Tesina del Máster en Ingeniería de Computadores

Adecuación de la granularidad de las comunicaciones en aplicaciones MPI a las características de la red de interconexión

Héctor Montaner Mas

Directores:
Federico Silla Jiménez
Vicente Santonja Gisbert

14 de diciembre de 2007

Abstract

Este estudio trata sobre ciertos aspectos del modelo de comunicaciones que siguen los algoritmos paralelos basados en la interfaz Message Passing Interface (MPI). Más concretamente, el estudio se centra en la granularidad de los envíos, es decir, el tamaño (e inherentemente el número) de los mensajes intercambiados entre procesos. Se parte de la hipótesis de que particionar un mensaje grande en varios más pequeños permitirá a los procesos receptores comenzar su etapa de cómputo con antelación. De este modo, pasar de granularidad gruesa a fina puede significar una aceleración en el tiempo total de ejecución. Además, esta norma varía al pasar de una red lenta (penalización constante por mensaje) a una red rápida (sin penalización), lo que establece como objetivo del estudio demostrar que la granularidad óptima en una red lenta no sigue siendo tal en una red rápida.

Se presentan los patrones clave de comunicación interprocedural que hacen susceptible de ser optimizada, mediante el uso de granularidad fina, a la aplicación que los adopta. A continuación, se describen unos ejemplos de estas aplicaciones y se simula su ejecución en red lenta y rápida pasando por todo un rango de granularidades, lo que nos proporcionará los resultados para estudiar la potencia de una disminución del grano de las comunicaciones.

Finalmente se da una visión global de los pros y contras de esta variación de la granularidad. Se remarcan algunos inconvenientes no contemplados a simple vista, como que el coste de manejar un número elevado de mensajes puede contrarrestar la aceleración conseguida a base de envíos de pequeño tamaño. También se discute la posible ganancia en escalabilidad, aunque se deduce que este concepto es ajeno al programador y por tanto no tiene lugar en este estudio. Además, como conclusión final, se indica que la granularidad óptima en una red lenta está muy cerca de la óptima en una red rápida, con lo que la aceleración no resulta notoria.

Índice general

Capítulo 1

Introducción

Típicamente, el concepto de clúster de computadores y de multicomputador ha estado ligado a la paralelización de las aplicaciones. La disponibilidad de grandes capacidades de cálculo ha impulsado el diseño de aplicaciones que distribuyen el cómputo entre varios nodos aprovechando todos los recursos disponibles, más aún cuando su ejecución serializada es inviable.

Una de las características más comunes de estos sistemas de cómputo es su memoria distribuída. Esto implica que los distintos hilos de ejecución solo tienen acceso directo a su propia memoria, no a la del resto. Por este motivo, el paso de mensajes es un paradigma de comunicación ampliamente empleado en este tipo de sistemas. Como un proceso no puede acceder por su cuenta a la memoria de otro proceso, el intercambio de información se realiza a través de mensajes. Se dice que se emplean mensajes explícitos debido a que tanto el emisor como el receptor deben ejecutar el código adecuado para que el mensaje viaje de uno a otro proceso, o dicho de otro modo, el programador debe escribir el código necesario para que cada proceso maneje las pateciones de información que recibe.

Gracias a la posibilidad de este intercambio de datos, un problema complejo puede dividirse en subproblemas que cada proceso solventará en paralelo, y si en algún punto de su ejecución los procesos necesitan basarse en cálculos de otros procesos, se los intercambiarán mediante envíos de mensajes a través de la red de interconexión.

En un principio, cada fabricante proveía a su sistema de una implementación propia de paso de mensajes, adaptada a las especifaciones de un clúster o multicomputador determinado. Obviamente, esto era un problema a la hora de portar aplicaciones de un sistema a otro. Sin embargo, la semántica del paso de mensajes era prácticamente la misma. Por este motivo, se realizaron esfuerzos para implementar un sistema de paso de mensajes que fuese eficiente y portable. De aquí nació MPI (*Message Passing Interface*) ([1] [7]), con el ánimo

de establecer una interfaz estándar para la comunicación por mensajes explícitos.

Estandarizar una interfaz implica que el programador que la emplee no sepa sobre qué tipo de sistema se ejecutará finalmente su aplicación. En concreto, el programador debe preparar su aplicación para que pueda ejecutarse sobre distintos tipos de red de interconexión, como por ejemplo Gigabit Ethernet, Infiniband, Myrinet, etc. Además, independientemente de la red, el programador deberá tener en cuenta la latencia, mayor o menor, que añada el sistema de interconexión. Es más, no es descartable que la aplicación termine por utilizar sockets TCP comunes, lo que quiere decir que el mensaje deberá atravesar toda la pila TCP/IP. Esto, unido a otros factores como el retardo insertado en llamadas a la biblioteca de MPI y demás, provocan que toda llamada a la interfaz de comunicaciones tenga asociado un tiempo mínimo que no se pueda rebajar aun disminuyendo el volumen de datos a enviar.

Por consiguiente, el programador debe atender a estos factores a la hora de programar una aplicación paralela. Por ejemplo, debido a la penalización inherente al envío de mensajes, es recomendable que el número de mensajes a enviar sea mínimo. Esto equivale a retardar el envío de datos tanto como sea posible para que con un solo mensaje se pueda agrupar la mayor cantidad de información. De forma práctica, enviar un mensaje de 100 kB requiere menos tiempo que enviar 100 mensajes de 1kB, porque cada mensaje individual, independientemente del tamaño, debe ser sometido a una serie de gestiones que introducen un retardo constante en el proceso de envío. El programador es el encargado de disimular este retardo tan molesto creando aplicaciones que eviten a toda costa estos inconvenientes.

No obstante, esta filosofía de diseño de las aplicaciones puede no ser la más indicada cuando el contexto de ejecución cambia. Esta incógnita surge a raíz de la introducción en el mercado de los chips *multi-core*. En principio, la estructura de ambos escenarios es muy similar: una serie de nodos de procesamiento unidos por una red; es por esto por lo que cabe esperar que las mismas aplicaciones que se ejecutaban en un clúster o multicomputador puedan también hacerlo en este tipo de chip. De hecho, las guías de diseño de los futuros sistemas on-chip contemplan el paso de mensajes como un mecanismo a ser incroporado. En [8] se habla de permitir la comunicación entre nodos vía paso de mensajes dirigida directamente por el programador, y en [6] se indica que los sistemas on-chip del futuro deberán soportar tanto coherencia de memoria compartida como paso de mensajes, debido a la diversidad tanto de los requerimientos de las aplicaciones como de las formas de programar el software que se ejecutará en estos sistemas.

Obviamente, existen una serie de diferencias entre los dos escenarios, y una de las más relevantes para este estudio es el tipo de red empleada. No es necesario decir que la red interior al chip (*on-chip*) es muchísimo más rápida que la red con la que puede estar equipado

un clúster, sin embargo, esta diferencia tan patente no es la promotora de este análisis. La diferencia que puede implicar un cambio en el diseño de las aplicaciones (concepto sobre el que gira toda esta disertación) es la penalización por envío que se comentaba anteriormente y de la que está exenta una red interior al chip. Como se ha comentado, se espera que los sistemas on-chip ofrezcan soporte para paso de mensajes, incluyendo en su juego de instrucciones algunas dedicadas a este fin, al igual que existen instrucciones para leer o escribir datos de memoria.

Esta última característica contradice la filosofía de diseño que siguen las aplicaciones MPI a ejecutar sobre clústers, ya que dentro del chip no es necesario minimizar el número de mensajes. Dentro del chip, las comunicaciones entre cores no deben atravesar la pila TCP, es más, una operación de alto nivel de MPI se traducirá a un número muy reducido de instrucciones en ensamblador. Esto abre la posibilidad de enviar un mayor número de mensajes con menor cantidad de información en cada uno de ellos. Esta idea es lo que se conoce como granularidad fina, frente a la granularidad gruesa que trata de enviar el mínimo número de mensajes con la mayor cantidad de información. Como se puede apreciar, este cambio está acercando el tipo de tráfico generado por MPI con el generado por el uso de memoria compartida, con la diferencia de que el envío de información se hace de forma explícita e implícita respectivamente.

En líneas generales, enviar mensajes más pequeños permitirá a los procesos receptores computar cuando, con un esquema de granularidad gruesa, estarían esperando. A lo largo de este estudio se presentará una serie de patrones clave de comunicación entre procesos. Las aplicaciones que sigan estos patrones serán susceptibles de ser aceleradas pasando de granularidad gruesa a fina, es decir, desglosando los mensajes grandes en mensajes de menor tamaño. Además, se describirá una serie de aplicaciones que siguen estos patrones y se implementarán parametrizando el grano de las comunicaciones. Para estudiar el tiempo de ejecución en los escenarios mencionados, las aplicaciones se simularán para obtener unos resultados empíricos con que contrastar las suposiciones teóricas.

A lo largo de las siguientes páginas, se espera demostrar que en una red off-chip la granularidad gruesa es la óptima, mientras que en la red on-chip, libre de penalizaciones por envío, una granularidad fina permite una ejecución más rápida.

Capítulo 2

La Hipótesis

Este estudio se desarrolla entorno a una hipótesis cuya veracidad se intentará demostrar. Para entender la idea subyacente vamos a introducir los dos escenarios clave en este estudio, que al mismo tiempo se caracterizan básicamente por la red de interconexión que emplean, que puede ser:

- Red lenta exterior al chip.
- Red rápida interior al chip.

En pocas palabras, una red exterior al chip (Infiniband, Gigabit Ethernet, Myrinet, etc) puede ser más o menos sofisticada pero siempre será más lenta (al menos uno o dos órdenes de magnitud) que la red dentro del chip. Sin embargo, este aspecto no es la diferencia fundamental entre los dos escenarios. Lo que distingue estas dos redes y lo que constituye la piedra angular de nuestra hipótesis es que la red exterior al chip inserta una penalización a las comunicaciones de la cual la red on-chip está exenta. De aquí en adelante, se empleará el término *red lenta* para referirse a una red que introduce esta penalización en los envíos y se usará el término *red rápida* para las redes que no tengan este sobrecoste.

Cuando se desea enviar un mensaje a través de una red lenta, este mensaje va a atravesar una serie de etapas que generarán la penalización comentada. Desde que el programa realiza la llamada a una operación de comunicación determinada hasta que el mensaje sale finalmente por la interfaz de red, el camino a seguir puede ser muy largo, por ejemplo, tiene que atravesar la biblioteca de comunicaciones (MPI), después debe recorrer la pila de comunicaciones que corresponda (si la comunicación se hace por sockets generalmente será la pila TCP/IP) y demás retardos que constituyen una penalización para cada mensaje (el mismo camino se correrá en el destino en sentido inverso). Este retardo es prácticamente

independiente del tamaño del mensaje, lo que conlleva a decir que la red exterior al chip introduce un sobrecoste constante en cada envío.

En contrapartida, una red on-chip no funciona de la misma manera si se proporciona el soporte adecuado. Una llamada a una operación de comunicación se puede traducir en una o unas pocas instrucciones en ensamblador que se encargarán de mover la información de un sitio a otro. Además, no existe ningún tipo de pila a recorrer, lo que en total produce un coste por mensaje proporcional al tamaño del mismo. Es decir, el coste del envío de una palabra es menor que el envío de dos, cuando en una red externa es posible que cueste lo mismo por ser mensajes muy cortos.

Una vez visto esto, tenemos que para sistemas con una red lenta se han diseñado aplicaciones que intentan evitar los problemas de estas redes. Esto significa que, debido a que tanto un mensaje grande como un mensaje corto van a tener implícito un retardo constante, lo mejor sería enviar cuanto menos mensajes mejor. A su vez, esto implica que los mensajes que se envíen sean lo más grande posible. De esta forma, la penalización total debida al número de mensajes se reduce. Los programadores de MPI conocen estas técnicas y por eso las aplicaciones intentan ser lo más independientes posible de la red.

Existen numerosas aplicaciones MPI, sobre todo en temas científicos, como biología, química, física, etc, con un gran uso actual. ¿Qué ocurre si estas aplicaciones se desean portar a un entorno de sistema on-chip? Ya que en un futuro dispondremos de un chip con una serie de núcleos capaces de ejecutar aplicaciones MPI, debemos preguntarnos cómo este cambio de contexto afectará al rendimiento de las aplicaciones que fueron diseñadas adrede para ejecutarse en un entorno donde el envío de mensajes sufre penalización.

Como hemos visto, la diferencia entre escenarios radica principalmente en la penalización por mensaje que no aparece en la red on-chip. Esto lleva a pensar que el único cambio que experimentará la aplicación portada a red on-chip será una más rápida ejecución, tanto por la no existencia de esta penalización como por el mayor ancho de banda y menor latencia existente.

Esto es cierto, sin embargo, la pregunta es: ¿esta aceleración que sufre la aplicación es la mayor posible? Esta cuestión surge a raíz de la marcada filosofía de diseño que tienen las aplicaciones MPI, como ya se ha dicho, intentar enviar el mínimo número de mensajes. Parece que dentro del chip ya no es necesario preocuparse tanto por este hecho ya que los envíos de poca información no sufren penalización. Esto deriva en la siguiente duda: ¿la forma de programar que era óptima para ejecuciones en clústers, es también óptima para ejecuciones en redes on-chip?

En resumidas cuentas, en una red on-chip tenemos la posibilidad de enviar tantos mensajes como se desee sin que el hecho en sí repercuta en el tiempo total de ejecución. Esta nueva facilidad es posible que permita rediseñar el modo en que las aplicaciones se comunican para que requieran un menor tiempo de ejecución.

Vamos a profundizar un poco más en el concepto de acelerar la ejecución. Grosso modo, un proceso puede estar en dos fases principalmente: ejecutándose y no ejecutándose. Se dice que un proceso está en ejecución cuando se encuentra dentro del procesador, es decir, la CPU está ejecutando instrucciones del proceso. Hay veces en que el proceso no se encuentra dentro del procesador, por ejemplo cuando es expulsado por el planificador del sistema operativo, o cuando el proceso está esperando una operación de entrada/salida o alguna situación similar.

Para un procesador dado, no se puede reducir el periodo de tiempo en que el proceso está computando, ya que en este estudio no estamos tratando el tema de procesadores. Como aquí se habla de redes, lo que vamos a intentar mejorar es el aspecto de redes, es decir, el tiempo en que un proceso está esperando algún evento de red. De esta forma, el objetivo de todo esto es que cambiando la filosfía de programar las comunicaciones el tiempo que los procesos estaban esperando se pueda reducir, con lo que se obtendría una aceleración global.

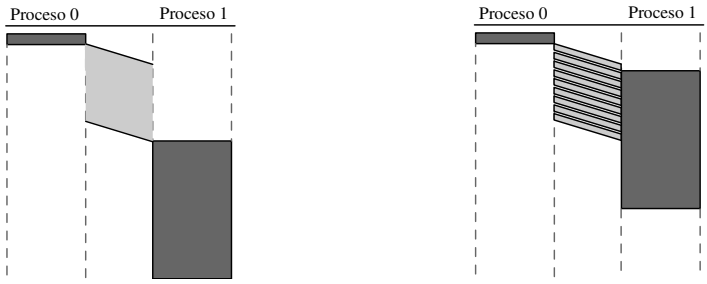
Es hora de introducir los conceptos de granularidad gruesa y granularidad fina. La granularidad gruesa refleja la forma en que se comunican los procesos en un entorno de clúster, es decir, pocos mensajes de gran tamaño. La granularidad fina consiste en enviar el máximo número posible de mensajes pequeños. De forma genérica, pasar de granularidad gruesa a fina consiste en descomponer el envío de un mensaje en pequeños envíos.

Este concepto de descomposición se puede observar desde dos puntos de vista. Para ilustrarlos adecuadamente, vamos a ver un par de ejemplos.

La primera situación es la siguiente: un proceso llega a un determinado momento en que debe enviar una cierta cantidad de información al resto de procesos (para ser precisos, a cada proceso le corresponde una información distinta) (figura 2.1). Con granularidad gruesa, este proceso encapsularía toda la información en un solo mensaje (uno por proceso) y lo enviaría al destinatario. En cambio, en términos de granularidad fina, este envío se descompondría en mensajes más pequeños.

Ahora, si los demás procesos estaban esperando el mensaje para empezar a computar sobre esos datos, en el caso de granularidad gruesa, hasta que no llegue el mensaje entero no podrán empezar. En cambio, con granularidad fina, reciben el primer mensaje corto en poco tiempo, con lo que es posible que los procesos puedan comenzar con el cómputo

sobre una parte de los datos. De este modo, cuando los procesos terminen con el cómputo del primer mensaje pequeño, es posible que ya les haya llegado el siguiente, con lo que pueden seguir computando inmediatamente.



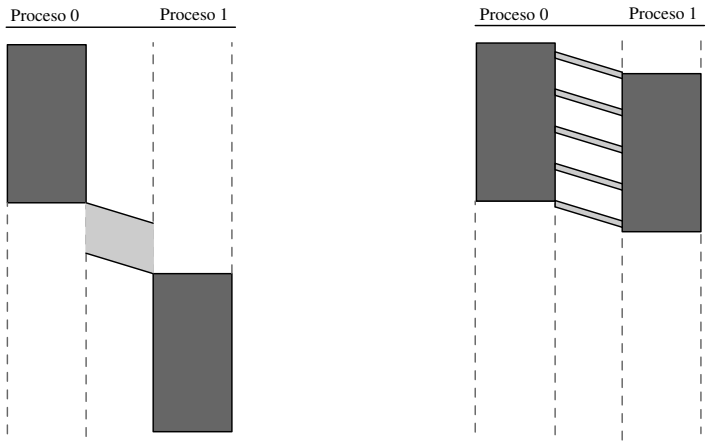
(a) Situación 1 (b) Situación 2

Figura 2.1. Ejemplo de paso de mensajes entre procesos: a la izquierda tenemos un ejemplo de granularidad gruesa y a la derecha uno de granularidad fina

Lo que conseguimos es que con granularidad fina el cómputo comience en cuanto llegue el primer mensaje pequeño, frente a granularidad gruesa en donde los procesos deben esperar a que los mensajes grandes sean recibidos completamente. Supongamos que el proceso que envía los datos tiene tan solo un canal de salida por el que enviar mensajes. Esto significa que el envío de mensajes se serializa, con lo que en granularidad gruesa se enviará primero un mensaje, después el siguiente, etc. Si la cantidad total de información a enviar asciende a, por ejemplo, 10 megabytes, y el número de procesos es 10, el proceso que recibe el último mensaje deberá esperar a que se envíen los 9 megabytes anteriores para poder empezar él con su cómputo. Si esto lo traducimos a granularidad fina, y en cada mensaje enviamos 10 bytes, el último proceso en recibir la primera fracción de información puede empezar a computar después de que el origen haya enviado 90 bytes.

El ejemplo anterior refleja un solapamiento de la recepción con el cómputo, lo que apunta a una clara aceleración en el tiempo total de ejecución. El segundo punto de vista mencionado, refina la situación anterior y se plasma en el siguiente ejemplo (figura 2.2).

Imaginemos ahora una situación parecida a la anterior, en donde un proceso llega al instante en donde debe repartir información al resto. Anteriormente nos hemos centrado en que los procesos pudiesen recibir la información en cuanto antes, aunque solo fuese una fracción. Pero puede darse el caso en que esa información a repartir haya sido computada con anterioridad por el proceso emisor. Esto quiere decir que es posible adelantar aún más el proceso de reparto y es especialmente recomendable cuando el proceso receptor está esperando también con anterioridad. En esta ocasión, la información a repartir se enviará en



(a) Situación 1 (b) Situación 2

Figura 2.2. Ejemplo de paso de mensajes entre procesos: a la izquierda tenemos un ejemplo de granularidad gruesa y a la derecha uno de granularidad fina

cuanto esté disponible, es decir, en cuanto sea calculada. Como se ve en la ilustración, en el proceso de cómputo de la información a enviar se intercalarán los envíos de esta información. Como se puede observar, de esta forma se acelera en gran medida la ejecución total, mucho más que en el ejemplo anterior.

Esto mismo se puede extender al caso en donde varios procesos envían al mismo destino; en vez de una operación de reparto ahora es una operación de recolección. Lo visto hasta ahora es igualmente aplicable a esta situación: fragmentar el envío y repartirlo a lo largo del tiempo de cómputo, permite que la red no se congestione en un cierto momento ya que se hace un uso constante y repartido de la red.

En resumidas cuentas, si en una aplicación se dan las situaciones que se han descrito, para ejecución en clúster es mejor una granularidad gruesa, que camufla las penalizaciones por envío que el sistema provoca, pero para ejecución en redes on-chip es mejor una granularidad fina que permite reducir el tiempo de espera de los procesos. Esto que se acaba de formular es la hipótesis sobre la que se va a trabajar a lo largo de este estudio.

Es tremendamente importante definir qué es lo que se quiere estudiar, qué es lo que se quiere comparar. Formulado brevemente se quiere demostrar que: ***la granularidad óptima en un clúster no resulta ser la óptima en un entorno de red on-chip***. Se quiere hacer hincapié en nuestro objetivo porque es susceptible de ser malinterpretado, tal y como se comentará más adelante.

Todos estos conceptos se ilustrarán en los capítulos posteriores a través de una serie de aplicaciones y de su comportamiento bajo distintas condiciones de granularidad.

Capítulo 3

Metodología

Para poder corroborar la hipótesis que en este estudio se plantea, resulta necesario lanzar una serie de ejecuciones en unos determinados escenarios. Estamos hablando de ejecuciones en clústers de computadores con un número elevado de procesadores y también de ejecuciones en chips multi-core. Existe un problema con esto y es que en el primer caso, debido a que se requiere de un sistema completamente dedicado, puede ser difícil encontrar los recursos suficientes. El segundo caso presenta mayor dificultad porque no se dispone de un chip con múltiples cores (8, 16, 32, etc). Por estos motivos, es necesario una simulación del escenario requerido. La metodología de simulación consta de dos partes principales:

- Ejecución de las aplicaciones y obtención de trazas.
- Simulación de la red con las trazas anteriores.

Durante la primera parte, las aplicaciones a estudiar se ejecutan de forma real, esto es, se lanzan sobre un procesador real en una red real. De hecho, no importa sobre qué plataforma se lancen, no importa ni el tipo de red ni si los procesadores, la memoria o la red misma están siendo compartidos por otras aplicaciones. De estas ejecuciones solo interesa obtener un listado de eventos de comunicación; al final de la primera parte tendremos unos archivos en los que vendrán indicados cada uno de los mensajes que se han intercambiado cada uno de los procesos. Estos eventos se describen mediante ciertos parámetros como son el origen del mensaje, el destino o los destinos, el tamaño en bytes y una marca de tiempo. Esta marca de tiempo debe ser independiente del tipo de red, y lo que es más importante, independiente al hecho de que el proceso puede estar compartiendo procesador con otras aplicaciones. Esto significa que la marca temporal se debe basar en un contador de tiempo que no atienda al tiempo de pared sino al tiempo de procesamiento (esto se explicará a continuación con más detalle). Gracias a todo esto, es posible ejecutar las aplicaciones con

cualquier número de procesos en una máquina con un número reducido de procesadores, sin que esto interfiera en los resultados finales.

En la segunda etapa, se simula el paso de los mensajes a través de una red con unas características determinadas. Con este proceso se obtienen los tiempos de red que complementan a los tiempos de cómputo calculados en la etapa anterior. Una vez que se dispone de todos los tiempos, se puede calcular cuál es el tiempo total de ejecución, que es el que nos servirá para comparar las distintas versiones de las aplicaciones a estudiar.

La principal ventaja de esta metodología es el desacople que existe entre las dos etapas. Esto es beneficioso a la hora de construir todo el sistema ya que permite que sea un desarrollo progresivo. Además, a la hora de simular, una vez obtenidas las trazas para una aplicación y para un número de procesos dado, se puede simular cualquier tipo de red sin tener que repetir la primera fase. Esto es especialmente deseable si la ejecución es muy costosa y se requiere probar distintos parámetros de la red.

Cabe remarcar la diferencia existente entre unas trazas comunes y éstas: en las comunes, cada evento está programado para un instante determinado, independientemente de la simulación de la red. En cambio, las trazas que nos ocupan no programan los envíos y recepciones de igual manera, sino que establecen, mediante el tiempo de cómputo, un tiempo mínimo de separación entre un determinado evento y el siguiente; es el propio simulador de red quien determinará finalmente en qué instante de tiempo se producen verdaderamente los eventos, añadiendo a los tiempos de cómputo el tiempo de espera.

A continuación se describe con más detalle cada elemento del sistema.

3.1. El contador de tiempo

El elemento fundamental sobre el que se basa el resto de la metodología es el contador de tiempo. Necesitamos un contador que nos indique el tiempo que un proceso ha estado en ejecución, es decir, dentro de la CPU. Además, una buena precisión en este cálculo del tiempo es imprescindible.

Como el sistema operativo sobre el que se va a lanzar la simulación es Linux, vamos a ver qué herramientas de medición de tiempo nos ofrece.

- *gettimeofday*, es una llamada al sistema que devuelve, con una resolución máxima de microsegundos, el tiempo actual de pared. La precisión es bastante aceptable, el problema aquí es que no nos proporciona el tiempo que el proceso ha permanecido en estado de ejecución. Con este contador no podemos lanzar ejecuciones de varios procesos en un solo procesador porque la toma de tiempos no sería independiente.

- También encontramos las llamadas *times* y *clock*. Mediante *times* podemos averiguar cuanto tiempo el proceso invocante ha estado ejecutándose, y lo que es más, distingue este tiempo entre tiempo de usuario y tiempo de sistema. Con *clock* obtenemos lo mismo, pero el tiempo no viene desglosado (de hecho, *clock* se basa en *times* para obtener los tiempos). En esta ocasión sí que obtenemos lo que queremos, sin embargo, la precisión de estas llamadas es muy pobre ya que estamos hablando de milisegundos en el mejor de los casos (en los sistemas actuales, estos contadores se actualizan cada 10 milisegundos). Esta precisión no es nada adecuada para nuestros propósitos, en donde el envío de un par de mensajes puede separarse por menos de un milisegundo.

  Adicionalmente, la fiabilidad de la llamada *times* es bastante pobre ya que se puede fácilmente realizar un programa que se sincronice con la toma de tiempos de dicha llamada y provocar que *times* nos indique que el proceso prácticamente no ha consumido recursos cuando la realidad es que casi todo el tiempo ha estado ocupando la CPU. Un ejemplo de esto puede ser un programa que realice una cierta cantidad de cálculo y que intercalado con este cálculo se introduzcan llamadas a *sleep*. Si la frecuencia con que el proceso se duerme es suficiente, el tiempo que el proceso está computando entre llamadas a *sleep* es muy reducido, lo que provoca que *times* no tome adecuadamente las medidas de tiempo e indique que ese proceso no ha consumido CPU.

De esta forma, nativamente un sistema operativo Linux estándar no ofrece la funcionalidad requerida. Tras analizar el estado del arte en materia de instrumentación de toma de tiempos, no hemos encontrado ninguna solución que, de forma clara y sencilla, nos permitiese disponer del contador mencionado. Por este motivo, hemos decidido diseñar e implementar por nosotros mismos las modificaciones necesarias en el kernel. Aquí presentamos la definición de la llamada al sistema resultante:

```
int my_cpu_time(unsigned long long *cpu_ticks);
```

my_cpu_time calcula el tiempo que el proceso invocante ha pasado en ejecución desde que fue creado. Devuelve en la variable *cpu_ticks* el número de ciclos de procesador que el proceso ha estado ejecutándose. Si la llamada tiene éxito, *my_cpu_time* devuelve 0 y si surge cualquier problema devolverá 1. En comparación con la llamada *time*, *my_cpu_time* devuelve el tiempo de usuario más el tiempo de sistema, pero a diferencia de *time*, *my_cpu_time* no lo calcula en milisegundos, sino en ciclos de procesador (el número de ciclos por segundo viene determinado por la frecuencia a la que opera el procesador). Para obtener el número

de ciclos de procesador, se ha empleado la instrucción de ensamblador x86 *rdtsc* (*read time stamp couter*). Es interesante indicar que el hecho de que el proceso cambie de procesador no afecta a esta medida de tiempos, ya que el cálculo del número de ciclos que el proceso ha estado en ejecución se realiza cada vez que el proceso abandona un procesador (y se suma a lo acumulado). Esto significa que se compara el contador de ciclos cuando el proceso entra y cuando sale de la CPU, lo que asegura que nunca se compararán dos contadores de distintos procesadores.

En el apéndice A se describen las modificaciones realizadas sobre el kernel para poder disponer de esta llamada.

Se ha testado el buen funcionamiento de la llamada, y el error introducido en las situaciones más desfavorables (un gran número de procesos compartiendo el mismo procesador con llamadas a *sleep* aleatorias) no es superior a un 1 por 1000. Además, el mismo programa que se dormía muy frecuentemente y provocaba un mal funcionamiento de *sleep*, no consigue engañar a *my_cpu_time*.

Finalmente, solo queda establecer cual es el error introducido en la medida por el propio hecho de medir. Como se indica en la especificación de la instrucción *rdtsc*, ésta vacía el *pipeline* del procesador para tomar adecuadamente el número de ciclos. Ante la duda sobre si esta particularidad puede afectar a la toma del tiempo, se ha diseñado un programa que realiza una cierta cantidad de cálculo. Intercalado con este calculo, se han introducido llamadas a *my_cpu_time*. El objetivo es estudiar el coste del vaciado del *pipeline*. Se han introducido operaciones de cálculo en el programa para que el procesador pueda solapar el tiempo que cuesta la llamada al sistema (gracias a las técnicas que el procesador incorpora de ejecución fuera de orden y ejecución especulativa). Lo que no se podrá disimular es el vaciado del *pipeline* si realmente tiene un impacto.

Después de realizadas las pruebas, los resultados indican que una ejecución incluyendo las llamadas a *my_cpu_time* y una sin incluirlas, tardan lo mismo a efectos prácticos. De aquí se concluye que la propia instrucción *rdtsc* no altera la toma de tiempos.

Falta averiguar si el hecho de hacer una llamada al sistema para calcular el tiempo no afecta en el propio cálculo. Para analizar esto, se ha implementado un programa que imita las condiciones en las que *my_cpu_time* se va a llamar en la biblioteca MPI (sin las operaciones de cálculo de la prueba anterior que se solapaban con la llamada al sistema). En este nuevo caso, se ha establecido que una llamada a *my_cpu_time* consume 233 ciclos de procesador (en el contexto en que se va a utilizar). Este tiempo se descontará de la toma de tiempos cuando se emplee en la biblioteca MPI.

3.2. La biblioteca MPI

MPI es solo una interfaz. Para ejecutar un programa hay que emplear una implementación concreta, como puede ser *LAM* o *MPICH*. En este caso hemos empleado la librería MPICH 1.1 y es ésta sobre la que se han llevado a cabo las modificaciones necesarias para que la ejecución de la aplicación produzca el archivo de trazas deseado.

La propia librería ya dispone de algunas herramientas de log que listan los eventos de comunicación de la aplicación, pero a alto nivel, es decir, si la aplicación llama a una operación de broadcasting, se anotará un evento de broadcasting. Sin embargo, nosotros queremos hilar más fino y capturar los eventos a bajo nivel, es decir, a nivel de socket (la implementación de MPI sobre la que ejecutaremos las aplicaciones utiliza sockets para comunicación entre procesos). Si una llamada a broadcast se descompone en un protocolo específico de sincronización o cualquier otro desglose, queremos registrar estos eventos tal y como se producen en la realidad.

Otro motivo por el que no utilizamos las herramientas de monitorización ya disponibles es porque queremos tener un control total sobre las llamadas que se hacen en la libería MPI, ya que hasta el tiempo que cuestan las llamadas a estas herramientas de monitorización es algo que debemos tener en cuenta. Con esto, lo más fiable es elaborarlas nosotros mismos de forma que podamos saber la repercusión exacta que tienen sobre las medidas que realizan.

Después de estudiar la configuración de la biblioteca, se ha encontrado que todas las operaciones de alto nivel acaban por converger en el archivo *mpid/ch_p4/p4-1.4/lib/p4_sock_util.c*. Por esto, es aquí en donde se han realizado la mayoría de las modificaciones. Justo antes de la llamada al socket, vamos a capturar los mismos parámetros con que se llama a *read* o a *write*. Como ya se ha dicho, se anotará qué tipo de llamada es (recepción o emisión del mensaje), qué proceso es el invocante, a qué proceso o procesos se les envía (o desde los que se recibe), y el tamaño de la información intercambiada.

Para el caso en que se quiera simular la red rápida, la toma de tiempos no se realizará justo antes de las llamadas al socket. En esta ocasión, se supone que no será necesario navegar por la biblioteca MPI, ya que se tendrá soporte en forma de instrucciones ensamblador. Por este motivo, para simular la red rápida se toman los tiempos en cuanto se realiza la llamada a la operación MPI de alto nivel.

Además, a cada evento se le asocia una marca temporal. Esta marca temporal no consiste en el tiempo de pared, sino en la cantidad de tiempo que el proceso en cuestión ha estado computando desde la finalización del último evento (o desde el inicio del programa si se trata del primer evento). De esta forma, los tiempos de red que se calcularán mediante

simulación podrán ser intercalados fácilmente entre los tiempos de cómputo.

Vamos a ver un ejemplo de traza:

```
  Time  Operation Source Destination    Size
-----------------------------------------------
 32005          s      4           8     200
139167          s      4           9     200
   307          r      4           8    4900
   380          r      4           9    4900
```

Aquí, transcurridos 32005 ciclos desde el inicio del programa, el nodo 4 ha iniciado una operación de envío al nodo 8 de 200 bytes. Después de 139167 ciclos, realiza otra operación de envío pero esta vez al nodo 0. A continuación, cuando han transcurrido 307 ciclos desde el segundo envío, el nodo 4 inicia una operación de recepción de 4900 bytes con respecto al nodo 8. Finalmente, 380 ciclos después de completarse la recepción anterior, se vuelve a realizar una recepción similar.

Como se ve, son las marcas temporales asociadas a cada evento lo que confiere a las trazas un carácter realista frente a otro tipo de trazas, digamos, sintéticas.

Cada evento se almacena en formato binario (todos los eventos ocupan el mismo número de bytes) y se guarda en un búfer de 1 megabyte en memoria. Cuando este búfer se llena es volcado al disco duro, pretendiendo que la captura de eventos incida en la menor medida posible en la ejecución normal de la aplicación. Como ya se ha dicho, en las tomas de tiempo se descuentan los 233 ciclos que se ha calculado que cuesta la llamada al sistema.

Un aspecto que cabe remarcar es que de las llamadas al socket solo se capturan las que realizan una escritura o lectura, no las que comprueban si existe información a leer o si es posible escribir en el socket. Estas últimas llamadas se emplean para operaciones MPI del tipo *MPI_Irecv* y *MPI_Isend*, es decir, llamadas no bloqueantes. Estas llamadas intentan escribir o leer si es posible, retornando inmediatamente en cualquier caso. La metodología de simulación empleada asume que estas llamadas no están presentes en las aplicaciones y así es en las que aquí están estudiadas.

3.3. El simulador de red

El último elemento es el simulador de red, que calcula el tiempo que los mensajes tardarían en llegar desde el origen al destino atravesando una red con las características que se le especifican.

Se ha tomado el simulador de redes *wormhole* usado en el Grupo de Arquitecturas Paralelas y se ha traducido de Modula a lenguaje C para facilitar las posteriores modificaciones.

El simulador está diseñado para testar la red a partir de tráfico sintético, es decir, que permite especificar cuánto tráfico se requiere por unidad de tiempo, con qué media de distancia y otros parámetros, y el simulador genera un patrón de tráfico acorde a lo requerido. Sin embargo, para satisfacer nuestros intereses se necesita que el simulador acepte los ficheros de trazas y simule el paso por la red de los mensajes que las aplicaciones han generado. Por el mencionado motivo, se han introducido cambios para que el simulador lea los archivos y maneje el avance de los procesos *off-line* .

Dos premisas se han seguido para modelar este sistema:

- Cuando un proceso llega a un evento de recepción, no se puede avanzar en su traza hasta que el mensaje que está esperando llegue.

- Cuando un proceso llega a un evento de envío, puede seguir con su ejecución. Esto significa que el envío del mensaje a través de la red se hace en paralelo al avance del proceso emisor.

De esta forma, sumando el tiempo de cómputo (calculado en la etapa de ejecución de la aplicación) más el tiempo de espera (calculado con cada recepción de mensaje) se obtiene el tiempo total de cada proceso. Como todos los tiempos están expresados en ciclos de reloj, si queremos calcular el tiempo en segundos deberemos dividir el resultado por la frecuencia a la que operen los procesadores (aunque esto no tiene demasiada importancia).

Todo este manejo de tiempos de procesos ha sido añadido al simulador y de esta forma, también nos produce el tiempo total desglosado en tres tipos:

- Tiempo total, que es el tiempo transcurrido desde que el proceso se inició hasta que terminó.

- Tiempo de espera, que es el tiempo total menos el tiempo que el proceso ha estado dentro de la CPU, es decir, el tiempo que el proceso ha estado esperando.

- Tiempo de red, que está incluido en el tiempo de espera, y que es el tiempo que un proceso ha estado esperando y a la vez el mensaje estaba viajando por la red.

Este desglose nos permite saber la posible mejora de la aplicación, es decir, si existe mucho tiempo de espera es posible que modificando la forma en que se comunican los procesos este tiempo se pueda disminuir y por consiguiente acelerar la aplicación.

3.4. Parámetros de configuración de la red

A continuación se listan los parámetros con que se han configurado las dos redes en que se va a basar nuestro estudio. La red rápida se ha intentado asemejar a una red on-chip, mientras que para modelar la red lenta nos hemos fijado en la red de interconexión *Myrinet* [4].

Además, en las simulaciones con red lenta, en los tiempos de cómputo se han contemplado las penalizaciones constantes por envío mencionadas (en la toma de tiempos para las trazas destinadas a simulación en red lenta, se ha contado el tiempo de biblioteca MPI). En cambio, en red rápida, el tiempo de cómputo solo comprende hasta el momento en que se invoca a la interfaz MPI.

	Red rápida	*Red lenta*
Tamaño de palabra	8 bytes	8 bytes
Tiempo de transmisión	1 ciclo	15 ciclos
Tiempo de cruce	1 ciclo	15 ciclos
Tiempo de encaminamiento	1 ciclo	345 ciclos
Tiempo de propagación	2 ciclos	120 ciclos
Tamaño de las colas de entrada	5 palabras	80 palabras
Tamaño de las colas de salida	5 palabras	80 palabras
Número de canales de acceso a memoria	1 canal	1 canal
Topología	Malla 2D	Malla 2D
Encaminamiento	Determinista	Determinista
Número de canales virtuales	3 canales	3 canales

Cuadro 3.1. Características de las redes simuladas

Capítulo 4

Estudio de aplicaciones paralelas características

4.1. La multiplicación matricial

Después de ver los paradigmas susceptibles de ser mejorados con el empleo de una granularidad fina, vamos a adentrarnos en el mundo práctico, es decir, la encarnación de estos modelos en aplicaciones reales. Además, servirá para descubrir conceptos, pros y contras, que resultan difíciles de percibir desde el punto de vista teórico.

4.1.1. Presentación del algoritmo

Como primer ejemplo vamos a estudiar el algoritmo de multiplicación matricial paralelo, en su versión más simple. Aunque existen otras versiones mucho más sofisticadas y eficientes, nos quedaremos con ésta porque un algoritmo demasiado complejo puede distraer la atención de nuestro objetivo: reprogramar una aplicación para disminuir el grano de sus comunciaciones. Además, a medida que aumenta la complejidad del programa, modificarlo para que el tamaño de los mensajes sea parametrizable resulta cada vez más costoso.

De este modo, como primer paso vamos a analizar el proceso de multiplicación paralelo, programado con granularidad gruesa, es decir, encapsulando en un mensaje tanta información como sea posible.

Generalizando, el algoritmo se puede dividir en tres fases claramente diferenciadas:

- *Reparto de los datos*: En esta fase, el proceso 0 reparte las matrices a multiplicar de forma adecuada entre todos los procesos de la ejecución.

- *Cómputo de la multiplicación*: Ahora, cada proceso opera con los datos suministrados para formar su submatriz de resultados.
- *Recolección de resultados*: Finalmente, todos los procesos envían su submatriz al proceso 0 para que éste construya la matriz resultante.

Estas tres fases del algoritmo son a la vez tres fases de ejecución, es decir, que cada una consume su propio tiempo independiente. En otras palabras, el tiempo total de ejecución se reparte entre estas tres fases, con lo que si se quiere acelerar la ejecución habrá que recortar tiempo de alguna de ellas. El tiempo de cómputo no es reducible ya que aquí nos centramos en la red, no en los procesadores (fíjese que no se puede reducir el tiempo de cómputo, aunque en la siguiente sección veremos que sí se puede aumentar, desgraciadamente). De la fase de reparto y de recolección, que conforman el tiempo de espera, es de donde se podrá recortar tiempo o solapar fases.

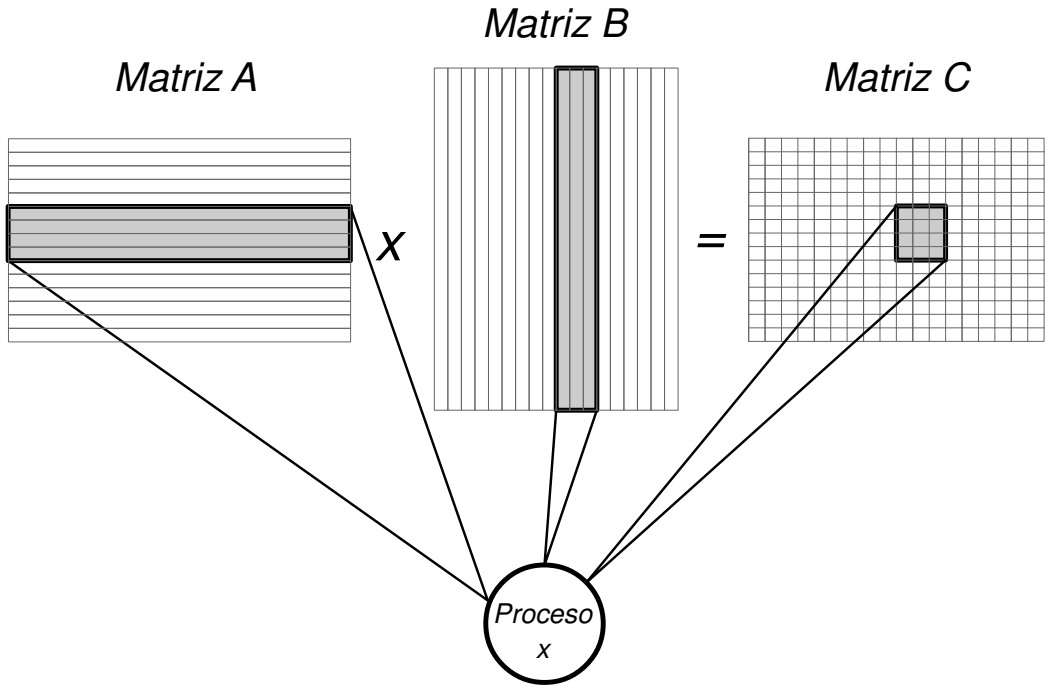


Figura 4.1. Esquema de funcionamiento de la multiplicación matricial

Es importante que veamos, al menos brevemente, el mecanismo de reparto de datos. Cada proceso se va a encargar de computar un sector de la matriz de resultados (ver figura 4.1). Para ello necesitará un grupo de filas y de columnas de las matrices a multiplicar. El proceso 0 se encargará de repartir estas filas y columnas o delegará en otros procesos para optimizar las operaciones de broadcast. Dependiendo de la granularidad elegida, este grupo de filas y columnas se repartirán en subgrupos más o menos pequeños.

La ventaja de repartir los datos iniciales en mensajes cortos radica en que los procesos pueden empezar a computar antes. Si los datos iniciales se encapsulan en un solo mensaje, hasta que no se reciba este mensaje completamente no se puede inciar la multiplicación. En cambio, si se reparten filas y columnas individuales, con la primera fila y columna ya se puede computar una celda de la matriz de resultados y tal vez, mientras se ha computado esta operación, la siguiente fila o columna ya ha sido recibida. Como se puede ver, la clave de la aceleración es el solapamiento de cómputo con recepción.

Imaginemos que los datos iniciales a repartir a cada proceso son de 1 MB de tamaño y que solo disponemos de un canal de memoria. Esto significa que el último proceso recibirá su mensaje muy tarde, tanto como lo que cueste de enviar 1 mensaje multiplicado por el número de procesos. Como vemos, esto representa un problema al que sin duda hay que prestar atención. Si fragmentamos el mensaje a nivel de filas, la primera fila le llega al último proceso casi de inmediato, ya que enviar un mensaje de una fila no cuesta prácticamente nada comparado con el envío de 1 MB (obviamente, en red on-chip).

Con la etapa de recolección de resultados ocurre algo parecido. Si las submatrices de resultados son muy grandes, enviar al receptor todos los resultados al mismo tiempo puede ocasionar una congestión en la red. Si conseguimos espaciar en el tiempo estos envíos de forma que cuando llegue el momento límite en que se requieran los datos estos ya se hayan recibido, se podrá acelerar la aplicación. Para lograrlo, cada vez que se obtenga un cierto número de resultados (puede ser una sola celda en la granularidad más fina) se enviarán al receptor. De esta forma, estos envíos se solapan con el cómputo y no provocan que el receptor se convierta en un cuello de botella. Si conseguimos plasmar todo esto en el algoritmo, estaremos usando granularidad fina.

4.1.2. La importancia del sincronismo

El tipo de llamadas MPI empleadas son, como se puede adivinar, *MPI_Send* y *MPI_Recv*. MPI ofrece versiones no bloqueantes para estas dos llamadas de forma que no se bloquean cuando son invocadas, sino que el sistema operativo se encarga de completarlas y el programador se encarga de asegurarse que se han realizado satisfactoriamente. A pesar de que a primera vista parecen las más indicadas en cualquier caso, esto no es así cuando la granularidad aumenta el número de mensajes. Cuando se tienen varias llamadas no bloqueantes pendientes de completarse, esto provoca una gran sobrecarga al sistema que ralentiza notablemente la ejecución. Con esto, si enviamos muchos mensajes de poco tamaño, antes de que se complete el primer envío o recepción ya se han invocado un gran número de estas llamadas, con lo que el sistema consume muchos recursos por tener tantos envíos

o recepciones sin completar. Por este motivo se han empleado llamadas bloqueantes. De hecho, una primera implementación de la multiplicación matricial utilizando llamadas no bloqueantes provocaba que la versión de granularidad fina fuese tremendamente más lenta (casi dos órdenes de magnitud) que la versión de granularidad gruesa. Al cambiar a llamadas bloqueantes, esta diferencia se redujo en gran medida.

A raíz de esto cobra importancia el tema del sincronismo entre procesos, lo que nos servirá para introducir este nuevo concepto realmente importante relacionado con la granularidad fina y las operaciones bloqueantes.

Para simplificar las cosas vamos a suponer que los procesos reciben algunos datos de las matrices a multiplicar, hacen las operaciones pertinentes y envían los resultados al proceso 0, repitiéndose este ciclo multitud de veces. Por su parte, el proceso 0 envía datos a los procesos, también computa su parte correspondiente y espera resultados de cada proceso. La siguiente figura ilustra esta situación.

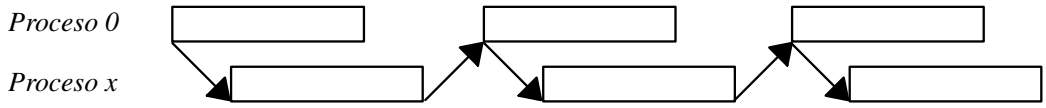


Figura 4.2. Ejemplo de mala sincronización

Como se ve, aunque la figura no está dibujada a escala, en cada iteración se produce un tiempo de espera, tanto en el proceso 0 como en el proceso x. Además, se debe tener en cuenta que este mecanismo de sincronización se produce con todos los procesos, con lo que el tiempo de espera crece con el número de procesos.

Una de las mejoras en el algoritmo que más ha acelerado la ejecución es la solución de este problema. Para solventarlo, basta con proporcionar a los procesos un margen de actuación, es decir, que el proceso 0 no espere a recibir resultados inmediatamente sino que posponga la recolección unas iteraciones. De esta forma se eliminarán los tiempos de espera tal y como se muestra en la siguiente figura:

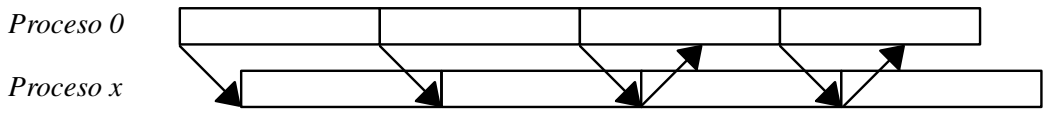


Figura 4.3. Ejemplo de buena sincronización

En esta nueva situación, las esperas en cada iteración del bucle se han reducido a cero. Sirva esto para demostrar que, en un algoritmo de granularidad fina, es necesario cuidar todos los detalles respecto a las comunicaciones, ya que una espera minúscula en cada

iteración puede provocar que el tiempo de ejecución total se dilate notablemente. Esto es, sin duda, una desventaja unida al uso de granularidad fina, ya que requiere que los programadores consuman más tiempo atendiendo a estas particularidades. En el siguiente apartado comprobaremos empíricamente la importancia de un buen sincronismo.

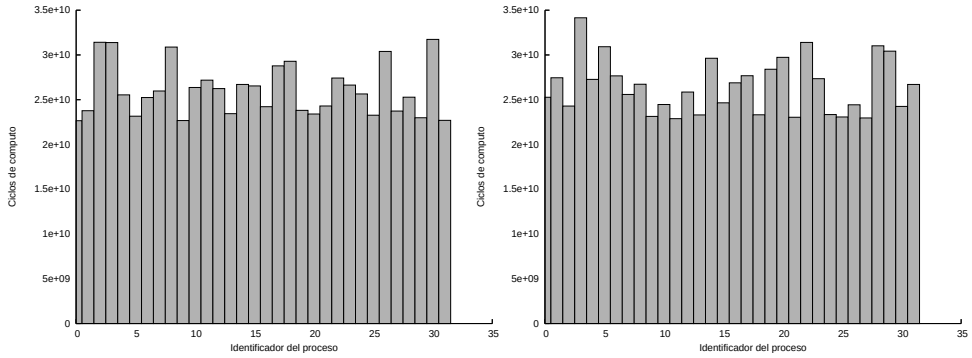
4.1.3. Experimentos

Siguiendo la metodología descrita en el capítulo 3, primeramente hemos ejecutado la aplicación para extraer los tiempos de cómputo. Más tarde, simularemos los tiempos de red para obtener el tiempo total de ejecución. Como comprobación rutinaria, una vez obtenidos los tiempos de cómputo, se llevan a cabo una serie de comprobaciones sobre los ficheros de trazas. Entre otras cosas, comparamos los tiempos de cómputo de cada proceso. En el algoritmo de multiplicación de matrices, al menos todos los procesos exceptuando al proceso 0 tienen en principio un comportamiento idéntico, es decir, sus tiempos de cómputo deben ser prácticamente iguales.

Sin embargo, se ha comprobado que, de forma aparentemente aleatoria, los tiempos de cómputo de los diferentes procesos son variables. Es más, esta variabilidad ronda el 10 % respecto al tiempo total de ejecución. Esto significa que los tiempos de espera generados por esta variabilidad resultan ser bastante grandes, lo que implica una ralentización de la aplicación que provoca que se diluyan las mejoras ocasionadas por el empleo de una granularidad fina. Esto se acentúa cuando se comprueba que el tiempo que tarda en enviarse los mensajes de granularidad gruesa es despreciable en comparación con el tiempo de cómputo. Por consiguiente, aunque se emplee granularidad fina, en esta aplicación en concreto no se experimentan cambios significativos, ya que el tiempo total de ejecución eclipsa la mejora ocasionada, más aún cuando existe la variabilidad mencionada.

En la figura 4.4 se muestran los tiempos de cómputo de dos ejecuciones, a modo de ejemplo de la variabilidad comentada. Se han lanzado 32 procesos para que multipliquen dos matrices de 4000 por 4000 elementos. La plataforma empleada ha sido un computador de memoria compartida con 4 procesadores Dual-Core AMD Opteron, lo que suma un total de 8 nucleos a 2010.246 MHz con una memoria cache de 1 MB y un total de memoria principal de 8 GB. Como estamos empleando la llamada al sistema para medir tiempos comentada anteriormente, en principio cabría esperar que ningún factor pudiese alterar los tiempos de cálculo. No obstante, como se puede observar, la variabilidad es clara. A pesar de esto, de este ejemplo se han obtenido algunas conclusiones.

La primera de ellas está relacionada con la variabildiad. La causa de esta variabilidad no se ha logrado establecer: la cantidad de datos con la que se opera es idéntica para cada pro-



(a) Ejecución 1 (b) Ejecución 2

Figura 4.4. Ejemplo de variabilidad en los tiempos de cómputo entre los distintos procesos de la ejecución

ceso e incluso se han realizado pruebas repartiendo los mismos datos (los mismos valores) a cada proceso y los tiempos de cómputo siguen variando. Esta disparidad en los tiempos puede ser causada por la distribución no ecuánime de procesos entre procesadores, es decir, que en algunos procesadores existan más procesos compitiendo que en otros. Esto produce que los procesos que compartan CPU con un mayor número de procesos experimenten un mayor número de cambios de contexto. El tiempo de cambio de contexto está considerado como tiempo de cómputo del proceso, con lo que esto podría causar la variabilidad.

Otro factor a tener en cuenta son los fallos de cache, que afectan de forma poco predecible al tiempo de cómputo de cada proceso. Por alguna razón, unos procesos podría tener más fallos que otros, lo que generaría esta variabilidad.

No se ha prestado más atención a esta variabilidad ya que para el resto de experimentos no se ha producido. El motivo de esto es que el resto de aplicaciones intercalan un mayor número de operaciones de red, es decir, operaciones que duermen al proceso, con lo que los procesos no están sometidos al planificador de la CPU (y el número de procesos activos al mismo tiempo es más reducido que en el caso de la multiplicación matricial). Pero esta variabilidad es un concepto interesante que sin duda se produce en el mundo real. Por ejemplo, si cada proceso de una ejecución ocupa una CPU diferente con un nivel de ocupación distinto o incluso con una frecuencia de operación distinta, la variabilidad se hace patente.

Y es aquí donde entra en juego la granularidad fina ([2]). La posibilidad de enviar mensajes pequeños, o lo que es lo mismo, mayor cantidad de mensajes, ofrece una posibilidad para contrarrestar los retardos ocasionados por la variabilidad en los tiempos de cómputo. Mediante una granularidad fina, la carga computacional se puede repartir dinámicamente

entre los procesos, para que el proceso que se ejecute más rápido realice más cálculos y todos terminen al mismo tiempo. Este es un campo bastante complejo ya que administrar la carga en tiempo real es una tarea difícil, pero muy interesante ya que un solo proceso que por la razón que sea compute a mitad velocidad que el resto, provoca que el tiempo de ejecución total se multiplique por dos.

Debido a esta variabilidad no se ha podido llevar a cabo ninguna simulación concluyente. No obstante, para no dejar esta aplicación sin su estudio empírico, vamos a tratar de sacar alguna conclusión que se pueda medir en términos de tiempo de ejecución. Para esto, se han generado unos archivos de trazas con tiempos de computo sintéticos, es decir, que no provienen directamente de una ejecución sino que se han calculado a partir de los tiempos obtenidos en la ejecución pero eliminando la variabilidad. En concreto, hemos imitado la media de tiempo de cómputo de una ejecución de 16 procesos al multiplicar una matriz de 10000 por 4000 elementos por otra de 4000 por 10000 elementos. En definitiva, el tiempo de cómputo por proceso es de 200000 millones ciclos (entre los que se intercalarán los envíos y recepciones pertinentes).

A partir de aquí se han simulado los tiempos de red obteniendo los tiempos de ejecución totales. Los resultados confirman la primera de las ideas fundamentales: una granularidad fina acelera la aplicación en una red rápida.

De esta forma, después de simular los tiempos en la red rápida, se han obtenido dos tiempos de ejecución: granularidad gruesa y fina. El objetivo principal era conseguir reducir los tiempos de espera de los procesos y esto se ha conseguido. El tiempo de espera del algoritmo de granularidad fina es tan solo un 20 % del tiempo de espera del algoritmo de granularidad gruesa. En la figura 4.5 se ilustran los resultados, tanto de granularidad gruesa como fina, desglosando esta última en buena y mala sincronización, tal y como se ha descrito anteriormente. Concretamente, para lograr el buen sincronismo, se ha retrasado 100 iteraciones la recepción de datos por parte del proceso 0.

Atendiendo a los tiempos de espera, se demuestra que un buen sincronismo es esencial. Si se presta atención a este aspecto y se logran sincronizar adecuadamente todos los procesos, una granularidad fina puede reducir en gran medida el tiempo de espera con respecto a una granularidad gruesa. Sin embargo, los tiempos de cómputo diluyen estos tiempos de espera, con lo que la mejora obtenida resulta despreciable. Esto es así debido a la gran cantidad de cómputo de la multiplicación matricial, como ya se ha mencionado.

En los siguientes capítulos abordaremos otros ejemplos que sí que darán más juego al modular el grano de las comunicaciones.

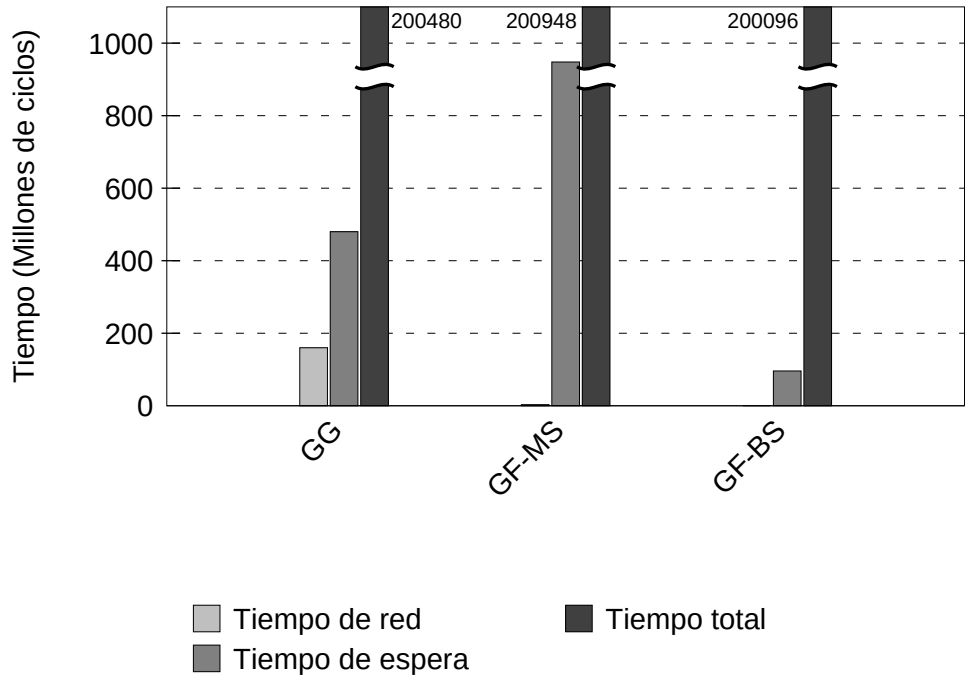


Figura 4.5. Multiplicación matricial: GG (granularidad gruesa), GF-MS (granularidad fina, mala sincronización) y GF-BS (granularidad fina, buena sincronización)

4.2. La búsqueda jerárquica

Del ejemplo anterior hemos concluído en un par de puntos claros:

- El tiempo de espera de un algortimo puede reducirse al pasar a usar granularidad fina.
- Para que esta mejora se pueda percibir, el tiempo de espera debe ser una parte notable del tiempo total de ejecución.

4.2.1. Presentación del algoritmo

Las especificaciones que acabamos de ver describen un tipo de algoritmo que repite al esquema de la multiplicación matricial, esto es, el envío inicial se produce una serie de veces o lo que es lo mismo, se sigue un patrón de reparto de datos, computación, reparto de datos, computación, etc. Además, si el tiempo de cómputo fuese menor en comparación al tiempo de reparto de la información, mayor aceleración se podrá conseguir.

Puede surgir como primera idea, el estudio de la repetición de la multiplicación matricial dentro de un bucle, es decir, lanzar varias operaciones de multiplicación en serie. Sin

embargo esto no funcionará porque el envío de los datos de las matrices a multiplicar en segundo lugar, puede solaparse con el cómputo de la primera multiplicación (aun usando granularidad gruesa). Este ejemplo ilustra que el algoritmo susceptible de ser acelerado con granularidad fina debe formar una especie de barrera sincronizadora justo después de cada etapa de cómputo, para que no se puedan solapar los envíos.

Esto se traduce a que los datos a enviar en cada iteración sean dependientes de los resultados de la computación en la iteración anterior. De esta manera, el envío no puede producirse hasta que todos los procesos terminen con sus cálculos.

Una aplicación representativa de este tipo de aplicaciones es la búsqueda jerárquica, que sigue el patrón descrito. El algoritmo comienza analizando un vector. Cada celda del vector tiene un puntero a otro vector y además también tiene un campo que en nuestro caso guarda un número. El algoritmo repasa este primer vector para encontrar el número máximo. Esta celda que contiene el número máximo apunta a otro vector sobre el que se repite el proceso, que acabará indicándonos un tercer vector, etc.

La paralelización de este algoritmo se basa en paralelizar la búsqueda del número máximo en cada vector. Procediendo de esta forma, se reparte el vector entre los procesos. A continuación, cada proceso busca el máximo en su subvector. Finalmente, cada proceso envía el máximo (y su posición) al proceso 0. Después de esta iteración, el proceso 0 sabe qué vector del siguiente nivel de la jerarquía debe repartirse, procediendo de igual forma.

Lo que tenemos aquí es la repetición del patrón de la multiplicación matricial un cierto número de veces. Esto, unido a que el tiempo de cómputo en proporción a los datos es mucho menor (en la búsqueda jerárquica solo tenemos que repasar el subvector operando mediante comparaciones), por todo lo dicho anteriormente, es de esperar que al pasar la implementación a granularidad fina el tiempo necesario para ejecutarla se reduzca.

El proceso de cambio de granularidad es el mismo que el usado en la multiplicación matricial: variamos el tamaño de los mensajes a la hora de repartir el vector. El único cambio es el proceso de recolección de resultados, que en la búsqueda jerárquica no se puede cambiar de granularidad ya que se envía tan solo un par de enteros.

4.2.2. Experimentos

Como paso inicial, hemos realizado simulaciones sobre una red rápida, ya que lo primero es cerciorarse de que al bajar la granularidad, al menos en red rápida, vamos a acelerar la aplicación.

De los resultados de estas simulaciones no pretendemos obtener conclusiones absolutas, sino identificar nuevos conceptos que deberán tener presentes otras implementaciones de

algoritmos paralelos. Por esta razón, primero presentaremos los resultados de algunas de las pruebas realizadas para, seguidamente, extrapolar los resultados y obtener un estudio generalista sobre la granularidad de las comunicaciones.

Como primer resultado, tenemos la reducción del tiempo de espera. La ejecución se ha realizado bajo estas condiciones: cada vector sobre el que buscar el máximo tiene un tamaño de 1600000 elementos y la profundidad de búsqueda ha sido de 10000 niveles (es decir, se han buscado 10000 máximos), cómputo que se ha distribuído entre 16 procesos. De momento, vamos a simular la red rápida para ver cual es la aceleración obtenida mediante le cambio de granularidad, que se ha establecido en 100000 bytes por mensaje para granularidad gruesa y en 100 bytes por mensaje para granularidad fina. Los resultados de la simulación se presentan en la tabla 4.1 (como siempre, se toman los tiempos del proceso director, es decir, el proceso 0, y se expresan en ciclos de reloj).

	Tiempo total	Tiempo de espera	Tiempo de cómputo
Red lenta	22880221857 $\cong$ 23M	6306272404 $\cong$ 6M	16573949453 $\cong$ 17M
Red rápida	85709843156 $\cong$ 86M	25601 $\cong$ 0M	85709817555 $\cong$ 86M

Cuadro 4.1. Resultados del experimento en búsqueda jerárquica

Si nos fijamos primeramente en los resultados en red lenta, observamos que el tiempo de espera constituye un buen porcentaje del tiempo total, en concreto, un 27 %. Ahora, viendo el tiempo de espera cuando hemos reducido el tamaño de los mensajes, vemos que es de tan solo 25601 ciclos, que en comparación es prácticamente 0. Con una granularidad fina el tiempo de espera se ha eliminado de forma práctica. ¿Significa esto que la aplicación se ha acelerado al reducir el grano de sus comunicaciones?

La respuesta a la pregunta es que no, tal y como se ve en los tiempos totales, hecho que nos ayudará a introducir un nuevo concepto muy interesante. Si el tiempo de espera se reduce pero el tiempo total no se reduce (incluso aumenta), es, sin duda, porque el tiempo de cómputo no es el mismo. Este hecho no tiene nada que ver con la variabilidad anteriormente comentada. En esta ocasión, a medida que aumenta la granularidad, los procesos necesitan un mayor tiempo de cómputo. Por esta razón, si el grano de las comunicaciones aumenta, el tiempo de espera se reduce, pero al mismo tiempo aumenta el tiempo de cómputo, con lo que el tiempo total no disminuye, sino que incluso se ve incrementado.

Este aumento del tiempo de cómputo está ocasionado por el manejo de los mensajes, con lo que en granularidad fina, esta sobrecarga es mucho mayor que en granularidad gruesa. Anteriormente hemos dicho que la red lenta inserta una latencia constante en los mensajes, mientras que la red rápida no. También hemos dicho que en la red rápida se traduce una

llamada a MPI en unas pocas instrucciones en ensamblador. Pero a pesar de esto, incluso en red rápida existe una penalización por mensaje.

Por ejemplo, a la hora de repartir el vector entre los procesos, necesitaremos crear un bucle para iterar por todos ellos. Además, para cada proceso haremos una llamada a MPI que por poco que cueste, hay que pasar una serie de parámetros que seguramente no se conocerán en tiempo de compilación ([5]). En adición, si se emplean grupos, habrá que descubrir qué procesos se encuentran en el grupo. Si el tamaño de los mensajes viene determinado por una estructura que debe analizarse en tiempo de ejecución, eso es un retardo más añadido. A pesar de que la red rápida optimice las operaciones MPI, siempre encontraremos una penalización aunque sea pequeña. Si el número de procesos es elevado y el número de mensajes es muy grande, es posible que no se compense esta sobrecarga con la aceleración debida al uso de granularidad fina.

En la implementación realizada de la búsqueda jerárquica, se ha dedicado esfuerzo a disminuir el impacto en tiempo de cómputo que supone una granularidad fina, buscando formas de manejar esta cantidad de envíos y recepciones de la mejor forma posible. Sin embargo, estos intentos han sido completamente infructuosos.

Sirva esto para indicar que es muy difícil conseguir una penalización cero a pesar de usar una red rápida con soporte de instrucciones ensamblador para MPI. Por esto, no es aconsejable implementar una aplicación con la mínima granularidad sin comprobar que la sobrecarga no contrarresta la optimización.

¿Es este un resultado completamente desalentador? No, en el próximo capítulo abordaremos otra aplicación cuyo margen de mejora es mayor que el ofrecido por las dos estudiadas hasta el momento.

4.3. Algoritmo de Merge

En la sección anterior no se consiguió una gran aceleración. Sin embargo, ha servido para ejemplificar una situación básica de posible mejora con granularidad fina. En este apartado vamos a ilustrar otra situación susceptible de ser acelerada. Básicamente es similar a la anterior, es decir, que consiste en partir un mensaje grande en varios pequeños. La diferencia radica en que el tiempo de espera a reducir ahora es mucho mayor que anteriormente, con lo que las posibilidades de mejora son mayores.

En la situación anterior, los procesos estaban esperando a que les llegase la información, y lo estaban haciendo desde que el primer bit se empieza a enviar. En esta ocasión, los procesos están también en estado de espera, pero lo están desde mucho antes de que el proceso emisor esté en disposición de enviar. Mientras el proceso emisor está computando

los datos a enviar, los procesos restantes ya están ociosos, con lo que tenemos más opciones de aceleración. Para entender todo esto y aclarar las dudas, vamos a pasar al ejemplo: el algoritmo de *merge* o algoritmo de fusión.

En pocas palabras, el algoritmo de *merge* toma dos vectores y produce un único vector fusionando de la manera adecuada los elementos de los vectores iniciales. De forma práctica, los dos vectores iniciales vienen ordenados y el algoritmo produce un vector también ordenando, lo que constituye el corazón del algoritmo de ordenación *merge*.

4.3.1. Presentación del algoritmo

El algoritmo que nos va a servir de ejemplo es un algoritmo de ordenación, pero claro está, ordenación paralela. Partiendo de un vector con números a ordenar, la manera a proceder es: repartir este vector entre los procesos, que cada proceso ordene su subvector y finalmente reunir los subvectores. Nosotros vamos a centrarnos en la última etapa, es decir, cuando los procesos ya tienen ordenados sus subvectores. Esta tercera fase consiste en fusionar (algoritmo de *merge*) los subvectores para obtener un vector completamente ordenado.

Una primera aproximación podría consistir en enviar todos los subvectores a un proceso que se encargase de fusionarlos. Sin embargo, este método no paralelo estaría desaprovechando recursos. Su coste computacional se puede describir de la siguiente forma:

$$O(nx)$$

donde n es el tamaño del vector y x es el número de procesos en la ejecución (supondremos que es potencia de dos). Esto es así porque el algoritmo debe obtener n números y para buscar cada uno de ellos debe comparar todos los mínimos de cada subvector.

La alternativa paralela a esto es distribuir esta fusión de subvectores. Para esto, en vez de fusionar en una sola vez los x subvectores, se van a fusionar pares de subvectores. La operación de *merge* global se realizará por etapas. Durante la primera etapa, los procesos impares fusionarán su subvector y el subvector vecino, obteniendo unos subvectores de tamaño mayor. Estos se fusionarán a la vez de dos en dos repitiéndose el proceso hasta llegar a la última etapa en donde se fusionarán los dos últimos subvectores (ver figura 4.6). La complejidad de esto es:

$$O(\tfrac{4n(x-1)}{x})$$

Si consideramos que el número de procesos es elevado, la expresión puede simplificarse así:

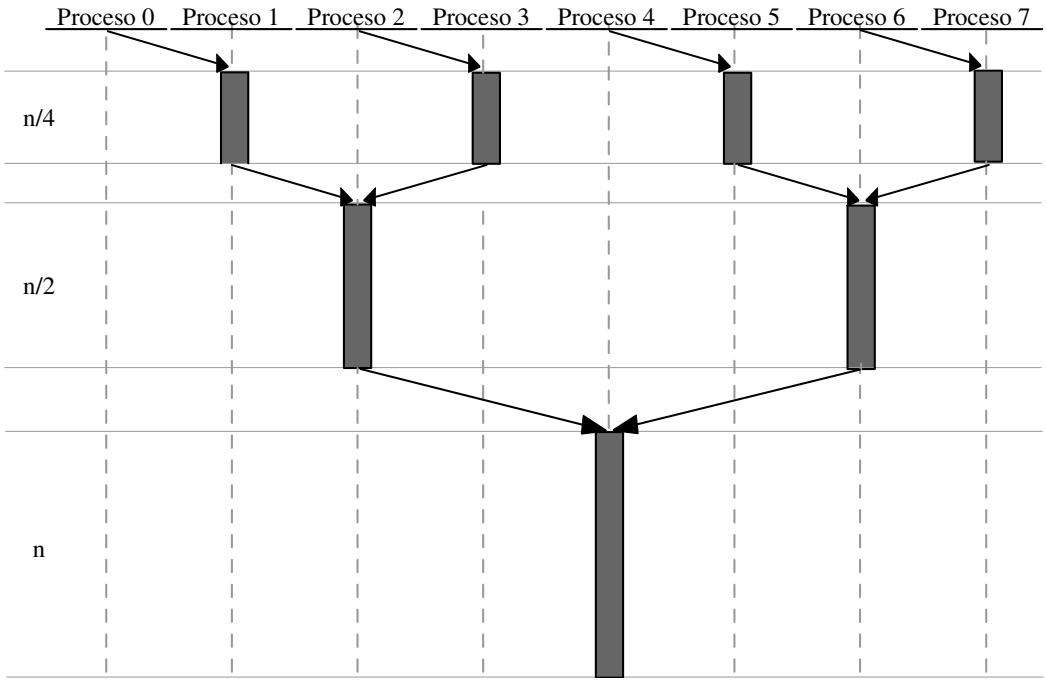


Figura 4.6. Esquema del algoritmo de merge

$$O(\tfrac{4nx}{x}) = O(4n)$$

A primera vista puede verse que esta nueva aproximación es mucho más rápida que la anterior. No olvidamos que habría que valorar adecuadamente las constantes a añadir a la expresión del coste computacional, aunque de momento no tienen mayor importancia porque vamos a ir un paso más adelante.

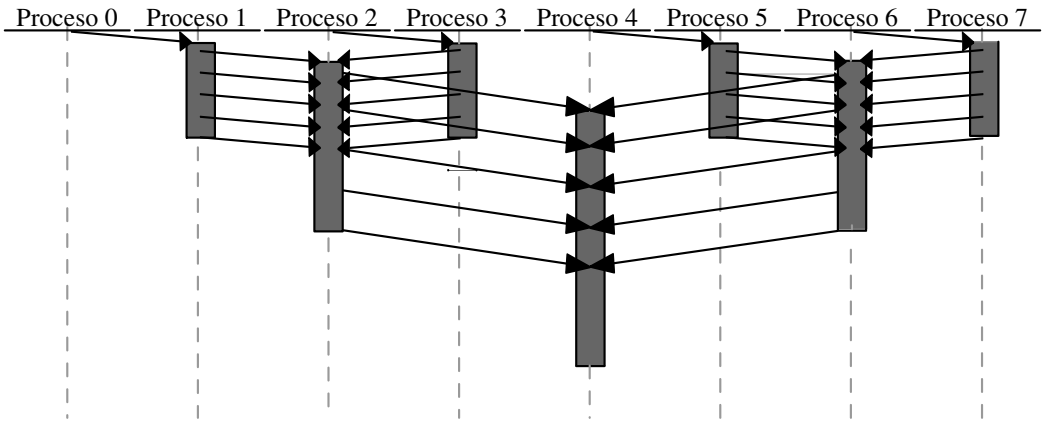


Figura 4.7. Esquema del algoritmo de merge con granularidad fina

Hasta ahora no hemos hablado de granularidad. La aplicación de la granularidad a este algoritmo se muestra en la figura 4.7. Como se ve, las diferentes etapas del algoritmo pa-

ralelo pueden superponerse. Esto nos lleva a reformular el coste del algoritmo teniendo en cuenta la granularidad:

$$O(2n + ti(2x - 2))$$

donde ti es el retardo en llegar el primero de los mensajes en que se han desglosado los mensajes grandes. Con una granularidad muy fina, ti es prácticamente 0, con lo que el coste total viene en orden de n. Lo que tenemos aquí es un algoritmo paralelo cuya granularidad en las comunicaciones puede ser modulada muy fácilmente con resultados directos.

El esqueleto básico del algoritmo es el siguiente: primero se recibe los primeros fragmentos de los correspondientes subvectores a fusionar. A continuación, sobre estos fragmentos empieza el procesado, generando parte del subvector resultante. En cuanto se obtenga una cantidad adecuada de resultados, ya se puede enviar al destino. En cuanto el proceso agote alguno de los dos subvectores, debe esperar a recibir otro fragmento.

4.3.2. Experimentos

Como en el capítulo anterior, vamos a realizar una serie de pruebas tanto en la red considerada como lenta como en la rápida. Como diferencia tenemos que la red lenta no se va a simular, sino que se empleará una red real, es decir, se tomarán tiempos de una ejecución real.

Esta vez se ha optado por no simularla porque de esta forma vamos a obtener una cota pesimista. Aunque parecería que esta ejecución real no sea conveniente y deba compararse con los resultados de una simulación, esta cota pesimista será fundamental a la hora de extraer resultados y cumplirá todos los requisitos para llegar a conclusiones, como se verá posteriormente.

De esta forma, la red rápida está parametrizada de la misma forma que la red del capítulo anterior. La red lenta es la siguiente: un clúster dedicado formado por 16 nodos, cada uno de ellos equipado con 2 AMD Opteron 1.4 GHz, memoria cache de 1 MB y 2 GB de memoria principal. Con respecto a la red empleada, se trata de Gigabit Ethernet también dedicada.

Para cada una de las dos redes, vamos a probar distintas granularidades. La gráfica que aparece a continuación refleja los experimentos realizados. Estos resultados pertenecen a ejecuciones de 8 procesos con un tamaño de vector total de 640000000 elementos. Debido a la naturaleza del algoritmo, no existe posibilidad de escalado, con lo que no tiene mayor interés presentar aquí los resultados de las pruebas de 16 y 32 procesos.

El eje de las ordenadas representa el tiempo total de ejecución medido en segundos. El eje de las abscisas representa la granularidad empleada, donde el 100 % significa que

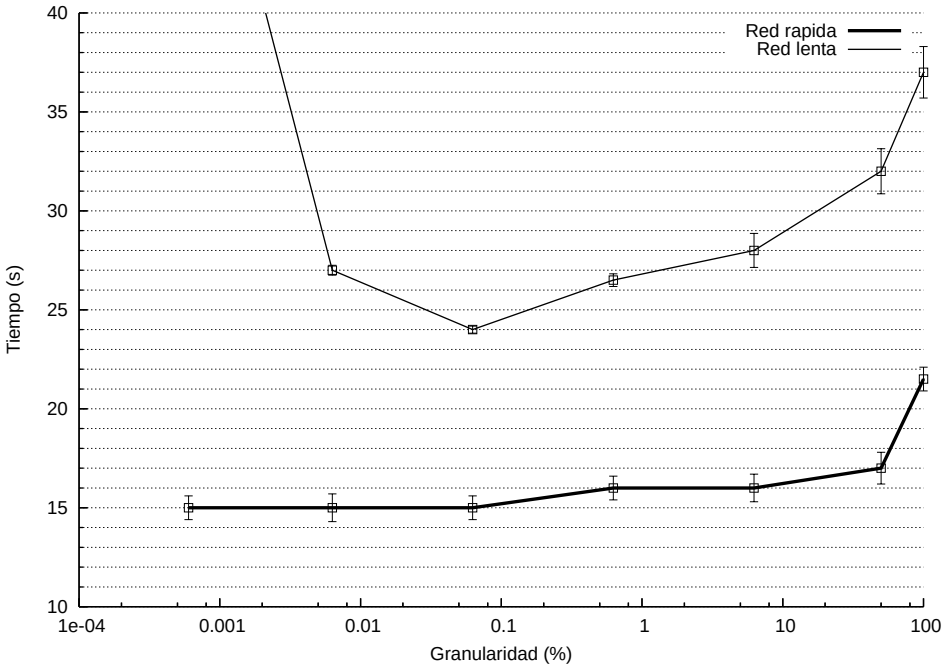


Figura 4.8. Aceleración de la aplicación según la granularidad empleada

los mensajes se han enviado con la mayor cantidad de información posible, es decir, el envío de los mensajes se ha retrasado hasta el límite. Además cabe observar que la escala está expresada logarítmicamente.

Lo primero que se puede observar en los resultados es que la ejecución en red rápida es más rápida que la ejecutada en red lenta. Esto tiene bastante lógica si prestamos atención al nombre con que se han designado las dos redes. Sin embargo, el factor de mayor influencia no es la red en este caso. Téngase en cuenta que la ejecución en red lenta es una ejecución real, esto significa que las comunicaciones se realizan a través de sockets TCP/IP, y no solo eso, sino que las comunicaciones se realizan a través de *ssh*, es decir, encriptadas. Esto y otros motivos provocan la diferencia.

Pero esta no es la cuestión del estudio. Estamos intentando demostrar que la granularidad óptima en una red lenta no es la óptima en una red rápida. Desde el punto de vista de clasificar la granularidad en fina y gruesa, observamos según la gráfica que la granularidad más gruesa es mucho mejor que la granularidad más fina, en la red lenta. El motivo de esto es que cada mensaje tiene asociada una penalización y a medida que aumenta el número de mensajes aumenta esta penalización global.

No obstante, hemos probado una serie de granularidades, lo que nos ofrece una visión más detallada. Si nos fijamos en la gráfica para red lenta, vemos que no existe una relación lineal entre la velocidad del programa y la granularidad empleada. Partiendo de granulari-

dad gruesa, según la disminuimos, empieza a reducir el tiempo necesario de la ejecución. Este descenso sigue hasta llegar a un punto mínimo, que marca la granularidad óptima que produce la ejecución más rápida. A partir de aquí, la penalización de los mensajes pesa más que la mejora de la granularidad fina y el tiempo se dispara.

Sin atender aún a la red rápida podemos deducir algunas conclusiones. La granularidad óptima en red lenta no es ni la más fina ni la más gruesa, sino un punto intermedio. Este hecho lleva a pensar que para cada tipo de red es necesario una ”puesta a punto”de la aplicación, para que adopte la granularidad más propicia. De hecho, es habitual que en los grandes centros de cómputo las aplicaciones sean particularizadas a las características del sistema donde se ejecutan.

Ahora, la hipótesis se confirmará si esta granularidad óptima para red lenta no es la mejor para red rápida. Sin embargo, volviéndo a la gráfica, a partir de la granularidad óptima en red lenta, la red rápida ya no acelera más la aplicación (al menos significativamente), y vemos que se estanca en los 15 segundos. Esto es así porque el tamaño del mensaje ya es muy pequeño, con lo que la aceleración conseguida se encuentra muy próxima a la máxima alcanzable.

Al menos en la aplicación aquí estudiada, no es necesario un cambio en la filosofía a la hora de mudar la aplicación a una red on-chip. Si la granularidad está adecuadamente sintonizada con las posibilidades de una red lenta, entonces con esa misma granularidad ya se está alcanzando la máxima mejora en una red rápida.

4.3.3. Análisis teórico

Estamos viendo que al refinar el planteamiento de la hipótesis de partida (ya no distinguimos la granularidad en gruesa o fina sino que adopta todo un rango de posibilidades) las mejoras que en principio cabría esperar parece que no son tales. Acabamos de ver un ejemplo que ilustra esta idea, y como complemento a esto vamos a realizar un análisis teórico de la situación.

Volvamos a la idea básica reflejada en la figura 2.2, patrón que forma la base del algoritmo de *merge*. Vamos a expresar la mejora en términos de tiempo ahorrado que se puede producir en función de la granularidad. Concretamente, esta mejora se traduce en la cantidad de tiempo que el proceso paciente deja de estar esperando para estar procesando, con respecto a la granularidad más gruesa posible. Dicho de otra manera, la mejora es el decremento en el tiempo necesario para completar la ejecución total.

Cabe remarcar que este análisis teórico no está restringido al algoritmo de *merge* aquí estudiado, sino que es extensible a cualquier aplicación que siga el patrón dibujado en la fi-

gura 2.2. En realidad, el desarrollo teórico está basado en dicha figura y su aplicación a un determinado algoritmo, como el de *merge*, dependerá de cuántas veces repita el citado patrón.

Si consideramos que *t* es el tiempo que el proceso paciente está esperando, que *n* es el número de mensajes en que se desglosa el mensaje grande y que *p* es la penalización constante por mensaje, tenemos que la mejora se expresa de la siguiente manera:

$$M(n) = t - \frac{t}{n} - n \cdot p$$

Para entender el significado práctico de esta expresión vamos a poner un ejemplo: si sustituyendo las variables por los valores correspondientes la expresión anterior produce por ejemplo 10, significa que con el número de mensajes indicado la aplicación finaliza 10 unidades de tiempo antes que con el envío de 1 solo mensaje.

Vemos que la mejora depende de los tres factores mencionados, donde *n* está reflejando la granularidad: a mayor granularidad menor tamaño de mensaje y mayor número de ellos.

Es fácil de deducir que el tiempo de ejecución necesario puede expresarse de esta forma:

$$T(n) = \frac{t}{n} + n \cdot p$$

suponiendo que el tiempo total de ejecución es igual a t. La gráfica 4.9 ilustra esta última ecuación para distintos valores de p, suponiendo que t es igual a 100 unidades de tiempo (eje de las abscisas está en escala logarítmica). Como se ve, a medida que p disminuye, el número óptimo de mensajes va en aumento.

Ahora buscaremos cual es la máxima mejora alcanzable. Si suponemos que *t* y *p* son invariables, solo podemos jugar con el número de mensajes. En el siguiente desarrollo obtenemos el máximo:

$$M(n) = t - \frac{t}{n} - n \cdot p \qquad M'(n) = \frac{t}{n^2} - p$$

$$\frac{t}{n^2} - p = 0 \rightarrow n = \sqrt{\frac{t}{p}}$$

Hemos obtenido el número óptimo de mensajes en que se debería desglosar un mensaje grande. Para un sistema dado con una penalización por mensaje concreta y una aplicación dada con un tiempo de espera determinado, existe siempre un número óptimo de mensajes.

Ahora, sustituyendo, obtendremos cual es la mejora máxima alcanzable:

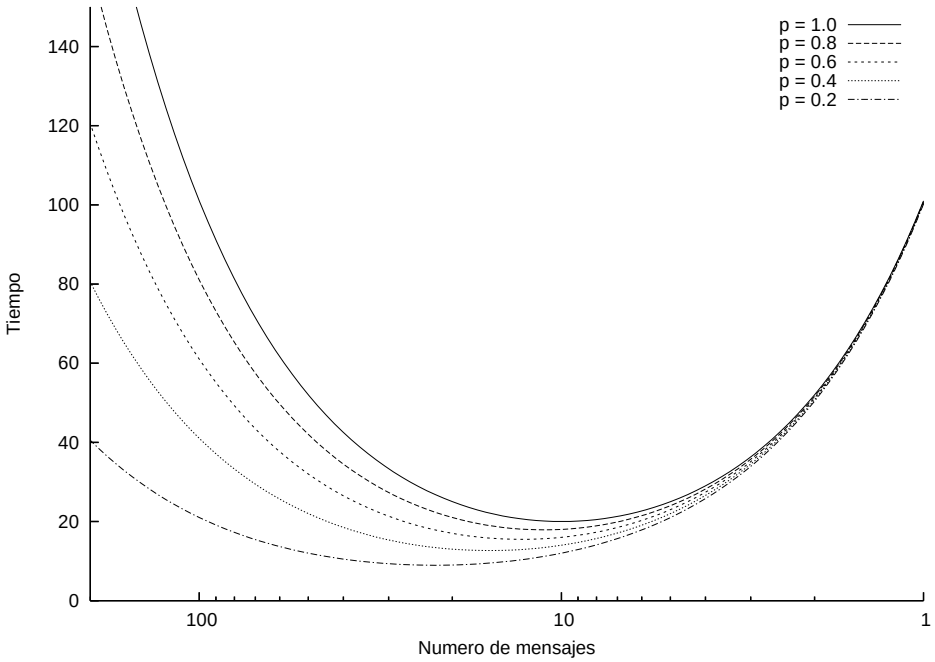


Figura 4.9. Tiempo de ejecución respecto al número de mensajes y a la penalización por mensaje

$$t - \frac{t}{\sqrt{\frac{t}{p}}} - \sqrt{\frac{t}{p}} \cdot p = t - \frac{t \cdot \sqrt{\frac{t}{p}}}{\sqrt{\frac{t}{p}}^2} - \sqrt{\frac{t}{p}} \cdot p = t - \sqrt{\frac{t}{p}} \cdot p - \sqrt{\frac{t}{p}} \cdot p =$$

$$= t - 2\sqrt{\frac{t}{p}} \cdot p = t - 2\sqrt{t} \cdot \sqrt{p}$$

En el modelo presentado se han ignorado varios factores que también tienen un papel a la hora de calcular de forma exacta la aceleración posible. Sin embargo, no suponen una contradicción para la idea que se intenta recalcar. Algunos de estos parámetros son que la penalización tiene una componente proporcional al tamaño del mensaje, que hay que tener en cuenta la latencia de los mensajes que viajan a través de la red, etc. A pesar de la abstracción acometida en las fórmulas, sigue siendo válida la conclusión: no estamos comparando granularidad gruesa con fina, sino granularidad fina con granularidad muy fina.

El tiempo de ejecucución necesario, en el caso óptimo, puede deducirse nuevamente de

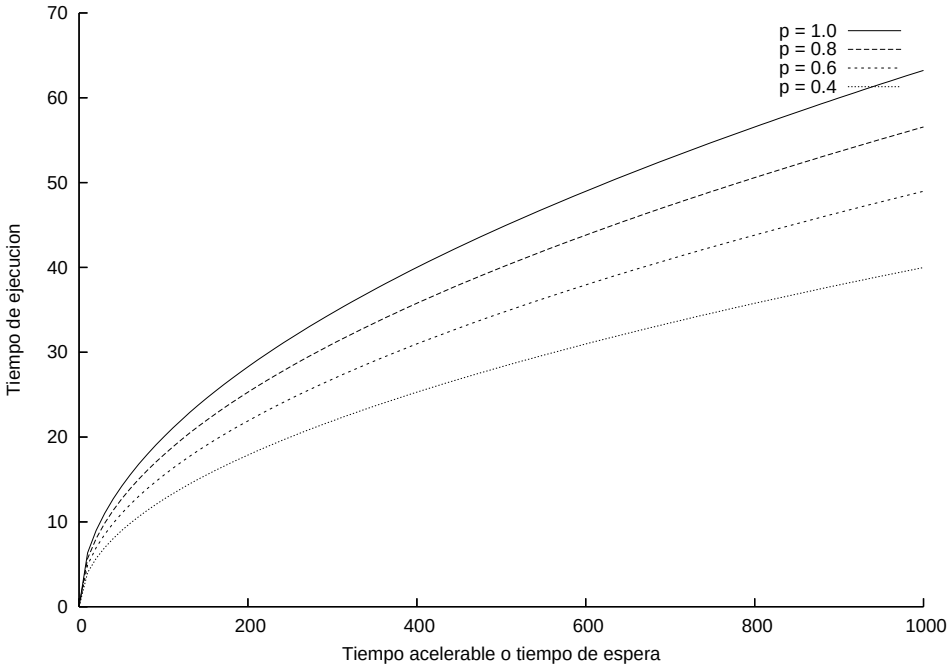


Figura 4.10. Tiempo óptimo en función del tiempo acelerable

forma sencilla:

$$T_{optimo}(n) = 2\sqrt{t} \cdot \sqrt{p}$$

Esta vez ilustramos esta ecuación en función del tiempo t y de la penalización p (figura 4.10). Como se ve, cuanto mayor es el tiempo de espera (tiempo acelerable), mayor es la aceleración que se puede conseguir al ajustar adecuadamente el número de mensajes. También es de esperar que a menor penalización por mensaje mayor aceleración se conseguirá, ya que permite un número más elevado de mensajes.

Como detalle final, si la penalización tiende a cero, es decir, en una red extremadamente eficiente, tenemos lo siguiente:

$$\lim_{p \to 0}(t - 2\sqrt{t} \cdot \sqrt{p}) = t$$

La mejora posible es equivalente a todo el tiempo de espera existente.

Capítulo 5

Sobre la Escalabilidad

Hasta este momento, la discusión ha girado entorno a si un tipo de granularidad u otro puede acelerar en mayor o menor medida la ejecución del programa. Cabe plantearse si esta granularidad puede influir en la escalabilidad del programa. En otras palabras, existe la incógnita sobre si cambiando la filosofía de comunicaciones de la aplicación, se consigue que ésta se pueda ejecutar en más procesadores, lo que significaría todo un logro.

Remarquemos el concepto de escalabilidad: por este concepto se entiende la cualidad de una aplicación por la que al disponer de más recursos (procesadores) disminuye en proporción el tiempo requerido para completar sus tareas. Una escalabilidad perfecta significa que al duplicar el número de procesos, el tiempo necesario se reduce a la mitad. Obviamente, es muy difícil alcanzar la escalabilidad perfecta, entre otras razones porque ciertas partes del programa pueden tener un límite de paralelización posible. La escalabilidad también puede venir limitada por cuestiones de red, por ejemplo, si el número de procesadores aumenta pero no lo hace en igual medida las prestaciones de la red (por ejemplo, red en bus), al aumentar el número de procesos aumentan las comunicaciones lo que hace aumentar el tráfico de red; si esta red no tiene mayores prestaciones, se convierte en el cuello de botella que limíta la escalabilidad.

Vamos a suponer que el algoritmo de la aplicación es susceptible de ser paralelizado infinitamente. Esto significa que aumentando el número de procesos, la tarea a realizar se reparte perfectamente entre ellos, con lo que a mayor número de procesadores menor cantidad de cómputo realiza cada proceso.

Parece razonable que la cantidad de cómputo tenga relación directa con el tamaño de los datos a procesar, es decir, que a mayor número de procesadores menor cantidad de computo se asigna a cada uno, con lo que menor información hay que enviar a cada proceso. Supongamos que el tiempo que tardaba un mensaje en enviarse a un proceso se solapaba

con el cómputo que realizaba el proceso receptor. Siguiendo con la relación descrita, si aumenta el número de procesadores disminuirá la cantidad de computo que debe realizar cada proceso, pero como también disminuirá la cantidad de información a enviar a cada uno de ellos, el solapamiento seguirá funcionando correctamente.

Para la red lenta, donde el envío de un mensaje implica una penalización constante, esto no es así. Cuando el número de procesadores aumenta, el tamaño de los mensajes disminuye, pero no lo hace en igual proporción el coste de un envío, ya que llegados a cierto límite, este coste no es proporcional a la cantidad de datos a enviar. Llegados a este punto, la sobrecarga de los envíos no compensa la aceleración obtenida por el empleo de un mayor número de procesadores.

Lo que parece evidente es que la red lenta hace que las aplicaciones escalen peor que una red rápida. Sin embargo, esto no es de lo que se dudaba ya que el propio nombre de la red indica este hecho. El estudio de cómo escala una cierta aplicación en una cierta red puede ser más o menos interesante, pero no es el tema que nos ocupa. Lo que se ponía en duda es si la granularidad óptima en una red lenta era también la óptima en una red rápida, en este caso, óptima con respecto al escalado. Dicho en otras palabras, si con una cierta granularidad se consigue el mejor escalado en una red lenta hay que ver si esa misma granularidad permite el mejor escalado en la red rápida. Aquí está la cuestión a dilucidar.

Para dar contestación a esta pregunta, debemos plantearnos qué dos filosofías podemos emplear a la hora de diseñar el algoritmo para la red lenta. Anteriormente se ha visto que o bien se pueden enviar mensajes lo más grande posible o bien se pueden fraccionar estos envíos para utilizar mensajes de más reducido tamaño. No obstante, con respecto a la escalabilidad no podemos operar de la misma forma, aunque a priori parezca que sí.

Vamos a poner un caso práctico para ilustrar esto. Supongamos que un proceso debe distribuir una cierta cantidad de información entre *n* procesos. Un posible pseudocódigo que puede emplearse es el siguiente:

```
t_i=t_t/n_p;
for(i=0;i<n_p;i++)
    enviar(t_i,i);
```

Donde t_i es el tamaño del mensaje individual a enviar a cada proceso, t_t es el tamaño total de la información a distribuir y n_p es el número de procesos en la ejecución. Como se ve, este fragmento de código sirve de igual modo para cualquier número de procesos. Digamos que el tamaño máximo a enviar a cada proceso lo determina el número de procesos, no depende del programador. De este modo, no es necesario corregir una aplicación para que escale correctamente en una red rápida, ya que si la aplicación está bien hecha, el

número de procesadores modulara automáticamente el tama de los mensajes.

Volviendo a temas anteriores, si dado un cierto número de procesos la información a enviar a cada uno de ellos es de por ejemplo 1 MB, es decisión del programador que ese MB se envíe en un solo mensaje o en porciones más pequeñas. Esto ya se ha tratado y sirve para mejorar en mayor o menor medida la velocidad del programa; sin embargo, en este apartado estamos tratando la escalabilidad. Que el programador decida particionar un mensaje en varios, no produce un mejor escalado de la aplicación.

La aplicación deja de escalar cuando el tamaño máximo a enviar a cada proceso es demasiado pequeño, hecho que el programador no puede evitar de ninguna forma. Supongamos que el programador había optado por emplear una granularidad fina; en un principio, con pocos procesos los mensajes los podía particionar en varios de menor tamaño. A medida que el número de procesos aumente, el número de mensajes a cada proceso disminuirá hasta llegar a uno. A partir de aquí y según aumente el número de procesos, el programador se verá obligando a reducir el tamaño de este único mensaje.

En definitiva, en este ejemplo se ve que el programador no puede modular el grano de las comunicaciones para que la aplicación escale mejor. Con esto, si no es necesario que el programador intervenga, la pregunta que planteaba si era necesario cambiar la aplicación para ejecutarla en red rápida queda contestada.

Capítulo 6

Conclusiones

A lo largo de los anteriores capítulos, hemos discutido qué repercusiones tiene el grano de las comunicaciones sobre el tiempo de ejecución de las aplicaciones. Esto se ha hecho siguiendo un esquema de trabajo consistente en descubrir cuales son los patrones de comunicaciones que pueden ser explotados por el cambio de granularidad antes de adentrarse en una aplicación concreta.

Gracias a este enfoque, hemos podido analizar de forma concisa las bases sobre las que juega el concepto de grano fino o grueso. Partiendo de aquí, se han analizado programas que utilizan estos patrones mencionados y se han simulado para obtener resultados empíricos que confirmen o no las suposiciones iniciales.

En primer lugar, se ha comprobado que tomar una aplicación y pasarla de granularidad gruesa a fina no es una tarea simple: primero debe entenderse la lógica del problema a resolver; después hay que comprender la paralelización realizada y estudiar el código fuente. Finalmente, en la mayoría de casos habrá que modificar semánticamente el programa, añandiéndole la complejidad que supone intercalar las comunicaciones con el cómputo. Sobre esto es importante comprobar que la mayor necesidad de sincronismo impuesta por la granularidad fina no afecta a las prestaciones de la aplicación (requisito difícil de conseguir). Es decir, nos enfrentamos a un árduo cometido del que es complicado calcular a priori las ventajas, sobre todo a medida que la aplicación crece en complejidad.

La siguiente ventaja es la más importante que durante este estudio se ha encontrado. Nos estamos refiriendo al empleo de un grano fino en las comunicaciones para compensar una posible variabilidad en los tiempos de cómputo entre los distintos procesos. Ya se ha explicado a qué puede ser debida esta variabilidad, por ejemplo a que algunos procesadores se encuentren más ocupados que otros. Reducir el tamaño de los mensajes puede facilitar el reparto de la carga computacional entre los procesos, ya que dinámicamente y según la

velocidad con que hagan sus operaciones se les puede asignar más o menos tarea. A pesar de que el mecanismo de reparto en tiempo de ejecución se prevee de una complejidad nada despreciable, esta medida puede suponer una gran aceleración en el tiempo de ejecución, ya que un solo proceso que compute a mitad velocidad que el resto provocará que la aplicación también reduzca su velocidad a la mitad. Esta línea de estudio se deja planteada como futuro trabajo.

Como contras, a parte de la mencionada dificultad en el manejo de las comunicaciones intercaladas con el cómputo, tenemos la sobrecarga añadida por este manejo. A pesar de que se ha supuesto que en una red on-chip las operaciones MPI se podrán traducir a unas pocas instrucciones en ensamblador, el reparto de información a varios procesos requiere de más instrucciones. Es más, cuando los parámetros de las llamadas a MPI no son conocidos en tiempo de compilación, resulta difícil pensar que se podrían traducir en una o dos instrucciones en ensamblador, cuando en tiempo de ejecución se deberá resolver qué procesos se encuentran dentro del grupo especificado en la llamada y de qué tamaño es el mensaje a enviar, por mencionar algunos parámetros. Todo esto juega en contra de nuestra hipótesis, ya que resta ventajas al uso de granularidad fina. No obstante, depende mucho del tipo de aplicación y la forma en que se requiere que se repartan los datos.

Como conclusión general, tenemos que la hipótesis inicial no ha resultado tan fácilmente alcanzable como parecía. Incluso hemos descubierto varios inconvenientes que no se habían contemplado. Uno de los obstáculos más notorios ha sido que el margen de mejora no ha resultado ser el que se esperaba, con lo que las posibles optimizaciones no han tenido la repercusión esperada.

También nos hemos encontrado con algunas incógnitas que presentaba la hipótesis inicial y que hemos tenido que resolver, en concreto: la granularidad óptima en red lenta. La granularidad óptima en red lenta no ha resultado ser la granularidad más gruesa, con lo que a partir de este hecho ya no se ha comparado granularidad gruesa con granularidad fina, sino granularidad fina con granularidad muy fina. Pongamos un ejemplo práctico: enviar mensajes de 1 MB en red lenta puede no ser la elección idónea; es posible que realizar diez envíos de 100 KB no comporten una penalización grande y en cambio ayuden a reducir en un 90 % el tiempo de espera de la aplicación.

Después de lo visto, cada aplicación MPI debe sintonizarse a las características propias de cada red, para adaptar su tamaño de mensaje al adecuado. Con esto, la granularidad óptima en una red lenta puede suponer unos mensajes de tamaño moderado. Ahora, transladándose a una red rápida, la mejora obtenida por pasar de este tamaño a uno menor es posible que no se haga de notar.

Apéndice A

Implementación de my_cpu_time

Ahora describiremos las modificaciones que hay que realizar en el kernel para tener disponible esta funcionalidad. La llamada al sistema se ha implementado sobre el código fuente del kernel versión 2.6.20, aunque portarla a cualquier otra versión es muy sencillo ([3]). También se ha pensado en la arquitectura *x86_64*, aunque ligada a ella solo está el método de extracción de ciclos de procesador (se emplea la instrucción de lenguaje ensamblador rdtsc de *x86*).

El proceso se puede desglosar en dos fases: en la primera se dará de alta la llamada a la función del sistema para que sea accesible desde el espacio de usuario y en la segunda fase se implementará su funcionalidad como tal.

Primera fase:

En el fichero *include/asmx86_64/unistd.h* podemos dar de alta la nueva llamada:

```
#define __NR_my_cpu_time 280
__SYSCALL(__NR_my_cpu_time, sys_my_cpu_time )
```

de esta forma, cuando lancemos una interrupción de llamada al sistema, al pasar como argumento el número 280, el kernel sabrá a qué función llamar, en este caso a *sys_my_cpu_time*. También debemos actualizar el valor de *__NR_syscall_max*, aumentándolo en 1. NOTA: Las versiones del kernel anteriores a la 2.6 difieren ligeramente en este punto.

Segunda fase:

Ahora debemos implementar la función. Esto se hace en el archivo *kernel/sys.c*:

```
asmlinkage int sys_my_cpu_time(unsigned long long * tiempo){
    unsigned long long retorno;
    struct task_struct *aux;
    aux=current;
```

```
    //Para ser precisos, hay que sumar el tiempo que
    //actualmente lleva en la CPU
    retorno=aux->tiempo_cpu+(tiempo_actual()-aux->ultimo_acceso_cpu);
    if (copy_to_user(tiempo, &retorno, sizeof(unsigned long long)))
        return -1;
    return 0;
}
```

La macro *current* devuelve un puntero a la *task_struct* del proceso que actualmente está en la CPU. A esta estructura se le han añadido dos campos: *tiempo_cpu* y *ultimo_acceso_cpu*, cuyo significado es el tiempo total que el proceso ha estado en la CPU y el instante en que el proceso entró por última vez en la CPU respectivamente. *tiempo_actual()* es una función que devuelve el valor del contador de ciclos del procesador. Finalmente, una vez realizados los cálculos oportunos copiamos el resultado en la variable del usuario.

Para añadir los dos campos anteriormente mencionados, modificaremos el archivo *include/linux/sched.h*, en donde encontramos la definición de la estructura *task_struct*. Al final de ésta añadimos los dos campos (es importante que sea al final de la estructura porque en otras partes del kernel se accede a los primeros campos de ella por desplazamiento):

```
  unsigned long long ultimo_acceso_cpu;
  unsigned long long tiempo_cpu;
```

También es necesario que cuando se cree el proceso se inicialicen estos campos. Los procesos se crean mediante llamadas a la función *fork*; por esto añadiremos las siguientes líneas en el archivo *kernel/fork.c*:

```
//Linea 1371
p = copy_process(clone_flags, stack_start, regs, stack_size,
 parent_tidptr, child_tidptr,nr);
if (!IS_ERR(p)) {
    ////////////////////////////
    //Vamos a inicializar los contadores de tiempo de CPU
    ////////////////////////////
    p->ultimo_acceso_cpu=0;
    p->tiempo_cpu=0;
    ...
```

Solo queda actualizar estos campos cuando el proceso entre o salga de la CPU. Estos eventos se producen en *kernel/sched.c*:

```
//Linea 3550
if (likely(prev != next)) {
    ////////////////////////////
```

```
    //Aqui el cambio de contexto es inminente.
    //prev es el proceso que sale y next el que entra
    ////////////////////////////
    if(prev!=NULL)
        prev->tiempo_cpu+=tiempo_actual()-prev->ultimo_acceso_cpu;
    if(next!=NULL)
        next->ultimo_acceso_cpu=tiempo_actual();
    ////////////////////////////
    //Fin de actualizaciones de contadores
    ////////////////////////////
    ...
```

Para finalizar, debemos implementar *tiempo_actual*. Podemos definir esta función en, por ejemplo, *include/linux/time.h* (debemos recordar incluir este fichero en *sched.c* y en *sys.c*):

```
static inline unsigned long long tiempo_actual(void){
    uint32_t lo, hi;
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    return (unsigned long long)hi << 32 | lo;
}
```

NOTA: Esta función utiliza instrucciones en ensamblador propias de arquitecturas *x86*.

Una vez compilado el kernel, ya está listo para usar esta función. Para llamar a *my_cpu_time* desde un programa en C debemos incluir estas líneas:

```
#include "/linux-2.6.20.4/include/asm-x86_64/unistd.h"
int my_cpu_time(unsigned long long *tiempo){
    return syscall(__NR_my_cpu_time, tiempo);
}
```

Bibliografía

[1] The MPI forum: MPI-2: Extensions to the message-passing interface. 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[2] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoe-flinger. Object-based adaptive load balancing for MPI programs. *In Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[3] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.

[4] José Flich Cardo. *Mejora de las prestaciones de las redes de estaciones de trabajo con encaminamiento fuente*. PhD thesis, Universidad Politécnica de Valencia, Jan 2001.

[5] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. *In Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2002.

[6] J. Owens, W. Dally, R. Ho, and D. Jayasimha. Research challenges for on-chip interconnection networks. *IEE Micro*, 27:96–108, Sep,Oct 2007.

[7] P.S. Pacheco. *Parallel Programming with MPI*. O'Reilly, 2005.

[8] D. Wentzlaff, P. Griffin, and H. Hoffmann et al. On-chip interconnection architecture of the tile processor. *IEEE Micro*, pages 15–31, Sep-Oct 2007.