eficiencia. Con 8 threads la eficiencia es del 97%, con 16 threads se reduce hasta un 83% y cuando ejecutamos con 32 threads la eficiencia queda en un 51%

A continuación comparemos el rendimiento obtenido al ejecutar el algoritmo nbody-A paralelo en los distintos nodos que estamos analizando. En la figura 56, se muestra una gráfica resumen de los tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-A paralelo, en los cuatro sistemas analizados (utilizando el máximo número de threads posible por nodo).

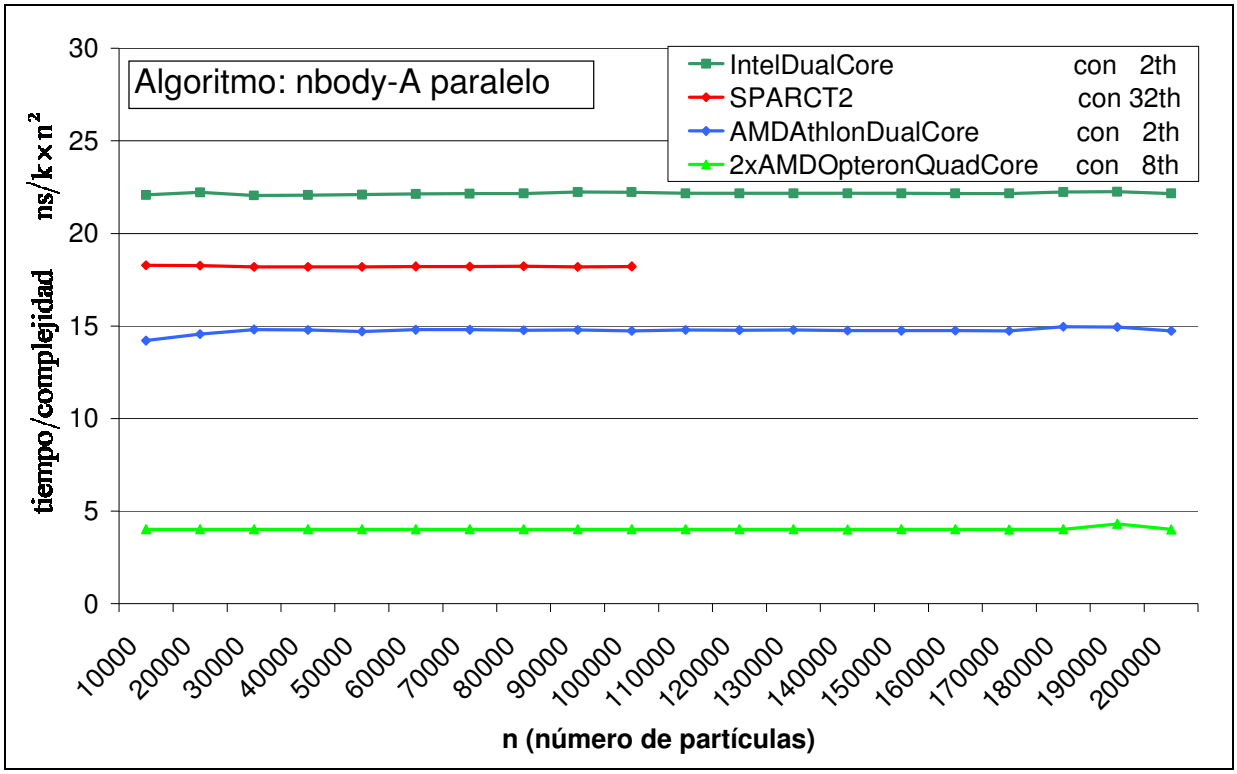



**Figura 56:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-A paralelo en los nodos IntelDualCore, SPARCT2, AMDAthlonDualCore y 2xAMDOpteronQuadCore, con **k**=1, **n** entre 10.000 y 200.000, y el máximo número de threads posible por nodo.

Con el nodo 2xAMDOptheronQuadCore es con el que se obtiene el mejor rendimiento. Utilizando 8 threads obtenemos un speedup de 3.5 respecto al AMDAthlonDualCore (2 threads), un speedup de 4.5 respecto al SPARCT2 (32 threads) y un speedup de 5.4 respecto al IntelDualCore (2 threads). Es importante tener en cuenta que a diferencia de los otros tres nodos el SPARCT2 ejecuta las operaciones en orden.

**Resultados para el algoritmo nbody-F paralelo**

En las figuras 57, 58, 59, y 60 se muestran tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-F paralelo, en los sistemas AMDAthlonDualCore, 2xAMDOpteronQuadCore, IntelDualCore y SPARCT2

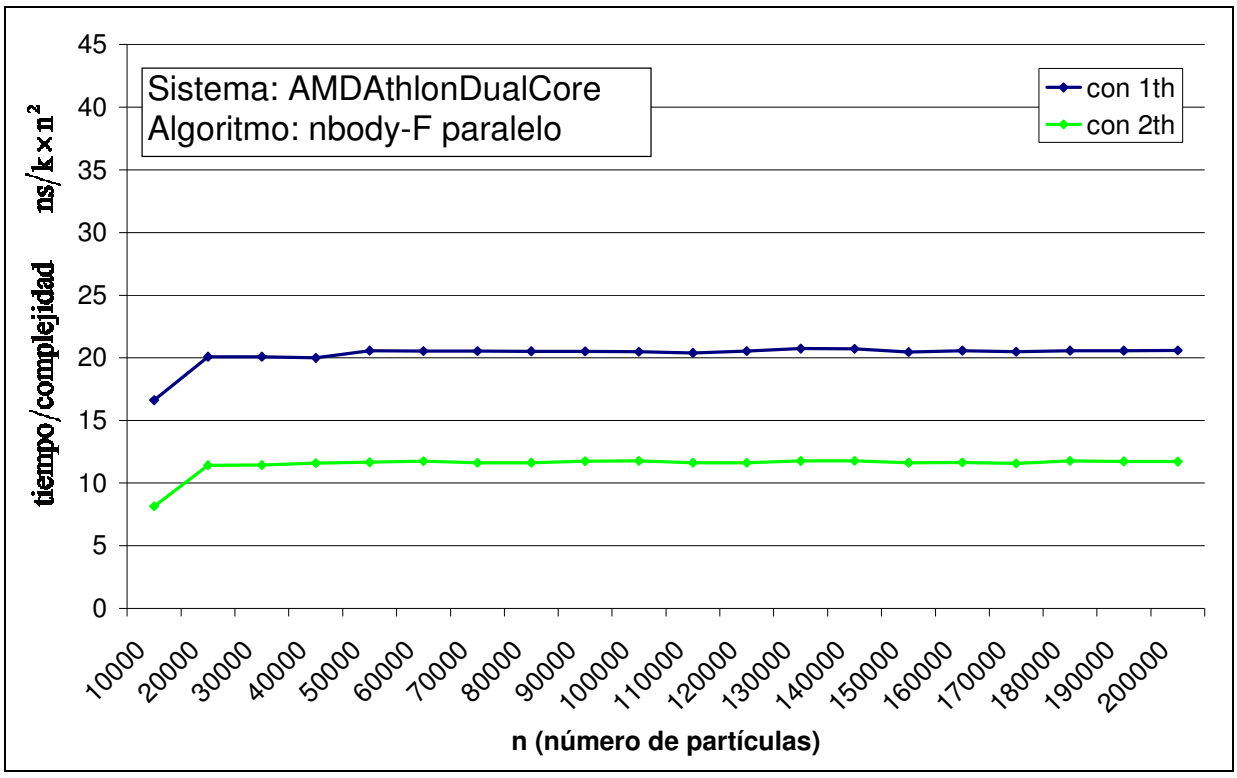



**Figura 57:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-F paralelo en el procesador AMDAthlonDualCore, con **k**=1, y **n** entre 10.000 y 200.000.

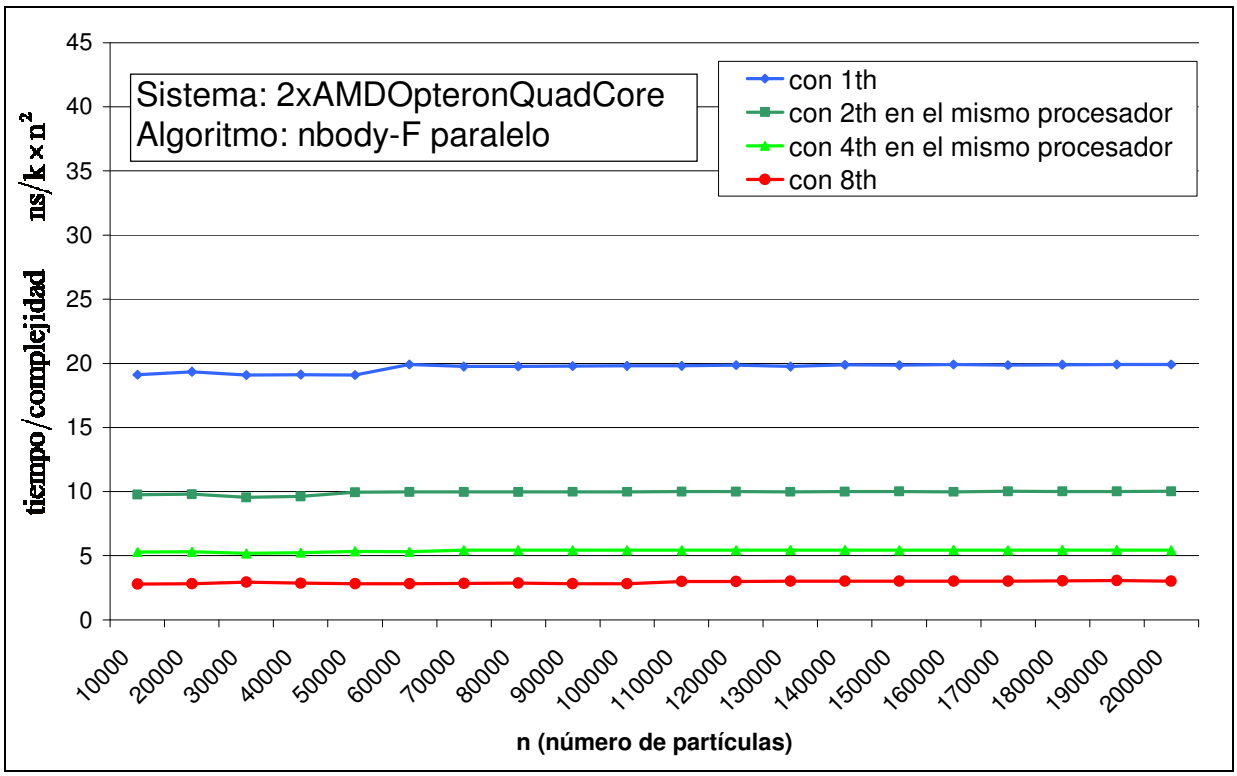



**Figura 58:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-F paralelo en el procesador AMDOpteronQuadCore, con **k**=1, y **n** entre 10.000 y 200.000.

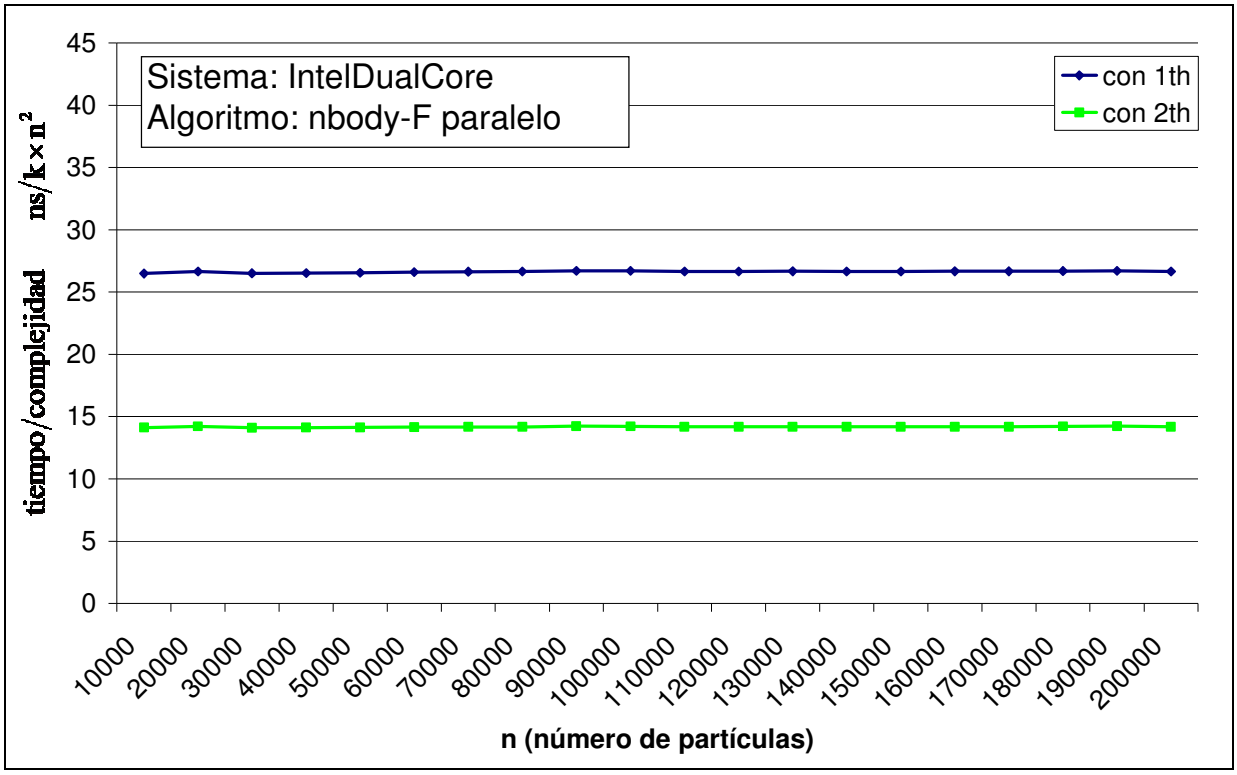



**Figura 59:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-F paralelo en el procesador IntelDualCore, con **k**=1, y **n** entre 10.000 y 200.000.

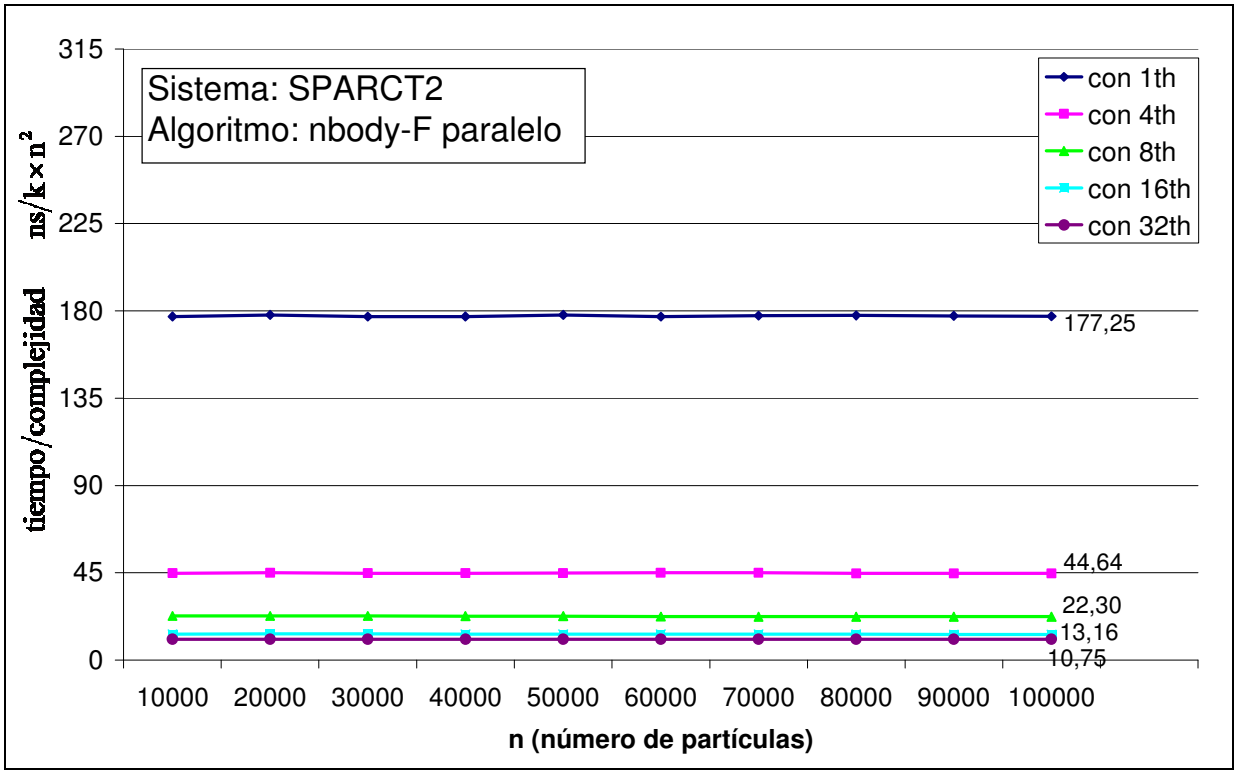



**Figura 60:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-F paralelo en el procesador SPARCT2, con **k**=1, y **n** entre 10.000 y 100.000.

En el nodo AMDAthlonDualCore cuando ejecutamos el algoritmo nbody-F paralelo con dos threads no llega a reducirse a la mitad. Obtenemos un speedup es de 1.7, por lo que la eficiencia por thread es del 88%. Si lo comparamos con los resultados obtenidos para nbody-A paralelo observamos que se esta consiguiendo un speedup de 1.27 (cuando utilizamos dos threads en ambas aplicaciones)

En el nodo 2xAMDOpteronQuadCore, al ejecutar el algoritmo nbody-F paralelo con dos threads el tiempo de ejecución tampoco se reduce a la mitad. Obtenemos un speedup de 1.95 y una eficiencia por thread de 97%, al ejecutar con cuatro threads el speedup es de 3.61 y la eficiencia por thread es del 90% y finalmente al ejecutar con ocho threads el speedup es del 6.84, y la eficiencia por thread es de un 85%. Si lo comparamos con los resultados obtenidos para nbody-A paralelo observamos que se está consiguiendo un speedup de 1,33 (cuando utilizamos 8 threads en ambas aplicaciones)

En el nodo IntelDualCore al ejecutar el algoritmo nbody-F paralelo con dos threads igual que en los nodos anteriores el tiempo no llega a reducirse a la mitad. Obtenemos un speedup de 1.87, por lo que la eficiencia por thread es del 93.7%. Si lo comparamos con los resultados obtenidos para nbody-A paralelo observamos que se esta consiguiendo un speedup de 1.56 (cuando utilizamos 2 threads en ambas aplicaciones)

En el nodo SPARCT2 al aumentar el número de threads con el que ejecutamos la aplicación nbody-A paralelo, se va perdiendo eficiencia por thread. Hasta cuatro threads la eficiencia por threads es prácticamente del 100%, a partir de aquí se va perdiendo eficiencia. Con 8 threads la eficiencia es del 97%, con 16 threads se reduce hasta un 82% y cuando ejecutamos con 32 threads la eficiencia queda en un 51%. Si lo comparamos con los resultados obtenidos para nbody-A paralelo observamos que se esta consiguiendo un speedup de 1.7 (cuando utilizamos 32 threads en ambas aplicaciones).

A continuación comparemos el rendimiento obtenido al ejecutar el algoritmo nbody-F paralelo en los distintos nodos que estamos analizando. En la gráfica siguiente se muestra un resumen de los resultados obtenidos, ejecutando con el máximo número de threads por nodo.

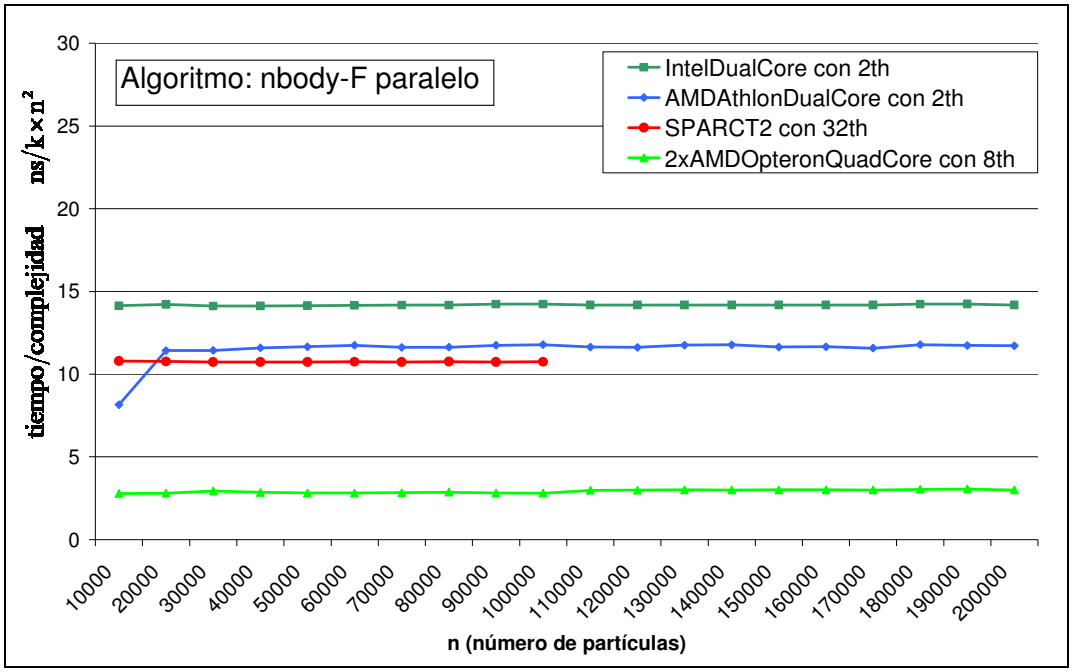



**Figura 61:** Tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo nbody-F paralelo en los nodos IntelDualCore, SPARCT2, AMDAthlonDualCore y 2xAMDOpteronQuadCore, con **k**=1, **n** entre 10.000 y 200.000, y el máximo número de threads posible por nodo.

Con el nodo 2xAMDOptheronQuadCore es con el que se obtiene el mejor rendimiento. Utilizando 8 threads obtenemos un speedup de 3.9 respecto al AMDAthlonDualCore (2 threads), un speedup de 3.8 respecto al SPARCT2 (32 threads) y un speedup de 4.7 respecto al IntelDualCore (2 threads).

Para cuantificar con más detalle el rendimiento y poder explicar las diferencias entre los distintos nodos, son necesarios experimentos específicos para evaluar el efecto de los distintos overhead. En el capítulo siguiente (5.4) se dedicará al análisis del overthead.

## 5.4. Análisis específico del overhead en el rendimiento

Para este apartado se han creado instrumentos basados en las soluciones n-body estudiadas. A partir de los algoritmos n-body estudiados, se han creado versiones modificadas en las que se evitan determinados eventos a estudiar. El objetivo de disponer del algoritmo modificado es poder realizar experimentos y comparar los resultados obtenidos del algoritmo modificado y del algoritmo sin modificar. De esta forma, por comparación, se puede medir el impacto del evento estudiado.

### Medida del overhead de la paralelización y de la sincronización

Se puede apreciar en los capítulos 5.1 y 5.2 que los algoritmos n-body están paralelizados según el paradigma fork-join. Durante la ejecución serie del algoritmo existe un único thread (master thread). Al entrar en una región paralela se crean una serie de threads entre los que posteriormente se repartirá el trabajo a realizar. Una vez finalizado el trabajo por parte de todos los threads, estos son eliminados, quedando el master thread.
A continuación se explicará el overhead provocado por la creación/eliminación de los threads y el overhead provocado por la sincronización entre las memorias cache. También se hará una explicación introductoria al overhead provocado por la contención entre los threads al acceder a niveles compartidos de memoria caché o a la memoria principal (debidos a los fallos de cache).

En los siguientes experimentos el número de iteraciones se mantendrá fijo a 10000 para la obtención de los overheads por creación y sincronización de threads, y para la obtención de los overheads por sincronización de datos entre las caches L1, entre las caches L2 y entre las caches L3. El motivo es que para cada iteración únicamente se realizan dos fork-join, por lo que para tener una media del overhead utilizamos 10000 iteraciones.

### Overhead por fork-join

En este apartado nos interesa medir el overhead provocado por la creación/eliminación de threads (fork-join). El experimento que se realiza a continuación se basa en comparar los tiempos de ejecución de dos algoritmos. El primero es el algoritmo nbody-A paralelo. El segundo es el algoritmo nbody-A paralelo modificado, a este algoritmo se le han quitado las directivas de openmp. De esta forma esta versión no realiza fork-join.
Como precauciones, nos aseguramos de que todos los datos a procesar quepan en la cache L1, evitando así que los fallos de cache alteren los resultados.
Las ejecuciones se realizan en los nodos AMDAthlonDualCore, y AMDIntelDualCore con 10000 iteraciones, un único thread, para valores de n menores a 500.

No esperamos un gran overhead por creación y destrucción de threads, ya que nbody es $O(\mathbf{n^2})$ y en nuestra implementacion de n-body-A paralelo únicamente hay un fork-join por iteración.

De nuestros experimentos obtenemos dos tiempos para distintos valores de n.

- TNAP: Tiempo de ejecución de Nbody-A paralelo (con directivas de openmp)
- TNAPM: Tiempo de ejcución de Nbody-A paralelo modificado (sin directivas de openmp)

La figura 62 muestra el siguiente cálculo: (TNAP-TNAPM) x 100 / TNAP, para la ejecución de los algoritmos nbody-A paralelo y nbody-A paralelo (modificado), en los sistemas IntelDualCore y AMDAthlonDualCore

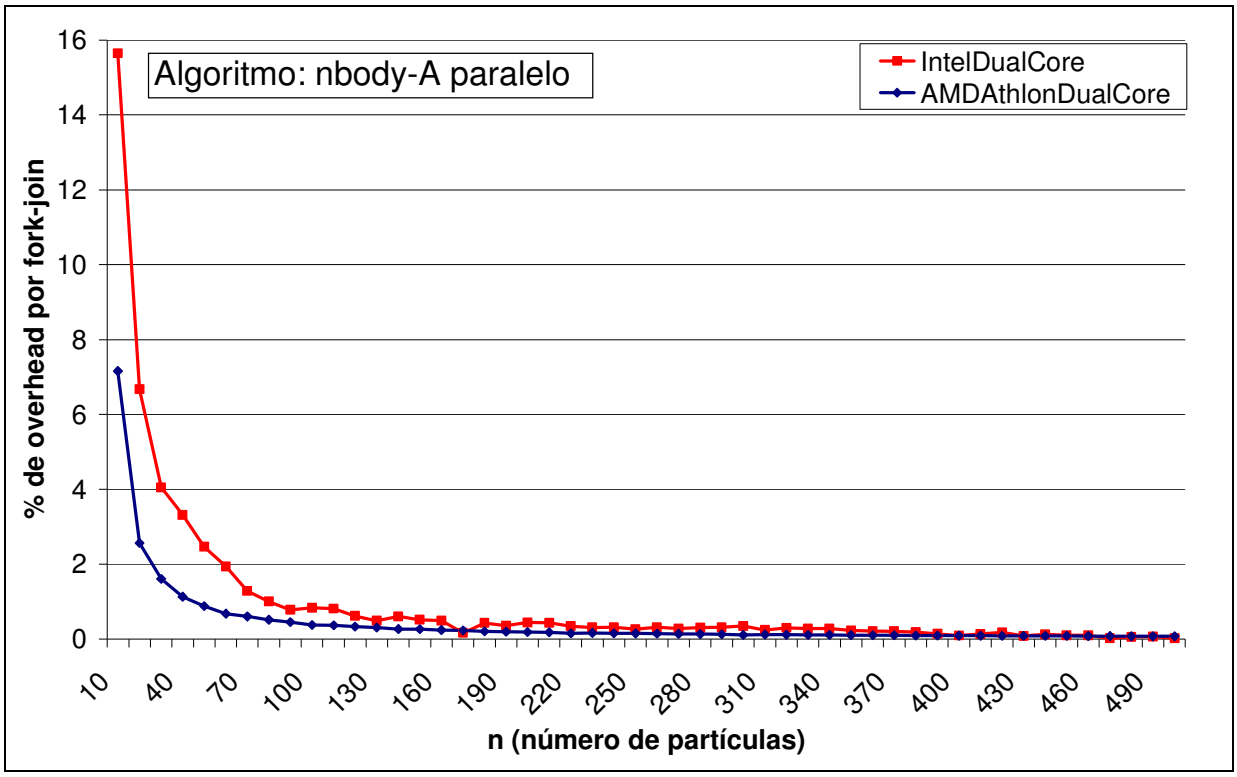



**Figura 62:** % de overhead provocado por fork-join para las ejecuciones del algoritmo nbody-A paralelo en los procesadores IntelDualCore y AMDAthlonDualCore, con **k**=10000, y **n** entre 10 y 500.

Tanto para el nodo IntelDualCore como para el AMDAthlonDualCore se observa el mismo comportamiento. Los resultados confirman lo que suponiamos, a medida que aumenta n, el overhead provocado por fork-join, se hace despreciable. Con el algoritmo nbody-F paralelo ocurre lo mismo.
Como nuestro estudio se centra en valores de n comprendidos entre n=10000 y n=200000, es obvio que este overhead lo podemos ignorar.

## Overhead por comunicación

Cuando un thread de una región paralela escribe en una posición de memoria de la memoria cache del nucleo (core) en el que se está ejecutando, invalida el valor de dicha posición de memoria de las caches que utilizan el resto de threads. Cuando cualquiera de estos threads necesite leer dicha posición de memoria, las caches de los procesadores deberán sincronizarse. Esta sincronización entre caches de diferentes procesadores supondrá un consumo en tiempo, provocando un overhead. A este overhead provocado por la necesidad de los threads de leer los datos escritos por otros threads, le llamaremos “overhead por comunicación”.

En este apartado nos interesa medir el overhead por comunicación. El experimento que se realiza a continuación se basa en comparar los tiempos de ejecución de dos programas en

los que dos threads escriben sobre un vector. En el primero de ellos cada thread escribe en las dos mitades del vector alternativamente. En el segundo cada thread trabaja sobre su propia zona de memoria, tanto en lectura como en escritura. De esta forma evitamos que los threads provoquen sincronizaciones entre las caches de los procesadores.
Lo que se pretende es que el algoritmo modificado realice el mismo número de operaciones y en el mismo orden que el primero, evitando la sincronización de datos entre memorias. El resultado de la ejecución será diferente ya que para evitar dicha sincronización de datos, los diferentes threads que intervienen trabajarán con datos propios.

Las ejecuciones se realizarán con 1 y 2 threads, y comparando la diferencia de speedup obtenido en los dos algoritmos podremos cuantificar el impacto de la sincronización de datos entre memorias.

En este experimento nos interesa ver el impacto del overhead por comunicación en L1, L2 y memoria principal.

La figura 63 muestra el % de overhead provocado por comunicación en el AMDAthlonDualCore, utilizando dos threads que escriben 10000 veces alternativamente en las dos mitades de un vector de tamaño. Las ejecuciones se han realizado para tamaños de vector desde 31,25Kbytes a 5000Kbytes.

De nuestros experimentos obtenemos dos tiempos para distintos valores de n.
- T: Tiempo de ejecución del programa (con sincronización entre caches).
- TM: Tiempo de ejcución del programa modificado (sin sincronización entre caches).

Calculamos
S=Tserie T (con 1th) / Tparalelo T (con 2th)
SM=Tserie TM (con 1th) / Tparalelo TM (con 2th)

La gráfica muestra el siguiente cálculo: (S-SM) x 100 / S

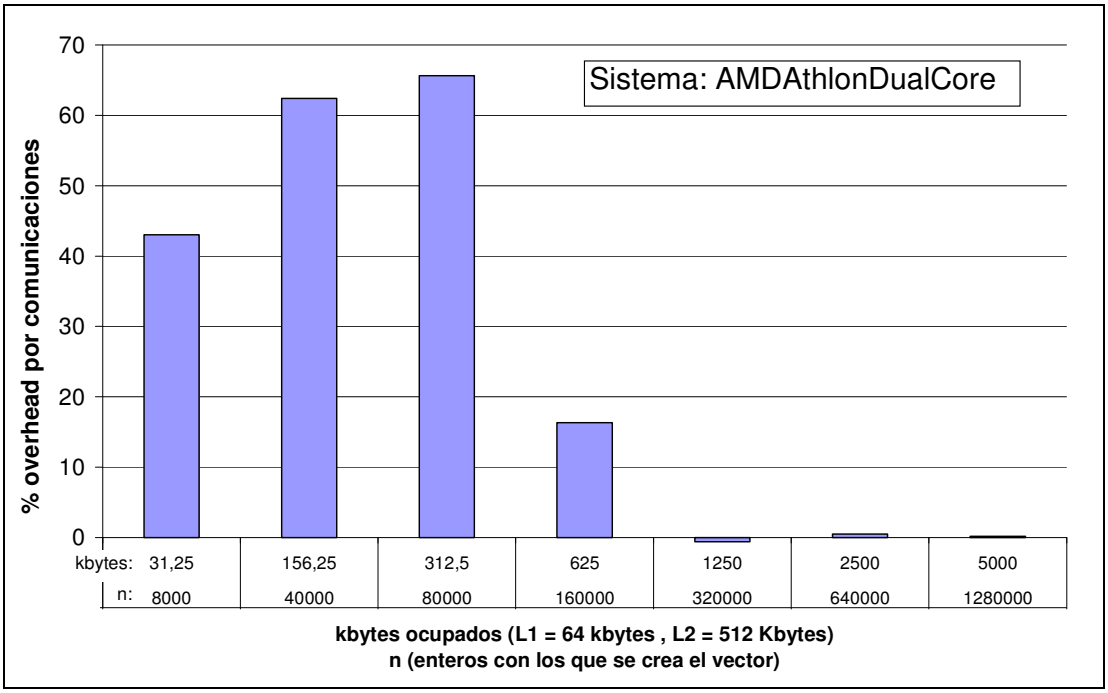



**Figura 63:** % de overhead provocado por comunicación en el AMDAthlonDualCore, utilizando dos threads que escriben 10000 veces, alternativamente en las dos mitades de un vector. Tamaño del vector de 31,25Kbytes a 5000Kbytes.
.

Al analizar la gráfica hay que tener en cuenta el tamaño de las caches del nodo AMDAthlonDualCore en el que estamos realizando los experimentos, L1=64kbytes, y L2=512Kbytes.

Podemos distinguir cómo el overhead provocado por las sincronizaciones entre caches L2 es superior al provocado por las sincronizaciones entre caches L1. La razón es que los accesos a cache L2 tienen un mayor tiempo de acceso que los accesos a cache L1. También se puede observar que para valores de n donde los requerimientos de memoria son superiores al tamaño de la cache L2, el overhead por comunicación disminuye hasta hacerse despreciable. Esto se debe a que no se produce ninguna comunicación adicional entre threads, sino que todos los accesos a datos se leen de memoria principal. Da igual si los datos leídos fueron modificados por "el mismo" thread o por el otro" thread: siempre se leen de memoria.

Es importante relacionar el overhead por comunicación, detectado en este apartado, con los resultados de rendimiento obtendidos a lo largo del capítulo. El overhead de comunicación medido en la figura 63 es causante de parte de la reducción de eficiencia de la paralelización, pero el valor cuantitativo obtenido no se traslada directamente a la reducción del rendimiento de la aplicación. Sería necesario un análisis más detallado para cuantificar la relación entre uno y otro.

## Overhead por contención en L2 y en memoria

Este overhead lo podemos describir como, el overhead provocado por la contención generada al producirse fallos de L1, y de L2, provocados por varios threads ejecutandose en distintos núcleos (cores) de un procesador.

Cuando los requerimientos de memoria de una aplicación sobrepasan el tamaño de las caches, se provocan “fallos de cache”, en los que se consume un tiempo extra. Este tiempo supone un overhead. Pero, además cuando los fallos de cache son provocados por varios threads trabajando en paralelo, se genera una contención. Esta contención es debida a la gestión de fallos de cache, por varios núcleos (cores) simultaneamente. En esta gestión simultanea de fallos de cache, los distintos núcleos (cores) tienen que, compartir simultaneamente el acceso al siguiente nivel de la jerarquia de memoria.

# 6 Metología de diseño y análisis de aplicaciones paralelas.

A partir de la experiencia obtenida, y a modo de conclusión de este trabajo, se propone una posible generalización de la metodología de análisis y sintonización de aplicaciones, para su ejecución en un computador paralelo.

**1 Estudio del problema secuencial**

1.a Análisis del rendimiento de la aplicación, independiente del hardware.

- Describir formalmente (notación matemática) el algoritmo y posibles variantes.
- Identificar los parámetros que determinan el tamaño del problema.
- Para cada variante del algoritmo identificar secciones independientes, de forma que la suma de los tiempos de cada sección sea el tiempo total de ejecución.
- Para cada sección de cada variante del algoritmo:
    - Calcular la complejidad: volumen de operaciones básicas en función del tamaño del problema.
    - Identificar la "operación abstracta": Se repite múltiples veces y consta de las operaciones primitivas que más veces se ejecutan en la sección del algoritmo, es decir, las que se ejecutan dentro del bucle más interno. En general se puede definir en relación al problema.
    - Identificar las operaciones primitivas de que consta cada operación abstracta:
        - **Punto-flotante:** Suma/resta/comparación (SUMApf), multiplicación (MULTpf), división (DIVpf), raíz cuadrada (RAIZpf).
        - **Entera:** Suma/resta/comparación (SUMAent), multiplicación (MULTent), división (DIVent).
        - **Salto:** Condicional (SLTcnd), incondicional (SLTinc).
        - **Acceso a memoria:** Lecturas y Escrituras a Memoria (LecMem, EscMem).
    - Analizar caminos de dependencias de datos internos a la operación abstracta.
    - Identificar los patrones de acceso de las instrucciones LecMem y EscMem
        - Tamaño de la zona de memoria accedida por cada instruccón
        - Orden de acceso a los datos dentro de las zonas
            - Accesos regulares: Secuencial (stride=1), No secuencial (stride>1), Otros
            - Irregulares
    - Identificar el patrón de comportamiento de los saltos condicionales
        - Comportamiento repetitivo regular de los resultados Salta (S) y No Salta (NS)
        - Comportamiento irregular, dependiente de los datos

1.b Análisis del rendimiento de la aplicación secuencial en un procesador base

- Modelar rendimiento del procesador
    - Frecuencia de reloj y topología de la jeraquía de memoria
    - Latencias y anchos de banda para las operaciones básicas, obtenidos mediante la ejecución de programas sintéticos de prueba
- Definir las ejecuciones del programa para diferentes tamaños de problema
    - Entrada, Número de repeticiones y valores estadísticos, mecanismo para filtrar "*outliers*"
    - Métricas: tiempo de ejecución, fallos de caché .. dividido por complejidad
- Ejecutar y medir tiempos para cada entrada y tamaño de problema
    - Generar gráficas en función del tamaño del problema
        - Si no hay ineficiencias, la gráfica tendrá un valor constante. Las ineficiencias se visualizan en las variaciones o anomalías de la gráfica.
        - Si es posible, solucionar los problemas de ineficiencia
    - Explicar los resultados
        - Usar los modelos de rendimiento para encontrar los factores más determinantes.
        - Realizar experimentos para identificar y evaluar el impacto de las operaciones críticas.

## 2 Estudio del problema paralelo

2.a Análisis de la aplicación, independiente del hardware

- Describir formalmente el algoritmo paralelo y sus posibles variantes
- Para cada variante y para cada sección significativa del algoritmo:
    - Identificar dependencias de datos entre threads y la necesidad de sincronización
        - Cuantificar el overhead provocado
            - Nº de Barriers (sincronización entre threads)
            - Nº de Zonas críticas (exclusión mútua)
    - Cuantificar el volumen de datos transferido entre los threads y el patrón de comunicación
        - Volumen en datos escritos por cada thread, para otros threads.
        - Volumen en datos leídos por cada thread, provenientes de otros threads.
        - Patrón de comunicación: vecinos, todos a uno, uno a todos, todos con todos, etc.
    - Analizar desbalanceo del volumen de cómputo entre los threads
        - Depende del tamaño del problema, del nº de threads y de los datos.
    - Analizar incremento de volumen de cómputo debido a operaciones redundantes.

2.b Análisis del rendimiento de la aplicación ejecutada en un multiprocesador

- Modelar rendimiento del nodo de cómputo multiprocesador.
    - Número de núcleos y de threads concurrentes por núcleo (*multi-threading H/W*).
    - Topología de la jerarquía de memoria compartida (memoria UMA/ NUMA, ... ).
    - Latencias y anchos de banda para las operaciones de acceso a memoria compartida, de comunicación, y de sincronización obtenidos mediante la ejecución de programas sintéticos de prueba.
- Definir las ejecuciones del programa para diferentes tamaños de problema
    - Métricas: speedup, eficiencia por thread, overhead por comunicación, overhead por fork/join...
- Ejecutar y medir tiempos para cada entrada, tamaño de problema y diferente nº de threads.
        - Generar gráficas en función del tamaño del problema y del nº de procesadores usados.
            - Analizar ineficiencias: anomalías de la gráfica, o limitaciones de speedup.
            - Si es posible, solucionar los problemas de ineficiencia.
        - Explicar los resultados.
            - Usar los modelos de rendimiento para encontrar los factores más determinantes.
            - Realizar experimentos para identificar y evaluar el impacto de las operaciones críticas realizadas entre threads por ejemplo, medir experimentalmente el incremento en el número de fallos de cache L1/L2 debido a la comunicación

# 7 Conclusiones

El objetivo principal del trabajo consistía en iniciarse en el proceso de investigación. En esta línea se debía en primer lugar, analizar un problema clásico de la literatura (N-body), codificar el algoritmo utilizando C, y paralelizarlo con OpenMP. En segundo lugar se debía realizar un proceso de análisis de prestaciones, planificando y ejecutando los experimentos en nodos de cómputo de características diferentes, e interpretando los resultados para extraer conclusiones que permitieran optimizar el rendimiento en cada nodo de cómputo.

Para comprender mejor las conclusiones que se explicarán en este capítulo, debemos primero comentar brevemente los siguientes resultados obtenidos.

Al final del proceso de estudio, implementación y paralelización del problema n-body, se obtienen dos implementaciones paralelas. En la primera implementación se prioriza minimizar los requisitos de memoria. Esta variante se caracteriza por realizar el cálculo de aceleraciones recíprocas en lugar de cálcular fuerzas recíprocas, por lo que nos referimos a ella como nbody-A paralela. En la segunda implementación se prioriza minimizar el volumen de cómputo. Esta variante realiza la mitad de volumen de cómputo que la anterior, sin incrementar los requisitos de memoria por thread. Como en esta variante se calculan las fuerzas recíprocas, nos referiremos a ella como nbody-F paralela.

Posteriormente, se realiza un proceso completo de análisis de prestaciones. En este proceso se modela, implementa e instrumenta la aplicación, así como diferentes variantes de ésta. Estas variantes de la aplicación se ejecutan en distintos procesadores, obteniendo resultados para distintas métricas, generando gráficas e interpretando los resultados.

### Conclusiones del trabajo

El estudio de este caso concreto ha proporcionado una experiencia que ha permitido esbozar una incipiente metodología de análisis de rendimiento, identificación de problemas y sintonización de un algoritmo a un nodo de cómputo concreto. En concreto, se ha trabajado en los siguientes aspectos:

- Realizar un modelo sencillo de rendimiento secuencial y paralelo de los algoritmos y de los nodos de cómputo.
- Estudiar alternativas para presentar los resultados de forma que sean visualmente significativos. Escoger una métrica apropiada y saber interpretar los resultados obtenidos es fundamental.
- Proponer reglas de interpretación de los resultados de rendimiento, de identificación de ineficiencias de rendimiento y de actuación para resolverlas.

A lo largo de la memoria se han presentado y justificado las diferentes estrategias y métricas para analizar el rendimiento, el overhead, y el impacto de eventos relacionados con el rendimiento, que ocurren en un procesador, durante la ejecución de una aplicación. Entre estas estrategias y métricas podemos destacar las siguientes.

Se ha descrito y justificado una métrica (tiempo de ejecución / complejidad del algoritmo), que es adecuada para mostrar gráficamente el rendimiento obtenido, y para interpretarlo visualmente con facilidad. En esta métrica el tiempo de ejecución lo hemos medido en nanosegundos y la complejidad del algoritmo es el volumen de cómputo realizado. Otra manera de definir la complejidad del algoritmo, es utilizando el concepto de “operación abstracta” descrito en el capítulo (4.1). Utilizando este concepto, la complejidad del algoritmo es el número de veces que se ejecuta la operación abstracta.
Tambíen se han aportado reglas para identificar ineficiencias en el rendimiento y explicarlas.

Se ha descrito una estrategia para el análisis del coste que tiene cada tipo de operación sobre el rendimiento obtenido en un procesador. Esta estrategia se basa en eliminar del camino

crítico, la operación a analizar, y posteriormente ejecutar la aplicación modificada y calcular el speedup obtenido. Utilizando esta estrategia se ha mostrado el impacto sobre el tiempo de ejecución, de las diferentes operaciones (del camino crítico del algoritmo nbody) en los nodos estudiados.

Se ha descrito como obtener el CPI de una aplicación ejecutada en un procesador en el que no se tiene acceso a los contadores hardware. En el capítulo 4.7, hemos explicado y justificado la obtención del CPI. En un equipo en el que no disponemos de acceso a los contadores hardware, podemos deducir la métrica (instrucciones/"operación abstracta") a partir de (instrucciones en ensablador/"operación abstracta"). Una vez conocemos los valores (Instruciones / k x $n^2$), (ns / k x $n^2$), y frecuencia de reloj del procesador (ciclos/ns), calculamos (Instruciones/ns) y seguidamente (ciclos/Instrucción) CPI.

En cuanto a los procesadores analizados, y teniendo en cuenta que el estudio se ha realizado utilizando los algoritmos nbody-A, nbody-F, y sus versiones paralelas, podemos concretar lo siguiente. Los nodos AMDAthlonDualCore, 2xAMDOpteronQuadCore, y IntelDualCore ofrecen rendimientos que se encuentran en el mismo rango, cuando ejecutamos con 1 y 2 threads las variantes del algoritmo nbody obtenidas en este trabajo. En cambio, en el nodo SPARCT2 se obtiene un rendimiento 10 veces menor que utilizando los nodos AMD y unas 6.5 veces menor que utilizando el nodo Intel (ejecutando las variantes nbody con un thread). En el SPARCT2, es necesario ejecutar las variantes nbody con 32 threads, para que el rendimiento se aproxime al obtenido en el resto de nodos (ejecutando en éstos, con 2 threads). Lo que nos lleva a afirmar que en el SPARCT2 el modelo de ejecución MultiThread, más la "ejecución de instrucciones en orden", requiere un esfuerzo especial por parte del programador. Este esfuerzo, es necesario para obtener (ejecutando una aplicación), un rendimiento cercano al obtenido en los procesadores DualCore actuales.

Para finalizar este capítulo comentaremos lo más destacable de las implementaciones del problema nbody, en su método de resolución directo (partícula-partícula).

Nbody es un problema de cómputo intensivo. El volumen de cómputo a procesar es $c \times k \times n^2$, donde n es el número de partículas del sistema, k es el número de pasos (iteraciones) a simular, y c es una constante. Por lo que la complejidad es $O(n^2)$.

El número de creaciones/eliminaciones de threads, así como el número de sincronizaciones y el de comunicaciones entre threads, es de orden k. Esto implica que el overhead provocado por la paralelización debido a creación/eliminación de threads, sincronización de threads, y comunicación de threads, crece en menor medida que el volumen de cómputo, al aumentar el número de partículas.

Nbody es cache-friendly. Los requerimientos de memoria tienen un crecimiento de orden n, mientras que el volumen de cómputo lo tiene de orden $\mathbf{n^2}$. Además el acceso a memoria es secuencial con stride=1.

**Problemas metodológicos**

En la ejecución de este trabajo, nos hemos encontrado con algunos problemas metodológicos, entre los que cabría destacar los siguientes.

Dificultad para seleccionar una métrica de rendimiento adecuada. En el apartado anterior de conclusiones se ha explicado la métrica que finalmente se ha utilizado, (tiempo de ejecución / complejidad del algoritmo).

Dificultad para diseñar experimentos. Inicialmente algunos de estos experimentos no eran adecuados para obtener la información que se pretendía o para confirmar/desmentir determinados comportamientos en el rendimiento o en los overheads estudiados. Una vez detectado este problema al analizar los resultados obtenidos, se rediseñaron los experimentos.

Estos dos problemas supusieron un proceso iterativo de análisis, implementación, ejecución, toma de resultados, procesado de resultados, presentación de los mismos, y detección de insuficiencias.

## Líneas futuras de investigación

A continuación se describen líneas futuras de investigación particulares y generales

Líneas futuras de investigación particulares

A lo largo de la memoria, se han presentado estrategias y métricas interesantes para analizar el rendimiento, el overhead, y el impacto de eventos relacionados con el rendimiento que ocurren en un procesador durante la ejecución de una aplicación. Pero queda mucho trabajo por hacer.

En esta línea, sería interesante analizar la métrica (fallos de cache / complejidad del algoritmo), y determinar la mejor forma de mostrar los resultados. Este análisis podría ir complementado con una serie de reglas para interpretar la métrica y explicar los resultados.
La importancia del análisis de esta métrica se comenta en el análisis de fallos de cache del capítulo 4.7.

En este trabajo se ha detectado un problema de rendimiento relacionado con el alineamiento en los accesos a memoria por parte de un conjunto de threads que trabajan en paralelo. Queda pendiente analizar otra solución alternativa a la que se ha utilizado en este trabajo, que consiste en intercalar los elementos de los vectores Px, Py, Pz.

Algunos equipos con procesador AMD, no arrancan correctamete de la infraestructura portable que se ha preparado para el acceso a los contadores hardware de los procesadores, (descrita en el capítulo 4.7). Este inconveniente ha quedado pendiente de corregir y sería interesante solucionarlo para futuros análisis de rendimiento.

Líneas futuras de investigación generales

Profundizar en los modelos cuantitativos de rendimiento secuencial y paralelo de los algoritmos y de los nodos de cómputo, y en la combinación de estos modelos para estimar el tiempo de ejecución.

Extender el uso de métricas relacionadas con la complejidad del algoritmo, por ejemplo métricas para los fallos de caché (comentada con más detalle en el apartado anterior), o el volumen de sincronización y comunicación.

Extender las reglas de interpretación de los resultados de rendimiento y de actuación para resolver problemas de rendimiento.

# 8 Bibliografía


Hennessy, J.L. y Patterson, D.A. (1996). *Computer Architecture: A Quantitative Approach*. San Francisco, California: Morgan Kaufmann

Brunett, S.; Thornley, J. y Ellenbecker, M. (1998). An initial evaluation of the Tera Multithreaded Architecture and programming system using the C3I parallel benchmark suite. *IEEE Computer Society,1 -19*

Corbalán, J. (2002). *Coordinated scheduling and dynamic performance analysis in multiprocessors systems*. Tesis doctoral. Barcelona: UPC.

Durán, A. (2008). *Self-tuned parallel runtimes: A case of study for OpenMP*. Tesis doctoral. Barcelona: UPC.

Gal-On, S. y Levy, M. (2008). Measuring Multicore Performance, *IEEE Computer Society*, Volume: 41 , Issue: 11 , 99 - 102

Hristea, C.; Lenoski, D. y Keen, J. (1997). Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks, *ACM New York, NY, USA*, 1 - 12.

Marathe, J.; Nagarajan, A. y Mueller, F. (2004). Detailed cache coherence characterization for OpenMP benchmarks, *ACM New York, NY,* 287 - 297

Marowka, A. (2008). Performance of OpenMP Benchmarks on Multicore Processors*, Springer-Verlag,* Volume: 5022, 208-219

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

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

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

Performance Application Programming Interface, http://icl.cs.utk.edu/