UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Comparación de modelos de sincronización en programación paralela sobre cluster de multicores
Autor: A.P.U. Enzo Rucci
Director: Ing. Armando E. De Giusti
Codirector: Lic. Franco Chichizola
Carrera: Licenciatura en Informática (P03).

Resumen

Como objetivo inicial de esta tesina se encuentra la investigación de modelos de sincronización para aplicaciones paralelas de alta complejidad computacional sobre cluster de multicores, teniendo en cuenta el auge de esta arquitectura. Como caso de prueba se utiliza el algoritmo de Smith-Waterman para determinar el grado de similitud de dos secuencias de ADN, cuya paralelización emplea un esquema de pipeline debido a la dependencia de datos inherente al problema. Resulta de interés comparar diferentes técnicas de descomposición paralela y mapeo de tareas concurrentes a procesadores para la aplicación mencionada anteriormente. También interesa analizar alternativas de los algoritmos paralelos conocidos para el alineamiento de secuencias, considerando la arquitectura multiprocesador de soporte. Finalmente, estudiar métricas de perfomance para los estudios experimentales que se realicen de manera de obtener una comparación real de la solución a esta clase de problemas, empleando sincronización por mensajes, por memoria compartida e híbrida.

Palabras Claves

Cluster de multicores, modelo de comunicación híbrido, pipeline, algoritmo Smith-Waterman.

Trabajos Realizados

Se estudiaron los fundamentos del procesamiento paralelo para poder desarrollar, analizar y evaluar diferentes alternativas de implementación del algoritmo Smith-Waterman, las cuales emplean distintos modelos de comunicación: memoria compartida, pasaje de mensajes y diferentes combinaciones de ambos.

Conclusiones

Finalizado el trabajo experimental, se analizaron y compararon los comportamientos de las soluciones implementadas. Puede observarse que el algoritmo que emplea únicamente pasaje de mensajes obtiene un rendimiento superior en todos los casos. Esto se debe a dos factores: el poco requerimiento de memoria que tienen los algoritmos, lo cual no permite aprovechar los beneficios del uso de memoria compartida, y la optimización que poseen las librerías de pasaje de mensajes hoy en día para trabajar en ambientes de memoria compartida.

Trabajos Futuros

A partir del estudio y los resultados obtenidos, quedan abiertos un conjunto de temas para su futura investigación:

- analizar la escalabilidad del problema estudiado asegurando un determinado nivel de eficiencia;
- estudiar el comportamiento de la aplicación al utilizar ambientes heterogéneos, donde se combine el cluster de multicores empleado en éste trabajo con clusters heterogéneos tradicionales;
- y analizar y optimizar soluciones híbridas para determinadas clases de problemas, especialmente para los que admiten una solución paralela compuesta (combinando más de un paradigma de interacción).

Fecha de la presentación: Marzo, 2011.

Comparación de modelos de sincronización en programación paralela sobre cluster de multicores

Autor: A.P.U. Enzo Rucci

Director: Ing. Armando E. De Giusti

Co-Director: Lic. Franco Chichizola

Tesina de grado

Licenciatura en Informática

Facultad de Informática

Universidad Nacional de La Plata

Indice

Agradecimientos

A mis padres por enseñarme los valores y principios que hoy poseo y por su apoyo incondicional constante.

A mis abuelos y a mi madrina por todo el afecto brindado.

A mi tío Edgard por enseñarme el valor del trabajo y a estar preparado para afrontar cualquier circunstancia.

A Maitén, a quien le robé múltiples horas durante varios años para alcanzar esta meta, por su paciencia eterna y cariño diario.

A todos los amigos que me dio la vida, por acompañarme en mis momentos tristes y permitirme compartir sus momentos felices.

A mi director Armando De Giusti por sus respuestas acertadas y a mi codirector Franco Chichizola por su predisposición infinita.

A mis profesores. De cada uno de ellos aprendí algo.

A mis compañeros de la facultad. Sin ellos el camino hubiese sido más largo y mucho menos ameno.

Al III-LIDI por recibirme desinteresadamente para realizar mi trabajo de grado y a la Facultad de Informática y a la Universidad Nacional de La Plata por todas las oportunidades brindadas.

Objetivos

Como objetivo inicial de esta tesina se encuentra la investigación de modelos de sincronización para aplicaciones paralelas de alta complejidad computacional, sobre cluster de multicores. Resulta de interés comparar diferentes técnicas de descomposición paralela y mapeo de tareas concurrentes a procesadores, para una aplicación de alta complejidad computacional, como es el alineamiento de secuencias de ADN. También interesa analizar alternativas de los algoritmos paralelos conocidos para el alineamiento de secuencias, considerando la arquitectura multiprocesador de soporte. Finalmente, estudiar métricas de rendimiento para los estudios experimentales que se realicen de manera de obtener una comparación real de la solución a esta clase de problemas, empleando sincronización por mensajes, por memoria compartida e híbrida.

Capítulo 1

Introducción

1.1 ¿Qué es el procesamiento paralelo?

Una computadora tradicional tiene una unidad de procesamiento individual para ejecutar las instrucciones especificadas en un programa. Los problemas son divididos en series discretas de instrucciones, donde las mismas son ejecutadas unas tras otras y sólo una de ellas puede ser ejecutada en un determinado momento. Una manera de incrementar el poder de procesamiento es usar más de una unidad de procesamiento dentro de una única computadora o bien utilizar varias computadoras, trabajando todas juntos sobre el mismo problema. El problema es dividido en partes discretas, donde cada una de ellas es resuelta por una unidad de procesamiento individual de manera paralela. Se puede definir al procesamiento paralelo como el uso simultáneo de múltiples recursos computacionales para resolver un problema [BAR10] [WIL05].

1.2 ¿Por qué usar procesamiento paralelo?

Existen diversas razones que permiten explicar la importancia del paralelismo. A continuación se analizan las principales de ellas.

1.2.1 Resolver problemas más grandes

Algunos problemas son tan grandes y/o complejos que se hace poco práctico resolverlos utilizando una computadora individual, en especial aquellas que poseen una memoria limitada. Por ejemplo, el modelado de enormes estructuras de ADN. Éste problema forma parte de una familia de problemas denominados problemas *Grand Challenge*. Un problema Grand Challenge es aquel que no puede ser resuelto en una cantidad de tiempo razonable con las computadoras estándares. Este tipo de problema suelen requerir enormes cantidades de cálculos repetitivos sobre grandes cantidades de datos para obtener resultados [BAR10] [WIL05].

1.2.2 Resolver problemas con límite de tiempo

Algunos problemas tienen un límite temporal para realizar sus cálculos, por ejemplo el pronóstico del tiempo. Si se necesita de dos días para poder pronosticar el tiempo del día siguiente, entonces la predicción es inútil. En la industria, los cálculos de los ingenieros y las simulaciones deben lograrse en segundos o minutos, de ser posible. Una simulación que toma dos semanas para alcanzar una solución es usualmente inaceptable en un ambiente de diseño, ya que el tiempo de la simulación

debe ser lo suficientemente corto como para que el diseñador pueda trabajar de forma efectiva [WIL05].

1.2.3 Resolver problemas con mayor precisión

Además de obtener el potencial para aumentar la velocidad de un problema determinado, el uso de múltiples computadoras/unidades de procesamiento permite a veces una solución más precisa de un problema a ser resuelto en una cantidad razonable de tiempo. Por ejemplo, pronosticar el clima requiere dividir el aire en una cuadrícula tridimensional donde cada celda representa una solución. Utilizar múltiples computadoras o unidades de procesamiento permite a menudo calcular más soluciones en un tiempo dado, y por lo tanto una solución más precisa [WIL05].

1.2.4 Límites del procesamiento serial

Existen razones tanto físicas como prácticas que ponen límites significativos a la construcción de computadoras secuenciales cada vez más rápidas. Entre ellas se encuentran:

- Velocidades de transmisión: la velocidad de una computadora serial depende directamente de cuan rápido puede mover los datos a través del hardware. Como límites absolutos tenemos la velocidad de la luz (30 cm/ns) y el límite de transmisión del cable de cobre (9cm/ns). Incrementar la velocidad implica aproximar las unidades de procesamiento.
- Límites a la construcción a pequeña escala: la tecnología actual de las unidades de procesamiento está permitiendo colocar una gran cantidad de transistores por chip. Sin embargo, aún con componentes de nivel molecular o atómico, se alcanzará un límite con respecto al tamaño de los mismos.
- Limitaciones económicas: resulta cada vez más caro hacer que las unidades de procesamiento individuales sean más rápidas. En lugar de eso, se puede obtener el mismo rendimiento (o mejor) al utilizar un mayor número de unidades de procesamiento moderadamente veloces, lo cual resulta más barato.

Las arquitecturas de computadoras actuales se apoyan cada vez más sobre el paralelismo a nivel de hardware para mejorar su rendimiento. Múltiples unidades de procesamiento, instrucciones en forma de pipeline y procesadores con múltiples núcleos son muestra de ello [BAR10].

Durante las últimas décadas, la tendencia ha sido hacia redes cada vez más veloces, sistemas distribuidos y arquitecturas con múltiples unidades de procesamiento (aún a nivel de escritorio), lo cual claramente demuestra que el futuro del procesamiento es el paralelismo [BAR10].

1.3 Limitaciones del rendimiento del sistema de memoria

El rendimiento efectivo de un programa sobre una computadora no sólo depende de la velocidad de la unidad de procesamiento sino también de la habilidad del sistema de memoria de alimentar con datos a la misma. Aquí intervienen dos factores que son críticos para obtener un buen desempeño: uno es la *latencia* y el otro es el *ancho de banda*. La latencia es el tiempo que transcurre desde que el sistema de memoria recibe un pedido por una palabra de memoria hasta que retorna un bloque de datos

conteniendo dicha palabra. El ancho de banda es la velocidad a la cual los datos pueden ser transmitidos desde la memoria hasta la unidad de procesamiento [GRA03].

1.3.1 Mejorando la efectividad de la latencia de memoria usando cachés

Manejar la diferencia de velocidad entre la unidad de procesamiento y la memoria DRAM ha motivado un número de innovaciones arquitectónicas en el diseño de sistemas de memoria. Una de esas innovaciones maneja la diferencia colocando una memoria más pequeña, pero más rápida, entre la unidad de procesamiento y la memoria DRAM. Ésta memoria, conocida como *caché*, funciona como un almacenamiento de baja latencia y alto ancho de banda. Los datos requeridos por la unidad de procesamiento son primero colocados en la caché. Todos los accesos subsecuentes a datos que residen en la caché son proporcionados por la misma. Por lo tanto, en principio, si un dato es usado repetidamente, la latencia efectiva de éste sistema de memoria puede ser reducida por la caché. La fracción de datos referenciados satisfechos por la caché es llamada *porcentaje de éxito* (*hit ratio*) de la caché del cómputo del sistema. La tasa de cómputo efectiva de varias aplicaciones no está limitada por la tasa de procesamiento de la CPU, sino por la tasa en que los datos pueden ser proporcionados a la CPU. Esos cómputos son conocidos como *limitados por memoria*. El rendimiento de los programas limitados por memoria se ve impactado críticamente por el porcentaje de éxito de la caché [GRA03].

La mejora en el rendimiento obtenida a partir de la presencia de la caché se basa en la suposición de que un dato será referenciado repetidamente. Ésta noción de referencias repetidas a un dato en un pequeño intervalo de tiempo es llamado *localidad temporal* de las referencias. El reuso de datos es crítico para el rendimiento de la caché dado que si cada dato es usado sólo una vez, aún así debería buscar el dato cada vez que lo usa a la DRAM, debiendo pagar por la latencia de la DRAM en cada operación [GRA03].

1.3.2 Impacto del ancho de banda

El ancho de banda de memoria referencia la tasa a la cual los datos pueden ser movidos entre la unidad de procesamiento y la memoria. Está determinado por el ancho de banda del bus de memoria como de las unidades de memoria. Una técnica muy usada para mejorar el ancho de banda es incrementar el tamaño de los bloques de memoria. Se asume que un pedido de memoria individual retorna un bloque contiguo de cuatro palabras. El grupo de cuatro palabras en este caso es llamado *línea de caché*. Las computadoras convencionales pueden traer desde dos hasta ocho palabras juntas a la caché [GRA03].

Al aumentar el tamaño de bloque, la unidad de procesamiento dispone de más datos en la misma cantidad de tiempo al realizar un pedido de memoria (cuatro palabras en lugar de una). Esto le permite acelerar la ejecución de las operaciones de aquellas aplicaciones que no reutilizan datos. Se debe notar que incrementar el tamaño de bloque no modifica la latencia del sistema. Sin embargo, en este caso cuadriplica el ancho de banda.

Otro aspecto importante que influye en el rendimiento general del sistema y que incide sobre el trabajo del programador está relacionado con la forma en que los datos son distribuidos. Se asume que los datos son distribuidos de manera que las palabras consecutivas en memoria son utilizadas por instrucciones sucesivas. En otras palabras, desde el punto de vista del cómputo, existe una *localidad espacial* en el acceso a los datos. Desde un enfoque centrado en la distribución de los datos, el cómputo se ordena de modo que los sucesivos cálculos requieran datos contiguos. Si

el patrón de acceso a los datos no tiene localidad espacial, entonces es muy probable que el rendimiento del sistema sea pobre [GRA03].

Mientras que el rendimiento de las unidades de procesamiento ha crecido de forma significativa durante las últimas décadas, la latencia y el ancho de banda no le han podido seguir el ritmo. Se presentaron técnicas que permiten combatir esta diferencia, reduciendo la latencia e incrementando el ancho de banda. Para optimizar el rendimiento de una aplicación resulta vital aplicar los siguientes conceptos:

- Explotar la localidad espacial y temporal amortizará la latencia de memoria e incrementará el ancho de banda efectivo.
- Determinadas aplicaciones tienen inherentemente mayor localidad temporal que otras, y por lo tanto mayor tolerancia a un ancho de banda bajo. La proporción del número de operaciones con respecto al número de accesos a memoria es un buen indicador de la tolerancia anticipada al ancho de banda.
- Organizar el cómputo y la distribución de los datos pueden impactar de forma significativa sobre la localidad temporal y especial.

1.3.3 Enfoques alternativos para ocultar la latencia de la memoria

Se considera la navegación por Internet durante las horas pico de tráfico. La falta de respuesta del navegador puede ser aliviada mediante tres simples esquemas:

- anticipar que páginas se van a visitar y realizar los pedidos de ellas con antelación;
- abrir múltiples navegadores y acceder a diferentes páginas en cada uno, por lo que mientras se espera que una página cargue, se puede estar leyendo otras; o
- acceder a un montón de páginas de una vez – amortizando la latencia de varios accesos.

El primer enfoque se conoce como *prebúsqueda*, el segundo como *multihilado*, y el tercero corresponde a la localidad espacial al acceder a las palabras de memoria. De estos tres enfoques, la localidad espacial en el acceso a las palabras de memoria ya se analizó en la sección anterior. A continuación se profundiza en multihilado y prebúsqueda como métodos para ocultar la latencia.

1.3.3.1 Multihilado para ocultar la latencia

Un hilo de ejecución es la unidad de trabajo más pequeña que un sistema operativo puede planificar. Normalmente es el resultado de la división de un programa en dos o más tareas de ejecución concurrente. Múltiples hilos pueden existir dentro del mismo proceso y compartir recursos como la memoria. En particular, los hilos de un proceso comparten las instrucciones de este último (su código) y su contexto. Se profundiza este tema en el Capítulo 4.

En una unidad de procesamiento individual, el multihilado generalmente se da dividiendo el tiempo en que estos se ejecutan: la unidad de procesamiento alterna entre diferentes hilos. Este intercambio de contexto ocurre tan frecuentemente que los usuarios perciben que los hilos se ejecutan en paralelo. En un sistema con más de una unidad de procesamiento, los hilos efectivamente se ejecutan en paralelo, donde cada una de ellas ejecuta uno particular. Las unidades de procesamiento multihiladas son capaces de mantener el contexto de un número de hilos con pedidos pendientes (accesos a memoria, E/S, o pedidos de comunicación) y ejecutarlos a medida que los pedidos son satisfechos. En este tipo de sistemas las unidades de procesamiento

pueden intercambiar el contexto de ejecución en cada ciclo. En consecuencia, son capaces de ocultar la latencia de manera efectiva, proveyendo la concurrencia suficiente para mantener a las unidades de procesamiento ocupadas [GRA03] [KI05]. En la Figura 1.1 se muestra en esquema de la forma en que trabaja esta técnica.

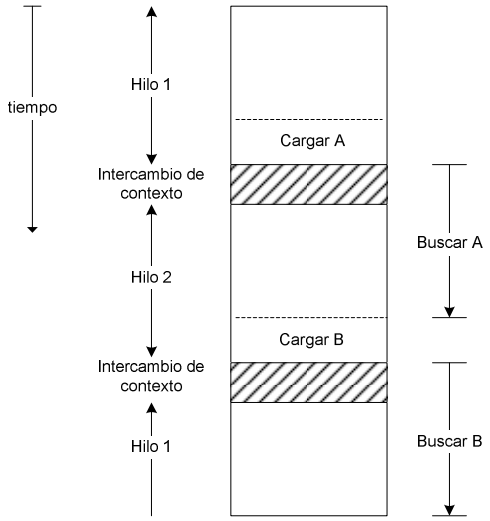

Figura 1.1. Multihilado para ocultar la latencia.

1.3.3.2 Prebúsqueda para esconder la latencia

Normalmente las unidades de procesamiento cargan y usan un dato durante una pequeña fracción de tiempo. Si el dato no está en la caché, entonces la unidad de procesamiento debe esperar hasta que el dato llegue. La idea básica de la prebúsqueda consiste en realizar los pedidos de datos antes del uso real del mismo. Lo ideal sería que todos los datos sean traídos a la caché antes de ser utilizados por la unidad de procesamiento. Podría suceder que un dato sea sobrescrito entre su carga inicial y su uso, lo cual requiere una recarga del mismo. Se debe observar que, aún así, esto no es peor que la situación en la cual la carga no fue anticipada. Desde el punto de vista de la unidad de procesamiento, la prebúsqueda puede mejorar la utilización de la misma al solapar procesamiento con accesos a memoria. Si se revisa detalladamente esta técnica se puede notar que la prebúsqueda funciona por las mismas razones que el multihilado. Al anticipar las cargas, se trata de identificar hilos de ejecución que no tengan dependencias de recursos (es decir, usan los mismos registros) con respecto a otros hilos [GRA03] [KI05]. En la Figura 1.2 se muestra un esquema de la forma en que trabaja esta técnica.

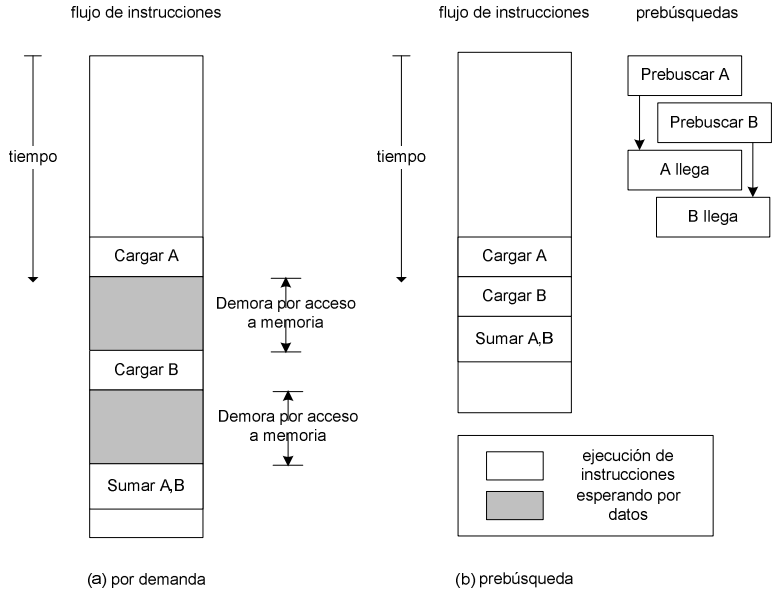

Figura 1.2. Prebúsqueda para ocultar la latencia

1.3.4 Comparación entre multihilado y prebúsqueda

Por lo analizado anteriormente parecería que el multihilado y la prebúsqueda resuelven todos los problemas asociados al rendimiento del sistema de memoria. Sin embargo, ambas técnicas se ven críticamente impactadas por el ancho de banda de la memoria [GRA03].

Para demostrar lo enunciado en el párrafo anterior, se consideran dos casos: primero, la ejecución de un único hilo; y segundo, una ejecución multihilada. En el primer caso, como sólo un hilo se ejecuta, éste tiene disponible la caché en forma completa para su uso. En cambio, cuando la ejecución es multihilada, la caché es dividida entre todos los hilos. De acuerdo a las prestaciones de hardware y a la fracción de éxitos de la caché, los requerimientos del ancho de banda de un sistema multihilado podrían incrementarse en forma significativa debido a la pequeña porción de caché que le corresponde a cada hilo. En estos casos, los sistemas multihilados se ven limitados por el ancho de banda en lugar de serlo por la latencia. Es importante notar que el multihilado y la prebúsqueda solo tratan el problema de la latencia y que a veces pueden agravar el problema del ancho de banda.

Otro aspecto importante que no se debe ignorar son los recursos de hardware adicionales requeridos para usar efectivamente la prebúsqueda y el multihilado. Se supone que se han anticipado 10 cargas de datos a registros. Estas cargas requieren que 10 registros estén disponibles. Como se mencionó anteriormente, podría ocurrir que uno de estos registros sea sobrescrito y necesite ser cargado otra vez. Esto no incrementa la latencia de la búsqueda más que en el caso en el cual no hay prebúsqueda. Sin embargo, como se está buscando el mismo dato dos veces, el requerimiento de ancho de banda del sistema de memoria se duplica. Se puede aliviar esta problemática al contar con archivos de registros más grandes y cachés para soportar prebúsqueda y multihilado [GRA03] [KI05].

1.4 Definiciones y conceptos básicos

El primer paso en la programación paralela es el diseño de un algoritmo o programa paralelo que permita resolver un problema dado. El diseño comienza con la

descomposición de las computaciones del problema en varias partes, a las cuales se las llama *tareas*. La manera en que el problema es dividido se la denomina *técnica de descomposición*. En ocasiones seleccionar la descomposición a utilizar puede ser complicada y trabajosa, dado que usualmente existen diferentes métodos de descomposición que se pueden aplicar a un mismo algoritmo. El número y el tamaño de las tareas que son producto de la descomposición aplicada determinan la *granularidad* de la misma. Una descomposición que produce un gran número de pequeñas tareas es llamada *de grano fino*, mientras que una que produce una pequeña cantidad de tareas de gran tamaño se la denomina *de grano grueso* [GRA03] [RAU10]. Se tratan con mayor detalle las técnicas de descomposición en el Capítulo 4.

Las tareas en las cual se descompone una aplicación son codificadas utilizando un lenguaje de programación paralela y asignadas a *procesos* o *hilos*, los cuales a su vez serán más tarde asignados a las distintas *unidades de procesamiento*. Un proceso es un bloque de programa secuencial, con flujo de control propio. Se puede ver a un hilo como un proceso liviano. Las unidades de procesamiento son los dispositivos físicos sobre los cuales se ejecutan las tareas. La asignación de procesos o hilos a estas unidades físicas se denomina *mapeo* y del mismo se encarga usualmente el sistema operativo aunque puede ser influido a veces por el programador [RAU10]. Se profundiza sobre los diferentes métodos de mapeo en el Capítulo 4.

Las tareas pueden tener *dependencias* entre sí, las cuales pueden imponer un orden de ejecución específico de las tareas paralelas: si una tarea necesita datos producidos por otra, entonces deberá esperar a que esta se ejecute para poder hacerlo ella. Es por ello, que las dependencias entre tareas limitan el paralelismo. Además, los procesos o hilos necesitan *sincronizar* y *coordinar* para una ejecución correcta. Los métodos de sincronización y coordinación en el procesamiento paralelo están fuertemente conectados con la manera en que la información es intercambiada entre procesos o hilos, y esto generalmente depende de la organización de memoria del hardware [RAU10].

Se pueden clasificar a las máquinas paralelas de acuerdo a la organización de la memoria. Como primera clasificación, se las puede dividir entre *máquinas de memoria compartida* y *máquinas de memoria distribuida*. Generalmente se asocia el término hilo a las máquinas de memoria compartida, mientras que sucede lo mismo con el término proceso y las máquinas de memoria distribuida. Las máquinas de memoria compartida poseen una memoria compartida global la cual almacena los datos del programa y es accesible por todas las unidades de procesamiento. Un hilo intercambia información escribiendo variables que luego serán leídas por otro. Éste mecanismo de comunicación se conoce como *memoria compartida*. Las máquinas de memoria distribuida mantienen una memoria privada para cada unidad de procesamiento, la cual sólo puede ser accedida por ella. Los procesos intercambian información enviándose mensajes, lo cual se conoce como *pasaje de mensajes* [AND00] [RAU10]. Los diferentes tipos de máquinas paralelas son tratados en el Capítulo 2 y los distintos modelos de sincronización/comunicación en el Capítulo 4.

1.5 Concepto de sistema paralelo

Un algoritmo secuencial suele ser evaluado a partir de su tiempo de ejecución, el cual se expresa como una función del tamaño de su entrada. Evaluar a un algoritmo paralelo de la misma manera que a uno secuencial sería injusto, ya que el tiempo de ejecución no sólo depende del tamaño de la entrada sino también del número de unidades de procesamiento utilizadas y de las velocidades de las mismas para computar y comunicar datos. Es por ello, que un algoritmo paralelo no puede ser evaluado en forma aislada de una arquitectura paralela sin perder precisión. Un

sistema paralelo es la combinación de un algoritmo junto a la arquitectura paralela sobre la cual se ejecuta [GRA03].

1.6 Dificultades en el procesamiento paralelo

Paralelizar una aplicación puede resultar ser un proceso muy costoso desde el punto de vista del esfuerzo de programación. A continuación se enumeran algunas de las dificultades que se pueden encontrar a la hora de escribir un programa paralelo.

1.6.1 Grado de paralelización

Los problemas son paralelizables en distintos grados. Existen problemas que no pueden ser paralelizados y sólo pueden ser resueltos de manera secuencial. Como por ejemplo el cálculo de la sucesión de Fibonacci, cuya fórmula se observa en la Figura 1.3.

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

Figura 1.3. Fórmula para calcular la sucesión de Fibonacci

Éste problema no es factible de ser paralelizado. Si se observa la fórmula, se puede notar que para poder calcular el valor de *f(n)* es necesario que *f(n-1)* y *f(n-2)* hayan sido calculados previamente. Estos tres términos no pueden ser computados en forma independiente y por lo tanto, tampoco en paralelo.

Otros problemas, en cambio, requieren muy poco esfuerzo de programación y proveen un alto grado de paralelismo. Como por ejemplo la predicción del clima. Esta tarea requiere comparar diferentes escenarios, los cuales son independientes entre sí. Esto permite simular los distintos escenarios en paralelo, sin necesidad de que las tareas deban sincronizar o comunicarse.

Es tarea del programador identificar aquellas porciones del problema que son independientes entre sí, para luego asignarlas a tareas que se ejecuten en paralelo.

1.6.2 Complejidad adicional

Escribir un algoritmo paralelo involucra una serie de pasos que no están presentes en la escritura de un algoritmo secuencial, como pueden ser la descomposición del problema en tareas, el mapeo de tareas a unidades de procesamiento y la sincronización/comunicación entre tareas. Las herramientas de hoy en día no proveen soporte directo para la mayoría de ellos, lo cual implica que el programador muchas veces debe basarse en su experiencia para poder llevarlos a cabo. Naturalmente, estos pasos adicionales y complejos son una de las razones por las cuales la programación paralela es considerada difícil.

1.6.3 Propensión a errores

Al momento de escribir un algoritmo paralelo, no sólo se cuenta con la dificultad de tener más pasos que resolver sino también con que estos pasos son propensos a errores. Si la descomposición en tareas elegida no es la correcta, el rendimiento del programa será pobre. Si no se realiza el mapeo adecuadamente, el overhead generado por la sincronización entre las tareas podría opacar toda mejora obtenida por la paralelización. Y si se cometen errores durante las fases de

sincronización/comunicación, mas allá de la pérdida de rendimiento, el programa podría no ejecutarse o producir resultados incorrectos.

1.6.4 Difícil prueba y depuración

La prueba y la depuración de programas paralelos es un problema. El no determinismo asociado a las ejecuciones concurrentes y paralelas implica que dos ejecuciones del mismo programa no necesariamente serán idénticas. Esto complica la predicción, comprensión y depuración de los programas paralelos.

1.6.5 Dependencia del hardware

Anteriormente se mencionó la necesidad de evaluar a un algoritmo paralelo en conjunto con la arquitectura de soporte y no hacerlo de forma aislada. El número de unidades de procesamiento y sus prestaciones, las características de caché, la red de interconexión, entre otros, son elementos que inciden sobre el rendimiento final del algoritmo. Es por ello, que la optimización del algoritmo se realiza de manera de aprovechar las características del hardware subyacente. Esto implica que un algoritmo que obtiene un rendimiento óptimo sobre una arquitectura particular, podría no desempeñarse de la misma forma sobre una arquitectura diferente.

Capítulo 2

Arquitecturas paralelas

2.1 Plataformas paralelas

Una plataforma paralela consiste de dos o más unidades de procesamiento vinculadas a partir de algún tipo de red de interconexión. Se pueden clasificar a las plataformas paralelas según su organización lógica así como también según su organización física. La organización lógica se refiere a la manera en que el programador ve a la plataforma paralela, mientras que la organización física hace referencia al hardware real de la plataforma. A continuación se describen los diferentes tipos de plataformas paralelas.

2.1.1 Plataformas de memoria compartida

Un multiprocesador de memoria compartida consiste de múltiples unidades de procesamiento conectadas a múltiples módulos de memoria. La conexión entre unidades de procesamiento y memorias se da a través de algún tipo de red de interconexión, como puede ser un bus o un switch crossbar. En un sistema de memoria compartida, existe un único espacio de direcciones, por lo que cualquier dirección de memoria es accesible por todas las unidades de procesamiento, las cuales interactúan modificando datos almacenados en éste espacio compartido. Si el tiempo que le toma a una unidad de procesamiento acceder a una dirección de memoria es idéntico para todas las unidades de procesamiento, entonces decimos que el multiprocesador tiene acceso uniforme a memoria (UMA). Por otro lado, si el tiempo que toma acceder a determinadas direcciones de memoria es mayor que a otras, entonces el multiprocesador es llamado de acceso no uniforme a memoria (NUMA) [AND00] [GRA03].

Idealmente, se desea que el sistema sea UMA. Sin embargo, los grandes sistemas de memoria compartida suelen ser de tipo NUMA dado que resulta muy difícil implementar hardware que provea rápido acceso a toda la memoria compartida. La mayoría de los grandes sistemas de memoria compartida cuentan con alguna estructura de memoria jerárquica o distribuida [WIL05].

En cualquiera de los sistemas descritos en el párrafo anterior, una memoria caché de alta velocidad está siempre presente para mantener los contenidos de las direcciones de memoria principal recientemente referenciados. La presencia de cachés en las unidades de procesamiento también acarrea la problemática de tener múltiples copias de una única palabra de memoria siendo manipulada por más de una unidad de procesamiento al mismo tiempo. El mecanismo que permite que múltiples operaciones concurrentes sobre una misma palabra de memoria tengan una semántica bien definida se llama coherencia de caché. Más adelante se detalla este mecanismo [GRA03].

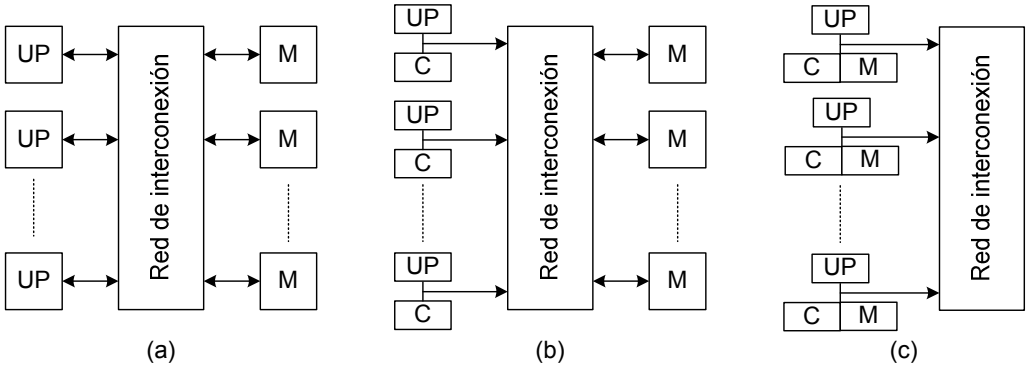

Figura 2.1. Plataformas de memoria compartida típicas. (a) Computadora de memoria compartida de acceso uniforme a memoria; (b) computadora de memoria compartida de acceso uniforme a memoria con cachés; (c) Computadora de memoria compartida de acceso no uniforme a memoria con memoria local.

2.1.2 Plataformas de memoria distribuida

Lógicamente, una plataforma de memoria distribuida consiste de *p* nodos de procesamiento unidos por una red de interconexión. Cada uno de éstos puede ser una computadora individual o un multiprocesador de memoria compartida (una tendencia que está ganando impulso en las plataformas de memoria distribuida). De cualquier forma, cada nodo posee su propio espacio de direcciones, el cual no es accesible por el resto y la red de interconexión les permite a los nodos enviarse mensajes entre ellos. Éste intercambio de mensajes es utilizado para transferir datos, trabajo y sincronizar acciones entre nodos [GRA03].

Resulta más fácil escalar físicamente a los sistemas de memoria distribuida que a los sistemas de memoria compartida [WIL05].

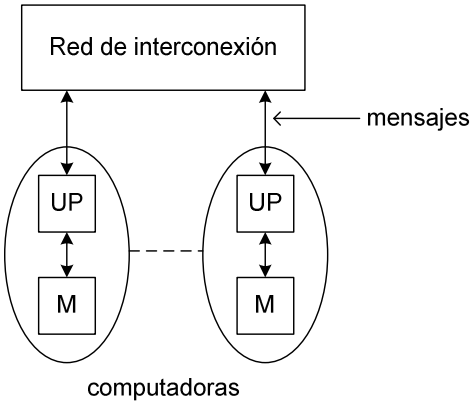

Figura 2.2. Plataforma de memoria distribuida.

2.1.3 Plataformas de memoria compartida distribuida

A menudo los programadores encuentran más atractivo el paradigma de memoria compartida que el de pasaje de mensajes. A favor del paradigma de pasaje de mensajes podemos destacar que no necesita de mecanismos de sincronización especiales para controlar el acceso simultáneo a los datos. Por otro lado, requiere que los programadores hagan de forma explícita el intercambio de datos entre procesos, lo cual hace a los programas propensos a errores y difíciles de depurar. Los datos son copiados y no compartidos. Esto puede resultar problemático en aquellas aplicaciones

que requieren de múltiples operaciones a lo largo de grandes cantidades de datos [WIL05].

El problema del modelo de memoria compartida se encuentra en el hardware. Los multiprocesadores de memoria compartida tradicionales conectados a través de un bus tienen un límite en la cantidad de unidades de procesamiento que puede vincular dicho bus. En contraste, las plataformas de memoria distribuida son fáciles de escalar y a un bajo costo [GRA03] [WIL05].

Teniendo en cuenta que el paradigma de memoria compartida es deseable desde el punto de vista del programador y que las plataformas de memoria distribuida son más económicas y fáciles de escalar, diversos investigadores plantearon la idea de memoria compartida distribuida. Al igual que en los sistemas de memoria distribuida, en los sistemas de memoria compartida distribuida, la memoria se encuentra físicamente distribuida entre todas las unidades de procesamiento, pero a diferencia de los primeros, cada unidad de procesamiento puede acceder a la memoria completa mediante un único espacio de direcciones. Cuando una unidad de procesamiento desea acceder a una dirección que no está en su memoria local, se utiliza pasaje de mensajes para pasar los datos, pero de alguna manera automatizada que permite ocultar el hecho de que la memoria está físicamente distribuida. Obviamente, los accesos a ubicaciones remotas tardarán más que aquellos que se realizan a la memoria local [AND00] [WIL05].

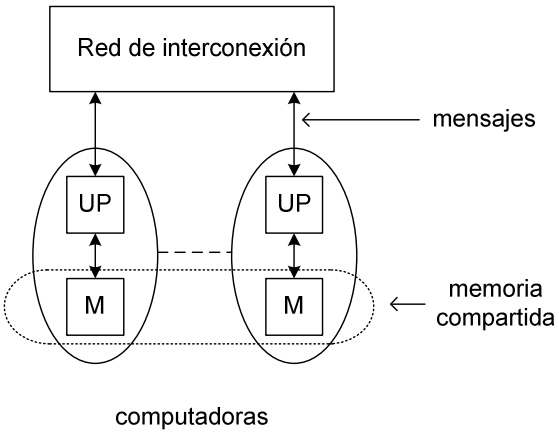

Figura 2.3. Plataforma de memoria compartida distribuida.

2.2 Clasificación de Flynn

En 1966, Michal J. Flynn propuso clasificar a las máquinas de acuerdo a la cantidad de flujos de instrucciones y de datos que las mismas pueden manejar. Los flujos de instrucciones como los de datos se dividen en simples o múltiples. A continuación la clasificación de Flynn.

2.2.1 SISD (Single Instruction, Single Data)

En ésta clase se encuentran las computadoras secuenciales que cuentan con una única unidad de procesamiento. Un sólo flujo de instrucciones opera sobre un único flujo de datos en cada ciclo del reloj. Es el tipo más viejo y, aún hoy, más común. Como ejemplo se puede nombrar a las PC's tradicionales.

2.2.2 SIMD (Single Instruction, Multiple Data)

En éste tipo de computadora paralela, todas las unidades de procesamiento ejecutan la misma instrucción en cada ciclo de reloj, pero cada una de ellas opera sobre diferentes datos. Hay un número importante de aplicaciones en las cuales una máquina de éstas características puede obtener ventajas en el rendimiento. La mayoría de ellas opera sobre arreglos de datos. Por ejemplo, simulaciones por computadoras de sistemas físicos (como el pronóstico del tiempo) o el procesamiento de imágenes. Las soluciones a estos problemas requieren aplicar operaciones similares a grandes cantidades de datos. Contar con un sistema preparado para ello, permite desarrollar soluciones eficientes en hardware con relativa facilidad.

2.2.3 MISD (Multiple Instruction, Single Data)

Cada unidad de procesamiento opera independientemente del resto sobre el mismo flujo de datos. Existen pocos ejemplos reales de éstas máquinas y ninguna se ha producido en masa. Se pueden clasificar en éste grupo a algunas arquitecturas en forma de pipeline, o también algunos sistemas de tolerancia a fallos.

2.2.4 MIMD (Multiple Instruction, Multiple Data)

Es el tipo más común de computadora paralela y la mayoría de ellas cae en esta categoría. Las unidades de procesamiento pueden ejecutar diferentes flujos de instrucciones sobre distintos flujos de datos. Las plataformas de memoria compartida y distribuida descriptas anteriormente se clasifican en éste grupo.

2.3 Máquinas de memoria distribuida

2.3.1 Costos de las comunicaciones en máquinas de pasaje de mensajes

Durante la ejecución de un programa paralelo, las unidades de procesamiento se comunican para poder resolver un problema determinado. Esta comunicación que se da entre unidades es una de las mayores fuentes de overhead que afectan la ejecución de dichos programas y el costo de la misma depende de múltiples factores como pueden ser el modelo de programación, la topología de red, el mecanismo de asignación de ruta, entre otros [GRA03].

Comunicar un mensaje entre dos nodos de una red toma un tiempo que está dado por la suma del tiempo tomado para preparar el mensaje para su transmisión y el tiempo tomado por el mensaje para atravesar la red hasta llegar a su destino. Los principales parámetros que determinan la comunicación son:

- Tiempo de inicio (t_s): El tiempo de inicio es el tiempo requerido para manejar un mensaje en los nodos emisor y receptor. Éste incluye el tiempo de preparación del mensaje (agregar información inicial y final y también para la corrección de errores), el tiempo para ejecutar el algoritmo de asignación de ruta, y el tiempo para establecer un punto de contacto entre el nodo emisor y el nodo receptor. Éste retraso sólo es necesario una única vez por cada transferencia individual de un mensaje.
- Tiempo por salto (t_h): Luego de que un mensaje abandona un nodo, le toma una cantidad de tiempo finita alcanzar el próximo nodo en su camino. El tiempo tomado por la cabecera de un mensaje para viajar entre dos nodos directamente conectados en la red es llamado tiempo por salto. También es conocido como *latencia de nodo*. El tiempo por salto está directamente relacionado a la latencia del conmutador de asignación

de ruta para determinar por cual buffer de salida o canal el mensaje deberá ser reenviado.

- Tiempo de transferencia por palabra (t_w): Si el ancho de banda del canal es de r palabras por segundo, entonces cada palabra toma $t_w = 1 / r$ segundos para atravesar el enlace. Éste tiempo es llamado tiempo de transferencia por palabra e incluye los overheads incurridos por la red y por el almacenamiento en buffers.

2.3.1.1 Asignación de ruta *Store-and-Forward*

Se desea enviar un mensaje el cual debe atravesar múltiples nodos intermedios hasta alcanzar su destino. En el método de asignación de ruta *Store-and-Forward*, cada nodo intermedio reenvía el mensaje al siguiente nodo luego de que lo haya recibido y almacenado en forma completa [GRA03].

Sea m la cantidad de palabras del mensaje del párrafo anterior y n la cantidad de nodos intermedios que el mismo debe atravesar. En cada nodo, el mensaje requiere de $t_h + t_w m$ unidades de tiempo para que tanto la cabecera como el resto del mensaje puedan atravesar el nodo. Como hay l de esos nodos, el tiempo total es de $(t_h + t_w m)l$. Luego, el tiempo total de comunicación utilizando la técnica de asignación de ruta Store-and-Forward para enviar un mensaje de m palabras que atraviesa l nodos es el de la Ecuación 2.1.

$$t_{com} = t_s + (t_h + t_w m)l \qquad (2.1)$$

En las computadoras paralelas actuales, el tiempo por salto t_h es bastante pequeño. Para la mayoría de los algoritmos paralelos, es menor que $t_w m$ aún para pequeños valores de m, por lo que puede ser ignorado, permitiendo simplificar como se muestra en la Ecuación 2.2.

$$t_{com} = t_s + t_w ml \qquad (2.2)$$

2.3.1.2 Asignación de ruta por paquetes

La asignación de ruta Store-and-Forward no aprovecha los recursos comunicacionales. Un mensaje es enviado desde un nodo al siguiente luego de que el mensaje completo haya sido recibido. Una alternativa posible consiste en dividir al mensaje original en partes iguales antes de enviarlo. Los nodos intermedios no esperan la recepción del mensaje completo, sino que reenvían cada pedazo al siguiente nodo luego de recibirlo. De esta forma, no sólo se reduce el tiempo que cada nodo espera sino también se incrementa la utilización de los recursos. Además, éste principio ofrece otras ventajas: menor overhead por pérdida de paquetes (errores), la posibilidad de que los paquetes tomen diferentes caminos, y mejores capacidades para la corrección de errores. Por las razones expuestas anteriormente, esta técnica es la base para las redes de comunicación de larga distancia como puede ser Internet, donde las tasas de errores, el número de saltos, y la variación del estado de la red pueden ser altos. Obviamente, éste método no está exento de overhead y el mismo se encuentra en la información de ruta, corrección de errores y secuencia que cada paquete debe cargar.

Se considera la transferencia de un mensaje de m palabras. El tiempo que lleva programar las interfases de red, calcular la información de ruta, entre otras tareas

preliminares, es independiente del tamaño del mensaje y el mismo está incluido en el tiempo de inicio t_s de la transferencia del mensaje. Se asume que las tablas de ruteo no se modifican durante la transferencia del mensaje (es decir, todos los paquetes siguen el mismo camino). Si bien esta suposición no es válida bajo todas las circunstancias, sirve al efecto de definir un modelo de costo para la transferencia de mensajes. El mensaje es dividido en paquetes, los cuales son ensamblados con sus campos de error, ruta y secuencia. El tamaño de un paquete está dado ahora por $r + s$, donde r representa el mensaje original y s la información adicional acarreada en el paquete. El tiempo que consume empaquetar el mensaje es proporcional al tamaño del mismo y lo denotamos mt_{w1}. Si la red es capaz de comunicar una palabra cada t_{w2} unidades de tiempo, incurre en una demora de t_h por cada salto, y si el primer paquete atraviesa l nodos, entonces éste paquete requiere de $t_h l + t_{w2}(r + s)$ unidades de tiempo para llegar a su destino. Pasado éste tiempo, el nodo final recibe un paquete adicional cada $t_{w2}(r + s)$ unidades de tiempo. Como hay $m/r - 1$ paquetes adicionales, el tiempo total de comunicación está dado por la Ecuación 2.5.

$$t_{com} = t_s + t_{w1}m + t_h l + t_{w2}(r + s) + (m/r - 1)t_{w2}(r + s) \quad (2.3)$$

$$= t_s + t_{w1}m + t_h l + t_{w2}m + t_{w2}\, m\, s/r \quad (2.4)$$

$$= t_s + t_h l + t_w m, \quad (2.5)$$

donde

$$t_w = t_{w1} + t_{w2}\,(1 + s/r). \quad (2.6)$$

La asignación de rutas por paquetes es ideal para redes donde los estados cambian continuamente y las tasas de error son elevadas, como las redes LAN y WAN. Esto se debe a que los paquetes podrían tomar diferentes rutas y las retransmisiones pueden limitarse a los paquetes perdidos [GRA03].

2.3.1.3 Asignación de ruta *Cut-Through*

El método de asignación de ruta *Cut-Through* no es más que una versión optimizada de la asignación de ruta por paquetes. Esta técnica se basa en imponer ciertas restricciones a la transferencia de mensajes en las redes de interconexión para computadoras paralelas, con el objetivo de reducir el overhead asociado al intercambio de paquetes. Al forzar a todos los paquetes a tomar el mismo camino, se elimina el overhead asociado a transmitir información de ruta en cada paquete. Al forzar la entrega en secuencia, la información de secuencia puede ser eliminada. Al asociar la información de error a nivel de mensaje en lugar de hacerlo a nivel de paquete, el overhead asociado a la detección y corrección de errores puede ser reducido. Finalmente, como la tasa de errores en redes de interconexión para máquinas paralelas son extremadamente bajas, se pueden usar mecanismos de detección de errores ligeros en lugar de esquemas costosos que permitan corregirlos.

Los mensajes son divididos en unidades de tamaño fijo llamadas *dígitos de control de flujo* o *flits*, los cuales son más pequeños que los paquetes ya que no

cargan con el overhead asociado a los mismos. Para establecer una conexión entre los nodos origen y destino, un flit trazador es enviado. Una vez establecida la conexión, los flits son enviados uno tras otro, los cuales siguen el mismo camino en forma ordenada. Un nodo intermedio pasa el flit al siguiente nodo tan pronto como lo recibe. Esto significa que no espera al mensaje completo antes de reenviarlo. A diferencia del método Store-and-Forward, ya no es necesario mantener buffers en cada nodo intermedio para almacenar el mensaje completo. Es por ello, que el método Cut-Through no sólo requiere de menos memoria y ancho de banda en los nodos intermedios, sino que también es más rápido.

Supongamos que enviamos un mensaje a través de la red. Si el mensaje atraviesa l nodos, y t_h es el tiempo por salto, entonces la cabecera del mensaje requiere de lt_h unidades de tiempo para alcanzar el destino. Si el mensaje consiste de m palabras, entonces el mensaje completo llega luego de $t_w m$ unidades de tiempo después del arribo de la cabecera. Luego, el tiempo total de comunicación para el método Cut-Through es el de la Ecuación 2.7.

$$t_{com} = t_s + lt_h + t_w m \qquad (2.7)$$

Éste tiempo representa una mejora sobre la técnica Store-and-Forward dado que los términos correspondientes al número de saltos y al número de palabras se suman en lugar de multiplicarse.

La mayoría de las computadoras paralelas actuales y varias redes LAN admiten éste método [GRA03].

2.3.1.4 Un modelo de costo simplificado para la comunicación de mensajes

Como se vio en la Sección 2.3.4, el costo de comunicar un mensaje entre dos nodos que se encuentran a l saltos de distancia usando el método de enrutamiento Cut-Through está dado por

$$t_{com} = t_s + lt_h + t_w m. \qquad (2.8)$$

Teniendo en cuenta la ecuación anterior, si se desea optimizar el costo de las transferencias de mensajes, se debe:

1. Comunicar al por mayor. Por cada mensaje enviado se debe pagar un costo de inicio t_s. Si en lugar de enviar pequeños mensajes, se juntan a todos en un único mensaje más grande entonces se puede amortizar la latencia de inicio. Esta optimización tiene sentido más que nada en plataformas como clusters y máquinas de pasaje de mensajes, donde el valor de t_s suele ser mucho mayor que el de t_h o t_w.
2. Minimizar el volumen de datos. De ser posible se debe reducir el volumen de datos a comunicar lo más que podamos, de manera de minimizar el overhead incluido en el tiempo de transferencia por palabra t_w.
3. Minimizar la distancia de las transferencias de datos. Minimizar el número de saltos l que un mensaje debe atravesar.

Mientras que los primeros dos objetivos son relativamente fáciles de lograr, la tarea de minimizar la distancia entre los nodos que se comunican es difícil, y en varios casos una carga innecesaria para el diseñador del algoritmo. Esto es una

consecuencia directa de las siguientes características de plataformas paralelas y paradigmas:

- En varias librerías de pasaje de mensajes, el programador tiene poco control sobre el mapeo de procesos a unidades de procesamiento. En estos paradigmas, mientras las tareas podrían tener topologías bien definidas y puedan comunicarse sólo entre vecinos de dicha topología, el mapeo de procesos a nodos podría destruir esta estructura.
- Varias arquitecturas se apoyan sobre enrutamientos aleatorios, en la cual un mensaje es enviado a un nodo al azar desde el nodo origen y desde éste nodo intermedio hasta el nodo destino. Esto alivia las zonas congestionadas y la competencia en la red. Minimizar el número de saltos en una red con enrutamiento aleatorio no tiene beneficios.
- El tiempo por salto (t_h) está típicamente dominado por la latencia de inicio (t_s) en los mensajes pequeños o por la componente por palabra ($t_w m$) en el caso de los mensajes grandes. Como el número máximo de saltos (l) en la mayoría de la redes es relativamente pequeño, el tiempo por salto puede ser ignorado con una pérdida pequeña de precisión.

Los puntos anteriores llevan a pensar en un modelo de costo más simple en el cual el costo de transferir un mensaje entre dos nodos en una red está dado por la Ecuación 2.9:

$$t_{com} = t_s + t_w m \qquad (2.9)$$

Esta expresión influye no sólo en el diseño de algoritmos independientes de la arquitectura sino también en la precisión de las predicciones de ejecución. Éste modelo de costos asume que la cantidad de tiempo requerida para comunicar un mensaje entre nodos es la misma para cualquier par de ellos, lo cual corresponde a una red completamente conectada. En lugar de diseñar algoritmos para cada arquitectura particular (por ejemplo, en forma de malla, hipercubo, o árbol), se puede diseñar algoritmos con éste modelo de costos en mente y luego transportarlo a cualquier computadora paralela.

Existe una pérdida de precisión (o fidelidad) en la predicción cuando el algoritmo es transportado desde nuestro modelo simplificado (el cual asume una red completamente conectada) a una arquitectura real. Si la suposición inicial es válida (el término t_h es típicamente dominado por el término t_s o t_w), entonces la pérdida de precisión debería ser mínima. Sin embargo, se debe notar que el modelo de costo básico sólo es válido para redes no congestionadas. No todas las arquitecturas se congestionan al mismo tiempo; cada una de ellas posee un límite particular. Por ejemplo: un arreglo lineal tiende a congestionarse más rápido que un hipercubo. Además, la congestión que produce en la red cada patrón de comunicación es diferente. En consecuencia, el modelo de costos simplificado es válido mientras el patrón de comunicación subyacente no congestione la red [GRA03].

2.3.2 Clusters y Multiclusters

2.3.2.1 Clusters

Un *cluster* puede definirse como una colección de computadoras individuales interconectadas que trabajan en conjunto como un único recurso integrado de computación. Lo que caracteriza a los clusters es que cada nodo de procesamiento es un sistema de cómputo en sí mismo, dotado de características de hardware y sistema

operativo propios. Aún más, un nodo podría ser incluso un multiprocesador de memoria compartida. Los componentes de un cluster usualmente se encuentran interconectados mediante algún tipo de red de alta velocidad, como puede ser una red LAN, y en muchos aspectos pueden aparecer como un sistema individual para usuarios y aplicaciones. Éste tipo de sistemas ofrece una manera rentable de mejorar el rendimiento (velocidad, disponibilidad, rendimiento, etc.) comparado con supercomputadoras de similares características [TRO09].

Los clusters pueden tener diferentes objetivos y se construyen de acuerdo al mismo. Por ejemplo, los *clusters de alta disponibilidad* buscan mejorar la disponibilidad de los servicios ofrecidos, es por ello que cuentan con nodos redundantes de manera de no discontinuar los servicios ante la falla de algún componente. También se puede nombrar a los *clusters de balance de carga*, los cuales son diseñados con la finalidad de distribuir el trabajo tanto como se pueda entre los nodos del cluster.

Cuando todas las máquinas que componen un cluster tienen las mismas características, se dice que el cluster es *homogéneo*. De otro modo, es *heterogéneo*.

Los clusters permiten obtener un alto rendimiento a un bajo costo. Si a esto se le suma que pueden expandirse fácilmente, se explica porqué el uso de clusters es hoy una de las formas de cómputo paralelo/distribuido más populares.

2.3.2.2 Multiclusters

Un multicluster es el resultado de interconectar múltiples redes de computadoras dedicadas a una aplicación paralela con el objetivo de desarrollar un algoritmo o implementar un sistema de software, compartiendo parcialmente recursos e incrementando la potencia de cómputo global [ABB05].

Diferentes tipos de redes LAN y WAN se han empleado para interconectar las diferentes unidades de procesamiento de cada cluster y entre clusters.

2.4 Máquinas de memoria compartida

Las máquinas de memoria compartida cuentan con múltiples unidades de procesamiento, donde cada una posee una memoria caché propia y acceso a una memoria global compartida. Podría suceder que las diferentes unidades de procesamiento trabajen con el mismo dato, lo que provoca tener múltiples copias del dato en los diferentes niveles de memoria. Es por ello que las máquinas de memoria compartida requieren de hardware adicional para poder mantener las múltiples copias de un dato de forma consistente.

2.4.1 Coherencia de caché

El problema de mantener las caches de forma coherente en los sistemas de memoria compartida es aún más complejo que en los sistemas que cuentan con una única unidad de procesamiento. Esto se debe a que además de tener múltiples copias de un dato, se cuenta con más de una unidad de procesamiento que podría modificarlo. Si se considera la siguiente situación: dos unidades de procesamiento, UP_0 y UP_1, se encuentran conectadas a través de un bus a una memoria global compartida. Ambas unidades de procesamiento leen la misma variable. A partir de esto, se tienen tres copias de la misma variable (la que está en la memoria global más las copias individuales que se encuentran en la caché de cada unidad de procesamiento). Podría ocurrir que una unidad de procesamiento esté escribiendo un nuevo valor en la variable y que la otra acceda a la misma antes de que la operación de escritura finalice. Para evitar la problemática anterior, el mecanismo de coherencia

debe garantizar que las operaciones sean ejecutadas manteniendo la propiedad de consistencia secuencial. El término consistencia secuencial fue introducido por Leslie Lamport en 1979 y sostiene que el resultado de cualquier ejecución debe ser el mismo que si las operaciones de todas las unidades de procesamiento fueran ejecutadas en algún orden secuencial, y las operaciones de cada unidad de procesamiento individual deben aparecer en ésta secuencia en el orden especificado por sus programas. En otras palabras, cualquier ejecución paralela se debe corresponder con alguna ejecución entrelazada en una unidad de procesamiento individual. De ésta forma se evita que una variable que está siendo escrita, sea leída hasta que la operación de escritura finalice. Cuando una unidad de procesamiento modifica su copia de la variable, debe ocurrir una de dos cosas: las otras copias son actualizadas o bien, son marcadas como inválidas. Si ninguno de los dos procedimientos ocurriera, las unidades de procesamiento podrían estar trabajando con valores incorrectos de la variable. Los procedimientos nombrados anteriormente son conocidos como protocolos de actualización y de invalidación respectivamente [GRA03] [LAM79].

En un protocolo de actualización, cuando una unidad de procesamiento escribe un dato, las múltiples copias que se encuentran en el sistema son actualizadas. En cambio, utilizando un protocolo de invalidación, ante la misma situación, todas las copias restantes son marcadas como inválidas. Dada la situación en la que una unidad de procesamiento simplemente lee un dato una vez y nunca vuelve a usarlo. Utilizando un protocolo de actualización, las actualizaciones que se den a éste dato por parte de otras unidades de procesamiento provocan exceso de overhead y aumentan el tráfico en la red de interconexión. En cambio, al emplear un protocolo de invalidación, el dato es marcado como inválido ante la primera actualización remota y las siguientes actualizaciones ya no necesitan ser realizadas sobre ésta copia [GRA03]. En la Figura 2.4 se esquematiza el funcionamiento de ambos protocolos.

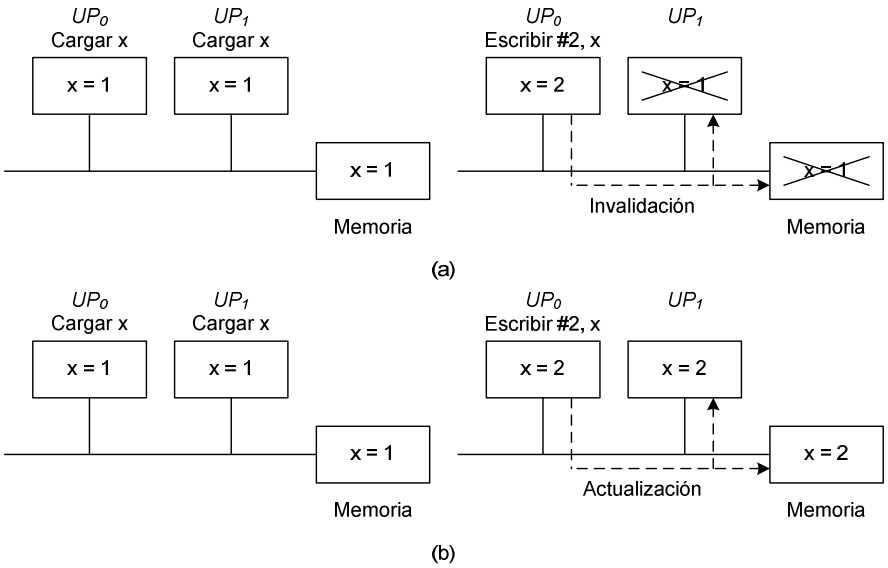

Figura 2.4. Coherencia de caché en máquinas de memoria compartida (a) Protocolo de invalidación; (b) Protocolo de actualización.

Otro factor que puede afectar el rendimiento de éstos protocolos es la situación denominada *false sharing*. La misma se da cuando las diferentes unidades de procesamiento realizan actualizaciones simultáneas de elementos individuales en la misma línea de caché. Empleando un protocolo de invalidación, cada actualización de un elemento individual en una línea de cache implica invalidar la línea completa (incluso cuando éstas actualizaciones son lógicamente independientes unas de otras). Si las otras unidades de procesamiento acceden a un elemento diferente de la misma

línea de cache, se verán forzados a buscar una copia del dato en la memoria principal aún cuando el elemento accedido no haya sido modificado. Esto se debe a que la coherencia de caché se mantiene por línea, y no por elementos individuales. Es fácil notar que false sharing puede causar que una línea de cache pase de una unidad de procesamiento a otro de forma sucesiva. Al utilizar un protocolo de actualización, la situación mejora levemente dado que todas las lecturas pueden realizarse de manera local mientras que las escrituras son las que requieren actualizaciones [GRA03] [SUN04].

La elección entre esquemas de invalidación y de actualización representa un costo-beneficio entre overhead en la comunicación (actualizaciones) y ociosidad (quedarse con los inválidos). La generación actual de máquinas suele adoptar protocolos de invalidación [GRA03].

2.4.1.1 Manteniendo la coherencia usando protocolos de invalidación

Las múltiples copias de un dato que pueden existir en una máquina paralela se mantienen consistentes almacenando información de las mismas. En esta sección de detalla un posible conjunto de estados para las copias junto a los eventos que disparan las transiciones entre ellos. Debe quedar claro que éste no es el único conjunto de estados y transiciones posibles, y que otros pueden definirse [GRA03].

Dadas dos unidades de procesamiento, UP_0 y UP_1, interconectadas a una memoria global a través de un bus trabajan con la misma variable. Inicialmente la variable (*x*) reside en memoria global. El primer paso llevado a cabo por ambas unidades de procesamiento consiste en llevar a *x* a las respectivas cachés. En este punto, el estado de la variable es *compartida*, dado que justamente es compartida por las múltiples unidades de procesamiento. Cuando UP_0 escribe un valor en esta variable, debe marcar todas las copias restantes como *inválidas*. Además debe marcar su propia copia como *sucia*. Estos pasos son requeridos para garantizar que todos los accesos subsiguientes a la variable *x* por parte de otras unidades de procesamiento sean servidos por la unidad de procesamiento UP_0 y no por la memoria. En éste punto, si UP_1 realiza otra lectura sobre la variable *x*. UP_1 intenta buscar la variable pero, como esta fue marcada como *sucia* por UP_0, justamente ella se encargar de satisfacer el pedido. Las copias de esta variable que se encuentran en la caché de UP_1 y en la memoria global son actualizadas y la variable vuelve al estado *compartida*. Existen 3 estados –*compartida*, *inválida* y *sucia*- a través de los cuales una línea de caché puede atravesar [GRA03].

En la Figura 2.5 se puede ver el diagrama de estados completo de un protocolo simple de tres estados. Las líneas sólidas indican acciones de las unidades de procesamiento mientras que las líneas discontinuas representan acciones de coherencia. Por ejemplo, si una unidad de procesamiento realiza una lectura sobre un bloque inválido, el bloque es buscado y se pasa de estado inválida a estado compartida. De forma similar, si una unidad de procesamiento escribe sobre un bloque compartido, el protocolo de coherencia propaga un C_escribir (escritura de coherencia) sobre el bloque. Esto dispara una transición desde el estado compartida a inválida para todos los otros bloques [GRA03].

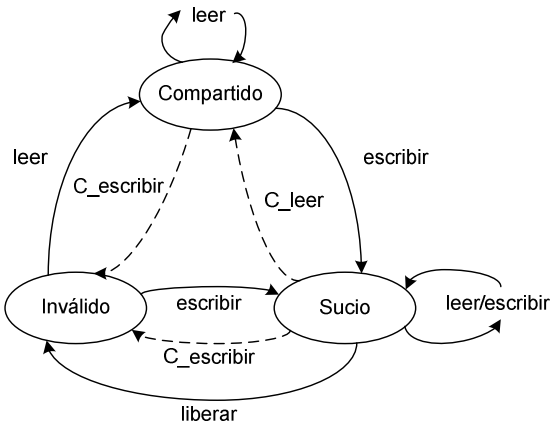

Figura 2.5. Diagrama de estados de un protocolo de coherencia simple de tres estados.

2.4.1.2 Sistemas de cache Snoopy

Los protocolos Snoopy le sientan bien a las máquinas de memoria compartida que utilizan una red de interconexión broadcast como un bus. Cada bloque de caché tiene un conjunto de estados posibles y existe un conjunto de transiciones definido para ellos. Para representar el estado de un bloque de cache se utiliza un conjunto de bits, el cual es actualizado de acuerdo al diagrama de estados asociado al protocolo de coherencia. Las unidades de procesamiento monitorizan el bus en búsqueda de operaciones que impliquen modificar el estado de los bloques. Por ejemplo, cuando el hardware snoop detecta que una operación de lectura fue realizada sobre un bloque de caché del cual tiene una copia marcada como *sucia*, toma el control de bus y coloca la copia para satisfacer el pedido. De forma similar, cuando el hardware snoop detecta que una operación de escritura fue realizada sobre un bloque de caché del cual mantiene una copia, procede a invalidar el bloque [GRA03].

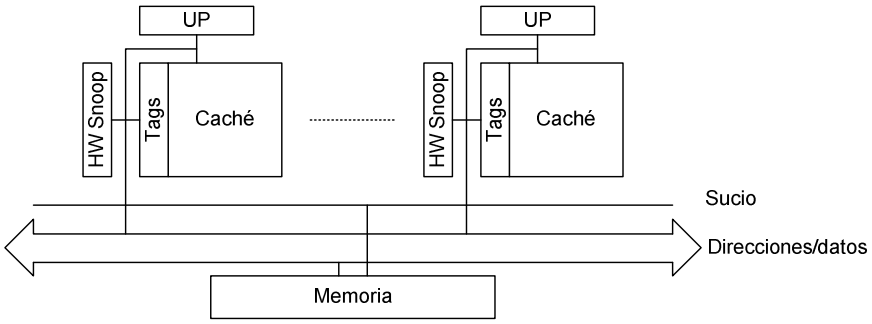

Figura 2.6. Un sistema de coherencia de caché Snoopy simple.

Los protocolos Snoopy son ampliamente utilizados en los sistemas comerciales debido a su simplicidad y a la mejora que se obtiene al incluirlos en los sistemas basados en bus existentes. Cuando distintas unidades de procesamiento utilizan diferentes datos, se obtienen beneficios al colocar estos datos en la caché. De esta forma, las unidades de procesamiento trabajan de forma local con los datos hasta que cada uno de ellos sea marcado como inválido, evitando tráfico externo. Por otro lado, si las distintas unidades de procesamiento realizan operaciones de lectura y escritura sobre el mismo dato, entonces se generan operaciones de coherencia que incrementan el tráfico del bus y pueden llegar a saturarlo. En consecuencia, sólo aquellos sistemas con una cantidad pequeña a media de unidades de procesamiento pueden admitir protocolos Snoopy [GRA03] [MOH02].

Los protocolos Snoopy no resultan ser una solución escalable ya que, como se mencionó anteriormente, al contar con un gran número de unidades procesamiento, el tráfico generado por las operaciones de coherencia puede llevar a saturar el bus. Una solución a éste problema consiste en propagar las operaciones de coherencia sólo a aquellas unidades de procesamiento involucradas en la misma (es decir, a aquellas que mantienen copias del dato). La solución requiere registrar qué unidades de procesamiento mantienen copias de los datos, así como también, los estados de las mismas. La información es almacenada en un directorio, y el mecanismo de coherencia basado en la misma se conoce como sistemas basados en directorio [GRA03] [MOH02].

2.4.1.3 Sistemas basados en directorio

Los sistemas basados en directorio agregan a la memoria principal un directorio que mantiene una matriz de bits que representan los bloques de cachés y las unidades de procesamiento en los cuales son cacheados. Las entradas de la matriz son llamadas *bits de presencia*. Al igual que antes, seguimos asumiendo un protocolo de 3 estados: *inválido*, *sucio* y *compartido*. El principal atractivo de los sistemas basados en directorio es el rendimiento que los mismos proporcionan. En este tipo de sistemas sólo aquellas unidades de procesamiento que mantienen un bloque particular (o que lo están leyendo) participan en las transiciones de estado provocadas por las operaciones de coherencia. Otras transiciones de estados podrían ser disparadas por operaciones de lectura, escritura o liberación (*flush*, retirar una línea de caché) pero estas transiciones pueden ser manejadas localmente manipulando los bits de presencia y el estado en el directorio [GRA03] [MOH02].

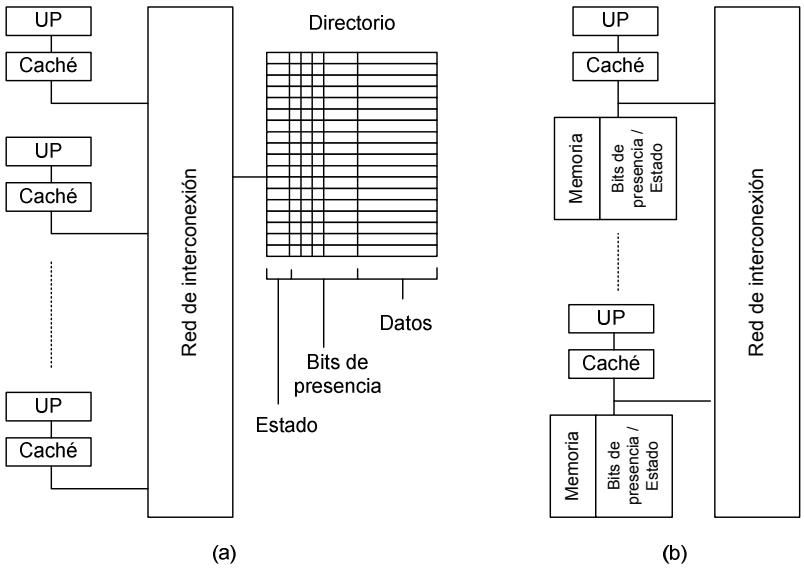

Figura 2.7. Arquitectura de un sistema basado en directorio típico: (a) un directorio centralizado; y (b) un directorio distribuido.

Volviendo al ejemplo de la Sección 2.4.1, el acceso al bloque donde se encuentra la variable *x* por parte de las unidades de procesamiento UP_0 y UP_1, implica cambiar el estado del bloque a *compartido* y actualizar los bits de presencia de manera de indicar que tanto UP_0 como UP_1 comparten el bloque. Cuando UP_0 escribe un nuevo valor sobre la variable, el estado en el directorio se cambia a *sucio* y el bit de presencia de UP_1 se resetea. Todas las operaciones siguientes que realice UP_0 con esta variable pueden ser realizadas localmente. El directorio, ante una operación de lectura de esta variable por parte de una unidad de procesamiento diferente a UP_0, observa que el bloque está marcado como *sucio* y utiliza los bits de presencia para

redirigir el pedido. UP_0 actualiza el bloque en memoria y lo envía a la unidad de procesamiento adecuada. Nuevamente los bits de presencia correspondientes son actualizados, al igual que el estado que pasa a ser *compartido* [GRA03].

Los esquemas basados en directorio se comportan de la misma forma que los protocolos Snoopy en determinadas situaciones. Si diferentes unidades de procesamiento trabajan con bloques de datos distintos, entonces estos bloques pasan a estar *sucios* en las cachés correspondientes y todas las operaciones siguientes a la primera pueden realizarse localmente. Tampoco se generan operaciones de coherencia cuando más de una unidad de procesamiento lee (pero no actualiza) un bloque de datos determinado, ya que el mismo es replicado en las cachés correspondientes con el estado *compartido* [GRA03].

Las operaciones de coherencia se presentan cuando múltiples unidades de procesamiento intentan actualizar el mismo dato. Se debe tener en cuenta que además de las transferencias de datos necesarias, las acciones de coherencia agregan overhead al tener que propagar las actualizaciones de estado (invalidaciones o actualizaciones) y generar información de estado desde el directorio. Como el directorio se encuentra en memoria principal y el sistema de memoria sólo puede dar servicio a un número limitado de operaciones de lectura/escritura por unidad de tiempo, la cantidad de actualizaciones de estado está limitada en última instancia por el directorio. Si un programa paralelo requiere un gran número de acciones de coherencia (gran número de operaciones de lectura/escritura sobre bloques de datos compartidos) entonces el directorio limitará su rendimiento paralelo [GRA03].

Finalmente, se debe recordar que el directorio se mantiene en memoria y que el tamaño del mismo crece con orden $O(mp)$, siendo m la cantidad de bloques de memoria y p el número de unidades de procesamiento. Luego, la cantidad de memoria requerida para almacenar el directorio podría volverse un cuello de botella en sí mismo si el número de unidades de procesamiento se incrementa. Una solución a éste problema podría ser aumentar el tamaño de los bloques de memoria (y de esta forma reducir m). Sin embargo, esto podría causar la ocurrencia del fenómeno false sharing descripto anteriormente y producir los problemas que el mismo acarrea [GRA03].

Dado que el directorio es el centro de la competencia, es razonable dividir la tarea de mantener la coherencia a través de las múltiples unidades de procesamiento. Básicamente, cada unidad de procesamiento debe mantener la coherencia de sus propios bloques de memoria, asumiendo que existe un particionamiento físico (o lógico) de los bloques de memoria a través de los procesadores. Los esquemas que utilizan esta metodología se conocen como sistemas basados en directorios distribuidos [GRA03].

En arquitecturas escalables, la memoria se encuentra físicamente distribuida entre las diferentes unidades de procesamiento. Luego, los correspondientes bits de presencia de los bloques también lo están. Cada unidad de procesamiento es responsable de mantener la coherencia de sus propios bloques. Como cada bloque de memoria tiene un dueño (el cual puede ser típicamente calculado a partir la dirección del bloque), la ubicación del directorio es conocida implícitamente por todas las unidades de procesamiento. Cuando una unidad de procesamiento quiere leer un bloque por primera vez, debe pedírselo al dueño. El mismo recibe el pedido y lo satisface de acuerdo a los bits de presencia y la información de estado disponible localmente. De forma similar, cuando una unidad de procesamiento escribe sobre un bloque de memoria, envía a continuación un mensaje de invalidación al dueño, el cual finalmente propaga el mensaje a todas las unidades de procesamiento que mantienen una copia del bloque en sus cachés. De esta manera, el directorio se encuentra descentralizado y la competencia asociada al directorio centralizado se ve aliviada.

Aún así, se debe notar que el overhead provocado por la comunicación de las actualizaciones de estado se mantiene [GRA03].

Queda evidenciado que los esquemas de directorio distribuido permiten *O(p)* operaciones de coherencia simultáneas, siempre y cuando la red de interconexión pueda sostener los mensajes asociados a las actualizaciones de estados. Desde éste punto de vista, los directorios distribuidos son inherentemente más escalables que los sistemas Snoopy o los directorios centralizados. La latencia y el ancho de banda se convierten en cuellos de botellas para el rendimiento de estos sistemas [GRA03].

2.4.2 Multicores

Históricamente se ha buscado incrementar el poder computacional de los sistemas de computación. En 1965, Gordon Moore, uno de los fundadores de Intel, afirmó que el número de transistores en un circuito integrado se duplicaría cada 18 meses. Aunque parezca increíble esta afirmación se mantiene en vigencia y es lo que conocemos como Ley de Moore. Sin embargo, hoy en día resulta cada vez más difícil acelerar la velocidad de los procesadores incrementando la frecuencia de los mismos. Son dos los problemas que los arquitectos de hardware deben enfrentar. El primero es la generación de calor y el segundo es el consumo de energía. La solución que presentaron a estos problemas ha sido integrar dos o más núcleos computaciones dentro de un mismo chip, lo cual se conoce como procesador *multicore* o *multinúcleo*. Los procesadores multicore mejoran el rendimiento de una aplicación al distribuir el trabajo entre los núcleos disponibles [CHA07] [MCC07].

Existen diferentes alternativas con respecto a la organización de la jerarquía de caché. Normalmente, cada uno de los núcleos de un multicore posee su propio nivel L1 de caché y comparte de a pares el nivel L2.

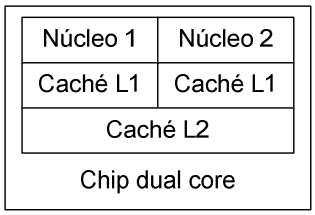

Figura 2.8. Arquitectura de procesador dual core o doble núcleo.

2.5 Máquinas de memoria combinada: distribuida y compartida

2.5.1 Cluster de multicores

La computación con clusters ha demostrado ser uno de los modelos de programación paralela más populares de las últimas décadas. El alto poder de procesamiento, el bajo costo y la facilidad con que pueden ser expandidos, convierten a los clusters en una alternativa formidable. La aparición de la tecnología multicore ha impactado sobre los clusters, introduciéndolos en una nueva etapa. Un cluster de multicores es similar a un cluster tradicional, sólo que en lugar de monoprocesadores tenemos procesadores multicore [CHA07].

Los clusters de multicores agregan un nivel más a la jerarquía de memoria de los clusters tradicionales. Además de la caché compartida entre pares de núcleos y la memoria compartida entre todos los núcleos de un mismo procesador, contamos con

la memoria distribuida accesible vía red. Un esquema de este tipo de arquitectura se muestra en la Figura 2.9.

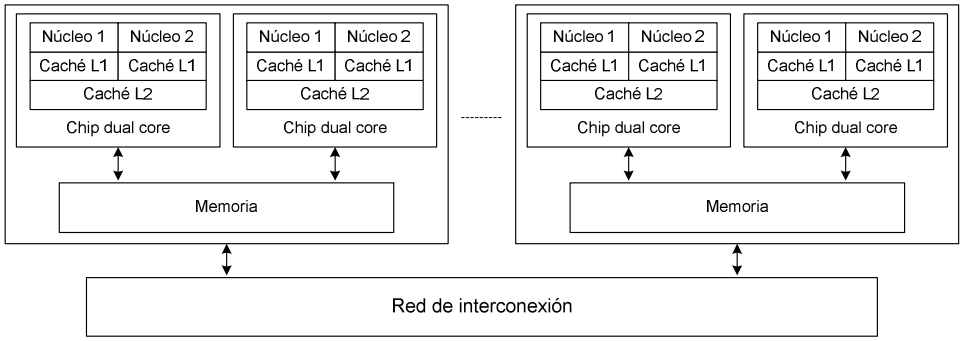

Figura 2.9. Una posible arquitectura cluster de multicores.

A la hora de escribir un algoritmo paralelo, no puede obviarse la jerarquía de memoria presente en los clusters de multicores, ya que esta incide sobre el rendimiento alcanzable por los mismos. Es por ello que se han estudiado nuevas técnicas de programación de algoritmos paralelos con el objetivo de aprovechar de manera eficiente la potencia de estas arquitecturas. Entre ellas, se encuentran distintas alternativas que combinan pasaje de mensajes con memoria compartida. Además se cuenta con el tradicional paradigma de pasaje de mensajes. Se debe notar que el paradigma de memoria compartida no puede utilizarse, ya que justamente no tenemos una memoria compartida entre nodos del cluster.

Capítulo 3

Evaluación de sistemas paralelos

3.1 Fuentes de overhead en programas paralelos

Duplicando los recursos de hardware, se espera razonablemente que un programa se ejecute dos veces más rápido. Sin embargo, esto no suele ocurrir en los programas paralelos debido a que los mismos incurren en diferentes overheads asociados a la administración del paralelismo.

Además del cómputo asociado al problema a resolver, un programa paralelo puede dedicar tiempo a la comunicación entre procesos, al ocio, y al exceso de computación (computación que no es realizada por la implementación secuencial) [GRA03].

3.1.1 Interacción entre procesos

Cualquier implementación paralela requiere que sus unidades de procesamiento interactúen y comuniquen datos (por ejemplo, resultados intermedios). La interacción entre procesos es quizás la fuente más significativa de overhead en el procesamiento paralelo.

3.1.2 Ocio

Las unidades de procesamiento en un sistema paralelo pueden estar ociosas debido a diferentes razones como desbalance de carga, sincronización, y presencia de código serial en un programa.

Si diferentes unidades de procesamiento tienen distintas cargas de trabajo, entonces algunas de ellas pueden estar ociosas mientras otras trabajan sobre el problema. Lo anterior puede ocurrir, por ejemplo, cuando la generación de tareas es dinámica. En dicho caso, se hace difícil estimar el tamaño de las subtareas asignadas a las unidades de procesamiento, lo cual puede causar que las cargas de trabajo no sean uniformes.

En algunos programas paralelos, las unidades de procesamiento deben sincronizar en determinados puntos de la ejecución. Si no todas están listas al mismo tiempo, entonces las que lo estén primero estarán ociosas hasta que el resto lo haga.

Un programa paralelo podría contener partes que deben ser ejecutadas secuencialmente por una unidad de procesamiento. Durante ese cómputo secuencial, las otras unidades de procesamiento deben esperar.

3.1.3 Exceso de computación

El mejor algoritmo secuencial para resolver un problema puede ser difícil o imposible de paralelizar, lo cual obliga a utilizar un algoritmo paralelo basado en una

solución secuencial más pobre pero que garantice un grado de concurrencia más alto. Encontramos overhead en el exceso de computación realizada por el programa paralelo con respecto al mejor programa secuencial.

También puede ocurrir que un programa paralelo debe realizar más cómputos que la versión secuencial debido a que los mismos permiten evitar intercambios de datos entre las unidades de procesamiento. Esto implica que algunos cálculos sean realizados múltiples veces por diferentes unidades de procesamiento.

3.2 Métricas de rendimiento para sistemas paralelos

Es importante estudiar el rendimiento de programas paralelos con el fin de determinar el mejor algoritmo, evaluar plataformas de hardware, y examinar los beneficios del paralelismo. Para ello se introduce una serie de métricas que permitirán realizar el análisis deseado.

3.2.1 Tiempo de ejecución

El tiempo de ejecución secuencial de un programa es el tiempo que pasa entre el inicio y el fin de su ejecución en una computadora con una única unidad de procesamiento. El tiempo de ejecución paralelo es el tiempo que transcurre desde el momento en que el procesamiento paralelo comienza hasta que el último elemento de procesamiento finaliza su ejecución. Se expresa el tiempo de ejecución secuencial como T_s y el tiempo de ejecución paralelo como T_p [GRA03].

3.2.2 Speedup

El *speedup* refleja el beneficio relativo de paralelizar la solución a un problema. Se define como el cociente entre el tiempo requerido por la solución secuencial utilizando una única unidad de procesamiento y el tiempo requerido por la solución paralela de interés utilizando más de una unidad de procesamiento. Denotamos al speedup como *S(p)*, siendo *p* el número de unidades de procesamiento utilizadas. La Ecuación 3.1 muestra esta métrica.

$$S(p) = \frac{T_s}{T_p} \qquad (3.1)$$

En otras palabras, el speedup indica cuántas veces más rápido se obtiene la solución al problema utilizando la solución paralela con *p* unidades de procesamiento con respecto a la solución secuencial utilizando una única unidad de procesamiento.

Cuando hacemos referencia a la solución secuencial se debe tener en cuenta que puede existir más de un algoritmo que resuelva el problema. Se debe elegir el algoritmo secuencial que soluciona el problema en la menor cantidad de tiempo. De otra manera no es justa la comparación con el algoritmo paralelo [GRA03].

3.2.2.1 ¿Cuál es el máximo speedup alcanzable?

En teoría, el máximo *speedup* alcanzable con *p* unidades de procesamiento es *p*, lo que se conoce como *speedup lineal*. Para ello cada unidad de procesamiento debería tardar T_s/p unidades de tiempo, siendo T_s el tiempo requerido por el mejor algoritmo secuencial para resolver el problema. Para obtener un *speedup* superior a *p*,

es necesario que cada unidad de procesamiento tarde menos de T_s/p unidades de tiempo. En ese caso, una única unidad de procesamiento podría emular las p unidades de procesamiento y resolver el problema en menos de T_s unidades de tiempo. Esto es una contradicción dado que el *speedup*, por definición, es calculado a partir del mejor algoritmo secuencial. Si T_s es el tiempo requerido por el mejor algoritmo secuencial, entonces el problema no puede ser resuelto en menos de T_s utilizando una única unidad de procesamiento [GRA03].

$$S(p) \leq \frac{T_s}{T_s/p} = p \qquad (3.2)$$

En la práctica, puede observarse un speedup superior a p, el cual es llamado *speedup superlineal*. Esto puede deberse a que el trabajo que debe realizar el algoritmo secuencial es mayor que el correspondiente al algoritmo paralelo. Por ejemplo, al emplear una descomposición exploratoria en la búsqueda de un nodo en un árbol binario. La descomposición exploratoria se describe en la Sección 4.1.3. Se supone que cada nodo tiene una etiqueta asociada y el objetivo es encontrar aquel que tiene una etiqueta específica, en este caso 'X'. En el árbol de la Figura 3.1, dicho nodo es la hoja que se encuentra más a la derecha.

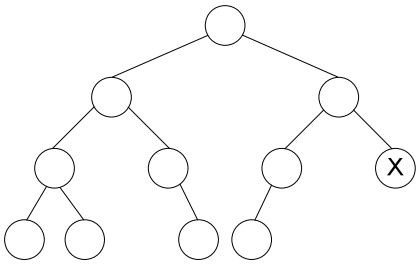

Figura 3.1. Árbol binario.

Una implementación secuencial que emplee un recorrido en profundidad explorará los 11 nodos del árbol. Si visitar un nodo toma un tiempo t_v, entonces el tiempo que requiere el recorrido es $11t_v$. Dada una implementación paralela a ser ejecutada sobre dos unidades de procesamiento, una podría encargarse del subárbol izquierdo mientras que la otra hacer lo correspondiente con el subárbol derecho. Ambas soluciones se muestran en la Figura 3.2.

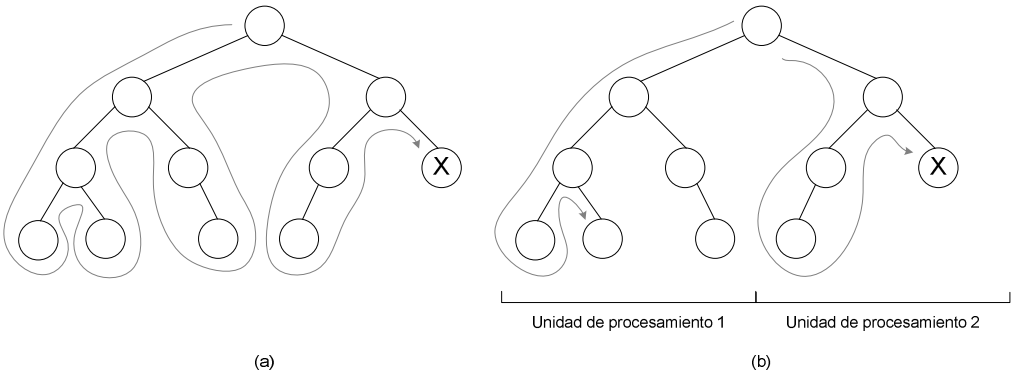

Figura 3.2. Búsqueda de un nodo en un árbol binario. (a) Implementación secuencial. (b) Implementación paralela empleando dos unidades de procesamiento.

Si ambas unidades de procesamiento exploran el árbol a la misma velocidad, la implementación paralela explorará menos nodos que la secuencial. Se debe notar que el algoritmo paralelo visita sólo 9 nodos, por lo que el trabajo total paralelo es $9t_v$. Si la visita al nodo raíz se realiza de manera secuencial entonces el tiempo paralelo es $5t_v$ (la vista al nodo raíz seguida de las cuatro visitas realizadas por las unidades de procesamiento). Por lo tanto, el speedup conseguido con dos unidades de procesamiento es $11t_v/5t_v = 2,2$.

La superlinealidad ocurre por la diferencia en el trabajo que realiza cada una de las soluciones. Se debe notar que al utilizar la descomposición exploratoria, la cantidad de trabajo correspondiente al algoritmo secuencial y al paralelo depende de la ubicación de la solución, y que ocasionalmente no es posible encontrar un algoritmo serial que sea óptimo en todas las situaciones.

También se puede obtener speedup superlineal debido a características de hardware que ubican en desventaja a la implementación secuencial. Por ejemplo, si se tiene que la memoria principal asociada a cada unidad de procesamiento en la computadora paralela es la misma que la de la unidad de procesamiento en la computadora secuencial. La cantidad de datos para un problema podría ser muy grande para caber en la computadora secuencial, lo cual implica tráfico a disco. Ahora, al dividir los datos entre las distintas unidades de procesamiento de la computadora paralela, podríamos obtener particiones lo suficientemente pequeñas como para caber en la memoria principal de cada procesador. Esto lleva a un mejor aprovechamiento del sistema de memoria y por ende a obtener un mejor rendimiento general. En estos casos, se debe analizar la posibilidad de que el algoritmo secuencial quizás no sea el mejor y pueda ser optimizado modificando el patrón de acceso a los datos o la manera en que los mismos son alocados [GRA03] [WIL05].

3.2.3 Eficiencia

La *eficiencia* es una medida que refleja cuan bien utilizadas son las unidades de procesamiento para resolver el problema. Se define como el cociente entre el speedup y p, siendo éste el número de unidades de procesamiento utilizados, y se denota mediante la letra E, como se indica en la Ecuación 3.3.

$$E = \frac{S(p)}{p} \tag{3.3}$$

En la teoría, se puede obtener una eficiencia igual a 1 si el speedup obtenido con p unidades de procesamiento es igual a p. En la práctica, debido a los costos que implican la sincronización y la comunicación, el speedup suele ser menor a p. Esto lleva a la eficiencia a ser menor a 1.

3.2.4 Ley de Amdahl

Comúnmente algunas partes del cómputo de un programa paralelo no pueden ser divididas entre los procesos intervinientes, por lo que deben ser ejecutadas en forma secuencial. Por ejemplo, algún tipo de inicialización previa al procesamiento paralelo. Al momento de ejecutar estas partes secuenciales, una única unidad de procesamiento se encontrará ocupada mientras el resto estarán ociosas, lo cual causa overhead como se vio anteriormente [WIL05].

Asumiendo que un programa paralelo tendrá partes donde sólo una unidad de procesamiento estará ocupada (partes seriales) y que durante el resto de las partes

todas las unidades de procesamiento se encontrarán ocupadas. Si la fracción del programa que no puede ejecutarse en forma concurrente es *f*, y las partes concurrentes son ejecutadas por las unidades de procesamiento sin incurrir en overhead, entonces el tiempo requerido por el programa paralelo utilizando *p* unidades de procesamiento está dado por $fT_s + (1 - f)\ T_s\ /p$. Luego, el speedup está dado por la Ecuación 3.4.

$$S(p)=\frac{T_s}{fT_s+(1-f)T_s/p}=\frac{p}{1+(p-1)f} \qquad (3.4)$$

Esta ecuación es conocida como *Ley de Amdahl* y dice que la mejora obtenida por un modo de ejecución más rápido está limitada por el tiempo en que ese modo se ejecuta. En otras palabras, se necesita que las partes seriales de un programa paralelo (en nuestro caso *f*) representen una fracción pequeña de la computación total si el objetivo es obtener un incremento significativo en la velocidad de procesamiento. Aún con un número infinito de unidades de procesamiento, el speedup máximo está limitado a 1/*f*, como se ve en la Ecuación 3.5.

$$\lim_{p\to\infty} S(p)=\frac{1}{f} \qquad (3.5)$$

Por ejemplo, con sólo un 5% de computación serial, el máximo speedup es 20, sin importar que cantidad de unidades de procesamiento se estén utilizando [AND00] [WIL05].

3.2.5 Escalabilidad

Dentro del área del procesamiento paralelo, el término *escalabilidad* es utilizado para hacer referencia a diferentes comportamientos. Por ejemplo, es usado para indicar aquellos diseños de hardware que permiten al sistema mejorar su rendimiento al incrementar su tamaño. Esta podría ser descrita como *escalabilidad arquitectónica* o *de hardware*. También puede ser utilizado para indicar aquellos algoritmos paralelos que ante un incremento del tamaño del problema, aumentan la cantidad de trabajo de forma razonable. Esta podría ser descrita como *escalabilidad algorítmica* [WIL05].

Es deseable que todos los sistemas paralelos sean arquitectónicamente escalables, aunque esto depende del diseño de hardware. Normalmente, agregar unidades de procesamiento a un sistema paralelo implica también expandir la red de interconexión. Si se piensa en un sistema de memoria compartida donde todas las unidades de procesamiento se encuentran interconectadas a través de un bus, adicionar nuevas unidades de procesamiento, aumentaría el tráfico en el mismo y podría llegar a saturarlo. Como resultado, la eficiencia del sistema se reduce. El objetivo de la mayoría de los diseños de hardware es lograr escalabilidad, y eso se ve reflejado en la gran cantidad de redes de interconexión que han sido diseñadas [WIL05].

Contar con escalabilidad arquitectónica y algorítmica permite pensar que un incremento en el tamaño del problema puede ser asimilado con un aumento del tamaño del sistema para una arquitectura y un algoritmo en particular. Cuando se habla de un aumento en el tamaño del sistema, queda claro que se hace referencia a agregar más unidades de procesamiento. Ahora, incrementar el tamaño del problema

requiere mayor precisión. Inicialmente, se podría pensar en la cantidad de elementos a procesar como una medida de tamaño. Sin embargo, doblar el tamaño del problema no necesariamente dobla la cantidad de trabajo. Esto depende de la complejidad del algoritmo. Por ejemplo, sumar dos matrices, tiene el efecto mencionado anteriormente, pero multiplicarlas no. La cantidad de trabajo para multiplicar matrices se cuadruplica. Por lo tanto, la cantidad de trabajo a realizar depende del problema a resolver [WIL05].

3.2.6 Ley de Gustafson-Barsis

Basándose en los conceptos de escalabilidad y en una ecuación de E. Barsis, John Gustafson presentó argumentos que demostraban la falta de realismo de la Ley de Amdahl. Gustafson observó que, en la práctica, usualmente el tamaño del problema se elige a partir de la cantidad de unidades de procesamiento disponibles. Por lo tanto, asumir que el tamaño del problema es fijo (Ley de Amdahl) es tan válido como asumir que el tiempo de ejecución paralela también lo es. A medida que se incrementa el número de unidades de procesamiento, también se aumenta el tamaño del problema de manera de mantener constante el tiempo de ejecución paralela. Al incrementar el tamaño del problema, Gustafson también notó que la parte secuencial del código normalmente es fija y no crece con el tamaño del problema [GUS88] [WIL05].

Al considerar que el tiempo de ejecución paralela es constante, el speedup resultante se calcula de forma diferente al speedup de Amdahl y se conoce como *speedup escalado*. La fórmula del speedup de Gustafson utiliza los mismos términos que la Ley de Amdahl, aunque es necesario separar la partes serial y paralelizable del tiempo de ejecución secuencial T_s. Luego, $T_s = fT_s + (1 - f)T_s$, donde fT_s es una constante. Por simplicidad algebraica, se establece que $T_p = fT_s + (1 - f)T_s / p = 1$. A continuación se manipula algebraicamente la igualdad anterior para definir el tiempo de ejecución secuencial como $T_s = fT_s + (1 - f)T_s = p + (1 - p)fT_s$. Finalmente, el speedup escalado se define por la Ecuación 3.6.

$$S_s(p) = \frac{fT_s + (1-f)T_s}{fT_s + (1-f)T_s/p} = \frac{p + (1-p)fT_s}{1} = p + (1-p)fT_s \qquad (3.6)$$

La ecuación 3.6 se conoce como *Ley de Gustafson* y mantiene dos suposiciones: la primera es que el tiempo de ejecución paralela es constante, mientras que la segunda es que la parte que se ejecuta secuencialmente, fT_s, es también constante y no es función de p. Por ejemplo, si la parte serial representa el 5% y se cuenta con 20 unidades de procesamiento, entonces el speedup según Gustafson sería 19.05, en lugar de 10.26 de acuerdo a la fórmula de Amdahl [WIL05].

Capítulo 4

Principios para el diseño de algoritmos paralelos

A la hora de resolver problemas utilizando computadoras resulta muy importante el desarrollo de algoritmos. Un algoritmo secuencial es esencialmente una secuencia de pasos básicos para resolver un problema determinado utilizado una computadora serial. De forma similar, un algoritmo paralelo es una receta para resolver un problema dado usando múltiples unidades de procesamiento. Sin embargo, la especificación de un algoritmo paralelo involucra otras tareas además de la especificación de los pasos a ejecutar. En la práctica, se pueden encontrar algunas o todas las tareas siguientes:

- Identificación de las porciones de trabajo que pueden ser realizadas concurrentemente.
- Asignación de las diferentes porciones de trabajo identificadas en el paso anterior a los procesos que se ejecutan en paralelo.
- Distribución de los datos de entrada, salida e intermedios asociados con el programa.
- Administración del acceso a los datos compartidos por las múltiples unidades de procesamiento.
- Sincronización de los distintos procesos en diferentes etapas de la ejecución del programa paralelo.

Típicamente, existen varias opciones para cada uno de los pasos descritos anteriormente. Aún así, relativamente pocas combinaciones de estas opciones llevan a un algoritmo paralelo a alcanzar un rendimiento acorde a los recursos computacionales y de almacenamiento empleados para resolver el problema. A veces, diferentes opciones alcanzan el mejor rendimiento sobre diferentes arquitecturas paralelas o bajo distintos paradigmas de programación paralela [GRA03].

4.1 Técnicas de descomposición

Uno de los pasos fundamentales en el desarrollo de algoritmos paralelos es la división del trabajo en un conjunto de tareas que puedan ejecutarse en forma concurrente. No existe una única técnica que permita descomponer un problema. La elección de la misma está supeditada a las características propias del problema y a la elección del diseñador del algoritmo. Se debe tener en cuenta que una descomposición dada no siempre permite alcanzar el mejor algoritmo paralelo para un problema determinado.

Se puede clasificar a las técnicas en las siguientes categorías: descomposición recursiva, descomposición de datos, descomposición exploratoria y descomposición especulativa. Las dos primeras técnicas son de propósito general y se pueden aplicar

a una gran variedad de problemas. Por otro lado, las dos últimas tienen por naturaleza un propósito más especial, es por eso que se las suele aplicar en clases específicas de problemas [GRA03].

4.1.1 Descomposición recursiva

La descomposición recursiva es un método que permite proveer de concurrencia a aquellos problemas que pueden resolverse utilizando la estrategia *divide-y-vencerás.* Esta técnica divide al problema en un conjunto de subproblemas que son similares al problema original pero más pequeños en tamaño, resuelve los subproblemas recursivamente, y luego combina estas soluciones parciales para obtener una solución al problema original. La estrategia divide-y-vencerás provee concurrencia de manera natural, ya que los diferentes subproblemas pueden ser resueltos en forma simultánea [COR90].

4.1.1.1 Quicksort

Dado el problema de ordenar una secuencia *A* de *n* elementos. El algoritmo quicksort es un algoritmo de ordenación que emplea la estrategia divide-y-vencerás, el cual comienza eligiendo un elemento pívot *x* para particionar la secuencia *A* en dos subsecuencias A_0 y A_1, tal que todos los elementos en A_0 son menores que *x* y todos los elementos en A_1 son mayores o iguales a *x*. Cada una de las subsecuencias A_0 y A_1 es ordenada invocando en forma recursiva a quicksort. Cada uno de estos llamados recursivos vuelve a particionar las secuencias. La recursión finaliza cuando cada subsecuencia contiene un solo elemento. Se puede observar como trabaja quicksort en la Figura 4.1.

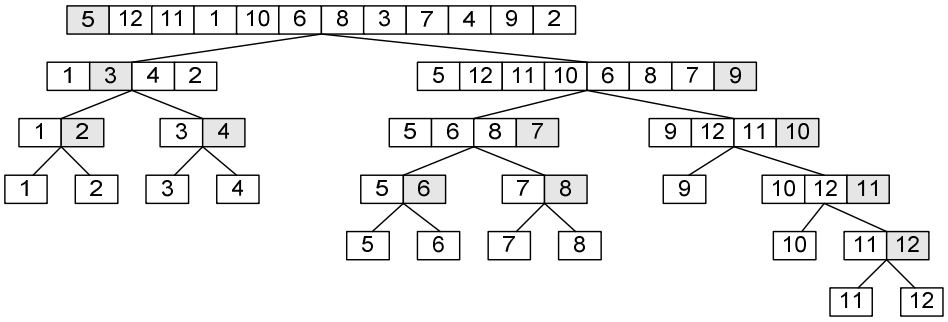

Figura 4.1. El grafo de dependencia de tareas del algoritmo quicksort basado en una descomposición recursiva para ordenar una secuencia de 12 números.

En la Figura 4.1, se define una tarea como el trabajo de particionar una determinada secuencia. Inicialmente hay una sola secuencia, por lo que usamos sólo un proceso para particionarla. El resultado de la partición anterior son dos subsecuencias (A_0 y A_1) y cada una de ellas puede ser particionada en paralelo. A medida que se desciende en el árbol, la concurrencia aumenta.

4.1.2 Descomposición de datos

La descomposición de datos es un método potente y comúnmente utilizado para derivar concurrencia en aquellos algoritmos que trabajan sobre grandes estructuras de datos. En esta técnica, la descomposición se realiza en dos pasos. En el primer paso, se particionan los datos sobre los cuales se opera, mientras que en el segundo paso, se utiliza el particionamiento anterior para inducir una partición del trabajo a realizar en

tareas. Por lo general, las tareas realizan el mismo trabajo o trabajos similares sobre las diferentes particiones de datos.

El particionamiento de los datos puede realizarse de diferentes formas. Resulta necesario evaluar todas las posibles maneras de particionar los datos para obtener una descomposición natural y eficiente del problema.

4.1.2.1 Particionamiento de los datos de salida

Esta técnica puede utilizarse en aquellos cómputos donde cada uno de los valores de salida es función de los valores de entrada y además puede calcularse en forma independiente del resto. La descomposición del problema en tareas se obtiene al particionar los datos de salida. Cada tarea recibe una partición y es responsable de calcular los valores de la misma.

4.1.2.2 Multiplicación de matrices

Un ejemplo característico es el problema de multiplicar dos matrices *A* y *B* de *nxn* para obtener una matriz *C* de igual dimensión. La Figura 4.2 muestra una descomposición del problema en cuatro tareas. Se considera que cada matriz está compuesta de cuatro bloques o submatrices definidas al dividir cada dimensión de la matriz en mitades. Los cuatro bloques de *C*, de tamaño *n/2* x *n/2*, son calculados independientemente por cuatro tareas como la suma de los productos adecuados de las submatrices de *A* y *B*.

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,1} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,1} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,1} \end{pmatrix}$$

(a)

Tarea 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Tarea 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Tarea 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Tarea 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

Figura 4.2. (a) Partición de las matrices de entrada y de salida en submatrices de 2 x 2. (b) Una descomposición de la multiplicación de matrices en cuatro tareas basada en la partición de las matrices en (a).

4.1.2.3 Particionamiento de los datos de entrada

El particionamiento de los datos de salida sólo puede realizarse cuando los cálculos de esos valores son función de los datos de entrada. En algunos algoritmos, no es posible o deseable particionar los datos de salida. Por ejemplo, en el caso de buscar el mínimo, el máximo o la suma de un conjunto de números, la salida es un valor individual desconocido. En estos casos, se pueden particionar los datos de entrada para obtener concurrencia. Por cada partición de los datos de entrada se crea una tarea, la cual es responsable de realizar todos los cálculos posibles sobre esa porción de datos. En general el trabajo realizado por las tareas quizás no solucione el problema original en forma directa. En esos casos, una fase final de cómputo permite combinar los resultados. Por ejemplo, para calcular el valor máximo de un conjunto de *N* números utilizando *p* procesos (*N* >> *p*). Se puede particionar la entrada en *p* subconjuntos de tamaño similar. Cada tarea computa el valor máximo entre los valores

del subconjunto que le es asignado. Al finalizar, se comparan los *p* resultados parciales para obtener el máximo total.

4.1.2.4 Particionamiento de los datos de entrada y de salida

En algunos casos, en los cuales es posible particionar los datos de salida, particionar los datos de entrada además puede ofrecer concurrencia adicional.

4.1.2.5 Particionamiento de los datos intermedios

Los algoritmos son a menudo estructurados como cómputos de múltiples etapas, donde la salida de una etapa es entrada para la siguiente. Una posible descomposición de ese algoritmo puede ser derivada al particionar los datos de entrada o de salida de una etapa intermedia del algoritmo. Éste método de particionamiento puede, de vez en cuando, conducir a un grado de concurrencia mayor comparado al particionamiento de los datos de entrada o de salida. No siempre los datos intermedios son generados explícitamente en el algoritmo secuencial para resolver el problema, lo cual conlleva a reestructurar el algoritmo original para poder aplicar éste método.

4.1.2.6 La regla “el dueño computa”

A una descomposición de datos que particiona los datos de entrada o los de salida también se la conoce como la regla *el dueño computa*. Esta regla supone que cada tarea es responsable de realizar todos los cálculos asociados a la partición que le es asignada. El significado de esta regla depende del tipo de descomposición de datos aplicado. Por ejemplo, cuado se particionan los datos de entrada, la regla del dueño computa significa que una tarea debe realizar todos los cálculos que pueda con los datos que le fueron asignados. En cambio, si la partición se realiza a los datos de salida, entonces la regla significa que una tarea debe realizar todos los cálculos para obtener los resultados de la partición que le fue asignada.

4.1.3 Descomposición exploratoria

La *descomposición exploratoria* es utilizada para descomponer problemas cuya solución se logra realizando una búsqueda en un espacio de posibles soluciones. El primer paso al utilizar esta técnica consiste en particionar el espacio de soluciones en partes más pequeñas. Luego se realiza una búsqueda en cada partición de forma concurrente hasta que la solución deseada sea encontrada. Por ejemplo, se puede utilizar la descomposición exploratoria para resolver el problema del juego Ta-Te-Tí.

4.1.3.1 El juego Ta-Te-Tí

El Ta-Te-Tí es un juego de lápiz y papel entre dos jugadores: O y X, que marcan los espacios de un tablero de 3×3 alternadamente. Un jugador gana si consigue tener una línea de tres con sus símbolos: la línea puede ser horizontal, vertical o diagonal. Si ningún jugador logra marcar una línea ganadora, el juego se considera empatado. En la Figura 4.3 se muestra un ejemplo de este juego.

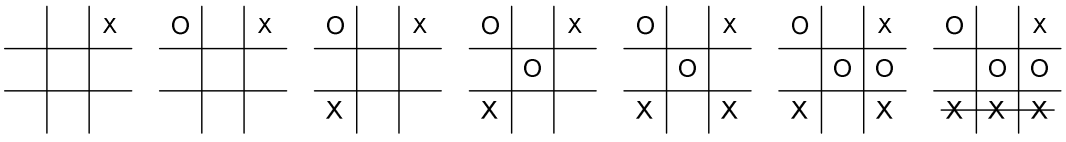

Figura 4.3. Partida de Ta-Te-Tí en la que gana el jugador X.

El Ta-Te-Tí puede resolverse utilizando una búsqueda en un árbol. Cada nodo del árbol representará una posible configuración del tablero y cada arista del árbol une toda configuración que puede ser alcanzada por otra mediante un movimiento. A partir de una configuración inicial, se generan todas las posibles configuraciones sucesoras. Una configuración puede tener hasta 8 posibles sucesoras, de acuerdo al número de espacios libres del tablero. La tarea de encontrar un camino desde la configuración inicial a la configuración final ahora se traduce a encontrar un camino desde alguna de estas configuraciones recientemente generadas a la configuración final. Como cada una de estas configuraciones está más cerca de la solución por un movimiento, se avanza en la búsqueda de una solución.

Una manera de resolver éste problema en paralelo es la siguiente: primero, a partir de la configuración inicial se generan todas las posibles configuraciones sucesoras. Luego, cada una de estas configuraciones se asocia a una tarea, la cual deberá continuar la búsqueda a partir de la configuración asignada. Se debe tener en cuenta que cada tarea podría generar a su vez otras tareas hijas. Cuando una tarea encuentra una solución, la envía a su tarea padre, la cual elegirá el mejor movimiento de todos los enviados por sus tareas hijas.

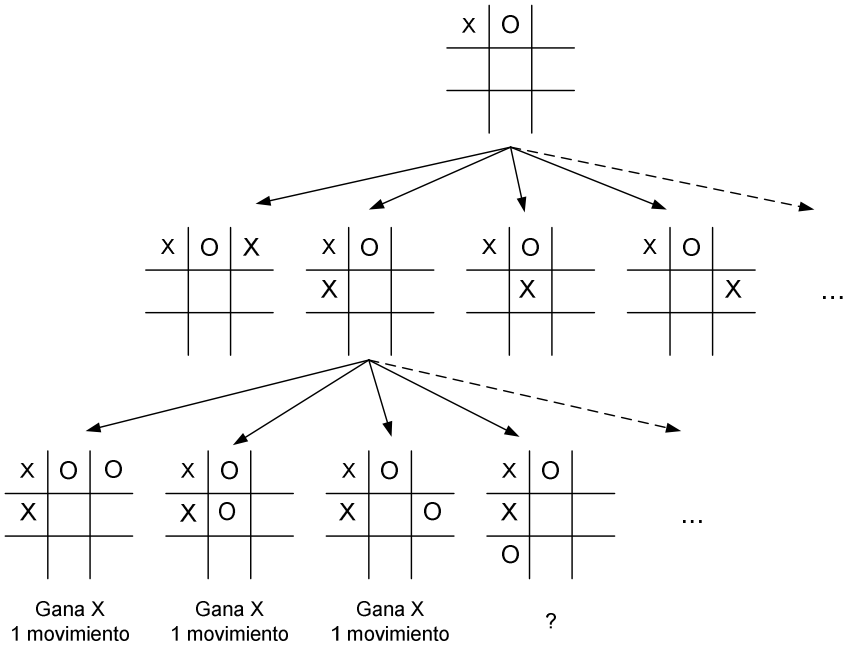

Figura 4.4. Descomposición exploratoria para el juego Ta-Te-Tí.

Si se ve al espacio de búsqueda como los datos a ser particionados, se podría decir que la descomposición exploratoria es similar a la descomposición de datos. Sin embargo, ambas técnicas de descomposición son fundamentalmente diferentes en la cantidad de trabajo que asignan a las tareas que generan. Las tareas producidas por la descomposición de datos se ejecutan en forma completa y cada una de ellas realiza trabajo valioso en función de alcanzar la solución al problema. Por otro lado, en la descomposición exploratoria, una tarea podría finalizar antes de concluir su trabajo en caso que una solución general ya haya sido encontrada. Luego, la porción del espacio de búsqueda recorrido (y la cantidad total de trabajo) por un algoritmo paralelo podría ser muy diferente al recorrido por el algoritmo secuencial. El trabajo realizado por la solución paralela puede ser menor o mayor comparado a la solución secuencial. Por ejemplo, al considerar un espacio de búsqueda que se ha particionado en cuatro tareas concurrentes como muestra la Figura 4.5. Si la solución se encuentra al principio del espacio de búsqueda correspondiente a la tarea 3, luego será encontrada rápidamente por el algoritmo paralelo. El algoritmo secuencial habría encontrado la

solución luego de haber recorrido los espacios de búsquedas correspondientes a las tareas 1 y 2. Por otro lado, si la solución yace al final del espacio de búsqueda correspondiente a la tarea 1, entonces el algoritmo paralelo realizará casi 4 veces más trabajo que el algoritmo secuencial, lo cual afectará al speedup. El primer caso es un claro ejemplo de speedup superlineal, el cual fue mencionado en la Sección 3.2.2.1.

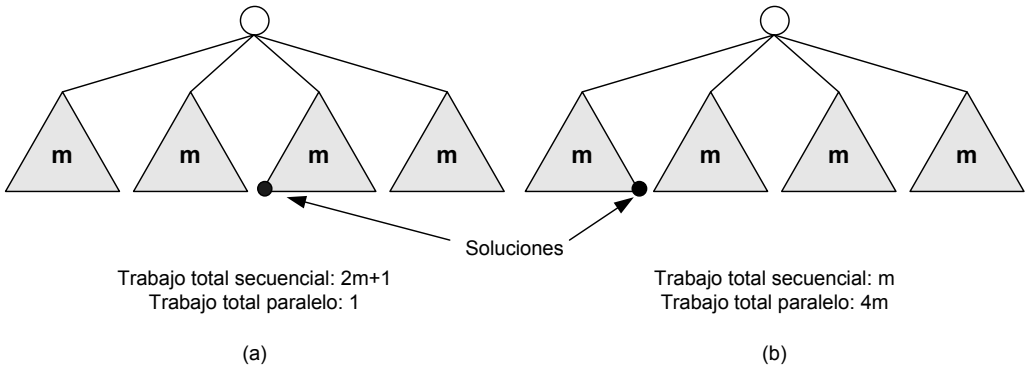

Figura 4.5. Ejemplo de speedup anómalo producto de la descomposición exploratoria.

4.1.4 Descomposición especulativa

En algunas aplicaciones no es posible determinar a priori las dependencias entre tareas. Es por ello que para esta clase de aplicaciones no es posible identificar tareas independientes. La *descomposición especulativa* es utilizada cuando un programa puede tomar uno de varios caminos computacionales posibles dependiendo de la salida de otras computaciones que lo preceden. En éste caso, mientras una tarea realiza la computación cuya salida será utilizada para decidir la siguiente, otras tareas pueden empezar las computaciones de la siguiente etapa. Se puede pensar a la descomposición especulativa como la evaluación en forma paralela de una o más ramas de una sentencia *switch* en C antes de que la entrada del *switch* esté disponible. Mientras una tarea realiza la computación que permitirá decidir que rama del switch es elegida, otras tareas podrían estar computando las diferentes ramas. Cuando la entrada del switch se encuentra disponible, la computación correspondiente a la rama elegida será utilizada mientras el resto serán descartadas. Obviamente el tiempo de ejecución paralelo es menor que el tiempo de ejecución serial ya que el tiempo que se utiliza para evaluar la condición que determinará la próxima tarea es empleado para realizar computaciones valiosas para la siguiente etapa en forma paralela. Sin embargo, esta versión paralela del switch no puede evitar realizar computaciones que más tarde serán desechadas.

Dado el juego Ta-Te-Tí presentado en la Sección 4.1.3. Se supone que el jugador X ya ha realizado su movimiento. Mientras se espera que O realice el suyo, X puede evaluar en forma paralela y anticipada cual será el próximo mejor movimiento de acuerdo al casillero que marque O. Para el momento en que O haya realizado su movimiento, ya se sabe con anterioridad cual es el próximo movimiento de X (o al menos se habrá avanzado en la búsqueda del mismo).

La descomposición especulativa se diferencia de la descomposición exploratoria en dos puntos. Primero, en la descomposición especulativa la entrada de una bifurcación a múltiples tareas paralelas es desconocida, mientras que en la descomposición exploratoria lo que se desconoce es la salida producida por múltiples tareas paralelas generadas en una bifurcación. Segundo, en la descomposición especulativa, el algoritmo paralelo realiza más trabajo total que el algoritmo serial. Esto se debe que al momento de la bifurcación, el algoritmo serial sólo evalúa una de las ramas computacionales, a diferencia del algoritmo paralelo que evaluará múltiples de

ellas. En la descomposición exploratoria en cambio, el algoritmo secuencial explora las diferentes alternativas una tras otra, ya que la rama que contiene la solución no se conoce de antemano, y el algoritmo paralelo podría realizar más, menos, o igual cantidad de trabajo total comparado al algoritmo secuencial dependiendo de la ubicación de la solución en el espacio de búsqueda.

4.1.5 Descomposición compuesta

Las técnicas de descomposición presentadas previamente no son exclusivas y a menudo pueden combinarse para lograr una mayor concurrencia de tareas. Con frecuencia, una computación es estructurada en múltiples etapas y a veces resulta necesario aplicar diferentes tipos de descomposición a las distintas etapas. Por ejemplo, si se aplica una descomposición recursiva al problema de encontrar el valor máximo de un conjunto de *N* números, se podría producir una cantidad de tareas mucho mayor que el número de procesos *p* disponibles. Como alternativa se puede pensar en particionar el conjunto de números en *p* porciones de igual o similar tamaño y que cada tarea se encargue de calcular el máximo de la porción asignada. El resultado final se obtiene al encontrar el máximo de los *p* resultados intermedios utilizando la descomposición recursiva de la Figura 4.6.

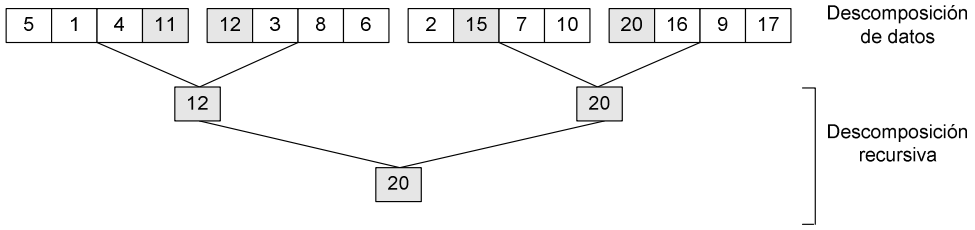

Figura 4.6. Descomposición compuesta en cuatro tareas aplicada al problema de encontrar el valor máximo en un arreglo de 16 elementos.

4.2 Técnicas de mapeo

El siguiente paso luego de que el problema haya sido descompuesto en tareas, consiste en asociar dichas tareas a las unidades de procesamiento donde serán ejecutadas. A esta asociación se la conoce como mapeo y tiene como objetivo minimizar los overheads ligados a la ejecución paralela de las tareas. Existen básicamente dos fuentes de overhead: la comunicación entre tareas y la ociosidad de las mismas. Las unidades de procesamiento pueden pasar tiempo ociosas por una variedad de razones. Una distribución del trabajo despareja causará que algunas tareas terminen antes que otras. Algunas veces, las dependencias asociadas al problema obligan a que una tarea deba esperar un resultado o a la terminación de otra. Con frecuencia estas fuentes de overhead dependen del mapeo. Es por ello, que un buen mapeo debe intentar lograr dos objetivos: (1) reducir el tiempo que las tareas utilizan para interactuar, y (2) reducir el tiempo en que algunas tareas pasan ociosas mientras otras realizan trabajo útil.

Estos dos objetivos algunas veces pueden resultar incompatibles. Por ejemplo, para minimizar las interacciones entre tareas se podría asignar el conjunto de tareas que interactúan entre ellas a la misma unidad de procesamiento. En la mayoría de los casos, éste mapeo produciría un desbalance de carga entre las unidades de procesamiento y en consecuencia las que tengan una carga de trabajo liviana estarán ociosas mientras que aquellas con una carga de trabajo pesada estarán intentado terminar sus tareas. Por otro lado, con el objetivo de equilibrar la carga entre unidades de procesamiento, puede resultar necesario asociar tareas que interactúan

intensamente a unidades de procesamiento diferentes. Como se puede notar encontrar un buen mapeo no es un problema simple.

Las técnicas de mapeo utilizadas en los algoritmos paralelos pueden clasificarse en tres categorías: estático, dinámico y jerárquico. Entre los factores que determinan que técnica emplear se encuentran el paradigma de programación elegido, las características de las tareas y la interacción que se da entre estas [GRA03].

4.2.1 Mapeo estático

En las técnicas de mapeo estático las tareas son asignadas a las unidades de procesamiento antes de que el algoritmo se ejecute. Cuando las tareas son generadas estáticamente, es decir, se conoce de antemano a las tareas que resolverán el problema, se puede usar tanto mapeo estático como dinámico. La elección de un buen mapeo no depende de un solo factor sino de varios. Entre ellos podemos nombrar al conocimiento del tamaño de las tareas, el tamaño de los datos asociados a las tareas, las características de la interacción entre tareas, e incluso el paradigma de programación paralela empleado. Aún conociendo el tamaño de las tareas, en general, obtener un mapeo óptimo es un problema NP-completo para tareas de diferente tamaño. Sin embargo, en la práctica, existen buenas heurísticas que permiten obtener soluciones cercanas al mapeo óptimo con un grado de error aceptable.

El mapeo estático suele emplearse en aquellos problemas que utilizan técnicas de descomposición basadas en los datos.

4.2.2 Mapeo dinámico

Las técnicas de mapeo dinámico se utilizan cuando las tareas son asignadas en ejecución a las unidades de procesamiento. Si las tareas son generadas dinámicamente, es decir, son creadas en ejecución, estamos obligados a mapearlas de forma dinámica también. Utilizar un mapeo estático cuando el tamaño de las tareas es desconocido puede provocar un desbalance de carga. Es por ello que en esos casos se recomienda usar técnicas de mapeo dinámico. En cambio, cuando la cantidad de datos asociados a las tareas es mayor que la cantidad de computación que cada tarea realiza, utilizar un mapeo dinámico puede resultar perjudicial para el rendimiento debido al movimiento de datos que se produce entre unidades de procesamiento. Aún así, el mapeo dinámico puede ofrecer un rendimiento aceptable si el entorno es de memoria compartida y la interacción es sólo de lectura.

4.2.3 Mapeo jerárquico

Resulta natural expresar algunos algoritmos como un grafo de tareas. Sin embargo, realizar un mapeo de acuerdo al mismo puede producir un desbalance de carga o una concurrencia pobre. Por ejemplo, dado el grafo de dependencias de tareas en forma de árbol binario de la Figura 4.7.

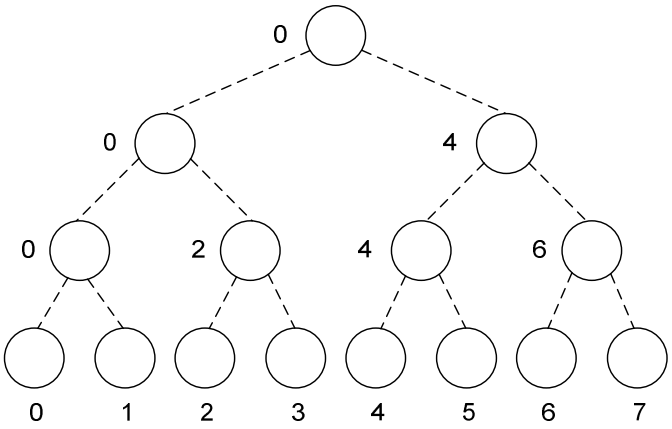

Figura 4.7. Mapeo de un grafo de tareas en forma de árbol binario a ocho unidades de procesamiento.

Si se realiza el mapeo de acuerdo a éste grafo se puede notar que habrá unidades de procesamiento ociosas en los niveles superiores del mismo. Para solucionar esta problemática, podemos combinar las técnicas de mapeo. Si las tareas son lo suficientemente grandes, un mejor mapeo puede obtenerse al descomponer las tareas en subtareas más pequeñas. Siguiendo con el ejemplo de la Figura 4.7, la tarea raíz puede ser dividida entre las ocho unidades de procesamiento, las tareas del siguiente nivel pueden ser divididas entre cuatro unidades de procesamiento cada una, seguido por las tareas del penúltimo nivel las cuales son dividas entre dos unidades de procesamiento cada una. Las ocho tareas hojas son mapeadas una a cada unidad de procesamiento. En la Figura 4.8 se puede observar éste mapeo jerárquico.

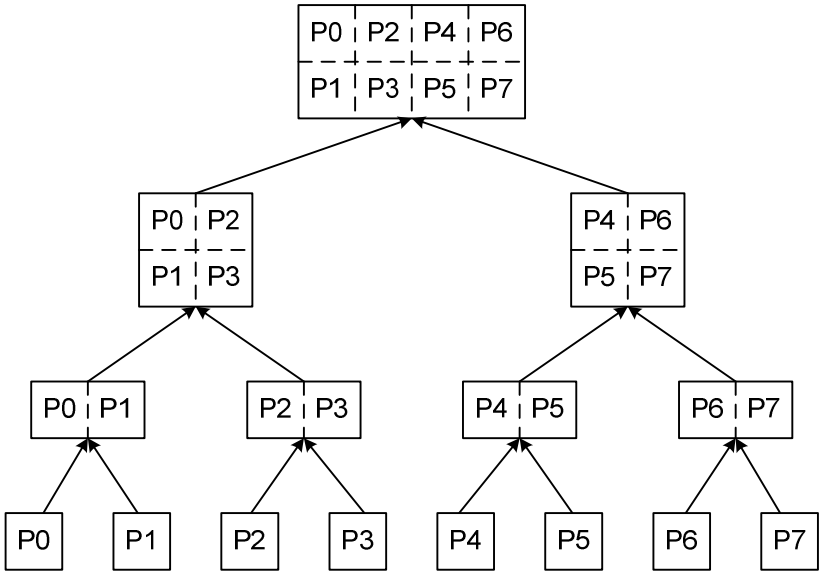

Figura 4.8. Mapeo jerárquico aplicado a un grafo de dependencias de tareas. Cada nodo, representado por un arreglo, es una tarea. El particionamiento de los arreglos produce las subtareas, las cuales son mapeadas a ocho unidades de procesamiento.

El algoritmo quicksort paralelo que se presentó anteriormente tiene un grafo de dependencias similar al de la Figura 4.7, y por lo tanto se le puede aplicar el mapeo jerárquico de la Figura 4.8.

4.3 Métodos para reducir overhead

Como se vio anteriormente, el overhead generado por la interacción entre procesos limita la eficiencia que un programa paralelo puede alcanzar. Son varios los factores que inciden sobre el overhead generado por la interacción: el volumen de datos intercambiados, la frecuencia de la interacción, el patrón de interacciones temporal y espacial, etc.

A continuación se presentan algunas técnicas que permiten reducir el overhead generado por la interacción. Estas técnicas funcionan manipulando uno o más de los factores mencionados en el párrafo anterior [GRA03].

4.3.1 Maximizar la localidad de los datos

En la mayoría de los programas paralelos, los procesos requieren acceder a alguna estructura de datos compartida para poder realizar su trabajo. También es común que los procesos deban interactuar para intercambiar datos. El overhead generado por la interacción entre procesos puede ser reducido utilizando técnicas que permitan maximizar la localidad de los datos. Entre estas técnicas se encuentran las que intentan minimizar el volumen de datos intercambiados y minimizar la frecuencia de las interacciones.

4.3.1.1 Minimizar el volumen de datos intercambiados

A menor cantidad de accesos a datos compartidos por todos los procesos, menor será el overhead generado por interacción entre los mismos. La idea básicamente consiste en maximizar la localidad temporal de los datos, es decir, intentar realizar todas las referencias a un dato lo más seguido en el tiempo que sea posible. De esta manera se ahorra el costo de tener que traer datos de memoria principal a caché para que el proceso pueda realizar su tarea.

Otra manera de decrementar el volumen de datos accedido por múltiples procesos consiste en utilizar datos locales para almacenar resultados intermedios, y sólo acceder a los datos compartidos para colocar los resultados finales de la computación.

4.3.1.2 Minimizar la frecuencia de las interacciones

Las interacciones entre procesos de un programa paralelo tienen un alto costo de inicio sobre varias arquitecturas. Ésta técnica se basa en reducir la frecuencia de las interacciones reestructurando el algoritmo de manera que los datos sean accedidos y usados en grandes piezas. De esta manera, se amortiza el costo de inicio a través de los diferentes accesos y, por lo tanto, se reduce el overhead generado por interacción. En otras palabras, lo que busca esta técnica es incrementar la localidad espacial en el acceso a los datos.

4.3.2 Minimizar la competencia por recursos

La competencia por los recursos ocurre cuando múltiples procesos intentan acceder a uno de ellos de forma concurrente. Por ejemplo, múltiples transmisiones simultáneas de datos a través del mismo enlace de comunicación, múltiples accesos simultáneos al mismo bloque de memoria, múltiples procesos enviando mensajes al mismo proceso en el mismo instante de tiempo, entre otros. Sólo una de todas las operaciones podrá proceder, mientras que las otras serán encoladas y procederán secuencialmente. El patrón de acceso a los datos y el de interacción entre tareas pueden llevar a producir competencia entre los procesos, lo cual incrementa el overhead por interacción.

La estrategia para reducir la competencia por los recursos es utilizar un patrón de acceso a los datos que no la genere.

4.3.3 Solapar cómputo con interacciones

En muchas ocasiones, los procesos de un programa paralelo realizan un pedido por datos o trabajo y quedan ociosos a la espera de recibir los mismos. Existen diferentes técnicas que permiten reducir estos tiempos ociosos solapando cómputo con interacciones.

Una manera de evitar los tiempos ociosos consiste en contar con los datos necesarios antes de que estos sean requeridos. Para ello se debe iniciar la interacción con anterioridad al momento en que los datos son necesarios e identificar alguna computación que no dependa de los mismos. De esta manera mientras se espera que los datos arriben, se realizan cómputos valiosos que permiten evitar, o al menos, reducir los tiempos ociosos, lo cual se traduce en una reducción del overhead del programa paralelo.

Aún pudiendo reestructurar el algoritmo paralelo, la posibilidad de solapar interacciones con cómputo dependerá del paradigma de programación, del sistema operativo y del hardware subyacente. El paradigma de programación debe proveer un mecanismo que permita que interacciones y cómputo puedan realizarse de forma concurrente, y éste mecanismo debe ser soportado por el hardware subyacente.

4.3.4 Replicar datos o cómputo

Otra forma de overhead en algoritmos paralelos se produce cuando múltiples procesos realizan repetidamente operaciones de lectura a una estructura de datos compartida. Si cada uno de los procesos contara con una copia propia de la estructura de datos, entonces los accesos a la misma no provocarían overhead por interacción.

La replicación de datos es una técnica que permite reducir el overhead generado por interacción, a costo de incrementar los requerimientos de memoria de un programa paralelo. Es por ello que se debe analizar la viabilidad de utilizarla.

Además de los datos iniciales, los procesos de un programa paralelo a menudo comparten resultados intermedios. En algunas ocasiones, puede resultar más efectivo para un proceso computar estos resultados intermedios que interactuar con otros procesos para obtenerlos.

4.3.5 Utilizar operaciones colectivas de interacción optimizadas

Frecuentemente en los algoritmos paralelos se encuentra la ocurrencia de operaciones de interacción colectivas. Ejemplos de ellas son enviar datos a todos los procesos mediante *broadcasts,* o sumar números cada uno perteneciente a un proceso diferente. Se clasifican a las operaciones de interacción colectivas en tres categorías: operaciones que son usadas por los procesos para acceder a los datos, operaciones que son utilizadas para realizar computaciones de comunicación intensiva, y operaciones empleadas para sincronización.

Se han desarrollado y optimizado implementaciones para estas operaciones colectivas con el objetivo de minimizar los overheads asociados tanto a la transferencia de los datos como a la competencia por los recursos.

4.3.6 Solapar interacciones con otras interacciones

Solapar interacciones entre múltiples pares de procesos puede reducir el volumen efectivo de comunicaciones, siempre que el hardware subyacente lo permita.

Por ejemplo, si se considera la operación de comunicación colectiva broadcast de uno-a-muchos empleada en el paradigma de pasaje de mensajes con cuatro procesos P_0, P_1, P_2 y P_3. Un algoritmo muy utilizado para implementar esta operación trabaja de la siguiente manera. En el primer paso, P_0 envía los datos a P_2. En el segundo paso, P_0 envía los datos a P_1, y concurrentemente, P_2 envía los datos que recibió de parte de P_0 a P_3. Luego, la operación completa requiere de dos pasos dado que los dos envíos del segundo paso se realizan simultáneamente. Por otro lado, un algoritmo simplista enviaría los datos de P_0 a P_1, de P_1 a P_2 y de P_2 a P_3, necesitando de tres pasos.

4.4 Modelo de algoritmos paralelos de acuerdo al patrón de interacción

Un modelo de algoritmo permite estructurar la computación de un algoritmo paralelo. Es la combinación de las técnicas de descomposición y mapeo elegidas junto a una estrategia para reducir overhead. Existen diferentes modelos de algoritmos paralelos, los cuales se describen a continuación [GRA03].

4.4.1 Modelo de datos paralelos

El modelo de datos paralelos es uno de los modelos de algoritmos más simple. En éste modelo todas las tareas realizan computaciones similares. Es por ello que como técnica de descomposición del problema, suele emplearse la descomposición de datos. Éste tipo de paralelismo, donde las tareas realizan las mismas operaciones sobre diferentes conjuntos de datos de forma concurrente, se conoce como *paralelismo de datos*. El cómputo de cada tarea puede estar dividido en fases y típicamente las tareas sincronizan o interactúan entre cada una de ellas. Las tareas suelen mapearse a las unidades de procesamiento estáticamente, lo que, combinado a la descomposición de datos, garantiza una carga balanceada.

Los algoritmos que siguen éste modelo pueden ser implementados utilizando tanto el paradigma de memoria compartida como el de pasaje de mensajes. La ventaja de elegir el paradigma de memoria compartida se encuentra en la facilidad de programación, especialmente si la distribución de datos es diferente en cada fase. Por otro lado, el espacio de direcciones distribuido del paradigma de pasaje de mensajes puede permitir un mejor aprovechamiento de la localidad de los datos.

Como estrategia para reducir el overhead debe emplearse un particionamiento de los datos que permita minimizar la interacción entre tareas. También, de ser posible, debe solaparse cómputo con interacción y utilizar rutinas optimizadas para comunicaciones colectivas.

Un ejemplo de un algoritmo de datos paralelos es la multiplicación de matrices vista en la Sección 4.1.2.1. En la descomposición de la Figura 4.2 se puede observar que todas las tareas aplican las mismas operaciones a datos diferentes.

4.4.2 Modelo de grafo de tareas

En la Sección 1.4 se vio que las tareas en las cuales se descompone un problema pueden tener dependencias entre ellas. Para representar estas dependencias suele emplearse una abstracción conocida como *grafo de dependencias de tareas*. Un grafo de dependencias de tareas es un grafo acíclico dirigido en el cual los nodos representan tareas y las aristas dirigidas indican las dependencias entre ellas.

El modelo de grafo de tareas suele emplearse para resolver aquellos problemas donde la cantidad de datos asociados a las tareas es mucho mayor comparado a la

cantidad de computación que realizan las mismas. El grafo de dependencias de tareas es utilizado para particionar los datos y para realizar el mapeo. Las interrelaciones entre las tareas son usadas para promover la localidad de los datos o para reducir los costos de interacción. El mapeo usualmente se realiza de forma estática, aunque puede ser dinámico también. De cualquiera de las dos maneras, la información del grafo de dependencias de tareas es usada para asignar tareas a unidades de procesamiento. Tanto el paradigma de memoria compartida como el de pasaje de mensajes pueden ser utilizados para implementar los algoritmos de éste modelo.

De manera de reducir el volumen y la frecuencia de las interacciones se puede utilizar el grafo de dependencias de tareas para promover la localidad de los datos al particionar los mismos y también para mapear las tareas de acuerdo al patrón de interacción. Usar comunicación asincrónica permite solapar cómputo con interacción.

Como ejemplo de algoritmo basado en el modelo de tareas se puede nombrar a quicksort, el algoritmo de ordenación descrito en la Sección 4.1.1.

4.4.3 Modelo *Bag of tasks*

En el modelo *Bag of tasks* todas las tareas son colocadas en una estructura compartida por todos los procesos, llamada *bolsa de tareas*. Mientras quedan tareas en la bolsa, los procesos toman tareas de la misma y las ejecutan. Es posible que cada proceso al ejecutar una tarea, genere otras nuevas, las cuales serán agregadas a la bolsa. En éste modelo el mapeo de las tareas a procesos es de forma dinámica, y puede ser tanto centralizado como descentralizado. Si el mapeo es descentralizado y el trabajo se genera dinámicamente, entonces un algoritmo de detección de terminación será necesario para que todos los procesos dejen de trabajar una vez que la bolsa se encuentre vacía.

Tanto el paradigma de pasaje de mensajes como el de memoria compartida pueden utilizarse para implementar algoritmos de éste tipo. Uno de los factores que incide sobre la elección anterior es la cantidad de datos asociados a las tareas. Utilizar una solución distribuida cuando la cantidad de datos asociados a las tareas es mucho mayor que la computación de las mismas, puede resultar muy costoso debido al overhead por interacción que se genera. Se debe ajustar la granularidad de las tareas de manera de obtener un equilibrio entre el desbalance de carga y el costo asociado al acceso a la bolsa para agregar y quitar tareas.

Para minimizar el overhead de interacción se puede permitir que los procesos saquen *grupos de tareas*. El problema potencial de éste esquema es que puede llevar a un desbalance de carga si la cantidad de tareas que conforman el grupo resulta muy grande. Se reduce éste problema al permitir que el tamaño del grupo se vaya decrementando a medida que el programa progresa.

El modelo Bag of tasks puede ser utilizado para implementar soluciones recursivamente paralelas, en el cual las tareas son los llamados recursivos. También para resolver problemas iterativos que tengan un número fijo de tareas independientes.

4.4.4 Modelo *Master-Slave*

En el modelo *Master-Slave* se encuentran dos clases de procesos: los maestros y los esclavos. Los procesos maestros se encargan de generar trabajo y los procesos esclavos de llevarlo a cabo, aunque algunas veces los maestros también trabajan. La asignación de tareas a los procesos esclavos por parte del maestro puede realizarse de forma dinámica o estática. Esta última tiene sentido sólo si el maestro es capaz de estimar el tamaño de las tareas de antemano o bien, si una asignación aleatoria de las tareas puede resultar en una distribución del trabajo balanceada.

Éste modelo permite ser implementado eficientemente tanto en paradigmas de memoria compartida como en pasaje de mensajes dado que la interacción es naturalmente de dos vías: el maestro sabe que tiene que darle trabajo a los esclavos y los esclavos saben que deben recibir trabajo del maestro.

Una gran cantidad de esclavos o un tamaño pequeño de tareas pueden convertir al maestro en un cuello de botella. La granularidad de las tareas debe elegirse de manera de que el costo de realizar el trabajo sea mayor que el costo de transferirlo y el de sincronizar. La comunicación asincrónica puede ayudar a solapar la interacción con la generación de trabajo por parte del maestro.

4.4.5 Modelo productor-consumidor o pipeline

Un pipeline es una colección lineal de procesos, donde cada proceso consume la salida de su predecesor y produce una salida para su sucesor. A excepción del primer proceso del pipeline, la llegada de datos a un proceso indica que el mismo tiene trabajo por realizar. Un pipeline puede verse como una cadena de productores y consumidores. Cada proceso es consumidor de los datos que produce su antecesor y productor de los datos que consume su sucesor. El mapeo de tareas a procesos suele realizarse de forma estática.

Todos los procesos tienen trabajo cuando el pipeline está lleno. La granularidad de las tareas debe ajustarse de manera de garantizar un balance de carga. A mayor tamaño de tarea, mayor es el tiempo que tarda el pipeline en llenarse, o lo que es lo mismo mayor es el tiempo que tenemos procesos ociosos. Por otro lado, un tamaño de tarea muy pequeño causará que los procesos pasen más tiempo comunicándose que realizando trabajo. La técnica de reducción de overhead más utilizada consiste en solapar interacción con computación.

4.4.6 Modelo compuesto

Ocasionalmente puede resultar útil aplicar más de un modelo a un problema dado, lo cual produce un modelo compuesto. Un modelo compuesto consiste de múltiples modelos los cuales son aplicados de forma jerárquica o secuencial a las diferentes fases del algoritmo paralelo.

4.5 Modelo de programación según el espacio de direcciones

Diversos lenguajes de programación y librerías han sido desarrollados para la programación paralela explícita. Estas difieren principalmente en la manera en que el usuario ve al espacio de direcciones. Los modelos se dividen básicamente en los que proveen un espacio de direcciones compartido o uno distribuido, aunque también existen modelos híbridos que combinan las características de ambos. El espacio de direcciones influye significativamente sobre la manera en que los hilos o procesos intercambian la información. A continuación se detallan los diferentes modelos [GRA03].

4.5.1 Paradigma de pasaje de mensajes

Cuando el espacio de direcciones es distribuido, cada proceso tiene su memoria local, y no existe una memoria compartida a la cual todos los procesos puedan acceder para intercambiar información. Es por ello que el intercambio de información entre procesos se da enviando y recibiendo mensajes.

4.5.1.1 Principios del paradigma de pasaje de mensajes

Dos claves son las que caracterizan al paradigma de pasaje de mensajes: se asume que el espacio de direcciones está particionado y sólo permite programación paralela explícita.

Al estar particionado el espacio de direcciones, los datos deben pertenecer a una partición en especial. Esto agrega complejidad a la programación pero a su vez también estimula la localidad de los datos la cual es crítica para obtener un alto rendimiento, ya que una unidad de procesamiento puede acceder a sus datos locales mucho más rápido que a los datos que no lo son. Otro aspecto provocado por el particionamiento del espacio de direcciones, es la necesidad de que dos procesos –el proceso que tiene el dato y el proceso que lo requiere- cooperen en cualquier interacción. Esto, en determinadas circunstancias puede llevar a programas poco naturales y complejos en la codificación. Por otro lado, como el programador es consciente de todas las interacciones, puede determinar fácilmente el costo de las mismas y pensar el algoritmo de manera de minimizarlas.

El paradigma de pasaje de mensajes requiere que el código paralelo sea codificado explícitamente por el programador. En otras palabras, el programador deberá analizar el algoritmo secuencial e identificar aquellas porciones que puedan ser paralelizadas. Es por ello que la programación bajo éste paradigma suele ser difícil y demandante en materia intelectual. Sin embargo, los programas escritos correctamente suelen ofrecer un alto rendimiento, aún escalándolos a un gran número de procesos.

El paradigma de pasaje de mensajes se divide en dos subparadigmas: el paradigma *asincrónico* y el *débilmente sincrónico*. En el primero de ellos, todos los procesos se ejecutan asincrónicamente, lo que permite implementar cualquier algoritmo. Por otro lado, los programas son más difíciles de razonar y pueden tener comportamientos no determinísticos. El paradigma débilmente asincrónico intenta maximizar las ventajas y minimizar las desventajas del paradigma asincrónico. En él, todos los procesos deben sincronizar al momento de interactuar. No obstante, entre esas interacciones, los procesos se ejecutan asincrónicamente. Como las interacciones ocurren sincrónicamente, es más fácil razonar sobre el programa. La mayoría de los algoritmos conocidos pueden ser escritos bajo éste paradigma.

Si bien el paradigma de pasaje de mensajes permite la ejecución de programas diferentes sobre cada uno de los *p* procesos, la mayoría de los programas escritos bajo éste paradigma suelen emplear el enfoque *SPMD* (*single program multiple data*). En ésta clase de programas, el código que ejecuta cada proceso estará determinado por la rama de una gran sentencia de condición múltiple que el mismo elija. Normalmente la mayoría de los procesos suelen ejecutar el mismo código, a excepción de unos pocos (los llamados *maestros* o *raíces*). Los programas SPMD pueden ser asincrónicos o débilmente asincrónicos.

4.5.1.2 Operaciones Send y Receive

Como se vio anteriormente, los procesos en el paradigma de pasaje de mensajes interactúan enviando y recibiendo mensajes. Es por ello que se presentan las operaciones básicas `send` y `receive`, cada una con sus diferentes protocolos. Los prototipos de las operaciones básicas son los siguientes:

```
send (void *sendbuf, int nelems, int dest)
receive (void *recvbuf, int nelems, int source)
```

El parámetro `sendbuf` apunta al buffer donde se encuentran los datos a enviar, `recvbuf` apunta al buffer que almacenará los datos a recibir, `nelems` es la cantidad

de elementos que serán enviados, `dest` es el identificador del proceso receptor y `source` es el identificador del proceso emisor.

Presentar sólo estas dos operaciones sería simplificar en demasía la programación con pasaje de mensajes. Para poder profundizar sobre la misma analiza el siguiente ejemplo.

```
P0                          P1
a = 10;                     receive(&a,1,0);
send(&a,1,1);               printf("%d\n",a);
a = 0;
```

En éste simple ejemplo, el proceso P0 le envía un mensaje al proceso P1, y luego modifica el valor de la variable `a`. La semántica del `send` debe garantizar que el valor recibido por el proceso P1 sea 10 y no 0.

Existen diferentes protocolos para las operaciones `send` y `receive`, los cuales ayudan a asegurar que la semántica de estas operaciones se mantenga. A continuación se describen los mismos.

4.5.1.3 Operaciones de pasaje de mensajes bloqueantes

Una manera de garantizar que se respete la semántica de la operación `send` consiste en no devolver el control de la misma hasta que el dato a transmitir esté seguro. La política anterior puede llevarse a cabo con o sin la utilización de buffers.

Cuando se utilizan buffers para las operaciones bloqueantes, la operación `send` se bloquea hasta que el proceso receptor alcance la sentencia `receive` correspondiente. Cuando esto sucede, el mensaje es enviado y la operación `send` retorna una vez completada la operación de comunicación.

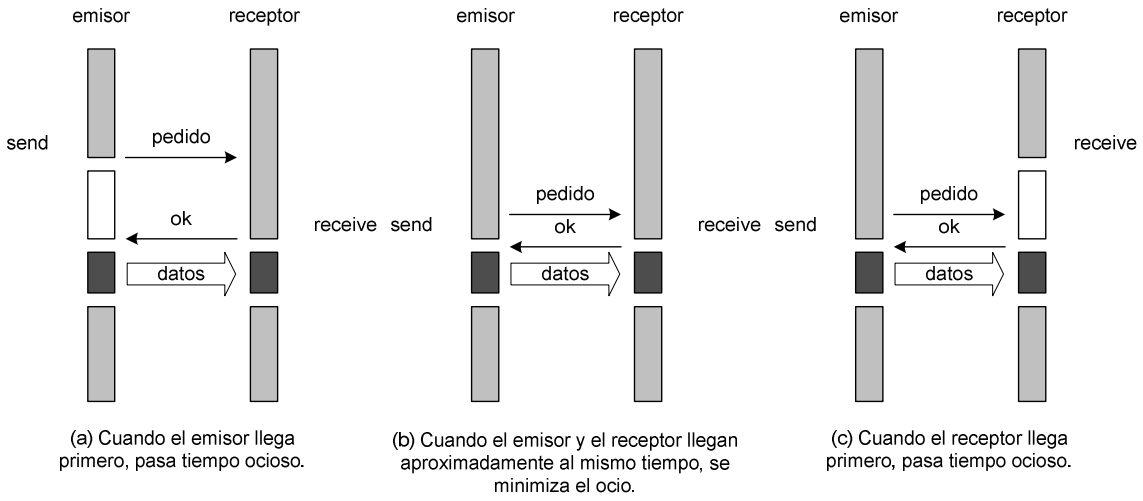

Figura 4.9. Protocolo de una operación send/receive bloqueante sin buffers.

Las operaciones bloqueantes sin buffers tienen dos aspectos negativos. El primero de ellos es la posibilidad de que algunos procesos pasen tiempos ociosos a la espera de que una operación de comunicación finalice. Se pueden observar los diferentes casos en la Figura 4.9. El segundo aspecto negativo es la mayor probabilidad de que ocurra *un bloqueo mutuo* o *deadlock* debido a errores de programación, los cuales en otros casos no ocurrirían. Un bloqueo mutuo se da cuando tenemos una espera circular entre dos o más procesos concurrentes.

El protocolo de comunicación bloqueante con buffers consiste en utilizar buffers prealocados tanto en los emisores como en los receptores. Cuando un proceso encuentra una operación `send` simplemente copia los datos a enviar en el buffer correspondiente. Una vez que la operación de copiado finalizó, el proceso puede continuar con el programa. La transmisión del mensaje al proceso receptor puede ser realizada de diferentes maneras dependiendo de los recursos de hardware disponibles. Si se cuenta con hardware para comunicación asincrónica (sin intervención de la CPU), entonces una transferencia hacia el buffer del receptor es iniciada luego de que el emisor ha copiado los datos en su buffer. Si no se cuenta con esta clase de hardware dedicado, entonces el emisor será el responsable de realizar la transferencia al buffer del receptor y no podrá continuar hasta que complete la misma. En la Figura 4.10 podemos observar las diferentes posibilidades de implementación del protocolo bloqueante con buffers.

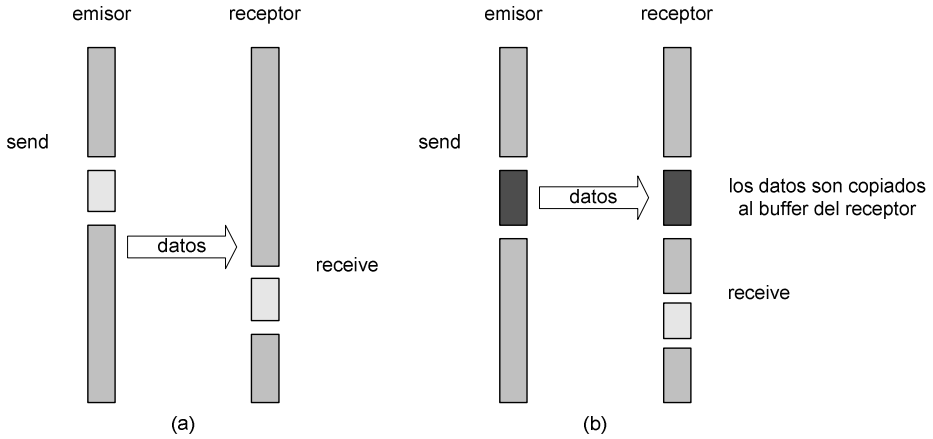

Figura 4.10. Protocolos para la transferencia bloqueante con buffers: (a) cuando contamos con hardware de comunicación que emplea buffers tanto en el emisor como en el receptor; y (b) sin hardware de comunicación, el emisor interrumpe al receptor y copia los datos en el buffer del mismo.

Queda claro que los protocolos que emplean buffers reducen el overhead generado por ociosidad. Si bien el manejo de los buffers agrega cierto overhead, el mismo no es significativo. Salvo en aquellas aplicaciones paralelas donde las comunicaciones son altamente sincrónicas, las comunicaciones con buffers brindan un mejor rendimiento.

Por último, los protocolos con buffers no evitan que ocurra un bloqueo mutuo pero sí permiten que se reduzcan la cantidad de situaciones en las que el mismo puede producirse.

4.5.1.4 Operaciones de pasaje de mensajes no bloqueantes

Los protocolos bloqueantes garantizan la corrección de sus operaciones a costo de generar overhead. Para evitar éste overhead podemos delegar en el programador la garantía de la corrección semántica y de esta forma obtener operaciones `send`/`receive` que generen poco overhead. Como esta clase de protocolos no bloqueantes retornan de las operaciones `send` y `receive` antes de que sea seguro semánticamente, es responsabilidad del usuario no alterar datos que podrían participar potencialmente de una comunicación. Luego de regresar de una operación no bloqueante, el proceso puede realizar cualquier computación que no depende de la misma.

Al igual que con las operaciones bloqueantes, las operaciones no bloqueantes pueden o no emplear buffers. Cuando no se utilizan buffers, el proceso emisor

simplemente le deja un mensaje pendiente al proceso receptor y continúa con su trabajo. Cuando el proceso receptor alcanza la sentencia `receive` correspondiente, se inicia la transferencia. En el caso que sí usan buffers, se utiliza acceso directo a memoria (*DMA, Direct Memory Access*) para copiar los datos a enviar en un buffer prealocado. Mientras la operación de copiado se lleva a cabo, el proceso emisor puede continuar con su trabajo. Una vez que la operación de copiado terminó, el proceso receptor inicia una transferencia desde el buffer del emisor hasta un buffer local. El empleo de buffers permite reducir el tiempo en que los datos no están seguros. En la Figura 4.11 podemos observar las diferentes posibilidades de implementación del protocolo no bloqueante sin buffers.

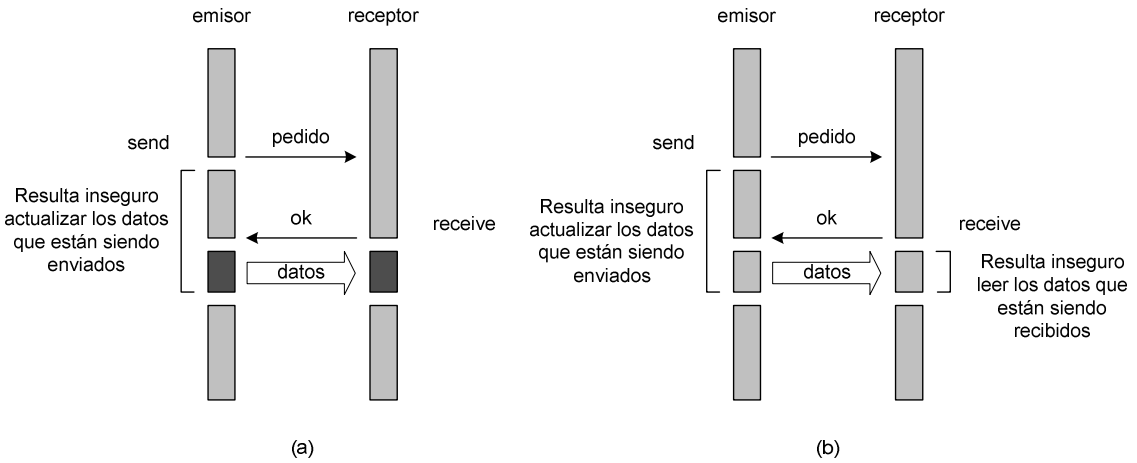

Figura 4.11. Operaciones send y receive no bloqueantes sin buffers: (a) sin hardware de comunicación; y (b) con hardware de comunicación.

4.5.1.5 MPI

La librería MPI (*Message Passing Interface*) define un estándar para el pasaje de mensajes que puede ser utilizado desde los lenguajes C o Fortran, y potencialmente desde otros también. MPI define tanto la sintaxis como la semántica de un conjunto básico de rutinas, las cuales resultan útiles a la hora de escribir programas con mensajes. En la actualidad, existen diversas implementaciones para diferentes proveedores de hardware [OHI96].

Para poder utilizar las rutinas provistas por MPI es necesario que todos los procesos invoquen la operación `MPI_Init`. La misma inicializa el ambiente MPI. La rutina `MPI_Finalize` debe ser llamada al finalizar la computación, ya que se encarga de realizar diferentes tareas de mantenimiento para poder cerrar el ambiente MPI.

MPI ofrece diferentes rutinas que implementan las operaciones básicas `send` y `receive`. Para comunicación bloqueante, ofrece una primitiva de recepción y cuatro primitivas de emisión que difieren en el modo de transmisión.

`MPI_Recv`	La operación se completa sólo después de que los datos hayan sido recibidos y copiados.
`MPI_Send`	La operación se completa cuando el sistema puede copiar el mensaje en el buffer del emisor (no está obligado) o cuando el mensaje es recibido.
`MPI_Bsend`	La operación se completa cuando el mensaje es copiado en el buffer del receptor, o cuando el mensaje es recibido.

`MPI_Ssend`	La operación se completa cuando el mensaje es recibido.
`MPI_Rsend`	La operación no debe iniciarse hasta que el receptor alcance la sentencia de recepción. En ese caso la operación finaliza inmediatamente.

Al igual que para la comunicación bloqueante, MPI provee primitivas de recepción y emisión para comunicación no bloqueante. El término “no bloqueante” en MPI implica que la rutina regresa inmediatamente y que sólo ha iniciado la operación de transferencia del mensaje, no necesariamente la ha completado. La aplicación no podrá reutilizar el buffer de manera segura luego de que la rutina no bloqueante regrese. Las primitivas son `MPI_Irecv` y `MPI_Isend`. Además, para poder chequear la finalización de las operaciones `send` y `receive` no bloqueantes, MPI provee dos funciones: `MPI_Test` y `MPI_Wait`. La primera chequea si una operación ha finalizado o no, y la segunda permite bloquear al proceso hasta que una operación no bloqueante realmente finalice.

Anteriormente se mencionó la importancia de emplear comunicaciones colectivas, siempre que sea posible, para lograr reducir el overhead de los programas paralelos. MPI provee un conjunto de rutinas para ello. Por ejemplo:

`MPI_Bcast`	Envía un mensaje a todos los procesos restantes.
`MPI_Scatter`	Separa y distribuye datos entre todos los procesos.
`MPI_Gather`	Recibe y concatena los datos enviados por todos los procesos restantes.
`MPI_Reduce`	Combina mensajes de todos los procesos.

4.5.2 Paradigma de memoria compartida

Cuando el espacio de direcciones es compartido, los procesos o hilos intercambian información leyendo y escribiendo sobre variables que son compartidas.

4.5.2.1 Hilos

En el modelo de hilos, cada proceso puede consistir de múltiples flujos de control independientes los cuales son llamados hilos. La palabra hilo se emplea para indicar que se ejecutará una secuencia de instrucciones continua potencialmente larga.

Un atributo característico es que los hilos de un proceso comparten el espacio de direccionamiento del mismo, es decir, tienen un espacio común. Cuando un hilo almacena un valor en el espacio de direcciones compartido, otro del mismo proceso también podrá accederlo [RAU10].

4.5.2.2 ¿Por qué usar hilos?

Los modelos de programación con hilos tienen ventajas sobre los modelos de pasajes de mensajes, aunque también algunas desventajas. A continuación se mencionan algunas de ellas [GRA03].

- Portabilidad del software: las aplicaciones hiladas pueden ser desarrolladas en máquinas secuenciales y luego ser trasladadas a máquinas paralelas sin tener que realizar cambios.
- Ocultamiento de la latencia: uno de los mayores overheads, tanto en programas secuenciales como paralelos, es la latencia del acceso a

memoria, de la E/S y de la comunicación. Como se vio en el Capítulo 1, mediante la técnica de multihilado podemos ocultar la latencia y de esta manera reducir el overhead.

- Planificación y balance de carga: usualmente resulta difícil obtener una distribución del trabajo balanceada en aplicaciones poco estructuradas y dinámicas. Los hilos permiten al programador especificar un gran número de tareas concurrentes que serán mapeadas a unidades de procesamiento de forma dinámica. De esta manera, se libera al programador de la responsabilidad de la planificación explícita y del balance de carga.
- Facilidad de programación y amplio uso: debido a las ventajas mencionadas anteriormente, los programas hilados resultan más fáciles de escribir que los correspondientes programas utilizando pasaje de mensajes. Sin embargo, lograr igual rendimiento para ambos programas puede requerir un esfuerzo mayor. Gracias a la gran aceptación de la API para hilos POSIX, hoy se cuenta con una amplia gama de herramientas de desarrollo.

4.5.2.3 Pthreads

Un hilo es un proceso liviano. Muchos sistemas operativos han provisto mecanismos que permitían a los programadores escribir aplicaciones multihiladas. Sin embargo, como estos mecanismos eran diferentes, los programas no resultaban portables, incluso entre distintas versiones del mismo sistema operativo. Para solucionar éste problema, un gran número de personas se reunieron a mitad del año 1990 para definir un conjunto estándar de rutinas en el lenguaje C para la programación multihliada. A la misma se la llamó Pthreads.

La librería Pthreads se compone de más de una docena de funciones para la administración y sincronización de hilos. A continuación se describen las más relevantes de ellas.

Para poder escribir un programa multihilado, es necesario contar con una función que permita crear hilos. Para ello la API Pthreads provee la función `pthread_create`. En muchas ocasiones un hilo debe esperar a que otro termine para poder realizar su trabajo. La función `pthread_join` suspende la ejecución del hilo invocador hasta que el hilo especificado como parámetro finalice con su trabajo.

Cuando múltiples hilos intentan manipular el mismo dato, los resultados pueden ser incoherentes si no se toman los cuidados apropiados. Esto se debe a que en el paradigma de memoria compartida la comunicación se da de forma implícita, no así la sincronización. Los programadores deben garantizar que las diferentes tareas concurrentes accedan a los datos en forma sincronizada de manera de obtener programas hilados que sean correctos. Las APIs de hilos proveen soporte para implementar secciones críticas y operaciones atómicas usando *mutex-locks*. Los mutex-locks tienen dos estados: trabado y destrabado. En un momento dado, sólo un hilo puede trabar un mutex-lock, ya que es una operación atómica generalmente asociada con un pedazo de código que manipula datos compartidos. Para poder acceder a esos datos compartidos, un hilo debe primero intentar adquirir un mutex-lock. Si el mutex-lock se encuentra trabado, entonces el hilo que intenta adquirirlo se bloquea. Esto se debe a que un mutex-lock trabado implica que actualmente hay otro hilo en la sección crítica y que ningún otro hilo debe acceder. Cuando un hilo abandona la sección crítica, debe destrabar el mutex-lock de manera de permitir que otros hilos puedan entrar en la sección crítica.

La API Pthreads provee una serie de funciones para el manejo de mutex-locks. La función `pthread_mutex_lock` se emplea para trabar un mutex-lock. Si el mutex-lock se encuentra trabado, el hilo invocador se bloquea; de otra manera el mutex-lock es trabado y el hilo invocador puede continuar.

Al abandonar la sección crítica, un hilo debe destrabar el mutex-lock asociado con dicha sección. Si no lo hace, ningún otro hilo podrá acceder a esta sección, lo que provoca un bloqueo. Para ello contamos con la función Pthreads `pthread_mutex_unlock`, la cual destraba un mutex-lock. Al llamar a esta función, en el caso de un mutex-lock normal, el mismo se destraba y uno de los hilos bloqueados es planificado para entrar en la sección crítica.

El empleo de locks implica la serialización de porciones de un programa ya que las secciones críticas deben ser ejecutados por los hilos uno después de otro. Es probable que suframos una pérdida de rendimiento significativa si los locks encierran una gran cantidad de instrucciones. Es por ello que las secciones críticas deben ser lo más pequeñas posibles. A veces es posible reducir el ocio asociado al uso de locks usando una función alternativa, `pthread_mutex_trylock`. Esta función intenta trabar un mutex-lock. Si el mismo está destrabado, entonces lo traba; si ya está trabado por otro hilo, en lugar de bloquear al hilo, le permite continuar su ejecución. De esta forma el hilo puede realizar otro trabajo y más tarde consultar el estado del mutex-lock.

Como se mencionó en el párrafo anterior, el uso indiscrimnado de locks puede generar overhead por el ocio de los hilos bloqueados. Si bien la función `pthread_mutex_trylock` reduce éste overhead, introduce otro por la consulta del estado de los locks. Una solución natural a éste problema consiste en suspender la ejecución de un hilo hasta que el lock esté disponible. El hilo suspendido será despertado cuando el lock se encuentre disponible. Esta funcionalidad se obtiene utilizando *variables condición*.

Una variable condición es un objeto usado para sincronizar hilos. Esta variable permite a un hilo bloquearse hasta que un dato específico alcance un estado predeterminado. A toda variable condición se le asocia un predicado. Cuando éste predicado se vuelve verdadero, la variable condición es utilizada para señalizar al o a los hilos que se encuentran en espera.

Una variable condición siempre tiene un mutex asociado a ella. El hilo traba al objeto mutex y luego verifica el estado del predicado; si el predicado es verdadero, el hilo espera en la variable condición asociada con el predicado utilizando la función `pthread_cond_wait`. El hilo que invoque a esta función se bloquea hasta que reciba una señal por parte de otro hilo o sea interrumpido por el sistema operativo. Además de bloquear al hilo, la función destraba el mutex. Esto permite que otros hilos puedan adquirirlo. Cuando el hilo es señalizado, espera a readquirir el mutex antes de reanudar la ejecución. Podemos pensar que detrás de cada variable condición tenemos una cola. Los hilos que invocan a la función `pthread_cond_wait` sobre la variable condición renuncian a la adquisición del mutex y entran en la cola. Cuando la variable condición es señalizada, uno de los hilos en la cola es desbloqueado, y cuando el mutex se vuelva disponible, le será asignado al hilo. Para poder señalizar las variables condición, la API Pthreads cuenta con la función `pthread_cond_signal`, la cual desbloquea a algún hilo que esté actualmente esperando en la variable condición asociada.

Capítulo 5

Bioinformática

5.1 ¿Qué es la bioinformática?

Hasta hace algunos años la idea de una aplicación directa de métodos informáticos en las ciencias naturales era una idea extraña y poco convincente. Sin embargo en la actualidad resulta evidente que cualquier avance serio en nuestro conocimiento y comprensión de, por ejemplo, los complejos mecanismos celulares sería un imposible si no se contara con la ayuda de poderosos algoritmos y veloces computadoras.

El ADN es la manera de diferenciar entre especies, o los llamados “tipos”. Es por ello que la tipificación de secuencias de ADN es un esfuerzo a escala mundial. Con el desarrollo de técnicas que permiten desentrañar la información que contiene, se propició el surgimiento de la bioinformática que busca ya no sólo adquirir, almacenar y organizar la información biológica que contiene la molécula de ADN, sino también analizar e interpretar estos datos. La bioinformática involucra la solución de problemas complejos usando herramientas de sistemas de computación [ATT02].

5.2 ¿Por qué es importante la bioinformática?

El desafío central de la bioinformática es la racionalización de la masa de información de las secuencias, con vistas no sólo a derivar medios más eficaces de almacenamiento de datos, sino también a diseñar herramientas de análisis más incisivas. El objetivo de este proceso analítico consiste en convertir la información de secuencias en conocimiento bioquímico y biofísico, y descifrar las pistas estructurales, funcionales y evolutivas codificadas en el lenguaje de las secuencias biológicas.

La adquisición de secuencias por sí sola da poca más información sobre la complicada biología de los sistemas. La extracción de sentido biológico de la información de las secuencias es una ciencia en vías de ser exacta. Haciendo una analogía, nos enfrentamos al problema de decodificar un lenguaje desconocido. Este lenguaje pude descomponerse en frases (proteínas), palabras (motivos), y letras (aminoácidos), mientras que el código puede ser abordado en varios de estos niveles. Las letras por sí mismas no tienen un significado mayor, pero al combinarlas en palabras es cuando adquieren importancia. Un cambio trivial, quizás en una sola letra de una palabra, puede modificar su significado (por ejemplo, cara - cera), y en consecuencia el sentido completo de la frase. Es por ello que resulta fundamental descifrar el código correctamente. Por ejemplo, si se considera el sencillo cambio de una base de la cadena A de la hemoglobina humana del codón para el ácido glutámico (*GAA*) a valina (*GUA*); en individuos homocigóticos, esta diferencia minúscula produce un cambio del estado normal a una anemia falciforme fatal.

Por último, el desafío final de la bioinformática consiste en lograr comprender las palabras en una frase de secuencias que forman una estructura proteica particular y escribir frases (diseñar proteínas) propias. En la actualidad, la aplicación de métodos computacionales nos permite reconocer palabras que forman patrones característicos o firmas, pero todavía no entendemos la complicada sintaxis necesaria para unir unos patrones a otros y construir estructuras proteicas completas [ATT02].

5.3 Secuencias de ADN

Una secuencia de ADN es una sucesión de letras representando la estructura primaria de una molécula real o hipotética de ADN, con la capacidad de transportar información.

Las posibles letras son *A*, *C*, *G*, y *T*, que simbolizan las cuatro subunidades de nucleótidos de una banda ADN - adenina, citosina, guanina, timina - que son bases covalentemente ligadas a cadenas fosfóricas. Normalmente las secuencias se presentan pegadas unas a las otras, sin espacios, como por ejemplo la secuencia *AAAGTCTGAC*.

Las secuencias pueden derivarse de material biológico de descarte a través del proceso de secuenciación de ADN. El mismo hace referencia a un conjunto de métodos y técnicas que permiten determinar el orden de los nucleótidos (*A*, *C*, *G* y *T*) en una molécula de ADN. El conocimiento de las secuencias de ADN se ha convertido en un recurso indispensable para áreas de investigación básica, como la biología, así como también para áreas de investigación aplicada, como la biotecnología y la biología forense. El desarrollo de la secuenciación de ADN ha acelerado significativamente la investigación y los descubrimientos en la biología. La velocidad de las técnicas de secuenciación actuales permite que proyectos a gran escala puedan ser llevados a cabo, como por ejemplo, el Proyecto Genoma Humano. Otros proyectos relacionados, a menudo gracias a la colaboración a nivel mundial de científicos, han generado la secuencia completa de muchos animales, plantas y microorganismos.

5.4 Alineamiento de secuencias de ADN

En parte, el centro de todas las operaciones y análisis en el mundo bioinformático, lo tiene el alineamiento de secuencias, tanto para búsqueda de patrones entre secuencias de aminoácidos y nucleótidos, como para la búsqueda de relaciones filogenéticas entre organismos. De esta manera el alineamiento de secuencias permite responder una serie de preguntas para la biología: ¿Qué tan parecidas son dos secuencias ya conocidas? ¿A que puede llegar a parecerse una secuencia desconocida? Encontrarles una respuesta no ha sido una tarea fácil y ha existido una gran brecha entre los conocimientos y volúmenes de datos adquiridos durante largos años de investigación, y la eficiencia con la que estos pueden ser analizados. Sin embargo en la actualidad con la ayuda de sofisticados algoritmos y el desarrollo del campo de la biología computacional se ha logrado acortar la diferencia mencionada.

5.4.1 Comparación de dos secuencias

Dado el desarrollo de un algoritmo que permita determinar la similitud entre dos secuencias, cada una elegida de un alfabeto de complejidad 20. Si se comienza con una aproximación simplista, la cual consiste en alinear las dos secuencias una frente a

la otra e insertar caracteres adicionales para poner las dos series en un alineamiento vertical, como se muestra en la Figura 5.1.

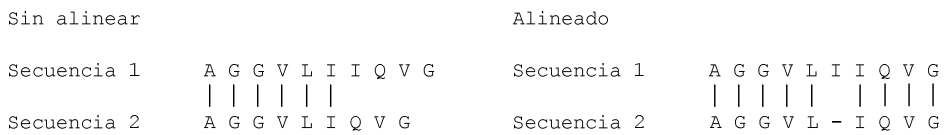

Figura 5.1. Uso del carácter '-' para alinear dos secuencias. Las barras verticales denotan emparejamientos idénticos: seis en el primer alineamiento, nueve en el segundo.

El proceso de alineamiento puede medirse mediante dos parámetros: el número de huecos (*gaps*) introducidos y el número de desemparejamientos (*mismatches*) que persisten en el alineamiento. Una métrica relativa a tales parámetros representa la distancia entre dos secuencias (la conocida como distancia de edición). Existen diversas métricas, y diferentes implementaciones de algoritmos similares pueden emplear distintas medidas de distancia para calcular y puntuar alineamientos [ATT02].

5.4.2 Subsecuencias

El ejemplo de la Sección 5.4.1 es muy simple: las secuencias son muy cortas, tienen casi la misma longitud y son casi idénticas. La realidad indica que esto normalmente no ocurre. Si se considera entonces un par más realista de secuencias., donde la secuencia A tiene una longitud de 450 residuos y la B contiene 600 residuos. Si la secuencia A es en su totalidad idéntica a una porción de la secuencia B, entonces se dice que A es una subsecuencia de B. Para poder poner a A en fase con B, insertamos todos los huecos que hagan falta, como se muestra en la Figura 5.2.

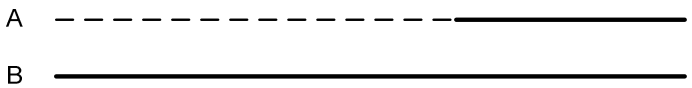

Figura 5.2. Alineamiento de una subsecuencia A con una secuencia completa B, cuando A es idéntica a una parte de B y la inserción de un bloque de huecos permite el alineamiento completo de las dos secuencias.

Si ahora la secuencia A tiene 2 regiones extensas que muestran identidad con la secuencia B. Una vez que se identifican dichas regiones, se insertan huecos en A para alinearlas con B, como se muestra en la Figura 5.3. El algoritmo podría parar en ese punto, una vez encontrada la puntuación de las subsecuencias más alta entre A y B. Este es un ejemplo de un algoritmo heurístico que tiene una implementación directa, en el que las regiones de identidad son obvias [ATT02].

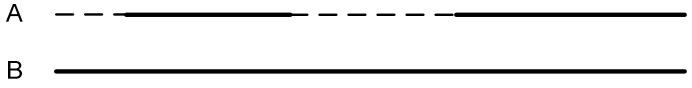

Figura 5.3. Alineamiento de una subsecuencia A con una secuencia completa B, cuando A es idéntica a diferentes partes de B, de modo que se debe insertar más de un bloque de huecos para hacer coincidir las dos secuencias.

5.4.3 Identidad y similitud

Si la comparación de secuencias dependiera sólo de encontrar regiones de identidad estricta entre dos secuencias, se podría desarrollar este método en un programa razonable. Sin embargo, en general, el alineamiento no se restringe al emparejamiento de subsecuencias, sino que implica la comparación de secuencias en toda su longitud. Un alineamiento completo debe determinar las posiciones de todos los residuos en ambas secuencias. Esto significa que es posible que muchos residuos tengan que situarse en posiciones que no son estrictamente idénticas. En ese caso la posición de los huecos en el alineamiento se hace más compleja de calcular. Una solución simple y rápida consiste en insertar huecos de forma no restringida, maximizando así el número de emparejamientos idénticos. Si bien esta solución nos permite alcanzar una puntuación óptima, el resultado de tal proceso no tendría sentido biológico. Es por ello que se introducen penalizaciones en la puntuación para minimizar el número de huecos que se inician (se abren) y a continuación se utilizan penalizaciones de extensión cuando el hueco tienen que ser extendido.

Realizar un alineamiento de secuencias por computadora significa generar un emparejamiento entre dos secuencias según un modelo matemático. El modelo describe, en términos generales, el concepto de alineamiento de dos cadenas de secuencias y sus detalles: penalizaciones por huecos, impacto de las diferentes longitudes de las secuencias, efecto de la complejidad del alfabeto, entro otros. Estos se tratan mediante el uso de parámetros. La elección adecuada de parámetros minimizará el número de huecos, mientras que su relajación permitirá, teóricamente, el alineamiento de cualquier par de secuencias arbitrarias. El hecho de que un programa produzca un alineamiento de dos secuencias no debe tomarse como prueba, por sí mismo, de que exista una relación entre ellas [ATT02].

5.4.4 Similitud local y global

Como se mencionó anteriormente, los alineamientos son modelos matemáticos cuyo comportamiento es susceptible de ser modificado mediante el uso de parámetros. Existen diferentes modelos, cada uno diseñado para recoger una variedad de características físicas de las secuencias biológicas, incluyendo, por ejemplo, su parentesco estructural, funcional o evolutivo. En este contexto, se debe, por lo tanto, tener en cuenta que no hay alineamientos correctos o incorrectos sino modelos distintos que reflejan diferentes perspectivas biológicas.

Generalmente, existen dos modelos que contemplan los alineamientos en forma bastante diferente. El primero considera la similitud en toda la extensión de las secuencias, por lo que es llamado alineamiento global. El segundo se centra en regiones de similitud sólo en partes de las secuencias, lo que se conoce como alineamiento local. Resulta importante comprender estas diferencias de manera de poder apreciar que las secuencias no son uniformemente semejantes y que, por lo tanto, no tiene sentido realizar un alineamiento global sobre secuencias que tienen sólo semejanzas locales.

La razón para buscar similitudes locales es que los sitios fundamentales (por ejemplo, sitios catalíticos de las enzimas) se localizan en regiones relativamente cortas, que se conservan con independencia de eliminaciones o mutaciones en las partes restantes de la secuencia. Por lo tanto, una búsqueda por similitud local puede producir resultados con más sentido y sensibilidad biológicos que una búsqueda que pretenda optimizar el alineamiento sobre la longitud completa de las secuencias [ATT02].

5.5 Algoritmo Smith-Waterman

En 1981, Smith y Waterman describieron un método, conocido como el algoritmo Smith-Waterman, para encontrar regiones comunes de similitud. El mismo consiste en un enfoque matricial y se emplea el retroceso para reconstruir los alineamientos con huecos. El método de Smith-Waterman ha servido como base para el desarrollo de otros algoritmos posteriores e incluso se lo utiliza como medida de referencia para diferentes técnicas de alineamiento. Es realmente una técnica sensible, pero no debemos olvidar que cuando se usa cualquiera de sus implementaciones, la función del algoritmo es hallar pequeñas regiones con similitud local [ATT02].

A continuación se realiza una explicación del funcionamiento del algoritmo para encontrar el puntaje de similitud entre dos secuencias de ADN.

Dadas dos secuencias: $A = a_1a_2a_3\ldots a_M$ y $B = b_1b_2b_3\ldots b_N$, se construye una matriz H de $(N+1)x(M+1)$, de tal forma que las bases nucleótidos que forman la secuencia A etiquetan las filas (a partir de la 1) y los de B las columnas (a partir de la 1). A través de los siguientes pasos de calculan los valores de H que darán el puntaje de similitud entre A y B:

4. Inicializar en 0 la fila 0 y la columna 0 de H, como se indica en la Ecuación 5.1.

$$H_{i0} = H_{0j} = 0 \qquad \text{para } 0 \le i \le N \text{ y } 0 \le j \le M \tag{5.1}$$

5. Calcular el valor de H_{ij} para $\forall i \in [1,..,N]$ y $\forall j \in [1,..,M]$ por medio de la Ecuación 5.2. Este valor indica la máxima similitud entre dos segmentos que terminan en a_i y b_j respectivamente.

$$H_{ij} = \max\begin{cases} 0 \\ H_{i-1,j-1} + V(a_i, b_j) \\ C_{ij} \\ F_{ij} \end{cases} \tag{5.2}$$

   - $V(a_i, b_j)$ es la función de coincidencias que indica el puntaje dado por hacer coincidir a a_i y b_j. Se basa en una tabla de valores llamada *matriz de sustitución* que describe la probabilidad de que una base nucleótida de la secuencia A en la posición i, tenga ocurrencia en la secuencia B en la posición j. La matriz más común es aquella que premia con un valor positivo cuando a_i y b_j son idénticos, y castiga con un valor negativo en caso contrario.
   - C_{ij} es el puntaje considerando un *gap* en la columna j, y se calcula por medio de la Ecuación 5.3.

$$C_{ij} = \max_{1 \le k \le i}\{H_{i-k,j} - g(k)\} \tag{5.3}$$

   - F_{ij} es el puntaje considerando un *gap* en la fila i, y se calcula por medio de la Ecuación 4.

$$F_{ij} = \max_{1 \le l \le j}\{H_{i,j-l} - g(l)\} \tag{5.4}$$

   - $g(x)$ es la función de penalidad por un *gap* de longitud x, y se obtiene por medio de la Ecuación 5.5, siendo q la penalidad por la apertura de un *gap* y r por la prolongación del mismo.

$$g(x) = q + rx \qquad (q \ge 0; r \ge 0) \tag{5.5}$$

6. Obtener el puntaje de similitud como se indica en la Ecuación 5.6.

$$G = \max_{(0 \le i \le N)(0 \le j \le M)}\{H_{ij}\} \tag{5.6}$$

7. A partir de la posición de la matriz *H* donde se encontró el valor *G* (que representa el final del alineamiento de puntuación más elevada entre las dos secuencias) se realizar un proceso de retroceso para obtener el par de segmentos con máxima similitud, hasta que se llega a una posición cuyo valor es 0, siendo este punto el inicio del segmento.

En la Figura 5.4 se muestra un ejemplo de la aplicación del algoritmo Smith-Waterman a dos secuencias *A* = *CAGCCUCGCUUAG* y *B* = *AAUGCCAUUGACGG*, con los siguientes parámetros:

- $V(a_i,b_j) = 1$ si $a_i = b_j$.
- $V(a_i,b_j) = -1/3$ si $a_i \neq b_j$.
- $q = 1$.
- $r = 1/3$.

	x	*C*	*A*	*G*	*C*	*C*	*U*	*C*	*G*	*C*	*U*	*U*	*A*	*G*
x	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	1	1	0	0	0	0	0	0	0	0	0	1	0
A	0	0	1	0,7	0	0	0	0	0	0	0	0	1	0,7
U	0	0	0	0,7	0,3	0	1	0	0	0	1	1	0	0,7
G	0	0	0	**1**	0,3	0	0	0,7	1	0	0	0,7	0,7	1
C	0	1	0	0	**2**	1,3	0,3	1	0,3	2	0,7	0,3	0,3	0,3
C	0	1	0,7	0	1	**3**	1,7	1,3	1	1,3	1,7	0,3	0	0
A	0	0	2	0,7	0,3	1,7	**2,7**	1,3	1	0,7	1	1,3	1,3	0
U	0	0	0,7	1,7	0,3	1,3	**2,7**	2,3	1	0,7	1,7	2	1	1
U	0	0	0,3	0,3	1,3	1	2,3	**2,3**	2	0,7	1,7	2,7	1,7	1
G	0	0	0	1,3	0	1	1	2	**3,3**	2	1,7	1,3	2,3	2,7
A	0	0	0	0	1	0,3	0,7	0,7	2	3	1,7	1,3	2,3	2
C	0	1	1	0,7	1	2	0,7	1,7	1,7	3	2,7	1,3	1	2
G	0	0	0,7	1	0,3	0,7	1,7	0,3	2,7	1,7	2,7	2,3	1	2
G	0	0	0	1,7	0,7	0,3	0,3	1,3	1,3	2,3	1,3	2,3	2	2

Figura 5.4. Aplicación del algoritmo Smith-Waterman a dos secuencias A y B.

En este caso el par de segmentos con máxima similitud es:

G C C A U U G

G C C _ U C G

Capítulo 6

Trabajo experimental, resultados y conclusiones

6.1 Alineamiento de secuencias de ADN

La bioinformática nace a partir del desarrollo de técnicas que permiten descifrar la información contenida en las moléculas de ADN. Dentro del mundo de la bioinformática, el alineamiento de secuencias de ADN representa una de las aplicaciones más relevantes ya que es la base de muchas otras, como el diagnóstico y tratamiento de enfermedades o la producción de alimentos genéticamente modificados. El algoritmo de Smith-Waterman es una técnica empleada para realizar alineamientos locales. El objetivo del mismo es hallar pequeñas regiones similares. La complejidad de este algoritmo es de $O(mxn)$, siendo m y n las longitudes de las secuencias a alinear. Es por ello que al aumentar el tamaño de las secuencias se incrementa la complejidad del algoritmo. Si tenemos en cuenta las dependencias de datos inherentes al problema, la complejidad del algoritmo y que las secuencias pueden tener una longitud de hasta 10^9 nucleótidos, hacen que el estudio de la paralelización de este problema se realmente interesante.

6.2 Solución secuencial

En esta sección se analiza la solución secuencial del algoritmo Smith-Waterman con el objetivo de determinar el puntaje de similitud entre dos secuencias de ADN. Esto significa que no se tiene en cuenta el proceso de retroceso para obtener el segmento que representa la alineación óptima (no se realiza el punto *d* del algoritmo explicado en la Sección 5.5 del Capítulo 5).

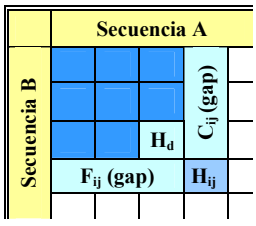

Figura 6.1. Esquema de dependencia de datos

En la Figura 6.1 se muestra la dependencia de datos que existe para calcular los valores de la matriz. Para obtener $H_{i,j}$ se requiere el resultado de $H_{i-1,j-1}$ (H_d en la Figura 6.1) y se necesita saber el puntaje al considerar un *gap* en la fila *i* y otro en la columna *j*. Esta restricción nos lleva a calcular los valores de *H* de arriba hacia abajo y de izquierda a derecha (H_{11}, H_{12}, H_{13}, …H_{21}, H_{22}, H_{23}, …..).

Teniendo en cuenta que no se realiza el paso *d* del algoritmo, no es necesario almacenar la matriz *H* completa, en su lugar se necesita:

- Un vector *h* de longitud *M+1* que mantiene en cada posición el valor obtenido en la última fila procesada sobre esa columna. En la Ecuación 6.1 se indican los valores de *h* en el ejemplo de la Figura 6.1.

$$h_k = \begin{cases} H_{i,k} & k < j-1 \\ H_{i-1,k} & k \geq j-1 \end{cases} \quad (6.1)$$

- Un elemento *e* para guardar en forma temporal el último valor calculado en la fila que se está procesando. En la Figura 6.1, $e = H_{i,j-1}$.
- Un vector *c* de longitud *M+1* que mantiene en cada posición el máximo puntaje considerando un gap en esa columna. En la Ecuación 6.2 se indican los valores de *c* en el ejemplo de la Figura 6.1.

$$c_k = \begin{cases} C_{ik} & k < j \\ C_{i-1,k} & k \geq j \end{cases} \quad (6.2)$$

- Un elemento *f* que mantiene el máximo puntaje considerando un gap en la fila que se está procesando. En el ejemplo de la Figura 6.1, $f = F_{i,\, j-1}$.

6.3 Soluciones paralelas

6.3.1 Solución paralela general

La dependencia de datos mencionada en la sección anterior lleva a resolver el problema con un esquema de pipeline en el que las *E* etapas realizan el mismo trabajo sobre diferentes subconjuntos de nucleótidos consecutivos de la primera secuencia (A en Figura 6.1). En cada ciclo la etapa e_i (para $i \in [1, E\text{-}1]$) recibe un bloque de datos de e_{i-1} y a partir de ellos resuelve parte de su trabajo, y a continuación envía esos resultados a e_{i+1} (excepto la última etapa que no necesita enviarle los resultados a ninguna otra). La primera etapa (e_0) sólo realiza su trabajo enviando los resultados parciales (correspondientes a un bloque) a su sucesor.

Un punto importante de esta solución es seleccionar la cantidad (*TB*) de elementos de la secuencia *B* que forman los bloques de datos que se comunican entre procesos consecutivos, teniendo en cuenta que:

- Recién se comienza a aprovechar al máximo el paralelismo del pipeline cuando han pasado *E-1* ciclos. Es decir cuando a todas las etapas les ha llegado trabajo. Cuanto más grande es *TB*, mayor es el tiempo que tarda en llenarse el pipe, y por consiguiente menor el aprovechamiento del mismo. Desde este punto de vista, *TB* debe tender a 1.
- Si el tamaño *TB* es muy chico, las etapas pasan más tiempo comunicando resultados parciales que procesando información. Desde este punto de vista *TB* debe tender a *N*.

Se debe encontrar un tamaño de bloque adecuado de manera que se pueda solapar la comunicación de los datos con el procesamiento de los mismos. El tamaño óptimo no sólo depende de la arquitectura utilizada, sino también del modelo de comunicación empleado.

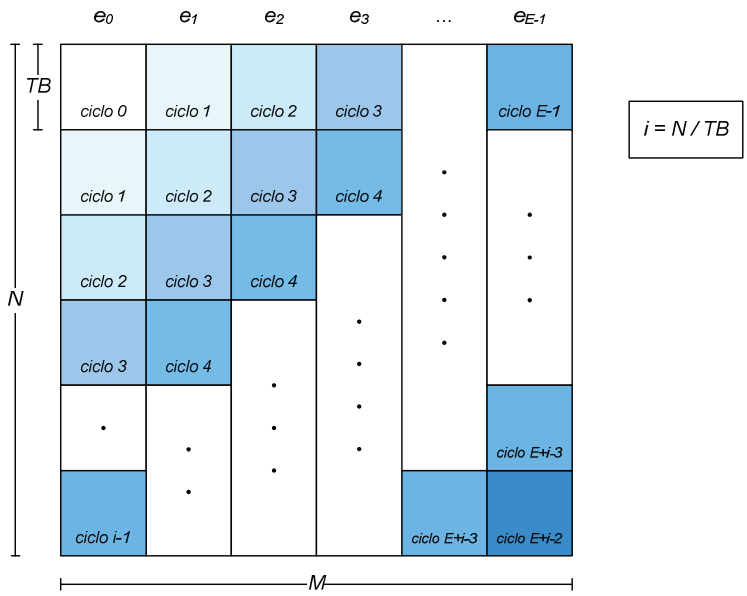

Figura 6.2. Solución paralela general.

6.3.1.1 Pasaje de mensajes como modelo de comunicación

En este caso, cada etapa del pipeline se lleva a cabo por un proceso p_i (para $i \in [0, E\text{-}1]$) diferente, y la comunicación de los resultados parciales se realiza enviando mensajes entre procesos consecutivos. La primera secuencia es distribuida por p_0 entre los E procesos que forman el pipeline.

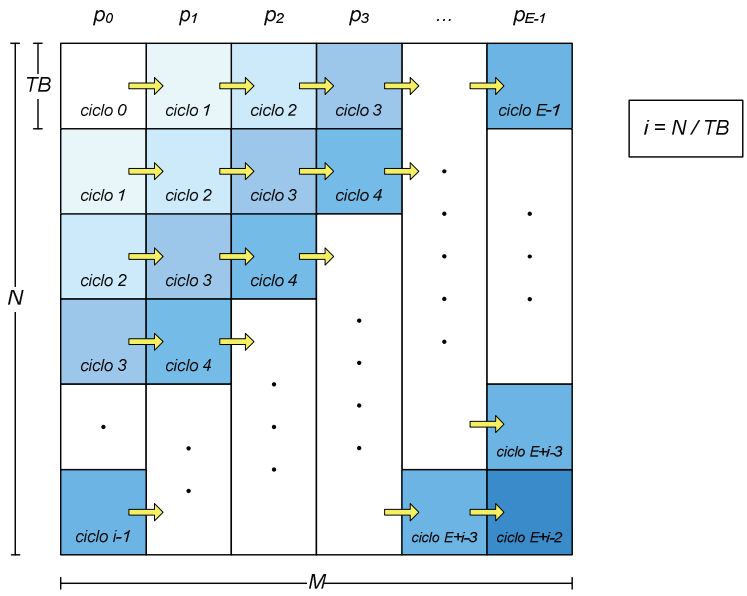

Figura 6.3. Solución paralela con pasaje de mensajes como modelo de comunicación.

6.3.1.2 Memoria compartida como modelo de comunicación

En este caso, cada etapa del pipeline se lleva a cabo por un hilo t_i (para $i \in [0, E\text{-}1]$) diferente. En lugar de comunicar los resultados parciales por medio de pasaje de mensajes, estos se mantienen en la memoria compartida en una única estructura (como en el algoritmo secuencial). Se usa sincronización entre hilos consecutivos para

indicar que se puede comenzar a trabajar con un nuevo bloque de datos, y la misma puede darse de dos formas diferentes: semáforos y variables condición.

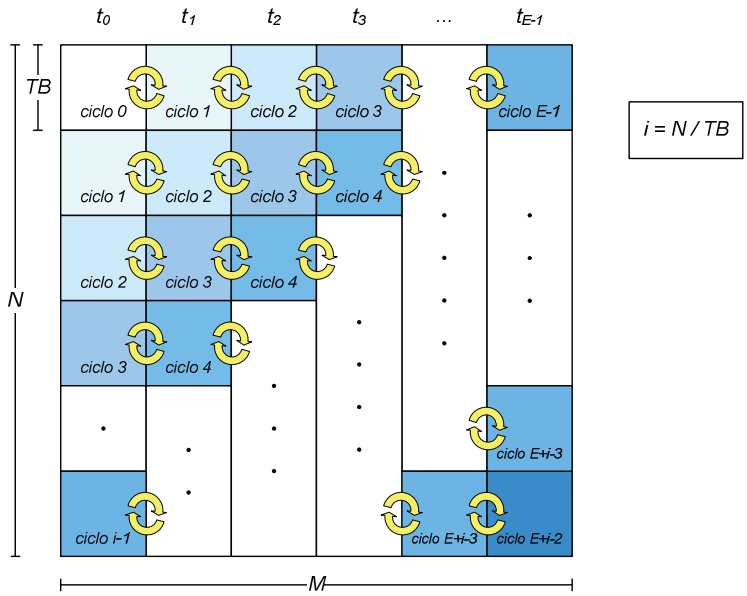

Figura 6.4. Solución paralela con memoria compartida como modelo de comunicación.

6.3.2 Soluciones paralelas híbridas integrando pasaje de mensajes y memoria compartida

Al utilizar una arquitectura híbrida se debe tener en cuenta los diferentes niveles de memoria (entre núcleos de un mismo nodo) y la red de interconexión (entre núcleos de diferentes nodos) para determinar el tamaño *TB* óptimo. Esto lleva a plantear soluciones que combinan el uso de pasaje de mensajes con memoria compartida.

6.3.2.1 Solución paralela híbrida 1

Esta solución híbrida se basa en usar un pipeline de *P* etapas como el descrito en la Sección 6.3.1.1, empleando en cada una de ellas un pipeline de *T* fases como el detallado en la Sección 6.3.1.2.

Al comenzar cada proceso p_i (para $i \in [0, P\text{-}1]$) genera *T-1* hilos para resolver en conjunto los bloques de datos correspondientes a los diferentes ciclos. Esto lleva a trabajar con *P*x*T* hilos (los *P* procesos más los *T-1* hilos generados por cada uno de ellos), por lo que el conjunto de nucleótidos de la primera secuencia (A en la figura 6.1) es distribuido equitativamente entre los *P*x*T* hilos.

Cuando el proceso p_i (para $i \in [0, P\text{-}1]$) debe resolver un bloque de datos (de TB_{pm} elementos), lo divide en subbloques de TB_{mc} nucleótidos cada uno para ser resuelto por el pipeline correspondiente a ese proceso. Para aprovechar las características de la arquitectura hay que determinar los valores de TB_{pm} y TB_{mc} que resulten óptimos en cada caso.

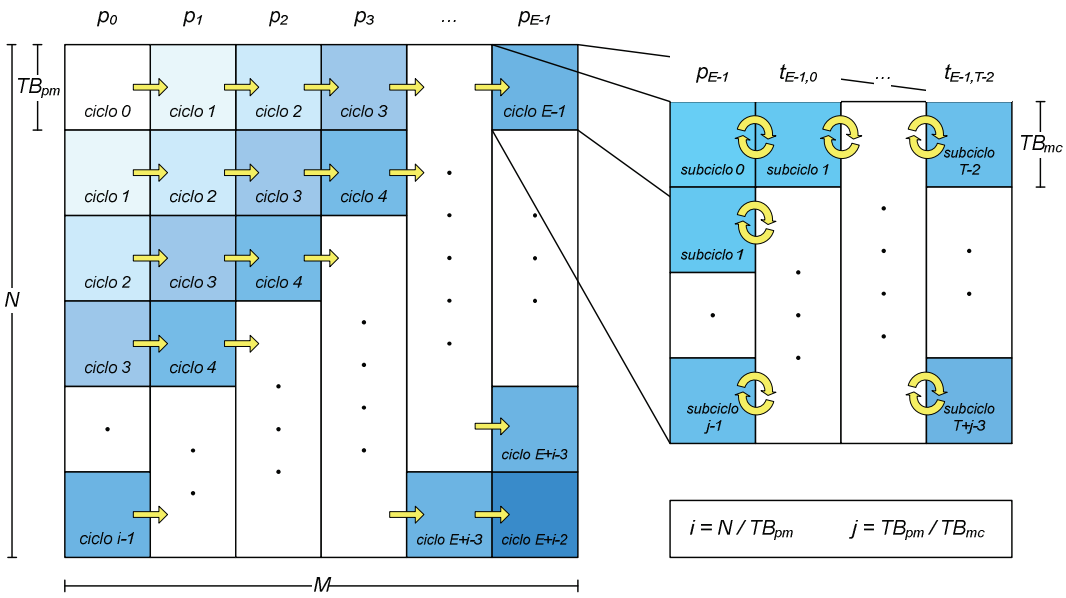

Figura 6.5. Solución paralela híbrida 1.

6.3.2.2 Solución paralela híbrida 2

Esta solución híbrida se basa en usar un pipeline de *E* etapas como el descrito en la Sección 6.3.1.

Al comenzar cada proceso p_i (para $i \in [0, P\text{-}1]$) genera *T-1* hilos ($E = PxT$) para resolver en conjunto los bloques de datos correspondientes a los diferentes ciclos. Esto lleva a trabajar con *PxT* hilos (los *P* procesos más los *T-1* hilos generados por cada uno de ellos), por lo que el conjunto de nucleótidos de la primera secuencia es distribuido equitativamente entre los *PxT* hilos.

Los hilos que se encuentran en el mismo nodo sincronizan y comunican mediante memoria compartida, mientras que los que están en diferentes nodos lo hacen mediante pasaje de mensajes. La sincronización en memoria compartida se da de 2 formas diferentes: semáforos y variables condición.

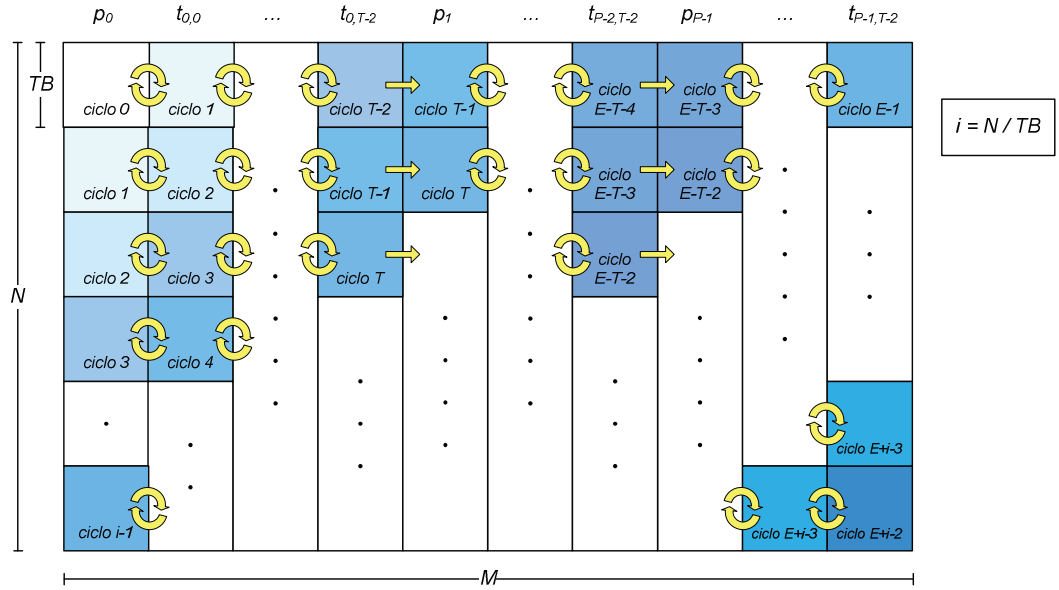

Figura 6.6. Solución paralela híbrida 2.

6.4 Experimentos realizados

En este trabajo se utilizó el lenguaje C con las librerías OpenMPI y/o Pthreads para el pasaje de mensajes y el manejo de hilos respectivamente.

Para realizar la experimentación se ha utilizado un Blade de 8 hojas, con 2 procesadores quad core Intel Xeón e5405 de 2.0 GHz en cada una de ellas. Cada hoja tiene 2 Gb de memoria RAM (compartido entre ambos procesadores) y cache L2 de 2 x 6Mb entre par de núcleos.

Las pruebas fueron realizadas en dos fases:

- El objetivo de la primera fase de prueba fue determinar de forma empírica los tamaños adecuados de bloque de datos para los algoritmos paralelos puros y en base a estos identificar los correspondientes a los algoritmos híbridos. Los algoritmos que emplean memoria compartida como modelo de comunicación se probaron usando una sola hoja del Blade, ya que éste no cuenta con un nivel de memoria compartida entre hojas. En el caso del algoritmo que utiliza pasaje de mensajes como modelo de comunicación, también se probó usando una hoja del Blade para determinar el tamaño adecuado de bloque de datos para la comunicación intra-hoja y, además, se lo probó usando un núcleo de cada hoja de manera de poder determinar el tamaño adecuado de bloque de datos para la comunicación inter-hoja.
- La segunda fase de prueba consiste en probar los diferentes algoritmos híbridos junto al algoritmo que usa sólo pasaje de mensajes utilizando la arquitectura completa para poder analizar y comparar el comportamiento de los mismos.

6.4.1 Primera fase de pruebas

6.4.1.1 Pruebas con una hoja del Blade

Se realizaron pruebas sobre una única hoja del Blade (utilizando los ocho núcleos) para analizar el comportamiento de los algoritmos paralelos puros descritos en la Sección 6.3.1.1 y 6.3.1.2.

Las pruebas realizadas varían en cuanto a la longitud de las secuencias (*N = 65536, 131072, 262144, 524288, 1048576*) y el tamaño de los bloques (*TB = 8, 16, 32, 64, 128, 256, 512, 1024, 2048*). A continuación se describen las pruebas realizadas:

- *PM*: se usa el algoritmo que sólo utiliza pasaje de mensajes.
- *MC1*: se usa el algoritmo que utiliza variables condición para la sincronización entre etapas. Se emplea un proceso *p* y 7 hilos.
- *MC2*: se usa el algoritmo que utiliza semáforos para la sincronización entre etapas. Se emplea un proceso *p* y 7 hilos.

6.4.1.2 Pruebas con un núcleo de cada hoja del Blade

Se probó el algoritmo que solo emplea pasaje de mensajes utilizando un núcleo de cada hoja del Blade (ocho núcleos en total). Al igual que en la pruebas con una sola hoja, se varió el tamaño de las secuencias (*N = 65536, 131072, 262144, 524288, 1048576*) y el tamaño de los bloques (*TB = 8, 16, 32, 64, 128, 256, 512, 1024, 2048*).

6.4.2 Segunda fase de pruebas

Para analizar el comportamiento de los algoritmos híbridos se compara con el algoritmo que emplea sólo pasaje de mensajes, utilizando los ocho núcleos de distinta cantidad de hojas (cuatro u ocho). Al igual que en la Sección 6.4.1.1 y 6.4.1.2, se varía la longitud de las secuencias (*N = 65536, 131072, 262144, 524288, 1048576*). A continuación se describen las pruebas realizadas:

- *PM*: se usa el algoritmo que sólo utiliza pasaje de mensajes. Se varía el tamaño de los bloques (*TB = 8, 16, 32, 64, 128, 256, 512, 1024, 2048*).
- *H1*: se usa el algoritmo híbrido de la Sección 6.3.2.1 con un proceso p_i y 3 hilos por cada procesador de cada hoja. Es decir que cada pipeline de memoria compartida utiliza un procesador quad core completo. En las pruebas se mantiene fijo el tamaño de bloque del pipeline interno (TB_{mc} = 16) y se varía el tamaño de bloque del pipeline de pasaje de mensajes (TB_{pm} *= 128, 256, 512, 1024, 2048*).
- *H2*: se usa el algoritmo híbrido de la Sección 6.3.2.1 con un proceso p_i y 7 hilos por cada hoja. Es decir que cada pipeline de memoria compartida utiliza los dos procesadores quad core completos. En las pruebas se mantiene fijo el tamaño de bloque del pipeline interno (TB_{mc} = 16) y se varía el tamaño de bloque del pipeline de pasaje de mensajes (TB_{pm} *= 128, 256, 512, 1024, 2048*).
- *H3*: se usa el algoritmo híbrido de la Sección 6.3.2.2 que utiliza variables condición para la sincronización entre etapas de una misma hoja. Se emplea un proceso p_i y 7 hilos por cada hoja. Se varía el tamaño de los bloques (*TB = 8, 16, 32, 64, 128, 256, 512, 1024, 2048*).
- *H4*: se usa el algoritmo híbrido de la Sección 6.3.2.2 que utiliza semáforos para la sincronización entre etapas de una misma hoja. Se emplea un proceso p_i y 7 hilos por cada hoja. Se varía el tamaño de los bloques (*TB = 8, 16, 32, 64, 128, 256, 512, 1024, 2048*).

6.5 Resultados

Para evaluar el comportamiento de los algoritmos desarrollados al escalar el problema y/o la arquitectura se analiza la eficiencia, la cual se describe en la Sección 3.2.3 del Capítulo 3.

6.5.1 Resultados de la primera fase de pruebas

Como se mencionó anteriormente, el objetivo de la primera fase de prueba fue determinar los tamaños adecuados de bloque de datos para cada uno de los algoritmos paralelos puros, dado que el mismo no sólo depende de la arquitectura sino también del modelo de comunicación utilizado.

En las Figuras 6.7 y 6.8 se pueden observar las eficiencias logradas por los algoritmos *MC1* y *MC2* para diferentes tamaños de secuencia y de bloques de datos utilizando una hoja (ocho núcleos) de la arquitectura.

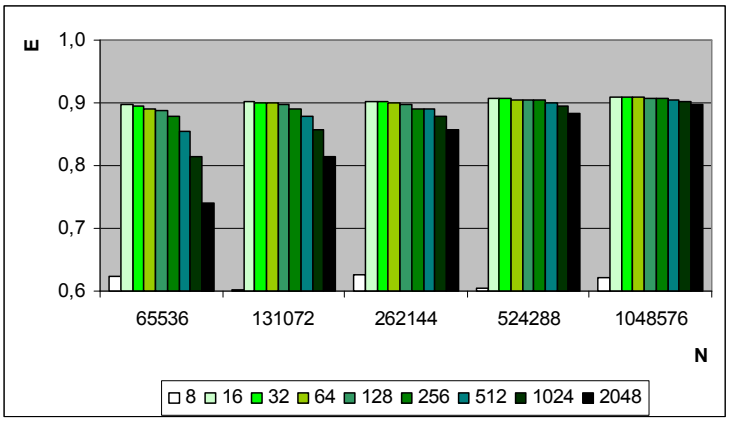

Figura 6.7. Eficiencia lograda por el algoritmo *MC1* para diferentes tamaños de secuencia (N) y de bloques de datos utilizando una hoja (ocho núcleos) de la arquitectura.

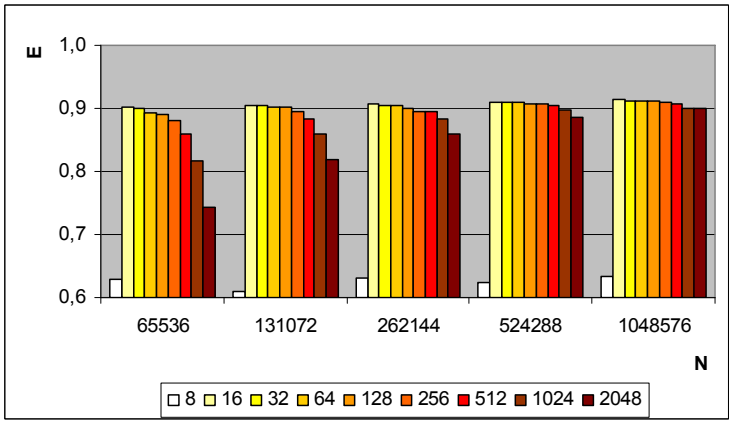

Figura 6.8. Eficiencia lograda por el algoritmo *MC2* para diferentes tamaños de secuencia (N) y de bloques de datos utilizando una hoja (ocho núcleos) de la arquitectura.

Los dos algoritmos se comportan de manera similar ante la variación del tamaño de bloque de datos. Ambos tienen un desempeño pobre al utilizar el tamaño de bloque de datos más pequeño, *TB* = 8. Sin embargo, obtienen su pico de rendimiento al usar un tamaño de bloque de datos igual a 16. Luego, a medida que se incrementa el tamaño del bloque de datos, la eficiencia lograda se reduce. Se puede concluir que el tamaño óptimo de bloque de datos para los algoritmos *MC1* y *MC2* es 16.

En las Figuras 6.9 se puede observar la eficiencia lograda por el algoritmo *PM* para diferentes tamaños de secuencia y de bloques de datos utilizando una hoja de la arquitectura, mientras que en la Figura 6.10 se muestra lo mismo pero usando un núcleo de cada una de las ocho hojas.

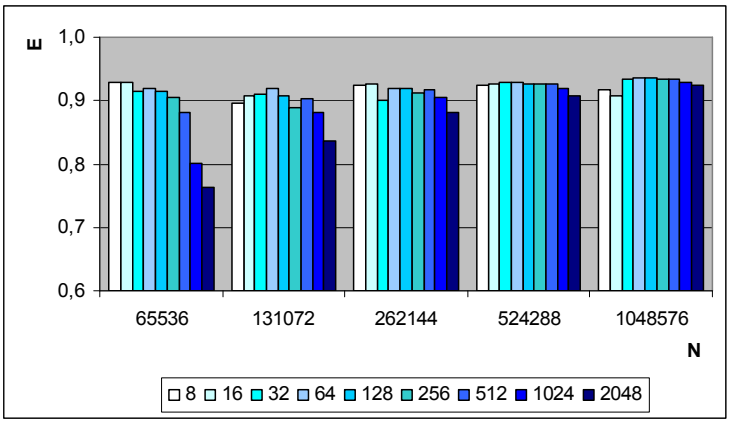

Figura 6.9. Eficiencia lograda por el algoritmo *PM* para diferentes tamaños de secuencia (N) y de bloques de datos utilizando una hoja (ocho núcleos) de la arquitectura.

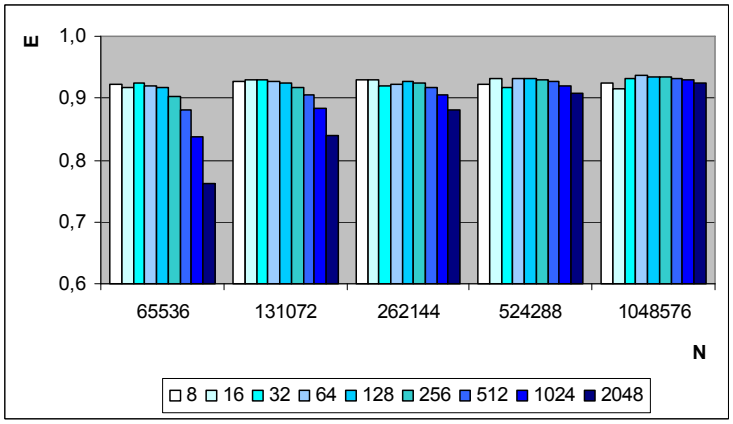

Figura 6.10. Eficiencia lograda por el algoritmo *PM* para diferentes tamaños de secuencia (N) y de bloques de datos utilizando un núcleo de cada una de las ocho hojas de la arquitectura.

Se puede ver en ambos gráficos que el algoritmo *PM* tiene un desempeño similar en las dos pruebas. También se puede observar que no existe un patrón claro de comportamiento de la eficiencia con respecto a la variación del tamaño de bloque de datos para cada tamaño de secuencia. Es por ello que no se puede identificar un tamaño óptimo de bloque de datos. Aún así, se puede decir que emplear tamaños grandes de bloques de datos reduce la eficiencia lograda, en especial si el tamaño de las secuencias es pequeño.

En la Figura 6.11 se presenta un resumen de la mejor eficiencia lograda por cada uno de los tres algoritmos paralelos puros (*PM*, *MC1* y *MC2*) utilizando una hoja (ocho núcleos) de la arquitectura. Para los algoritmos *MC1* y *MC2* la mejor eficiencia se consigue en todos los casos con el tamaño de bloque *TB* = 16, mientras que para *PM* el tamaño óptimo de bloque de datos varía de acuerdo al tamaño de la secuencia.

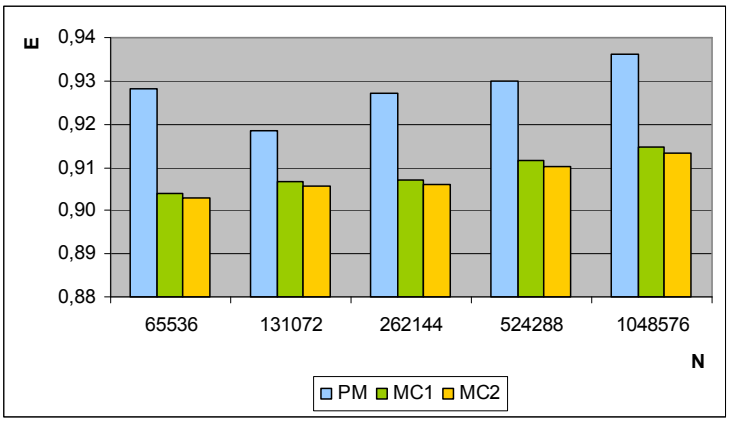

Figura 6.11. Mejor eficiencia lograda por los algoritmos PM, MC1 y MC2 para diferentes tamaños de secuencia (N) utilizando ocho núcleos.

Este gráfico permite identificar que el algoritmo *PM* logra una mayor eficiencia que los algoritmos de memoria compartida para todos los tamaños del problema. También podemos observar que *MC1* logra una mayor eficiencia que *MC2* para cada tamaño de problema dado. Como es de esperar, los tres algoritmos aumentan la eficiencia lograda a medida que incrementamos el tamaño del problema.

6.5.2 Resultados de la segunda fase de pruebas

La segunda fase de pruebas tiene como objetivo analizar y comparar el rendimiento del algoritmo que sólo emplea pasaje de mensajes y de los algoritmos híbridos. Los algoritmos híbridos pueden ser aparejados de acuerdo a su esquema de resolución. Los algoritmos *H1* y *H2* utilizan el mismo esquema de resolución aunque difieren en la cantidad de hilos que cada proceso genera. Los algoritmos *H3* y *H4* resuelven el problema de la misma forma pero se diferencian en el tipo de sincronización para memoria compartida que cada uno utiliza. Por lo anterior, se procede a comparar los algoritmos híbridos según las agrupaciones realizadas, para luego analizar el mejor de cada una de ellas junto al algoritmo que sólo emplea pasaje de mensajes.

6.5.3 Comparación de los resultados de los algoritmos *H1* y *H2*

En la Figura 6.12 y en la Figura 6.13 se muestra la eficiencia lograda por los algoritmos *H1* y *H2* para aquellos tamaños de bloque más significativos (TB_{pm} = 128, 512 y 2048) utilizando cuatro y ocho hojas de la arquitectura respectivamente. Se puede observar que ambos gráficos muestran resultados similares, con un leve incremento en las eficiencias logradas por el primero.

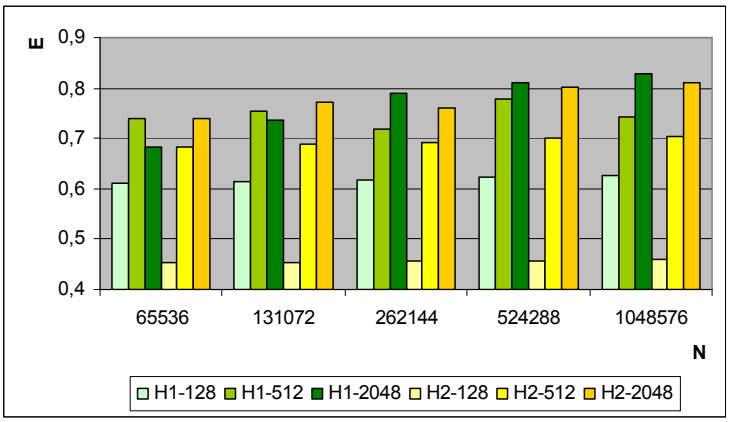

Figura 6.12. Eficiencia lograda por los algoritmos *H1* y *H2* para diferentes tamaños de secuencia (N) y de bloques de datos usando 32 núcleos.

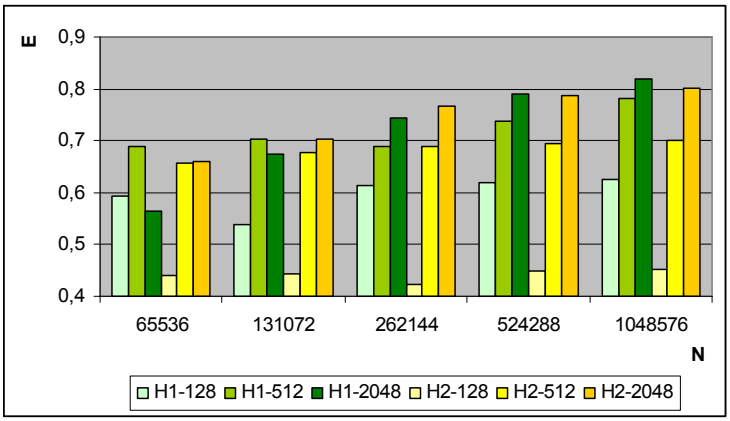

Figura 6.13. Eficiencia lograda por los algoritmos *H1* y *H2* para diferentes tamaños de secuencia (N) y de bloques de datos usando 64 núcleos.

En los gráficos se puede observar que los dos algoritmos incrementan la eficiencia al aumentar el tamaño del problema (longitud de las secuencias). También se puede notar que ambos algoritmo tienden a mejorar la eficiencia al aumentar el tamaño de los bloques de datos. Esta mejora puede entenderse fácilmente a través de la Figura 6.14.

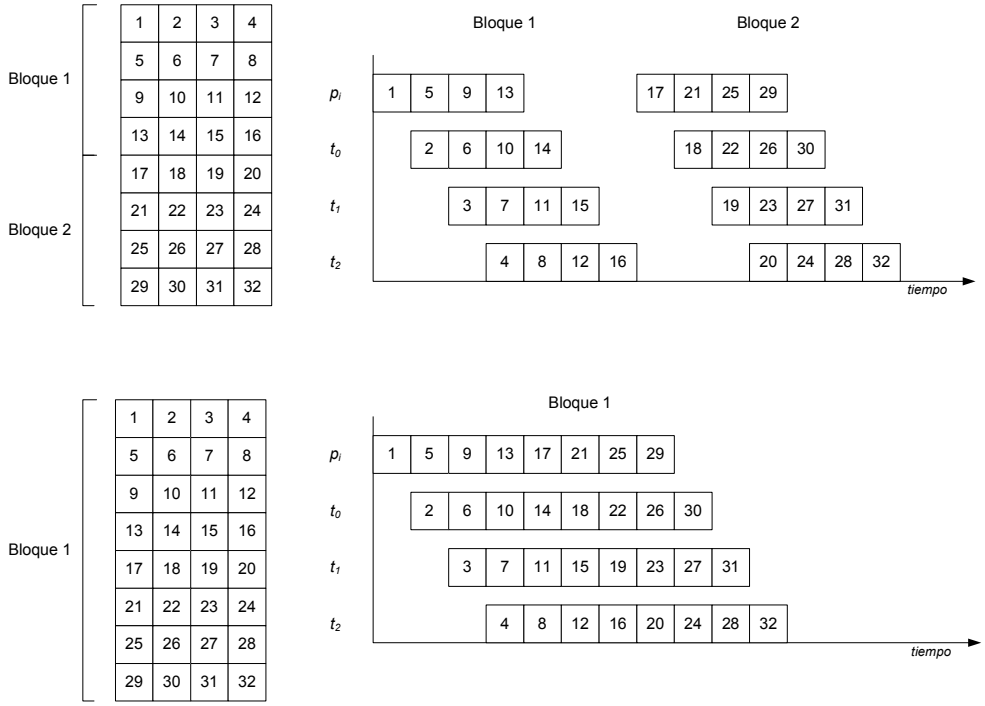

Figura 6.14. Efecto del aumento del tamaño del bloque de datos en el procesamiento utilizando un proceso y tres hilos.

Se puede observar en el gráfico anterior el efecto que produce aumentar el tamaño del bloque de datos. Al agrandar el mismo, se maximiza el paralelismo del pipeline interno de memoria compartida que resuelve cada bloque. En consecuencia, se reduce el tiempo en que procesos e hilos se encuentran ociosos, una de las principales causas de overhead de los programas paralelos, como se mencionó en la Sección 3.1.

En la Figura 6.15 se presenta un resumen de la mejor eficiencia lograda por cada uno de los algoritmos (*H1* y *H2*) para diferentes tamaños de secuencia al utilizar cuatro y ocho hojas de la arquitectura. El algoritmo *H1* alcanza la mejor eficiencia con TB_{pm} = 1024 para todos los tamaños de secuencia, a excepción de 1048576 donde la consigue con TB_{pm} = 2048. En el caso de *H2*, la mejor eficiencia se consigue con TB_{pm} = 1024 para tamaños de secuencia igual a 65536 y 131072, mientras que para el resto de los tamaños el TB_{pm} óptimo es 2048.

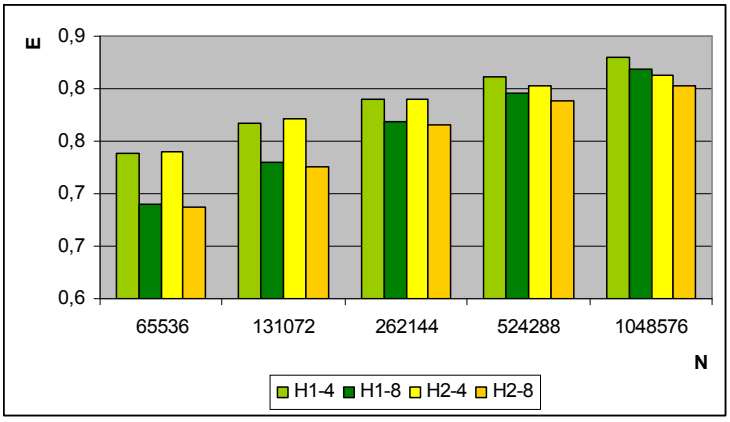

Figura 6.15. Resumen de la mejor eficiencia lograda por los algoritmos *H1* y *H2* para diferentes tamaños de secuencia (N) con cuatro y ocho hojas de la arquitectura.

En este gráfico se puede observar que la diferencia entre las eficiencias logradas por los algoritmos *H1* y *H2* para tamaños pequeños de secuencias no es significativa. Sin embargo, a medida que se aumenta dicho tamaño la diferencia se va incrementando, siendo *H1* el que obtiene un mejor rendimiento. Por otro lado, como es de esperar en la mayoría de los sistemas paralelos, la eficiencia se reduce al incrementar la cantidad de núcleos usados.

6.5.4 Comparación de los resultados de los algoritmos *H3* y *H4*

En la Figura 6.16 y en la Figura 6.17 se muestra la eficiencia lograda por los algoritmos *H3* y *H4* para aquellos tamaños de bloque más significativos (TB_{pm} = 16, 1024 y 2048) utilizando cuatro y ocho hojas de la arquitectura respectivamente. Se puede observar que ambos gráficos muestran resultados similares, con una leve superioridad por parte del primero en las eficiencias logradas.

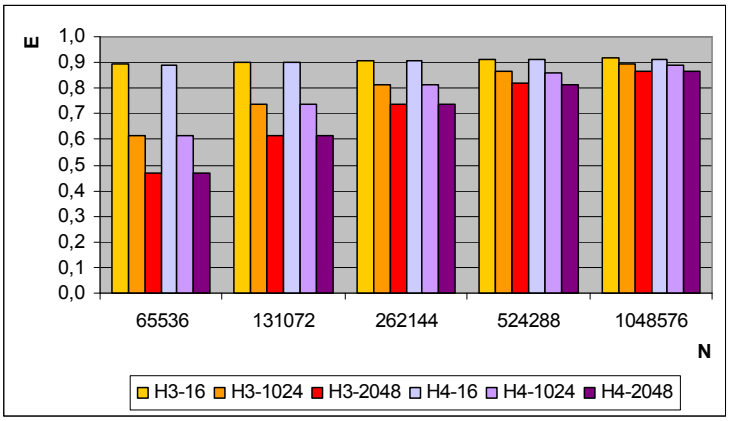

Figura 6.16. Eficiencia lograda por los algoritmos *H3* y *H4* para diferentes tamaños de secuencia (N) y de bloques de datos usando 32 núcleos.

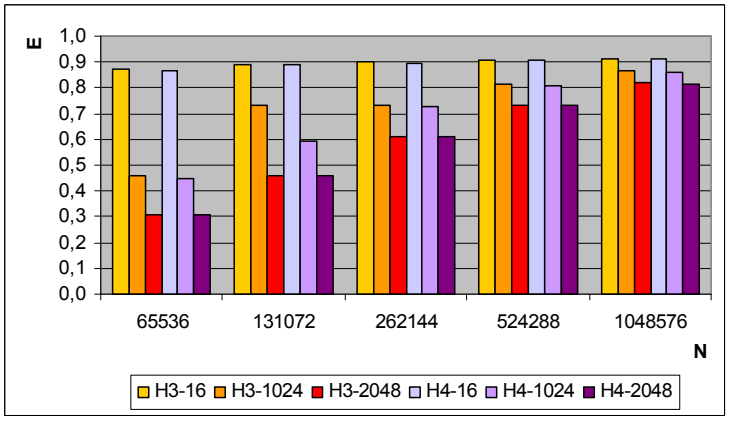

Figura 6.17. Eficiencia lograda por los algoritmos *H3* y *H4* para diferentes tamaños de secuencia (N) y de bloques de datos usando 64 núcleos.

Al observar los gráficos, se puede notar que ambos algoritmos tienen un comportamiento casi idéntico. Al igual que los algoritmos *H1* y *H2*, los dos algoritmos incrementan su eficiencia al agrandar el tamaño del problema. A diferencia de los algoritmos *H1* y *H2*, aumentar el tamaño de los bloques de datos produce un decremento en la eficiencia lograda. Se puede concluir que el tamaño ideal de bloque de datos es 16.

En la Figura 6.18 se presenta un resumen de la mejor eficiencia lograda por los algoritmos *H3* y *H4* al utilizar cuatro y ocho hojas de la arquitectura. En ambos algoritmos la mejor eficiencia se consigue con el tamaño de bloque *TB* = 16.

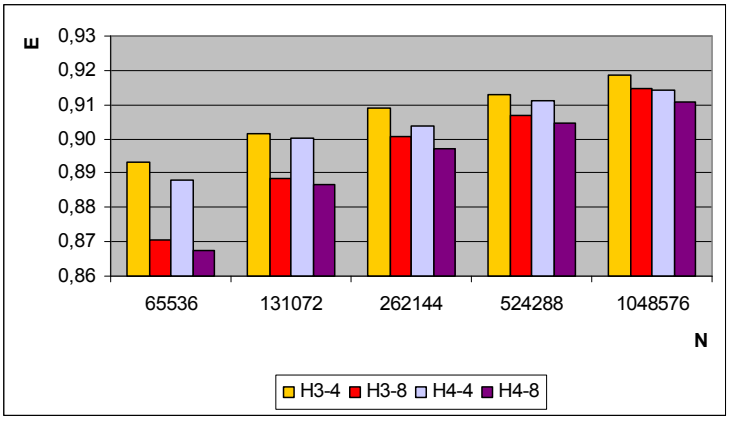

Figura 6.18. Resumen de la mejor eficiencia lograda por los algoritmos *H3* y *H4* para diferentes tamaños de secuencia (N) con cuatro y ocho hojas de la arquitectura.

En este gráfico se puede notar que, si bien no existen diferencias significativas entre los rendimientos de ambos algoritmos, *H3* logra una eficiencia levemente mayor en todos los casos con respecto a *H4*. Además, al incrementar la cantidad total de núcleos utilizados se reduce la eficiencia lograda, aunque esta diferencia se acorta a medida que aumentamos el tamaño del problema.

6.5.5 Resultados del algoritmo *PM*

En la Figura 6.19 y en la Figura 6.20 se muestra la eficiencia lograda por el algoritmo *PM* para los diferentes tamaños de secuencia y tamaños de bloque de datos probados empleando cuatro y ocho hojas de la arquitectura respectivamente. Se puede observar que ambos gráficos muestran resultados similares, con una leve superioridad por parte del primero en las eficiencias logradas.

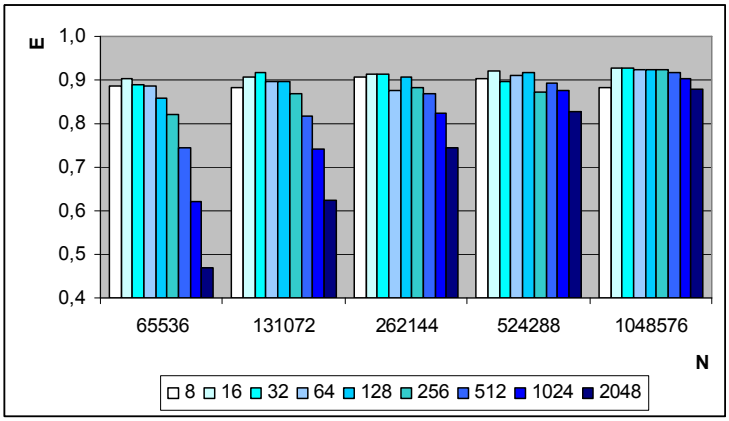

Figura 6.19. Eficiencia lograda por el algoritmo *PM* para diferentes tamaños de secuencia (N) y de bloques de datos usando 32 núcleos.

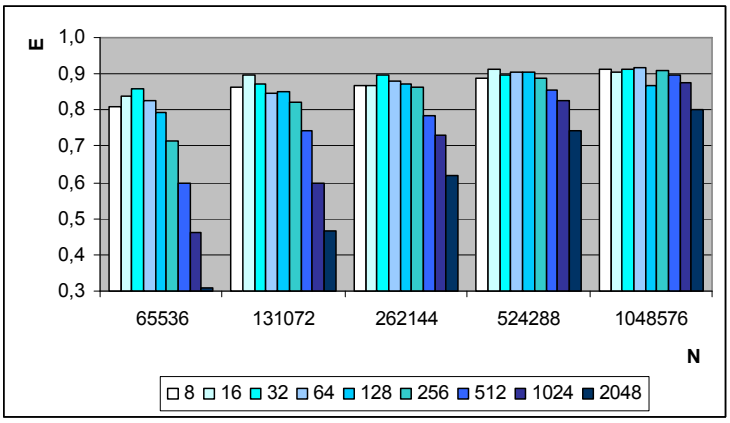

Figura 6.20. Eficiencia lograda por el algoritmo *PM* para diferentes tamaños de secuencia (N) y de bloques de datos usando 64 núcleos.

Los resultados del algoritmo *PM* utilizando ocho hojas de la arquitectura son similares a los descritos en la Sección 6.5.1, en el sentido de que no es posible identificar un tamaño ideal de bloque de datos, pero sí que los tamaños grandes de bloque de datos reducen la eficiencia lograda.

En la Figura 6.21 se presenta un resumen de la mejor eficiencia lograda por el algoritmo *PM* para diferentes tamaños de secuencia al utilizar cuatro y ocho hojas de la arquitectura. El algoritmo *PM* alcanza la mejor eficiencia con *TB* = 16 para *N* = {65536, 262144, 524288} y *TB* = 32 para *N* = {131072, 1048576} utilizando cuatro hojas. En cambio al usar ocho hojas de la arquitectura, se emplea *TB* = 16 para *N* = {131072, 524288}, *TB* = 32 para *N* = {65536, 262144} y *TB* = 64 para *N* = 1048576, para conseguir la mejor eficiencia.

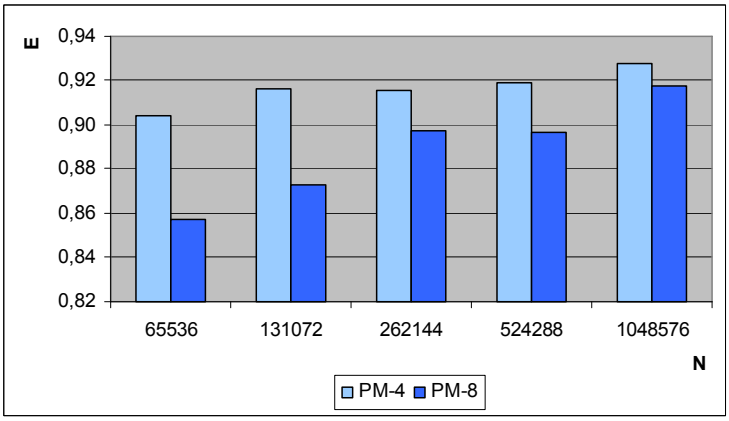

Figura 6.21. Resumen de la mejor eficiencia lograda por el algoritmo *PM* para diferentes tamaños de secuencia (N) con cuatro y ocho hojas de la arquitectura.

Se puede ver claramente en el gráfico anterior que se obtiene un incremento en la eficiencia conseguida a medida que se aumenta el tamaño de las secuencias. En sentido opuesto, la eficiencia se reduce al aumentar la cantidad total de núcleos. Dicha diferencia también se reduce al agrandar el tamaño del problema.

6.5.6 Comparación de los resultados de los algoritmos *PM*, *H1* y *H3*

En las secciones anteriores se compararon los algoritmos *H1* y *H2* por un lado, y *H3* y *H4* por otro. Del primer análisis se pudo concluir que *H1* obtiene en general un mejor rendimiento que *H2*. A partir del segundo análisis, se puede decir lo mismo de

H3 respecto a *H4*. Es por ello que, a continuación, se analizan los rendimientos de los algoritmos *H1*, *H3* y *PM*.

En la Figura 6.22 se presenta un resumen de la mejor eficiencia lograda por cada uno de los algoritmos (*PM*, *H1* y *H3*) para diferentes tamaños de secuencia al utilizar cuatro y ocho hojas de la arquitectura. Los tamaños óptimos de bloque de datos de cada algoritmo fueron detallados en las secciones anteriores.

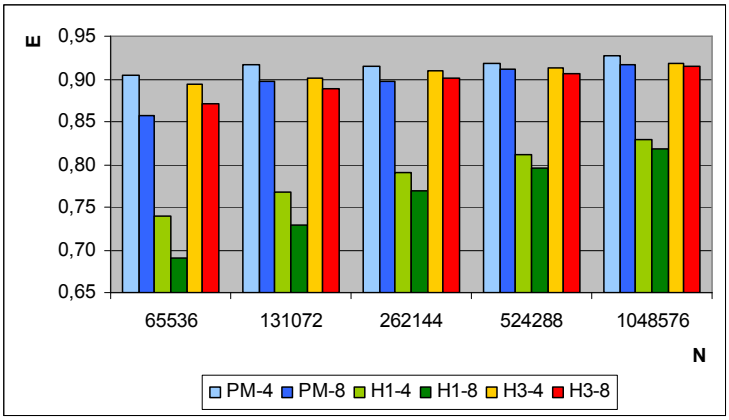

Figura 6.22. Resumen de la mejor eficiencia lograda por los algoritmos *PM*, *H1* y *H3* para diferentes tamaños de secuencia (N) con cuatro y ocho hojas de la arquitectura.

Este gráfico permite identificar que el algoritmo *PM* logra una mayor eficiencia con respecto a los híbridos. El rendimiento de *PM* y el de *H3* son muy similares, teniendo *PM* una leve ventaja. Tanto *PM* como *H3* superan ampliamente a *H1*. Esto se debe al tiempo que pasan ociosos los procesos e hilos que forman parte de la solución *H1*, lo cual no ocurre en *PM* y en *H3*. Si bien la técnica de aumentar el tamaño de bloque de datos ayuda a reducir el ocio, no lo elimina de forma completa. Se puede ver claramente la superioridad de *H3* con respecto a *H1* en la Figura 6.23, la cual muestra el porcentaje de mejora en la eficiencia del primero respecto del segundo.

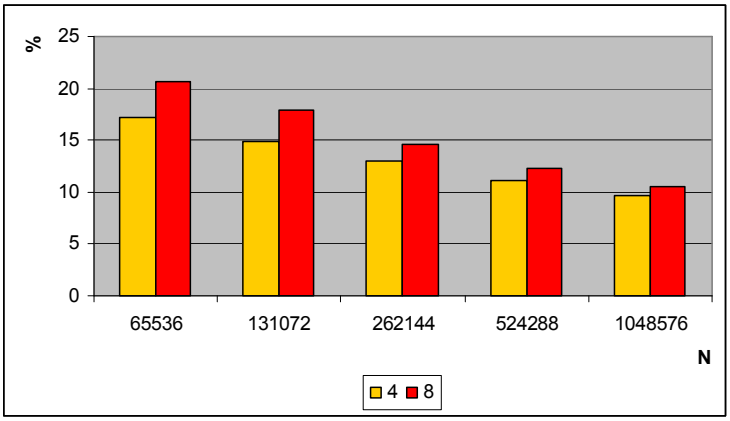

Figura 6.23. Porcentaje de mejora en la eficiencia obtenida por *H3* con respecto a *H1* para diferentes tamaños de secuencia (N) con cuatro y ocho hojas de la arquitectura.

Se puede notar en este gráfico que la diferencia entre las eficiencias logradas por *H1* y *H3* se reduce a medida que se aumenta el tamaño del problema y, en sentido opuesto, se agranda al incrementar la cantidad total de núcleos.

Si se vuelve a observar la Figura 6.22, se puede notar que los tres algoritmos reducen su eficiencia al incrementar la cantidad total de núcleos usados, aunque la diferencia se acorta al aumentar el tamaño del problema.

6.6 Conclusiones y trabajos futuros

En este trabajo se paraleliza el algoritmo de Smith-Waterman para el alineamiento de secuencias de ADN por medio de un esquema de pipeline debido a la dependencia de datos inherente al problema. Se utiliza como arquitectura de experimentación un cluster de multicores (ocho hojas con ocho núcleos cada una).

Dadas las características de la arquitectura se implementó - inicialmente - el pipeline con dos modelos de comunicación diferentes: pasaje de mensajes (*PM*) y memoria compartida (*MC1* y *MC2*). Se comparó la eficiencia de ambos algoritmos al utilizar una única hoja de la arquitectura, logrando una leve ventaja de *PM* respecto a *MC1* y *MC2*.

Pensando en los diferentes niveles de memoria (entre núcleos de un mismo nodo) y la red de interconexión (entre núcleos de diferentes nodos) de la arquitectura, se implementaron distintas alternativas (*H1*, *H2*, *H3* y *H4*) usando como modelo de comunicación un híbrido que combina pasaje de mensajes con memoria compartida. Los algoritmos *H1* y *H2* emplean un esquema de pipeline entre procesos que se comunican por pasaje de mensajes, y dentro de cada etapa existe un pipeline de memoria compartida para resolver cada bloque de datos. Es decir que implementan un esquema de pipeline de pipeline. Estos dos algoritmos se diferencian en la cantidad de hilos generados. Los algoritmos *H3* y *H4* utilizan un esquema de pipeline formado por procesos e hilos, donde aquellos que se encuentran en la misma hoja se comunican mediante memoria compartida, mientras que los que están en diferentes hojas lo hacen por pasaje de mensajes. Estos algoritmos se distinguen por el tipo de sincronización para memoria compartida empleado. Mientras que *H3* utiliza variables condición, *H4* usa semáforos.

Se analizó el comportamiento de estos algoritmos híbridos respecto a *PM* utilizando cuatro y ocho hojas completas de la arquitectura. Como resultado de este análisis se obtiene que *PM* logra una mayor eficiencia que los algoritmos híbridos. En el caso de *H3* y *H4*, la diferencia es mínima. No ocurre lo mismo con *H1* y *H2*, donde la diferencia sí es amplia. Los tiempos ociosos que pasan procesos e hilos en estas dos soluciones se pagan con una reducción en la eficiencia debido al overhead generado. La mínima diferencia entre las eficiencias de *PM* y de *H3* y *H4* se debe a dos factores. El primero es el poco requerimiento de memoria que tienen los algoritmos, lo cual no permite aprovechar los beneficios del uso de memoria compartida. El segundo factor es la optimización que poseen las librerías de pasaje de mensajes hoy en día para trabajar en ambientes de memoria compartida. La unión de los dos factores anteriores posibilita que *PM* tenga un mejor desempeño con respecto a *H3* y *H4*.

A partir del estudio y los resultados obtenidos en esta tesina, quedan abiertos un conjunto de temas para su futura investigación:

- analizar la escalabilidad del problema estudiado asegurando un determinado nivel de eficiencia;
- estudiar el comportamiento de la aplicación al utilizar ambientes heterogéneos, donde se combine el cluster de multicores empleado en éste trabajo con clusters heterogéneos tradicionales;
- y analizar y optimizar soluciones híbridas para determinadas clases de problemas, especialmente para los que admiten una solución paralela compuesta (combinando más de un paradigma de interacción).

Apéndice A

Detalle de los resultados

Como se mencionó en el Capítulo 6 se realizaron diferentes pruebas para cumplir con los objetivos de éste trabajo. A continuación se muestra el detalle de los mismos. Cabe destacar que todos los tiempos se midieron en segundos.

A.1 Algoritmo secuencial

N	65536	131072	262144	524288	1048576
	203,67961	814,549966	3258,201538	13063,06618	52459,6643

Figura A.1. Tiempos del algoritmo secuencial para diferentes tamaños de secuencia.

A.2 Algoritmo *MC1*

TB / N	65536	131072	262144	524288	1048576
8	40,841384	168,985435	650,891518	2698,576634	10572,007891
16	28,676411	113,264891	457,163535	1806,709656	7264,776880
32	28,335845	112,917017	451,241733	1801,235495	7206,079490
64	28,420570	113,117612	451,752807	1801,941892	7208,017539
128	28,612565	113,536173	452,348309	1803,961316	7209,216226
256	28,999497	114,288493	453,952437	1807,064748	7219,331335
512	29,768265	115,797326	456,987269	1813,266883	7232,435306
1024	31,297335	118,873959	463,269036	1825,657814	7254,104997
2048	34,378886	125,062581	475,629054	1849,906062	7304,274308

Figura A.2. Tiempos del algoritmo *MC1* para diferentes tamaños de secuencias y de bloque de datos al utilizar una hoja de la arquitectura.

A.3 Algoritmo *MC2*

TB / N	65536	131072	262144	524288	1048576
8	40,500094	166,850386	645,684399	2618,995387	10341,385106
16	28,593211	112,823627	454,835786	1802,080061	7280,197730
32	28,201876	112,408597	449,515116	1794,096005	7180,641494
64	28,293954	112,580162	449,896195	1794,781866	7182,569385
128	28,487531	112,954353	450,650237	1796,159441	7187,046837
256	28,869034	113,708210	452,144911	1799,147829	7192,981327
512	29,637347	115,260116	455,218328	1805,414817	7203,582323
1024	31,166375	118,317413	461,393368	1817,499225	7230,155167
2048	34,227556	124,449434	473,561810	1841,871765	7277,129305

Figura A.3. Tiempos del algoritmo *MC2* para diferentes tamaños de secuencias y de bloque de datos al utilizar una hoja de la arquitectura.

A.4 Algoritmo *PM*

TB / N	65536	131072	262144	524288	1048576
8	27,429649	113,547923	439,297144	1766,885024	7148,214216
16	27,427642	112,234955	440,521685	1761,051595	7219,398230
32	27,808116	112,013542	451,638183	1757,928818	7017,734959
64	27,699999	110,832858	443,290676	1761,465344	7004,775072
128	27,811045	112,073545	442,542075	1756,177435	7010,701085
256	28,130136	114,644664	446,111921	1760,443737	7016,640770
512	28,880499	112,793649	444,561406	1761,060301	7027,640963
1024	31,785889	115,619745	450,104507	1773,780283	7051,535397
2048	33,387062	121,710159	461,686852	1797,648381	7098,565664

Figura A.4. Tiempos del algoritmo *PM* para diferentes tamaños de secuencias y de bloque de datos al utilizar una hoja de la arquitectura.

TB / N	65536	131072	262144	524288	1048576
8	27,619386	109,754042	438,582577	1750,342423	7093,529362
16	27,739748	109,635364	438,162013	2577,403734	7164,478888
32	27,529489	109,639868	442,807829	1778,924148	7032,497791
64	27,698457	109,836023	441,226955	1750,960232	7002,204602
128	27,776975	110,168777	438,886116	1751,455672	7023,865705
256	28,160521	110,979729	440,538165	1756,528351	7015,611488
512	28,885925	112,316258	443,739225	1761,890339	7026,109063
1024	30,386001	115,354572	449,641035	1775,648603	7051,907187
2048	33,370210	121,334099	461,614905	1796,986759	7099,266527

Figura A.5. Tiempos del algoritmo *PM* para diferentes tamaños de secuencias y de bloque de datos al utilizar un núcleo de cada hoja de la arquitectura.

TB / N	65536	131072	262144	524288	1048576
8	7,193975	28,782723	112,262291	452,199473	1854,801511
16	7,040391	28,076960	111,231517	444,188166	1770,159680
32	7,158987	27,774989	111,429172	456,163365	1767,471524
64	7,170832	28,377215	116,274952	448,281242	1772,695671
128	7,427724	28,381969	112,372503	445,170612	1775,983005
256	7,755677	29,264194	115,381859	467,404973	1774,222946
512	8,545896	31,137874	117,153865	456,330675	1785,575856
1024	10,265888	34,320314	123,672333	465,838105	1811,596461
2048	13,599072	40,895225	136,795076	492,309060	1863,389121

Figura A.6. Tiempos del algoritmo *PM* para diferentes tamaños de secuencias y de bloque de datos al utilizar cuatro hojas de la arquitectura.

TB / N	65536	131072	262144	524288	1048576
8	27,619386	109,754042	438,582577	1750,342423	7093,529362
16	27,739748	109,635364	438,162013	2577,403734	7164,478888
32	27,529489	109,639868	442,807829	1778,924148	7032,497791
64	27,698457	109,836023	441,226955	1750,960232	7002,204602
128	27,776975	110,168777	438,886116	1751,455672	7023,865705
256	28,160521	110,979729	440,538165	1756,528351	7015,611488
512	28,885925	112,316258	443,739225	1761,890339	7026,109063
1024	30,386001	115,354572	449,641035	1775,648603	7051,907187
2048	33,370210	121,334099	461,614905	1796,986759	7099,266527

Figura A.7. Tiempos del algoritmo *PM* para diferentes tamaños de secuencias y de bloque de datos al utilizar ocho hojas de la arquitectura.

A.5 Algoritmo *H1*

TBpm / N	65536	131072	262144	524288	1048576
128	10,400004	41,420664	164,804477	656,514526	2619,883997
256	10,254433	36,009959	143,314860	568,110396	2264,477370
512	8,610819	33,698733	141,962846	525,635702	2211,810016
1024	8,677021	33,156804	128,936434	506,889651	2010,612669
2048	9,319366	34,514933	128,778212	502,747706	1976,367365

Figura A.8. Tiempos del algoritmo *H1* para diferentes tamaños de secuencias y de bloque de datos al utilizar cuatro hojas de la arquitectura (TB_{mc} = 16).

TBpm / N	65536	131072	262144	524288	1048576
128	5,364086	23,686547	83,006922	329,173067	1308,231885
256	4,755891	18,289769	72,545169	325,798012	1131,611608
512	4,612551	18,085690	74,028795	276,134277	1047,441159
1024	4,870367	17,450834	66,214573	256,658338	1012,671555
2048	5,643048	18,852402	68,532445	258,317954	1002,188637

Figura A.9. Tiempos del algoritmo *H1* para diferentes tamaños de secuencias y de bloque de datos al utilizar ocho hojas de la arquitectura (TB_{mc} = 16).

A.6 Algoritmo *H2*

TBpm / N	65536	131072	262144	524288	1048576
128	14,044586	55,952963	222,968212	892,137397	3556,236354
256	10,827990	43,078534	170,798580	689,282985	2731,203394
512	9,326696	36,973697	147,094646	582,608936	2326,644057
1024	8,681576	34,051145	146,305889	533,150478	2240,310838
2048	8,607844	32,998916	128,894056	508,236264	2018,484407

Figura A.10. Tiempos del algoritmo *H2* para diferentes tamaños de secuencias y de bloque de datos al utilizar cuatro hojas de la arquitectura (TB_{mc} = 16).

TBpm / N	65536	131072	262144	524288	1048576
128	7,237421	28,622654	120,389329	453,646518	1811,425704
256	5,605427	22,011558	87,116071	347,333515	1384,054797
512	4,849889	18,800240	73,960917	293,804753	1170,947657
1024	4,631395	17,545825	71,049762	269,153667	1067,858876
2048	4,823400	18,108061	66,466887	258,862344	1020,982918

Figura A.11. Tiempos del algoritmo *H2* para diferentes tamaños de secuencias y de bloque de datos al utilizar ocho hojas de la arquitectura (TB_{mc} = 16).

A.7 Algoritmo *H3*

TB / N	65536	131072	262144	524288	1048576
8	8,597433	37,610591	140,393330	578,336333	2288,371888
16	7,416787	28,242195	112,028029	470,185634	1784,634623
32	7,124051	28,229066	112,051572	447,138669	1859,416175
64	7,208336	28,399662	112,051572	447,817991	1786,241981
128	7,407740	28,797171	113,160727	449,530273	1788,743454
256	7,817559	29,650844	114,843200	452,660326	1795,158014
512	8,656714	31,293734	118,129220	459,367855	1809,424757
1024	10,323781	34,640227	124,897391	472,721791	1835,720415
2048	13,667836	41,325029	138,227609	499,600585	1889,636093

Figura A.12. Tiempos del algoritmo *H3* para diferentes tamaños de secuencias y de bloque de datos al utilizar cuatro hojas de la arquitectura.

TB / N	65536	131072	262144	524288	1048576
8	6,231465	17,213608	62,853125	251,411084	1166,293634
16	3,656047	14,778599	56,530299	225,120775	896,278792
32	3,658821	14,322764	56,533088	225,161441	896,284396
64	3,754359	14,490901	56,879360	225,831888	897,626836
128	3,944701	14,906460	57,710634	227,434247	900,858562
256	4,363049	15,735794	59,365241	230,782293	907,563103
512	5,214088	17,433568	62,769471	237,605567	921,106381
1024	6,911896	17,433568	69,557419	251,180664	948,154047
2048	10,321217	27,635440	83,185050	278,447825	1002,586998

Figura A.13. Tiempos del algoritmo *H3* para diferentes tamaños de secuencias y de bloque de datos al utilizar ocho hojas de la arquitectura.

A.8 Algoritmo *H4*

TB / N	65536	131072	262144	524288	1048576
8	8,568121	30,211979	116,894303	459,978081	1831,134120
16	7,166493	28,287440	112,639361	448,080297	1793,091647
32	7,173557	28,279302	112,720791	448,101796	1793,050427
64	7,241847	28,445930	112,912980	448,732001	1875,516480
128	7,440439	28,842186	113,716236	465,013858	1797,660059
256	7,853724	29,672455	115,349864	453,614535	1804,026426
512	8,684864	31,336280	118,673831	460,513097	1817,491878
1024	10,347183	34,673049	125,330156	473,620646	1844,130805
2048	13,685967	41,360330	138,694015	500,503935	1897,509527

Figura A.14. Tiempos del algoritmo *H4* para diferentes tamaños de secuencias y de bloque de datos al utilizar cuatro hojas de la arquitectura.

TB / N	65536	131072	262144	524288	1048576
8	6,355745	17,019490	60,845480	233,152682	921,270300
16	3,668526	14,353698	56,751049	225,669803	941,082858
32	3,778665	14,367588	56,776678	225,697797	899,879921
64	3,761145	15,132002	57,121357	226,408504	901,194006
128	3,960739	14,958742	57,898603	228,022042	904,536317
256	4,383121	15,776021	59,587173	231,334016	911,021398
512	5,230337	17,471190	62,980816	238,158758	924,563582
1024	7,085780	21,524211	69,789607	251,737967	951,839394
2048	10,417834	27,658895	83,394061	278,925718	1006,203194

Figura A.15. Tiempos del algoritmo *H4* para diferentes tamaños de secuencias y de bloque de datos al utilizar ocho hojas de la arquitectura.

Referencias

[ABB05] Abbas A, “Grid Computing. A Practical Guide to Technology and Applications”, Firewall Media, 2005.

[AND00] Andrews G R, “Foundations of Multithreaded, Parallel, and Distributed Programming”, Addison Wesley Higher Education, 2000.

[ATT02] Attwood T K, Parry-Smith D J, “Introducción a la bioinformática”, Prentice Hall, 2002.

[BAR10] Barney B, “Introduction to parallel computing”, Lawrence Livermore National Laboratory, 2010.

[CHA07] Lei Chai, Qi Gao, Dhabaleswar K Panda, “Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System”, IEEE International Symposium on Cluster Computing and the Grid 2007 (CCGRID 2007), pp. 471-478, 2007.

[COR90] Cormen T H, Leiserson C E, Rivest R L, “Introduction to algorithms” , 1era Edición, MIT Press and McGraw-Hill, 1990.

[GRA03] Grama A, Gupta A, Karypis G, Kumar V, “Introduction Parallel Computing”, Pearson Addison Wesley, 2da Edición, 2003.

[GUS88] Gustafson J L, “Reevaluating Amdahl's Law”, CACM, 31(5), 1988.

[KI05] Ando Ki, “Memory Latency Hiding Techniques”, 2005.

[LAM79] Lamport L, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE Trans. Comput. C-28,9, 1979.

[MCC07] Mc Cool M, “Programming models for scalable multicore programming”, 2007, http://www.hpcwire.com/features/17902939.html.

[MOH02] Sang-Man Moh, Woo-Jong Hahn, Suk-Han Yoon, “Cache Coherence Protocols in NUMA Multiprocessors”, 2002.

[OHI96] Ohio Supercomputer Center, “MPI Primer / Developing with LAM”, The Ohio State University, 1996.

[RAU10] Rauber T, Rünger G, “Parallel programming for Multicore and Cluster Systems”, Springer, 2010.

[SUN04] Sun Microsystems, “OpenMP API User’s Guide”, 2004.

[TRO09] Trobec R, Vajtersic M, Zinterhof P, “Paralell Computing”, Springer, 2009.

[WIL05] Wilkinson B, Allen M, “Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers”, 2da Edición, Pearson Prentice Hall, 2005.