UNIVERSITAT POLITÈCNICA DE CATALUNYA

Departament d'Enginyeria Electrònica

SIMULACIÓN MONTE CARLO DE TRANSISTORES BIPOLARES DE HETEROUNIÓN ABRUPTA (HBT)

Autor: Pau Garcias Salvà
Director: Lluís Prat Viñas

4. Implementación del simulador MCHBT.

Debido a los elevados requerimientos de tiempo de cálculo de las simulaciones MC, es recomendable buscar soluciones que minimicen el problema. Mientras que una simulación típica por métodos convencionales de DD puede representar unos minutos, una simulación MC puede representar días e incluso semanas de espera en una estación de trabajo de potencia media.

La búsqueda de soluciones debe avanzar en todas direcciones: desde la búsqueda de algoritmos alternativos, pasando por la adaptación de los mismos a la arquitectura del computador en uso, hasta la aplicación de técnicas de supercomputación.

Las tendencias actuales de la supercomputación se basan en la concurrencia de procesos o computación en paralelo. Gracias al avance de esta rama de la Ciencia y a los últimos avances tecnológicos en el campo de la Informática, se puede hablar hoy de la computación masivamente paralela, en la que un número elevado de procesadores es capaz de realizar cálculos de forma coordinada. En el caso de los simuladores MC de dispositivos está claro que hay oportunidades de paralelismo, puesto que se simula un conjunto elevado de partículas con un tratamiento equivalente (y en general, independiente) para cada partícula.

Si bien la aplicación de técnicas de paralelismo ofrece un gran potencial para reducir de forma efectiva el tiempo de cálculo de las simulaciones MC, no hay que olvidar que un paso previo, puede que mucho más importante, es la adaptación de los algoritmos a la arquitectura del procesador, optimizando el uso de las unidades funcionales disponibles y los accesos a memoria.

Otro aspecto fundamental en el rendimiento global de la simulación son las mejoras directas sobre los algoritmos de la dinámica MC. En este sentido trataremos el método WMC (*weighted* MC) que asocia un peso variable a la carga de las partículas para reducir así el número total de partículas en la simulación.

Iniciaremos el capítulo presentando la problemática de generación eficiente de secuencias seudoaleatorias de calidad, dada su importancia en cualquier método MC. En el segundo apartado se dará un marco de referencia sobre la supercomputación y el paralelismo, que se particularizará en el apartado 3 a la implementación de nuestro simulador MCHBT. Los apartados 3 y 4 están dedicados, respectivamente, a las optimizaciones secuenciales (monoprocesador) y paralelas (multiprocesador) aplicadas en MCHBT. Finalmente, en el sexto apartado se presenta la implementación del método WMC.

4.1 La generación de números aleatorios.

La generación de números aleatorios es una parte esencial de cualquier simulación MC, puesto que la base del método está en la utilización de los mismos como mecanismo para escoger una muestra de la población del problema estudiado. La solución del problema se obtendrá de la aplicación de métodos estadísticos sobre esta muestra seleccionada.

Los verdaderos números aleatorios provienen de la observación de procesos físicos como la emisión de partículas radioactivas, el ruido térmico, la detección de partículas llegadas desde el espacio cósmico, etc. Existen series de números aleatorios (de longitud cercana al millón de números) recogidas por observación de estos procesos físicos y

grabadas en soporte magnético para ser utilizadas en simulaciones del tipo MC. La longitud de estas series es del orden de un millón de números que, con todo, puede resultar totalmente insuficiente para simulaciones largas, dada la capacidad cada día mayor de proceso de datos de los ordenadores.

La solución más práctica en simulaciones por ordenador es la generación de secuencias seudoaleatorias, que se basan en la aplicación de un determinado algoritmo matemático de generación sobre un conjunto reducido de los valores ya calculados de la secuencia. Los valores de inicialización necesarios para poder empezar a aplicar el algoritmo constituyen la *semilla* de la secuencia. El resultado es una secuencia de *apariencia aleatoria* para quien no conozca el algoritmo utilizado. De hecho, existen pruebas para verificar la calidad de la secuencia obtenida por cada algoritmo de generación [Hammersley,1964], [Knuth,1986]. La desventaja inicial de estas secuencias respecto a las verdaderamente aleatorias queda compensada si se consigue un algoritmo que obtenga buenas propiedades estadísticas, eficiencia en la generación en términos de tiempo de CPU y de espacio de memoria, período de repetición suficientemente largo y, finalmente, garantía de reproductibilidad para poder repetir el experimento en caso necesario. La longitud del período de repetición es un parámetro de suma importancia para conseguir una buena simulación, puesto que es importante no agotar nunca el ciclo del generador. De hecho, se considera que no debería utilizarse más del 5% del ciclo.

En las simulaciones MC es necesario disponer de números aleatorios con distintas funciones de distribución (dependiendo, por ejemplo, del mecanismo de dispersión considerado). Si bien existen algunos algoritmos para generar números aleatorios con diferente función de distribución, lo más habitual es generar secuencias con distribución uniforme en el intervalo real [0,1] y aplicar después una transformación matemática o algorítmica adecuada. Las transformaciones más comunes son la inversión de la función de distribución, el método del rechazo (o *hit-miss*) y los algoritmos dinámicos basados en cadenas de Markov (algoritmo de Metropolis, dinámica de Langevin, etc.) [Press,1992], [Kalos,1986], [Knuth,1981], [Rubinstein,1981], [Jacoboni,1988].

El método más extendido para la generación de secuencias seudoaleatorias con distribución uniforme es el generador congruencial lineal:

$$x_i = (a \cdot x_{i-1} + c)_{\mathrm{mod}ulo_m}$$

$$\eta_i = \frac{x_i}{m} \qquad \eta_i \in [0,1)$$

(4. 1)

donde a, c, m son parámetros enteros positivos. Para inicializar la secuencia es suficiente con proporcionar una sola semilla, x_0. El valor m es la longitud del ciclo del generador, limitada por la capacidad máxima de representación de datos enteros en el ordenador. En aritmética de 32 bits, $m \cong 2^r \cong 10^9$. Habitualmente se toma $c=0$, con lo que el generador pasa a llamarse generador congruencial lineal multiplicativo.

La calidad de un generador congruencial lineal vendrá determinada por la elección de los valores de sus parámetros, pero la calidad de una secuencia concreta puede cambiar mucho según la elección de la semilla. Así, por ejemplo, el generador RANDU de la serie IBM360 usaba $a=65539$, $c=0$ y $m=2^{29}$. La semilla la escogía el usuario dando un número impar cualquiera. Pero según el valor escogido los resultados presentaban muchas correlaciones con efectos verdaderamente catastróficos sobre las simulaciones. El generador SURAND también de IBM ($a=16807$, $c=0$ y $m=2^{31}-1$) era algo mejor, pero en general todos los generadores congruenciales lineales adolecen de un ciclo demasiado corto y de presentar demasiadas correlaciones (efecto Marsaglia, según el cual los vectores formados agrupando d números aleatorios de la secuencia tienden a concentrarse en hiperplanos del espacio d-dimensional). Sus ventajas son la sencillez de implementación y que sus limitaciones son conocidas gracias a estudios teóricos. Existen ciertos tratamientos algorítmicos que permiten combinar dos de estas secuencias para mejorar relativamente el resultado final de estos generadores (métodos como el *shuffling* y el *bit-mixing*) [Press,1992], [Knuth,1981].

Una variante más reciente de generador y bastante extendida es la conocida por generadores de registro de desplazamiento (*shift-registers*) o Tausworthe.

$$b_i = (b_{i-p} \oplus b_{i-q})_{\mathrm{mod}ulo_2}$$

(4. 2)

De acuerdo con la fórmula anterior se genera un nuevo bit de la secuencia aplicando la función lógica OR-exclusiva sobre dos bits anteriores de la secuencia, a distancia p y q respectivamente del que se va a calcular. Los números aleatorios se forman agrupando el número necesario de bits. Su estudio teórico, basado en polinomios primitivos en los

campos de Galois, no está muy desarrollado pero han superado bien las pruebas de evaluación. Una de sus ventajas es que su ciclo no depende de la longitud de los números aleatorios generados sino del parámetro p, con lo que se consiguen secuencias con periodo arbitrariamente largo. Ejemplos de estos generadores son los obtenidos con (p,q) igual a (250,103), (521,32) y (1279,1063). En el primer ejemplo el ciclo obtenido es $2^p-1 \cong 10^{75}$.

Según James los únicos generadores de calidad aceptable son RANECU de l'Ecuyer, RANMAR de Marsaglia, Zaman y Tsang y ACARRY de Marsaglia y Zaman [James,1990], que no son precisamente los que acostumbran a estar disponibles en las bibliotecas de funciones de los ordenadores. Por sus mejores características nosotros hemos escogido el algoritmo ACARRY, que está basado en operaciones de suma con desbordamiento (*add & carry* o, alternativamente, *substract & borrow*):

$$x_n = (x_{n-r} \pm x_{n-s} \pm c)_{\text{mod}\,ulo_b} \tag{4. 3}$$

donde r y s son los parámetros de desplazamiento y c es un bit de desbordamiento de la operación. Una buena elección para generar números aleatorios reales de 32 bits con 24 bits de mantisa (como los reales de simple precisión del Fortran) es tomar $b=2^{24}$, $r=24$ y $s=10$. En este caso se obtiene una secuencia con un ciclo de 10^{171}. El estado del generador queda especificado por 24 semillas, dos índices enteros y un bit de desbordamiento. La inicialización de la secuencia es posible con una pequeña rutina de inicialización que tiene como único parámetro de entrada un entero de valor comprendido entre 0 y 900 millones (9 dígitos decimales). Una gran ventaja de este generador, a parte de sus buenas propiedades estadísticas y que es totalmente portable sobre distintas plataformas informáticas, es que permite la generación fácil de secuencias disjuntas: hasta 10^9 secuencias por *usuario* con longitud de ciclo 10^{161}. Para ello basta con asignar a cada usuario una combinación concreta de los cuatro últimos dígitos decimales del entero utilizado en la inicialización de la secuencia. Esto será muy útil como veremos en la paralelización del simulador, donde el *usuario* no será un investigador sino cada proceso (*thread*) de la ejecución paralela.

4.2 Supercomputación y paralelismo.

4.2.1 Tendencias en la supercomputación.

La supercomputación ha ido evolucionando con el desarrollo de diferentes arquitecturas para los computadores: escalares, escalares segmentados, vectoriales y paralelos.

Los computadores escalares son aquellos que en un determinado instante de tiempo sólo pueden estar procesando una única instrucción que se lleva a cabo, además, sobre un dato escalar. Su rendimiento depende, pues, de utilizar la tecnología más rápida. Una evolución consiste en utilizar un procesador escalar segmentado, es decir, capaz de subdividir la ejecución de una instrucción cualquiera en unos segmentos básicos (por ejemplo: búsqueda, decodificación, lectura de datos, cálculos y escritura de datos) que se pueden realizar en bloques funcionales específicos e independientes del procesador. De esta manera, se puede llegar a encadenar la ejecución de tantas instrucciones como segmentos tenga definidos el procesador, con el consiguiente incremento de velocidad de proceso del computador. Ésta técnica se conoce como *pipelining* y su introducción está ligada a la aparición de la filosofía RISC (*Reduced Instruction Set Code*). Otra posibilidad es la utilización de un procesador superescalar, aquel que tiene replicadas algunas de sus unidades funcionales, en especial las de aritmética en coma flotante, con el fin de poder lanzar múltiples instrucciones en un mismo ciclo máquina. Se basan también en la filosofía RISC y en utilizar la tecnología más rápida disponible.

Hacia 1975 empezaron a aparecer los primeros procesadores vectoriales, que deben su nombre al hecho de trabajar con datos (e instrucciones) de tipo vectorial. Procesan simultáneamente todo un vector de datos escalares (cuya longitud está fijada de antemano por el propio diseño del procesador), sobre los que se aplica una misma operación. Si bien esto representa una limitación algorítmica, es una situación muy habitual en el cálculo científico. La velocidad de cálculo se basaba en el proceso vectorial de datos y en la utilización de la tecnología más rápida del momento (léase también, más cara) para el diseño del procesador.

Las últimas generaciones de supercomputadores se basan en reunir en una misma máquina un conjunto de procesadores que pueden cooperar en la realización de una tarea común. Estos computadores, llamados paralelos, son sistemas escalables, es decir, concebidos para poder formar máquinas con capacidad de proceso adaptable a las necesidades (y posibilidades económicas) del cliente. En este sentido, su estructura ha de permitir ir agregando procesadores y unidades de memoria, así como facilidades de comunicación (red de interconexión) entre los nodos según una estructura regular. En algunos casos se pueden llegar a reunir cientos o miles de procesadores, con lo que se habla de computación masivamente paralela.

La idea básica del procesamiento en paralelo es la subdivisión del problema en un conjunto de partes resolubles de forma concurrente, de manera que el tiempo total de resolución del problema quede dividido por el número de procesadores utilizado. El grado de consecución del objetivo dependerá básicamente de dos factores: la sobrecarga computacional generada por la paralelización del problema (creación de procesos subordinados, inicialización y finalización de regiones paralelas, sincronización de procesos, etc.) y del equilibrio de carga conseguido entre el conjunto de procesadores.

Actualmente la computación masivamente paralela se ve como la única alternativa viable para superar los grandes retos pendientes de la supercomputación. Las máquinas paralelas presentan ventajas evidentes sobre sus antecesoras superescalares y vectoriales, puesto que pueden ofrecer un rendimiento mucho mayor a un costo muy inferior. Su irrupción en el mundo de la supercomputación es un hecho que se ha venido consolidando en los últimos años, no sólo ya en entornos de centros de investigación y universidades sino también en empresas privadas. Como ejemplo del cambio de tendencia citemos el caso de la empresa Mobil Corp. que, para cálculos relacionados con movimientos sísmicos, optó poco después de su aparición en el mercado en 1993 por la compra de una Connection Machine CM-5 de 128 procesadores frente al supercomputador vectorial Cray Y-MP. Mientras que la primera necesitaba 10 días para realizar los cálculos y costaba 100.000 dólares, la segunda tardaba 29 semanas y costaba 2.8 millones de dólares. La inversión y mejoras tecnológicas necesarias para conseguir un supercomputador clásico superescalar o vectorial que mejore su versión predecesora suponen un esfuerzo enorme en todas las fases de su realización, en especial de las costosas tecnologías de sus componentes básicos (procesadores basados en materiales superconductores). Ante este panorama, los computadores masivamente paralelos

presentan una alternativa muy interesante puesto que pueden configurarse a partir de elementos mucho más modestos sin precisar de elementos de base tan caros ni sofisticados. Por supuesto, las máquinas multiprocesador pueden construirse y de hecho se construyen con procesadores segmentados y superescalares para obtener así mejores resultados, pero con los de tecnología más común del momento, puesto que la potencia global de la máquina no se fundamenta en la potencia individual de cada procesador sino en la del conjunto.

4.2.2 Sistemas paralelos

El concepto de computador paralelo engloba todo un abanico de sistemas distribuidos de diversa índole. Su clasificación es posible atendiendo a los siguientes criterios: flujo de instrucciones y de datos, organización de la memoria, topología de interconexión, modelo de programación y granularidad del paralelismo. A continuación pasamos a detallar brevemente cada uno de estos aspectos.

Atendiendo al flujo de instrucciones y de datos, Flynn clasifica los sistemas de computación en cuatro grupos [Flynn,1972]:

a) SISD (Single Instruction, Single Data), que engloba a todos los sistemas secuenciales.

b) SIMD (Single Instruction, Multiple Data), que constan de un conjunto de procesadores sincronizados que ejecutan simultáneamente la misma instrucción sobre los datos locales de cada procesador. Todo el funcionamiento gira alrededor de una única unidad de control, que es la encargada de interaccionar con el usuario y de controlar el flujo de instrucciones a ejecutar. Las matrices de procesadores o *array processors* son el ejemplo típico de esta concepción de sistema.

c) MISD (Multiple Instruction, Single Data), que no corresponde a ningún sistema real.

d) MIMD (Multiple Instruction, Multiple Data), que se atribuye a los sistemas formados por un conjunto de elementos independientes, tanto en el proceso

como en el control, de forma que cada uno puede realizar tareas diferentes sobre datos disjuntos.

Las primeras máquinas paralelas respondían al tipo SIMD por ser más simples de construcción y por su menor costo. Reunían un número masivo (miles) de procesadores, representaron un cambio radical en el estilo de programación y consiguieron resultados espectaculares de velocidad de proceso. Pero el hecho de que sólo se adapten bien a ciertos tipos de problemas propició el desarrollo de máquinas MIMD actuales, más complejas pero también más eficientes en problemas generales.

En cuanto a la organización de la memoria y la forma en que ésta es accedida por los procesadores, los sistemas se clasifican en tres categorías:

a) sistemas de memoria compartida,

b) sistemas de memoria distribuida y,

c) sistemas mixtos.

La memoria compartida da lugar al modo de programación más simple. La memoria suele estar estructurada en módulos para permitir el acceso simultáneo a memoria por parte de varios procesadores. Aún así, los conflictos de acceso pueden degradar mucho el rendimiento del sistema. En los sistemas de memoria distribuida cada procesador cuenta con su memoria local privada. Cualquier intercambio de información se lleva a cabo a través de la red de interconexión por medio de paso de mensajes. El costo de la comunicación dependerá de la topología de la red, en la que cabe considerar dos aspectos: su regularidad, de la que se desprenderá la facilidad para encontrar un camino entre los procesadores involucrados en la comunicación y, su diámetro o distancia mínima entre dos nodos cualesquiera, que repercutirá sobre el retardo en la recepción del mensaje. En los sistemas mixtos la memoria está físicamente distribuida entre el conjunto de procesadores que forman el sistema, pero son presentados al programador como un único espacio virtual de memoria.

La topología de la red es también una característica importante de los sistemas multiprocesador. Hay muchas posibilidades que van desde las topologías de bus, anillo, árbol, red bidimensional, etc. hasta toroides, redes tridimensionales e hipercubos. En cada caso se establece un compromiso entre el costo de la comunicación y el coste de la

estructura. De todas las anteriores, la topología hipercubo es la que posee menor diámetro y mayor conectividad entre procesadores.

El modelo de programación permite clasificar los sistemas paralelos en tres grandes grupos:

a) computación distribuida, por la que se subdivide el problema a resolver en un cierto número de partes relativamente independientes que se pueden resolver, por tanto, de forma local en procesadores distintos (por ejemplo, un grupo de estaciones de trabajo) y con pocas comunicaciones.

b) paso de mensajes; el habitual en sistemas masivamente paralelos. Está basado en el intercambio de paquetes de información cuyo encaminamiento puede ser automático o, frecuentemente, a cargo del programador de la aplicación. La eficiencia de un programa dependerá de la fracción de tiempo gastado por el conjunto de procesadores en el establecimiento de comunicaciones respecto al tiempo de proceso efectivo de datos.

c) paralelismo de datos o SPMD (Single Program, Multiple Data), término que frecuentemente se aplica de forma confusa cuando el modelo de programación no se corresponde a ninguno de los dos anteriores. Es el modelo típico usado en las arquitecturas SIMD.

Por último, los sistemas paralelos se distinguen por el grado de paralelismo que pueden explotar, denominado habitualmente "granularidad". Se habla de granularidad gruesa cuando un procesador puede realizar muchos cálculos (instrucciones) sin necesidad de comunicarse con el resto, y de granularidad fina en caso contrario. La granularidad gruesa es adecuada para arquitecturas MIMD (con procesadores potentes y poco interconectados) y del modelo de programación de computación distribuida. La granularidad fina es más explotable en arquitecturas SIMD y paralelismo de datos.

La programación de sistemas paralelos no ha logrado encontrar todavía ningún mecanismo de programación que sea independiente de la máquina. Uno de los mecanismos habituales de programación es la utilización de directivas de compilación (en general, dependientes de la máquina). Es el programador quien las debe insertar en el código fuente. Sirven para especificar aspectos relativos al almacenamiento de datos

en memoria, distinguir los datos privados de los compartidos, especificar las partes de código a ejecutar de forma paralela o secuencial, etc. En entornos científicos y de ingeniería los lenguajes más utilizados continúan siendo el Fortran y el C sobre los que se han desarrollado y se continúan desarrollando versiones para explotar mejor las características de los nuevos sistemas paralelos (Fortran 90, Fortran D, HPF, Vienna Fortran, o C*, pC, etc.). El HPF (High-Performance Fortran) es un claro ejemplo del esfuerzo realizado para establecer una plataforma estándar sobre la que se puedan basar los fabricantes, tanto de software como de hardware. A pesar de todo, en muchos casos se continúan utilizando los lenguajes secuenciales estándar (Fortran 77 o C), a los cuales se les añaden las directivas de paralelización para el compilador a través de líneas de comentario especiales en los ficheros fuente.

Con el fin de facilitar la programación por paso de mensajes, se han desarrollado paquetes de comunicaciones independientes de la topología que transfieren al compilador la labor de encaminamiento de los mensajes: PARMAC, PVM, MPI, etc.

4.2.3 Parámetros de medida del paralelismo

Hay una serie de conceptos básicos en la terminología de la computación paralela que, por su uso frecuente, revisaremos aquí brevemente. El primero de ellos es la aceleración o *speedup*, que es una medida de la ganancia de velocidad conseguida por la ejecución en paralelo del problema respecto a su ejecución secuencial. Denominando $T(p)$ al tiempo de ejecución necesario para la resolución del problema usando p procesadores, se tiene que el *speedup*, $S(p)$, se expresa como:

$$S(p) = T(1) / T(p) \tag{4. 4}$$

Idealmente se tendría que $S(p)=p$, aunque esto difícilmente se consigue debido a la existencia de partes no paralelizables en el problema (fracción secuencial, $1-f$, frente a la fracción paralela f) y a la sobrecarga computacional (*overhead*, T_{ovh}) generada por el propio proceso de paralelización (división de tareas, sincronización, intercambio de datos, etc.). La Figura 4. 1 muestra la evolución típica del *speedup* con el número de procesadores utilizado. Para cada problema, arquitectura de computador e

implementación existirá un número de procesadores por encima del cual no se conseguirá acelerar ya la resolución del problema. El límite superior impuesto por la simple existencia de una fracción secuencial en el problema es conocido como ley (o regla) de Amdahl. Puesto que los tiempos se pueden expresar como:

$$T(1) = f \cdot T(1) + (1-f) \cdot T(1) \tag{4.5}$$

$$T(p) = \frac{f}{p} \cdot T(1) + (1-f) \cdot T(1) + T_{ovh} \tag{4.6}$$

es inmediato aplicar la definición del *speedup* para hallar la ley de Amdahl:

$$S(p) \leq \frac{1}{(1-f)} \tag{4.7}$$

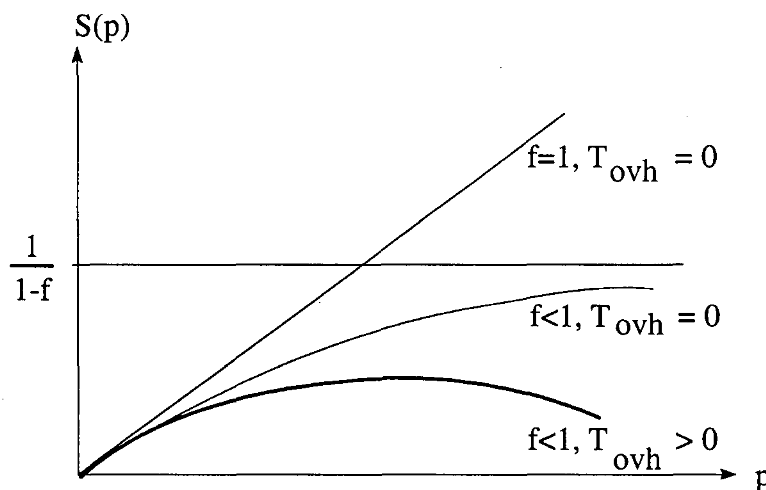

Figura 4. 1 Curvas típicas de evolución del *speedup* con el número de procesadores.

Operando con las expresiones anteriores y asumiendo un T_{ovh} despreciable se puede encontrar una aproximación por defecto de la fracción de código que se ha conseguido paralelizar con los p procesadores utilizados:

$$f \leq \frac{p}{(p-1)} \cdot \frac{S(p)-1}{S(p)} \tag{4.8}$$

La Figura 4. 2 representa *S(p)* en función de la fracción *f* de código paralelizable con el número de procesadores como parámetro. Se observa como para valores de *f* inferiores al 80% no tiene mucho sentido utilizar un número elevado de procesadores, puesto que el beneficio en *S(p)* es escaso.

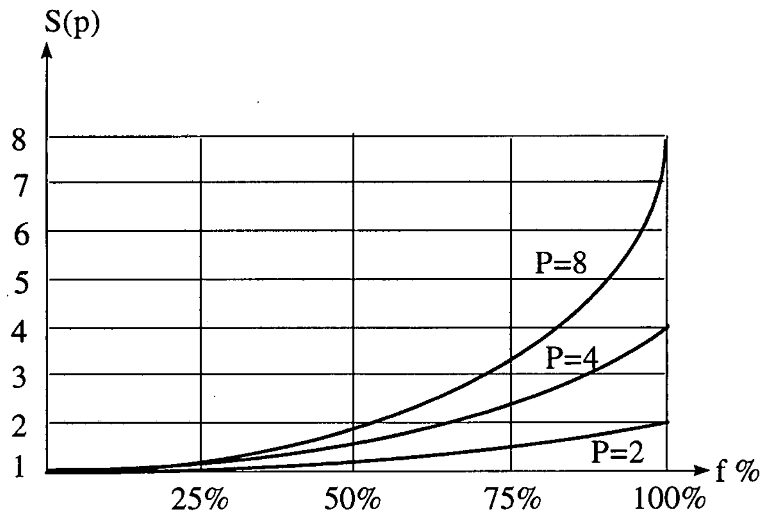

Figura 4. 2 *Speedup* en función de la fracción paralelizable del código.

Sin embargo, fijarse sólo en el *speedup* y la ley de Amdahl es la visión pesimista del problema. Cuando se tiene acceso a un mayor número de procesadores se puede aumentar el tamaño del problema planteado o resolverlo con mayor precisión reduciendo el tamaño de la malla de discretización. Además, en algunos casos se obtienen beneficios no considerados en la formulación anterior, puesto que el incremento en el número de datos puede aumentar las oportunidades de paralelismo o, simplemente, porque puede aumentar la localidad de los datos, con lo cual los tiempos de acceso a memoria disminuyen y se mejora el tiempo de ejecución total. En casos excepcionales estos factores hacen que se puedan conseguir valores de *speedup* incluso mayores que el ideal o lineal; se trata del llamado *speedup* superlineal.

Otro término de uso común es la eficiencia *E(p)* del sistema paralelo, que está relacionada con el *speedup* por la siguiente expresión:

$$E(p) = \frac{S(p)}{p} \tag{4.9}$$

Así, pues, en el caso ideal o lineal la eficiencia será del 100% pero en general, decaerá a medida que aumente el número de procesadores. En definitiva, el parámetro *E(p)* da una medida del rendimiento que se obtiene del número de procesadores utilizado.

4.3 Supercomputación y MCHBT.

El simulador de dispositivos MCHBT fue desarrollado en sus primeras fases sobre estaciones de trabajo UNIX Sun SPARC10, de ejecución secuencial sobre un único procesador de tipo RISC. Desde el principio el estilo de programación utilizado en la implementación del código en lenguaje Fortran 77 tuvo en su punto de mira la búsqueda de portabilidad posterior a máquinas más potentes basadas en técnicas de supercomputación paralela. Cuando empezó a quedar consolidada su estructura básica y empezaron a salir los primeros resultados de simulación, se vio pronto que había llegado el momento esperado de cambio de plataforma.

Por proximidad y accesibilidad se escogió utilizar los recursos disponibles en el CEPBA, el Centre Europeu de Paral·lelisme de Barcelona, y en concreto el supercomputador paralelo *karnak*. Se trata de una máquina perteneciente a la familia Origin 2000 de SiliconGraphics Inc. Está concebida para análisis de grandes aplicaciones de ingeniería, simulación de choques, cálculos sísmicos, procesado de imagen, etc. Es un sistema escalable desde 1 hasta 128 procesadores RISC R10000 a 250MHz. La máquina del CEPBA consta actualmente de 64 procesadores. Las prestaciones del sistema de 128 procesadores son las siguientes: 51.2 Gflops, 256 GB de memoria RAM, ancho de banda para E/S de 81.9 GB/s en modo sostenido (102.4 GB/s de pico), canal de fibra de 74 TB, Ultra SCSI de 14 TB y 20.5 GB/s de ancho de banda de bisección (26 GB/s de pico).

Por cuestiones de precio, el sistema de memoria es jerárquico y consta de los siguientes elementos enumerados por orden decreciente de velocidad de acceso: registros internos de la unidad central de proceso (CPU), memorias *cache* primarias (L1) de instrucciones y de datos, memoria *cache* secundaria (L2) unificada para instrucciones y datos,

memoria principal del sistema (RAM) y memoria virtual (discos). Tanto los registros como el primer nivel de memorias *cache* son internos al circuito integrado de cada procesador R10000. Existe además otro tipo de memoria *cache* interna dedicada a guardar un cierto número de correspondencias entre las direcciones virtuales y las direcciones físicas de los datos accedidos más recientemente (TLB, *Translation Lookaside Buffer*). Cuanto más rápido más caro resulta un elemento de memoria y por tanto se diseña para capacidades de almacenamiento más reducidas. Los elementos más rápidos se ubican más cerca del procesador, puesto que deben ser capaces de suministrarle los datos que necesita al elevado ritmo al que los solicita para conseguir las velocidades de computación para las que ha sido diseñado. Su capacidad limitada de almacenamiento hace que tarde o temprano fallen en su misión y el nuevo dato tenga que ser localizado y traído desde niveles inferiores de la jerarquía de memoria, con el consiguiente retraso que esto introduce en la velocidad de cálculo del procesador. Es evidente que por su escaso número y capacidad los datos requeridos no estarán casi nunca disponibles en los registros del propio procesador, pero sí que es deseable y se espera (puesto que se utilizan algoritmos de prebúsqueda de datos) que en general se hallen ya preparados en el primer o segundo nivel de memoria *cache*. Cuando esto no sucede, se habla de fallo de *cache* (*cache miss*) y constituye uno de los principales focos de atención en el desarrollo de programas eficientes. Para reducir el coste que supone traer un dato desde jerarquías inferiores de memoria, la operación se realiza por bloques tomando también los datos adyacentes al dato solicitado, con la esperanza de que serán requeridos en breve por el procesador. Así, las transferencias de disco a memoria principal (RAM) se hacen por *páginas* y las de *cache* se hacen por *líneas*.

Para conseguir un sistema escalable tanto en prestaciones (latencia y ancho de banda) como en precio, la arquitectura de Origin posee una topología de interconexión entre procesadores del tipo hipercubo, una interconexión procesador-memoria de tipo *modular crossbar* y modelo de memoria compartida (*shared memory*) aunque implementado en forma de memoria físicamente distribuida. Esta implementación recibe el nombre de DSM (*Distributed Shared Memory*) y consigue utilizar el modelo de una única memoria compartida a nivel virtual, mientras que a nivel físico se trata de memoria distribuida por motivos de escalabilidad. Esté donde esté un dato, su dirección es la misma para cualquier procesador que quiera acceder a él.

Tanto por motivos jerárquicos del sistema de memoria como por su implementación física distribuida, se trata de un sistema NUMA (*Non-Uniform Memory Access*), es decir, que el tiempo de acceso a un dato no es uniforme, pues depende enormemente de donde esté ubicado el dato en cada momento (Tabla 4. 1). Este aspecto es muy importante y puede llegar a ser el factor que domine por completo el tiempo de ejecución de una aplicación. Una buena política de almacenamiento de datos y de acceso con reutilización es la clave para conseguir buenos rendimientos en procesadores RISC y sistemas jerárquicos de memoria.

Registros	0 ns
L1-*cache*	5 ns
L2-*cache*	75
RAM local	313
RAM hubs	497
RAM routers	497+n*103 ns

Tabla 4. 1 Latencias en la jerarquía de memoria del Origin2000.

Los procesadores R10000 son procesadores RISC, poseen 5 unidades funcionales de cálculo (3 para operaciones en coma flotante y 2 unidades aritmético-lógicas para operar sobre datos enteros), y son cuádruplamente superescalares (pueden llegar a operar simultáneamente con 2 de las unidades de coma flotante y con las 2 ALUs). Con la aplicación de técnicas de *pipelining* son capaces de generar en régimen permanente hasta dos resultados en coma flotante por ciclo máquina. Para favorecer un flujo continuo y constante de datos de entrada son capaces de reordenar instrucciones (dentro de una ventana de observación de 32 instrucciones) y ejecutarlas fuera de orden, por supuesto respetando en todo momento la coherencia de resultados. Con el mismo objetivo especulan en todo momento hasta el cuarto nivel de profundidad sobre posibles caminos a seguir por la ejecución y preparan los datos involucrados en función de las condiciones de salto previstas en el flujo de instrucciones del programa. En general estos procedimientos generan fallos de *cache* anticipadamente y de forma solapada con

la ejecución regular del programa, por lo que su efecto catastrófico queda anulado o minimizado.

Las memorias *cache* primarias (L1) están integradas con el procesador. La de datos tiene una capacidad total de 32 KB y se actualiza por líneas de 32 octetos (correspondiente, en Fortran 77, a 8 reales simples o 4 de doble precisión que ocupan posiciones consecutivas de memoria). La de instrucciones tiene una capacidad total de 32 KB y se actualiza por líneas de 128 octetos. Ambas utilizan una política de renovación del tipo *2-way set associative*, para minimizar las sobreescrituras de datos todavía necesarios. El ancho de banda de conexión con los bancos de registros es de 1.56 GB/s, mientras que el de conexión con la *cache* secundaria (L2) es 2.08 GB/s.

La Tabla 4. 2 muestra la latencia y el ciclo de repetición (medidos en ciclos máquina) asociados a cada unidad funcional de cálculo y a la unidad de direcciones. Destacan claramente el elevado coste de las operaciones de división y de cálculo de raíces cuadradas. Cabe destacar que es posible solapar el funcionamiento de todas las unidades funcionales (excepto FP3 con FP2: *4-way scalar*).

Cada procesador tiene una memoria *cache* secundaria asociada, que es externa al integrado que aloja al procesador y a sus *caches* primarias. Su capacidad total puede ser de 1, 2 o 4 MB, con líneas de 128 octetos y estructurada en 2 bloques asociativos para la renovación de su contenido. Almacena datos y direcciones indistintamente. La tecnología es SSRAM. El ancho de banda de su unión con la memoria principal es de 780 MB/s y de 2.08 GB/s con las *caches* primarias.

UNIDADES DE ENTEROS:	LATENCIA	C. REPET.
* **ALU1:** add, sub, logic ops, shift, branches	1	1
* **ALU2:** add, sub, logic ops multiply (32/64 bits) divide (32/64 bits)	1 6/10 35/67	1 6/10 35/67
UNIDADES DE DIRECCIONES: load INT/FP store	 2/3 	 1 1
UNIDADES DE COMA FLOTANTE:		
* **FPU1:** add, sub, compare, convert	2	1
* **FPU2:** multiply multiply-add (MADD))	 2 4	 1 1
* **FPU3:** divide (32/64 bits) sqrt (32/64 bits)	 12/19 18/33	 14/21 20/35

Tabla 4. 2 Latencias de las unidades funcionales en el Origin.

El bloque básico de la arquitectura de Origin es el nodo, que constructivamente forma un sistema completo sobre una sola placa de circuito impreso. Un nodo consta de 2 procesadores con sus respectivas memorias *cache*, un bloque de memoria RAM (de 64 MB a 4GB) y un bloque de interconexión (*hub*) entre: los dos procesadores (780 MB/s de pico de ancho de banda), procesador-memoria RAM local (780 MB/s), procesador-

puertos E/S (2*780 MB/s) y procesador-red de interconexión global (2*780 MB/s). A partir de la interconexión de nodos según la topología de hipercubo el sistema puede ir creciendo hasta el máximo de 128 procesadores, aportando cada nuevo nodo su parte de capacidad de cálculo, de memoria, de ancho de banda de E/S, etc. Al conjunto de 8 nodos se le denomina módulo.

La ejecución de los programas puede ser secuencial sobre un solo procesador o paralela seleccionando cualquier número disponible de procesadores. El número de procesadores deseado se puede escoger desde la misma línea de comando con la que se envía el programa a ejecutar, o asignando una variable de entorno específica e incluso desde el mismo programa incluyendo en el código fuente una llamada a funciones de la biblioteca de utilidades ***mp***.

Para ejecuciones paralelas Origin2000 puede utilizar los modelos de programación de memoria compartida o el de paso de mensajes. Existe incluso la posibilidad de utilizar un método híbrido que utiliza el paso de mensajes para paralelización de grano grueso y aspectos del modelo de memoria compartida (con autoparalelización y directivas al compilador) para la granularidad fina. En lo que respecta al modelo de memoria compartida existen directivas al compilador para: a) optimización del uso de las prestaciones de las CPUs, b) para el control de paralelización y c) para la ubicación de datos en memoria y su distribución entre procesadores. En cuanto al modelo de paso de mensajes se incluyen versiones de los paquetes MPI y PVM puestas a punto para la arquitectura de Origin.

La metodología de desarrollo y optimización se puede dividir en dos partes: secuencial y paralela. Éste es el tema de los dos apartados siguientes, que se centrarán en los aspectos abordados durante el desarrollo y optimización del programa MCHBT. En general, y más especialmente cuando se va a paralelizar utilizando la aproximación por directivas al compilador, se recomienda empezar por el desarrollo de una versión secuencial del código. Una vez se ha comprobado que los resultados proporcionados por el código son correctos y se ha optimizado su ejecución para uno cualquiera de los procesadores, se puede proceder a paralelizarlo.

4.4 Optimizaciones secuenciales en MCHBT.

Las optimizaciones secuenciales en un procesador RISC como es el caso del R10000 del computador Origin 2000 se basan en dos aspectos fundamentales: la optimización del uso de la CPU y la optimización del uso del sistema de memoria.

La optimización de uso de las unidades funcionales de la CPU se puede abordar desde un punto de vista de alto nivel creando más posibilidades de ejecución superescalar, o desde un punto de vista de bajo nivel mejorando la planificación de la secuencia de instrucciones a realizar (*instruction scheduling*). En ambos casos el fin perseguido es conseguir rebajar el tiempo de ejecución total de la tarea encomendada intentando solapar las latencias de unas unidades funcionales con otras, manteniéndolas así a pleno rendimiento.

La optimización del uso del sistema jerárquico de memoria se basa en utilizar un buen patrón de acceso a los datos, en un uso óptimo de las líneas de las memorias *cache* y en la reutilización de los datos que han sido traídos hasta las memorias *cache*.

Recordemos que los datos son transferidos por páginas a la memoria principal RAM, por líneas a las memorias *cache* y por elementos a los registros del procesador. Puesto que en FORTRAN los elementos de un vector o de una matriz se almacenan en posiciones consecutivas de memoria, un buen patrón de acceso a los datos querrá decir que las instrucciones, en especial las que formen el cuerpo de bucles iterativos, deberían estar organizadas de forma que hagan referencia a los datos siguiendo el mismo orden correlativo con el que están guardados en memoria (*stride* = 1). Además, en FORTRAN está normalizado que los elementos de matrices se guarden por columnas, es decir, que a partir del elemento más bajo de la matriz encontraríamos todos los demás elementos en posiciones correlativas haciendo variar primero el índice correspondiente a la dimensión más baja. Así, pues, aquellos algoritmos que recorran las matrices por columnas tendrán a su favor un menor tiempo de acceso a los datos respecto a aquellos que las recorran por filas o siguiendo cualquier otro patrón.

Un buen uso de las líneas de las memorias *cache* consistirá no sólo en acceder a los datos de forma correlativa (*stride* = 1) sino también en evitar que diferentes datos implicados en un mismo cálculo compitan por ocupar una misma posición de la

memoria *cache*, lo que provocaría fallos de *cache* continuos cada vez que el procesador referenciara uno de estos datos, con la consiguiente penalización de tiempo que ello supondría. La mejor forma de entender el problema es con un ejemplo. Para ello consideremos el siguiente código, que trabaja con vectores de 64KB cada uno, que es múltiplo de la capacidad total de la *cache* primaria (32KB):

```
parameter (nmax=16*1024)
real*4   a(nmax),b(nmax),c(nmax),...
...
do j =1,n
   c(j) = a(j)*b(j)
end do
```

Cuando se inicien los cálculos el procesador necesitará acceder a los elementos a(j) y b(j) e, inmediatamente después, a c(j) para guardar el resultado del producto. La dirección de memoria *cache* de los tres elementos es la misma, puesto que se calcula como el resto de la división entera por el tamaño de la *cache*. Aún así, los dos elementos a(j) y b(j) podrán encontrarse simultáneamente en la *cache* primaria puesto ésta es asociativa de dos bloques, lo cual significa que puede resolver el conflicto ubicando un dato en cada bloque. Pero para guardar el resultado se deberá acceder a c(j). Como c(j) no cabe simultáneamente, se deberá ir a buscar el dato a la *cache* secundaria o a la memoria principal. Cuando llegue el dato sobreescribirá a(j), lo cual no sería muy grave en este bucle, porque en principio a(j) ya no se va a necesitar de nuevo. El problema reside en que c(j) se carga conjuntamente con toda una línea de datos de la *cache*, con lo que estaremos sobreescribiendo no sólo a(j) sino también sus elementos adyacentes, algunos de los cuales deberán ser accedidos a continuación. Esto provocará fallos consecutivos de *cache* primaria a cada iteración del bucle, con lo cual la ejecución será lentísima.

De forma análoga, el problema puede presentarse con los accesos a datos de la *cache* secundaria, con el agravante que los fallos de *cache* son más graves puesto que la latencia en este nivel es bastante mayor.

Para solucionar el problema basta con redimensionar los vectores evitando utilizar longitudes que sean potencias de 2. Si no es posible redimensionar los vectores, la solución consiste en introducir una distancia relativa entre los elementos que deban ser accedidos simultáneamente. Esta técnica se conoce como padding y se consigue

intercalando un vector auxiliar de longitud adecuada en la declaración de datos (que es donde se fija la posición relativa que ocuparán los datos en memoria). La longitud adecuada en este caso sería de 32 elementos del tipo *real*4*, que son los que caben en una misma línea de *cache* secundaria. Así se minimizarían los fallos de *cache* primaria y secundaria sacrificando una porción insignificante de memoria total.

```
parameter (nmax=16*1024, L2 = 32)
real*4    a(nmax),pad1(L2),b(nmax),pad2(L2),c(nmax)
```

Por último, otro buen objetivo de un algoritmo que quiera sacar rendimiento de los accesos a memoria es intentar reutilizar al máximo los datos que han llegado hasta las memorias *cache*. Dicho de otro modo, cada vez que se acceda a un dato cualquiera el algoritmo debería intentar, en la medida de lo posible, realizar inmediatamente todas las operaciones en las que tarde o temprano deba intervenir aquel dato, de forma que se amortice mejor el coste del acceso a memoria. Por supuesto, la reutilización se debería intentar no sólo con ese dato sino con todos los de la línea que han llegado "gratuitamente" con el mismo acceso.

Existe un conjunto de técnicas habituales aplicadas con fines de optimización de prestaciones en procesadores de tipo RISC. En su nomenclatura inglesa habitual se trata de las siguientes técnicas: *software pipelining (SWP), loop unrolling, loop fusion, loop fission (splitting), loop interchange, loop blocking* y *procedure inlining*. Otra herramienta útil es la directiva de compilación *prefetch*(). Algunas son beneficiosas tanto para las prestaciones de la CPU como para las del sistema de memoria. Otras, por el contrario, pueden beneficiar un aspecto y ser perjudiciales para el otro. Su utilización dependerá, pues, de una solución de compromiso particular del problema que nos ocupe en cada caso. La Tabla 4. 3 resume sinópticamente los beneficios generales esperables con cada una de estas técnicas.

	instrucción	memoria
SWP	++	--
loop unrolling	++	+
loop fusion	+	++
loop fission	+	++
loop interchange	-	++
loop blocking	-	++
inlining	++	++

Tabla 4. 3 Técnicas habituales de optimización en arquitecturas RISC

Todas las técnicas anteriores pueden ser aplicadas de forma automática por el compilador del Origin 2000, aunque éste no es siempre capaz de reconocer todas las situaciones en las que su uso es beneficioso. En general, el compilador obtiene buenos resultados en algoritmos estándar y bien estructurados. Para casos más complejos se pueden usar opciones de compilación para instruir al compilador y facilitarle la labor. En cualquier caso, también es posible aplicar las técnicas de forma manual durante el proceso de programación escogiendo el código adecuado para ello. En algunos casos la solución puede ser muy simple, pero en otros puede convertirse en una labor más bien tediosa. El compilador podrá obtener buenas optimizaciones a nivel de instrucción; el programador deberá hacer uso de su visión global del problema para identificar optimizaciones de alto nivel.

El proceso de optimización de la aplicación informática comienza, pues, con la concepción y desarrollo de la misma. Puesto que en los sistemas RISC los accesos a memoria tienden a ser más cruciales en las prestaciones finales del sistema que las optimizaciones en las operaciones, se recomienda:

a) desarrollar el programa teniendo en mente el sistema jerarquizado de memoria y sus implicaciones sobre el tiempo de acceso a los datos (NUMA),

b) estructurar los datos en función del uso y acceso que vaya a ser necesario para las operaciones a realizar con ellos (definición de las matrices para conseguir *stride* unidad),

c) cuidar el estilo de programación para no violar los estándares del FORTRAN (por ejemplo, evitar el *out-of-bound array indexing*),

d) potenciar el uso de bucles DO (con el número de iteraciones prefijado) frente al uso de sentencias GOTO y salidas precipitadas de bucles iterativos, con el fin de evitar saltos inesperados que causan truncamiento de secuencia, fallos de *cache* e interrupciones en el ritmo de trabajo de la CPU,

e) intentar evitar algoritmos que generen dependencias de datos en los bucles iterativos, puesto que impedirán la aplicación del SWP o la paralelización del bucle,

f) evitar el uso de sentencias EQUIVALENCE que utilizan distintos nombres (referencias) para lo que en realidad constituye una sola variable (una sola posición de memoria),

4.4.1 Técnicas de optimización en procesadores RISC

Una de las técnicas más potentes de optimización para procesadores RISC es el *software pipelining* (SWP), que es posible gracias al *hardware pipelining*, que posibilita la ejecución segmentada de las instrucciones. Consiste en ejecutar de forma superpuesta las instrucciones de los bucles iterativos. Para ello el bucle original se subdivide en tres fases: el prólogo, el régimen permanente y el epílogo. La parte fundamental y deseada es la de régimen permanente, durante la cual están en proceso de ejecución simultánea un cierto número de instrucciones que corresponden a iteraciones consecutivas del bucle original. Considérese el siguiente bucle a modo de ejemplo:

```
do j=1,n
   a(j)=a(j)+b(j)*c(j)
enddo
```

Las operaciones a realizar en cada iteración del bucle se pueden realizar con una sola instrucción de lenguaje máquina del procesador R10000: la instrucción de multiplicación-suma (*madd*) que, como se vio, se lleva a cabo en la unidad funcional FPU2. Esta instrucción tiene una latencia de 4 ciclos máquina y una tasa de repetición de un ciclo. Ello quiere decir que en una ejecución normal, desde el momento en que el procesador dispusiera de los datos a(j), b(j) y c(j) tardaría 4 ciclos en obtener el resultado para poder almacenarlo. En el siguiente ciclo empezaría la ejecución de la siguiente iteración y así sucesivamente se repetiría el mismo esquema, con lo que este esquema de trabajo produciría un resultado cada 4 ciclos:

Ciclo	Instrucciones en coma flotante	Instrucciones enteras
1	madd a1, b1, c1	---
2	---	---
3	---	---
4	---	store a1
6	madd a2, b2, c2	---

Tabla 4. 4 Ejecución de una instrucción *madd* sin aplicar SWP.

La aplicación de SWP en el Origin 2000 al bucle anterior daría como resultado el siguiente bucle de régimen permanente:

Ciclo	Instrucciones en coma flotante	Instrucciones enteras
j	madd a(j), b(j), c(j)	store a(j-4)
j+1	madd a(j+1), b(j+1), c(j+1)	store a(j-3)
j+2	madd a(j+2), b(j+2), c(j+2)	store a(j-2)
j+3	madd a(j+3), b(j+3), c(j+3)	store a(j-1)
j+4	madd a(j+4), b(j+4), c(j+4)	store a(j)

Tabla 4. 5 Ejecución de una instrucción *madd* aplicando SWP.

Con este esquema de trabajo se produce un resultado por ciclo, lo que representa un factor 4 de aceleración respecto al esquema anterior sin *pipelining*. Gracias al SWP se ocultan latencias de instrucciones y se puede explotar la naturaleza superescalar del procesador. Para que la aplicación del SWP sea posible es necesario que no haya dependencias de datos entre iteraciones consecutivas del bucle.

La técnica de *loop unrolling* consiste en replicar un cierto número *k* de veces la iteración de un bucle con el fin de simultanear su ejecución. Esquemáticamente, la idea es la siguiente. Dado el siguiente bucle:

```
do j=1,n,1
   ...(j)...
enddo
```

La técnica de *loop unrolling* lo descompondría de la siguiente manera:

```
do j=1,n,k
   ...(j)...
   ...(j+1)...
   ...(j+2)...
   ...(j+k-1)...
enddo
```

Las ventajas son que se favorecen las posibilidades de ejecución superescalar y de reutilización de datos (aquellos datos comunes a las *k* iteraciones *desenroscadas* se pueden mantener en registros durante la ejecución del bucle). Como inconvenientes, en

general habrá que introducir código específico para controlar que el número total de iteraciones sea el deseado (cuando la longitud total del bucle, *n*, no sea múltiplo del factor de desenroscado, *k*) y que la demanda de más registros puede llegar a saturar la disponibilidad de los mismos.

Las técnicas de *loop fusion* y de *loop fission* (o *loop splitting*) se basan en ideas contrapuestas aunque persiguen el mismo objetivo. La elección de una u otra dependerá del código contenido en los bucles considerados. En algunos casos, la fusión de bucles permite la reutilización de datos comunes a ambos bucles y crear más oportunidades de ejecución superescalar. Por contra, la fusión puede desembocar en un bucle con un exceso de variables compitiendo por un espacio en la memoria *cache* y una mayor demanda de registros. La subdivisión o fisión de un bucle puede resolver problemas de dependencias de datos en las iteraciones (con la consiguiente mejora de posiciones para poder aplicar SWP o paralelización) y mejorar la reutilización de datos (matrices independientes que en el bucle original causaran contención en las líneas de *cache*).

Cuando en un programa aparecen bucles anidados se debe examinar las ventajas de intercambiar el orden de ejecución de los mismos (*loop interchange*). Las posibles ventajas de esta técnica son la mejora en el patrón de acceso a memoria y la eliminación de dependencias de datos para incrementar las posibilidades de optimizaciones de CPU y la aplicación de paralelismo. El intercambio de bucles para conseguir alguna de las ventajas anteriores puede tener como consecuencia negativa que algún bucle corto pase a ocupar la posición más interna de la estructura de anidamiento, lo cual reduciría la eficiencia en la posible aplicación del SWP a ese bucle. Considérese por ejemplo el siguiente bucle:

```
do i = 1, n
   do j = 1, m
   c(i,j) = a(i,j) + b(i,j)
   enddo
enddo
```

Puesto que en FORTRAN las matrices se almacenan por columnas y el bucle más interno no corresponde al primer índice de las matrices, la ejecución de este bucle provocará muchos fallos y sobreescrituras (*trashing*) de datos de las memorias *cache*, a menos que el valor de *n* sea muy pequeño. Un simple intercambio en el orden de ejecución de los bucles haría coincidir el orden de almacenamiento con el orden de

acceso a los datos, con lo que se explotarían de forma óptima todos los elementos de cada línea de *cache* para cualquier valor del parámetro *n*.

```
do j = 1, m
   do i = 1, n
      c(i,j) = a(i,j) + b(i,j)
   enddo
enddo
```

En algunos casos es posible que el intercambio de bucles sea beneficioso para el acceso a unas matrices y perjudicial para otras. Éste caso se daría en el ejemplo anterior si la operación a realizar fuera:

```
c(i,j) = a(i,j) + b(j,i)
```

Para desencallar el problema habría que valorar la posibilidad de redefinir la matriz *b* intercambiando el orden de sus dimensiones en la instrucción de declaración al inicio del programa. Lo cierto es que, aunque la idea básica subyacente en esta técnica de optimización es muy simple y clara, su aplicación en casos prácticos requiere un análisis detallado del problema bajo estudio. Por ejemplo, el estudio de las seis combinaciones posibles de ordenar tres bucles anidados que realizan una operación *madd* sobre matrices tridimensionales cúbicas ($a(n,n,n)$ con $n=300$) indican diferencias de hasta un factor 35 en tiempo de ejecución.

La técnica de *loop blocking* es una optimización indicada en problemas que manipulan conjuntos de datos excesivamente grandes para caber en las memorias *cache*. Su aplicación aumenta la localidad temporal (reutilización) y espacial (utilización total de las líneas de *cache*) de los datos en aquellos casos en que no se puede acceder a los datos en el mismo orden en que están almacenados. Un ejemplo típico sería el producto de dos matrices, $C = A * B$. Supongamos por simplicidad que se trata de matrices cuadradas, NxN. Para calcular cada elemento de C deberíamos recorrer toda una fila de A y toda una columna de B. Para dimensiones razonablemente largas de las matrices, el hecho de recorrer toda una fila o columna provocará que tarde o temprano se tendrán que traer nuevos datos a las memorias *cache* que sobreescribirán los previamente almacenados en ellas. Además, los datos sobreescritos corresponden a elementos que serán necesitados en N operaciones posteriores, con lo cual el proceso de sobreescritura de datos se repetirá muchas veces. La solución consiste en subdividir las matrices en bloques imaginarios (e.g., submatrices $C11$, $C12$, $C21$, $C22$), realizar operaciones

parciales de producto sobre estos bloques (*A11*B11, A12*B21*) y finalmente combinar adecuadamente los resultados parciales para obtener la solución final deseada(*C11= A11*B11 + A12*B21*). El tamaño de los bloques se debe escoger en función del tamaño de la *cache* para asegurar que simultáneamente van a caber todos los bloques que participen en el cálculo parcial a realizar.

La técnica de *procedure inlining* es interesante porque puede beneficiar los dos aspectos esenciales de optimización, los de instrucciones y los de acceso a memoria. La idea también es bien simple. Consiste en substituir las llamadas a procedimientos (subrutinas o funciones) por el cuerpo de los mismos. Con ello se consiguen los siguientes beneficios: supresión de la carga de trabajo que supone la llamada a los procedimientos (creación/destrucción dinámica de variables locales), incremento de las posibilidades de optimización de CPU (por ejemplo SWP) y de paralelización de tareas. Los mejores candidatos para esta técnica son aquellos procedimientos pequeños en código, llamados con mucha frecuencia y cortos en tiempo de ejecución. Evidentemente, esta técnica va en contra de la filosofía de programación modular y estructurada que facilita el desarrollo claro y el mantenimiento de aplicaciones informáticas. Por tanto es recomendable utilizarla sólo de forma automática, dejando que sea el compilador el que se encargue del trabajo sin modificar para nada el código fuente original.

Por último, hemos citado una herramienta más que una técnica de optimización que es la directiva de compilación *prefetch*(var). Esta directiva se inserta en el código fuente como línea de comentario especial que es utilizada por el compilador para generar código de bajo nivel para que se empiece a buscar un dato en memoria principal (RAM) o virtual (discos) y a cargarlo en la *cache*. Esto es útil cuando por análisis del algoritmo se prevé que en breve el procesador necesitará un dato que, por no haber sido utilizado recientemente, provocaría un fallo de *cache*.

4.4.2 Herramientas de análisis de rendimiento

Seguir las pautas generales de programación que se han descrito más arriba y desarrollar el programa teniendo en mente la arquitectura del procesador y su jerarquía de memoria hacen que el programa resulte de entrada bastante eficiente. No obstante, y visto el volumen de factores a tener en cuenta y sus múltiples interacciones, es necesario

ejecutar el programa y hacer medidas de rendimiento utilizando las herramientas de análisis disponibles en el computador. Las cuatro herramientas básicas de análisis en Origin son *perfex*, *SpeedShop* (o *ssrun*), *prof* y *dprof*.

La herramienta *perfex* permite detectar los problemas principales que padece el programa. Se basa en ofrecer los valores de unos contadores internos del procesador (2 en total) especializados en monitorizar una lista de 32 sucesos específicos: ciclos máquina ejecutados, número de instrucciones decodificadas, número de instrucciones realmente ejecutadas, número de datos leído, número de datos enviados a memoria, número de instrucciones en coma flotante ejecutadas, fallos de *cache* primaria, fallos de *cache* secundaria, fallos de *TLB*, número de predicciones de salto incorrectas, etc. Durante la ejecución del código el procesador puede monitorizar exactamente dos eventos cualesquiera de los 32 en total que están definidos. Alternativamente, puede multiplexar la monitorización al conjunto de eventos y ofrecer al final de la ejecución una aproximación del valor de cada uno.

La herramienta *SpeedShop* (o el comando equivalente *ssrun*) muestrea durante la ejecución del programa el estado del contador del programa y de la pila, y recopila datos que guarda en un fichero para su posterior análisis con el comando *prof*. El análisis de estos datos permite saber dónde está el problema: qué subrutinas y qué líneas de código son las responsables del aspecto estudiado. Escogiendo entre diferentes opciones permite generar informes sobre aspectos como el tiempo de ejecución, los fallos de datos en la *cache* primaria, los fallos de datos en la *cache* secundaria, lo mismo para fallos de instrucciones, los fallos de *TLB*, las operaciones en coma flotante, etc.

Por su parte, la herramienta *dprof* permite detectar qué estructuras de datos están involucradas en los aspectos anteriores. La información de *dprof* proviene de muestreos a los accesos a memoria (páginas y direcciones de los datos accedidos), con los que se confecciona un histograma.

4.4.3 Particularización al simulador MCHBT

El proceso de optimización del simulador MCHBT incluyó el uso de las herramientas de análisis de rendimiento anteriores. Los datos obtenidos sugirieron modificaciones en

el código y cambios en las opciones de compilación. A continuación se describirán brevemente los aspectos más notables en el proceso de optimización de MCHBT y su repercusión sobre el tiempo de ejecución. Los tiempos en si mismos no tienen mucho interés, sino más bien a nivel comparativo entre ellos mismos para observar la progresión de resultados en la secuencia de optimizaciones.

Las primeras fases de desarrollo del simulador MCHBT (edición del código básico y primeras simulaciones) se llevaron a cabo sobre estaciones de trabajo Sun modelo SPARC10 y clónicos Axil320: 48-64MB de RAM, 60-75MHz. Para la simulación de un BJT de GaAs se hizo el seguimiento durante 5ps (5000 iteraciones con pasos temporales de 1fs) de un conjunto de unas 980000 partículas según el método MC autoconsistente estándar descrito en el capítulo anterior. La simulación duró 12 días, lo que representa una media de 210 microsegundos por iteración y partícula. Otra simulación realizada sobre el mismo ordenador y el mismo dispositivo BJT pero utilizando la mitad de partículas resultó en un tiempo de simulación de 2'5 días para 3'5ps (3500 iteraciones con pasos de 1fs), lo que representan unos 120 microsegundos por iteración y partícula. La comparación de éstos y otros datos de simulaciones previas hacía intuir que había aspectos de la arquitectura del computador que convenía controlar, como por ejemplo la ley de dependencia entre el incremento de tiempo de simulación con el incremento del número de datos a tratar. En cualquier caso, el tiempo de espera para tener los resultados de la simulación era demasiado largo para ser aceptable. Puestos a optimizar era mejor hacerlo sobre un computador más potente.

Para comparar a grosso modo los beneficios del cambio de plataforma se simuló de nuevo el mismo transistor considerando unas 500000 partículas durante 1ps (1000 iteraciones con paso de 1fs). La estación Sun tardó 17 horas y 10 minutos, mientras que la ejecución sobre uno de los procesadores de Origin 2000 tardó tan solo 1h 25min., es decir, un factor aproximado de 12:1, lo que significa un avance importante puesto que simulaciones que antes obligaran a esperar 12 días se podrían obtener en un solo día.

Para agilizar el proceso de optimizaciones sucesivas se preparó un problema de prueba más pequeño que redujera el tiempo de espera aunque fuera a costa de perder fiabilidad en los resultados físicos referentes al dispositivo simulado. Lo importante era disminuir suficientemente el tamaño del problema para acortar el tiempo de resolución pero sin llegar a extremos en los que se distorsionaran los tiempos relativos de ejecución de las

distintas partes del simulador. De ser así, podríamos estar optimizando el programa sobre una base errónea. Se escogió, pues, la simulación de unas 250000 partículas durante 0'1ps (100 iteraciones con pasos de 1fs). La ejecución sobre la estación Sun tardó 1h 28 min., aunque repitiendo la simulación eliminando ciertos aspectos propios de las fases de depuración de programas (como la comprobación dinámica de referencias fuera de rango en matrices o la detección de divisiones por cero) tardó sólo 31min. El mismo problema tardó 3min 28s (208s) sobre Origin 2000. Esto representa un factor de mejora de 27:1 en el primer caso y de 9:1 en el segundo.

Para las simulaciones anteriores, tanto en una máquina como en la otra, se permitía a los compiladores respectivos aplicar el máximo nivel de optimización automática. En Origin se utilizó la directiva *–O3*, que en principio intenta aplicar muchas de las técnicas de optimización que hemos presentado antes. La utilización de las opciones estándar recomendadas por el fabricante (*f77 ... -n32 –mips4 .-Ofast=ip27 –OPT:IEEE_arithmetic=3*) no supusieron beneficios apreciables sobre el tiempo de simulación. Estas opciones estándar están pensadas para informar al compilador de que los direccionamientos de memoria principal no superan los 2 GB (32 bits), que debe intentar el nivel máximo de optimización, concretar las optimizaciones para la arquitectura de Origin, aplicar optimizaciones interprocedimientos (IPA) y favorecer la velocidad sobre la precisión en las reglas de redondeo numérico.

Forzando manualmente al compilador (opción *INLINE:must=procedure*) a aplicar *procedure inlining* a las subrutinas *drift*() y *scatt*() se consiguió rebajar unos diez segundos la simulación. Estas subrutinas corresponden al cálculo de la dinámica de una partícula, por lo que forman el núcleo central del programa y son llamadas un número muy elevado de veces, por lo que se esperaba una mejora más significativa.

Instruyendo al compilador para que optimizara el código par el procesador R10000 y para que hiciera uso de la biblioteca de funciones matemáticas *libfastm* la ejecución del problema tipo se rebajó a 1min 43s (103s), la mitad respecto a la fase de optimización anterior. Las funciones disponibles en *libfastm* son versiones muy optimizadas de un conjunto reducido de funciones de la biblioteca matemática estándar *libm*: *sin*, *cos*, *tan*, *log* y *pow*.

Analizando con *SpeedShop* la distribución del tiempo de simulación total por subrutinas se llegó al siguiente resultado:

PROCEDIMIENTO	TIEMPO (%)
MCdynam	70 %
chargeCIC	25 %
fastm_log	1.8 %
rcarry	1.5 %
Mcoutput4	1.0 %
initiaMC	0.2 %
fastm_pow	0.1 %
resto de procedimientos	0.4 %

Tabla 4. 6 Distribución del consumo de CPU por procedimientos.

Se observa como el tiempo de simulación está dominado por el cálculo de la dinámica de las partículas en *MCdynam*() y, en segundo término, por el cálculo de la concentración espacial de carga en *chargeCIC*(), proceso que incluye el recuento de partículas por celdas de discretización del dispositivo. La subrutina *MCdynam*() incluye las subrutinas *drift*() y *scatt*(). El factor 70% se distribuye de hecho de la siguiente forma: 15% para *MCdynam*() puramente, 45% para *drift*() y 10% para *scatt*(). A la vista de estos resultados es evidente que conviene centrar los esfuerzos de optimización en estas subrutinas.

Un análisis detallado por instrucciones permitió ver cómo gran parte del tiempo de cálculo (un tercio aproximadamente) se dedicaba a ejecutar las instrucciones relacionadas con el algoritmo utilizado para determinar la celda de discretización a la que pertenece cada partícula. Éste cálculo debe realizarse en la subrutina *drift*() para determinar el campo eléctrico que actúa como fuerza motriz de la partícula y también en *chargeCIC*() para poder asignar la contribución de la carga de cada partícula a la celdas

UNIVERSITAT POLITÈCNICA DE CATALUNYA

correspondientes según el esquema de nube de carga o *cloud-in-cell* (CIC). Para cada partícula el método de Monte Carlo memoriza su coordenada espacial x. A partir de ésta y teniendo en cuenta las coordenadas de cada nodo de la discretización (almacenadas también en un vector puesto que no siguen un paso constante) debe encontrarse el nodo más cercano a la partícula en cuestión. Algorítmicamente el problema equivale a encontrar la posición que debe ocupar un nuevo elemento dentro de una lista ordenada de elementos. Un buen algoritmo para resolver el problema es el de subdivisión sucesiva del intervalo que contiene al elemento, descartando en cada paso la mitad del intervalo que queda por explorar. La complejidad del algoritmo medida por el número de pasos a realizar para ubicar cada elemento es del orden de $log_2 M$, siendo M el total de nodos de la malla de discretización.

A la vista de los datos anteriores, se buscó una solución alternativa y se decidió almacenar en un vector la celda a que pertenece cada partícula, *jcell*, como un dato más que lo caracteriza, igual que se hace con su posición x, vector de onda (k_x, k_y, k_z) o valle *iv* que ocupa. De hecho esta opción no significó incrementar la memoria necesaria puesto que la nueva variable substituyó a otra que informaba de la región del dispositivo en la que se hallaba la partícula (colector, base o emisor). Después de cada proceso *drift*() será necesario actualizar x y *jcell*. Para la actualización de *jcell* el nuevo algoritmo se basará en que el nuevo valor debe estar cercano al antiguo, puesto que así se asume y se impone cuando se elige el valor del paso temporal Δt por motivos de estabilidad general del algoritmo de MC autoconsistente con la resolución de la ecuación de Poisson.

Después de introducir en el programa todos los cambios relacionados con este nuevo enfoque se consiguió resolver el problema tipo en tan solo 55 segundos, lo que significa prácticamente dividir por dos el tiempo de CPU de la solución anterior. Efectivamente, con el nuevo algoritmo de cálculo de *jcell* el tiempo de ejecución pasaba a estar dominado por otras operaciones, en especial:

a) 16% para el cálculo de la posición de la partícula después de cada vuelo libre en la subrutina *drift*(), puesto que se trata de una operación muy repetida durante la simulación y con un costo de computación elevado: incluye el cálculo de una raíz cuadrada y de una división en coma flotante (véanse las latencias de estas operaciones en la Tabla 4. 2).

b) 10% para el cálculo de la energía cinética de la partícula a partir de su vector de onda en la subrutina *scatt*(). Es otra operación muy repetida y que también se basa en una raíz cuadrada.

c) 9% en comparación de dos variables reales en *MCdynam*() para determinar si la partícula debe continuar su dinámica o si por el contrario hay que muestrear su estado para recalcular la nueva densidad espacial de carga en el dispositivo. Operación también muy frecuente por construcción intrínseca del algoritmo MC.

d) 25% en varias operaciones de lectura de los vectores que describen el estado de cada partícula. Si bien representa una fracción importante del tiempo de ejecución su presencia se atribuye a fallos de *cache* "inevitables" dado el volumen elevado de datos a tratar y la relativamente baja capacidad de las memorias *cache*.

En resumen, llegados a este punto se habían conseguido reducciones del tiempo de cálculo en un factor 3'8 respecto a las primeras optimizaciones sobre Origin y de 34 (hasta 96) comparando con las ejecuciones sobre la estación de trabajo Sun. A partir de aquí se optó por paralelizar el código como vía para continuar reduciendo el tiempo total de la simulación.

4.5 Optimizaciones paralelas en MCHBT.

4.5.1 Visión general y objetivos.

Las optimizaciones de la versión paralela de un programa consisten en buscar un buen equilibrio de carga entre los procesadores, explotar preferentemente la granularidad gruesa antes que la fina, y minimizar la sobrecarga computacional inherente a toda paralelización.

La paralelización de aplicaciones científicas proviene en general de la existencia de bucles iterativos largos que repiten un mismo cálculo sobre un conjunto de datos disjuntos. Para que la paralelización sea posible el algoritmo debe evitar las dependencias entre datos de una y otra iteración. Con el modelo de programación de memoria compartida, que es el utilizado en el computador Origin 2000, no es necesario preocuparse, en principio, de saber dónde está ubicado cada dato. Recuérdese que la memoria principal está físicamente distribuida en forma de bloques elementales asociados a cada nodo, y que cada procesador cuenta con su propia memoria *cache* primaria y secundaria. Por tanto, puede haber copias de un mismo dato en muchas localizaciones distintas, de las cuales es necesario mantener la coherencia. En este sentido, el de la coherencia, no hace falta preocuparse por los datos puesto que el propio sistema se encarga de ello; está implícito en cualquier modelo de memoria compartida igual que también lo están las comunicaciones entre procesadores. Pero para conseguir mejores resultados sí que es conveniente repartir los datos de forma que se almacenen en bloque de memoria más próximo al procesador que vaya a hacer mayor uso de ellos.

La paralelización de un programa a partir de una versión secuencial que haya sido desarrollada con un buen estilo de programación estructurada y con un mínimo de previsión es posible introduciendo pequeñas modificaciones en el código y algunos cambios eventuales en los algoritmos o en las estructuras de datos. Para el caso de códigos "típicos" y "simples" se puede intentar la paralelización automática con la opción de compilación *–pfa (power Fortran analyzer)*. Pero frecuentemente el compilador se ve obligado a hacer suposiciones demasiado restrictivas sobre las dependencias de datos para asegurar que los resultados sean correctos en todas las situaciones posibles. En otros casos, el compilador simplemente no es una herramienta tan perfecta como uno desearía. Por consiguiente, el programador puede aportar su visión global de alto nivel del problema y orientar al compilador mediante directivas para facilitarle el camino hacia una solución eficiente (explotación del paralelismo de grano grueso). La directiva básica para la paralelización de bucles es *c$doacross*, y se explicará más adelante.

En el proceso de paralelización de una aplicación no se pueden perder de vista tres aspectos generales sobre el rendimiento: la sobrecarga por paralelización (*parallelization overhead*), el equilibrio de trabajo entre los procesadores (*load balance*) y la granularidad del código paralelizable.

La sobrecarga por paralelización es inevitable puesto que al trabajo que realizaba un solo procesador hay que añadir ahora el tiempo necesario para crear los procesos esclavos, iniciar y finalizar las regiones paralelas, sincronizar las ejecuciones en cada procesador, ejecutar código adicional específico de la paralelización, garantizar la coherencia entre distintas copias de una misma variable, etc. Hay que tener en cuenta, pues, que el tiempo de CPU global consumido será siempre mayor paralelizando que ejecutando secuencialmente el programa. Lo que se pretende es reducir el tiempo de ejecución o tiempo real de espera para la finalización de la tarea. Para reducir la sobrecarga es recomendable:

a) paralelizar los bucles más externos y más largos; continuar ejecutando secuencialmente los bucles demasiado pequeños,

b) evitar la escritura sobre variables escalares compartidas, ya que la necesidad de mantener su coherencia en todo el sistema de memoria provocaría muchos fallos de *cache* y una penalización muy importante en el tiempo total de ejecución. Esta problemática se conoce como compartición falsa o *false sharing*. En caso necesario, se tendrán que promocionar estas variables escalares a matrices, asignando una columna a cada procesador,

c) no utilizar más procesadores de los estrictamente necesarios para la aplicación.

El equilibrio de carga entre los procesadores depende de la equitatividad con que pueda hacerse el reparto del trabajo. Por una parte, siempre existirá una fracción de trabajo que por su naturaleza deberá ser ejecutada de forma secuencial. Este trabajo recae siempre sobre el mismo procesador, el *master*, que es además el encargado de coordinar al resto de procesadores, *slaves*. Por otra parte, lo importante es conseguir un buen reparto del trabajo entre todos los procesadores (*master* incluido) cada vez que se ejecuta una zona paralela. Hay que tener en cuenta que el tiempo de ejecución de una zona paralela viene determinado por el tiempo que tarde el subproceso más lento de todos los generados. Existen directivas de compilación encaminadas a controlar la planificación del trabajo (*scheduling*).

La granularidad de las zonas paralelas es importante puesto que de ella depende la relación entre el trabajo a realizar y las mejoras esperables en los resultados. Una granularidad gruesa requiere un conocimiento más profundo de la aplicación y del

código pero permite conseguir buenos resultados con pocas modificaciones de código. La granularidad fina es propia del paralelismo de cada bucle iterativo. No es necesario saber mucho sobre la aplicación que se desarrolla, pero exige paralelizar muchos lazos para obtener resultados significativos. Es la adecuada para la paralelización automática por parte del compilador.

4.5.2 Paralelización manual con directivas de compilación.

La paralelización manual de un programa mediante el uso de directivas de compilación es compatible con la versión secuencial del código fuente puesto que todas las directivas de paralelización empiezan por los caracteres *c$*, de forma que en una compilación secuencial son interpretadas como líneas de comentario. Para que el compilador interprete estas líneas y genere una versión paralelizada del código hay que compilar los ficheros con la opción *–mp*.

Una línea del fichero fuente que empiece con *c$* sirve para insertar sentencias en el código paralelo que no deben ejecutarse en la versión secuencial. Las continuaciones de línea de instrucción se marcan con la combinación *c$&*.

La directiva más simple y potente para paralelizar bucles en Origin con el modelo de programación de memoria compartida es *c$doacross*. Su sintaxis es la siguiente:

```
c$doacross
c$& share (lista de variables compartidas),
c$& local (lista de variables locales),
c$& lastlocal (lista de variables lastlocal),
c$& reduction (lista de variables con reducción),
c$& if (expresión lógica)
c$& mp_schedtype= tipo
c$& chunk= entero(>=1)
```

Esta directiva situada justo antes de un bucle del tipo DO permitirá ejecutarlo en paralelo entre todos los procesadores asignados. Todas las variables que se utilicen dentro del cuerpo del bucle deberán aparecer en una de las listas anteriores: *share* si se trata de variables compartidas entre los procesadores, *local* si se trata de variables que se pueden tratar como variables locales de cada procesador, *lastlocal* si se quiere conservar el valor de la variable en la última iteración del bucle o *reduction* para

variables escalares sobre las que se realicen operaciones de reducción dentro del bucle (suma, producto escalar, máximo o mínimo). La cláusula *if(expresión lógica)* se utiliza para poder ejecutar el bucle de forma paralela o secuencial en función del resultado de la expresión lógica. La elección de una u otra opción suele estar relacionada con la cantidad de trabajo que encierra el bucle, valorándose si habrá o no el suficiente trabajo como para compensar la sobrecarga de la paralelización. En Origin 2000 la cantidad de trabajo mínima (*quantum work*) para amortizar la sobrecarga de paralelización es del orden de unos 400 ciclos de CPU o, dicho de otro modo, unas 100 operaciones en coma flotante por procesador.

La cláusula *mp_schedtype* permite seleccionar el tipo de planificación a aplicar para repartir las iteraciones del bucle entre los procesadores disponibles. Los tipos de planificación disponibles son: *simple*, *interleave*, *dynamic* y *gss*. Las dos primeras son estáticas, de forma que al iniciarse la ejecución paralela del bucle cada procesador sabe perfectamente qué iteraciones le han correspondido (cuántas y cuáles). Las dos últimas son dinámicas, pues los procesadores se autoasignan una nueva porción de trabajo cada vez que completan la porción que tenían asignada, para lo cual tienen que entrar en una *zona crítica* (acceso simultáneo limitado a un solo procesador). Las planificaciones estáticas tienen como ventaja un coste de planificación menor, pero cuando la carga de trabajo por iteración no es uniforme provocan más desequilibrios de carga entre los procesadores.

La planificación simple divide el número total de iteraciones del bucle en tantas partes iguales como procesadores haya. Cada parte corresponde a un grupo de iteraciones consecutivas. La planificación intercalada también divide el número total de iteraciones del bucle en partes iguales de acuerdo con el número de procesadores, sólo que las iteraciones asignadas a cada procesador no son correlativas sino que son asignadas alternativamente entre los procesadores por bloques del tamaño especificado por la cláusula *chunk*. Este tipo de planificación es adecuado para aquellos bucles en los que la carga de trabajo crece (o decrece) monótonamente en cada iteración.

Para aquellos problemas en los que las planificaciones estáticas generan un excesivo desequilibrio de carga se puede recurrir a las planificaciones dinámicas. Con planificación dinámica pura (*dynamic, chunk=1*) cada procesador se autoasigna una iteración del bucle; cuando finaliza la ejecución de la misma vuelve a ejecutar la

subrutina especial que le permite localizar y autoasignarse la siguiente iteración libre del bucle. La planificación *dynamic* con *chunk=k* se autoasigna *k* iteraciones consecutivas, con lo que se pretende disminuir el coste de planificación (menor número de llamadas a las subrutinas de autoasignación para completar el bucle) sin perjudicar significativamente el desequilibrio de carga entre procesadores. La realidad es que, en el caso particular de Origin, esto no es así debido a una implementación muy eficiente (una sola instrucción de lenguaje máquina) de la operación de autoasignación, que simplemente se repite el número de veces especificado por la cláusula *chunk*.

La planificación *gss* (*guided self-scheduling*) realiza una planificación dinámica asignando un número de iteraciones variable en función del número de iteraciones pendiente. Asignando grupos relativamente grandes de iteraciones al principio y relativamente pequeños hacia el final se espera conseguir un buen balance de carga a un coste moderado de planificación por reducción del número de entradas en la zona crítica.

La elección del tipo de planificación idóneo puede ser el resultado de un proceso de prueba y error. La valoración se puede llevar a cabo con las mismas herramientas de optimización que se describieron para el caso secuencial, sólo que ahora hay que generar un fichero de datos para cada proceso (*thread*) y después realizar un análisis global de los resultados.

Otro aspecto fundamental a valorar en las ejecuciones paralelas que está muy relacionado con el tipo de planificación elegido es el de los accesos a los datos, en especial a los que residen en la *cache*. Las planificaciones con *chunk =1* o reducido son más propensas a generar fallos de *cache* por la aparición del fenómeno de compartición falsa (*false sharing*). Cada vez que un procesador cualquiera escribe sobre un dato, el sistema encargado de asegurar la coherencia entre datos marca toda la línea de *cache* con la etiqueta de "modificado", por lo que cualquier procesador que tenga en su propia *cache* una copia de esa misma línea de datos deberá actualizarlos antes de utilizarlos de nuevo. Esto provoca la invalidación de toda la línea en todos los procesadores que se encuentren en esa situación, lo que puede llegar a afectar muy gravemente la eficiencia de los cálculos realizados. De hecho, una forma de poner de manifiesto la compartición falsa es observando los tiempos de CPU de cada procesador; cuando ocurre la compartición falsa la suma de tiempos de CPU de los procesos en paralelo puede llegar

a superar netamente el tiempo de CPU requerido por la ejecución secuencial del problema sobre un solo procesador. La solución en estos casos es cambiar de tipo de planificación o adaptar las estructuras de datos con los accesos dictados por el algoritmo de forma que a cada procesador le corresponda manejar los datos de toda la línea de *cache* en exclusiva.

En el mejor de los casos, la paralelización de bucles anidados se puede realizar sin problemas si el bucle más externo es paralelizable (reducción del *overhead* de paralelización) y el acceso a los datos es en el orden adecuado, por columnas (buen uso de las líneas de *cache*). En otros casos se tienen que buscar soluciones de compromiso. Considérese el siguiente ejemplo:

```
do j=1,N
   do i=1,M
      a(i) = a(i) + b(j) * c(i,j)
```

Gráficamente podemos esquematizar los cálculos según la Figura 4. 3 adjunta.

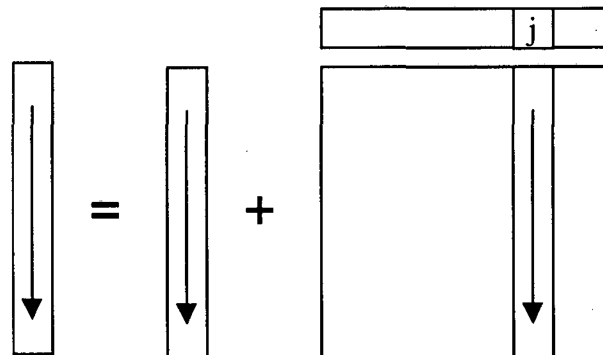

Figura 4. 3 Diagrama esquemático del algoritmo secuencial.

El bucle en *i* es paralelizable pero el bucle en *j* no lo es. Si se aplica la técnica de intercambio de bucles convirtiendo el bucle en *i* en bucle exterior se disminuye el *overhead* puesto que aumenta el *quantum work*, pero el acceso a los datos *c(i,j)* empeora. Dejar los bucles en el orden original y paralelizar tan solo el bucle interno puede ser aceptable si *M* es suficientemente grande (*quantum work*):

```
do j=1,N
c$doacross ...
   do i=1,M
      a(i) = a(i) + b(j) * c(i,j)
```

Otra solución es aplicar un proceso de reducción con los *k* procesadores disponibles. El precio a pagar es en aumento de memoria (matriz auxiliar *aux(i,k)*) y complicación del código (sobre todo porque el compilador no es capaz de generar automáticamente el código necesario para la reducción sobre un elemento que no sea de tipo escalar). El seudocódigo de la reducción sería el siguiente:

```
c$doacross ...
 do k
    do j
       do i
          aux(i,k) = aux(i,k) + b(j) * c(i,j)
       enddo
    enddo
 enddo

 do k
    do i
       a(i) = a(i) + aux(i,k)
```

De forma gráfica, el esquema indica el aumento de memoria necesario por la promoción del vector a matriz:

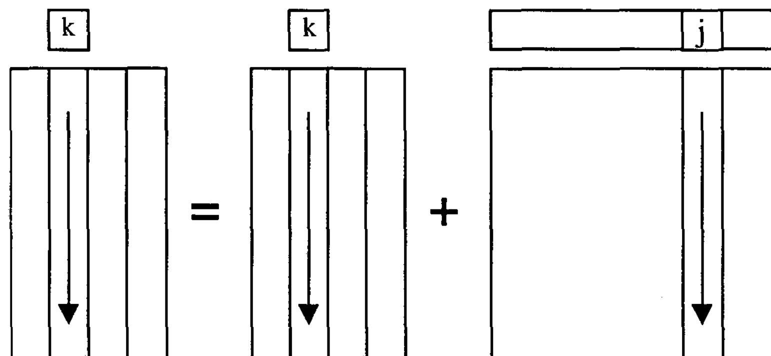

Figura 4. 4 Diagrama esquemático del algoritmo paralelizado.

A parte de distribuir las iteraciones de forma adecuada entre los procesadores (planificación del bucle) también es importante distribuir los datos de forma que se almacenen en el bloque de memoria RAM más cercano al procesador que vaya a utilizarlos más frecuentemente (planificación del datos). Para conseguirlo existen otras directivas de compilación como *c$distribute*, *c$distribute_reshape* o la cláusula *affinity* que puede complementar a la directiva *c$doacross*. La directiva *c$distribute* distribuye

los elementos de un vector o una matriz asignándolos a memorias de forma similar a cómo *mp_schedtype* reparte las iteraciones de un bucle entre los procesadores. En el contexto de datos la opción *simple* se denomina *BLOCK*, mientras que la opción *interleaved, chunk k* pasa a denominarse *CYCLIC(k)*. La directiva *c$distribute_reshape* es equivalente a la anterior pero reajustando las páginas de memoria para que cada dato quede almacenado donde realmente se desea. La cláusula *affinity* permite aparejar de forma coherente el subconjunto de datos con el subconjunto de iteraciones que se asignan a un procesador.

El programa paralelizado puede ser ejecutado por cualquier número de procesadores sin necesidad de ninguna modificación interna del código. El número de procesadores a utilizar en cada caso se puede fijar de varias maneras, por ejemplo desde la línea de comandos del sistema operativo asignando el valor deseado a la variable de entorno *MP_SET_NUMTHREADS* mediante el comando *setenv*. Durante la ejecución del programa puede ser necesario tener conocimiento internamente del número de procesadores que están en uso dentro de una zona paralela o que cada procesador sepa qué número tiene asignado. Estas informaciones y otras similares son accesibles a los procesadores mediante llamadas a funciones como *mp_numthreads*() o *mp_my_threadnum*(). Este tipo de llamadas se insertan como cualquier sentencia más del lenguaje de programación teniendo la precaución de encabezar con los caracteres *c$* la línea del fichero fuente en que aparezcan.

4.5.3 Paralelización de MCHBT.

La paralelización del simulador MCHBT se ha llevado a cabo siguiendo el modelo de programación de memoria compartida y con la inserción de directivas de compilación como las acabadas de presentar. Tal como se vio en el estudio realizado para la versión secuencial del simulador el 95% del tiempo de cálculo estaba concentrado en la subrutina *MCdynam*() –que incluye las subrutinas *drift*() y *scatt*()– y la subrutina *chargeCIC*(). Por este motivo, los esfuerzos de paralelización deberían ir dirigidos a estas subrutinas. A priori, las expectativas de paralelización son buenas puesto que gran parte del código de estas subrutinas consiste en la ejecución de un bucle iterativo del tipo *DO n=1,npt* que realiza cálculos equivalentes sobre cada una de las partículas del

dispositivo. El número total de partículas, *npt*, no es constante a lo largo de la simulación sino que va cambiando y se actualiza (precisamente en *chargeCIC*()) a cada ciclo de paso temporal Δt, antes de resolver la ecuación de Poisson. Así pues, justo antes de entrar en estos bucles el número de partículas está bien definido y no cambia durante su ejecución.

Para la paralelización de *MCdynam*() se puede optar en principio por una planificación estática simple de las iteraciones, repartiendo las *npt* iteraciones a partes iguales entre todos los procesadores. Con esta elección se vislumbran dos problemas: la planificación de datos y el equilibrio de carga. La planificación de datos con las directivas *c$distribute*, y *c$distribute_reshape* se efectúa al principio del programa, después de las sentencias de declaración de variables y justo antes de la primera instrucción ejecutable del programa. La planificación de datos se hace, pues, no sobre el valor instantáneo de *npt* sino sobre su valor máximo posible, *nptmax*, de acuerdo con la reserva de espacio de memoria efectuada en el momento de la declaración de la estructura de datos. Desde este punto de vista, una planificación simple del trabajo (*simple*) y de los datos (*BLOCK*) no resulta adecuada puesto que no conseguiría la concordancia perseguida entre iteraciones y datos para un procesador y su bloque de memoria principal más cercano. Por otra parte, la planificación simple del trabajo tampoco resulta adecuada puesto que el coste por iteración no sólo no es constante sino que es imprevisible. La razón radica en la misma base del algoritmo MC: el número de vuelos libres y choques (subrutinas *drift*() y *scatt*()) que realiza una partícula en el intervalo de muestreo Δt es aleatorio. Aún así, puesto que el número de partículas por procesador es elevado, el efecto de promediado debería conseguir un equilibrio de carga bastante uniforme. Una planificación dinámica pura tendría dos inconvenientes graves: el *overhead* de planificación y la compartición falsa. El primer inconveniente resulta obvio. El segundo se comprende si observamos la estructura de las dos matrices que almacenan el estado de cada partícula: *ipt(2,nptmax)* y *pt(5,nptmax)*. La primera es de tipo entero y la segunda de tipo real de doble precisión. Las dos matrices deben ser leídas y escritas constantemente. Fijándonos en la matriz *pt*, vemos como el estado de cada partícula requiere 5 variables de doble precisión. En una línea de la *cache* secundaria de Origin caben 16 variables de este tipo. Por tanto, con planificación dinámica pura, cada vez que un procesador estuviera actualizando una cualquiera de las cinco variables de estado de la partícula en curso estaría "invalidando" los datos de los tres procesadores que

estuvieran utilizando los datos de las tres partículas que caben (total o parcialmente) en la misma línea de *cache*. Las consecuencias serían desastrosas. Un primer remedio podría ser redefinir las estructuras de datos de forma que cada partícula ocupara toda una línea de *cache* secundaria: *ipt(32,nptmax)* y *pt(16,nptmax)*. Esto arreglaría la compartición falsa de datos pero al precio de gastar inútilmente casi 50 veces más de memoria RAM. Otro remedio sería utilizar un parámetro *CHUNK* mayor que la unidad, pero sabemos que en Origin esto agrava todavía más la sobrecarga de la planificación dinámica. En definitiva, parece que la mejor opción para paralelizar este bucle es una planificación estática del tipo *interleaved* (con poca sobrecarga de planificación) escogiendo un parámetro *CHUNK* = *LB* relativamente bajo para evitar desequilibrios de carga y de valor adecuado para evitar problemas de compartición falsa. Para ello bastará con tomar bloques de *LB* partículas que hagan que el número de octetos de datos asociados de las matrices *ipt* y *pt* sean un múltiplo cualquiera de la capacidad de una línea de *cache* secundaria:

L2 = 128 bytes

dato real_dp (doble precisión) = 8 bytes

dato real_sp (precisión simple) = 4 bytes

dato entero = 4 bytes

Condición para *pt*: 5 eltos.* 8 bytes *LB = 128 bytes * n , n=1,2,3...

(4. 10)

Condición para *ipt*: 2 eltos.* 4 bytes *LB = 128 bytes * m, m=1,2,3...

(4. 11)

La solución mínima es *LB = 16* (partículas o iteraciones del bucle). Con *CHUNK* = *LB* = *16*, cada procesador tendría asignados bloques intercalados de 16 iteraciones y se ocuparían cinco líneas exactas (*n=5*) de *cache* secundaria para los datos de *pt* y una línea más (*m=1*) para los de *ipt*.

Si además añadimos la directiva *c$distribute_reshape pt (*,CYCLIC(LB)), ipt (*,CYCLIC(LB))* conseguiremos una buena planificación de datos.

Para poder paralelizar el bucle principal de *MCdynam()* nos queda todavía un problema importante por resolver: la generación de números aleatorios. Tal como se vio en el capítulo tercero, los números aleatorios en un ordenador se obtienen generando una *secuencia* pseudoaleatoria a partir de una o varias semillas. Se trata, pues, de un proceso puramente secuencial que choca de frente con nuestro objetivo de paralelizar el trabajo: las iteraciones del bucle se pueden repartir fácilmente entre los procesadores, pero repartir la secuencia de números aleatorios no es tan evidente puesto que a priori no podemos saber cuántos números aleatorios requerirá la simulación de la dinámica de cada partícula. Además, dedicar un procesador a gestionar la secuencia aleatoria y a transferir al resto de procesadores los números aleatorios que solicitaran sería ineficiente por el coste de las comunicaciones entre procesadores a través de la red de interconexión y las colas de espera que se generarían. La solución puede estar en que cada procesador genere su propia secuencia, pero para ello hay que asegurar dos detalles cruciales: que las secuencias sean disjuntas y que tengan continuidad temporal durante toda la simulación. El primer punto invalida muchos generadores. Por éste y otros motivos expuestos en el capítulo 3 de esta memoria, nosotros hemos escogido el método ACARRY, implementado por James como subrutina RCARRY() [James,1990]. Descuidar un detalle como éste podría introducir fuertes correlaciones numéricas entre dinámicas supuestamente independientes de partículas, lo que invalidaría los resultados físicos extraídos de la simulación. El segundo aspecto surge por el hecho de que las zonas paralelas sólo se crean cuando hay una parte de código por paralelizar, por lo que los procesadores esclavos son liberados cada vez que se finaliza una de estas partes. Como quiera que la calidad de una buena secuencia aleatoria está relacionada con su longitud máxima y que el proceso de inicializarlas es costoso y de posibilidades relativamente limitadas (si queremos asegurar que cada secuencia nueva sea disjunta con cualquier secuencia anterior), será conveniente crear e inicializar al principio de la simulación tantas secuencias como procesadores vayan a tomar parte en la misma y a partir de aquí mantener viva cada secuencia en una estructura de datos general del programa. Cuando se ejecute una zona paralela, cada procesador deberá acceder a la partición que corresponda a su secuencia aleatoria para poder continuarla en la medida que necesite consumir más números aleatorios.

En el simulador MCHBT la dinámica de una partícula, entendida como el binomio formado por el encadenamiento de una ejecución de *drift*() con otra de *scatt*(), consume

Lrns=4 números aleatorios: uno para calcular la duración del movimiento acelerado y tres para determinar el mecanismo responsable de la colisión y el estado final de la partícula después del choque. Para amortizar la llamada a la subrutina de generación de números aleatorios es mejor generar un bloque *Lbck* relativamente grande de números a la vez. Así, para almacenar estas secuencias aleatorias para cada procesador se ha definido una matriz de reales simples (32 bits) declarada de la siguiente manera:

```
real  rns (0 : 24 + Lrns * Lbck + D , nthdsmax)
c$distribute_reshape rns (*,CYCLIC(1))
```

La matriz es bidimensional. Cada columna pertenece a un procesador distinto y contiene una fracción de la secuencia propia más todos los datos necesarios para continuarla. Los primeros 25 elementos de la matriz son precisamente esos datos: el bit de *carry* y los últimos 24 números de la secuencia, que sirven a la vez de semilla para continuarla. A continuación viene la secuencia propiamente dicha (con capacidad para *Lrns * Lbck* números aleatorios). Con esta primera modificación respecto al algoritmo original propuesto por James se elimina el vector de índices y todas las operaciones de manipulación de la tabla cíclica que conlleva, lo cual representa una considerable mejora de tiempo de cálculo añadida al hecho de permitir la generación de números por bloques. El parámetro *D* es de ajuste para que cada columna ocupe un número entero de líneas de *cache* secundaria y se eviten problemas de compartición falsa de datos. Se deberá cumplir, por tanto que:

$$(1 + 24 + Lrns * Lbck + D) * 4\ bytes = L2 * n = 128\ bytes * n \qquad n = 1,2,3\ldots \tag{4.12}$$

Una solución de compromiso entre disminuir *overheads* en las llamadas a la subrutina de generación ($Lbck>1$) y evitar sobreescribir demasiadas posiciones de la *cache* secundaria a cada llamada ($Lbck<1000$) puede ser $Lbck=105$, $n=14$ y $D=3$. Para controlar el consumo de datos y avisar sobre la necesidad de generar nuevos datos que continúen la secuencia, el programa utiliza un contador de uso exclusivo para cada proceso. De nuevo, para evitar la compartición falsa y el consiguiente *trashing* los contadores se tienen que estructurar como una matriz:

```
integer  La (L2i, nthdsmax)
```

donde *L2i=32* es el número de datos enteros que caben en una línea de *cache* secundaria. Aunque sólo desearíamos un dato por columna, la compartición falsa obliga a malgastar 31 posiciones más por columna. Por fortuna, en términos absolutos representa poco derroche de memoria.

Para que los procesadores puedan acceder de forma selectiva a sus propios datos cada procesador se autoidentifica dentro de la zona paralela con una llamada a la función *mp_my_threadnum*() que le devuelve un entero entre 0 y *(nthds-1)*, donde *nthds* es el número total de procesadores en uso.

Volviendo a retomar las simulaciones del problema tipo, con la ejecución en paralelo de *MCdynam*() sobre 2, 4 y 8 procesadores se obtuvieron los resultados reflejados en la Tabla 4. 7.

$p = nthds$	T_r	$S(p)$	$E(p)$
1	60 s	1.00	1.00
2	45 s	1.33	0.66
4	28 s	2.14	0.53
8	28 s	2.14	0.26

Tabla 4. 7 Datos de la simulación con paralelización de *MCdynam*().

Se observa como efectivamente el tiempo de ejecución T_r baja hasta la mitad del tiempo empleado por la ejecución secuencial, pero con una ganancia muy moderada: con 4 procesadores apenas se va el doble de rápido; y añadiendo más procesadores no se obtiene ya más ganancia. La explicación nos la puede dar la ley de Amhdal: si la fracción de código paralelizada no es del orden del 85% o superior, no se obtienen ganancias apreciables por el hecho de incrementar el número de procesadores (véase la Figura 4. 2).

Para obtener resultados más esperanzadores se continuó el proceso de paralelización con la subrutina *chargeCIC*(). En principio se trataba de paralelizar un bucle del tipo *DO n=1, npt* que explora todas las partículas una tras otra como en el caso anterior. Pero la problemática de este caso era distinta puesto que aquí todos los procesadores

comparten una matriz, *cn(j), j=1,M*, sobre la que se almacena la densidad de carga hallada en cada celda de la discretización. Se trata de una operación de reducción sobre una matriz, por tanto no paralelizable automáticamente. La paralelización manual de este tipo de operaciones es algo aparatosa y engorrosa, ya que se necesitan matrices y código auxiliares que oscurecen el algoritmo secuencial.

El núcleo central de la subrutina secuencial se puede resumir de la siguiente forma:

```
do n=1,npt
   x=pt(ix,n)
   jp=ipt(icel,n)
   if (x .gt. xM1) then
      je=je+1
      jpte(je)=n
   else if (x .lt.x2) then
      jc=jc+1
      jptc(jc)=n
   endif

   xj = par(ixm,jp)
   xj1 = par(ixm,jp-1)
   cnj = (x-xj1) / (xj-xj1)
   cn(j) = cn(j) +cnj
   cn(j-1) = cn(j-1) + (1.-cnj)
enddo
```

En el bucle se van inspeccionando las partículas una a una. Por una parte, para la posterior aplicación de las condiciones de contorno en los contactos de emisor y colector, se detectan las partículas que caen dentro de las celdas adyacentes a dichos contactos. Por otra parte, en función de la coordenada espacial de cada partícula, se calcula la contribución a la densidad de carga de los dos nodos que delimitan su posición (método de la nube de carga o *cloud-in-cell, CIC*), que se va acumulando en el vector *cn(j)*.

Para poder paralelizar el bucle con esta operación de reducción sobre el vector *cn(j)* sin caer en el problema de compartición falsa, se han promocionado algunos vectores a matrices con lo que el bucle paralelizado se ha convertido en el siguiente:

```
  kmax=1+(npt/(nthds*LB))

  do 100 k=1,kmax
c$doacross local(ntd,n,x,jp,xj,xj1,cnj),
c$& shared(nthds,LB,k,pt,ipt,xM1,x2,
```

```
c$& jptcaux,jpteaux,par,cnaux),
c$& mp_schedtype = simple,
c$& affinity(ntd) = thread ( ntd-1 )
   do 110 ntd=1,nthds
    do 120 n=1+LB*(ntd-1+(k-1)*nthds),
                 min.(npt,LB*(ntd+(k-1)*nthds))
       x  = pt(ix,n)
       jp = ipt(icel,n)

       if (x.gt.xM1) then
        jpteaux(0,ntd)=jpteaux(0,ntd)+1
        jpteaux(jpteaux(0,ntd),ntd)=n
       else if (x.lt.x2) then
        jptcaux(0,ntd)=jptcaux(0,ntd)+1
        jptcaux(jptcaux(0,ntd),ntd)=n
       endif

       xj     = par(ixm,jp)
       xj1    = par(ixm,jp-1)
       cnj    = (x-xj1)/(xj-xj1)

       cnaux(jp-1,ntd)    = cnaux(jp-1,ntd) +
                                       (1.-cnj)
       cnaux(jp,ntd)  = cnaux(jp,ntd) + cnj

120    continue
110   continue
100  continue

  do k=1,nthds
   do j=1,M
     cn(j) = cn(j) + cnaux(j,k)
   enddo
   ji=jc+1
   jf=jc+jptcaux(0,k)
   jx=0
   do j=ji,jf
     jx=jx+1
     jptc(j)=jptcaux(jx,k)
   enddo
   jc=jc+jptcaux(0,k)
   ji=je+1
   jf=je+jpteaux(0,k)
   jx=0
   do j=ji,jf
     jx=jx+1
     jpte(j)=jpteaux(jx,k)
   enddo
   je=je+jpteaux(0,k)
  enddo
```

Se puede observar como los procesadores se van repartiendo de forma intercalada bloques de *LB* iteraciones hasta que se han completado todas. Con este algoritmo para *chargeCIC()* y la paralelización vista anteriormente para *MCdynam()*, las simulaciones del problema tipo sobre 2, 4 y 8 procesadores dieron los resultados reflejados en la Tabla 4. 8:

p = nthds	T_r	*S(p)*	*E(p)*
1	61 s	1.00	1.00
2	35 s	1.74	0.87
4	18 s	3.40	0.84
8	10 s	6.10	0.76

Tabla 4. 8 Datos de la simulación con paralelización de *MCdynam()* y *chargeCIC()*.

Analizando los resultados vemos como la estimación del porcentaje de código paralelizado gira en torno del 90%, lo cual permite reducir progresivamente el tiempo de ejecución a medida que se añaden más procesadores. Ahora, con 8 procesadores la eficiencia de la paralelización se mantiene todavía a un valor relativamente alto (76%) y un *speedup* mayor que 6, que rebaja el tiempo de simulación desde 61s a tan solo 10s. La Figura 4. 5 representa en forma de histograma los tiempos de ejecución por procesador en una ejecución con 4 procesadores. Se distingue entre el tiempo útil de cálculo, el de sobrecarga de la paralelización y el de inactividad. La parte secuencial del programa es ejecutada por el procesador P_0 (el *master*), lo que explica el desequilibrio de carga entre éste y el resto de procesadores. Mientras P_0 ejecuta la parte secuencial del programa, los demás procesadores están inactivos. Entre los procesadores esclavos se observa un buen equilibrio de carga. El *overhead* de la paralelización viene dado por la ejecución de las subrutinas de sincronización *wait_for_completion()* en el caso de P_0 o *slave_wait_for_work()* en el resto, y por las llamadas a *mp_my_threadnum()* realizadas por todos los procesadores dentro del bucle *MCdynam()*. Si fuera posible sacar fuera del lazo *DO* las llamadas a esta función se conseguirían mejoras notables en este aspecto (reducción del *overhead* en un 30% aproximadamente). La suma de tiempos útiles de todos los procesadores da un valor muy próximo al tiempo de ejecución real con un solo

procesador, lo que nos indica que la implementación adoptada evita efectivamente la compartición falsa de datos.

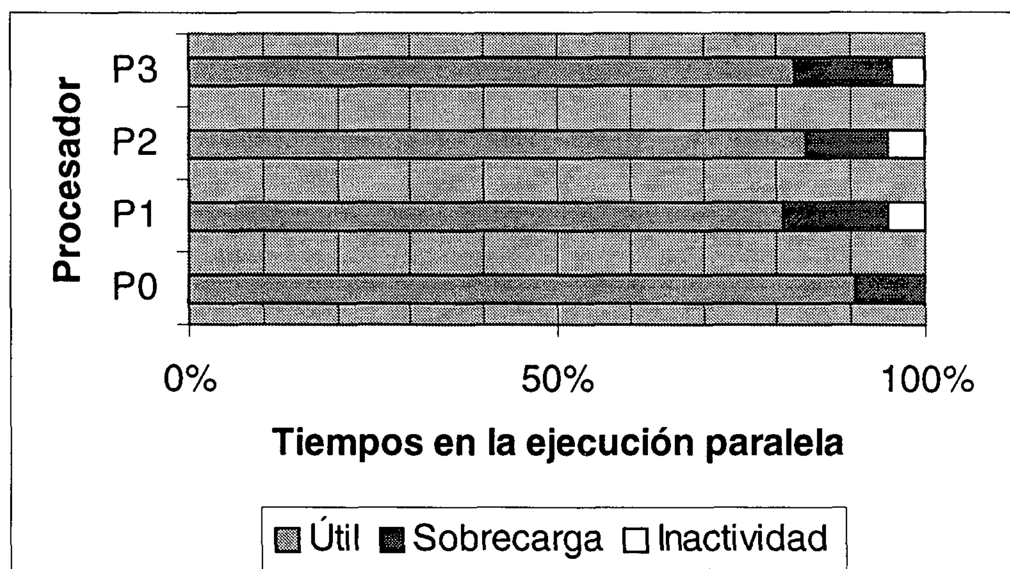

Figura 4. 5 Histograma de los diferentes tiempos en una ejecución paralela: a) tiempo útil, b) tiempo de sobrecarga (*overhead*) y c) tiempo de inactividad.

A continuación se presentan tiempos de ejecución correspondientes a simulaciones más realistas utilizando 8 procesadores en paralelo. En concreto, se repitió la simulación de 5ps de un transistor BJT de GaAs con unas 980000 partículas vista en el apartado de optimizaciones secuenciales. Esta simulación, que tardó 12 días en la estación de trabajo Sun, se pudo repetir con los 8 procesadores en tan solo 32 minutos y 8 segundos. La ganancia de tiempo conseguida con todas las optimizaciones presentadas a lo largo de este capítulo ha sido pues de (280h/0.5h) = 560, lo que da una muestra de los beneficios obtenidos con el proceso de mejora del simulador.

Posteriormente, cuando el desarrollo del simulador estaba ya en un estadio más avanzado (se calculaban más tipos de resultado) se vio como el tiempo de ejecución se pudo reducir de nuevo en un factor 4 evitando el *overhead* de las llamadas a la función *mp_my_threadnum()* dentro de la rutina *MCdynam()* cambiando la estructura del algoritmo por otra equivalente a la del bucle principal de *chargeCIC()*. El factor de reducción se debió al efecto combinado de este cambio y la permutación de los bucles imbricados de estas subrutinas. En concreto, el bucle original:

```
      kmax=1+(npt/(nthds*LB))

      do 100 k=1,kmax
c$doacross ...
       do 110 ntd=1,nthds
        do 120 n=1+LB*(ntd-1+(k-1)*nthds),
                               min.(npt,LB*(ntd+(k-
1)*nthds))
```

se cambió por otro algorítmicamente equivalente:

```
      kmax=npt/(nthds*LB)

c$doacross ...
      do 110 ntd=1,nthds
       do 100 k=0,kmax
        do 120 n=1+LB*(ntd-1+k*nthds),
                               min.(npt,LB*(ntd+k*nthds))
```

El bucle paralelizado es siempre el mismo, incluso con la misma distribución de iteraciones y datos por procesador. El beneficio viene de la reducción por el tiempo de sincronización y creación/eliminación de zonas paralelas. En el primer caso este proceso ocurre *kmax* veces, mientras que en el segundo caso esto sólo ocurre una vez.

Con este último cambio el tiempo de simulación de un BJT (siguiendo la dinámica de unas 185000 partículas durante 5ps para alcanzar el régimen estacionario y 3ps adicionales para extraer datos del régimen permanente con un paso temporal de 0.1ps) bajó de 6h:43' a 1h:39'. Con esta aceleración por un factor 4, la ganancia de tiempo de simulación acumulada desde las primeras simulaciones suponen un factor total de 560*4=2240.

4.6 El método de MC ponderado (WMC).

Un problema importante en la simulación MC de dispositivos es que aparecen zonas con niveles muy diferentes de concentración, con lo cual el número total de partículas simuladas en el método EMC debe ser extraordinariamente grande para asegurar que en las celdas de baja concentración del dispositivo haya un número suficiente de partículas

para obtener resultados precisos o representativos estadísticamente. El método de MC ponderado o *weighted* MC (WMC) propone asignar diferentes pesos a las superpartículas como vía de solución a este problema.

La idea original del método WMC se debe a Phillips y Price [Phillips,1977] que la introdujeron como una técnica de reducción de varianza en el estudio de la función de distribución para valores de energía elevados del electrón (zonas poco frecuentadas de la función de distribución: '*energy tails*' o '*rare events*' en general). En el seguimiento de la dinámica de la partícula (según el algoritmo SPMC), definen dos posibles regiones según el estado anterior al momento de choque o dispersión de la partícula: la región C o zona común y la región R o zona raramente frecuentada por la partícula. Debido a la escasa ocurrencia de sucesos dentro de la región R, la aplicación estándar del algoritmo SPMC conduce a una varianza muy elevada en la determinación de la función de distribución para los estados k asociados a esta región. La propuesta de los autores consiste en aplicar un procedimiento de ponderación de los estados que mejore la estadística de los estados improbables sin afectar a la versatilidad del algoritmo MC. Para ello introducen un factor multiplicativo M que se aplica a la partícula cada vez que consigue alcanzar un estado de la región R. Se siguen M dinámicas independientes con el mismo estado inicial con que la partícula penetró en la región R, cada una con un peso relativo $W=1/M$ para no alterar la estadística global del proceso, hasta que todas las dinámicas regresan de nuevo a la zona C. En este punto, se escoge uno de los estados finales de salida de la región R para ser asignado a la partícula, que continuará su dinámica habitual con peso $W=1$. Asimismo, los autores hacen notar que la idea es ampliable para diferentes niveles jerárquicos de factores multiplicativos (aplicación recursiva del factor multiplicativo) según las necesidades de mejora del proceso estadístico.

Basándose en esta idea original han ido apareciendo variaciones para aplicar el método a estados poco frecuentes tanto de espaciales como energéticos junto con el método EMC [Fischetti,1988], [Sangiorgi,1988], [Venturi,1989], [Laux,1991]. En [Venturi,1989] se presenta un perfeccionamiento del procedimiento presentado por los mismos autores en [Sangiorgi,1988] que les permite simular de forma más adecuada las fuertes variaciones de densidad espacial y energética de los portadores observadas en el volumen de un MOSFET. Para las colas en energía de la función de distribución los autores utilizan el factor multiplicativo de aplicación recursiva propuesto por Phillips y

Price. Para el caso espacial, asignan un peso W a cada partícula y un peso W_i a cada celda de la discretización. Las partículas sufren procesos de fisión o fusión dentro de cada celda para intentar mantener un número razonable de partículas con peso aproximadamente igual al de la celda que ocupan en cada instante.

También recordando la técnica de Phillips y Price para el problema particular de reducción de varianza, Jacoboni, Poli y Rota desarrollan un estudio teórico completo del método WMC, refinando la nomenclatura de acuerdo con diferentes enfoques del problema [Poli,1989], [Jacoboni,1989], [Jacoboni,1988]. Derivan el método WMC mediante una expansión iterativa de la ecuación de transporte de Boltzmann en forma integral. Entienden el método como la simulación de la dinámica del electrón utilizando probabilidades arbitrarias para los diversos sucesos responsables de las trayectorias de los portadores (duración del tiempo de vuelo y probabilidad de ocurrencia de cada mecanismo de dispersión). Se trata de una versión mucho más flexible que el algoritmo estándar EMC, que se convierte en un caso muy particular del WMC. En el algoritmo EMC los electrones se inicializan con un estado inicial $\boldsymbol{k_0}$ y evolucionan hasta un estado final $\boldsymbol{k}$ de forma que los electrones tienden a acumularse en los estados donde la función de distribución presenta valores más elevados. Sólo en estos estados se obtendrá buena precisión en el cálculo de la función de distribución para tiempos de simulación realistas y razonables. El método WMC, que ellos utilizan para ilustrar el cálculo eficiente de las colas de la función de distribución, se puede aplicar focalizando el cálculo sobre aspecto particular de interés. Para ello se manipulan las probabilidades que gobiernan la dinámica del electrón para escoger el estado inicial más conveniente para la simulación y, a continuación, para guiar el electrón hacia la región del espacio k donde se desee hacer el estudio. Presentan un estudio válido para sistemas homogéneos con campo aplicado constante pero generalizable a fenómenos con dependencia temporal o espacial. Finalmente, en [Poli,1989] deciden llamar BMC (*backward* MC) a la versión del algoritmo inicialmente presentada en [Jacoboni,1988] con nomenclatura ambigua entre WMC y BMC. En el algoritmo BMC todo el cálculo se dedica a la evaluación de la función de distribución en un valor concreto de $\boldsymbol{k}$. La simulación se inicia en este estado y retrocede en el tiempo hasta un estado "inicial" $\boldsymbol{k_0}$ que cae a menudo en una región donde la función de distribución es pequeña, por lo que contribuye escasamente sobre el valor buscado $f(\boldsymbol{k})$. Por tanto, el método BMC no resuelve completamente el problema de la evaluación de las colas de la función de distribución.

Recientemente [Canali,1996] hablan de un nuevo procedimiento WMC de propósito general para ser aplicado en su estudio de la ionización por impacto en HBTs. En estos estudios identifican dos problemas: la importancia de conocer bien las colas en la función de distribución y el rápido crecimiento en el número de partículas simuladas cerca de la zona de ruptura del dispositivo. Definen una técnica multiplicativa especial tanto en espacio real como en energía extendiendo la idea original de [Phillips,1977]. En su técnica WMC definen un peso estadístico $W(\mathbf{r},E_k)$ asociado a cada partícula. El procedimiento permite tratar dispositivos con regiones de nivel de impurezas muy dispar y obtener buenos resultados estadísticos de procesos poco frecuentes manteniendo acotado el número total de partículas a simular.

En nuestro simulador MCHBT también hemos incluido una variable de peso asociada a cada partícula, para permitir limitar el número total de partículas a simular sin perder precisión en ninguna región de la discretización del dispositivo y para tratar las colas de energía. La opción de la energía de momento está desactivada, por lo que nos centraremos en el tratamiento de las variaciones espaciales. El algoritmo funciona manteniendo el número de partículas en cada celda de la discretización entre un número mínimo y máximo preestablecidos y fácilmente modificable (e.g. npc=500...600 o npc=1000...1200 partículas por celda). Para ello, después de finalizar la dinámica de todas las partículas a lo largo de cada intervalo Δt, mientras se realiza el recuento de partículas presentes en cada celda dentro de la subrutina *chargeCIC()*se van generando unas listas con la distribución de partículas por pesos en cada celda y actualizando una matriz de puntero a cada partícula para identificar las partículas que recaen en una misma celda con un mismo peso. Finalizado este proceso, que se ejecuta de forma paralela, se buscan posibles celdas con déficit o exceso de partículas de acuerdo con los valores prefijados *npcmin* y *npcmax*. En caso necesario se agrupan o subdividen partículas de forma que se mantenga la carga total y su centro de masas (para no alterar la densidad de carga espacial recién calculada por el método CIC). Para simplificar el algoritmo de clasificación y fusión de partículas, y para facilitar una implementación más eficiente del mismo (en términos de tiempo de cálculo y requerimientos de memoria para confeccionar las listas citadas) se ha establecido que los pesos de las partículas sean potencias enteras de una cierta base: r^{iw} (actualmente, $r=10$, aunque se trata también de un parámetro fácilmente modificable). Esta elección no supone en principio ninguna limitación para las posibilidades del algoritmo MC. Más bien al

contrario, permite agrupar rápidamente las partículas que necesiten ser refundidas para limitar el exceso de partículas en una región concreta del dispositivo y con un abanico relativamente reducido de índices del exponente (e.g., *iw=0...iwmax=0...9*) permite representar amplios márgenes de variación de la densidad de carga. Así, pues, cada partícula debe tener asociado una nueva variable entera, *iw*, que se añade a las ya citadas para el índice de valle, *iv*, y para la celda *jp* de la discretización que ocupa.

Los resultados demuestran que la implementación no deteriora apreciablemente la paralelización global del simulador, aún cuando se ha añadido más parte secuencial al programa debido a los procesos de fusión/fisión de partículas. De todos modos, estos procesos son internos a cada celda de la discretización, por lo que en caso necesario también se podría distribuir esta tarea entre los procesadores repartiendo el trabajo en base a las celdas de la discretización.

La simulación de la estructura *n-p-n* de GaAs considerada para las medidas de rendimiento presentadas en este capítulo dan un tiempo de simulación de 1h:45' para la simulación WMC con 8 procesadores con 500...600 partículas por celda para 80000 iteraciones con paso temporal de 0.1fs. El número total de partículas en el dispositivo era del orden de 112000. La misma simulación con el algoritmo EMC estándar con unas 109000 partículas en todo el dispositivo, cada una de ellas equivalente a $4*10^{11}$ electrones, había dado un tiempo de simulación de 1h:39'. La diferencia esencial, no está pues en los tiempos de ejecución sino en las posibilidades de aplicación de cada algoritmo. Para poder simular BJTs con perfiles de dopado realistas el método WMC mantendría constante el número total de partículas a simular y, de aquí también, el tiempo de simulación y los requerimientos de memoria. También mantendría la calidad de la solución. En cambio el método estándar por una parte saturaría pronto los recursos del ordenador y por otra malgastaría recursos simulando excesivas partículas en las zonas más dopadas (de tipo N) e insuficientes en las zonas de baja concentración de electrones (base tipo P y zonas de vaciamiento como la unión base-colector en inversa). Nótese que para el ejemplo poco realista de estructura *n-p-n* presentado, con dopajes respectivos en emisor, base y colector de $5*10^{16}$ cm^{-3}, $5*10^{17}$ $cm^{-}3$ y $5*10^{16}$ cm^{-3} la dispersión de valores de número de partículas por celda era ya del orden de tres órdenes de magnitud (entre 20 y 8000 partículas según la celda).