Capítulo 1

Programación Paralela y Distribuida

Domingo Giménez[1], Murilo Boratto [2] y Leandro Coelho[3]

Abstract

The following text is the material of the class sessions of the Laboratory of Parallel and Distributed Programming, shown in ERBASE 2009, in which they are topics listed in the main advances in Parallel Computing and the art of parallel programming, through examples, and current trends in research and technology in Parallel, Distributed and Grid computing.

Resumo

O seguinte texto consiste no material didático das sessões de aula de Laboratório de Programação em Computação Paralela e Distribuída, apresentadas no ERBASE 2009, na qual se encontram enumeradas em tópicos os principais avanços em Computação Paralela, e da arte de programar em paralelo, através de exemplos, e das tendências atuais em pesquisa e tecnologia em Computação Paralela, Distribuídas e Grid.

1.1. Introducción

Se puede decir que el procesamiento paralelo se utilizó desde la aparición de los primeros ordenadores en los años 50 [13]. En 1955, el IBM 704 incluye um hardware para processamento de números de punto flotante (co-procesador). Ea 1956, IBM lanza el proyecto 7030 (conocido como STRETCH) para producir um "supercomputador" para Los Alamos National Laboratory (LANL). El objetivo en esa época era construir una máquina con capacidad computacional 100 veces mayor de la de cualquier otra máquina disponible. En el mismo año, el proyecto LARC (Livermore Automatic Research Computer) comienza a proyectar otro "supercomputador" para el Lawrence Livermore National Laboratory (LLNL). Estos dos proyectos tardarán tres años para producir los dos primeros productos: los supercomputadores conocidos como STRETCH y LARC.

[1]Departamento de Informática y Sistemas, Universidad de Murcia, Spain, domingo@um.es
[2]Universidade Estadual de Bahia
[3]Universidade Federal de bahis

En paralelo a estas iniciativas, muchas otras producen computadores con las arquiteturas más variadas y con diferentes tipos de software.

Las principales razones para la construcción de máquinas paralelas son: diminuir el tiempo total de ejecución de una aplicación, conseguir resolver problemas más complejos, de grandes dimensiones, y proporcionar concurrencia, o sea, permitir la ejecución simultánea de tareas.

Otras razones incluyen: obtener ventajas de los recursos no locales (por ejemplo, utilización de recursos que están en una red distribuida, WAN o la propia internet, cuando los recursos locales son escasos), disminuir costos (en vez de pagar para utilizar un supercomputador, podríamos utilizar recursos baratos disponibles remotamente), sobrepasar los límites de almacenamiento: memoria y disco (para problemas de grandes dimensiones, usar memorias de varios computadores poder resolver el problema de la limitación de memoria presente en una única máquina).

Finalmente, podemos citar una última razón: los límites físicos de computación de una máquina secuencial, que actualmente ya están en la frontera de lo que puede ser practicable en términos de velocidad interna de transmisión de datos y de velocidad de CPU. Además de esto, podemos decir que durante los últimos 10 años, las tendencias en computación apuntan a la presencia continua de la computación paralela, dado que las redes de interconexión continúan avanzando significativamente en términos de velocidad de comunicación y ancho de banda.

En estas sesiones del Laboratorio de Programación Paralela y Distribuida, hablaremos de los principales avances en Computación Paralela y del arte de programar en paralelo, a través de ejemplos, y de las tendencias actuales en desarrollo y tecnología en Computación Paralela.

1.2. Estructura Detallada del Curso

1.2.1. Objetivos del Curso

Las clases del Laboratorio de Programación Paralela y Distribuida tienen como objetivos principales:

- Mostrar la importancia y la innovación de la Computación de Altas Prestaciones.
- Propiciar el entendimiento de los conceptos de Computación Paralela y Distribuida.
- Presentar de manera práctica técnicas y estrategias de desarrollo de software paralelo.
- Enfatizar la utilización de estrategias para diversas plataformas de ejecución paralela.
- Aplicar los conocimientos en situaciones prácticas, principalmente con la formulación de estrategias para problemas reales.

1.2.2. Tipo de Curso

El Laboratorio tiene una orientación práctica, y está organizado en una serie de sesiones en el aula, acompañadas de este texto, donde se resumen las nociones tratadas en el curso y se incluye una serie de trabajos prácticos a realizar por los alumnos.

La propuesta educativa del curso está elaborada para propiciar una rápida y fácil absorción de los contenidos propuestos, pero sin dejar de lado la calidad y cantidad de los temas tratados.

El curso está dividido en **5** partes distribuidas en **4** sesiones:

- **Parte 1** - Introducción a la Computación Paralela: Perspectivas y Aplicaciones (**Sesión 1**)
- **Parte 2** - Programación OpenMP (**Sesiones 1 y 2**)
- **Parte 3** - Programación MPI (**Sesiones 2 y 3**)
- **Parte 4** - Programación Híbrida (**Sesiones 3**)
- **Parte 5** - Esquemas Algorítmicos Paralelos (**Sesiones 3 y 4**)

1.2.3. Material del Curso

Además de este artículo, el material utilizado en el curso consistirá en las transparencias (en `http://dis.um.es/~domingo/investigacion.html`) y de los programas ejemplo que acompañan al libro [1], que se encuentran como material de apoyo en la página del libro en la editorial (`http://www.paraninfo.es/`).

1.2.4. Detalles de los Temas Expuestos

- **Computación Paralela: Perspectivas y Aplicaciones:** En esta sesión inicial se tratarán los principales avances en Computación Paralela, y del arte de programar en paralelo, y de las tendencias atuales en investigación y tecnología en Computación Paralela y Distribuida y de las tendencias de computación en Grid.

- **Programación OpenMP:** OpenMP [24] es una API (Application Program Interface) que posibilita la programación paralela en entornos multiprocesador con memoria compartida, como es el caso de la mayoría de los procesadores actualmente en el mercado. Utilizando modificaciones en los compiladores, esta tecnología permite el desarrollo incremental de aplicaciones paralelas a partir de código fuente serie. Esta norma está definida por un consorcio que reúne importantes fabricantes de hardware y software. El objetivo de este módulo será la presentación de las nociones básicas de programación OpenMP. La metodología consistirá en la presentación de una introducción a los conceptos teóricos de la programación en OpenMP, seguida por una descripción de cómo preparar un entorno computacional para el desarrollo de aplicaciones.

- **Programación MPI:** MPI [18] (acrónimo de Message Passing Interface) es una propuesta de estándar para un interface de paso de mensajes para entornos paralelos, especialmente aquellos con memoria distribuida. En este modelo, una ejecución consta de uno o más procesos que se comunican llamando a rutinas de una biblioteca para recibir y enviar mensajes entre procesos. El objetivo de este módulo será la presentación de las nociones básicas de programación MPI. La metodología consistirá en presentar una introducción con los conceptos teóricos de la programación en MPI, seguida por una descripción de cómo preparar un entorno computacional para el desarrollo de aplicaciones.

- **Programación Híbrida:** Las aplicaciones en *clusters* se pueden programar para utilizar paso de mensajes entre todos los procesadores. Pero es posible obtener mejores prestaciones si se utiliza un modelo híbrido de comunicación con compartición de información por memoria compartida y con memoria distribuida, a través de la fusión de MPI con OpenMP. El objetivo de este módulo será la presentación de ejemplos de programación MPI+OpenMP en sistemas híbridos.

- **Esquemas Algorítmicos Paralelos:** Algunos de los principales esquemas algorítmicos se estudiarán en este módulo, a partir de implementaciones aplicadas a problemas reales. Se estudiarán los siguientes esquemas como caso de estudio: Paralelismo/Particionado de Datos, Esquemas Paralelos en Árbol, Computación *Pipeline*, Esquema Maestro-Esclavo, Granja de Trabajadores, Bolsa de Tareas, Computación Síncrona.

1.3. Computación Paralela: Perspectivas y Aplicaciones

En esta primera sesión introduciremos las ideas generales de la computación paralela, así como distintos enfoques de este tipo de programación y algunos campos de trabajo donde se aplica.

1.3.1. Tipos básicos de Computación Paralela

La Computación Paralela consiste en la explotación de varios procesadores para que trabajen de forma conjunta en la resolución de un problema computacional. Normalmente cada procesador trabaja en una parte del problema y se realiza intercambio de datos entre los procesadores. Según cómo se realice este intercambio podemos tener modelos distintos de programación paralela. Los dos casos básicos son:

- Disponer de una memoria a la que pueden acceder directamente todos los elementos de proceso (procesadores), que se usará para realizar la transferencia de datos. En este caso tenemos el modelo de **Memoria Compartida** (*Shared Memory Model*), que corresponde a sistemas que tienen una memoria compartida para todos los procesadores, pero también a sistemas de **Memoria Virtual Compartida**, donde la memoria está distribuida en el sistema (posiblemente porque módulos distintos de memoria están asociados a procesadores distintos) pero se puede ver como una memoria compartida porque cada procesador puede acceder directamente a todos los módulos de memoria, tanto si los tiene directamente asociados como si están asociados a otros procesadores. Existen herramientas específicas de programación en

memoria compartida. Las más extendidas son *pthreads* [22] y OpenMP [5, 24, 6], que se puede considerar en la actualidad el estándar de este tipo de programación, y que estudiaremos en este curso.

- Que cada procesador tenga asociado un bloque de memoria al que puede acceder directamente, y ningún procesador puede acceder directamente a bloques de memoria asociados a otros procesadores. Así, para llevar a cabo el intercambio de datos será necesario que cada procesador realice explícitamente la petición de datos al procesador que dispone de ellos, y este procesador haga un envío de los datos. Este es el modelo de **Paso de Mensajes**. Hay varios entornos de programación de Paso de Mensajes (PVM [11], BSP [4]...) y el estándar actual es MPI [18, 29], que utilizaremos en este seminario.

1.3.2. Necesidad de la Computación Paralela

La necesidad de la computación paralela se origina por las limitaciones de los computadores secuenciales: integrando varios procesadores para llevar a cabo la computación es posible resolver problemas que requieren de más memoria o de mayor velocidad de cómputo. También hay razones económicas, pues el precio de los computadores secuenciales no es proporcional a su capacidad computacional, sino que para adquirir una máquina el doble de potente suele ser habitual tener que invertir bastante más del doble; mientras que la conexión de varios procesadores utilizando una red nos permite obtener un aumento de prestaciones prácticamente proporcional al número de procesadores con un coste adicional mínimo.

La programación paralela es una solución para solventar esos problemas, pero presenta otras dificultades. Algunas dificultades son físicas, como la dificultad de integración de componentes y la disipación de calor que conlleva, o la mayor complejidad en el acceso a los datos, que se puede llegar a convertir en un cuello de botella que dificulta la obtención de buenas prestaciones aunque se aumente la capacidad computacional teórica del sistema. Hay también dificultades lógicas, como son la mayor dificultad de desarrollar compiladores y entornos de programación eficientes para los sistemas paralelos, que son más complejos que los secuenciales, o la necesidad de utilizar programación paralela en vez de la programación secuencial, que es bastante más sencilla.

La computación paralela se utiliza para reducir el tiempo de resolución de problemas computacionales, o bien para resolver problemas grandes que no cabrían en la memoria de un procesador secuencial. Y para esto es necesario utilizar sistemas de altas prestaciones y algoritmos paralelos que utilicen estos sistemas de manera eficiente.

Los problemas típicos que se abordan con la programación paralela son: problemas de alto coste, o problemas de no tan alto coste pero de gran dimensión, o problemas de tiempo real, en los que se necesita la respuesta en un tiempo máximo.

Así, la comunidad científica usa la computación paralela para resolver problemas que sin el paralelismo serían intratables, o que se pueden resolver con mayor precisión o en menor tiempo usando el paralelismo. Algunos campos que se benefician de la programación paralela son: predicciones y estudios meteorogógicos, estudio del genoma humano, modelado de la biosfera, predicciones sísmicas, simulación de moléculas... En

algunos casos se dispone de cantidades ingentes de datos que sería muy lento o imposible tratar con máquinas convencionales. En problemas de simulación (meteorología, moléculas...) en muchos casos se discretiza el espacio a simular, y una malla más fina produciría una simulación más precisa, pero necesita de más recursos computacionales, y por tanto de más velocidad de cómputo y espacio de memoria.

1.3.3. Paralelismo en computadores secuenciales

La idea del paralelismo no es exclusiva de los multicomputadores o de las redes de procesadores, sino que, como se ha indicado en la introducción, se viene utilizando en diferentes formas en sistemas computacionales secuenciales desde el desarrollo de los primeros ordenadores:

- La **segmentación encauzada** consiste en descomponer las instrucciones en una serie de instrucciones más simples, que se ejecutan en forma de tubería (*pipe*) de manera que al mismo tiempo se puede estar trabajando en varias instrucciones distintas en distintas partes de la segmentación.

- Es posible disponer de múltiples unidades funcionales, que llevan a cabo operaciones distintas al mismo tiempo, y quizás alguna de ellas especializada en operaciones de un cierto tipo, como pueden ser los coprocesadores matemáticos.

- El **paralelismo a nivel de instrucción** consiste en posibilitar la ejecución de varias instrucciones al mismo tiempo. Pueden utilizarse varias técnicas, como la segmentación encauzada o el uso de varias unidades funcionales, y se pueden combinar las distintas técnicas entre sí.

- La memoria se divide en bloques, de manera que es posible estar accediendo en el mismo momento a bloques distintos, posiblemente en un bloque leyendo y en otro escribiendo.

- La memoria está organizada jerárquicamente, con distinta velocidad de acceso a las memorias según el nivel en que se encuentran. Típicamente el acceso es más rápido a los registros, en un siguiente nivel están las memorias cache (que pueden tener a su vez varios niveles), a continuación la memoria principal, y por último la memoria secundaria. Así, una vez que se accede a un bloque de memoria, este pasa a la memoria cache más cercana al procesador, y el trabajo con esos datos se hará más rápido, mientras que se puede estar accediendo a zonas de memoria en otro nivel de la jerarquía para actualizar los datos que se acaban de modificar...

- La **ejecución fuera de orden** consiste en detectar en el código instrucciones que no dependen unas de otras, y ejecutarlas en un orden distinto al que aparecen. Puede realizarse creando una bolsa de instrucciones por ejecutar, que se realizarán cuando estén disponibles los operandos que necesitan.

- En la **especulación** se trata de ejecutar al mismo tiempo instrucciones distintas, de manera que cuando sea necesario utilizar el resultado de estas instrucciones no haya que esperar a que se obtengan. Un caso típico es tener una sentencia `if` con

un `else`, y disponer de dos unidades en las que se pueden evaluar al mismo tiempo las sentencias de los dos bloques (`if` y `else`) de manera que cuando se evalúa la condición se toma el resultado del bloque correspondiente. También se puede ejecutar sólo el bloque que parece más probable que haya que ejecutar, y cuando se llega al `if` seguramente tendremos ya disponible el resultado requerido.

- En los **procesadores vectoriales** se dispone de unidades vectoriales, que pueden tratar a la vez varios datos de un vector, ya sea llevando a cabo sobre ellos la misma operación sobre distintos componentes, o realizando distintas suboperaciones sobre compoentes distintos, utilizando un encauzamiento.

- Es también usual disponer de coprocesdores de entrada/salida, que permiten llevar a cabo estas operaciones simultánemente con operaciones de computación.

Esta lista, aun no siendo exhaustiva, da idea de la importancia de la noción de paralelismo y de su utilización en el diseño de arquitecturas secuenciales para acelerar su computación. El estudio detallado de la arquitectura de ordenadores y de la utilización del paralelismo tanto en sistemas secuenciales como paralelos cae fuera de este curso, pero hay multitud de libros que abordan este tema de una manera exhaustiva [16, 14, 26].

Por otra parte, la ley de Moore ([19, 27]) dice que el número de procesadores intregrados se dobla cada 18 meses. Esto produce un incremento en la velocidad de computación de los procesadores, pero si observamos la figura 1.1 comprobamos que este aumento se consigue en la actualidad en los procesadores de Intel con los Dual Core, que incluyen dos núcleos y que necesitan por tanto de la programación paralela de forma explícita para poder obtener las máximas prestaciones que estos sistemas pueden ofrecer. Este tipo de procesadores se usan como componente básico en los procesadores que se comercializan en la actualidad, por lo que se puede decir que la programación paralela se ha convertido ya en el paradigma de programación de la computación a nivel básico.

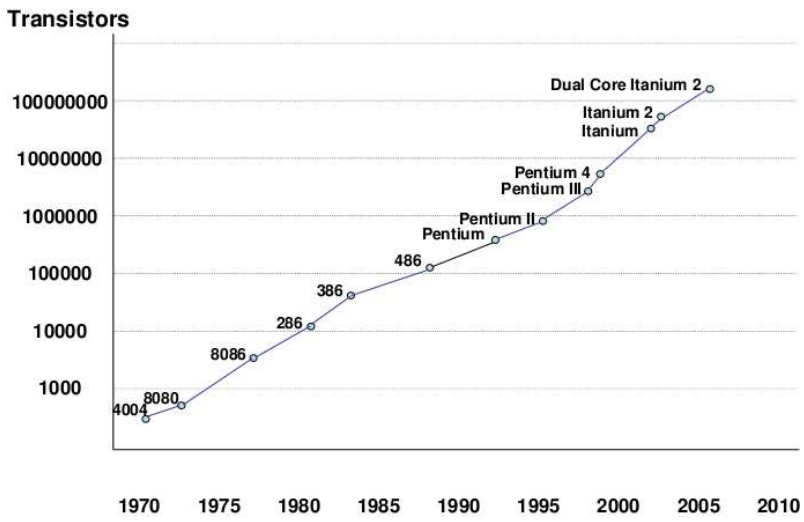

Figura 1.1. Ley de Moore en procesadores Intel

1.3.4. Modelos clásicos de computación

La clasificación de los sistemas paralelos más conocida es la taxonomía de Flynn [8], que los clasifica según el flujo de datos y de instrucciones:

- El modelo SISD (*Single Instruction Single Data*) corresponde al caso secuencial. Se tiene un único flujo de instrucciones que se tratan consecutivamente, y se trabaja sobre un único conjunto de datos. Sabemos que los procesadores secuenciales no siguen exactamente este modelo ya que los datos se agrupan en bloques distintos a los que se puede acceder simultáneamente, que se introduce algo de paralelismo en la ejecución de las instrucciones, por ejemplo, con segmentación, con el uso de múltiples unidades funcionales o de unidades vectoriales..., pero el modelo que se sigue es el SIMD pues el resultado de la ejecución es el mismo que si se ejecutaran las instrucciones una tras otra, y todos los datos se tratan como un único conjunto de datos.

- En el modelo SIMD (*Single Instruction Multiple Data*) se considera un único flujo de instrucciones pero actuando simultáneamente sobre varios conjuntos de datos. Es un modelo paralelo en el que trabajan varios elementos de proceso, ejecutando en cada momento la misma instrucción pero trabajando sobre datos distintos. Por ejemplo, cada proceso podría estar realizando operaciones de suma da datos de un vector, pero cada uno actuaría sobre un vector distinto. Es posible también inhibir procesos en algunas instrucciones, por ejemplo diciendo que los procesos pares trabajan y los impares no. De esta manera se particulariza la ejecución, y aunque todos los procesos están ejecutando las mismas instrucciones en realidad procesos distintos realizan ejecuciones distintas. Sería como tener programas distintos en los distintos procesos, pero con el inconveniente de la sincronización entre la instrucciones. Algunas máquinas paralelas (por ejemplo, algunas versiones de las Connection Machine) se considera que seguían este modelo, pero las máquinas paralelas de propósito general se alejan de él.

- En el modelo MISD se ejecutan varios flujos de instrucciones al mismo tiempo actuando todos ellos sobre el mismo conjunto de datos. Se considera en general que no ha habido desarrollos prácticos de este modelo teórico.

- La gran mayoría de los sistemas paralelos, y en particular de los de propósito general, siguen el modelo MIMD, donde se tienen varias unidades de proceso, cada una con un conjunto de datos asociado y ejecutando un flujo de instrucciones distinto. Si tenemos varios núcleos que comparten la memoria y varios *threads* que se asignan a los núcleos, los *threads* trabajan de manera independiente aunque ejecuten el mismo código, por lo que en un momento dado *threads* distintos van por instrucciones distintas del código, y además pueden acceder a zonas distintas de datos aunque compartan la memoria. Así, las máquinas de memoria compartida siguen el modelo MIMD. Lo mismo ocurre con las de memoria distribuida, pues en este caso cada procesador tiene un proceso que trabaja sobre los datos en la memoria del procesador y los procesos pueden estar ejecutando el mismo o distinto programa, pero no hay sincronización que asegure que vayan por la misma instrucción.

El modelo que utilizamos en este curso es el MIMD, que es el que siguen los multicomputadores actuales, tanto si se utiliza el paradigma de programación por memoria compartida como si se programan con paso de mensajes. Además, consideraremos el **modelo SPMD** (*Single Program Multiple Data*), en el que todos los *threads* o procesos ejecutan el mismo programa pero sin sincronizar la ejecución de las instrucciones, sino que cada elemento de proceso ejecuta las instrucciones a su propio ritmo. Evidentemente, en algunos puntos habrá sincronización, pero ésta vendrá dada por el programa y no será instrucción a instrucción. Por ejemplo, cuando hay un intercambio de mensajes entre dos procesos hay algún nivel de sincronización entre ellos, o puede ser necesario sincronizar todos los *threads* en un punto determinado de la ejecución para que a continuación unos *threads* usen datos generados por otros y que se han almacenado en la memoria común.

1.3.5. Paralelismo en los sistemas computacionales actuales

Una vez que hemos visto la importancia que tiene el paralelismo para aumentar las prestaciones de los sistemas computacionales, nos planteamos los tipos de sistemas paralelos que tenemos en la actualidad a nuestro alcance. Algunos de ellos son:

- Podemos disponer de varios procesadores en un chip. Esta posibilidad se ha universalizado recientemente, al haber sustituido en la práctica los **sistemas multinúcleo** a los monoprocesadores como sistema universal de cómputo, ya que en la actualidad los procesadores duales se comercializan como procesador básico, y es posible adquirir sin mucho coste también tetraprocesadores. Todas las marcas (Intel, SUN, AMD, CELL...) disponen de bi y tetraprocesadores (o sistemas con más núcleos), y en el futuro la tendencia es que el número de núcleos vaya subiendo. Estos sistemas son de memoria compartida, y se programan utilizando *threads*, normalmente con OpenMP. El estilo de programación se prevé que siga siendo de este tipo, quizás con herramientas específicas de desarrollo.

- En algunos casos podemos tener **sistemas empotrados** que realizan una labor determinada y crítica dentro de un proceso industrial. En este caso, al necesitarse de una respuesta en tiempo real, se puede diseñar el sistema con varios procesadores muy simples que realizan siempre la misma operación colaborando entre ellos. Se suelen utilizar por ejemplo FPGA (*Field Programmable Gate Array*) o DSP (*Digital Signal Processors*).

- Los procesadores de propósito específico también están adquiriendo importancia como sistemas paralelos de propósito general. Así, hay procesadores gráficos, DSP, FPGA, procesadores embebidos, procesadores para juegos (PS3...), que se utilizan en muchos casos para desarrollar software para el que no estaban pensados inicialmente. Por ejemplo, se usan GPU y PS3 para computación científica, y no sólo para algoritmos gráficos o de juegos. El principal problema en la actualidad para programar estos sistemas es que no hay herramientas estándar y la programación es más complicada que en las máquinas paralelas tradicionales. Se utilizan *threads* y alguna versión de OpenMP, pero estas herramientas no están estandarizadas, variando la forma de gestionar los *threads*, la representación de los datos... Es previsible que en el futuro se estandaricen las herramientas de programación, y que estos sistemas se

integren con otros en sistemas jerárquicos (GPU y multinúcleo conectados en una red, que a su vez forma parte de un sistema distribuido...) Los procesadores gráficos se están usando cada vez más para computación general, dando lugar a lo que se llama **GPGPU** (*General Processing Graphical Processing Unity*).

- En las **redes de ámbito local** (*Local Area Networks*, LAN) se conectan varios procesadores por medio de una red de conexión de alta velocidad, formándose un *cluster*, que se utiliza con programación por paso de mensajes, pero puede llegar a poderse utilizar OpenMP si se desarrollan versiones eficientes que distribuyan automáticamente los datos en la memoria y generen las comunicaciones necesarias para acceder a los datos. En muchos casos los procesadores básicos son procesadores multinúcleo, que se pueden programar con el paradigma de memoria compartida, y se obtiene así un **sistema híbrido** en el que se combina la memoria compartida y el paso de mensajes.

- Se habla de **supercomputación** para referirse a la resolución en los sistemas computacionales más potentes (supercomputadores) de los problemas que demandan de más computación (meteorología, estudio del genoma, simulación de moléculas...) Tradicionalmente estas computaciones se han venido haciendo en grandes centros de supercomputación, pero cada vez más se está trasladando la resolución de este tipo de problemas a sistemas distribuidos, por lo que se puede considerar la red como el mayor supercomputador actual. En el TOP500 ([30]), que muestra los 500 ordenadores más rápidos en el mundo, se puede ver la evolución de la supercomputación en los últimos años, y se ve que en la actualidad todos los supercomputadores son multicomputadores o *clusters* de ordenadores.

- La **Computación Distribuida** y el **Grid Computing** es computación con procesadores geográficamente distribuidos. Estos procesadores constituyen una red, pero para su utilización se requiere de unas herramientas más sofisticadas que en el caso de redes locales.

- Con la **Web Computing** se ve la red como un recurso computacional, de manera que se puede solicitar en la web la resolución de un problema, y este se asigna a algún o algunos sistemas para su resolución. Se puede tener **Computación P2P** o sistemas tipo SETI [28], donde los usuarios o los laboratorios de computación ponen sus sistemas a disposición de los usuarios para ofrecer recursos computacionales, siempre con algunas restricciones de seguridad, uso...

- Recientemente, nos referimos con **Cloud Computing** a una computación paralela donde se distribuyen los recursos de todo tipo: es posible tener unos determinados centros de computación, otros de almacenamiento, otros se servicios software..., de manera que un usuario podría solicitar un determinado software de un centro, tomar o almacenar datos en otro centro, y requerir de otro que resolviera el problema con que esté trabajando. La combinación de todos estos elementos (Grid, web, P2P, *Cloud computing*, móviles...) y a la vez con sistemas lógicos distintos (compiladores, librerías, software básico...), hace que surja la necesidad de herramientas que permitan la **virtualización**, que consiste en la ejecución simulada en un sis-

tema de la computación que se realizaría en otro de características distintas, y los **sercicios por demanda**.

- Además, los centros mencionados pueden ser laboratorios, computadores personales, dispositivos móviles..., con lo que tenemos lo que se llama **Computación Ubicua**, que se refiere a computación que se puede hacer en cualquier punto, incluso desde un teléfono móvil, ya sea requiriendo desde este la realización de un cálculo o interviniendo nuestro teléfono en la computación.

- Se tiene de esta manera sistemas que son **heterogéneos** (tienen distinta velocidad de computación, capacidad de almacenamiento, distinta forma de representación de los datos...) y que están organizados de forma jerárquica (en computación distribuida tendríamos varios sistemas geográficamente distribuidos, pudiendo ser estos sistemas de distinto tipo, unos de memoria compartida y otros redes de área local, compuestas a su vez por nodos que sean procesadores multinúcleo o procesadores gráficos). Algunos sistemas combinan una CPU y varios núcleos en una GPU; siendo la CPU más veloz, disponiendo de más memoria y con mayor precisión en la representación de los datos. Por otro lado, las GPU son mucho más baratas y más rápidas si se utilizan muchos elementos computacionales que permiten trabajar a muchos *threads* simultáneamente. La tendencia es que también en los sistemas de memoria compartida encontremos sistemas heterogéneos, con núcleos de distinta arquitectura en un mismo chip.

- Se está investigando en la utlización para computación de sistemas no convencionales: en la **Computación Cuántica** se utilizan las propiedades cuánticas de las partículas para acelerar la computación, y en la **Computación Biológica** se utilizan propiedades de sistemas biológicos, como moléculas, células, cadenas de ADN... Esta computación también sería paralela al realizarse al mismo tiempo varias de las acciones que en un procesador convencional hay que realizar una tras otra, pero el tipo de paralelismo es totalmente distinto del paralelismo tradicional que estudiamos en este curso.

1.3.6. Variantes de la programación paralela

Dada la complejidad de los sistemas paralelos, en este seminario nos centramos en el estudio de las nociones básicas de la programación paralela, que son utilizadas en el desarrollo de programas paralelos en todos estos sistemas, aunque en muchos casos se necesita de herramientas y tecnología adicional que nos permita desarrollar y ejecutar sobre ellos los programas paralelos que se diseñen.

También hay varios tipos de algoritmos y programación paralela, y en algunos casos sería necesario pensar los algoritmos de una manera alternativa al enfoque que usamos en el curso si se quiere obtener las máximas prestaciones en el sistema en el que se trabaje. Algunos tipos de programación paralela son:

- Se habla de **Programación Concurrente** para referirse al caso en que varios procesos o *threads* colaboran en la resolución de un problema. Estos procesos pueden compartir el uso de un mismo procesador, en cuyo caso no hablaríamos de programación paralela.

- Para nosotros será programación paralela la que se hace usando varios procesadores para acelerar la resolución de los problemas con que trabajemos. Trabajaremos con algoritmos paralelos, donde se especifica la forma en que los distintos elementos de proceso comparten los datos, y cómo colaboran en la resolución de los problemas. En general consideraremos que los procesadores son todos del mismo tipo, con lo que tendremos **sistemas paralelos homogéneos**.

- En los sistemas heterogéneos los componentes básicos pueden tener distinta velocidad, capacidades de memoria diferentes, o incluso la representación de los datos ser distinta. Los mismos programas de paso de mensajes utilizados en otros sistemas se pueden utilizar aquí, pero las prestaciones vendrán marcadas por la velocidad del procesador más lento. Para sacar el máximo rendimiento de un sistema heterogéneo habrá que desarrollar **algoritmos heterogéneos**, que tengan en cuenta la velocidad de los procesadores para asignarles cantidades distintas de datos en función de su velocidad.

- En los sistemas híbridos también serán válidos los programas de paso de mensajes, que pueden ejecutarse tanto con memoria compartida como distribuida, pero la combinación de MPI con OpenMP puede dar lugar a programas con mejores prestaciones.

- En **sistemas de carga variable** la capacidad de cómputo de los nodos no es siempre la misma (quizás porque es un sistema multiusuario y los trabajos que otros usuarios mandan dinámicamente hacen que durante la ejecución de nuestro programa las condiciones del sistema cambien), y se requiere de **algoritmos adaptativos**, que serán heterogéneos pero en los que además habrá que evaluar periodicamente las condiciones del sistema para cambiar la distribución del trabajo en función de las velocidades en cada momento.

En las sesiones siguientes analizaremos la programación paralela con OpenMP y MPI, básicamente para sistemas homogéneos, aunque este tipo de programación también se usa como base para los otros sistemas mencionados, y analizaremos algún ejemplo de combinación de OpenMP y MPI para programación híbrida.

1.4. Programación OpenMP

OpenMP es el estándar actual para programación en memoria compartida, que incluye los sistemas multicore y computadores de altas prestaciones con memoria virtual compartida. Típicamente se trata de sistemas con un número no demasiado alto de procesadores: desde 2 en un dual hasta alrededor de 128 en un supercomputador. También hay versiones de OpenMP para otros tipos de sistemas, como GPU y clusters de procesadores, pero en estos casos puede ocurrir que no tengamos todavía una herramienta estándar o que las prestaciones que se obtienen con programas en OpenMP estén lejos de las óptimas que se podrían obtener en el sistema si se utilizaran entornos de programación diseñados para ellos. Algunos de estos entornos son Threading Building Blocks [2] para multicore, NVIDIA [23] para GPU, CORBA [7] para sistemas distribuidos, o MPI para paso de mensajes.

En esta sección analizamos las características básicas de OpenMP, utilizando como ejemplos los que se encuentran en `http://www.paraninfo.es/`.

1.4.1. Ejemplo básico: Aproximación de integral definida

Un ejemplo típico es el cálculo del valor de π mediante integración numérica. El valor de π se aproxima con la integral:

$$\int_0^1 \frac{1}{1+x^2} dx = \frac{\pi}{4}$$

Se aproxima el área de la función por el área acumulada de figuras geométricas que la recubren. Una posibilidad es aproximar el área por rectángulos de una cierta base. Cuanto más pequeña sea la base más rectángulos tendremos, y la aproximación será mejor.

`codigo3-1.c` es un programa secuencial para este problema. Contiene un bucle `for` en el que se amumulan las áreas de los rectángulos con los que se aproxima la integral.

Una versión paralela en OpenMP se muestra en `codigo3-16.c`. Se consigue el paralelismo sin más que indicar al compilador la forma en que se va a distribuir el trabajo en ese bucle entre los distintos hilos. Aparecen algunos de los componentes de la interfaz C de OpenMP:

- Hay que incluir la librería de OpenMP (`omp.h`).
- El código es código C con una directiva, lo que se indica con `#pragma omp`.
- La directiva `parallel` indica que se ponen en marcha varios threads que trabajan en paralelo en el bloque de sentencias que hay a continuación (en este caso un bucle `for`).
- Hay una cláusula para indicar cómo se comparten algunas variables.

El modelo de ejecución de OpenMP es el modelo *fork-join*. La ejecución de `codigo3-16.c` tendría los siguientes pasos:

- Inicialmente, cuando se ejecuta el programa trabaja un único thread, que tiene una serie de variables (`int n, i; double pi, h, sum, x;`) que están en la memoria del sistema.
- Este thread pide el número de intervalos a usar e inicializa las variables `h` y `sum`. Este trabajo se hace en secuencial al trabajar un único thread.
- Al llegar al constructor `#pragma omp parallel` se ponen en marcha varios threads esclavos (parte *fork* del modelo). El thread que trabajaba inicialmente es el thread maestro de este conjunto de esclavos. Los threads están numerados desde 0 hasta el número de threads-1. El maestro y los esclavos trabajan en paralelo en el bloque que aparece a continuación del constructor.

- Al ser un constructor `parallel for` lo que se paraleliza y el trabajo que se divide entre el conjunto de threads es el bucle `for`. Como el bucle tiene n pasos, si disponemos por ejemplo de 4 threads, el reparto de trabajo consiste en asignar a cada thread $n/4$ pasos del bucle. Como no se indica cómo se realiza el reparto, se asignan los $n/4$ primeros pasos al thread 0, los siguientes $n/4$ pasos al thread 1, y así sucesivamente.

- Todas las variables de la memoria (`n, i, pi, h, sum, x`) se consideran en una memoria global a la que pueden acceder todos los threads, pero algunas se indica que son privadas a los threads (`private(x, i)`), otra se dice que es compartida de una manera especial (`reduction(+:pi)`), y de las que no se dice nada (`n, h, sum`) son compartidas.

- A las variables compartidas pueden acceder todos los threads en la zona paralela (el bucle `for`), y hay que asegurar en el código que este acceso no produce problemas de coherencia. A la variable `sum` no se accede, a la `n` se accede sólo para controlar el final del bucle, y a la `h` sólo se accede para leer. Al no modificar ningún thread ninguna de las variables compartidas no habrá problema en el acceso a ellas.

- A la variable `pi` acceden todos los threads para escribir en ella, pero como se ha indicado que la forma en que se va a actualizar es siempre con una operación asociativa (`reduction(+:pi)`) el sistema nos asegura que el resultado final sea correcto, independientemente del orden en que los threads accedan a actualizar la variable.

- Cada thread tiene un valor distinto de `i` pues cada uno realiza unas pasadas distintas del bucle, y un valor distinto de `x` porque calculan áreas de rectángulos distintos.

- Al acabar el bucle `for` hay una sincronización de todos los threads, y los esclavos mueren quedando únicamente el thread maestro (parte *join* del modelo).

- El maestro, trabajando en secuencial, es el que calcula el valor final y lo muestra por pantalla.

1.4.2. Compilación y ejecución

Vamos a ver con este primer ejemplo cómo se compila y se ejecuta en paralelo un código OpenMP.

Es necesario disponer de un compilador que pueda interpretar los pragmas que aparecen en el código. El `gcc` tiene esta capacidad desde la versión 4.1. También podemos disponer de versiones comerciales, como el compilador `icc` de Intel, del que suele haber disponible versiones de evaluación. La opción de compilación en `gcc` es `-fopenmp` o `-openmp` y en `icc` es `-openmp`. Así, si compilamos con:

```
gcc -O3 -o codigo3-16 codigo3-16.c -fopenmp
```

se genera `codigo3-16`, que se podrá ejecutar en paralelo poniendo en marcha varios threads. La ejecución se realiza como con cualquier otro programa, pero hay que determinar cuantos threads intervendrán en la región paralela. Hay una variable de entorno

(`OMP_NUM_THREADS`) que nos indica ese número. Si no se inicializa esa variable tendrá un valor por defecto, que suele coincidir con el número de núcleos del nodo donde estemos trabajando. Antes de la ejecución se puede establecer el valor de la variable. Dependiendo del *shell* que estemos usando la inicialización se hará de una u otra manera. Una posibilidad es hacer `export OMP_NUM_THREADS=6`, con lo que establecemos el número de threads en una región paralela a seis, independientemente del número de núcleos de que dispongamos.

Si a continuación ejecutamos el programa (`codigo3-16`) se ejecutará en paralelo con el número de threads que hemos indicado, pero en el código no se indica en ningún lugar el número de threads que se están usando. Además, como se piden los intervalos a utilizar el tiempo que se tarda en leer ese valor forma parte del tiempo de ejecución, y no se está paralelizando.

Se pueden generar varios ficheros de entrada con números de intervalos distintos. Por ejemplo, el fichero `inX` contiene el valor `X`, y si ejecutamos en la forma:

```
codigo3-16 <inX
```

estamos aproximando π con `X` intervalos. Podemos experimentar con un único thread y ejecutar el programa tomando tiempos con una entrada de tamaño pequeño, y a continuación variar el número de threads y volver a tomar tiempos con la misma entrada:

```
export OMP_NUM_THREADS=1
time codigo3-16 <in10000
export OMP_NUM_THREADS=2
time codigo3-16 <in10000
```

y así hasta el número de threads que nos apetezca (seguramente hasta el número de núcleos en el nodo en que estamos trabajando).

Observamos los tiempos de ejecución y es posible que no se tarde menos tiempo con el programa paralelo. Esto puede deberse a varios motivos. Si sólo tenemos un núcleo está claro que no podemos resolver el problema en paralelo más rápidamente que con una ejecución secuencial. Aunque dispongamos de varios núcleos el tamaño del problema (el número de intervalos) puede no ser suficiente grande como para que se note el efecto de la paralelización, si tenemos en cuenta que hay unas zonas que se ejecutan en secuencial y que la gestión de los threads genera una sobrecarga. Si hacemos el experimento anterior con tamaños mayores (quizás con `in10000000`) podemos llegar a ejecuciones en las que se note que el uso del paralelismo reduce el tiempo de ejecución.

Se puede experimentar y buscar explicación a los tiempos de ejecución que se obtienen al variar el número de threads, siendo este número menor que el número de núcleos y cuando excede el número de núcleos.

1.4.3. Formato de las directivas

Las directivas OpenMP siguen las convenciones de los estándares para directivas de compilación en C/C++, son case sensitive, sólo puede especificarse un nombre de directiva por directiva, y cada directiva se aplica, al menos, a la sentencia que le sigue, que

puede ser un bloque estructurado. En directivas largas puede continuarse en la siguiente línea haciendo uso del caracter \ al final de la línea.

El formato general es el siguiente:

```
#pragma omp nombredirec. [cláusulas, ...] nueva-línea
```

donde:

- **`#pragma omp`**. Se requiere en todas las directivas OpenMP para C/C++.
- **`nombredirec.`**. Es un nombre válido de directiva, y debe aparecer después del `pragma` y antes de cualquier cláusula. En nuestro ejemplo es `parallel for`.
- **`[cláusulas, ...]`**. Opcionales. Las cláusulas pueden ir en cualquier orden y repetirse cuando sea necesario, a menos que haya alguna restricción. En nuestro caso son `reduction` y `private`.
- **`nueva-línea`**. Es obligatorio separar la línea que contiene al pragma del bloque estructurado al que afecta.

1.4.4. Creación de threads

Como se ha indicado antes, la directiva con la que se crean threads esclavos es `parallel`:

```
#pragma omp parallel [cláusulas]
     bloque
```

donde:

- Se crea un grupo de threads y el thread que los pone en marcha actúa de maestro.
- Con cláusula `if` se evalúa la expresión y si da un valor distinto de cero se crean los threads, y si es cero se ejecuta en secuencial.
- El número de threads a crear se obtiene por la variable de entorno `OMP_NUM_THREADS` o con llamadas a librería (veremos a continuación cómo se hace).
- Hay barrera implícita al final de la región, con lo que el thread maestro espera a que acaben todos los esclavos para continuar con la ejecución secuencial.
- Cuando dentro de una región hay otro constructor paralelo, cada esclavo crearía otro grupo de threads esclavos de los que sería el maestro. A esto se llama paralelismo anidado y, aunque se puede programar de esta forma, en algunas implementaciones de OpenMP la creación de grupos de esclavos dentro de una región paralela no se realiza.
- Las cláusulas de compartición de variables que soporta la directiva `parallel` son: `private`, `firstprivate`, `default`, `shared`, `copyin` y `reduction`. El significado de estas cláusulas y otras de compartición se verá más adelante.

Los programas `codigo3-11.c` y `codigo3-12.c` muestran el uso de la directiva `parallel` con el típico ejemplo “Hello world". Además se muestran algunas de las funciones de la librería de OpenMP.

En `codigo3-11.c`:

- Cada uno de los threads que trabaja en la región paralela toma su identificador de thread (que están entre 0 y `OMP_NUM_THREADS`-1) usando la función `omp_get_thread_num`, y lo guarda en su copia local de `tid`.
- Por otro lado, todos obtienen el número de threads que hay en la región llamando a `omp_get_num_threads`, que devuelve el número de threads que se están ejecutando dentro de una región paralela. Si se llamara a esta última función desde una región secuencial el resultado sería 1, pues sólo se está ejecutando el thread maestro. El número de threads se guarda en `nthreads`, que es compartida por todos los threads al no haberse indicado entre las cláusulas de `parallel` la forma de compartición de esta variable, y ser `shared` el valor por defecto. Como todos escriben el mismo valor, el orden en que los threads actualicen `nthreads` no importa; pero el acceso a la variable compartida supondrá un tiempo de ejecución adicional por la gestión del sistema del acceso a variables compartidas.
- Finalmente cada thread saluda escribiendo en la pantalla. Como la ejecución es paralela los mensajes se pueden intercalar en la salida. A diferencia de con `codigo3-16.c`, aquí sí comprobamos el número de threads que se están ejecutando.

En el ejemplo `codigo3-12.c` vemos que:

- Al llamar a `omp_get_num_threads` desde fuera de una región paralela el resultado es 1.
- Se ejecutan dos regiones paralelas, y todo el código que hay fuera de las dos regiones se ejecuta en secuencial por el thread maestro.
- Dentro de las regiones paralelas, aunque están ejecutando todos los threads el mismo código, se puede decidir que threads distintos ejecuten códigos distintos, lo que se hace con una condición en la que se evalúa el identificador del thread. En las dos regiones el thread 0 ejecuta una parte del código que no ejecutan los demás.
- La función `omp_set_num_threads` determina el número de threads que trabajarán en la siguiente región paralela. En la primera región se ejecutan 4 y en la segunda 3. El valor establecido por esta función tiene prioridad sobre el que se hubiera puesto en `OMP_NUM_THREADS`.

1.4.5. Cláusulas de compartición de variables

Hemos visto en los ejemplos anteriores que las directivas OpenMP tienen cláusulas con las que indicar cómo se realiza la compartición de variables, que están todas en la memoria compartida. Cada directiva tiene una serie de cláusulas que se le pueden aplicar. Las cláusulas son:

- `private(lista)`: las variables de la lista son privadas a los threads, lo que quiere decir que cada thread tiene una variable privada con ese nombre. Las variables no se inicializan antes de entrar y no se guarda su valor al salir.
- `firstprivate(lista)`: las variables son privadas a los threads, y se inicializan al entrar con el valor que tuviera la variable correspondiente.
- `lastprivate(lista)`: son privadas a los threads, y al salir quedan con el valor de la última iteración (si estamos en un bucle `for` paralelo) o sección (se indica a continuación el funcionamiento de las secciones).
- `shared(lista)`: indica las variables compartidas por todos los threads. Por defecto son todas compartidas, por lo que no haría falta esta cláusula.
- `default(shared|none)`: indica cómo serán las variables por defecto. Si se pone `none` las que se quiera que sean compartidas habrá que indicarlo con la cláusula `shared`.
- `reduction(operador:lista)`: las variables de la lista se obtienen por la aplicación del operador, que debe ser asociativo.
- `copyin(lista)`: se usa para asignar el valor de la variable en el master a variables del tipo `threadprivate`.

1.4.6. Directivas de división del trabajo

Una vez se ha generado con `parallel` un conjunto de threads esclavos, estos y el maestro pueden trabajar en la resolución paralela de un problema. Hay dos formas de dividir el trabajo entre los threads: las directivas `for` y `sections`.

La directiva `for` tiene la forma:

```
#pragma omp for [cláusulas]
     bucle for
```

donde:

- Las iteraciones se ejecutan en paralelo por threads que ya existen, creados previamente con `parallel`.
- El `bucle for` tiene que tener una forma especial: la parte de inicialización debe ser una asignación; la parte de incremento una suma o resta; la de evaluación es la comparación de una variable entera sin signo con un valor, utilizando un comparador mayor o menor (puede incluir igual); y los valores que aparecen en las tres partes del `for` deben ser enteros.
- Hay barrera implícita al final del bucle, a no ser que se utilice la cláusula `nowait`.
- Las cláusulas de compartición de variables que admite son: `private`, `firstprivate`, `lastprivate` y `reduction`.

- Puede aparecer una cláusula `schedule` para indicar la forma como se dividen las iteraciones del for entre los threads, que puede ser:

  - `schedule(static,tamaño)`: Las iteraciones se dividen según el tamaño que se indica. Por ejemplo, si el número de pasos del bucle es 100, numerados de 0 a 99, y el tamaño es 2, se consideran 50 bloques cada uno con dos pasos (bloques con iteraciones 0-1, 2-3,...), y si disponemos de 4 threads, los bloques de iteraciones se asignan a los threads cíclicamente, con lo que al threads 0 le corresponden las iteraciones 0-1, 8-9,..., al thread 1 las iteraciones 2-3, 10-11,..., al 2 las 4-5, 12-13,... y al 3 las 6-7, 14-15,... Si no se indica el tamaño se dividen por igual entre los threads bloques de tamaño máximo: si tenemos 11 iteraciones y 4 threads, se asignan al thread 0 las iteraciones 0-2, al 1 las 3-5, al 2 las 6-8, y al 4 las 9-10.
  - `schedule(dynamic,tamaño)`: Las iteraciones se agrupan según el tamaño y se asignan a los threads dinámicamente cuando van acabando su trabajo. En el caso anterior de 100 iteraciones, tamaño 2 y 4 threads, se asignan inicialmente al thread i las iteraciones $2i$ y $2i+1$, y a partir de ahí el resto de bloques de dos iteraciones se asignan a los threads según vayan quedando sin trabajo. Cuando el volumen de computación de cada iteración no se conoce a priori puede ser preferible utilizar una asignación dinámica.
  - `schedule(guided,tamaño)`: Las iteraciones se asignan dinámicamente a los threads pero con tamaños decrecientes hasta llegar al tamaño que se indica (si no se indica nada el valor por defecto es 1). El tamaño inicial del bloque y la forma en que decrece depende de la implementación.
  - `schedule(runtime)`: Deja la decisión para el tiempo de ejecución, y se obtiene de la variable de entorno `OMP_SCHEDULE`.

El `codigo3-13.c` muestra un ejemplo de uso de la directiva `for`, que se usa dentro de una `parallel`. Se muestra qué thread realiza cada paso del bucle. Se puede utilizar este ejemplo para ver el efecto de la cláusula `schedule` en la división del trabajo.

La forma de la directiva `sections` es:

```
#pragma omp sections [cláusulas]
    {
        [#pragma omp section]
            bloque
        [#pragma omp section
            bloque
        ...
        ]
    }
```

donde:

- Cada sección se ejecuta por un thread. La forma en que se distribuyen las secciones a los threads depende de la implementación. Se puede utilizar el ejemplo `codigo3-14.c` para ver cómo se asignan cuando se utilizan menos threads que secciones, y se pueden incluir más secciones o secciones con distinto coste computacional.
- Hay barrera al final a no ser que se utilice la cláusula `nowait`.
- Las cláusulas de compartición de variables que admite son: `private`, `firstprivate`, `lastprivate` y `reduction`.

Dado que `for` y `sections` se usan con mucha frecuencia inmediatamente después de haber creado los threads con `parallel`, hay dos directivas combinadas:

```
#pragma omp parallel for [cláusulas]
    bucle for
```

y

```
#pragma omp parallel sections [cláusulas]
```

que son una forma abreviada de poner una directiva `parallel` que contiene una única directiva `for` o `sections`. Las dos tienen las mismas cláusulas que `for` y `sections`, salvo la `nowait`, que en este caso no está permitida pues acaba el `parallel`, con lo que el maestro tiene que esperar a que acaben todos los esclavos para seguir la ejecución secuencial.

El ejemplo `codigo3-16.c` contenía la directiva `parallel for`, y en el `codigo3-15.c` también se usa esta directiva, ahora mostrando el paso por cada iteración, y con la cláusula `default(none)`, con lo que hay que especificar la forma de compartición de todas las variables.

1.4.7. Directivas de sincronización

Dentro de una región paralela puede ser necesario sincronizar los threads en el acceso a algunas variables o zonas de código. OpenMP ofrece varias primitivas con esta finalidad:

- `single`: el código afectado por la directiva lo ejecutará un único thread. Los threads que no están trabajando durante la ejecución de la directiva esperan al final. Admite las cláusulas `private`, `firstprivate` y `nowait`. No está permitido hacer bifurcaciones hacia/desde un bloque `single`. Es útil para secciones de código que pueden ser no seguras para su ejecución paralela (por ejemplo, entrada/salida).
- `master`: el código lo ejecuta sólo el thread maestro. El resto de threads del equipo se saltan esta sección del código.
- `critical`: protege una sección de código para que acceda un único thread cada vez. Se le puede asociar un nombre de la forma:

```
#pragma omp critical [nombre]
```

de forma que puede haber secciones críticas protegiendo zonas distintas del programa, de manera que a secciones con nombres distintos pueden acceder al mismo tiempo varios threads. Las regiones que tengan el mismo nombre se tratan como la misma región. Todas las que no tienen nombre, son tratadas como la misma. No está permitido hacer bifurcaciones hacia/desde un bloque `critical`.

- `atomic`: asegura que una posición de memoria se modifique sin que múltiples threads intenten escribir en ella de forma simultánea. Se aplica a la sentencia que sigue a la directiva. La sentencia tiene que ser de la forma: `x op.bi. = expr`, `x++`, `++x`, `x--` o `--x`; con *x* una varible escalar, `expr` una expresión escalar que no referencia a *x* y `op.bi.` una de las operaciones binarias: `+`, `*`, `-`, `/`, `&`, `^`, `|`, `>>` o `<<`. Sólo se asegura en modo exclusivo la actualización de la variable, pero no la evaluación de la expresión.

- `barrier`: sincroniza todos los threads del equipo. Cuando un thread llega a la barrera espera a que lleguen los demás, y cuando han llegado todos, todos siguen ejecutando las siguientes sentencias.

- `threadprivate`: se utiliza para que varibles globales se conviertan en locales y persistentes a un thread de ejecución a través de múltiples regiones paralelas.

- `ordered`: asegura que el código se ejecuta en el orden en que las iteraciones se ejecutan en una ejecución secuencial. Puede aparecer sólo una vez en el contexto de una directiva `for` o `parallel for`. No está permitido hacer bifurcaciones hacia o desde un bloque `ordered`. Sólo puede estar un thread ejecutándose simultáneamente en una sección `ordered`. Una iteración de un bucle no debe ejecutar la misma directiva `ordered` más de una vez, y no debe ejecutar más de una directiva `ordered`. Un bucle con una directiva `ordered` debe contener una cláusula `ordered`.

- `flush`: tiene la forma:

  ```
  #pragma omp flush [lista-de-variables]
  ```

  y asegura que el valor de las variables se actualiza en todos los threads para los que son visibles. Si no hay lista de variables se actualizarían todas. Hay una directiva `flush` implícita en las directivas: `barrier`; `critical` y `ordered` al entrar y salir; y `parallel`, `for`, `sections` y `single` al salir. No es implícita si está la cláusula `nowait`.

Algunas de estas directivas se usarán en algunos de los ejemplos de la sesión de algoritmos paralelos. Con otras se puede practicar insertando puntos de sincronización en algunos de los códigos anteriores.

1.4.8. Funciones y variables

En los ejemplos anteriores hemos visto el uso de las funciones `omp_set_num_threads`, `omp_get_num_threads` y `omp_get_thread_num`. Otras funciones son:

`omp_get_max_threads`: obtiene la máxima cantidad posible de threads.
`omp_get_num_procs`: devuelve el máximo número de procesadores que se pueden asignar al programa.
`omp_in_parallel`: devuelve un valor distinto de cero si se ejecuta dentro de una región paralela.

Se puede poner o quitar el que el número de threads se pueda asignar dinámicamente en las regiones paralelas (`omp_set_dynamic`), o se puede comprobar si está permitido el ajuste dinámico, con la función `omp_get_dynamic`, que devuelve un valor distinto de cero si está permitido el ajuste dinámico del número de threads.

Se puede permitir o desautorizar el paralelismo anidado con `omp_set_nested`, o comprobar si está permitido, con `omp_get_nested`, que devuelve un valor distinto de cero si lo está.

Hay funciones para el manejo de llaves:
`void omp_init_lock(omp_lock_t *lock)` para inicializar una llave, que se inicializa como no bloqueada.
`void omp_init_destroy(omp_lock_t *lock)` para destruir una llave.
`void omp_set_lock(omp_lock_t *lock)` para pedir una llave.
`void omp_unset_lock(omp_lock_t *lock)` para soltar una llave.
`int omp_test_lock(omp_lock_t *lock)` para pedir una llave pero sin bloquearse.

Hay cuatro variables de entorno. Además de la `OMP_NUM_THREADS`:
`OMP_SCHEDULE` indica el tipo de scheduling para `for` y `parallel for`.
`OMP_DYNAMIC` autoriza o desautoriza el ajuste dinámico del número de threads.
`OMP_NESTED` autoriza o desautoriza el anidamiento. Por defecto no está autorizado.

1.4.9. Otros aspectos de OpenMP

Este curso pretende ser una introducción a la programación paralela a través de los que se pueden considerar como los dos estándares actuales y de los esquemas algorítmicos paralelos básicos. En lo que se refiere a OpenMP hemos analizado sus características básicas a través de unos pocos ejemplos también básicos. En el apartado de esquemas se volverá a utilizar OpenMP para la implementación para memoria compartida de los esquemas que se analizarán. Versiones particulares de distintos vendedores pueden tener ligeras variaciones de la descripción de OpenMP, y desde mayo de 2008 está disponible la especificación v3.0 [31], que contiene algunas directivas más (por ejemplo en lo referente al manejo de tareas), y funciones y variables nuevas.

1.5. Programación MPI

MPI (Message Passing Interface) es una librería de paso de mensajes. Se puede utilizar en programas C o Fortran, desde los que se hace llamadas a funciones de MPI para gestión de procesos y para comunicar los procesos entre sí. MPI no es la única librería disponible de paso de mensajes, pero puede considerarse el estándar actual para este tipo de programación. La librería PVM (Parallel Virtual Machine) es anterior a MPI. PVM fue desarrollada inicialmente para redes de ordenadores, por lo que incluía tolerancia a fallos,

facilidades de creación de procesos..., mientras que MPI surgió como una especificación para paso de mensajes para las máquinas paralelas del momento. Inicialmente no contenía las facilidades de PVM orientadas al uso en clusters, pero con el tiempo ha ido evolucionando y hay versiones de MPI orientadas a tolerancia a fallos (FT-MPI [10]) o a sistemas heterogéneos (HeteroMPI [15], MPICH-Madeleine [21]).

En esta sección analizamos las características generales de la primera especificación de MPI, publicada en mayo de 1994. Esta especificación fue desarrollada por el MPI Forum, una agrupación de universidades y empresas que especificó las funciones que debería contener una librería de paso de mensajes. A partir de la especificación los fabricantes de multicomputadores incluyeron implementaciones de MPI específicas para sus equipos, y aparecieron varias implementaciones libres, de las que las más difundidas son MPICH [20] y LAM-MPI [17]. MPI ha evolucionado y en la actualidad está disponible la versión MPI2 [9] y OpenMPI [25], que es una distribución de código abierto de MPI2.

Así, con MPI se ha conseguido:

- Estandarización, pues las implementaciones de la especificación han hecho que se convierta en el estándar de paso de mensajes y no sea necesario desarrollar programas distintos para máquinas distintas.
- Portabilidad, pues los programas en MPI funcionan sobre multiprocesadores de memoria compartida, multicomputadores de memoria distribuida, clusters de ordenadores, sistemas heterogéneos..., siempre que haya disponible una versión de MPI para ellos.
- Buenas prestaciones ya que los fabricantes han desarrollado implementaciones eficientes para sus equipos.
- Amplia funcionalidad, pues MPI incluye gran cantidad de funciones para llevar a cabo de manera sencilla las operaciones que suelen aparecer en programas de paso de mensajes. En este curso estudiaremos las funciones básicas de MPI e indicaremos el resto de facilidades que ofrece.

1.5.1. Conceptos básicos de MPI

Cuando se pone en marcha un programa MPI se lanzan a la vez varios procesos, todos ejecutando el mismo código, cada uno con sus propias variables (modelo SPMD). A diferencia de en OpenMP no hay un proceso distinguido (en OpenMP el thread maestro).

El código `codigo3-5.c` muestra una versión básica del típico “Hello World” en MPI. En él aparecen algunos de los componentes de MPI:

- Hay que incluir la librería de MPI (`mpi.h`).
- Todos los procesos ejecutan el mismo código desde el principio, con lo que todos tienen variables `myrank` y `size`. A diferencia de OpenMP estas variables son distintas y pueden estar en memorias distintas en procesadores distintos.

- Los procesos trabajan de manera independiente hasta que se inicializa MPI con la función `MPI_Init`. A partir de ese punto los procesos pueden colaborar intercambiando datos, sincronizándose...
- La función `MPI_Finalize` se llama cuando ya no es necesario que los procesos colaboren entre sí. Esta función libera todos los recursos reservados por MPI.
- Las funciones MPI tienen la forma `MPI_Nombre(parámetros)`.
- Al igual que en OpenMP es necesario que los procesos conozcan su identificador (entre 0 y el número de procesos menos 1) y el número de procesos que se han puesto en marcha. Para esto se utilizan las funciones `MPI_Comm_rank` y `MPI_Comm_size`.
- Vemos que estas funciones tienen un parámetro `MPI_COMM_WORLD`, que es una constante MPI y que identifica el **comunicador** constituido por todos los procesos. Un comunicador es un identificador de un grupo de procesos, y las funciones MPI tienen que indicar en qué comunicador se están realizando las operaciones.

1.5.2. Compilación y ejecución

Vamos a ver con este primer ejemplo cómo se compila y se ejecuta un código MPI. La forma de hacerlo puede variar de una implementación a otra. Una forma normal de compilar es (así se hace en MPICH y LAMMPI):

```
mpicc codigo3-5.c -o codigo3-5 -O3
```

donde `mpicc` llama al compilador de C que esté establecido y linca con la librería de MPI. Se indica el código a compilar y las opciones que se pasan al compilador de C.

Una vez generado el código (en este caso `codigo3-5`) la forma de ejecutarlo también depende de la compilación. Lo normal es llamar a `mpirun` pasándole el código a ejecutar y una serie de argumentos que indican los procesos a poner en marcha y la distribución de los procesos en los procesadores. Si se llama en la forma:

```
mpirun -np 4 codigo3-5
```

se están poniendo en marcha 4 procesos (`-np 4`) todos ejecutando el mismo código. No se dice a qué procesadores se asignan los procesos, por lo que se utilizará la asignación por defecto que esté establecida en el sistema: puede ser que todos los procesos se asignen al mismo nodo (que puede ser monoprocesador o tener varios núcleos) o que si hay varios nodos en el sistema se asigne un proceso a cada nodo.

Hay varias formas de indicar al lanzar la ejecución como se asignan los procesos. Por ejemplo, en:

```
mpirun -np 4 -machinefile maquinas codigo3-5
```

se indica que los procesos se asignen según se indica en el fichero `maquinas`, que podría tener la forma:

nodo1
nodo2

y si estamos ejecutando en el nodo0 los cuatro procesos se asignarían en el orden: el proceso 0 al nodo donde estamos ejecutando (nodo0), el proceso 1 al nodo1, el 2 al nodo2, y una vez se ha acabado la lista de máquinas del fichero se empieza otra vez por el principio, con lo que el proceso 3 se asigna al nodo1. La forma de asignación depende de la implementación, y puede ser que al nodo en el que estamos no se le asigne ningún proceso o que entre en la asignación cíclica una vez acabada la lista de máquinas del fichero. La forma de ejecución que hemos indicado es típica de MPICH. También se puede lanzar la ejecución especificando la asignación de procesos con un argumento de `mpirun`:

```
mpirun n0,1,2,3 codigo3-5
```

donde se están lanzando 4 procesos que se asignan a los nodos 0, 1, 2 y 3. Esta forma de trabajar es típica de LAMMPI.

Es posible que para poder lanzar procesos mpi haya que haber inicializado mpi. Así, en LAMMPI hay que inicializarlo con `lamboot` antes de ejecutar `mpirun`. Una vez que se deja de trabajar con mpi se puede llamar a `lamhalt`.

1.5.3. Comunicaciones punto a punto

En el ejemplo anterior los procesos no interaccionan entre sí. Lo normal es que los procesos no trabajen de manera independiente, sino que intercambien información por medio de paso de mensajes. Para enviar mensajes entre dos procesos (uno origen y otro destino) se utilizan las **comunicaciones punto a punto**. El `codigo3-6.c` muestra un "Hello world" con este tipo de comunicaciones. Se utilizan las funciones `MPI_Send` y `MPI_Recv` para enviar y recibir mensajes. La forma de las funciones es:

```
     int MPI_Send(void *buf, int count,
MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

```
     int MPI_Recv(void *buf, int count,
MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
MPI_Status *status)
```

Los parámetros tienen el siguiente significado:

- `buf` contiene el inicio de la zona de memoria de la que se van a tomar los datos a enviar o donde se van a almacenar los datos que se reciben.
- `count` contiene el número de datos a enviar o el espacio disponible para recibir (no el número de datos del mensaje que se recibe pues este tamaño lo determina el proceso que envía).
- `datatype` es el tipo de los datos a transferir, y tiene que ser un tipo mpi (`MPI_Datatype`), que en nuestro ejemplo es `MPI_CHAR`.
- `dest` y `source` son el identificador del proceso al que se envía y del que se recibe el mensaje. Se puede utilizar la constante `MPI_ANY_SOURCE` para indicar que se recibe de cualquier origen.
- El parámetro `tag` se utiliza para diferenciar entre mensajes, y tiene que coincidir su valor en el proceso que envía y el que recibe. En el ejemplo en los dos tiene el valor

1. Se puede utilizar `MPI_ANY_TAG` para indicar que el mensaje es compatible con mensajes con cualquier identificador.

- `comm` es el comunicador dentro del cual se hace la comnicación. Es del tipo mpi `MPI_Comm`, y en el ejemplo se usa el identificador del comunicador formado por todos los procesos.
- `status` referencia una variable de tipo `MPI_Status`. En el programa no se utiliza, pero contiene información del mensaje que se ha recibido, y se puede consultar para identificar alguna característica del mensaje, por ejemplo su longitud, el proceso origen...

El programa muestra la forma normal de trabajar con paso de mensajes. Se ejecuta el mismo programa en todos los procesos, pero procesos distintos ejecutan partes del código distintas: el proceso cero recibe cadenas con el saludo del resto de los procesos, y los demás procesos (`if (myrank!=0)`) saludan y envían otro saludo al cero.

El `codigo3-7.c` muestra una versión mpi del programa de aproximación de π. Cada proceso calcula parte de la integral, determinando los rectángulos a calcular por medio de su identificador de proceso y del número de procesos. Finalmente el proceso cero recibe de los demás las áreas parciales y las acumula. El programa es algo más complicado que el correspondiente programa OpenMP, debido a las comunicaciones necesarias para aceder a datos de otros procesos, lo que no era necesario en OpenMP al estar los datos en una memoria compartida.

En los mensajes hay que utilizar tipos de datos mpi. Los posibles tipos son:

Tipo MPI	**Tipo C**
`MPI_CHAR`	signed char
`MPI_SHORT`	signed short int
`MPI_INT`	signed int
`MPI_LONG`	signed long int
`MPI_UNSIGNED_CHAR`	unsigned char
`MPI_UNSIGNED_SHORT`	unsigned short int
`MPI_UNSIGNED`	unsigned int
`MPI_UNSIGNED_LONG`	unsigned long int
`MPI_FLOAT`	float
`MPI_DOUBLE`	double
`MPI_LONG_DOUBLE`	long double
`MPI_BYTE`	
`MPI_PACKED`	

La comunicación que se establece en `codigo3-6.c` se llama **bloqueante**, pues se realiza algún tipo de sincronización entre el proceso que envía y el que recibe. Con `MPI_Send` y `MPI_Recv` el proceso que envía sigue ejecutándose tras realizar el envío, y el que recibe, si cuando solicita la recepción no ha llegado el mensaje queda bloqueado hasta que este llega.

MPI proporciona otras posibilidades para envíos bloqueantes, por ejemplo `MPI_Ssend`, `MPI_Bsend` y `MPI_Rsend`. Estas funciones se diferencian en la forma en que se gestiona el envío, pudiéndose gestionar el buffer de comunicación o determinar que el proceso tome los datos directamente de la memoria.

La función `MPI_Sendrecv` combina en una llamada el envío y recepción entre dos procesos. Es una función bloqueante.

MPI proporciona también comunicación no bloqueante, en la que el proceso receptor solicita el mensaje y si este no ha llegado sigue la ejecución. Las funciones son `MPI_Isend` y `MPI_Irecv`. Si el proceso receptor no ha recibido el mensaje puede llegar un momento en que forzósamente tenga que esperarlo para seguir la ejecución. Se dispone de la función `MPI_Wait` para esperar la llegada de un mensaje y de la `MPI_Test` para comprobar si la operación se ha completado.

Como práctica se puede modificar `codigo3-7.c` para utilizar comunicaciones asícronas o para que el proceso cero no reciba los mensajes en un orden preestablecido.

1.5.4. Comunicaciones colectivas

Además de las comunicaciones punto a punto, MPI ofrece una serie de funciones para llevar a cabo comunicaciones en las que intervienen todos los procesos de un comunicador. Siempre que sea posible realizar las comunicaciones por medio de estas **comunicaciones colectivas** es conveniente hacerlo así, pues se facilita la programación evitándose por tanto errores, y si las funciones están optimizadas para el sistema donde estamos trabajando su uso dará lugar a programas más eficientes que usando comunicaciones punto a punto.

En el `codigo3-7.c` los procesos, una vez que han calculado las áreas locales las envían al proceso 0, que las acumula para obtener la integral. Esta comunicación se puede hacer con la función `MPI_Reduce`. En `codigo3-8.c` se muestra una versión del cálculo de π usando comunicaciones colectivas. Además de `MPI_Reduce` se usa la función `MPI_Bcast` para enviar al principio desde el proceso 0 a los demás el número de intervalos a usar en la integración. Esta es una forma normal de trabajar en programas por pasos de mensaje: un proceso distinguido (normalmente el 0) realiza la entrada de datos, los distribuye al resto de procesos, todos intervienen en la computación (posiblemente con comunicaciones intermedias) y finalmente la salida de los resultados la realiza también el proceso distinguido.

Enumeramos algunas de las comunicaciones colectivas más útiles:

- `int MPI_Barrier(MPI_Comm comm)` establece una barrera. Todos los procesos esperan a que todos lleguen a la barrera, para continuar la ejecución una vez han llegado todos. Se utiliza un comunicador que establece el grupo de procesos que se están sincronizando. Más adelante se aclarará el uso de comunicadores. Todas las funciones devuelven un código de error.
- `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)` realiza una operación de *broadcast* (comunicación uno a todos), donde se mandan `count`

datos del tipo `datatype` desde el proceso raíz (`root`) al resto de procesos en el comunicador. Todos los procesos que intervienen llaman a la función indicando el proceso que actúa como raíz. En el raíz los datos que se envían se toman de la zona apuntada por `buffer`, y los que reciben almacenan en la memoria reservada en `buffer`.

- `int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)` realiza una reducción todos a uno. Los datos sobre los que hacer la reducción se toman de la zona apuntada por `sendbuf` y el resultado se deja en la apuntada por `recvbuf`. Si el número de datos (`count`) es mayor que uno la operación se realiza elemento a elemento en los elementos del buffer por separado. El resultado se deja en el proceso `root`. La operación que se aplica a los datos viene indicada por `op`. El tipo de operaciones que se admiten viene en la siguiente tabla:

Operación	**Significado**	**Tipos permitidos**
`MPI_MAX`	máximo	Enteros y punto flotante
`MPI_MIN`	mínimo	Enteros y punto flotante
`MPI_SUM`	suma	Enteros y punto flotante
`MPI_PROD`	producto	Enteros y punto flotante
`MPI_LAND`	AND lógico	Enteros
`MPI_LOR`	OR lógico	Enteros
`MPI_LXOR`	XOR lógico	Enteros
`MPI_BAND`	bitwise AND	Enteros y Bytes
`MPI_BOR`	bitwise OR	Enteros y Bytes
`MPI_BXOR`	bitwise XOR	Enteros y Bytes
`MPI_MAXLOC`	máximo y localización	Parejas de tipos
`MPI_MINLOC`	mínimo y localización	Parejas de tipos

- Cuando todos los procesos tienen que recibir el resultado de la operación se usa la función `int MPI_Allreduce(void *sendbuf,void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`.

- Para enviar desde un proceso mensajes distintos al resto de procesos se puede usar `int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`. En el proceso raíz el mensaje se divide en segmentos de tamaño `sendcount`, y el segmento *i*-ésimo se envía el proceso *i*. Si se quiere enviar bloques de tamaños distintos a los distintos procesos o si los bloques a enviar no están contiguos en memoria, se puede utilizar la función `MPI_Scatterv`.

- La inversa de la función `MPI_Scatter` es `int MPI_Gather( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int`

`root, MPI_Comm comm)`. Todos los procesos (incluido el raíz) envían al proceso raíz `sendcout` datos de `sendbuf`, y el raíz los almacena en `recvbuf` por el orden de los procesos. Si se quiere que cada proceso envíe bloques de tamaños distintos se usará la función `MPI_Gatherv`.

- Para mandar bloques de datos de todos a todos los procesos se usa `int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`, donde el bloque enviado por el *i*-ésimo proceso se almacena como bloque *i*-ésimo en `recvbuf` en todos los procesos. Para enviar bloques de tamaños distintos se usa `MPI_Allgatherv`.

- Para mandar bloques de datos distintos a los distintos procesos se utiliza `int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`. De cada proceso *i*, el bloque *j* se envía al proceso *j*, que lo almacena como bloque *i* en `recvbuf`. Existe el correspondiente `MPI_Alltoallv`.

1.5.5. Agrupación de datos y tipos derivados

En este apartado vamos a ver algunas formas de agrupar datos cuando se hacen comunicaciones. Utilizaremos una integral definida pero con intervalos de integración determinados por el usuario. El proceso cero pide el intervalo de integración (dos números reales) y el número de subintervalos para aproximar la integral (un entero), y envía los tres datos al resto de procesos. Usaremos como ejemplos los que aparecen con el nombre `integral...` en la documentación del curso que se encuentra en `http://dis.um.es/~domingo/investigacion.html`.

En `integral.c` se realizan tres envíos para distribuir los datos de entrada, y además los datos se envían con comunicaciones punto a punto. El coste de esta comunicación inicial será $3(p-1)(t_s+t_w)$, pues el proceso cero realiza $3(p-1)$ envíos, llamamos t_s (start-up) al tiempo de iniciar una comunicación y t_w (word-sending) al tiempo de enviar un dato, y suponemos que es igual de costoso enviar un real que un entero. Este coste es excesivo, y en un entorno de paso de mensajes es preferible agrupar datos a comunicar y realizar la menor cantidad de comunicaciones posibles pues los valores de t_s y t_w son muy altos en relación con los costes de computación.

Una posibilidad es utilizar comunicaciones colectivas (programa `integralCC.c`). En este caso el coste es $3(t_{sb}+t_{wb})$, donde llamamos t_{sb} y t_{wb} a los costes de iniciar la comuniación y de enviar un dato cuando se utiliza la función de *broadcast*. Esos costes serán mayores que los de t_s y t_w, pero si la función está optimizada para el sistema no llegarán a ser $p-1$ estos valores.

También es posible agrupar los datos para hacer una única comunicación de tres datos, de forma que el coste sea $t_{sb}+3t_{wb}$. Si tenemos en cuenta que el coste de iniciar una comuniación suele ser mucho más elevado que el de enviar un dato, esta agrupación puede suponer una reducción importante en el tiempo.

Una posibilidad es almacenar los datos en un array de tres elementos y enviarlos a la vez (programa `integralAR.c`).

Otra posibilidad se muestra en `integralPack.c`. Se empaquetan los datos (función `MPI_Pack`) antes de mandarlos y se desempaquetan (función `MPI_Unpack`) una vez recibidos en el mismo orden en que se empaquetaron. En la comunicación se usa el tipo `MPI_PACKED`.

También es posible utilizar tipos derivados. En MPI el programador puede crear tipos derivados de otros tipos, con lo que a partir de los tipos básicos puede crear otros tipos y recursivamente nuevos tipos a partir de estos. La creación de tipos es costosa, por lo que no debe utilizarse si se va a realizar una única comunicación sino cuando se van a realizar muchas veces en el programa comunicaciones con el mismo patrón de datos. En nuestro ejemplo, como sólo se envían los datos una vez, no tiene mucho sentido utilizar tipos derivados, pero a modo de ejemplo mostramos en `integralTD.c` la versión de nuestro programa creando un tipo derivado con dos reales y un entero. Declaramos una estructura con tres campos, los dos primeros `float` y el tercero `int`. En la función de entrada de datos se llama a la función `Build_derived_type`, donde se crea el nuevo tipo `message_type`, que se utiliza a continuación para enviar los datos. Vemos que para la creación del tipo se utilizan dos funciones: en la `MPI_Type_struct` se define la estructura de los datos que conforman este tipo derivado, y con la `MPI_Type_commit` se crea el tipo.

Hay otros mecanismos para crear tipos derivados cuando están formados por datos que son subconjunto de una cierta entrada, por ejemplo:

- `int MPI_Type_contiguous( int count, MPI_Datatype oldtype, MPI_Datatype *newtype)` crea un tipo derivado formado por `count` elementos del tipo `oldtype` contiguos en memoria.

- `int MPI_Type_vector( int count, int block_lenght, int stride, MPI_Datatype element_type, MPI_Datatype *newtype)` crea un tipo formado por `count` bloques, cada uno de ellos con `block_lenght` elementos del tipo `element_type`. `stride` es el número de elementos del tipo `element_type` entre elementos sucesivos del tipo `new_type`. Los elementos pueden ser entradas igualmente espaciadas en un array.

- `int MPI_Type_indexed( int count, int *array_of_block_lengths, int *array_of_displacements, MPI_Datatype element_type, MPI_Datatype *newtype)` crea un tipo derivado con `count` elementos, habiendo en cada elemento `array_of_block_lengths[i]` entradas, y siendo el desplazamiento `array_of_displacements[i]` unidades de tipo `element_type` desde el comienzo de `newtype`.

1.5.6. Comunicadores y topologías

Hasta ahora sólo hemos mencionado unas 30 de las aproximadamente 120 funciones que contiene MPI. De todas ellas sólo son imprescindibles las de inicializar y finalizar el entorno, las dos para obtener el número de procesos y el identificador de proceso y una para enviar y otra para recibir. El resto de funciones están orientadas a facilitar el desarrollo de programas o a hacer estos más eficientes. No podemos en un curso de iniciación como este estudiar en detalle toda la funcionalidad que ofrece MPI, y acabaremos comentando las nociones básicas de comunicadores y topologías.

Hemos visto en los programas anteriores que en las funciones de comunicación se usa la constante `MPI_COMM_WORLD`. Esta constante identifica un comunicador que incluye a todos los procesos. Un **comunicador** es una serie de procesos entre los que se pueden realizar comunicaciones. Cada proceso puede estar en varios comunicadores y tendrá un identificador en cada uno de ellos, estando los identificadores entre 0 y el número de procesos en el comunicador menos 1. Al ponerse en marcha MPI todos los procesos están en `MPI_COMM_WORLD`, pero se pueden definir comunicadores con un número menor de procesos, por ejemplo para comunicar datos en cada fila de procesos en una malla, con lo que se puede hacer broadcast en las distintas filas de procesos sin que las comunicaciones de unos afecten a los otros.

Hay dos tipos de comunicaciones. Los `intracomunicadores` se utilizan para enviar mensajes entre los procesos en ese comunicador, y los `intercomunicadores` se utilizan para enviar mensajes entre procesos en distintos comunicadores. En nuestros ejemplos las comunicaciones serán siempre entre procesos en un mismo comunicador. Las comunicaciones entre procesos en comunicadores distintos pueden tener sentido si al diseñar librerías se crean comunicadores y un usuario de la librería quiere comunicar un proceso de su programa con otro de la librería.

Un comunicador consta de: un **grupo**, que es una colección ordenada de procesos a los que se asocia identificadores, y un **contexto**, que es un identificador que asocia el sistema al grupo. Adicionalmente, a un comunicador se le puede asociar una topología virtual.

Si suponemos que tenemos los procesos en una malla virtual, con $p = q^2$ procesos agrupados en q filas y columnas, y que el proceso x tiene coordenadas $(x \; div \; q, x \; mod \; q)$, para crear un comunicador para la primera fila de procesos se haría;

```
//Se declara el grupo asociado al comunicador de todos los procesos
MPI_Group MPI_GROUP_WORLD;

//Se declara el grupo y el comunicador que se van a crear
MPI_Group first_row_group;
MPI_Comm first_row_comm;


//Almacenamiento de los identificadores de procesos que se incluyen en el comunicador
int *process_ranks;
process_ranks=(int *) malloc(q*sizeof(int));
for(proc=0;proc<q;proc++)
```

```
     process_ranks[proc]=proc;
MPI_Comm_group(MPI_COMM_WORLD,&MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD,q,process_ranks,
     &first_row_group);
MPI_Comm_create(MPI_COMM_WORLD,first_row_group,
     &first_row_comm);
```

`MPI_Comm_group` y `MPI_Group_incl` son locales y no hay comunicaciones, y `MPI_Comm_create` es una operación colectiva, y todos los procesos del comunicador donde se está trabajando deben ejecutarla aunque no vayan a formar parte del nuevo grupo.

Si se quiere crear varios comunicadores disjuntos se puede usar la función `int MPI_Comm_split( MPI_Comm old_comm, int split_key, int rank_key, MPI_Comm *new_comm)`, que crea un nuevo comunicador para cada valor de `split_key`, formando parte del mismo grupo los procesos con el mismo valor. Si dos procesos *a* y *b* tienen el mismo valor de `split_key` y el `rank_key` de *a* es menor que el de *b*, en el nuevo grupo *a* tiene identificador menor que *b*, y si los dos tienen el mismo `rank_key` el sistema asigna los identificadores arbitrariamente. Esta función es una operación colectiva, por lo que todos los procesos en el comunicador deben llamarla. Los procesos que no se incluyen en ningún nuevo comunicador utilizan el valor `MPI_UNDEFINED` en `split_key`, con lo que el valor de retorno de `new_comm` es `MPI_COMM_NULL`.

Si consideramos una malla lógica de procesos como antes, se pueden crear *q* grupos de procesos asociados a las *q* filas:

```
MPI_Comm my_row_comm;
int my_row=my_rank/q;
MPI_Comm_split(MPI_COMM_WORLD,my_row,my_rank,&my_row_comm);
```

En MPI se puede asociar una topología a un grupo de procesos. Una **topología** describe cómo se comunican los procesos entre sí, y son topologías lógicas o virtuales, que se usan para describir el patrón de comunicaciones que nos interesa usar en nuestro programa, o también para facilitar el mapeo de los procesos en el sistema físico sobre el que se van a ejecutar.

Se pueden asociar topologías de grafo en general o de malla o cartesiana. Una topología cartesiana se crea con `int MPI_Card_create( MPI_Comm old_comm, int number_of_dims, int *dim_sizes, int *periods, int reorder, MPI_Comm *cart_comm)`, donde el número de dimensiones de la malla es `number_of_dims`, el número de procesos en cada dimensión está en `dim_sizes`, con `periods` se indica si cada dimensión es circular o lineal, y el valor 1 en `reorder` indica al sistema que se reordenen los procesos para optimizar la relación entre el sistema físico y el lógico.

En una topología de malla se puede obtener las coordenadas de un proceso conocido su identificador con `int MPI_Cart_coords( MPI_Comm comm, int rank, int number_of_dims, int *coordinates)`, y el identificador conocidas las

coordenadas con `int MPI_Cart_rank( MPI_Comm comm, int *coordinates, int *rank)`.

Una malla se puede particionar en mallas de menor dimensión con `int MPI_Cart_sub( MPI_Comm old_comm, int *varying_coords, MPI_Comm *new_comm)`, donde en `varying_coords` se indica para cada dimensión si pertenece al nuevo comunicador. Por ejemplo, si `varying_coords[0]=0` y `varying_coords[1]=1`, para obtener el nuevo comunicador no se varía la primera dimensión pero sí la segunda, con lo que se crea un comunicador por cada fila.

1.6. Programación Híbrida

Hemos analizado las ideas básicas de la programación en memoria compartida (con OpenMP) y por paso de mensajes (con MPI), pero la mayoría de los sistemas actuales combinan estas dos posibilidades, pues normalmente se trabaja con clusters de ordenadores donde cada nodo es un multicore donde se puede programar con OpenMP, pero para la colaboración entre los distintos nodos se utiliza paso de mensajes. En entornos de este tipo es posible utilizar alguna versión de OpenMP, pero al realizarse programación de memoria compartida sobre un sistema distribuido las prestaciones se reducen considerablemente.

Otra posibilidad es utilizar paso de mensajes incluso entre procesos en un mismo nodo, y dentro de un nodo las comunicaciones pueden estar optimizadas y ser más rápidas que entre procesos en nodos distintos. Otra posiblidad es combinar los dos tipos de programación en lo que se llama **programación híbrida**, que combina OpenMP y MPI. De esta manera el programador puede decidir cómo realizar la combinación para utilizar las ventajas de los dos tipos de programación. Comparamos distitos aspectos de los dos paradigmas:

- Grano de paralelismo: OpenMP es más apropiado para paralelismo de grano fino, que es cuando hay poca computación entre sincronizaciones entre los procesos.
- Arquitectura de ejecución: OpenMP funciona bien en sistemas de memoria compartida, pero en sistemas distribuidos es preferible MPI.
- Dificultad de programación: en general los códigos OpenMP son más parecidos a los secuenciales que los códigos MPI, y es más sencillo desarrollar programas OpenMP.
- Herramientas de desarrollo: se dispone de más herramientas de desarrollo, depuración, autoparalelización... para memoria compartida que para paso de mensajes.
- Creación de procesos: en OpenMP los threads se crean dinámicamente, mientras que en MPI los procesos se crean estáticamente, aunque versiones posteriores de MPI permiten la gestión dinámica.
- Uso de la memoria: el acceso a la memoria es más simple con OpenMP al usarse memoria compartida, pero la localización de los datos en la memoria y el acceso simultáneo puede reducir las prestaciones. El uso de memorias locales con MPI puede redundar en accesos más rápidos.

Así, algunas de las ventajas de la programación híbrida son: puede ayudar a aumentar la escalabilidad, a mejorar el balanceo del trabajo, se puede usar en aplicaciones que combinan paralelismo de grano fino y grueso o cuando se mezcla paralelismo funcional y de datos, o cuando el número de procesos MPI es fijo, y puede usarse para reducir el coste de desarrollo por ejemplo si se tiene funciones que usan OpenMP y se utilizan dentro de programas MPI. Pero por otro lado, la combinación de los dos paradigmas puede dificultar la programación.

Hay distintas maneras de combinar OpenMP y MPI, y aquí veremos la forma más simple y más difundida. Se generan procesos MPI, y cada uno de los procesos es el thread maestro de un conjunto de threads esclavos que se crean con OpenMP. Lo que tenemos es un modelo MPI+OpenMP.

En `hello_mpi+omp.c` se muestra un ejemplo híbrido de “Hola mundo". Se compila con `mpicc` y con la opción `-fopenmp` (o la del compilador correspondiente de C que se esté usando), y se ejecuta con `mpirun`. Se utiliza `MPI_Get_processor_name` para obtener el nombre del procesador donde se ejecuta un proceso.

Comparamos versiones OpenMP, MPI y MPI+OpenMP de la multiplicación de matrices:

- En `codigo6-6.c` se muestra una multiplicación con OpenMP. Sólo es necesario indicar que se distribuya el trabajo del bucle más externo de los tres que componen la multiplicación, lo que se indica con `#pragma omp parallel for` en la función `multmat`. El programa recibe tres argumentos que indican el tamaño inicial, el final y el incremento, y se hacen multiplicaciones con los tamaños desde el inicial, mientras no se llega al final, y utilizando el incremento para pasar de un tamaño al siguiente. Se muestra el tiempo que se tarda en cada multiplicación. En el programa hay unas zonas con compilación condicional, por lo que si se compila con la opción `-DDEBUG` se compilan esas partes y al ejecutar se muestran los valores de las matrices.

- En `codigo6-10.c` se muestra la multiplicación con MPI. En este caso se pasa un único tamaño (consideramos siempre matrices cuadradas). Podemos ver que el programa se diferencia bastante más que el programa OpenMP del código secuencial, ya que hay que inicializar y finalizar MPI, obtener el número de procesos y el identificador de proceso, enviar el tamaño de la matriz, distribuir los datos de las matrices que necesita cada proceso, y finalmente acumular los datos de la matriz resultado en el proceso 0. En este caso la comprobación de que los resultados son correctos se hace comparando la matriz resultado obtenida con el programa de paso de mensajes con la obtenida con una multiplicación secuencial. Se usa la función `MPI_Wtime` para la toma de tiempos.

- En `mmhibrido.c` se muestra una versión MPI+OpenMP. Se pasan como argumentos el tamaño de la matriz y el número de threads que cada proceso pondrá en marcha. Se toman tiempos sólo de la multiplicación de matrices, pero hay que tener en cuenta que se invierte tiempo en comunicaciones, al principio para enviar los

datos de las matrices a multiplicar y al final para acumular la matriz resultado. Se puede comprobar la diferencia en tiempos para resolver problemas de un tamaño fijo cuando se varía el número de procesos y threads.

1.7. Esquemas Algorítmicos Paralelos

En esta sección se analizan esquemas algorítmicos de los que más se utilizan en la resolución de problemas en entornos paralelos. Se estudiará un conjunto de esquemas de referencia que constituye la base para la resolución de una gama amplia de problemas reales. Para cada esquema, se indicarán sus características generales y se analizará la programación con OpenMP y/o MPI de algunos ejemplos básicos, utilizando para esto los códigos de [1], de manera que a partir de esos códigos básicos el lector pueda ser capaz de desarrollar otros códigos más elaborados.

1.7.1. Paralelismo de Datos

Con esta técnica se trabaja con muchos datos que se tratan de una forma igual o similar. Típicamente se trata de algoritmos numéricos en los que los datos se encuentran en vectores o matrices y se trabaja con ellos realizando un mismo procesamiento. La técnica es apropiada para memoria compartida, y normalmente se obtiene el paralelismo dividiendo el trabajo de los bucles entre los distintos threads (`#pragma omp for`). Los algoritmos donde aparecen bucles donde no hay dependencias de datos, o estas dependencias se pueden evitar, son adecuados para paralelismo de datos. Si se distribuyen los datos entre los procesos trabajando en memoria distribuida hablamos de particionado de datos.

Este tipo de paralelismo es el que tenemos en la multiplicación de matrices con OpenMP del `codigo6-6.c`, pues en ese caso se paraleliza la multiplicación simplemente indicando con `#pragma omp parallel for` en el bucle más externo que cada threas calcule una serie de filas de la matriz resultado. La aproximación de una integral (`codigo3-16.c`) también sigue este paradigma. Otros ejemplos básicos son:

- La suma secuencial de n números se realiza con un único bucle `for`, que puede paralelizarse con `#pragma omp parallel for`, con la cláusula `reduction` para indicar la forma en que los threads acceden a la variable donde se almacena la suma. El `codigo6-1.c` muestra esta versión de la suma. la paralelización se consigue de una manera muy simple, y se habla de **paralelismo implícito** pues el programador no es el responsable de la distribución del trabajo ni del acceso a las variables compartidas.

  En algunos casos puede interesar (por ejemplo por disminuir los costes de gestión de los threads) poner en marcha un trabajo por cada uno de los threads, y que el programador se encargue de la distribución y acceso a los datos, con lo que se complica la programación. En este caso hablamos de **paralelismo explícito**. El `codigo6-2.c` muestra una versión de la suma de números con paralelismo explícito.

- La ordenación por rango consiste en, dados una serie de datos en un array, obtener el rango que le corresponde a cada uno comparando cada elemento con los demás

y contando los que son menores que él (el rango), para posteriormente situarlo en el array en la posición indicada por su rango. El programa consta de dos bucles, el primero para recorrer todos los elementos y el bucle interno para comparar cada elemento con todos los demás. El `codigo6-3.c` muestra una versión en la que se paraleliza la obtención del rango y a continuación se paraleliza el bucle que se usa para situar cada elemento en la posición indicada por el rango.

Siempre que sea posible es preferible no crear y liberar grupos de procesos de manera innecesaria, pues esto nos añade un coste de gestión. En este ejemplo, como estamos dejando los elementos ordenados en otro array distinto no hay problema de acceso al mismo tiempo a leer y escribir en el array resultado, y cada thread escribe en posiciones distintas, con lo que se puede utilizar un único grupo de procesos (`codigo6-4`).

El `codigo6-5.c` muestra la versión de paralelismo explícito.

- La multiplicación de matrices que habíamos visto usa paralelismo implícito, y en `codigo6-7.c` se muestra la versión con paralelismo explícito.

1.7.2. Particionado de Datos

Se trabaja con volúmenes grandes de datos que normalmente están en vectores o matrices, y se obtiene paralelismo dividiendo el trabajo en zonas distintas de los datos entre distintos procesos. A diferencia de en paralelismo de datos en este caso hay que distribuir los datos entre los procesos, y esta distribución determina a su vez el reparto del trabajo. Lo normal es dividir el espacio de datos en regiones y que en el algoritmo haya intercambio de datos entre regiones adyacentes, por lo que se deben asignar regiones contiguas a los procesadores intentando que las comunicaciones no sean muy costosas. Hay comunicaciones para compartir datos que están distribuidos en la memoria del sistema, por lo que hay un coste de comunicaciones que suele ser elevado, por lo que para obtener buenas prestaciones es necesario que el cociente entre el volumen de computación y el de comunicación sea grande. Vemos algunos ejemplos:

- En `codigo4-3+4+5.c` se muestra una versión de particionado de datos de la suma de números. Inicialmente se generan los datos a sumar en el proceso 0, y se distribuye una cantidad igual de datos a cada proceso. Obviamente el coste de las comuniaciones tiene el mismo orden que las operaciones aritméticas, por lo que no tiene mucho sentido la resolución del problema de esta forma, y se muestra sólo como un ejemplo. En la toma de tiempos se toman por separado los tiempos de envío de datos y de computación, de manera que se puede comprobar el alto coste de las comunicaciones. Si comparamos con la suma de datos con OpenMP (`codigo6-1.c`) vemos que aunque el trabajo que hace cada thread/proceso es el mismo, el programa con paso de mensajes es más complicado.

- Una versión de particionado de datos de la ordenación por rango (`codigo6-8.c`) se obtiene observando que en el programa de paralelismo de datos se asigna a cada thread el cálculo del rango de un bloque de elementos. Así, cada procesador necesitará el bloque de elementos de los que calcula el rango, pero también todos los

elementos del array para poder compararlos con los elementos que tiene asignados. Inicialmente se hace un envío (broadcast) de todo el array desde el proceso 0 a todos los demás. Cada proceso obtiene los elementos de los que tiene que calcular el rango, y devuelve al proceso 0 los rangos que ha calculado, y es el proceso 0 el que obtiene los datos ordenados. El orden de la computación es ahora n^2, y el volumen de datos que se comunican es de orden n. A diferencia de la suma de datos los órdenes son distintos y puede interesar la paralelización.

En algunos casos la ordenación se realiza sobre datos ya distribuidos en el sistema. En `codigo6-9.c` se muestra una ordenación por rango en este caso. Cada proceso obtiene el rango de sus datos comparando con los elementos que tiene, y tenemos una serie de pasos para comunicar el resto de elementos (que se irán moviendo cíclicamente por el sistema considerando una topología lógica de anillo) de manera que cada proceso pueda comparar sus datos con los de los demás procesos. Este esquema se aleja algo del particionado de datos, y se puede ver más como computación síncrona. Muchas veces el considerar un programa como parte de un paradigma u otro es cuestión de preferencias personales.

- El `codigo6-10.c` muestra una multiplicación de matrices con MPI. Se sigue el mismo esquema que en OpenMP: cada proceso calcula un bloque de filas de la matriz resultado. Si multiplicamos $C = AB$, la matriz A se distribuye por filas entre los procesos (el proceso 0 se queda con las primeras filas) con comunicaciones punto a punto, la matriz B se distribuye con un broadcast a todos los procesos, cada proceso realiza la multiplicación de las matrices que tiene asignadas, y por último se acumulan los resultados en el proceso 0 con comunicaciones punto a punto. La computación tiene orden n^3 y las comunicaciones orden n^2. Se puede comprobar que con la versión paralela se puede reducir el tiempo de ejecución, aunque se están duplicando datos en el sistema (la matriz B está en todos los procesos) y en las comunicaciones iniciales intervienen muchos datos. Hay otras muchas versiones de la multiplicación de matrices en paralelo que solventan estos problemas.

1.7.3. Esquemas Paralelos en Árbol

Muchos problemas tienen una representación en forma de árbol o grafo, y su resolución consiste en recorrer el árbol o grafo realizando computaciones. Una técnica para desarrollar programas paralelos es, dado un problema, obtener el grafo de dependencias [12] que representa las computaciones a realizar (nodos del grafo) y las dependencias de datos (aristas), y a partir del grafo decidir la asignación de tareas a los procesos y las comunicaciones entre ellos, que vendrán dadas por las aristas que comunican nodos asignados a procesos distintos. La asignación de los nodos a los procesos debe hacerse de manera que se equilibre la carga, y para minimizar las sincronizaciones o comunicaciones hay que asignar nodos a procesos de manera que se minimice el número de aristas entre nodos asignados a procesos distintos.

Vemos como ejemplo básico el problema de la **suma prefija**: dada una serie de números $x_0, x_1, \ldots, x_{n-1}$ se trata de obtener todas las sumas $s_i = \sum_{j=0}^{i} x_j$, para $i = 0, 1, \ldots, n-1$. Consideraremos un dato por proceso y un número de procesos potencia de dos. Aunque estas condiciones son poco realistas, sólo tratamos de ver el esquema algo-

rítmico básico, no un problema real. En la figura 1.2 se muestra una posible dependencia de datos para obtener todas las sumas prefijas en el caso en que $n = 8$. Inicialmente cada proceso P_i tiene un dato x_i. Cada número indica el x_i inicial de la suma parcial correspondiente, y el final en P_i es siempre x_i. En `codigo6-11.c` y `codigo6-12.c` se muestran versiones OpenMP y MPI para resolver el problema. En los dos casos se utiliza una zona de memoria que contiene los datos (el array *a* en la versión OpenMP y la variable *x* en la MPI) y otras zonas auxiliares para contener datos temporales (array *b* y variable *y*), y se dan $\log n$ pasos. En cada paso se obtiene de qué proceso se toman datos y cuál los toma. En la versión OpenMP todos los threads tienen acceso a los dos arrays y se evitan los conflictos sincronizando antes de cada acceso con `#pragma omp barrier`. En la versión MPI la escritura en *b* se sustituye por un envío y la recepción en *y*. La lectura de *b* y la acumulación en *a* se sustituyen por la recepción y la acumulación sobre *x*, y no hay barreras porque las comunicaciones aseguran la sincronización.

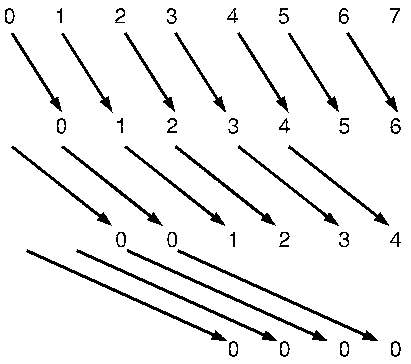

Figura 1.2. Dependencia de datos para el problema de la suma prefija.

Un esquema algorítmico secuencial que sigue una estructura de árbol es el divide y vencerás. Se resuelve un problema descomponiéndolo en una serie de subproblemas de las mismas características que el problema inicial (árbol de descomposición), cada proceso resuelve localmente los subproblemas que tiene asignados, y se combinan las subsoluciones para obtener la solución del problema completo (árbol de combinación).

Utilizamos como ejemplo de paralelización de esquemas divide y vencerás el problema de la ordenación por mezcla [3]. El problema de tamaño *n* se divide en dos subproblemas de tamaño $n/2$, estos en dos subproblemas de tamaño $n/4$, y así recursivamente hasta llegar a un tamaño base que se ordena por algún método directo, y después de van mezclando los subarrays ordenados de tamaño *t* para tener nuevos subarrays ordenados de tamaño $2t$.

El `codigo6-11.c` muestra la versión OpenMP. No hay parte de descomposición del problema, sino que si se dispone de *p* threads y el array de tamaño *n* se divide en subarrays de tamaño n/p, cada uno de los cuales lo ordena un thread, y después se entra en el proceso de mezclas, que se hace con la estructura de árbol. Inicialmente realizan mezclas la mitad de los threads, y el número de threads activo se divide por dos en cada paso. El tamaño inicial de los subarrays a mezclar es n/p, y se multiplica por dos en cada paso.

El `codigo6-12.c` muestra la versión MPI. En este caso la distribución del trabajo sí sigue un esquema de árbol, que es simétrico al usado en la combinación. Los procesos que reciben los datos en la combinación son los que realizan las mezclas. La última mezcla la realiza el proceso 0, que se queda con los datos ordenados.

1.7.4. Computación em Pipeline

Con el esquema *pipeline* se resuelve un problema descomponiéndolo en una serie de tareas sucesivas, de manera que los datos fluyen en una dirección por la estructura de procesos. Tiene una estructura lógica de paso de mensajes, pues entre tareas consecutivas hay que comunicar datos. Puede tener interés cuando no hay un único conjunto de datos a tratar sino una serie de conjuntos de datos, que entran a ser computados uno tras otro; cuando no se necesita que una tarea esté completamente finalizada para empezar la siguiente; o para problemas en los que hay paralelismo funcional, con operaciones de distinto tipo sobre el conjunto de datos, y que se deben ejecutar una tras otra.

Vemos este esquema con el algoritmo de sustitución progresiva para resolver un sistema de ecuaciones lineal triangular inferior:

$$\begin{array}{rcrcccrcl} a_{00}x_0 & & & & & & & = & b_0 \\ a_{10}x_0 & + & a_{11}x_1 & & & & & = & b_1 \\ & & \ldots & & & & & & \\ a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & \ldots & + & a_{n-1,n-1}x_{n-1} & = & b_{n-1} \end{array}$$

donde cada x_i se obtiene con la fórmula:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{ij}x_j}{a_{ii}}$$

El algoritmo secuencial consiste en calcular cada x_i empezando por x_0, y una vez calculados x_0 hasta x_{i-1} se puede calcular x_i. Así, para obtener una versión paralela se puede considerar una estructura pipeline si cada fila del sistema la trata un proceso que pasa la incógnita que calcula al proceso siguiente, de manera que la incógnita fluye por los procesos hasta llegar al encargado de la última ecuación. En `codigo6-17.c` se muestra una versión MPI con esta idea. Vemos que el código es el mismo en cada proceso, pero dependiendo del número de proceso el trabajo que se realiza es distinto: el proceso 0 calcula x_0 y lo envía al proceso 1, el último proceso recibe los distintos x_i y los usa para calcular x_{n-1}, y los procesos intermedios reciben los sucesivos x_i (y los envían inmediatamente para que se solape computación y comunicación) hasta que pueden obtener la incógnita que les corresponde. Esta no es la mejor forma de resolver este problema, y se ve sólo como ejemplo de pipeline. El paralelismo es de grano fino al tener un proceso por ecuación, lo que no tiene sentido en una situación real, pero el programa es fácilmente modificable para obtener una versión en la que un proceso (lo normal sería tener tantos procesos como procesadores en el sistema) se encargue de varias filas. Vemos también que el trabajo no está balanceado al dejar los procesos de trabajar una vez que han calculado su incógnita.

Se podría escribir una versión OpenMP con la misma idea, pero el esquema pipeline tiene una estructura clara de paso de mensajes, y en memoria compartida puede

ser preferible resolver el problema de otra manera. El `codigo6-18.c` muestra una versión OpenMP que usa paralelismo de datos: se paraleliza el bucle interno que actualiza el vector *b* cada vez que se calcula una nueva incógnita (las soluciones se almacenan en el mismo vector *b*). El paralelismo es de grano fino, pero con OpenMP se pueden obtener buenas prestaciones con este tipo de paralelismo.

1.7.5. Esquema Maestro-Esclavo

En el paradigma maestro-esclavo tenemos un thread/proceso distinguido, llamado **maestro**, que pone en marcha otros procesos, llamados **esclavos**, les asigna trabajo y recaba las soluciones parciales que generan.

En algunos casos se habla de **granja de procesos** cuando un conjunto de procesos trabaja de manera conjunta pero independiente en la resolución de un problema. Este paradigma se puede asimilar con el maestro-esclavo si consideramos que los procesos que constituyen la granja son los esclavos (o los esclavos y el maestro si es que éste interviene también en la computación).

Otro paradigma relacionado con estos dos es el de los **trabajadores replicados**. En este caso los trabajadores (threads o procesos) actúan de forma autónoma, resolviendo tareas que posiblemente dan lugar a nuevas tareas que serán resueltas por el mismo o por diferente trabajador. Se gestiona una **bolsa de tareas**, tomando cada trabajador una tarea de la bolsa, trabajando en la solución de esa tarea y posiblemente generando nuevas tareas que se incluyen en la bolsa, y una vez acabado el trabajo de la tarea asignada toma una nueva tarea de la bolsa para trabajar con ella, y así mientras queden tareas por resolver. Se plantea el **problema de la terminación**, pues es posible que un trabajador solicite una tarea y no haya más tareas en la bolsa, pero esto no quiere decir que no queden tareas por resolver, pues es posible que otro proceso esté computando y genere nuevas tareas. La condición de fin será que no queden tareas en la bolsa y que no haya trabajadores trabajando. Hay distintas maneras de abordar el problema de la terminación.

Así, considamos estas técnicas dentro del mismo paradigma maestro-esclavo, aunque puede haber algunas variantes.

Esta es la forma típica de trabajo en OpenMP. Inicialmente trabaja un único thread, y al llegar a un constructor `parallel` pone en marcha una serie de threads esclavos de los que se convierte en el maestro. La mayoría de los ejemplos de OpenMP que hemos visto podrían considerarse dentro de este esquema.

En programación por paso de mensajes también se suele utilizar este esquema. En MPI, al ponerse en marcha los procesos con una orden como `mpirun -np 8 programa`, se inicializan ocho procesos todos iguales, pero suele ocurrir que la entrada y salida se realice a través de un proceso, que actúa como maestro, que genera el problema, distribuye los datos, tras lo cual empieza la computación, en la que puede intervenir o no el maestro, y al final el maestro recibe de los esclavos sus resultados.

La asignación del trabajo a los esclavos puede realizarse de manera estática o dinámica:

- En la asignación estática el maestro decide los trabajos que asigna a los esclavos y

realiza el envío.

- En la asignación dinámica el maestro genera los trabajos y los almacena en una bolsa de tareas que se encarga de gestionar. Los esclavos van pidiendo trabajos de la bolsa de tareas conforme van quedando libres para realizar nuevas tareas. De esta forma se equilibra la carga dinámicamente, pero puede haber una sobrecarga de gestión de la bolsa y esta puede convertirse en un cuello de botella. Además, en algunos problemas, los esclavos al resolver una tarea generan nuevas tareas que deben incluirse en la bolsa, para lo que se generan más comunicaciones con el maestro.

Vemos como ejemplo el **problema de las reinas**. Este problema consiste en, dado un damero cuadrado de tamaño $n \times n$, encontrar las distintas formas en que pueden colocarse n damas de ajedrez en el tablero sin que haya dos damas que puedan comerse. Se resuelve normalmente por *backtracking*, pero consideramos una solución secuencial utilizando una bolsa de tareas, para analizar una paralelización por el esquema maestro-esclavo con bolsa de tareas.

El `codigo6-25.c` muestra una implementación secuencial. Se gestiona la bolsa con una lista de descriptores de tareas, que son configuraciones del tablero con una serie de damas. Como no puede haber más de una dama en cada fila, una configuración viene dada por un array s con valores de 0 a $n-1$, representando $s[i] = j$ que en la fila i se coloca una dama en la columna j. Una fila donde no hay una dama se señala con $s[i] = -1$. Se colocan reinas en las filas desde la primera en adelante, y cada configuración tiene una variable $fila$ para indicar la primera fila donde no se ha colocado una reina. La única tarea inicialmente en la bolsa es: $s[i] = -1,\ \forall i = 0, 1, \ldots, n-1,\ fila = 0$. Se toman tareas de la bolsa mientras no esté vacía. Por cada tarea se generan todas las configuraciones que se obtienen colocando una reina en cada columna en la fila $fila$, y se incluyen en la bolsa las configuraciones válidas.

El `codigo6-26.c` muestra una implementación OpenMP. Todos los threads toman tareas de la bolsa en exclusión mutua (`#pragma omp critical`), y la condición de fin es que todos los threads estén esperando tareas y no queden trabajos en la bolsa. El acceso a la bolsa se puede convertir en un cuello de botella ya que se accede a ella para tomar las tareas y también para insertar cada nueva configuración válida encontrada.

Para evitar el problema del acceso en exclusión mutua se pueden usar varias estrategias. Una consiste en que el proceso maestro genere hasta un cierto nivel del árbol de soluciones, creando una tarea por cada configuración válida en ese nivel. Después genera los threads esclavos para tratar esas tareas con `#pragma omp parallel for`. Ya no se trabaja con una bolsa de tareas, sino que tenemos un esquema maestro-esclavo con paralelismo de datos. El `codigo6-29.c` muestra esta versión. La asignación de las tareas a los threads se hace con `schedule(static)`, pero se podría hacer con asignación dinámica para que se balanceara mejor la asignación de trabajos.

El `codigo6-27+28.c` muestra una versión MPI. Una tarea se almacena en un array de $n+1$ datos, con las posiciones de las reinas en las posiciones de la cero a la $n-1$, y en la posición n la fila por la que se va trabajando. Los mensajes que recibe el maestro

de los esclavos se almacenan en un array de $2n+3$ posiciones. En las $n+1$ primeras posiciones se almacena la tarea que fue asignada al esclavo, en la posición $n+1$ se indica cuántas nuevas tareas se han generado, y las posiciones desde $n+2$ a $2n+1$ se utilizan para indicar las columnas donde se ponen las reinas en la fila $mensaje[n]$ de las $mensaje[n+1]$ tareas generadas. Finalmente, en la posición $2n+2$ se almacena el identificador del proceso que mandó el mensaje, ya que se utiliza la constante `MPI_ANY_SOURCE` para leer mensajes desde cualquier origen. El maestro, cuando recibe nuevas tareas comprueba si corresponden a soluciones, y las escribe, o si no se ha llegado a la última fila, en cuyo caso las inserta en la bolsa de tareas. A continuación envía tareas a cada proceso que está a la espera, y evalúa la condición de fin. Cuando se cumple la condición envía un mensaje a cada esclavo indicando con el valor -1 en la última posición del descriptor de tarea que se ha acabado el trabajo.

1.7.6. Computación Síncrona

Hablamos de paralelismo síncrono cuando se resuelve un problema por iteraciones sucesivas. Las características generales son:

- Cada proceso realiza el mismo trabajo sobre una porción distinta de los datos.
- Parte de los datos de una iteración se utilizan en la siguiente, por lo que al final de cada iteración hay una sincronización, que puede ser local o global.
- El trabajo finaliza cuando se cumple algún criterio de convergencia, para el que normalmente se necesita sincronización global.

Hay multitud de problemas que se resuelven con este tipo de esquemas. Por ejemplo, la ordenación por rango cuando los datos están distribuidos tiene varios pasos para hacer circular los datos por todos los procesos, el algoritmo de Dijkstra de cálculo de caminos mínimos tiene un número fijo de pasos en los que hay computación y comunicación intermedia, hay métodos iterativos de resolución de sistemas de ecuaciones, los algoritmos genéticos (y en general muchas metaheurísticas) trabajan con sucesivas iteraciones hasta que se cumple un criterio de parada...

El ejemplo básico que vamos a ver es lo que se llama **relajación de Jacobi**. Se considera una placa cuadrada con valores fijos en los bordes y unos valores en el interior que varían en el tiempo en función de los valores de sus vecinos, y se quiere calcular los valores que se obtienen en el interior de la placa una vez se estabilizan. Se discretiza el problema considerando una serie de puntos en la placa formando una malla cuadrada. Los valores en cada punto en un momento dado se calculan considerando los valores en sus cuatro vecinos en el instante anterior (se discretiza también el tiempo):

$$V^t(i,j) = \frac{V^{t-1}(i-1,j) + V^{t-1}(i+1,j) + V^{t-1}(i,j-1) + V^{t-1}(i,j+1)}{4} \qquad (1)$$

Cuando la diferencia entre los valores en el instante t y el $t-1$ es suficientemente pequeña acaba la iteración. Se necesitan sincronizaciones o comunicaciones entre procesos

vecinos tras cada iteración para acceder a los datos que no se tiene asignados y que cada proceso necesita para actualizar sus datos. También hay sincronización tras cada paso de computación para evaluar el criterio de convergencia.

No se puede paralelizar al mayor nivel, ya que los valores de cada iteración dependen de los de la iteración anterior. Se paraleliza cada iteración, que consiste en dos bucles para recorrer toda la malla. Se utilizan dos arrays para ir alternando entre ellos de cual se toman datos y en cual se actualizan. Es fácil obtener versiones de memoria compartida de este algoritmo. El `codigo6-19.c` muestra una versión con paralelismo implícito. En cada paso se paralelizan tres bucles, los dos primeros para actualizar los datos de la malla y el último para comprobar el criterio de convergencia (se está comprobando cada dos pasos). En `codigo6-20.c` se muestra una versión con paralelismo explícito. En este caso cada thread calcula el número de filas de la malla que tiene que actualizar, la sincronización se asegura con `barrier`, y el cálculo del valor global para el criterio de convergencia se hace en secuencial (`#pragma omp single`).

La versión MPI (`codigo6-21.c`) se obtiene sustituyendo las barreras del código anterior por comunicaciones, El array de datos tiene n filas, y los arrays locales tl. Hay un envío inicial de $tl+2$ filas a cada procesador, pues se almacenan dos filas adicionales para recibir filas de los dos procesos adyacentes en el array de procesos. Al final se acumulan los datos enviando cada proceso al cero tl filas.

En la relajación de Jacobi el trabajo acaba cuando se alcanza un criterio de convergencia. En otros casos puede haber un número fijo de pasos en los que hay computación seguida de comunicación, que es necesaria para poder dar el siguiente paso. Ejemplo de esto es el **método de Cannon** de multiplicación de matrices. Se consideran matrices cuadradas de dimensión $n \times n$, y una malla $\sqrt{p} \times \sqrt{p}$ de procesos, y por simplicidad suponemos que n es múltiplo de $\sqrt{p}$. Cada proceso tendrá un bloque $n/\sqrt{p} \times n/\sqrt{p}$ de cada una de las matrices. El proceso P_{ij}, con $i, j = 0, 1, \ldots, \sqrt{p}-1$, calcula el bloque C_{ij} de la matriz resultado, y contiene inicialmente los bloques $A_{i,(i+j) \ mod \ \sqrt{p}}$ y $B_{(i+j) \ mod \ \sqrt{p},j}$. La figura 1.3 muestra la distribución inicial y la dirección de las comunicaciones en una malla 3×3.

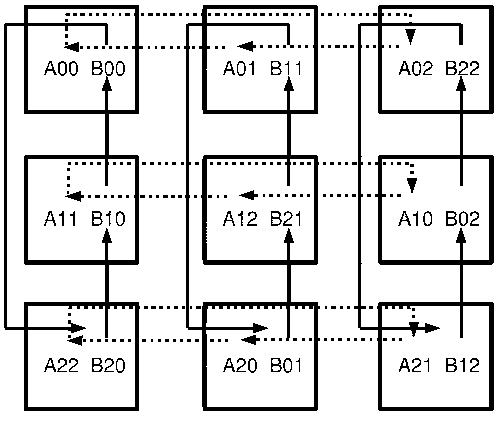

Figura 1.3. Asignación inicial y comunicación de bloques de matrices en la malla de procesos en el algoritmo de Cannon.

Haciendo circular los bloques de A cíclicamente en cada fila y los de B en cada columna se consigue que cada procesador pueda acceder a los bloques de A y B que necesita para calcular su bloque C_{ij}. Se realizan $\sqrt{p}$ pasos de computación, que consisten en multiplicar cada proceso los bloques de A y B que tiene asignados y acumularlos en el bloque C_{ij} que tiene que calcular. En `codigo6-22+23.c` se muestra la implementación de este algoritmo. La ventaja con respecto al método de multiplicación visto como ejemplo de particionado de datos, es que ahora la matriz B no se replica en todos los procesos, con lo que se ahorra memoria, aunque a costa de añadir comunicaciones.

Referências Bibliográficas

[1] Francisco Almeida, Domingo Giménez, José Miguel Mantas, and Antonio M. Vidal. *Introducción a la programación paralela*. Paraninfo Cengage Learning, 2008.

[2] Threading Building Blocks. http://www.threadingbuildingblocks.org/.

[3] G. Brassard and P. Bratley. *Fundamentals of Algorithms*. Prentice-Hall, 1996.

[4] BSP. http://www.bsp-worldwide.org/.

[5] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kauffman, 2001.

[6] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *OpenMP C and C++ Application Program Interface*. OpenMP Architecture Review Board. http://www.openmp.org/drupal/mp-documents/cspec20.pdf, 2002.

[7] CORBA. http://www.omg.org/.

[8] M. J. Flynn. Some computer organizations and their effectivness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[9] MPI Forum. http://www.mpi-forum.org/.

[10] FT-MPI. http://icl.cs.utk.edu/ftmpi/.

[11] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3.0 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Mathematical Sciences Section, Oak Ridge National Laboratory, 1996.

[12] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.

[13] Bob Greson, Angela Burgess, and Christine Miler. Timeline of Computing History, `http://www.computer.org/computer/timeline/timeline.pdf`.

[14] Hennessy. *Computer architecture: a quantitative approach, 3rd ed.* Morgan Kauffman, 2003.

[15] HeteroMPI. http://hcl.ucd.ie/profile/HeteroMPI.

[16] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability, 1st edition*. McGraw-Hill, 1992.

[17] LAM-MPI. http://www.lam-mpi.org/.

[18] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*. Univ. of Tennessee, Knoxville, Tennessee, 1995.

[19] Gordon Moore. *Cramming more components onto integrated circuits*. Electronics Magazine, 1965.

[20] MPICH. http://www-unix.mcs.anl.gov/mpi/mpich1/.

[21] MPICH-Madeleine. http://runtime.bordeaux.inria.fr/mpi/.

[22] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrel. *Pthreads programming: A Posix Standard for Better Multiprocessing*. O'Reilly, 1996.

[23] NVIDIA. http://developer.nvidia.com/object/gpu_programming_guide.html.

[24] OpenMP. http://www.openmp.org/blog/.

[25] OpenMPI. http://www.open-mpi.org/.

[26] J. Ortega, M. Anguita, and A. Prieto. *Arquitectura de Computadores*. Thomson, 2004.

[27] Robert R. Schaller. Moore's law: past, present, and future. *IEEE Spectrum*, 34:52–59, 1997.

[28] SETI@home. http://setiathome.ssl.berkeley.edu/.

[29] Marc Snir and William Gropp. *MPI. The Complete Reference. 2nd edition*. The MIT Press, 1998.

[30] TOP500. http://www.top500.org/.

[31] OpenMP v3.0. http://www.openmp.org/mp-documents/spec30.pdf, May 2008.