Introducción a las arquitecturas paralelas

Daniel Jiménez-González

PID_00184814

UOC
Universitat Oberta
de Catalunya

Los textos e imágenes publicados en esta obra están sujetos excepto que se indique lo contrario a una licencia de Reconocimiento-NoComercial-SinObraDerivada (BY-NC-ND) v.3.0 España de Creative Commons. Podéis copiarlos, distribuirlos y transmitirlos públicamente siempre que citéis el autor y la fuente (FUOC. Fundación para la Universitat Oberta de Catalunya), no hagáis de ellos un uso comercial y ni obra derivada. La licencia completa se puede consultar en http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es

Índice

Introducción 5

Objectivos 7

1. Paralelismo en uniprocesadores 9
- 1.1. Paralelismo a nivel de instrucción 9
  - 1.1.1. Procesador segmentado 9
  - 1.1.2. Procesador superescalar 11
  - 1.1.3. Procesador *Very Long Instruction Word* (VLIW) 13
- 1.2. Paralelismo a nivel de *threads* 13
  - 1.2.1. *Multithreading* de grano fino 14
  - 1.2.2. *Multithreading* de grano grueso 15
  - 1.2.3. *Simultaneous Multithreading* 16
- 1.3. Paralelismo a nivel de datos 17
  - 1.3.1. Ejemplos de vectorización 18
  - 1.3.2. Código vectorizable sin dependencias 19
- 1.4. Limitaciones del rendimiento de los procesadores 23
  - 1.4.1. Memoria 23
  - 1.4.2. Ley de Moore y consumo de los procesadores 24

2. Taxonomía de Flynn y otras 26
- 2.1. MIMD: Memoria compartida 29
- 2.2. MIMD: Memoria distribuida 30
- 2.3. MIMD: Sistemas híbridos 30
- 2.4. MIMD: *Grids* 31

3. Medidas de rendimiento 32
- 3.1. Paralelismo potencial 32
- 3.2. *Speedup* y eficiencia 35
- 3.3. Ley de Amdahl 36
- 3.4. Escalabilidad 40
- 3.5. Modelo de tiempo de ejecución 40
- 3.6. Casos de estudio 43
  - 3.6.1. Ejemplo sin *Blocking*: *Edge-Detection* 43
  - 3.6.2. Ejemplo con *Blocking*: *Stencil* 45

4. Principios de programación paralela 48
- 4.1. Concurrencia en los algoritmos 48
  - 4.1.1. Buenas prácticas 50
  - 4.1.2. Orden y sincronización de tareas 50
  - 4.1.3. Compartición de datos entre tareas 51

4.2. Problemas que aparecen en la concurrencia 51
4.2.1. *Race Condition* 51
4.2.2. *Starvation* 52
4.2.3. *Deadlock* 52
4.2.4. *Livelock* 52
4.3. Estructura de los algoritmos 53
4.3.1. Patrón *Task Parallelism* 53
4.3.2. *Patrón Divide & Conquer* 54
4.3.3. Patrón *Geometric Decomposition* 54
4.3.4. Patrón *Recursive Data* 54
4.3.5. Patrón *Pipeline* 54
4.3.6. Patrón *Event-based Coordination* 55
4.4. Estructuras de soporte 55
4.4.1. SPMD 55
4.4.2. *Master/Workers* 56
4.4.3. *Loop Parallelism* 56
4.4.4. *Fork/Join* 56

5. Modelos de programación paralela 57
5.1. Correspondencia entre modelos y patrones 57
5.2. MPI 57
5.2.1. Un programa simple 58
5.2.2. Comunicación punto a punto 60
5.2.3. Comunicación colectiva 64
5.3. OpenMP 70
5.3.1. Un programa simple 70
5.3.2. Sincronización y *locks* 71
5.3.3. Compartición de trabajo en bucles 73
5.3.4. Tareas en OpenMP 74
5.4. OmpSs 76
5.5. Caso de estudio 78
5.5.1. Código secuencial 78
5.5.2. OpenMP 79
5.5.3. MPI 82
5.5.4. Híbrido: MPI + OpenMP 86

Resumen 88

Ejercicios de autoevaluación 89

Bibliografía 91

Introducción

Son muchas las áreas de investigación, y sobre todo, programas de aplicación real donde la capacidad de cómputo de un único procesador no es suficiente. Un ejemplo lo encontramos en el campo de la bioinformática y, en particular, en la rama de genómica, donde los secuenciadores de genomas son capaces de producir millones de secuencias en un día. Estos millones de secuencias se deben procesar y evaluar con tal de formar el genoma del ser vivo que se haya analizado. Este procesar y evaluar requieren un tiempo de cómputo y capacidades de memoria muy grandes, sólo al alcance de computadores con más de un core y una capacidad de almancenamiento significativa. Otros ámbitos donde nos podemos encontrar con estas necesidades de cómputo son en el diseño de fármacos, estudios del cosmo, detección de petróleo, simulaciones de aeronaves, etc.

Durante años, el aumento de la frecuencia de los procesadores había sido la vía de aumentar el rendimiento de las aplicaciones, de una forma transparente al programador. Sin embargo, aunque la frecuencia de los uniprocesadores ha ido aumentando, se ha observado que este aumento no se puede mantener indefinidamente. No podemos hacer mover los electrones y los protones a mayor velocidad que la de la luz. Por otra parte, este aumento de frecuencia conlleva un problema de disipación de calor, haciendo que los uniprocesadores deban incorporar mecanismos de refrigeración de última generación. Finalmente, la mejora de la tecnología y por consiguiente, la consecuente reducción del tamaño del transistor, nos permiten incorporar más componentes al uniprocesador sin aumentar su área (más memoria, más *pipelines*, etc.). Sin embargo, esta disminución del tamaño del transistor para incorporar más mejoras y capacidades al uniprocesador tampoco es una solución que se pueda llevar al infinito (principio de incerteza de Heisenberg). Estos problemas tecnológicos contribuyeron a que, para poder tratar con problemas tan grandes evitando las limitaciones tecnológicas, los arquitectos de computadores comenzaran a centrar sus esfuerzos en arquitecturas paralelas.

Principio de incerteza de Heisenberg

"The position and momentum of a particle cannot be simultaneously measured with arbitrarily high precision. There is a minimum for the product of the uncertainties of these two measurements. There is likewise a minimum for the product of the uncertainties of the energy and time."

El soporte hardware para procesar en paralelo se puede introducir en varios niveles; desde el más bajo nivel, el uniprocesador, al más alto nivel con las `grids` de multiprocesadores o multicomputadores, donde la red de interconexión es Internet. Estos niveles abarcan el paralelismo a nivel de instrucción (`ILP`), a nivel de datos (`DLP`) y a nivel de `threads` (`TLP`).

Niveles de paralelismo

ILP: *Instruction Level Parallelism.*
DLP: *Data Level Parallelism.*
TLP: *Thread Level Parallelism*

A nivel de un uniprocesador, el paralelismo se incorporó con la ejecución segmentada* y los procesadores superescalares (procesadores con múltiples unidades funcionales de un mismo tipo que se pueden usar a la vez). Estas dos características permitieron que varias instrucciones se pudieran ejecutar al mismo tiempo en un uniprocesador. Con la misma finalidad de ejecutar más de una instrucción a la vez, pero aligerando el hardware necesario, aparecieron algunos procesadores que permitían ejecutar instrucciones muy largas

*** *Pipeline* en inglés**

(*Very Long Instruction Word*). En este caso, el papel del compilador es importante para organizar adecuadamente las instrucciones en el código.

Otra forma en la que se añadió paralelismo a nivel de uniprocesador fue permitiendo procesar más de un dato a la vez con una única instrucción (instrucciones vectoriales o instrucciones *SIMD*), con lo que se empezó a explotar el paralelismo a nivel de datos. Finalmente, se añadió soporte hardware para explotar el paralelismo a nivel de *thread*, pudiendo ejecutar más de un *thread* a la vez en un uniprocesador con un coste pequeño de cambio de contexto.

A partir del año 2000, debido a los problemas de disipación de calor y rendimiento final de los uniprocesadores, se empezaron a explorar nuevas formas de paralelismo dentro de un mismo chip con los procesadores *multicores*. Los procesadores *multicores* permiten aprovechar las mejoras tecnológicas de reducción de tamaño de los transistores, manteniendo o disminuyendo la frecuencia de cada CPU que forma parte del *multicore*. La disminución de frecuencia permite la reducción del consumo energético y como consecuencia, la necesidad de disipación de calor. Por otro lado, al tener más de una CPU se pueden alcanzar mejores rendimientos en las aplicaciones.

En cualquier caso, para ciertas aplicaciones con necesidades de un factor de paralelismo de más de 1000x (1.000 veces más rápidos) respecto a los uniprocesadores, es preciso unir cientos o miles de CPU, que conectadas de alguna forma, puedan trabajar eficientemente. Esto ha llevado a los grandes multiprocesadores y multicomputadores, que son computadores de memoria compartida y distribuida respectivamente, formados por un gran número de procesadores conectados con redes de interconexión muy rápidas. De hecho, este paralelismo, como tal, ya lleva muchos años siendo utilizado en el área de investigación y de aplicaciones numéricas, y no ha sido sólo consecuencia de las limitaciones de los uniprocesadores.

Finalmente, otra posible vía de adquirir mayor paralelismo es uniendo varios sistemas multiprocesadores o multicomputadores a través de la red. En este caso tenemos lo que llamamos *grid*s, que son multicomputadores.

En este módulo analizaremos primero la evolución de los uniprocesadores desde el punto de vista del paralelismo (apartado 1). En el apartado 2 veremos una clasificación de las arquitecturas de computadores. Esta clasificación diferencia entre los distintos niveles de paralelización que se pueden explotar de una aplicación. En el apartado 3 detallaremos cómo medir el rendimiento de las aplicaciones, el paralelismo potencial que podemos alcanzar, y describiremos un modelo sencillo de rendimiento para poder determinar qué parámetros son los mejores para obtener el mejor rendimiento de la estrategia de paralelización escogida. Posteriormente, en el apartado 4 detallaremos los principios de programación paralela, indicando cuáles son las estructuras algorítmicas que normalmente se utilizan para paralelizar algunos patrones comunes de aplicación. Finalmente, en el apartado 5 describiremos tres modelos de programación paralela (OpenMP, MPI, y una extensión del model OpenMP, OmpSs) y veremos un caso de estudio sencillo para practicar OpenMP y MPI.

Lectura recomendada

Sobre el modelo de programación paralela OmpSs, podéis leer: **Alejandro Duran; Eduard Ayguadé; Rosa M. Badia; Jesús Labarta; Luis Martinell; Xavier Martorell; Judit Planas** (2011). “Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures”.*Parallel Processing Letters* (vol. 21, núm. 2, págs. 173-193).

Objectivos

Los objetivos generales de este módulo didáctico son los siguientes:

1. Conocer los diferentes soportes *hardware* para explotar paralelismo en un uniprocesador.
2. Saber realizar programas pequeños que exploten el soporte *hardware* para el paralelismo a nivel de datos de un uniprocesador.
3. Conocer la taxonomía de Flynn.
4. Conocer las diferentes estrategias de paralelización, y saber realizar programas paralelos basados en modelos de programación basados, a su vez, en variables compartidas y paso de mensajes.
5. Saber utilizar las diferentes métricas para medir el rendimiento de programas paralelos.

En particular, los objetivos específicos serán que el estudiante sea capaz de:

1. Enumerar y describir brevemente los diferentes niveles de paralelismo que podemos explotar en los uniprocesadores.
2. Analizar si un código se puede o no vectorizar (explotar el paralelismo a nivel de datos con instrucciones vectoriales o SIMD).
3. Vectorizar un código vectorizable.
4. Definir la taxonomía de Flynn, y detallar las diferentes subcategorías de las arquitecturas MIMD.
5. Detectar posibles problemas de concurrencia en un programa que se ha pensado paralelizar.
6. Determinar cuál es la mejor forma de distribuir un programa en tareas para aprovechar una máquina concreta.
7. Enumerar los diferentes patrones de algoritmos paralelos, y estructuras de soporte para la paralelización de códigos.
8. Analizar una estrategia de paralelización de un código mediante las diferentes métricas de rendimiento.
9. Modelar una estrategia de paralelización basándose en un modelo básico de comunicación.

10. Analizar diferentes estrategias de paralelización con tal de minimizar T_P, y por consiguiente, se maximice el *speed up* (aceleración del programa paralelo con respecto al secuencial).

11. Diseñar una estrategia de paralelización usando la técnica de *blocking*.

12. Programar con los modelos de programación de memoria distribuida (MPI) y memoria compartida (OpenMP) un programa de complejidad media (100 líneas).

13. Enumerar las diferencias entre comunicaciones punto a punto *blocked* y no *blocked*.

14. Utilizar las colectivas de MPI de forma adecuada.

15. Programar aplicaciones paralelas con tareas de OpenMP para desarrollar una solución paralela con el patrón *Divide & Conquer*.

16. Programar aplicaciones paralelas con `OpenMP` para desarrollar una solución paralela con el patrón *Loop Parallelism*.

17. Realizar un programa híbrido utilizando MPI y OpenMP.

18. Describir tareas con la extensión de OpenMP, OmpSs.

1. Paralelismo en uniprocesadores

En este apartado repasaremos los mecanimos hardware que se han incorporado en los uniprocesadores con tal de explotar el paralelismo a nivel de instrucción, a nivel de `thread` y a nivel de datos. Algunos de estos mecanismos hardware permiten explotar el paralelismo sin necesidad de ningún esfuerzo por parte del programador ni del compilador. Otros, en cambio, precisan del programador o del compilador para poder explotarlos. Así, por ejemplo, la segmentación y los procesadores superescalares permiten, de forma transparente, explotar el paralelismo a nivel de instrucción. En cambio, en el caso de los procesadores *Very Long Instruction Word* (VLIW), que también tienen como objetivo explotar el paralelismo a nivel de instrucción, precisan del compilador para explotar adecuadamente este paralelismo. En el caso de querer explotar el paralelismo a nivel de datos y/o a nivel de `threads`, también es necesario que el compilador o el programador generen un binario o desarrollen el programa, respectivamente.

Los procesadores VLIW

Estos procesadores suponen que las instrucciones ya están en el orden adecuado, por los que necesitan compiladores específicos que conozcan la arquitectura del procesador.

1.1. Paralelismo a nivel de instrucción

1.1.1. Procesador segmentado

La segmentación en un procesador consiste en la división de la ejecución de una instrucción en varias etapas, donde cada etapa normalmente se realiza en un ciclo de CPU. El número de etapas, y los nombres que reciben pueden variar de un procesador a otro.

Ejecución segmentada

El origen de la ejecución segmentada se piensa que viene o bien del proyecto ILLIAC II o del proyecto IBM Strech.

Con la segmentación de la ejecución de las instrucciones se consigue aumentar el ratio de ejecución, es decir, el número medio de instrucciones que acaban por ciclo (*IPC-Instructions per cycle*). El aumento de ratio es gracias a que se solapan las etapas de ejecución de más de una instrucción*. De esta forma, podemos estar ejecutando en paralelo tantas instrucciones como número de etapas tenga la ejecución de las instrucciones, y depués de rellenar todas las etapas con tantas instrucciones como etapas haya en la segmentación, conseguiremos finalizar una instrucción por ciclo.

*** El paralelismo a nivel de instrucciones que se consigue con la segmentación coincide con el número de etapas.**

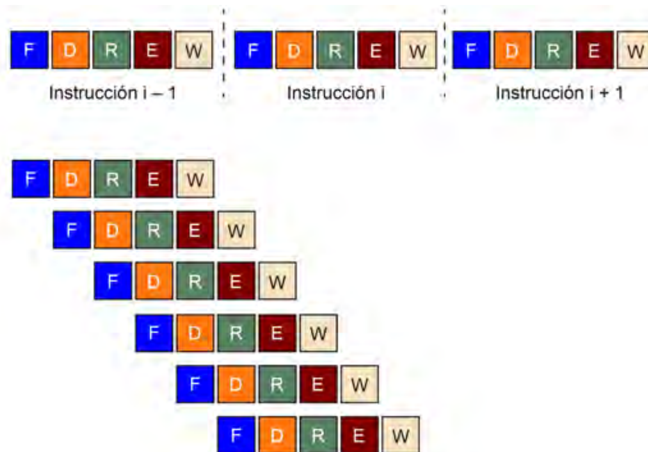
La figura 1 muestra un ejemplo de la segmentación de 5 etapas: `Instruction Fetch` o ir a buscar una instrucción a la memoria, `Instruction Decode` o decodificar la instrucción y los operandos, `Operand fetch unit` donde se van a buscar los operandos, `Instruction Execution unit` para la ejecución de las operaciones, y `Writeback` donde se escriben los datos en los registros o memoria. En la parte superior de la figura observamos la ejecución en un procesador donde no hay segmentación, y por consiguiente, cada instrucción debe esperar a que la instrucción anterior se acabe de ejecutar. En cambio, segmentando la ejecución, tal y como se muestra en la parte inferior de la figura, podemos observar que varias instrucciones se pueden estar ejecutando en paralelo; en concreto cinco

Etapas de un procesador

Las etapas clásicas de un procesador `RISC` (`Reduced instruction set computing`) varían un poco con respecto a las explicadas en este módulo, y son: `instruction fetch`, `decode`, `execute`, `mem` o escritura en memoria, y `writeback`. Algunos ejemplos de procesadores que tenían este tipo de segmentación son: MIPS, SPARC, Motorola 88000, y DLX.

instrucciones. También observamos que tras tener lleno el *pipeline* del procesador, acabará una instrucción a cada ciclo.

Figura 1. Ejecución en un procesador no segmentado (arriba) y un procesador segmentado (abajo).



Tal y como hemos comentado, el número de etapas puede variar mucho de un procesador a otro. Por ejemplo, el procesador Pentium 4 tenía 20 etapas, en cambio el procesador Prescott tiene 31 etapas, y el procesador Cell Broadband Engine (BE) tiene 17 etapas en los SPEs. La etapa más lenta determinará la frecuencia del procesador. Por consiguiente, una estrategia que se ha seguido para aumentar la velocidad de los procesadores es la de incrementar el número de etapas (la profundidad del *pipeline*), reduciendo el tamaño de éstas. Sin embargo, hay algunos inconvenientes en tener segmentaciones tan profundas.

Cell Broadband Engine

Procesador de Sony, Toshiba e IBM, formado por un PowerPC y 8 Synergistic Processing Element (SPE), y que lo podemos encontrar en las PlayStation 3.

¿Qué haremos cuando nos encontremos con un salto? ¿Y cuál es el destino del salto?

Hay dos opciones: tomarlos o no tomarlos. Normalmente, para no parar la ejecución y por consiguiente, aprovechar al máximo el paralelismo de la segmentación, se predice el sentido del salto, y el destino de éste. Esto se realiza con predictores de saltos, implementados en hardware, que normalmente tienen una tasa de acierto en la predicción del salto elevada (mayor que 90 %). Pero, en caso de predicción incorrecta de salto, la penalización de deshacer todas las instrucciones que no se deberían haber iniciado, es proporcional al número de etapas en la segmentación de la instrucción.

Esta penalización limita el número de etapas que podemos poner en la segmentación. Es por ello por lo que una forma de conseguir aumentar el ratio de instrucciones de ejecución, y de esta forma, acelerar los programas, es conseguir ejecutar más instrucciones por ciclo. Para conseguirlo, se añadió hardware para poder lanzar a ejecutar más de una instrucción por ciclo. Este tipo de procesador se llama *Superscalar*. Estos procesadores tienen normalmente unidades funcionales duplicadas, que permiten ejecutar más de una instrucción del mismo tipo en el mismo ciclo.

Procesadores superescalares

Los procesadores superescalares permiten lanzar a ejecutar más de una instrucción en el mismo ciclo.

1.1.2. Procesador superescalar

El término de procesador *Superscalar* apareció por primera vez en 1987, en el sentido de que el procesador tenía unidades funcionales duplicadas, y por consiguiente, podía lanzar más de una instrucción a la vez. Cuarenta años antes, el computador CDC 6600 ya tenía 10 unidades funcionales en las que podía estar ejecutando varias instrucciones. Sin embargo, este procesador no podía realizar el lanzamiento de más de una instrucción a la vez, y por consiguiente, no se le considera un procesador superescalar.

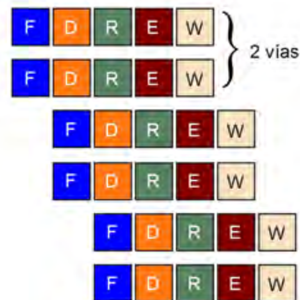
Lectura complementaria

Sobre los procesadores superescalares podéis leer: **T. Agerwala y D.A. Wood** (1987). *High Performance Reduced Instruction Set Processors*. IBM T. J. Watson Research Center Technical Report RC12434.

La figura 2 muestra un ejemplo de un procesador que tiene la habilidad de empezar a ejecutar (lanzar) dos intrucciones* a cada ciclo. Con este procesador superescalar de dos vías o `pipelines` se puede llegar a tener un ratio de ejecución de hasta 2 instrucciones por ciclo. Sin embargo, hay una serie de problemas que pueden limitar este ratio: dependencia de verdad**, dependencia de recursos, y dependencia de saltos o de procedimientos/funciones.

* ***Two-way*** **en inglés**
** ***True data dependency*** **en inglés**

Figura 2. Ejecución en un procesador superscalar con dos vías.



La dependencia de verdad es aquella que se produce cuando una instrucción necesita un operando que se está calculando por una instrucción previa en el programa, y que está siendo ejecutada. Estas dependencias se deben resolver antes de realizar el lanzamiento de las dos instrucciones en paralelo para ejecutarse. Esto tiene dos implicaciones:

1) Tiene que haber hardware dedicado para la resolución de esta dependencia en tiempo de ejecución.

2) El grado de paralelismo conseguido está limitado por la forma en que se codifica el programa.

La primera implicación conlleva introducir una complejidad hardware en el procesador para poder tratarlo. La segunda, un compilador que sea consciente del hardware puede mejorar el rendimiento reordenando las instrucciones.

La dependencia de recursos proviene del hecho de que no hay recursos infinitos. El número de unidades funcionales de un tipo puede limitar el número de instrucciones a poderse lanzar, aún teniendo suficientes *pipelines*. Por ejemplo, si tenemos un procesador superes-

calar con dos *pipelines*, pero una única unidad funcional de coma flotante, no podremos lanzar más de una instrucción que vaya a la unidad funcional.

La tercera dependencia es la dependencia por salto, o bien de procedimientos, que es la misma que se comentó para la segmentación de un procesador. El destino del salto sólo es conocido en el momento de la ejecución, y por consiguiente, seguir ejecutando instrucciones sin saber cuál es el resultado del salto puede llevar a errores. Sin embargo, como hemos visto, se usa la especulación con predicción para determinar la decisión del salto y el destino. En caso de error, se deshace todo lo que se había hecho. La frecuencia de instrucciones de saltos en el código es de aproximadamente uno de cada cinco o seis instrucciones. Esto lo hace un factor importante a considerar.

Limitaciones por dependencias

El paralelismo a nivel de instrucción en los procesadores superescalares está limitado por las dependencias de verdad, de recursos y de salto.

Para sobreponerse a todas estas limitaciones por dependencias, y aumentar el ratio de las instrucciones ejecutadas por ciclo, una de las mejoras en el diseño de los procesadores fue la de reordenar instrucciones en ejecución, para poder ejecutar en paralelo todas aquellas que no tuvieran dependencias. Notad que reordenan las instrucciones para ejecutarlas en fuera de orden, pero, en el momento de tener que escribir en registros o memoria se hace en orden. Estos procesadores se llaman en fuera de orden*. Hasta el momento los procesadores que habíamos estado mirando eran en orden**.

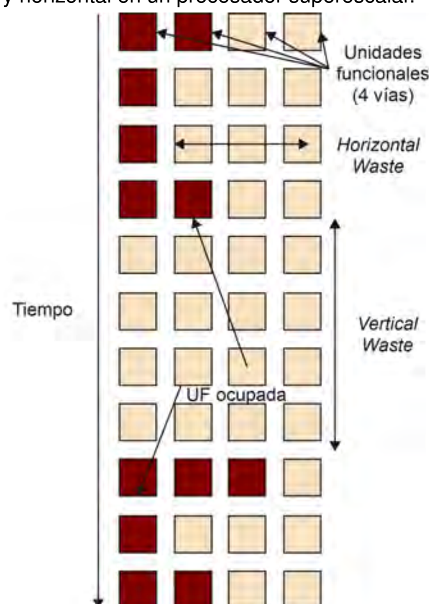
* `Out-of-order` en inglés
** `In-order` en inglés

A pesar de estas mejoras, nos podemos encontrar que el *pipeline* del procesador queda vacío durantes varios ciclos, o que algunas de nuestras unidades funcionales no se están usando durante un ciclo concreto. Estos casos se conocen, en inglés, como *vertical waste* y *horizontal waste* respectivamente. La figura 3 muestra las cuatro vías (una por columna) de ejecución en un procesador superescalar. Cada fila representa los ciclos de ejecución en dicho procesador. Cada recuadro por fila representa una unidad funcional, y que esté coloreada significa que está ocupada. Como podemos observar, se está desaprovechando del procesador tanto vertical como horizontalmente (cuadrados no coloreados). De esta forma, que el procesador superescalar no se está aprovechando al máximo.

Procesador fuera de orden

Los procesadores fuera de orden son aquellos procesadores que reordenan la ejecución de las instrucciones con tal de evitar dependencias. El primer microprocesador con ejecución fuera de orden fue el POWER1, de IBM.

Figura 3. `Waste` vertical y horizontal en un procesador superescalar.



El aprovechar mejor o peor el procesador dependerá, en parte, del hardware que tengamos dedicado a la reordenación de las instrucciones en tiempo de ejecución. Este hardware tiene un coste, que es función cuadrática del número de *pipelines* del procesador, y por consiguiente puede tener un coste elevado. Para el caso típico de un *four-way* tiene un coste el 5 % al 10 % del hardware del procesador.

Una opción alternativa es dejar al compilador el trabajo de buscar las instrucciones que se pueden ejecutar a la vez en el procesador. El compilador organizará el binario de tal forma que aquellas instrucciones que pueden ejecutarse a la vez estarán consecutivas en el programa. Así, el procesador podrá leerlas de una vez, como si fuera una instrucción larga, formada por varias instrucciones, y mandarlas a ejecutar sin tener que mirar si hay dependencia o no. Estos procesadores se conocen como *Very Long Instruction Word* o VLIW.

1.1.3. Procesador *Very Long Instruction Word* (VLIW)

El concepto de `VLIW` fue usado por primera vez con la serie TRACE creada por Multiflow y posteriormente como una variante de la arquitectura Intel IA64. Estos procesadores tienen la ventaja de ahorrarse el hardware dedicado a la reordenación en tiempo de ejecución de las instruccciones, dependiendo mucho del tipo de optimizaciones que haga el compilador. Normalmente también disponen de instrucciones dedicadas para controlar el flujo de ejecución del programa. Por otro lado, no disponen de predicción de saltos basado en la historia de la ejecución del programa, y les es muy difícil detectar situaciones de fallos de acceso en la caché, y por consiguiente se debe parar la ejecución del programa. Con lo que la ordenación que hizo el compilador fuerza a que la ejecución de muchas instruccciones se pare.

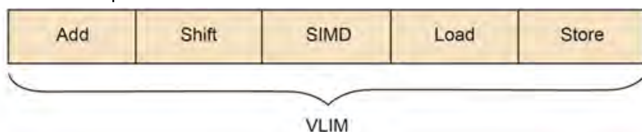
Procesadores VLIW

Los procesadores VLIW no tienen soporte hardware para detectar paralelismo entre las instrucciones en tiempo de ejecución, pero dependen totalmente de la compilación realizada.

La figura 4 muestra el esquema de una instrucción larga formada por una 5 instrucciones: una suma, un desplazamiento binario, una instrucción `SIMD`, una operación de lectura y una de escritura. El código binario generado por el compilador específico debe estar organizado en palabras de 5 instrucciones que se puedan ejecutar en paralelo. En caso de no poder obtener 5 instrucciones, se rellenarán con `NOPs`*.

*** *No operation* en inglés**

Figura 4. VLIW formada por 5 instrucciones.



1.2. Paralelismo a nivel de *threads*

Hasta el momento hemos visto diversas formas de explotar el paralelismo a nivel de instrucción (ILP). Este paralelismo, sin embargo, se pierde en el momento en que nos encontramos con fallos de acceso a los diferentes niveles de memoria caché. Esto es debido a que el procesador debe parar la ejecución de nuevas instrucciones hasta que el dato (línea de

ILP

El paralelismo a nivel de `thread` ayuda a explotar mejor el paralelismo a nivel de instrucción ocultando los fallos de acceso a memoria.

caché) se reciba. Una solución para conseguir que el procesador continúe ejecutando instrucciones es que pueda ejecutar otro programa o `thread`, enmascarando esta situación de bloqueo. Esto es lo que se llama, en inglés, *on-chip multithreading*.

Hay varias aproximaciones al *on-chip multithreading*: *multithreading* de grano fino, *multithreading* de grano grueso, y *simultaneous multithreading*, que veremos a continuación.

1.2.1. *Multithreading* de grano fino

Este tipo de paralelismo a nivel de *threads* intenta ocultar los bloqueos del procesador realizando una ejecución en *round robin* de las instrucciones de *threads* diferentes, en ciclos consecutivos.

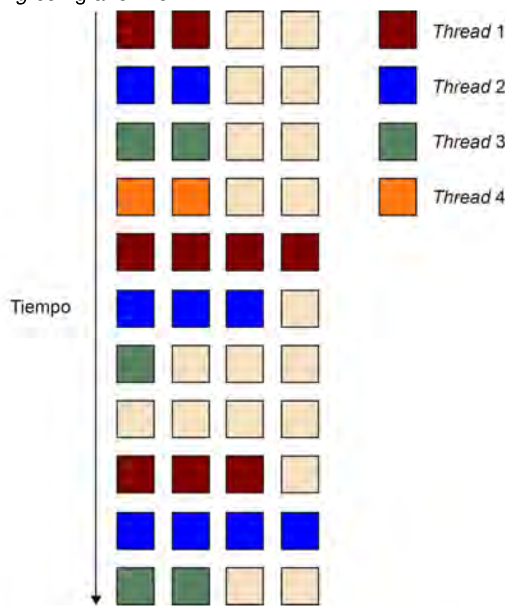
Round Robin

Es la política de asignación de ciclos del procesador en el que se le dan un número fijo de ciclos a un *thread* detrás de otro.

De esta forma, si tenemos tantos *threads* ejecutándose como ciclos de bloqueo que precisa, por ejemplo, un fallo de acceso a memoria, podremos reducir la probabilidad de que el procesador se quede sin hacer nada en esos ciclos.

La figura 5 muestra como 4 *threads*, con colores diferentes, se van ejecutando en las diferentes unidades funcionales del procesador superescalar. A cada ciclo de ejecución, se cambia de *thread*.

Figura 5. *Multithreading* con grano fino.



Para poder realizar esta ejecución de varios *threads*, independientes entre ellos, cada uno de ellos necesita un conjunto de registros asociados. Así, cada instrucción tiene asociada información para saber qué banco de registros usar en cada momento. Esto lleva a que el número de *threads* que se pueden ejecutar a la vez (en *round robin*) dependerá del hardware de que dispongamos.

***threads* independientes**

Si los *threads* no fueran independientes deberían compartir variables en memoria y podrían necesitar algún método de sincronización para poder acceder si algunos de ellos tuvieran que actualizar la memoria.

Otro motivo de bloqueo del procesador es la decisión que deben tomar en cuanto a los saltos que se encuentren. Eso dificulta saber si se hace o no el salto, y hacia dónde. Una solu-

ción es tener tantos *threads* que se puedan ejecutar en *round robin* como etapas en el *pipeline*, con lo cual sabríamos que siempre se estaría ejecutando un *thread*. Pero, tal y como hemos comentado, esto supondría, a nivel de soporte hardware, muchos bancos de registros.

Bancos de registros

El número de bancos de registros limita el número de *threads* a los que se le puede dar soporte hardware en grano fino.

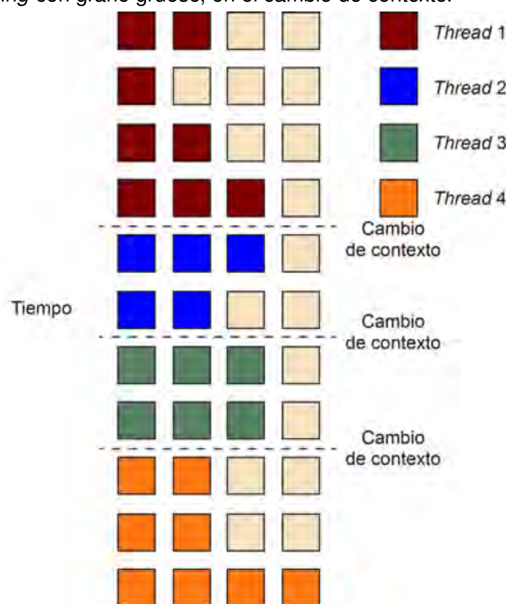
1.2.2. *Multithreading* de grano grueso

Esta aproximación consiste en poder ejecutar más de una instrucción de un *thread* en ciclos consecutivos. La figura 6 muestra una ejecución de cuatro *threads* en un sistema *multithreading* con grano grueso.

***Multithreading* de grano grueso**

En un procesador con soporte *Multithreading* de grano grueso los *threads* pueden ejecutarse más de un ciclo consecutivo, reduciendo así la necesidad de muchos bancos de registros.

Figura 6. *Multithreading* con grano grueso, en el cambio de contexto.



En este caso, lo que normalmente se hace es que un *thread* continúa la ejecución de instrucciones hasta que se produce un bloqueo debido a un salto, un conflicto de datos, etc. Con esta estrategia de dejar ejecutar más de un ciclo a un *thread* no son necesarios tantos *threads* activos como pasaba con el *multithreading* de grano fino con tal de aprovechar al máximo el procesador. En contrapartida, como siempre se espera a que haya un bloqueo para cambiar de *thread*, los ciclos que se necesiten para darse cuenta del bloqueo y cambiar de *thread* se perderán. En cualquier caso, si tenemos un número suficiente de *threads* activos, podremos conseguir mejor rendimiento que con el grano fino.

Otra posibilidad para reducir aún más el número de ciclos en los que el procesador está bloqueado, es que se realice el cambio de *thread* cada vez que una instrucción pueda provocar un bloqueo, y no cuando éste se produzca.

Con este tipo de *multithreading* de grano grueso puede también tener más sentido vaciar el *pipeline* cada vez que cambiemos de *thread*. De esta forma no tenemos que tener la información sobre qué banco de registros se tiene que usar a lo largo de toda la ejecución de una instrucción.

1.2.3. *Simultaneous Multithreading*

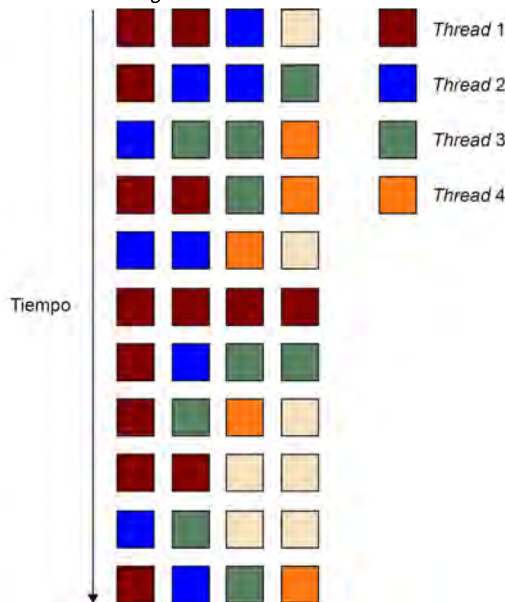
Esta última aproximación intenta reducir el *horizontal waste* en los procesadores superescalares con *multithreading*. Se puede considerar como un refinamiento del grano grueso, de tal forma que si en un ciclo del procesador una instrucción de un *thread* se bloquea, una instrucción de otro *thread* se puede usar para mantener el procesador y todas sus unidades funcionales ocupadas. También es posible que una instrucción de un *thread* pudiera quedar bloqueada porque hay un número limitado de unidades funcionales de un tipo. En este caso también se podría coger una instrucción de otro *thread*.

Procesadores con *Simultaneous Multithreading*

Los procesadores con *Simultaneous Multithreading* reducen el desaprovechamiento horizontal de las unidades funcionales, permitiendo que dos *threads* diferentes puedan ejecutarse en el mismo ciclo de procesador.

La figura 7 muestra un ejemplo de ejecución de 4 *threads* en un sistema *Simultaneous Multithreading*. En este caso, en cada ciclo de ejecución puede haber instrucciones de diferentes *threads*.

Figura 7. *Simultaneous Multithreading*.



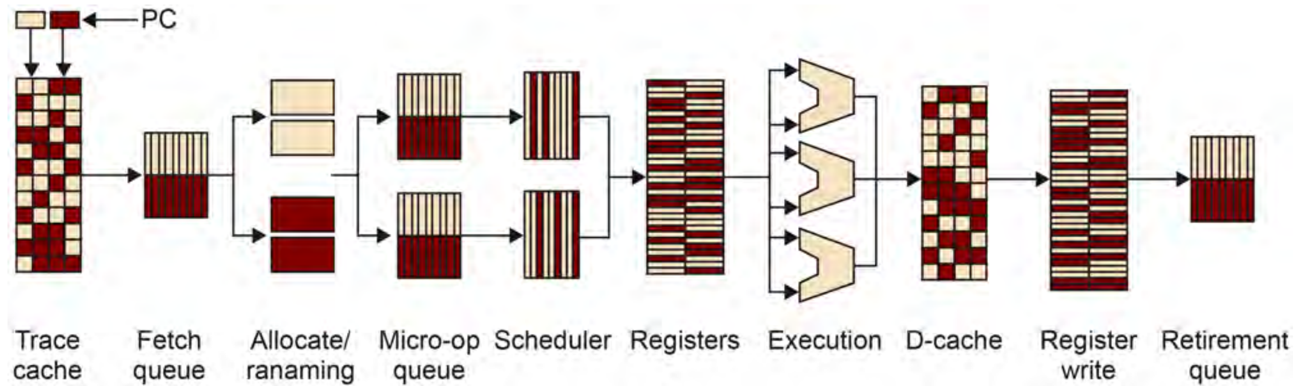
El primer procesador que incorporó el *Simultaneous Multithreading*, conocido como *hyperthreading*, fue el Pentium 4. Este tipo de soporte ha tenido continuidad en otros procesadores, como por ejemplo el Intel Corei7, un *multicore* que incorpora en cada uno de sus *cores* el *Simultaneous Multithreading*.

Multicore

Se trata de un chip con más de un *core* dentro de él.

Para el sistema operativo, un procesador con *Simultaneous Multithreading* es como un procesador con dos cores, que comparten caché y memoria. En cambio, en hardware, se deben contemplar qué recursos se comparten y cómo se deben gestionar. Intel contemplaba cuatro estrategias diferentes. Para comentarlas nos basaremos en la estructura básica del *pipeline* de estos procesadores, que se muestra en la figura 8.

Figura 8: Estructura del *pipeline* de un procesador Pentium 4 con *Hyperthreading*.



Estas estrategias son:

1) Duplicación de recursos: el contador de programa, así como la tabla de mapeo de los registros (*eax*, *ebx*, etc.) y controlador de interrupción, tienen que ser duplicados.

2) Particionado de recursos compartidos: particionar permite distribuir los recursos entre los *threads*, de tal forma que unos no interfieren en los otros. Lo cual permite evitar overheads de control, pero también, que algunos de estos recursos queden sin ser utilizados en algún momento. Un ejemplo de recurso particionado es la cola de instrucciones que se van a lanzar a través de dos *pipelines* separados. Uno para cada `thread`.

3) Compartición total de los recursos compartidos: en este caso se intenta solucionar la desventaja de tener uno de los recursos particionados poco aprovechado. Por ejemplo, un *thread* lento de ejecución podría llenar su parte de cola de instrucciones para ejecutar, haciendo que otro *thread* más rápido no pueda aprovechar que puede ejecutarse más rápido, para insertar más instrucciones en la cola. En el caso de la figura, las etapas de `renaming` son totalmente compartidas.

4) Compartición de compromiso*: en este caso no hay una partición fija, sino que se van pidiendo recursos y adquiriéndolos de forma dinámica, hasta un cierto máximo. En la figura, el scheduler es dinámicamente compartido, con un *threshold* máximo.

***** ***Threshold*** **en inglés**

1.3. Paralelismo a nivel de datos

El paralelismo a nivel de datos se refiere básicamente a la posibilidad de operar sobre dos o más datos con una única instrucción; en inglés se refieren a instrucciones *Single Instruction Multiple Data* (SIMD).

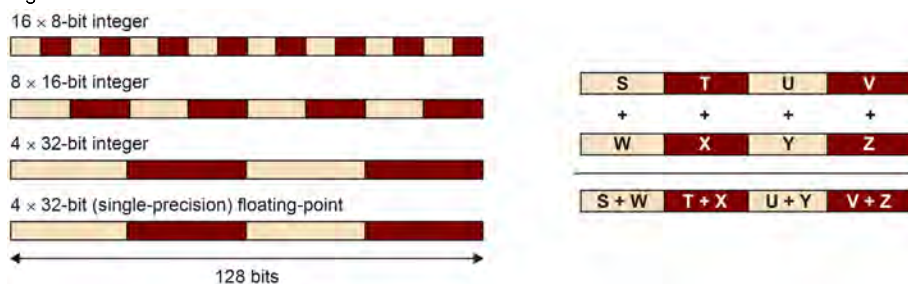
SIMD

Un uniprocesador con instrucciones SIMD es capaz de realizar una misma operación, en paralelo, con más de un dato a la vez.

El soporte hardware necesario para poder ejecutar instrucciones SIMD incluye tener registros más grandes, buses hacia memoria que permitan el acceso a datos del tamaño de los registros, y unidades funcionales que soporten operar con más de un dato a la vez. Además, dependiendo de la semántica de la instrucción se podrá operar con mayor o menor número de operandos.

El tamaño del registro, de 128 y 256 bits, se reparte equitativamente entre los elementos que indique la semántica de la instrucción. La figura 9 muestra la típica distribución de los bits del registro según la semántica de la instrucción, para un registro de 128 bits. Las operaciones que se aplican a cada elemento sólo afectan a esos elementos, a no ser que la semántica de la instrucción permita que se opere entre dos elementos consecutivos de un mismo registro. La figura 9 muestra, a la derecha, una operación de sumar sobre dos registros vectoriales, cada uno de los cuales con 4 elementos de tipo entero de 32 bits.

Figura 9. Instrucciones vectoriales.



Para explotar este tipo de paralelismo se necesita que el compilador y/o el programador utilicen las instrucciones SIMD. El compilador, en ocasiones, es capaz de detectar patrones de códigos en los que se pueden utilizar estas instrucciones. Sin embargo, la mayoría de las veces son los programadores los que deben hacer un análisis de las dependencias existentes en el programa, y en función de ello, ver si se pueden usar estas instrucciones.

Web complementaria

El compilador `gcc` tiene una página dedicada (http://gcc.gnu.org/projects/tree-ssa/vectorization.html) que indica qué mejoras ha hecho en la utilización de instrucciones SIMD de forma automática.

Desde que aparecieron los primeros procesadores con instrucciones SIMD en 1997, la evolución ha sido progresiva, y en general, los procesadores han ido incorporando unidades funcionales de tipo SIMD. Las primeras en aparecer fueron las conocidas como `MMX` (Intel). Estas instrucciones solo operaban con enteros de 32 bits, no soportando instrucciones de tipo *float*. Los registros que se usaban eran los del banco de registros reales, por lo que no se permitía mezclar operaciones de coma flotante y `MMX`.

Las siguientes fueron las 3D-Now de AMD, que sí que soportaban los *floats*. A partir de allá empezaron a aparecer las extensiones SSE, con la incorporación de *floats*, ampliación del número de registros y número de bits hasta 128-bits. Los procesadores Sandy Bridge, aparecidos en el 2011, disponen de las extensiones AVX, que pueden operar hasta 256 bits.

La tabla 1 muestra la relación de año, empresa y caracteríticas más importantes de la aparición de las instrucciones SIMD en los procesadores desde 1997 hasta 2011, año de esta publicación.

1.3.1. Ejemplos de vectorización

En este apartado analizaremos algunos códigos con el objetivo de saber determinar si se puede o no explotar el paralelismo a nivel de datos y, por consiguiente, si se pueden o no utilizar las instrucciones SIMD (vectorización de un código).

Lectura complementaria

Manual de Intel de Optimizaciones para IA32, capítulos 4, 5 y 6.

Tabla 1. Relación de instrucciones SIMD de Intel y AMD, y sus características principales.

Compañía	SIMD extension	Características
Intel (1997)	MMX,MMX2	64-bit, 8 vectores, Enteros, no mezcla MMX-Floats
AMD (1998)	3DNow	64-bit, 8 vectores, FP y enteros, mezcla MMX-Float
Intel (1999)	SSE/SSE1	128-bit, Banco de registros específico, FP y enteros
Intel (2001)	SSE2	Se evitan totalmente los registros MMX
Intel (2004)	SSE3	instrucciones para trabajar horizontalmente con los registros
Intel (2006)	SSSE3	instrucciones vectoriales más complejas
Intel (2007)	SSE4	instrucciones específicas sobre multimedia
Intel (2011)	AVX (*Advanced Vector Extensions*)	Extensión a registros de 256-bit
Intel (por aparecer)	VPU (*Wide Vector Processing Units*)	Extensión a registros de 512-bit

Primero analizaremos un código que, al no tener dependencias de verdad entre sus instrucciones, se puede vectorizar (código 1.1). Después veremos otro código que no se puede vectorizar (código 1.3) debido a que tiene dependencias de verdad entre sus instrucciones que no lo permiten. Y finalmente veremos dos códigos que se pueden vectorizar: uno que, aun teniendo dependencias de verdad entre sus instrucciones, la distancia entre las instrucciones en ejecución permite que se pueda vectorizar (código 1.4), y otro que se puede vectorizar tras reordenar las instrucciones del código original (código 1.6).

1.3.2. Código vectorizable sin dependencias

El primero de éstos (código 1.1) es un código que no tiene dependencias de verdad (una lectura de un dato después de la escritura de ese dato), y por consiguiente se puede vectorizar.

Dependencia de verdad

True data dependency: Una lectura de un dato después de una escritura sobre el mismo dato.

```
1 char A[16], B[16], C[16];
2
3 for (i=0; i<16; i++)
4     A[i] = B[i] + C[i];
```

Código 1.1: Suma de vectores

El código vectorizado, con instrucciones SSE2 del Intel, se muestra en el código 1.2. Para programar con las SSE2 utilizaremos las funciones intrínsecas (`_mm_load_si128`, `_mm_add_epi8`, `_mm_store_si128`) y los tipos que nos ofrece esta extensión. El tipo de las variables enteras, para que se guarden en registros vectoriales, es `__m128i`. Para ello debemos incluir la cabecera `emmintrin.h`. En esta implementación nos hemos asegurado que los vectores `A`, `B` y `C` estén alineados a 16 bytes. Esto es porque las operaciones de memoria que hemos utilizado (lectura: `_mm_load_si128`, y escritura:`_mm_store_si128`) precisan de esta alineación o por el contrario se producirá un acceso no alineado no permitido. Si no pudiéramos asegurar este alineamiento en la declaración de las variables, se pueden utilizar las funciones intrínsecas `_mm_loadu_si128` para las lecturas no alineadas (*u* de *unaligned*), y `_mm_storeu_si128` para las escrituras no alineadas. Estas operaciones son significativamente más lentas que las operaciones de lectura y escritura normales. La operación `_mm_add_epi8` indica que se va a hacer la suma vectorial (`_add`) de un paquete de elementos enteros (`_epi`) de tamaño 8 bits (`_epi8`).

Funciones intrínsecas

Son funciones que se traducen a una o pocas instrucciones de ensamblador. No hay salto a un código de una función, ni paso de parámetros en la traducción a ensamblador.

Lectura y escritura

Los accesos a memoria en las operaciones de lectura y escritura en las extensiones de Intel deben ser alineadas a 16 bytes. De lo contrario se deben usar operaciones no alineadas de lectura y escritura.

```
1 #include <emmintrin.h>
2 ...
3
4 char A[16] __attribute__ (( __aligned__ (16)));
5 char B[16] __attribute__ (( __aligned__ (16)));
6 char C[16] __attribute__ (( __aligned__ (16)));
7
8 __m128i a, b, c;
9
10 ...
11 a = _mm_load_si128((__m128i*) &A[i]);
12 b = _mm_load_si128((__m128i*) &B[i]);
13 c = _mm_add_epi8(a, b);
14 _mm_store_si128((__m128i*)&C[i], c);
```

Código 1.2: Suma de vectores con instrucciones `SIMD`

Al operar con elementos de 8 bits, podemos guardar hasta 16 elementos de tipo *char* en cada registro vectorial. Esto implica que haciendo la operación `_mm_add_epi8` conseguimos realizar todas las operaciones que había en el bucle, y por consiguiente, el bucle del código no vectorial desaparece.

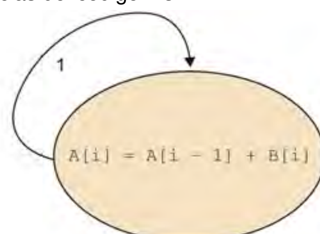
Código no vectorizable con dependencias

El código 1.3 no se puede vectorizar. Este código tiene una dependencia de verdad, que se muestra en la figura 10, que no permite que sea vectorizado. En el grafo de dependencias el sentido de la arista indica el origen y el destino de la dependencia, y la etiqueta de la arista es la distancia en número de iteraciones de la dependencia. En el ejemplo hay una dependencia de verdad de distancia uno con origen y final como la única instrucción que tiene el cuerpo del bucle. El origen es la escritura `A[i]=...` ya que se produce antes en el espacio de iteraciones del bucle, y el destino es la lectura `A[i-1]`. La distancia de la dependencia se calcula como $d = (i - (i - 1)) = 1$. Esta distancia es menor al número de elementos que podemos cargar en un registro vectorial, y por consiguiente, no podemos operar con todos estos elementos a la vez sin romper esta dependencia.

```
1 char A[16], B[16], C[16];
2 for (i=1; i<16; i++)
3     A[i] = A[i-1] + B[i];
```

Código 1.3: Suma de vectores con dependencia de verdad

Figura 10. Grafo de dependencias del código 1.3.



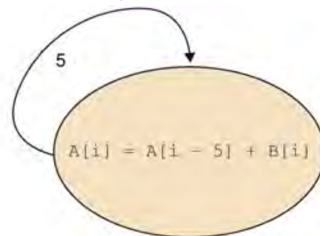
Código vectorizable con dependencias a una distancia suficiente

El código 1.4 es un código que también tiene una dependencia de verdad. Sin embargo, la distancia de la dependencia ($d = (i - (i - 5)) = 5$) es mayor al número de elementos que cabe en un registro vectorial (4 enteros de 32 bits). El grafo dependencias se muestra en la figura 11.

```
1 int A[N], B[N];
2
3 for (i=16; i<N; i++)
4     A[i] = A[i-5] + B[i];
```

Código 1.4: Suma de vectores con una dependencia de verdad con distancia mayor al número de elementos sobre los que se opera.

Figura 11. Grafo de dependencias del código 1.4.



El código vectorizado con instrucciones SSE2 del Intel se muestra en el código 1.5.

Vectorizar código

Un código se puede llegar a vectorizar a pesar de tener dependencias de verdad. Por ejemplo, cuando la distancia de las dependencias de verdad es mayor que el número de elementos que caben en el registro vectorial.

```
1 int A[N] __attribute__ (( __aligned__ (16)));
2 int B[N] __attribute__ (( __aligned__ (16)));
3 ...
4
5 for (i=16; i<N-3; i=i+4) {
6   __m128i a, a5, b;
7   a5 = _mm_loadu_si128((__m128i*) &A[i-5]);
8   b = _mm_load_si128((__m128i*) &B[i]);
9   a = _mm_add_epi32(a5, b);
10  _mm_store_si128((__m128i*)&A[i], a);
11 }
12
13 for (; i<N; i++)
14     A[i] = A[i-5] + B[i];
15
16 ...
```

Código 1.5: Vectorización del código 1.4.

Notad que en el código vectorial 1.5 hemos tenido que usar la lectura no alineada (`_mm_loadu_si128((__m128i) &A[i-5])`) aun habiendo alineado los vectores* `A` y `B` en su primer elemento con `__attribute__ (( __aligned__ (16)))`. Esto es debido a que hemos alineado su primer elemento a 16 bytes, pero $i - 5$ está a una distancia $i\,\%4 + 1$, que no está alineada. También es necesario realizar un epílogo** con tal de hacer las iteraciones que no se hayan efectuado de forma vectorial.

*** Un bucle vectorizado puede necesitar accesos alineados y no alineados a un vector si hay accesos desplazados con respecto a la variable de inducción del bucle.**
**** Normalmente es necesario realizar epílogos de los códigos vectorizados. Esto es debido a que el número de iteraciones no es múltiplo del número de elementos por registro vectorial.**

Código vectorizable con dependencias salvables

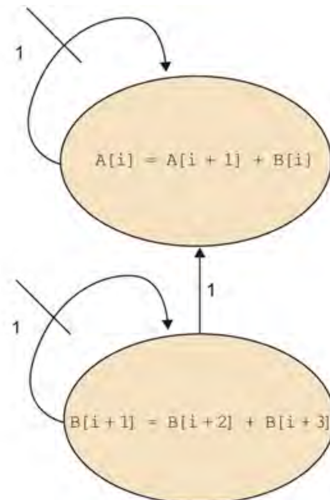
El código 1.6 es un código que tiene una dependencia de verdad y dos antidependencias* (escritura después de lectura). La figura 12 muestra estas dependencias entre las dos instrucciones del bucle. Las antidependencias se indican con una arista con una línea que la corta.

*** Una escritura de un dato después de una lectura sobre el mismo dato.**

```
1 int A[N], B[N];
2
3 for (i=0; i<(N-3); i++) {
4   A[i] = A[i+1] + B[i];
5   B[i+1] = B[i+2] + B[i+3];
6 }
```

Código 1.6: Suma de vectores con dependencias de verdad que se pueden reorganizar.

Figura 12. Grafo de dependencias del código 1.6.



Las antidependencias no afectan a la vectorización. En cambio, la dependencia de verdad (lectura de `B[i]` en la primera instrucción del bucle después de la escritura de `B[i+1]`, en la segunda instrucción) sí que rompe la posible vectorización.

Sin embargo, la dependencia de verdad existente nos la podríamos evitar si cambiamos el orden secuencial de las instrucciones. Por ejemplo, si hacemos primero las 4 iteraciones de la primera instrucción, y posteriormente, las 4 iteraciones de la segunda instrucción, tal y como mostramos en el código 1.7.

```
1 int A[N], B[N];
2
3 for (i=0; i<(N-3)-3; i+=4) {
4   A[i]   = A[i+1] + B[i];
5   A[i+1] = A[i+2] + B[i+1];
6   A[i+2] = A[i+3] + B[i+2];
7   A[i+3] = A[i+4] + B[i+3];
8   B[i+1] = B[i+2] + B[i+3];
9   B[i+2] = B[i+3] + B[i+4];
```

```
10   B[i+3] = B[i+4] + B[i+5];
11   B[i+4] = B[i+5] + B[i+6];
12 }
13
14 for (; i<(N-3); i++) {
15   A[i] = A[i+1] + B[i];
16   B[i+1] = B[i+2] + B[i+3];
17 }
```

Código 1.7: Suma de vectores con dependencias de verdad una vez reorganizado el código.

Con esta disposición de las instrucciones en el nuevo bucle podríamos vectorizar los dos grupos de cuatro instrucciones escalares del código, sin romper la dependencia de verdad.

Actividad

Realizar la vectorización del código 1.7 con funciones intrínsecas de SSE2.

1.4. Limitaciones del rendimiento de los procesadores

Hay dos factores que limitan el rendimiento final que se puede obtener con los procesadores. Uno de ellos es el acceso a memoria y el otro es el consumo energético y la ley de Moore.

1.4.1. Memoria

La diferencencia entre la velocidad a la que el procesador puede procesar datos y la que la memoria puede suministrar estos datos es un factor que limita el rendimiento potencial de las aplicaciones. En concreto, tanto la latencia como el ancho de banda de acceso a memoria pueden influir en el rendimiento de los programas.

La latencia es el tiempo que tarda en obtenerse un dato de memoria. Y el ancho de banda es la cantidad de bytes por segundo que la memoria puede ofrecer.

Latencia

Es el tiempo necesario para obtener un dato de memoria. Ancho de banda: cantidad de bytes que la memoria puede suministrar por unidad de tiempo (segundos).

La introducción de la caché, una memoria mucho más rápida con una latencia muy pequeña, y un ancho de banda grande, fue una forma de aliviar la diferencia de velocidad entre procesadores y memoria. Estas memorias tienen el objetivo de tener los datos más usados cerca del procesador con tal de evitar accesos a la memoria lejana y lenta. Cuando ocurre que estamos aprovechando un dato que se encuentra en la caché, se dice que se ha producido un acierto en la caché, y por consiguiente, que se está explotando la localidad temporal. En caso contrario, se dice que se ha producido un fallo en el acceso a la caché. Cada vez que tengamos un acierto, estaremos aprovechándonos de la baja latencia de la caché.

Por otra parte, además del dato que se pide, se produce una transferencia de todos los datos que se encuentran en la misma línea de caché. Con ello se intenta acercar al procesador los datos a los que posiblemente se acceda en un futuro próximo. En caso de que realmente se referencien más adelante, se dirá que se está explotando la localidad espacial. En este caso, aumentar el tamaño de las líneas de caché puede contribuir a mejorar la localidad espacial, además de ayudar a explotar mejor el ancho de banda que pueda ofrecer la memoria.

Línea de caché

Una línea de caché es la únidad básica de movimiento entre memoria y caché.

En cualquier caso, hay aplicaciones que pueden aprovechar mejor la localidad temporal que otras, aumentando la tolerancia de memorias con ancho de banda bajos. De la misma forma, las políticas de reemplazo de una línea de caché por otra, la asociatividad de la caché (*full associative*, *n-set associative*, *direct mapped*, etc.), etc. pueden afectar a la localidad temporal de una aplicación.

Otros mecanismos para mejorar el rendimiento final de un programa en un procesador, desde el punto de vista de la memoria, es ocultar de alguna forma la latencia existente en los accesos a memoria. Uno de estos mecanismos es programar varios *threads* (*multithreading* grano grueso), de tal forma que en procesadores con soporte para ejecutar más de un *thread*, si uno de estos *threads* se para por un fallo de caché, el otro puede continuar. Para que realmente se pueda solapar se debe cumplir que la memoria pueda soportar varias peticiones de memoria en curso*, y el procesador pueda hacer un cambio de contexto en un ciclo (como el caso de los procesadores de *multithreading* de grano grueso).

*** *Outstanding requests* en inglés**

Otra forma de ocultar la latencia de acceso es realizar `prefetch` de los datos que se necesitan. Hay procesadores que son capaces de realizar `prefetch` a nivel hardware cuando detectan patrones de accesos a direcciones consecutivas (o con una cierta distancia) de memoria. Hay compiladores que son capaces de avanzar las lecturas, con respecto a la instrucción que la necesita, con tal de no pagar la latencia de acceso. El programador y el compilador también pueden insertar instrucciones de `prefetch` si es que el procesador dispone de ellas.

`Prefetch` en el compilador gcc

El programador dispone de `__builtin_` de `prefetch` para el compilador de C de GNU (gcc).

En cualquier caso, ambas formas de ocultar la latencia tienen sus inconvenientes. Por ejemplo, el hecho de tener más de un *thread* ejecutándose y pidiendo datos de memoria, puede requerir un aumento en el ancho de banda que debería ofrecer la memoria. Por otra parte, el `prefetch` sobre registros implica que debemos disponer de más registros para poder guardar los datos, y además, realizar los cálculos que queríamos hacer, o de lo contrario, necesitaríamos pedir otra vez los datos.

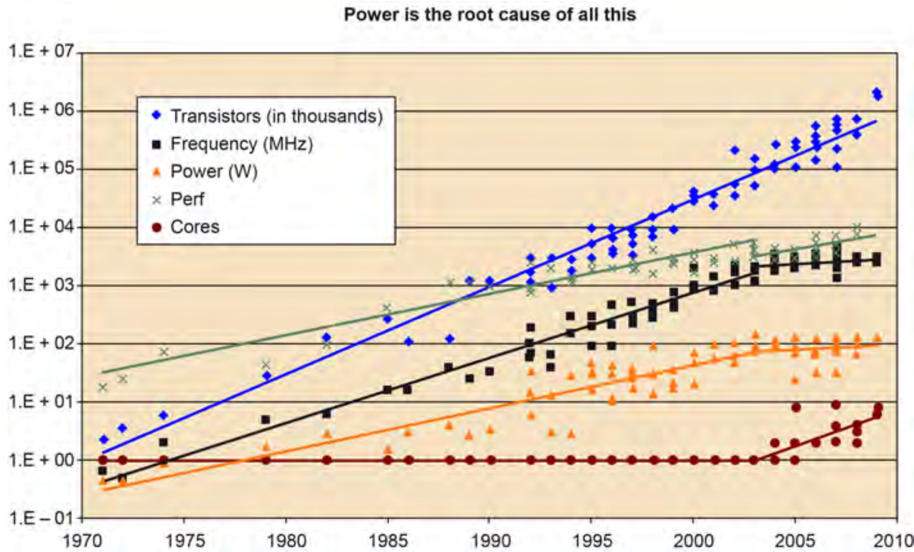
1.4.2. Ley de Moore y consumo de los procesadores

En 1965, Gordon Moore hizo la siguiente observación:

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000".

Y más tarde, en 1975, hizo una revisión del ratio de aumento en la complejidad de los circuitos. En esta ocasión, Moore predijo que esta complejidad se aumentaría cada 18 meses. Esta curva es la que conocemos como ley de Moore, y observando la figura 13, nos damos cuenta de que realmente se ha cumplido.

Figura 13. Evolución del número de transistores según la ley de Moore



Fuente: K. Olukotun, L. Hammond, H. Sutter, B. Smith, C. Batten, y K. Asanovic.

Sin embargo, aunque el número de transistores que podemos integrar en un circuito aumenta cada año, hay un consumo energético máximo que el chip puede soportar, que limita el crecimiento infinito del número de los transistores en un *core* para aumentar el ILP y la velocidad de éste. De hecho, se llegó a un punto en que este consumo comenzaba a afectar negativamente al rendimiento de las aplicaciones. Esta pérdida de rendimiento hizo que, a partir del año 2000 aproximadamente, los diseñadores de procesadores empezaran a optar por no intentar incrementar el ILP de un *core* o su velocidad, y en cambio, aumentar el número de *cores* dentro de un chip (*multicores*).

2. Taxonomía de Flynn y otras

En el apartado anterior hemos hecho una revisión de las principales innovaciones hardware que se han ido realizando a nivel de un uniprocesador.

En este apartado se describe la taxonomía de Flynn, que es una clasificación de las máquinas en función del paralelismo que se explote. La tabla 2 muestra la clasificación que realizó Flynn.

Michael J. Flynn

Nacido en 1934 en Nueva York y profesor emérito de la Universidad de Standford. Cofundó Palyn Associates con Max Paley y es presidente de Maxeler Technologies.

Tabla 2. Taxonomía de Flynn.

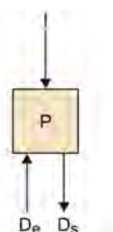
Flujo de Instrucciones	Flujo de Datos	Nombre	Ejemplos
1	1	SISD	von Neumann
1	≥ 1	SIMD	Vectoriales
≥ 1	1	MISD	No conocidos
≥ 1	≥ 1	MIMD	Multiprocesadores/Multicomputadores

Flynn clasifica las máquinas según el programa y los datos que trata. Así, podemos encontrarnos que un programa puede tener más de una secuencia de instrucciones, donde cada secuencia de instrucciones necesita de un contador diferente de programa. Una máquina con varias CPU dispone de varios contadores de programa, y por consiguiente, puede ejecutar varias secuencias de instrucciones. En cuanto a los datos, una secuencia de datos se puede definir como un conjunto de operandos.

Tener una o varias secuencias de instrucciones y una o varias secuencias de datos es independiente. De ahí que podamos distinguir cuatro tipos de máquina que detallamos a continuación:

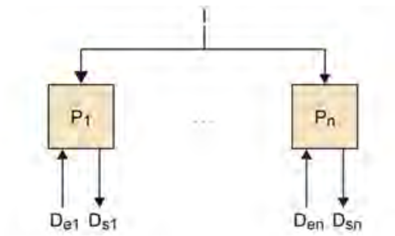
- **SISD**: Las máquinas SISD (*Single Instruction Single Data*) son las máquinas Von Neumann. Son máquinas que ejecutan una única secuencia de instrucciones sobre una secuencia de datos, tratados de uno en uno. La figura 14 muestra un esquema básico de este tipo de arquitectura. Un procesador ejecuta una secuencia de instrucciones (*I*) y las aplica a una secuencia de entrada de datos *D_e* (*e* de *entrada*) y devuelve una secuencia de resultados *D_s* (*s* de *salida*).

Figura 14. Arquitectura SISD.



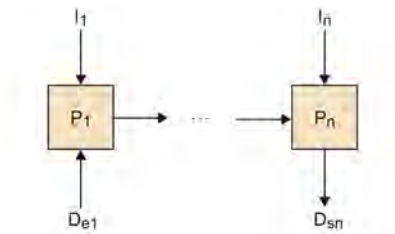
- **SIMD**: Estas máquinas procesan en paralelo más de una secuencia de datos (D_{e1} $\ldots D_{en}$) con una única secuencia de instrucciones (I). Así, todos los procesadores que forman la máquina SIMD toman la misma secuencia de instrucciones que aplican a las diferentes entradas y generan tantas salidas como entradas. La figura 15 muestra un esquema básico de este tipo de arquitectura.

Figura 15. Arquitectura SIMD.



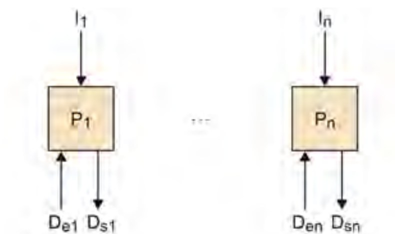
- **MISD**: Esta categoría es un poco difícil de imaginar, y no se conoce ninguna máquina que cumpla con el hecho de tener más de una secuencia de instrucciones que operen sobre los mismos datos. La figura 16 muestra un esquema básico de este tipo de arquitectura.

Figura 16. Arquitectura MISD.



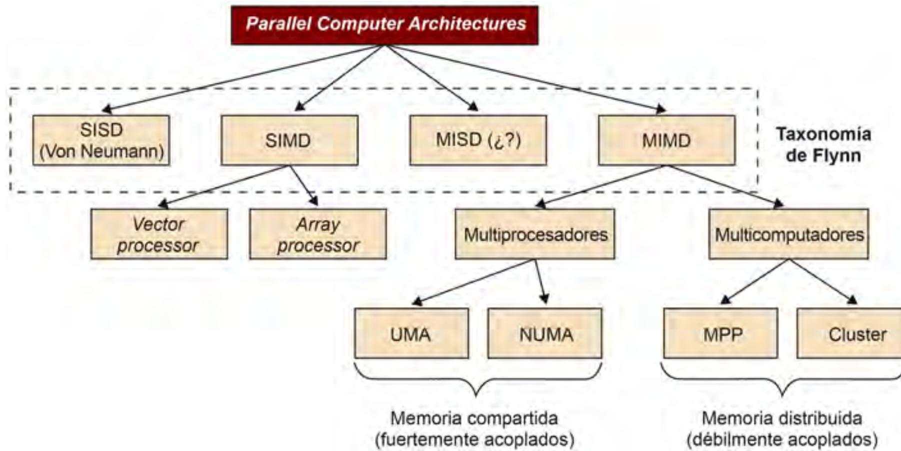
- **MIMD**: Esta es la última categoría que distingue Flynn. En esta categoría múltiples independientes procesadores operan sobre diferentes secuencias de datos. Cada uno de los procesadores aplica una secuencia diferente de instrucciones a una secuencia de entrada, para obtener una secuencia de salida. La mayoría de las máquinas paralelas caen en esta categoría. Figura 17 muestra un esquema básico de este tipo de arquitectura.

Figura 17. Arquitectura MIMD.



Por otro lado, Andrew S. Tanenbaum, en su libro sobre la organización de la estructura del computador amplía esta clasificación. La figura 18 muestra esta ampliación. Así, SIMD la vuelve a dividir en procesadores vectoriales y procesadores en *array*. Los primeros de ellos son procesadores numéricos que procesan vectores, realizando una operación en cada elemento del vector (por ejemplo, Convex Machine). La segunda tiene más que ver con una máquina paralela, como por ejemplo la ILLIAC IV, en la que utilizan múltiples independientes ALU cuando una determinada instrucción se debe realizar. Para ello, una unidad de control hace que todas funcionen cada vez.

Figura 18. Clasificación de las arquitecturas según Andrew S. Tanembaum.



Para el caso de MISD, tal y como hemos comentado, no se ha encontrado ninguna máquina que cumpla con esta clasificación.

Finalmente, para las máquinas tipo MIMD, éstas se pueden clasificar en dos tipos: multiprocesadores* (o máquinas de memoria compartida) y muticomputadores** (o máquinas basadas en paso de mensajes).

*** Multiprocesadores: máquinas MIMD con memoria compartida.**
**** Multicomputadores: máquinas MIMD que no tienen memoria compartida y necesitan paso de mensajes para compartir los datos.**

Dentro de las máquinas de memoria compartida, según el tiempo de acceso y cómo está compartida la memoria, podemos distinguir entre las llamadas UMA (*Uniform Memory Access* o memorias de acceso uniforme) y las llamadas NUMA (*NonUniform Memory Access* o memorias de acceso no uniforme).

Las máquinas UMA* se caracterizan porque todos los accesos a memoria tardan lo mismo. Esto ayuda a que el programador pueda predecir el comportamiento de su programa en los accesos a memoria y programe códigos eficientes.

*** UMA (*Uniform Memory Access*) son multiprocesadores en los que todos los accesos a memoria tardan el mismo tiempo.**

Por contra, las máquinas NUMA** son máquinas en las que los accesos a memoria pueden tardar tiempos diferentes. Así, si un dato está en la memoria cercana a un procesador, se tardará menos que si se accede a un dato que está lejano del procesador.

**** NUMA (*NonUniform Memory Access*) son multiprocesadores en los que los accesos a memoria pueden tardar tiempos diferentes.**

Por otro lado tenemos los multicomputadores, que se distinguen básicamente por el hecho de que el sistema operativo no puede acceder a la memoria de otro procesador únicamente utilizando operaciones de acceso a memoria `load/store`. En cuanto a multicomputadores, distinguen entre MPP (*Massively Parallel Processors*) y clusters de procesadores. Los primeros son supercomputadores formados por CPU que están conectados por redes de interconexión de alta velocidad. Los segundos consisten en PC o estaciones de trabajo, conectadas con redes de interconexión *off-the-shelf*. Dentro de esta última categoría, podemos encontrar los cluster de estaciones de trabajo o COW (*Cluster of Workstations*).

off-the-shelf

Que se puede comprar y no es específico para una determinada máquina.

Finalmente podemos tener sistemas híbridos: clusters en los que los nodos son máquinas con memoria compartida, y sistemas *grids*: máquinas de memoria compartida o de memoria distribuida que están conectadas vía LANs o/y WANs.

A continuación detallaremos más la división realizada de las máquinas MIMD, que son objetivo de este módulo.

2.1. MIMD: Memoria compartida

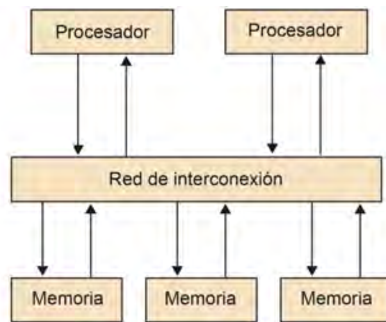
En una máquina de memoria compartida todos los procesadores comparten un mismo espacio de direcciones. En estos sistemas, los *threads* pueden comunicarse los unos con los otros a través de lecturas y escrituras a variables/datos compartidos.

Una de las clases de máquinas de memoria compartida más conocidas son los SMP (*symmetric multiprocessors*), que son máquinas UMA. Todos los procesadores son iguales y van a la misma velocidad, compartiendo todos ellos una conexión para acceder a todas las posiciones de memoria. La figura 19 muestra un ejemplo de un SMP con dos procesadores que se conectan a un `bus` compartido para acceder a la memoria principal.

SMP

Los SMP (*symmetric multiprocessors*) son multiprocesadores UMA que comparten un mismo `bus` para acceder a la memoria compartida.

Figura 19. Arquitectura SMP



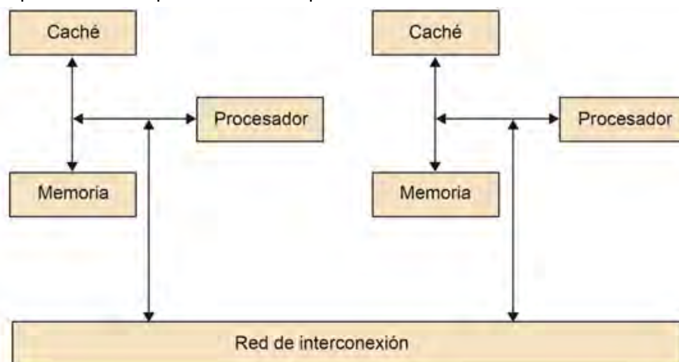
Debido a que todos los procesadores comparten la conexión, este tipo de máquinas no escalan a un gran número de procesadores. La conexión compartida se convierte en un cuello de botella. A pesar de ello, son los sistemas más fáciles de montar y de programar ya que el programador no se preocupa de a dónde van los datos.

El otro tipo de máquina son las NUMA, que permiten escalar a un mayor número de procesadores ya que su conexión a la memoria no es compartida por todos los procesadores. En este caso hay memorias que están cerca y lejos de una CPU, y por consiguiente, se puede tardar un tiempo de acceso diferente dependiendo del dato al que se accede. La figura 20 muestra un esquema básico de una arquitectura `NUMA` formada por dos procesadores.

Multiprocesadores NUMA

En los multiprocesadores NUMA, el programador debe ser consciente de dónde pueden guardarse los datos compartidos. Por ejemplo, la inicialización de los datos en un procesador puede hacer que éste tenga los datos en su memoria cercana, mientras que los otros la tienen en la lejana.

Figura 20. Arquitectura multiprocesador de tipo NUMA.



Para reducir los efectos de la diferencia de tiempo entre acceso a memoria cercana y acceso a memoria lejana, cada procesador dispone de una caché. Por el contrario, para mantener

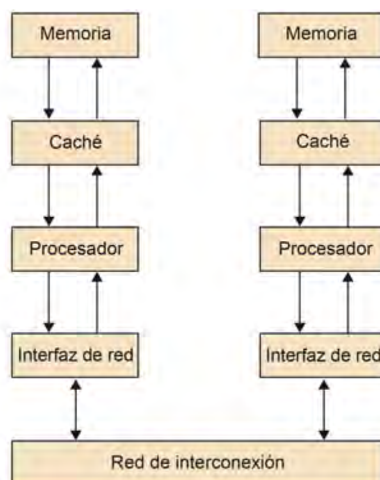
la coherencia de un dato en todas las caché, se precisan protocolos de coherencia de caché. A veces se les conoce como máquinas ccNUMA (*cache-coherent nonuniform memory access*). La forma de programar estas máquinas es tan fácil como una máquina UMA, pero aquí el programador debe pensar bien dónde pone los datos por temas de eficiencia.

2.2. MIMD: Memoria distribuida

En las máquinas de memoria distribuida cada procesador tiene su propio espacio de direcciones y, por consiguiente, los procesos se tienen que comunicar vía paso de mensajes (punto a punto o colectivas).

La figura 21 muestra cómo cada *core* (procesador) dispone de su memoria y que para obtener los datos de la memoria de otro *core*, éstos se deben comunicar entre ellos vía una red de interconexión. La latencia y ancho de banda de la red de interconexión puede variar mucho, pudiendo ser tan rápida como el acceso a memoria compartida, o tan lenta como ir a través de una red ethernet.

Figura 21. Arquitectura memoria distribuida



Desde el punto de vista del programador, éste se debe preocupar de distribuir los datos y, además, de realizar la comunicación entre los diferentes procesos.

Multicomputadores

En los multicomputadores, el programador debe distribuir los datos entre los diferentes procesos y hacer explícita la compartición de datos entre estos procesos mediante la comunicación con paso de mensajes.

2.3. MIMD: Sistemas híbridos

Según algunos autores, como Dongarra y van der Steen, los sistemas híbridos están formados por clusters de SMPs conectados con una red de interconexión de alta velocidad. Además, estos mismos autores, en el año de la publicación, indicaban que era una de las tendencias actuales. De hecho, en el año 2003, cuatro de los 5 principales computadores más potentes en el Top500 eran de este tipo.

Lectura complementaria

Aad J. van der Steen, Jack J. Dongarra (2003). *Overview of Recent Supercomputers.*

2.4. MIMD: *Grids*

Los *grids* son sistemas que combinan sistemas distribuidos, de recursos heterogéneos, que están conectados a través de `LANs` o/y `WANs`, soliendo ser Internet.

En un principio, los sistemas *grids* se vieron como una forma de combinar grandes supercomputadores para resolver problemas muy grandes. Después se ha derivado más hacia una visión de un sistema para combinar recursos heterogéneos como servidores, almacenamiento de grandes volúmenes de datos, servidores de aplicaciones, etc. En cualquier caso, la distinción básica entre un cluster y un *grid* es que en este último no hay un punto común de administración de los recursos. Cada organización que está dentro de una *grid* mantiene el control de la gestión de sus recursos, lo que repercute en los mecanismos de comunicación entre los computadores de las organizaciones.

Lectura complementaria

Ian Foster, Carl Kesselman (2003). *The Grid 2: Blueprint for a New Computing Infraestructure* (2a. ed.) Morgan Kaufmann.

3. Medidas de rendimiento

En este apartado trabajaremos con métricas que nos ayudarán a analizar el paralelismo potencial de un programa. Analizaremos cómo afecta el grano de paralelismo en la estrategia de paralelización al paralelismo potencial, y por consiguiente, al *speedup* (cuántas veces más rápido) ideal que podemos alcanzar. Finalmente, trabajaremos un modelo sencillo de tiempo que nos ayude a analizar cuál es el paralelismo y *speedup* real que podemos conseguir en un determinado computador.

Grano de paralelización

El grano de paralelización (trabajo a realizar por tarea) determina el paralelismo potencial que podremos explotar en nuestra estrategia de paralelización.

3.1. Paralelismo potencial

Para medir el paralelismo potencial que vamos a alcanzar en la resolución de un problema, debemos pensar primero en la distribución de trabajo que vamos a realizar. Hay dos formas de realizar la distribución del trabajo:

Distribución del trabajo

Se pueden realizar dos tipos de distribución del trabajo: *Task Decomposition* si pensamos en las operaciones a realizar, y *Data Decomposition* si pensamos en los datos a tratar.

1) Según una distribución desde un punto de vista de tareas (*Task Decomposition* en inglés).

2) Según un punto de vista de datos (*Data Decomposition*).

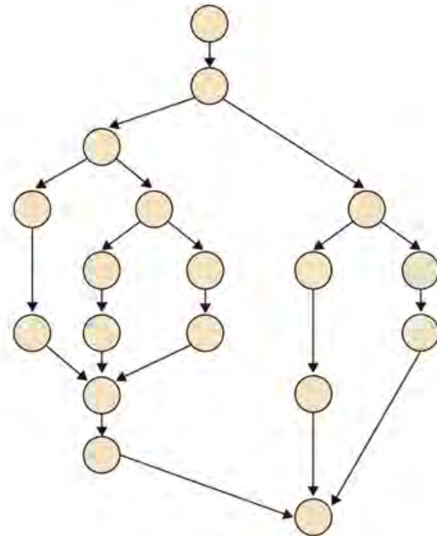
En el primer caso, la distribución podría ser cada llamada a función o cada iteración de un bucle que es una tarea. En el segundo caso, en el caso de tener que tratar una matriz, podríamos dividir la matriz entre los diferentes procesos/*threads*, y después asignar una tarea a cada fragmento.

Una vez hecha la distribución de trabajo entre tareas, ya sea después de una *Task Decomposition* o una *Data Decomposition*, debemos crear el grafo de dependencias entre tareas con tal de poder analizar el paralelismo existente. Este grafo está formado por nodos (tareas), y las aristas entre nodos indican las dependencias entre ellas (origen de la arista es la fuente de la dependencia, es decir, quién genera el dato). Cada dependencia indica que un nodo sólo puede proceder a hacer su cálculo cuando todos los nodos de los que depende ya han generado los datos. La figura 22 muestra un posible grafo de dependencias entre tareas. Este tipo de grafos son grafos acíclicos dirigidos (DAG - *Directed Acyclic Graph*). Para simplificar, vamos a suponer que disponemos de P procesadores, y cada uno de ellos podrá ejecutar un nodo en cada momento.

Grafo de tareas

El grafo de tareas resulta de analizar las dependencias entre las diferentes tareas a realizar.

Figura 22. Grafo de dependencias de las tareas.



A partir del grafo de dependencias de tareas podemos realizar el análisis del programa paralelo y qué paralelismo potencial podemos obtener en comparación con el programa secuencial. Definiremos primero una serie de conceptos, y después, con un ejemplo simple, veremos cómo aplicarlos.

- T_1: Es el tiempo de la ejecución secuencial de todos los nodos del grafo, es decir, la suma del coste de cada una de las tareas del grafo de dependencias:

$$T_1 = \sum_{i=1}^{nodos} (coste_nodo_i)$$

- T_∞: o *Span* en inglés, es el tiempo mínimo necesario para poder acabar todas las tareas ejecutándose con un número infinito de procesadores. Desde el punto de vista del grafo de dependencias, es el camino crítico desde la primera y la última tarea a ejecutarse.

- Paralelismo: definido como $\frac{T_1}{T_\infty}$. Éste es independiente del número de procesadores existentes, y depende únicamente de las dependencias entre tareas. Es el paralelismo potencial que podemos explotar si tuvieramos un número infinito de procesadores.

- Holgura de paralelismo*: se define como $\frac{T_1}{T_\infty}/P$, donde P es el número de procesadores. Nos indica un límite inferior del número de procesadores necesarios para conseguir el paralelismo potencial. Esto no significa que con este número de procesadores podamos conseguir T_∞.

Paralelismo

El paralelismo existente en un programa es independiente del número de procesadores de los que se dispone. Depende únicamente de la distribución de tareas realizada y las dependencias entre ellas.

* ***Parallel slackness* en inglés**

En el código 3.1 tenemos un código que no tiene más propósito que el de desarrollar un análisis del paralelismo potencial según la granularidad en la definición de tareas. La granularidad es la cantidad de trabajo que queremos asignar a cada tarea. Ésta puede ser fina si le asignamos un trabajo pequeño, y gruesa si le damos mayor cantidad de trabajo. Para el ejemplo en cuestión supondremos que el coste de realizar la instrucción del bucle más in-

terno del bucle anidado es *O*(1). Para el `print` del bucle *ij* posterior también supondremos que esa instrucción tiene coste *O*(1).

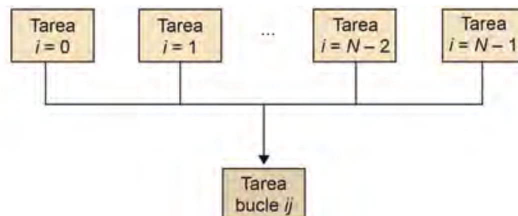
```
1
2   for (i=0; i<N; i++)
3     for (j=0; j<M; j++)
4       C[i][j] = A[i][j]*B[i][j];
5
6   ptr = (int *)C;
7   for (ij=0; ij<N*M; ij++, ptr++)
8      printf("%d\n", *ptr);
```

Código 3.1: Código muy simple para analizar el paralelismo potencial.

Una primera distribución de trabajo sería tener dos tareas de grano grueso: (tarea 1) los dos bucles anidados (bucles *i* y *j*) , y (tarea 2) el bucle *ij*. El coste de la primera y segunda tarea es $O(N * M)$. Por consiguiente T_1 es $2 * O(N * M)$. Para calcular el T_∞ debemos analizar el grafo de dependencias de tareas: la tarea 2 se tiene que esperar a que la tarea 1 acabe para poder imprimir por pantalla los datos correctos. Por consiguiente, el T_∞ es exactamente igual que el T_1.

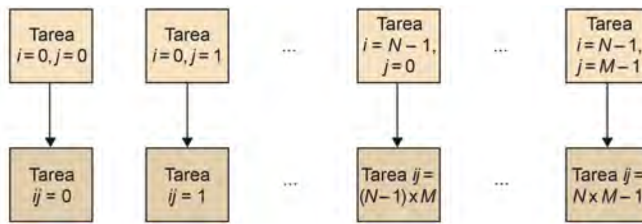
Otra posible distribución del trabajo en tareas sería tener una tarea para cada iteración del bucle *i*, y dejar la antigua tarea 2 igual. De esta forma, hemos realizado una distribución de tareas de un grano más fino. En este caso T_1 sigue siendo el mismo ya que no hemos cambiado el algoritmo secuencial. Sin embargo, T_∞ se ha reducido. El grafo de dependencias de tareas ha cambiado significativamente. La figura 23 muestra que ahora todas las tareas que corresponden con una iteración del bucle *i* se pueden realizar en paralelo. Cada una de estas tareas del bucle *i* tiene un coste de *O*(*M*). La tarea 2 no cambia. Por consiguiente, T_∞ es ahora $O(M) + O(N * M)$.

Figura 23. Grafo de dependencias de las tareas cuando cada iteración *i* de los bucles anidados es considerada una tarea.



Finalmente, si nos decantamos por un grano más fino en la distribución de tareas tanto del bucle anidado, como del bucle *ij*, podríamos definir cada tarea como cada iteración del bucle *j* y cada iteración del bucle *ij*. T_1 seguirá siendo el mismo, ya que no hemos tenido que hacer ningún cambio en el algoritmo. T_∞, sin embargo, se ha reducido significativamente. Suponiendo que podemos hacer un `print` de cada dato de forma paralela, todas las tareas del bucle *j* y las del bucle *ij* se pueden hacer en paralelo. Cada una de ellas tiene un coste de $O(1)$. Por lo tanto, T_∞ se ha reducido a $2 \times O(1)$. Esto es así ya que el grafo de dependencias de tareas queda como muestra la figura 24, con lo que cada elemento a imprimir solo depende del cómputo del dato correspondiente en el bucle anterior.

Figura 24. Grafo de dependencias de las tareas cuando cada iteración j de los bucles más internos es considerada una tarea.



Tal y como hemos visto, una granularidad fina favorece el T_∞. Por consiguiente, uno podría pensar, naturalmente, que la idea es hacer tareas de grano muy fino, con lo que podríamos aumentar el número de tareas que se pueden hacer en paralelo. Sin embargo, un primer límite que nos podemos encontrar está en el número máximo de tareas de grano muy fino que se pueden definir. En el caso anterior no podemos definir más de $2 \times N \times M$ tareas. Por otro lado, en el caso de tener que intercambiar datos entre un número elevado de tareas, esto podría significar un coste adicional excesivo. Otros costes adicionales, en el momento de crear el programa paralelo con todas estas tareas, son: el coste de creación de cada una de las tareas, la sincronización de éstas, la destrucción de las tareas, etc. Es decir, que debe haber un compromiso entre el grano fino, y por consiguiente el número de tareas que se van a crear, y los costes adicionales que supone tener que gestionar todas esas tareas y el intercambio de datos y sincronización entre ellas.

Granularidad fina

La granularidad muy fina favorece el T_∞, pero debemos tener en cuenta que los costes de gestión y sincronización de un número elevado de tareas puede afectar al rendimiento real del programa paralelo.

3.2. *Speedup* y eficiencia

En el subapartado anterior calculábamos el paralelismo potencial (T_∞) de una aplicación tras determinar la distribución en tareas y el camino crítico del grafo de dependencias existente entre ellas. Sin embargo, muchas veces el T_∞ es difícil de alcanzar en una máquina paralela MIMD con un número de procesadores P limitado.

Definiremos T_P como el tiempo real que tardamos en realizar los nodos del grafo de dependencia de tareas con P procesadores. Sabemos que T_P es normalmente mayor o igual que T_1/P y T_∞*, es decir, como mucho podremos dividir el trabajo entre los P procesadores, y no bajaremos de los T_∞ que habíamos calculado.

*** T_∞ es límite inferior a T_P**

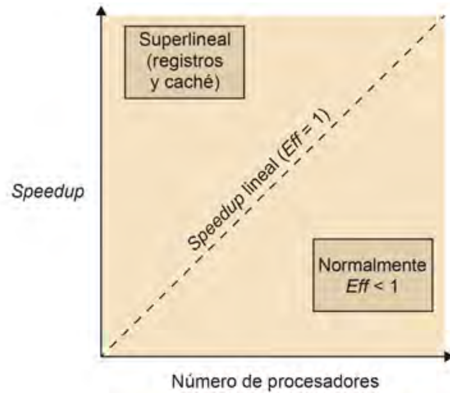
Así, el *speedup* conseguido por un programa paralelo con respecto al programa en secuencial se define como:

$$S_P = \frac{T_1}{T_P}$$

En concreto, podemos definir *speedup* como la relativa reducción de tiempo de ejecución, al procesar un tamaño fijo de datos cuando usamos P procesadores, con respecto al tiempo de ejecución del programa secuencial. La curva del *speedup* debería ser idealmente una función lineal de pendiente 1, es decir, que fueramos P veces más rápido con P procesadores. La figura 25 muestra las tres situaciones con las que nos podemos encontrar cuando hacemos un programa paralelo y analizamos el *speedup*. La situación normal es que nuestros programas, una vez paralelizados, consigan un *speedup* menor que el lineal, ya que

los costes de comunicación, sincronización y creación/destrucción de tareas suelen hacer que el coste de paralelización no sea a coste cero. Sin embargo, se pueden dar casos en los que obtenemos un *speedup* superior al lineal. Este es el caso de programas paralelos que ayudan a explotar la jerarquía de memoria, o los registros del procesador, cosa que antes no se podía con el programa secuencial.

Figura 25. Eficiencia lineal y situaciones donde es menor o superior: superlineal.



Otra medida que nos ayuda a determinar lo buena que es la paralelización realizada es la eficiencia. La eficiencia se define como la medida de la fracción de tiempo en la que cada procesador es usado para resolver el problema en cuestión de forma útil. Si nuestros procesadores se utilizan de forma eficiente, eso significará que el tiempo dedicado por cada procesador por el número de procesadores debería ser T_1. Así, eficiencia (Eff_P) se define como:

$$\mathit{Eff}_P = \frac{T_1}{T_P \times P}$$

$$\mathit{Eff}_P = \frac{S_P}{P}$$

3.3. Ley de Amdahl

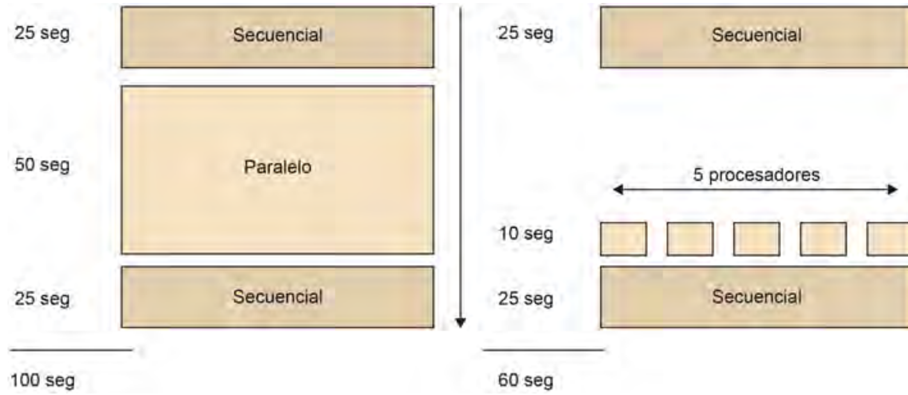
En la paralelización de una aplicación, normalmente hay alguna parte que no se puede paralelizar. Esta fracción de tiempo invertida en esta parte secuencial va a limitar la mejora de rendimiento que vamos a obtener de la paralelización, y por consiguiente, la eficiencia obtenida.

Ley de Amdahl

La fracción de tiempo invertida en esta parte secuencial va a limitar la mejora de rendimiento que vamos a obtener de la paralelización.

En la figura 26 mostramos el tiempo que tardamos en un código totalmente ejecutado en secuencial (izquierda) y un código en el que hay una parte, definida como paralela, que se ha paralelizado perfectamente entre 5 procesadores. Si calculamos el *speedup* que podemos conseguir, obtenemos que $S_P = 100/60 = 1{,}67$, aun habiendo conseguido una eficiencia de 1 en parte paralela del programa.

Figura 26. Ejecución en secuencial y en paralelo.



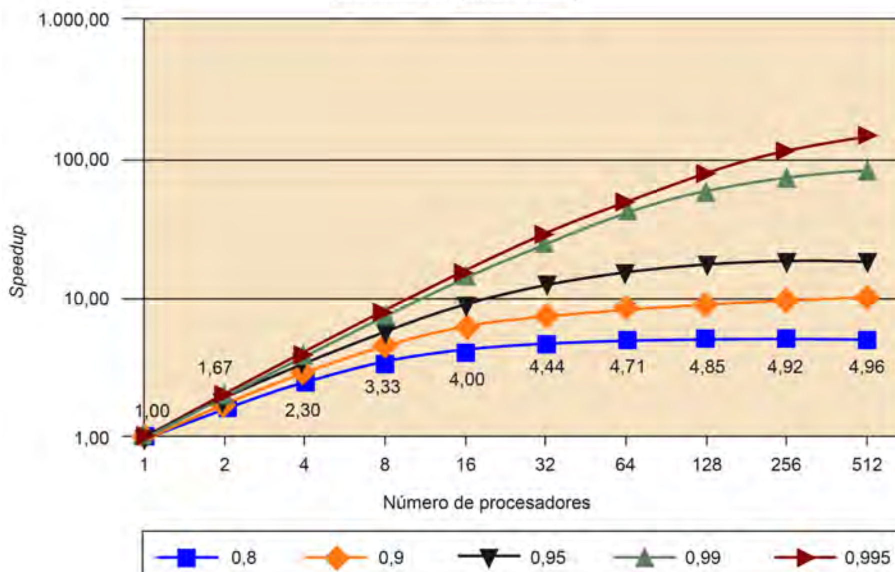
La ley de Amdahl indica que la mejora de rendimiento está limitada por la fracción de tiempo que el programa está ejecutándose en paralelo. Si llamamos a esta fracción ϕ, tendríamos que:

$$
\begin{aligned}
T_1 &= T_{seq} + T_{par} = (1-\phi) \times T_1 + \phi \times T_1 \\
T_P &= (1-\phi) \times T_1 + \phi \times T_1/P \\
S_P &= \frac{T_1}{T_P} \\
S_P &= \frac{1}{(1-\phi) + \phi/P}
\end{aligned}
$$

De tal forma que si el número de procesadores tiende a infinito, el *speedup* máximo que se podrá obtener únicamente dependerá de la fracción que no se puede paralelizar, tal y como muestra la siguiente fórmula, y como gráficamente nos muestra la figura 27:

$$
S_P \rightarrow \frac{1}{(1-\phi)} \quad para \quad P \rightarrow \infty
$$

Figura 27. Curva de *speedup* que se puede obtener para una determinada ϕ.



En esta figura podemos ver que aun disponiendo de un 80 % ($\phi = 0{,}8$) del tiempo de ejecución para paralelizar, el máximo *speedup* que podemos conseguir con 512 procesadores es de $4{,}96\times$, siempre y cuando no añadamos ningún tipo de coste adicional para conseguir la paralelización. Sin embargo, normalmente la paralelización no es gratis, y debemos pagar algunos costes.

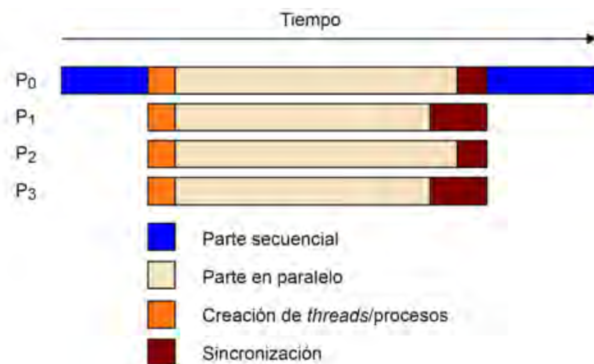
Algunos de los costes que se deben pagar al realizar la paralelización de una aplicación son debidos a:

- La creación y la terminación de procesos/*threads*. Aquí tenemos un procesamiento extra al iniciar y finalizar las tareas. Notad que la creación de procesos es mucho más cara que la creación de *threads*.
- Sincronización. Para poder asegurar las dependencias entre tareas existentes en el grafo de dependencias de las tareas.
- Compartición de datos. Esta comunicación puede ser que se tenga que realizar con mensajes explícitos o bien vía jerarquía de memoria.
- Cálculo. Hay cálculos que se replican con tal de no tener que hacer la comunicación entre las tareas.
- Contención cuando se realizan accesos a recursos compartidos, como la memoria, la red de interconexión, etc.
- La inactividad de algunos procesos/*threads*, debido a las dependencias entre tareas, el desbalanceo de carga, un pobre solapamiento entre comunicación y cálculos, etc.
- Estructuras de datos necesarias extras, debido a la paralelización del algoritmo.

La figura 28 muestra un gráfico de ejecución, para el caso de 4 procesadores, donde aparecen detallados algunos de estos costes de paralelización. Estos costes pueden ser constantes, o lineares con respecto al número de procesadores. T_P para el caso de un coste adicional es:

$$T_P = (1 - \phi) \times T_1 + \phi \times T_1/P + sobrecoste(P)$$

Figura 28. Costes adicionales que pueden aparecer en la ejecución de un programa paralelo.



En el primero de los casos, con un coste constante, hace que el rendimiento ideal, para una determinada fracción de ejecución de tiempo a paralelizar, siga la forma de la figura 29. Si comparamos el rendimiento final para $\phi = 0{,}9$ cuando no tenemos coste de paralelización y cuando lo tenemos de coste constante, observamos que el rendimiento ideal baja significativamente. Es más, si comparamos el caso de coste cero con el coste lineal en función del número de procesadores, este coste adicional* hace contraproducente aumentar el número de procesadores, haciendo que *speedup* sufra un retroceso, como podemos observar en la figura 30.

*** El sobrecoste de paralelización puede influir significativamente en el *speedup* ideal.**

Figura 29. Amdalh cuando tenemos un coste adicional de paralelización no despreciable y constante.

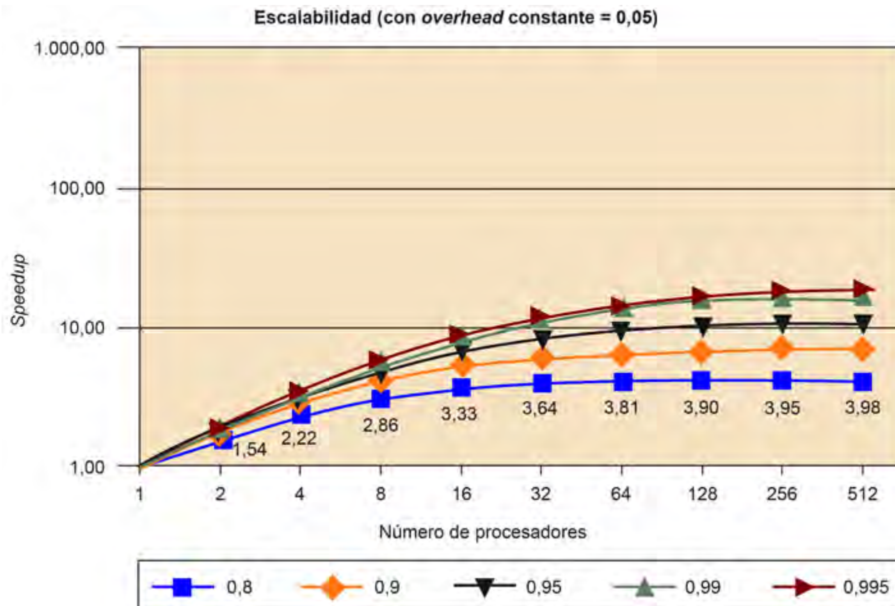
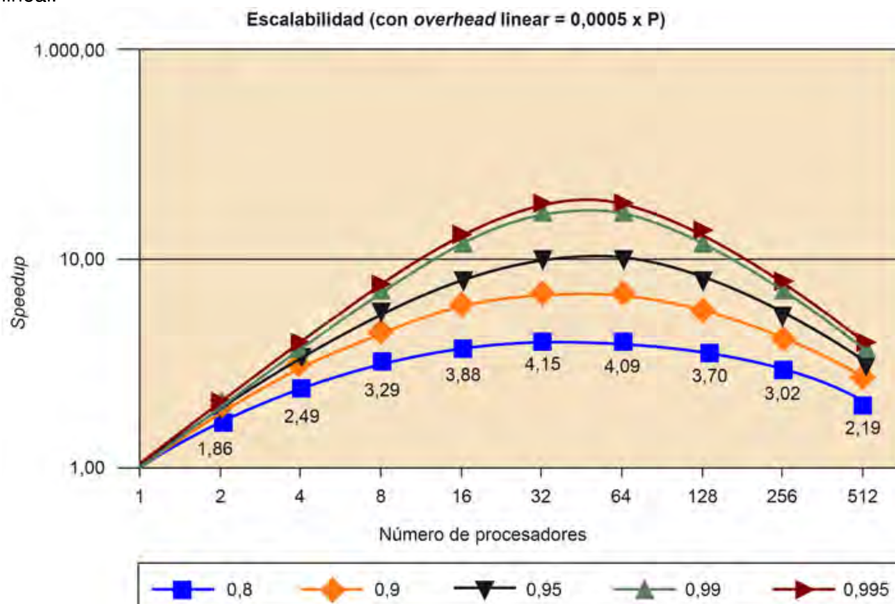


Figura 30. Amdalh cuando tenemos un coste adicional de paralelización no despreciable y lineal.

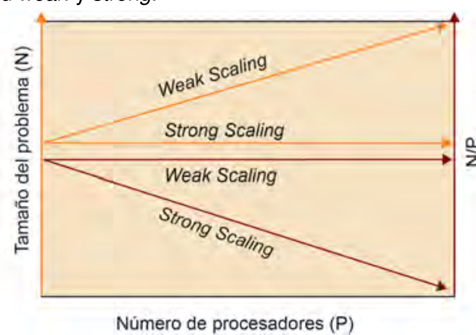


3.4. Escalabilidad

La escalabilidad es una medida de cómo se comporta un programa paralelo cuando aumentamos el tamaño del problema proporcionalmente al número de procesadores, o aumentamos el número de procesadores sin variar el tamaño del problema a tratar.

La figura 31 muestra las dos medidas de escalabilidad que se pueden hacer, según la miremos desde el punto de vista del tamaño total del problema a tratar (eje y izquierdo), y el tamaño del problema a tratar por procesador (eje y derecho). Así, la *Weak Scalability* es una medida que mantiene fijo el tamaño que le toca a cada procesador, y por consiguiente, el tamaño aumenta con el número de procesadores. Por el contrario, la *Strong Scalability* es una medida que mantiene el tamaño fijo del problema a tratar para cualquier número de procesadores a tratar.

Figura 31. Escalabilidad *weak* y *strong*.



Normalmente, cuando se analiza el *speedup* estamos analizando también la *Strong Scalability*.

Otra medida que nos indica cuán adaptable es nuestra estrategia de paralelización a problemas pequeños es $N_{\frac{1}{2}}$, que nos indica cuál es el tamaño mínimo necesario para conseguir una eficiencia (Eff_P) del 0,5 para un número determinado de procesadores P. Un valor grande de $N_{\frac{1}{2}}$ indica que el problema es difícil de paralelizar con la estrategia usada.

3.5. Modelo de tiempo de ejecución

El coste de compartición de los datos depende del tipo de arquitectura que tengamos y de la paralelización usada. La compartición en una máquina de memoria compartida es mucho más sencilla, pero es mucho más difícil de modelar. En cambio, en el caso de memoria distribuida es más difícil de programar, pero más sencillo de modelar.

En este subapartado nos centraremos en los costes de comunicación con paso de mensajes. En este tiempo de comunicación tenemos:

- Tiempo de inicialización* (t_s): que consiste en preparar el mensaje, determinar el camino del mensaje a través de la red, y el coste de la comunicación entre el nodo local y el *router*.

* ***Start up*** **en inglés**

- Tiempo de transmisión del mensaje, que a su vez consta de:

– por *hop* (t_h): tiempo necesario para que la cabecera del mensaje se transmita entre dos nodos directamente conectados en la red.

– por *byte* (t_w): tiempo de transmisión de un elemento (word).

Así, dependiendo del camino que tome el mensaje y el mecanismo de enrutamiento que siga, tendremos un coste de comunicación u otro. Distinguiremos entre tres mecanimos de enrutamiento:

1) Enrutamiento *Store-and-Forwarding*: en este caso los mensajes viajan de un nodo a otro. Cada nodo recibe el mensaje, lo almacena y luego hace el reenvío del mensaje.

El coste de comunicación en este caso es:

$$T_{comm} = t_s + (m \times t_w + t_h) \times l$$

donde l es el número de nodos que visitamos (conectados directamente), y m es el número de words del que está formado el mensaje.

Sin embargo, como t_h es normalmente muy pequeño, podemos simplificar la fórmula y quedarnos con la siguiente:

$$T_{comm} = t_s + m \times t_w \times l$$

2) Enrutamiento *Packet*: en este caso el mensaje se divide en partes de tal forma que podamos explotar mejor los recursos de comunicación, ya que pueden seguir caminos diferentes, evitando la contención de algunos caminos. Además, con la división de paquetes reducimos el número de errores.

Por el contrario, el tamaño del mensaje global se incrementará debido a que debemos añadir las cabeceras, información de control de errores y de secuenciamiento del mensaje. Además, hay un tiempo invertido en la división en partes. Esto hace que se aumente el t_s.

3) Enrutamiento *Cut-Through*: este tipo de enrutamiento intenta reducir el coste adicional de división en partes. Para ello todas las partes son forzadas a tener el mismo enrutamiento e información de error y secuenciamiento. Así, un traceador se envía para decidir el camino desde el nodo origen y el nodo destino.

Los mensajes, cuando se comunican de un nodo a otro, no son copiados y reenviados como se hacía con el *Store-and-Forward*. De hecho, no se espera a que todo el mensaje llegue para empezar a reenviarlo.

El coste de comunicación en este tipo de enrutamiento es:

$$T_{comm} = t_s + m \times t_w + t_h \times l$$

donde podemos observar que el coste de transmisión no se multiplica por el número de *links* que se deben atravesar.

La mayoría de la máquinas paralelas suelen tener este tipo de enrutamiento.

Suponiendo el coste de comunicación del enrutamiento *Cut-Through*, podríamos preguntarnos: ¿Cómo podemos reducir el coste de comunicación?

Hay varias formas de conseguirlo:

Reducir el coste

Formas de reducir el coste de comunicación: agrupar mensajes, reducir el volumen a comunicar, y reducir la distancia de comunicación.

- Agrupar datos a comunicar cuando sea posible. Normalmente t_s es mucho mayor que t_h y t_w. Así, si podemos juntar mensajes, reduciremos el número de mensajes y el peso de t_s.

- Reduciendo el volumen total de datos a comunicar. Así reducimos el coste que pagamos por word comunicado (t_w). Conseguirlo dependerá de cada aplicación.

- Reduciendo la distancia de comunicación entre los nodos. Éste es el más difícil de conseguir, ya que depende del enrutamiento realizado. Por otra parte, normalmente se tiene muy poco control del mapeo de los procesos a los procesadores físicos.

En cualquier caso, podemos hacer una simplificación de la fórmula anterior teniendo en cuenta que:

- El peso del factor t_s es mucho mayor que t_h cuando los mensajes son pequeños.

- El peso del factor t_w es mucho mayor que t_h cuando los mensajes son grandes.

- l no suele ser muy grande, y t_h suele ser pequeño.

Por lo que el término $l \times t_h$ lo podemos ignorar, simplificando el modelo en la siguiente fórmula:

$$T_{comm} = t_s + m \times t_w$$

La forma de obtener t_s y t_w de una máquina, para que ajuste el modelo realizado, se podría hacer mediante *Minimal Mean Square Error**.

*** Raj Jain (1991). "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling". *Wiley- Interscience*. Nueva York.**

Para realizar un modelo para el caso de trabajar con una máquina de memoria compartida, se tendrían que tener muchos factores en cuenta, como por ejemplo: cómo es la organización física de la memoria, las cachés de las que disponemos y su capacidad, el coste de mantener sistemas de coherencia, la localidad espacial y temporal, el *prefetching*, la compartición falsa, la contención de memoria, etc., que hacen muy difícil hacer un modelo razonable. Sin embargo, haciendo una serie de suposiciones, y con el único objetivo de tener un modelo de compartición de datos, podríamos considerar que este modelo de comunicación se podría tomar también para programas de memoria compartida, considerando como unidad básica de comunicación la línea de caché.

3.6. Casos de estudio

En este apartado realizaremos el modelo de tiempo de ejecución para dos problemas: un *Edge-detection* y un *Stencil*. El primero lo realizaremos sin aplicar la técnica de optimización *Blocking*, y el segundo aplicándola. *Blocking* permite distribuir el trabajo en bloques de tal forma que: ayude a mejorar el rendimiento de la jerarquía de memoria al explotar mejor la localidad temporal de datos (no analizado aquí), y favorecer el paralelismo entre procesos/*threads*.

Nota

En los casos de estudio estamos suponiendo que cada proceso se ejecuta en un procesador independiente.

3.6.1. Ejemplo sin *Blocking*: *Edge-Detection*

En este primer ejercicio vamos a:

- Modelar el tiempo invertido en la ejecución en paralelo.
- Calcular el `speedup`.
- Obtener la eficiencia de una paralelización del Edge-Detection.

Lectura complementaria

A. Grama y otros (2003). *Introduction to Parallel Computing*. Boston: Addison Wesley.

El código 3.2 muestra una aproximación del *Edge-Detection*. Este código refleja cómo se aplica un template de 3×3 a todas las posiciones de la variable `in_image`, dejando el resultado en `out_image`.

```
1
2 int apply_template_ij(int i, int j, int in_image[N+2][N+2], int
      template[3][3])
3 {
4    int ii, jj;
5    int apply = 0;
6
7    for (ii=i-1; ii<i+1; ii++)
8      for (jj=j-1; jj<j+1; jj++)
9        apply += in_image[ii][jj] * template[ii-i+1][jj-j+1];
10
11   return apply;
12 }
13
14 void Edge-Detection(int in_image[N+2][N+2], int out_image[N+2][N+2],
      int template[3][3])
15 {
16   int i, j;
17
18   for (i=1; i<N+1; i++)
19     for (j=1; j<N+1; j++)
20       out_image[i][j] = apply_template_ij(i,j,in_image,template);
21 }
```

Código 3.2: Código del Edge-Detection

Para realizar el cáculo del `speedup` tenemos que calcular el tiempo de ejecución secuencial y el tiempo de ejecución en paralelo. Vamos a suponer que el tiempo de cada

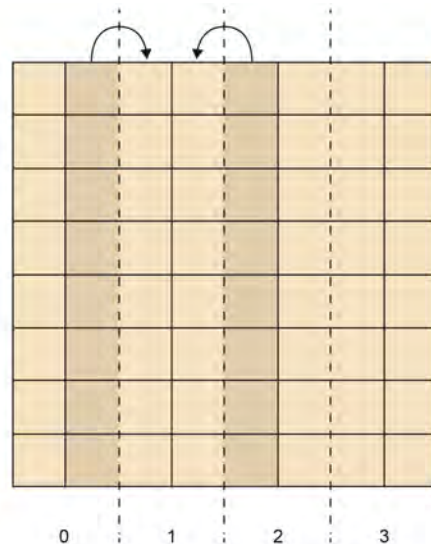
multiplicación y suma (una única instrucción) en la función `apply_template_ij` es de t_c.

Empezaremos primero por el cálculo del tiempo secuencial, T_1. Si analizamos el código, observamos que para cada elemento de la matriz llamamos a la función `apply_template_ij`, y que cada llamada significa realizar 9 operaciones de multiplicación y suma. Es decir, el tiempo total de ejecución en secuencial es:

$$T_1 = 9 \times t_c \times N^2$$

Para el cálculo del tiempo de la ejecución en paralelo necesitamos plantear una estrategia de paralelización y de distribución de trabajo. Los datos los distribuiremos por columnas. A cada proceso asignaremos $\frac{N}{P}$ columnas enteras, es decir $\frac{N}{P} \times N \rightarrow \frac{N^2}{P}$ píxeles (un segmento). Cada proceso necesitará dos columnas (*boundaries*), pertenecientes a los procesos izquierdo y derecho, para realizar el cálculo de su segmento. Cada *boundary* tiene N píxeles. La figura 32 muestra la distribución de los datos y las *boundaries* izquierda y derecha del proceso 1.

Figura 32. Distribución de los datos y *boundaries* para el *Edge-detection*.



Estrategia de paralelización:

Para realizar la paralelización de este código vamos a seguir la siguiente estrategia para cada proceso:

1) Primero, intercambiar las *boundaries* con los dos procesos adyaccentes.

2) Segundo, aplicar la función `apply_template` a su segmento.

A continuación calcularemos el tiempo de ejecución de esta paralelización (T_P). Para ello primero calcularemos el tiempo de comunicación de las *boundaries*. Cada proceso hace

dos mensajes, de N píxeles cada uno, y todos los procesos en paralelo. El tiempo de comunicación es:

$$T_{comm} = 2(t_s + t_w N)$$

Por otro lado, cada proceso, de forma independendiente y en paralelo con el resto de procesos, aplicará el template a cada uno de los píxeles de su segmento. El tiempo de cómputo es:

$$T_{computo} = 9t_c \frac{N^2}{P}$$

Esto nos lleva al tiempo de ejecución en paralelo con P procesadores (T_P):

$$T_P = 9t_c \frac{N^2}{P} + 2 \times (t_s + t_w N)$$

Siendo el *speedup* y la eficiencia de:

$$S_P = \frac{9 \times t_c \times N^2}{9 \times t_c \times \frac{N^2}{P} + 2 \times (t_s + t_w N)}$$

$$Eff_P = \frac{1}{1 + \frac{2 \times P \times (t_s + t_w N)}{9 \times t_c \times N^2}}$$

3.6.2. Ejemplo con *Blocking*: *Stencil*

En este caso vamos a analizar el tiempo de ejecución en paralelo del código del *Stencil* (Código 3.3).

```
1  #include <math.h>
2  void compute( int N, double *u) {
3      int i, k;
4      double tmp;
5
6      for ( i = 1; i < N-1; i++ ) {
7         for ( k = 1; k < N-1; k++ ) {
8            tmp = u[N*(i+1) + k] + u[N*(i-1) + k] + u[N*i + (k+1)] + u[N*
                 i + (k-1)] - 4 * u[N*i + k];
9            u[N*i + k] = tmp/4;
10        }
11     }
12 }
```

Código 3.3: Código de Stencil

La distribución de los datos entre los procesos es por filas. Cada proceso tiene que tratar N/P filas consecutivas de la matriz (o N^2/P elementos, un segmento). Para que la ejecución sea eficiente, cada proceso procesará las filas asignadas en bloques de B columnas, tal y como detallaremos en la estrategia de paralelización. En este caso también tenemos *boundaries* para cada proceso, la fila inmediatamente anterior a su segmento (*boundary* superior) y la fila inmediatamente posterior a su segmento (*boundary* inferior).

Estrategia de paralelización:

En primer lugar, y sólo una vez, cada proceso realizará la comunicación de la *boundary* inferior al proceso predecesor. Esta comunicación se puede hacer antes de empezar el cálculo, ya que no hay dependencias entre procesos. A continuación, cada proceso realizará, para cada bloque de B columnas, los siguientes pasos:

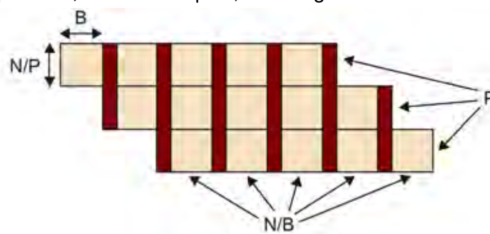
1) Esperar los B elementos de la última fila del bloque procesado por parte del proceso superior (B elementos de la *boundary* superior). En el caso del proceso cero, éste no tiene esta comunicación.

2) Aplicar el algoritmo de Stencil al bloque de $B \times \frac{N}{P}$ elementos.

3) Enviar los B elementos de la última fila del bloque que se acaba de procesar al proceso inferior. A excepción del último proceso.

La figura 33 muestra cómo se procesan en el tiempo los bloques de cada procesador. Las bandas verticales rojas reflejan la comunicación de las *boundaries*. En la figura no se muestra la comunicación de las *boundaries* inferiores.

Figura 33. Ejecución paralela, usando bloques, de código *Stencil*.



Realizando una simplificación de $N-2 \rightarrow N$, nos lleva a que el tiempo de comunicación de todas las *boundaries* inferiores en paralelo es de:

$$T_{comm} = (t_s + Nt_w)$$

Por otra parte, el tiempo de cómputo y comunicación para el conjunto de bloques a tratar es de:

$$T_{bloques} = (\frac{N}{P}B)(\frac{N}{B} + P - 1)t_c + (\frac{N}{B} + P - 2)(t_s + t_w B)$$

El primer término corresponde al cálculo de todos los bloques del proceso cero (N/B, en paralelo con todo el cómputo solapado de los bloques en otros procesos) más los bloques que se deben realizar para acabar de procesar todos los bloques por parte del resto de procesadores ($P - 1$). El segundo término corresponde a la comunicación que hay entre bloque y bloque a procesar. Estas comunicaciones se deben hacer entre cada bloque procesado ($N/B + P - 1$ bloques) menos para el último bloque. Cada comunicación es de B elementos.

Por lo que el tiempo total T_P, asumiendo que $P >>> 2$, es de:

$$\begin{aligned} T_p &\simeq (t_s + Nt_w) + (\frac{N}{P}B)(\frac{N}{B} + P)t_c + (\frac{N}{B} + P)(t_s + t_w B) \\ &= (t_s + Nt_w) + \frac{N^2}{P}t_c + NBt_c + t_s\frac{N}{B} + t_w N + t_s P + t_w PB \end{aligned}$$

Finalmente, se podría calcular el tamaño de bloque óptimo B_{opt} que hiciera minimizar el tiempo de ejecución T_P. Para ello, derivamos la fórmula anterior y, asumiendo que $N >>> P$, la igualamos a cero.

$$\begin{aligned} \frac{\partial T}{\partial B} &= Nt_c - t_s\frac{N}{B^2} + t_w P = 0 \\ B_{opt} &= \sqrt{\frac{t_s N}{Nt_c + t_w P}} = \sqrt{\frac{t_s}{t_c + t_w\frac{P}{N}}} \\ &\simeq \sqrt{\frac{t_s}{t_c}} \end{aligned}$$

Por lo que el tiempo paralelo óptimo T_{opt}* es de:

$$T_{opt} = t_s + Nt_w + (\frac{N^2}{P})t_c + 2N\sqrt{t_s t_c} + t_w N + t_s P + t_w P\sqrt{\frac{t_s}{t_c}}$$

*** El tiempo calculado es un tiempo teórico que seguramente se debería ajustar utilizando la bibliografía complementaria aconsejada para la realización de un modelo adaptado a la arquitectura del computador y del algoritmo.**

4. Principios de programación paralela

Cuando paralelizamos una aplicación, dos de los principales objetivos que se buscan son:

1) Mejorar el rendimiento de la aplicación, maximizando la concurrencia y reduciendo los costes adicionales de esta paralelización (con lo que maximizaremos el *speedup* obtenido), tal y como hemos analizado en el apartado anterior.

2) Productividad en el momento de programar: legibilidad, portabilidad, independencia de la arquitectura destino.

Para ello, se debe primero buscar una distribución adecuada del trabajo/datos de la aplicación (encontrar la concurrencia), después debemos elegir el esquema de aplicación paralela más adecuada (*Task parallelism*, *Divide and conquer*, etc.), finalmente, adaptar ese algoritmo a las estructuras de implementación conocidas (SPMD, *fork/join*, etc), y a los mecanismos de creación, sincronización y finalización de las unidades de procesamiento.

4.1. Concurrencia en los algoritmos

La idea es que, desde la especificación del problema original, podemos encontrar una distribución del problema para:

- Identificar tareas (partes del código que pueden ser ejecutadas concurrentemente), como hicimos, por ejemplo, en los dos casos de estudio del apartado anterior.

- Distribuir y analizar las estructuras de datos para saber cuáles son los datos que necesitan la tareas, los datos que producen las tareas, y qué datos se comparten. Con este análisis podemos intentar minimizar los movimientos de datos entre tareas, o los datos compartidos.

- Determinar las depedencias entre la tareas para intentar determinar el orden entre ellas y las sincronizaciones necesarias. Es decir, tenemos que determinar cuál es el grafo de dependencia de tareas y que el resultado final de nuestra paralelización sea el mismo que el del código secuencial.

Para encontrar la concurrencia/paralelismo en un algoritmo, primero debemos determinar si son los datos, las tareas, o el flujo de datos los que determinan este paralelismo. Dependiendo de lo que determinemos qué es más importante para distribuir el trabajo, haremos una distribución u otra:

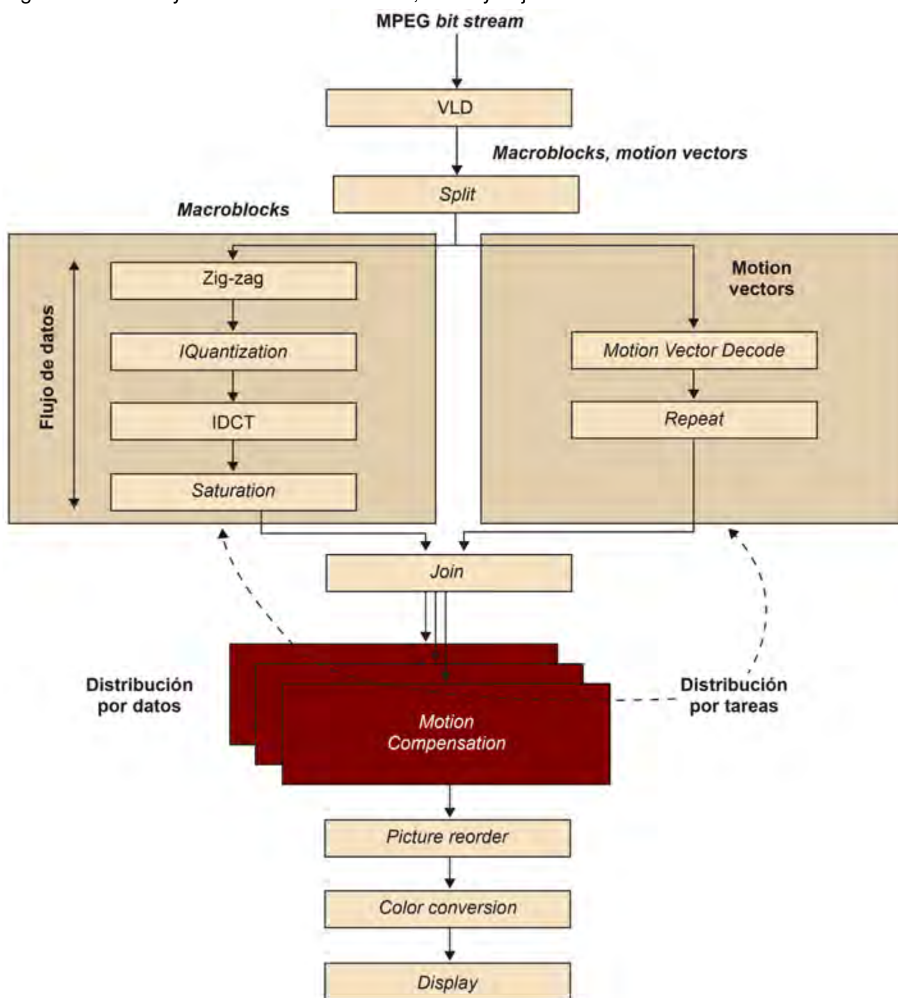
- *Data Decomposition*: en este caso, las estructuras de datos se particionan de tal forma que asignaremos una tarea a cada partición realizada.
- *Task Decomposition*: en este caso, se identifican partes del código como tareas, y esto implicará una distribución de los datos.
- *Data-Flow Decomposition*: en este caso, el flujo de los datos conllevará que unas tareas se activen y realicen el proceso de estos datos.

En cualquier caso, al final tendremos una serie de tareas, que posiblemente pueden tener dependencias de control y de datos entre ellas. Además, es posible que en un mismo problema tengamos los tres tipos de distribuciones tal y como mostramos en la figura 34. En esta figura mostramos cómo el programa de procesamiento de vídeo MPEG puede tener una distribución de trabajo por datos en la parte de *Motion*, distribución en tareas en la parte de macrobloques y vectores, y flujo de datos, entre las funciones que están dentro del procesamiento de los macrobloques.

Combinación de formas

Una misma aplicación puede combinar diferentes formas de distribuir el trabajo a realizar: *Data Decomposition*, *Task Decomposition* y *Data-Flow Decomposition*.

Figura 34. MPEG y la distribución en tareas, datos y flujo de datos.



4.1.1. Buenas prácticas

Una vez hemos identificado las distribuciones a realizar. Sería bueno seguir algunas buenas practicas cuando realicemos la distribución en tareas:

- **Flexibilidad:** ser flexibles en número y tamaño de las tareas generadas. Por ejemplo, el número de tareas no debería ser muy dependiente de la arquitectura específica elegida. Además, estas tareas sería bueno que fueran parametrizables.

- **Eficiencia:** pensar en la eficiencia de nuestro paralelismo, en el sentido de que cada una de estas tareas tenga el suficiente trabajo para poder amortizar la creación y la administración de estas tareas. Además, estas tareas deberían ser lo suficientemente independientes como para que el hecho de tener que gestionar estas dependencias no fuera un cuello de botella.

- **Simplicidad:** realizar una paralelización que sea fácil de mantener y de depurar si se diera el caso.

En cuanto a la distribución en datos, algunas guías de buenas prácticas son:

- **Flexibilidad:** ser flexibles en el número y tamaño de particiones* de los datos.

* `Chunks` en inglés

- **Eficiencia:** pensar también en la eficiencia de nuestra paralelización como resultado de esta distribución. Por ejemplo, deberíamos ver si este particionamiento de los datos significa tener o no desbalanceo en la carga de trabajo de las tareas. Además, en el caso de distribución por datos, suele considerarse la arquitectura de nuestro computador para poder tener en cuenta la jerarquía de memoria de éste.

- **Simplicidad:** realizar una distribución no muy compleja, ya que en caso contrario la depuración del programa puede ser difícil.

4.1.2. Orden y sincronización de tareas

Una vez enunciadas estas buenas prácticas, el primer paso que tenemos que efectuar tras realizar la distribución en tareas es analizar cuál es el orden entre éstas, y además, cuáles son las restricciones de compartición de datos de estas tareas. En caso de tener algún tipo de restricción de orden o de compartición de datos, tendremos que ver qué mecanismos de sincronización se deben usar. También puede ser que nos encontremos que haya una paralelización donde no se tenga que realizar ningún tipo de sincronización, siendo todas la tareas completamente independientes, en este caso estamos hablando de un paralelismo embarazoso. Esto no quita que tengamos que tener en cuenta aspectos de balanceo de carga o de localidad de datos que pueden traducirse en una paralelización poco eficiente del programa.

Paralelismo embarazoso

Embarransingly parallel o paralelismo embarazoso es cuando la paralelización de un programa se puede realizar de tal forma que las tareas no necesitan ningún tipo de comunicación o sincronización entre ellas. El hecho de que tengamos un paralelismo embarazoso no quita que podamos tener problemas de desbalanceo de carga o de localidad de datos.

Para conseguir la sincronización entre tareas, podemos secuencializar la ejecución de estas tareas, o bien realizar algún tipo de sincronización global. En el apartado de modelos de programación paralela veremos ejemplos de cómo puede el programador realizar esta sincronización en un sistema de memoria compartida y distribuida. También veremos un ejemplo donde el programador deja a la librería de gestión de recursos y tareas del modelo de programación* que gestione las sincronizaciones entre tareas según las dependencias de unas con respecto a las otras, indicadas por el programador.

* ***Runtime*** **en inglés**

4.1.3. Compartición de datos entre tareas

En lo que respecta a la compartición de datos en memoria compartida, todos *threads* tienen acceso a todos los datos en la memoria compartida. Por consiguiente, deben realizar algun tipo de sincronización en el acceso compartido pare evitar *Race conditions* o condiciones de carrera, que veremos en el siguiente subapartado. En el caso de memoria distribuida, cada procesador tiene los datos a los que puede acceder en su memoria local, por lo que no puede haber condiciones de carrera. En cambio, debe haber una comunicación explícita para poder compartir los datos.

Para la memoria compartida, una forma de evitar las *Race conditions* es usando zonas de exclusión mutua en las secciones críticas. Una sección crítica es una secuencia de instrucciones en una tarea que pueden entrar en conflicto con una secuencia de instrucciones en otra tarea, creando una posible condición de carrera. Para entrar en conlicto dos secuencias de instrucciones, una de ellas, al menos, debe modificar algun dato, y otra leerlo. Una zona de exclusión mutua es un mecanismo que asegura que solo una tarea a la vez ejecuta el código que se encuentra en una sección crítica.

En el momento de hacer una exclusión mutua nos podemos encontrar que hemos provocado un *deadlock*, que explicaremos en el siguiente subapartado. Además, nos podemos encontrar con un problema de rendimiento si realizamos un gran número de exclusiones mutuas o bien el tamaño de código que abarcan es extenso.

deadlock

Un *deadlock* se produce cuando una tarea necesita y espera un recurso que tiene otra tarea, al mismo tiempo que esta última necesita y espera un recurso de la primera.

En el apartado dedicado a los modelos de programación veremos ejemplos de cómo realizar zonas de exclusión mutua en el caso de memoria compartida.

Ved también

Los modelos de programación se tratan en el apartado 5.

4.2. Problemas que aparecen en la concurrencia

Cuando aparece la concurrencia en el cálculo o comunicación de una serie de acciones nos podemos encontrar con cuatro problemas típicos, que comentamos a continuación.

4.2.1. *Race Condition*

Race Condition o condición de carrera: múltiples tareas leen y escriben un mismo dato cuyo resultado final depende del orden relativo de su ejecución. La forma de solucionar este problema es forzar algún mecanismo para ordenar/sincronizar los accesos a estos datos.

Un ejemplo típico del problema de *Race Condition* es extraer dinero de una misma cuenta de un banco desde dos cajeros distintos. En este caso, si el banco no ofrece un mecanismo para sincronizar los accesos a la cuenta en cuestión, nos podríamos encontrar con la situación de que dos personas podrían sacar tanto dinero como el dinero que disponemos en un cierto momento en nuestra cuenta. Esto implicaría que podríamos sacar hasta el doble de la cantidad que se disponía en la cuenta.

4.2.2. *Starvation*

Starvation o muerte por inanición: una tarea no puede conseguir acceder a un recurso compartido y, por consiguiente, no puede avanzar. Este problema es más difícil de solucionar, y normalmente se suele hacer con mecanismos de control del tiempo que se pasa en el bloqueo. Una situación en la que nos podríamos encontrar en *Starvation* es justamente en la situación anterior de los cajeros. Imaginemos que una de las personas accede a la cuenta desde un cajero. Otra persona accede desde otro cajero y se queda bloqueada mientras que la primera persona se piensa los movimientos que quiere realizar. En este caso, la persona que se ha quedado bloqueada puede mantenerse a la espera por un tiempo indeterminado, a no ser que decida irse y probarlo en otro momento.

4.2.3. *Deadlock*

Deadlock o condición de bloqueo: dos o más tareas no pueden continuar porque unos están esperando que los otros hagan algo. Una forma de evitar condiciones de bloqueo es mediante una ordenación adecuada de las sincronizaciones que provocan este bloqueo. Por ejemplo, pensemos en la transferencia de dinero de una cuenta de una persona a la cuenta de otra persona. En esta transferencia debemos bloquear la cuenta de la persona origen y la cuenta de la persona destino, para poder actualizar el dinero de ambas cuentas. Si se diera el caso que la persona de la cuenta destino de la anterior transferencia intenta a su vez, y en paralelo, realizar una transferencia desde su cuenta a la cuenta de la primera persona, podríamos encontrar una condición de bloqueo. Esto es debido a que cada persona, por separado, bloquea primero su cuenta, y después intenta el bloqueo de la cuenta destino. En el momento de intentar bloquear la cuenta destino, ambas personas se quedan bloqueadas, ya que se ha forzado un ciclo en los bloqueos.

Una forma de evitar este problema, para esta situación concreta, es ordenar los bloqueos a realizar según un criterio que haga que las dos personas intenten bloquear primero la **misma** cuenta, y después la otra cuenta. Así, una persona podrá hacer la transferencia y la otra se quedará bloqueada en el primer intento de bloquear una cuenta.

4.2.4. *Livelock*

Livelock o condición de bloqueo pero con continuación: dos o más tareas cambian continuamente de estado en respuesta de los cambios producidos en otras tareas pero no logran realizar ningún trabajo útil.

El ejemplo típico es el de los 5 filósofos sentados a una mesa redonda. Entre cada dos filósofos contiguos hay un cubierto. Cada filósofo hace dos cosas: pensar y comer. Así, cada filósofo piensa durante un rato, y cuando los filósofos tienen hambre, paran de pensar y, en este orden, cogen el cubierto de la izquierda y después el de la derecha. Si por lo que fuera, uno de esos cubiertos lo hubiera cogido ya un filósofo, entonces deberá esperar hasta poder disponer de los dos cubiertos. Cuando un filósofo tiene los dos cubiertos, entonces puede comer. Una vez que ha comido, puede volver a dejar los cubiertos y continuar pensando.

En el caso de que todos los filósofos cojan un cubierto a la vez, todos los filósofos tendrán un cubierto en una mano y deberán esperar a tener el otro cubierto con tal de poder comer. Con lo cual, estamos en una situación de bloqueo *deadlock*. Para evitar el problema de *deadlock* podemos hacer que cada filósofo deje el cubierto, espere 5 minutos y después lo intente otra vez. Como podemos observar, los filósofos cambiarán de estado entre bloqueo y no bloqueo, pero no lograrán comer. En este caso están en un *livelock*.

Una posible solución es hacer lo mismo que hacíamos para solucionar el problema del *deadlock*: determinar un orden en el momento de coger los cubiertos que haga que los filósofos no se puedan quedar bloqueados.

4.3. Estructura de los algoritmos

Según el tipo de tareas concurrentes, y en función del tipo de distribución que hayamos realizado (en tareas o en datos) podemos clasificar las estrategias de paralelización según su patrón.

Según la distribución realizada tenemos:

- Distribución en tareas: patrones *Task Parallelism* y *Divide & Conquer*.
- Distribución en datos: patrones linear o *Geometric Decomposition* y *Recursive Decomposition*.
- Distribución por flujo de datos: patrones *Pipeline* y *Event Based*.

4.3.1. Patrón *Task Parallelism*

El problema se puede distribuir en una colección de tareas que se pueden ejecutar de forma concurrente. Estas tareas y sus dependencias pueden ser identificadas mediante un análisis del código. Un ejemplo de *Task Parallelism* es el conjunto de iteraciones de un bucle que se pueden ejecutar en paralelo. De alguna forma estas tareas están linearmente distribuidas.

Ved también

En el subapartado 5.5 realizaremos un ejemplo de aplicación del patrón *Task Parallelism* utilizando el modelo de programación de memoria compartida OpenMP.

4.3.2. *Patrón Divide & Conquer*

Este patrón es consecuencia de una solución recursiva (con estrategia *Divide & Conguer*) secuencial a un determinado problema. En este caso no hay una distribución linear. Con esta estrategia un problema se divide en subproblemas, y así sucesivamente, hasta llegar a un caso base donde se corta la recursividad.

> **Ved también**
>
> En el subapartado 5.5 realizaremos un ejemplo de aplicación del patrón *Divide & Conquer* utilizando el modelo de programación de memoria compartida OpenMP.

Podemos considerar dos estrategias en el momento de paralelizar la recursividad exsitente:

1) Sólo las hojas del árbol de recursividad se realizarán en paralelo.

2) El árbol de recursividad también se realiza en paralelo.

4.3.3. Patrón *Geometric Decomposition*

Este patrón es para distribuciones de datos en los que los datos a tratar son vectores u otras estructuras de datos lineales (no recursivos). Para estos tipos de datos, el problema normalmente se reduce a distribuir las tareas para que traten subestructuras de los datos, de la misma forma que dividimos regiones geométricas en subregiones. Normalmente para este caso usamos el patrón *Task Parallelism* (*Task Decomposition*).

> **Ved también**
>
> En el subapartado de estimación de tiempo de ejecución en paralelo (subapartado 3.5) desarrollamos dos ejemplos de patrón *Geometric Decomposition*.

En este tipo de patrón es normal que se tenga que acceder a datos de alguna de sus tareas vecinas.

4.3.4. Patrón *Recursive Data*

El patrón *Recursive Data* es un tipo de patrón para distribución de datos donde las estructuras de datos son estructuras de datos recursivas, como por ejemplo, grafos, árboles, etc.

Normalmente en estos casos se usa una opción recursiva para encontrar una solución, es decir, usando un patrón *Divide & Conquer*.

4.3.5. Patrón *Pipeline*

Este tipo de patrón consiste en el procesamiento por etapas de cada uno de los datos que pertenecen a un conjunto que se tienen que procesar. Estas etapas están bien definidas y están conectadas unas con otras para ir procesando cada dato. Una característica importante es que solo hay un sentido y no se producen ciclos.

La equivalencia entre etapa y tarea dependerá de cómo queramos hacer la agrupación de tareas, y la granularidad de nuestras etapas.

En este tipo de patrón es necesario tener algún tipo de control del último dato a tratar y de los posibles errores que se tengan que propagar, y es bueno incorporar flexibilidad para poder sumar nuevas etapas, etc. Un ejemplo de *pipeline* es una cadena de fabricación de coches.

4.3.6. Patrón *Event-based Coordination*

Es una clase de *pipeline* pero irregular. No hay una secuencia de etapas una detrás de otra, conectadas de dos en dos. Puede haber ciclos y puede haber dos sentidos en la conexión entre dos nodos del grafo de tareas. Además, no hay predicción de cómo irá evolucionando la comunicación entre los nodos.

En este tipo de patrón es muy importante poder controlar los casos de *deadlocks*, la ordenación de las tareas en algun sentido para cumplir con los requisitos de orden, etc.

4.4. Estructuras de soporte

Para poder implementar todos estos patrones, normalmente se trabaja con unos esquemas de programación que básicamente son el esquema *Single Program Multiple Data* (SPMD), el esquema *Fork/Join*, el esquema *Master/Workers* y finalmente el esquema *Loop Parallelism*. En cualquier caso, es posible que dos o más esquemas se puedan combinar, como por ejemplo un patrón *Master/Worker* podría ser implementado con un esquema SPMD o *Fork/Join*.

Patrones de programación

Los patrones de programación SPMD, *Fork/Join*, *Master/Workers* y *Loop Parallelism* se pueden combinar en la implementación de un programa.

4.4.1. SPMD

En este esquema, todos los *threads*/procesos o unidades de ejecución (UE) ejecutan, básicamente, el mismo programa, pero sobre diferentes datos. Aunque todos ejecutan el mismo programa pueden variar en el camino de ejecución seguido, según el identificador de la UE (por ejemplo, el identificador de proceso o el identificador de *thread*), al estilo del código 4.1.

```
1   ...
2   if (my_id==0)
3   {
4       /* Master */
5   }
6   else
7   {
8       /* Worker */
9   }
10  ...
```

Código 4.1: Ejemplo básico de un esquema SPMD

4.4.2. *Master/Workers*

Un proceso/*thread* `master` crea un conjunto de *threads*/procesos para que realicen una bolsa de tareas. Este conjunto de *threads* o procesos van tomando tareas de la bolsa de trabajo y las realizan hasta que no queden más en la bolsa de tareas.

En el caso de que las tareas se generasen de forma dinámica y/o bien el número total de tareas no se conociera a priori, sería necesario algún mecanismo para indicar a los *threads*/procesos que no hay más tareas a realizar.

4.4.3. *Loop Parallelism*

Este esquema es típico de programas secuenciales con bucles de cómputo intensivo. En este caso, muchas veces hacemos que las iteraciones del bucle se puedan ejecutar en paralelo.

4.4.4. *Fork/Join*

El esquema *Fork/Join* consiste en que un proceso o *thread* principal creará un conjunto de procesos/*threads* para realizar una porción de trabajo, y normalmente se espera a que estos acaben.

5. Modelos de programación paralela

A continuación haremos una breve explicación de MPI y OpenMP, que son los modelos de programación más aceptados para memoria distribuida y memoria compartida, respectivamente. También mostraremos algunas pinceladas de una extensión del modelo OpenMP (OpenMP Superescalar, OmpSs) con tal de poder realizar control de dependencias entre tareas en tiempo de ejecución. Para profundizar en estos modelos de programación paralela aconsejamos la lectura de la bibliografía básica.

5.1. Correspondencia entre modelos y patrones

Para desarrollar cada uno de los esquemas algorítmicos vistos en el apartado anterior disponemos de los modelos de programación paralela. Los modelos de programación nos ofrecen una API para poder paralelizar nuestros algoritmos tanto para memoria compartida como para memoria distribuida.

La tabla 3 muestra qué estructura de soporte se adecúa normalmente mejor a cada uno de estos modelos de programación paralela.

Tabla 3. Tabla de relación entre los esquemas de soporte para estos algoritmos paralelos y los modelos de programación paralela.

	OpenMP	MPI
SPMD	***	****
Loop Parallelism	****	*
Master/Worker	**	***
Fork/Join	***	

Además, en la tabla 4 se muestra una relación de idoneidad entre los esquemas de soporte y las estructuras de algoritmos paralelos. Con estas relaciones se intenta dar orientaciones de cómo implementar mejor la paralelización de los códigos.

Tabla 4. Relación de algoritmo de soporte y estrategia de paralelización.

	Task Paralellism	Divide and Conquer	Geometric Decomposition	Recursive Data	Pipeline	Event-Based Coordination
SPMD	****	***	****	**	***	**
Loop Parallelism	****	**	***			
Master/Worker	****	**	*	*	*	*
Fork/Join	**	****	**		****	****

5.2. MPI

MPI (*Message Passing Interface*) es un modelo de programación estandar para memoria distribuida. Este estandar incluye mecanismos para comunicarse entre procesos con

mensajes punto a punto* pero también para comunicaciones colectivas**, además de mecanismos de sincronización.

* ***Point to point*** **en inglés**
** ***Collective communications*** **en inglés**

MPI fue creado a comienzos de la década de 1990 para ofrecer un entorno de ejecución paralelo con paso de mensajes para clusters, MPP, e incluso máquinas de memoria compartida. Hay muchas implementaciones de MPI, como por ejemplo Open MPI (que incorpora otras implementaciones conocidas como la LAM/MPI, LA-MPI y FT-MPI) o MPICH, que soportan casi todo tipo de máquinas paralelas, incluyendo SMPs, máquinas NUMA, y cluster Linux.

Este subapartado se organiza de la siguiente manera: primero realizaremos un ejemplo sencillo, después pasaremos a realizar ejemplos de comunicación punto a punto, para acabar con ejemplos de colectivas.

5.2.1. Un programa simple

El código 5.1 muestra un "hola mundo" escrito en MPI.

```
1  #include <mpi.h>
2
3  int rank;
4  int nproc;
5
6  int main( int argc, char* argv[] ) {
7      MPI_Init( &argc, &argv );
8      MPI_Comm_size( MPI_COMM_WORLD, &nproc );
9      MPI_Comm_rank( MPI_COMM_WORLD, &rank );
10
11     /* Nothing to do */
12     printf("Proceso: %d de un total de %d: Hola mundo!\n",rank,nproc);
13
14     MPI_Finalize();
15 }
```

Código 5.1: Programa Simple de "Hola Mundo" en `MPI`.

Lo primero que observamos en el programa es que necesitamos incluir la cabecera `mpi.h`, con tal de tener el prototipo de las llamadas de MPI.

Posteriormente, todos los programas MPI deben realizar una llamada, y solo una, a `MPI_Init( &argc, &argv )`. Ésta inicializa el entorno MPI, pudiendo este entorno, de forma transparente al programador, introducir nuevos argumentos a los argumentos del programa. La llamada se debe hacer antes de realizar cualquier otra llamada a alguna función MPI.

Las siguientes dos llamadas, `MPI_Comm_size` y `MPI_Comm_rank` no son llamadas obligatorias pero normalmente todas las aplicaciones MPI las hacen. La primera de ellas devuelve el número de procesos que están dentro del *communicator MPI_COMM_WORLD*. Este *communicator* se crea por defecto, pero podríamos crear otros, lo que queda fuera de

Communicators

El uso de *communicators* sería idóneo si se quiere desarrollar una librería que use MPI y que no interfiera con el código de usuario.

los objetivos de esta asignatura. Un *communicator* tiene como objetivo crear un contexto para operaciones de comunicación de un grupo de procesos. De esta forma, los mensajes que se envían en un determinado contexto solo van a parar a procesos dentro de este contexto. Por consiguiente, los mensajes no pueden interferir con mensajes en otro contexto. Los procesos que forman parte de un *communicator* reciben el nombre de *process group*.

Por otro lado, `MPI_Comm_rank` devuelve el identificador del proceso dentro del *process group* asociado al *communicator*. Este identificador (variable `rank` en el código) va desde `0` hasta `nproc-1`.

Una vez llegados a este punto, cada proceso realiza su “hola mundo”, y todos deben acabar con una llamada a `MPI_Finalize()`, que debería ser la última llamada a una función MPI. Por otra parte, en algunas ocasiones queremos abortar el programa antes de llegar a `MPI_Finalize()`. Esto se puede hacer llamando a `MPI_Abort()`.

Otro programa simple que nos puede interesar es el de ver cuánto tiempo tardan en ejecutar una parte del programa en paralelo. Para poderlo hacer desde un mismo punto en el tiempo para todos los procesos, debemos sincronizarlos. Esto será necesario si no hay ya alguna comunicación que los sincronice.

En el código 5.2 vemos un ejemplo de cómo sincronizar con la llamada `MPI_Barrier()` todos los procesos. Con esta llamada conseguimos que cada proceso se bloquee en ella hasta que todos los procesos en el *communicator* lleguen.

```
1  #include <mpi.h>
2
3  int rank;
4  int nproc;
5
6  int main( int argc, char* argv[] ) {
7      MPI_Init( &argc, &argv );
8      MPI_Comm_size( MPI_COMM_WORLD, &nproc );
9      MPI_Comm_rank( MPI_COMM_WORLD, &rank );
10
11     MPI_Barrier(MPI_COMM_WORLD);
12     /* Comienza el Timing*/
13         /* TRABAJO A REALIZAR */
14     /* Finaliza el Timing*/
15     MPI_Finalize();
16 }
```

Código 5.2: Ejemplo de sincronización con código MPI

Finalmente, para realizar la compilación de un programa MPI se suele hacer con un *script* llamado *mpicc* que viene con el paquete de instalación de *MPI*. *mpicc* llama al compilador que hay por defecto en el sistema, utilizando los *includes* y librerías necesarias.

Paquetes de instalación

Dependiendo del paquete MPI y del entorno de ejecución que nos instalemos, podríamos tener otras formas de ejecutar los programas, como por ejemplo: *mpiexec*, *srun*, etc.

Para ejecutar el binario obtenido de la compilación se utiliza el comando:

```
1    mpirun -np <número_de_procesos> programa_mpi
```

Con la opción `-np <num>` se indica cuántas instancias queremos que se ejecuten del programa.

5.2.2. Comunicación punto a punto

Las comunicaciones punto a punto en MPI consisten básicamente en un mensaje de un proceso a otro, identificado por <`tag`, destino del mensaje, `communicator`>. Las comunicaciones punto a punto más usadas son:

- Comunicaciones punto a punto de tipo *blocking*: cuando volvemos de la llamada a una comunicación de este tipo es porque podemos usar cualquiera de los recursos que había en la llamada.

- Comunicaciones punto a punto de tipo *non-blocking*: en este caso es posible que se vuelva de la llamada antes de que se haya completado la operación. Por consiguiente, es posible que los recursos que se necesitaban en la llamada todavía no puedan ser usados.

Comunicaciones *blocking*

El código 5.3 muestra un ejemplo con comunicaciones de tipo *blocking*. En el ejemplo suponemos que hay solo dos procesos.

```
1  #include "mpi.h"
2
3  int rank, nproc;
4
5  int main( int argc, char* argv[] ) {
6      int isbuf, irbuf;
7      MPI_Status status;
8
9      MPI_Init( &argc, &argv );
10     MPI_Comm_size( MPI_COMM_WORLD, &nproc );
11     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
12
13     if(rank == 0) {
14         isbuf = 9;
15         MPI_Send( &isbuf, 1, MPI_INTEGER, 1, 1, MPI_COMM_WORLD);
16     } else if(rank == 1) {
17         MPI_Recv( &irbuf, 1, MPI_INTEGER, 0, 1, MPI_COMM_WORLD,
18                          &status);
19         printf( "%d\n", irbuf );
20     }
21     MPI_Finalize();
22 }
```

Código 5.3: Ejemplo de comunicación punto a punto de tipo `blocking`.

El proceso con identificador lógico $rank = 0$ realiza un envío bloqueante de un mensaje con MPI_Send. La sintaxis del MPI_Send se describe en el código 5.4. Los parámetros,

de arriba abajo, indican el *buffer* a enviar, el número de elementos del *buffer*, el tipo de datos de cada elemento, el identificador lógico del procesador destino, el identificador del mensaje (`tag`), y el *communicator*.

```
1  int MPI_Send( void* buf,              /* in */
2                int count,              /* in */
3                MPI_Datatype datatype,  /* in */
4                int destination,        /* in */
5                int tag,                /* in */
6                MPI_Comm comm );        /* in */
```

Código 5.4: Sintáxis del `MPI_Send`

Los tipos básicos de los elementos que se pueden comunicar son:

- MPI_CHAR
- MPI_SHORT
- MPI_INT
- MPI_LONG
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED_SHORT
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG_DOUBLE
- MPI_BYTE
- MPI_PACKED

Los mensajes enviados con MPI_Send los puede recibir cualquier proceso ya sea con la llamada a MPI_Recv o bien llamada MPI_Irecv.

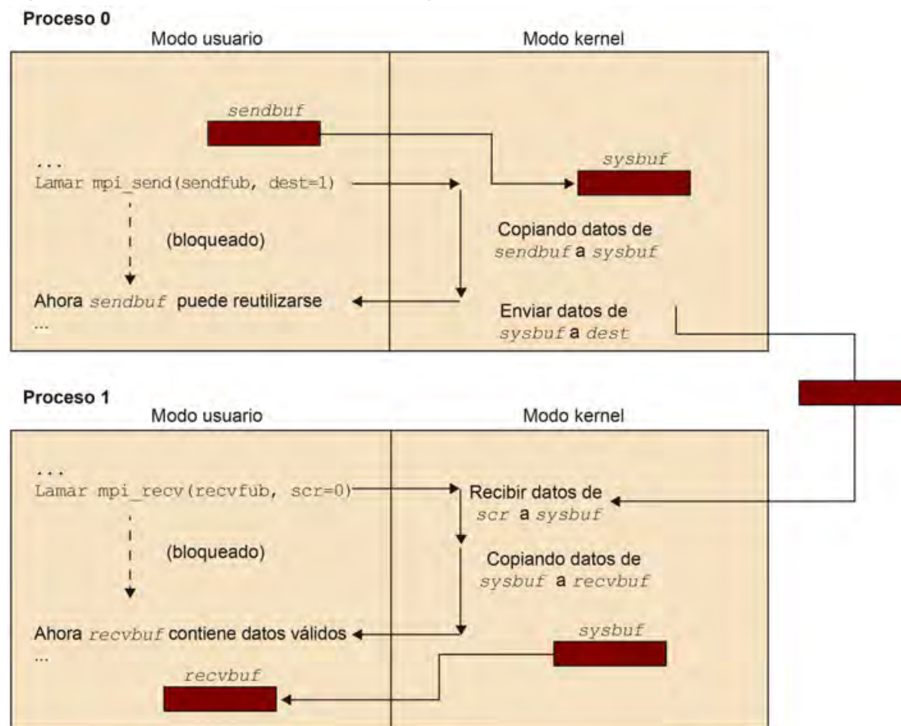
El proceso con identificador lógico $rank = 1$ realiza una recepción bloqueante con MPI_Recv, cuya sintaxis se muestra en el código 5.5. Los parámetros indican, de arriba abajo, el *buffer* de recepción, el número de elementos a recibir (debe ser menor o igual que la capacidad del *buffer* de recepción), el tipo de datos de cada elemento, el identificador lógico del procesador que envía el mensaje, el identificador del mensaje (`tag`), el *communicator*, y una variable para saber cómo ha ido el mensaje. El mensaje recibido puede que lo haya enviado un proceso con MPI_Send o con MPI_Isend.

```
1  int MPI_Recv( void* buf,              /* out */
2                int count,              /* in */
3                MPI_Datatype datatype,  /* in */
4                int source,             /* in */
5                int tag,                /* in */
6                MPI_Comm comm,          /* in */
7                MPI_Status* status );   /* out */
```

Código 5.5: Sintáxis del `MPI_Recv`

La figura 35 muestra gráficamente cuándo un MPI_Send y un MPI_Recv dejan de estar bloqueados.

Figura 35. Fases en la comunicación *blocking*.



Comunicaciones *non-blocking*

Veamos ahora un ejemplo de comunicaciones *non-blocking* con el código 5.6.

```
1  #include "mpi.h"
2
3  int main( int argc, char* argv[] )
4  {
5      int rank, nproc;
6      int isbuf, irbuf, count;
7      MPI_Request request;
8      MPI_Status status;
9
10     MPI_Init( &argc, &argv );
11     MPI_Comm_size( MPI_COMM_WORLD, &nproc );
12     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
13
14     if(rank == 0) {
15         isbuf = 9;
16         MPI_Isend( &isbuf, 1, MPI_INTEGER, 1, 1,
17                    MPI_COMM_WORLD, &request );
18     } else
19     if(rank == 1) {
20         MPI_Irecv( &irbuf, 1, MPI_INTEGER, 0, 1,
21                    MPI_COMM_WORLD, &request);
22
23         /* Trabajo solapado con la comunicación */
24
25         MPI_Wait(&request, &status);
26         MPI_Get_count(status, MPI_INTEGER, &count);
```

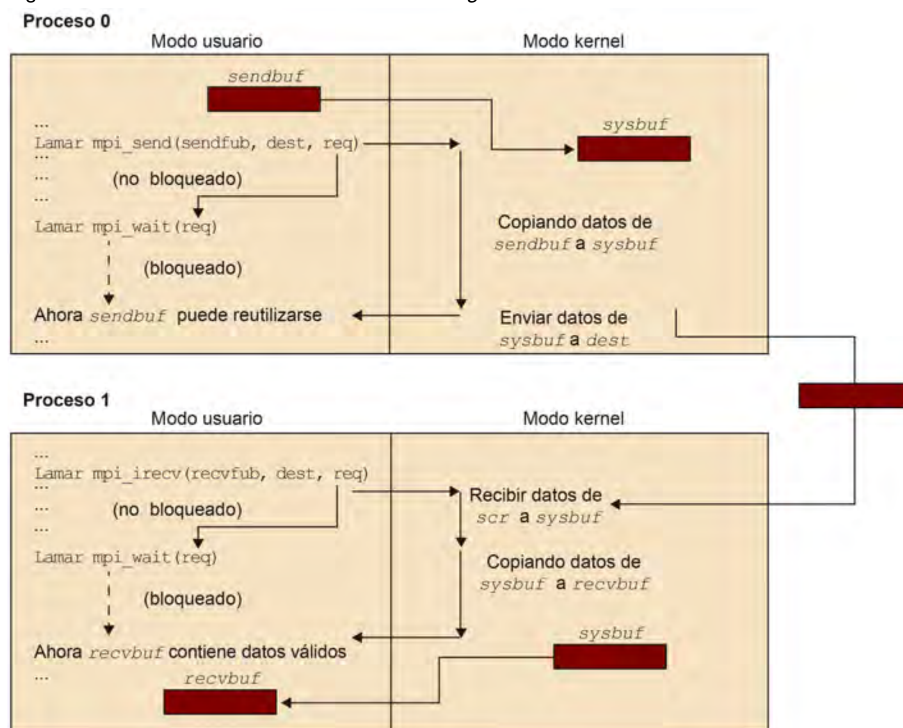
```
27          printf( "irbuf = %d source = %d tag = %d count = %d\n", irbuf,
                status.MPI_SOURCE,
28                  status.MPI_TAG, count);
29      }
30      MPI_Finalize();
31  }
```

Código 5.6: Ejemplo de comunicación punto a punto `non-blocking`.

La primera diferencia que observamos con respecto a las comunicaciones *blocking* es que las llamadas a MPI_Isend y MPI_Irecv tienen un parámetro más de tipo MPI_Request. Este parámetro nos permitirá saber el estado de la comunicación en una consulta posterior. En el ejemplo, el proceso con $rank = 1$ ha realizado una comunicación *non-blocking* para recibir el mensaje. Mientras que recibe el mensaje, éste puede realizar trabajo de cómputo, hasta que decide esperarse a que acabe la comunicación que realizó para poder proceder con el programa. La forma de esperarse a esta comunicación es mediante la llamada `MPI_Wait`. En esta llamada le pasamos la variable *request* y además una variable de tipo `MPI_Status` para poder tener información del mensaje recibido. En el caso de que el MPI_Irecv se hubiera hecho para recibirlo de cualquier proceso (identificador proceso origen igual a `MPI_ANY_SOURCE`) y/o con cualquier *tag*, es decir `MPI_ANY_TAG`, la variable *status* guardaría la información real de quien lo ha enviado (`status.MPI_SOURCE`) y con qué *tag* (`status.MPI_TAG`).

La figura 36 muestra gráficamente cómo un MPI_Isend y un MPI_Irecv vuelven inmediatamente después de realizar las llamadas, y posteriormente, cuando se llama a MPI_Wait puede ser que se tenga que bloquear hasta que la operación esté acabada.

Figura 36. Fases en la comunicación *non-blocking*.



También existe la posibilidad de consultar el estado de la comunicación sin quedarse bloqueado. Esto se puede realizar con la llamada MPI_Test. A esta llamada se le pasan la variable `request`, un puntero a variable entera (`flag`) y la variable `status` para saber cómo fue la comunicación. En caso de que `flag` adquiera el valor `MPI_SUCCESS`, eso significa que la comunicación ha acabado.

Finalmente, en el programa también hay una llamada a MPI_Get_count, que nos ayuda a saber cuál es la cantidad de datos que se comunican en la comunicación asociada a la variable `status`.

5.2.3. Comunicación colectiva

En este tipo de comunicaciones todos los procesos del *process group* invocan la comunicación. En la programación con MPI es muy común ver programas que únicamente hacen comunicaciones colectivas. A continuación describiremos brevemente un subconjunto de estas colectivas, exponiendo ejemplos para alguna de ellas.

Colectivas uno a todos y todos a uno

Unas de las colectivas más usadas son las de comunicación de uno a todos (`MPI_Bcast`) y la de todos a uno con reducción (`MPI_Reduce`). El código 5.7 muestra un ejemplo para el cálculo de `pi`.

```
1
2 #include <mpi.h>
3 void main (int argc, char *argv[])
4 {
5   int i, my_id, numprocs, num_steps;
6   double x, pi, step, sum = 0.0 ;
7
8   MPI_Init(&argc, &argv) ;
9   MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
10  MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
11
12  if (my_id==0) scanf(¿%d¿,&num_steps);
13
14  MPI_Bcast(&num_steps, 1, MPI_INT, 0, MPI_COMM_WORLD)
15  step = 1.0/(double) num_steps ;
16  my_steps = num_steps/numprocs ;
17
18  for (i=my_id*my_steps; i<(my_id+1)*my_steps; i++)
19  {
20    x = (i+0.5)*step;
21    sum += 4.0/(1.0+x*x);
22  }
23  sum *= step ;
24  MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE,
25             MPI_SUM, 0, MPI_COMM_WORLD) ;
26  MPI_Finalize() ;
27 }
```

Código 5.7: Ejemplo de comunicación colectiva con `MPI_Bcast` y `MPI_Reduce`.

Todos hacen la llamada a `MPI_Bcast`, pero uno de los argumentos indica qué proceso hace de emisor, mientras que el resto recibirá la información. En el ejemplo, el proceso `master` (`rank==0`) comunica al resto de procesos el número de pasos que deben realizar en la aproximación de `pi`. En el caso del `MPI_Reduce` sucede lo contrario, todos los procesos envían un dato de tipo doble al proceso cero, también indicado por un argumento. En este caso, todos los procesos aportan su cálculo parcial de `pi`, guardado en la variable `sum`, y se reduce, en el proceso `master` (`rank==0`), haciéndose la suma de todos los valores y guardándolo en la variable `pi`.

La sintaxis del `MPI_Bcast` se muestra en el código 5.8. El primer parámetro `buffer` indica al vector de `count` datos de tipo *datatype* que se van a comunicar a todos desde el procesador `root`. La comunicación la hacen dentro del comunicador `comm`.

```
1  int MPI_Bcast( void* buffer,           /* inout */
2                 int count,              /* in */
3                 MPI_Datatype datatype,  /* in */
4                 int root,               /* in */
5                 MPI_Comm comm);         /* in */
```

Código 5.8: Sintáxi del `MPI_Bcast`

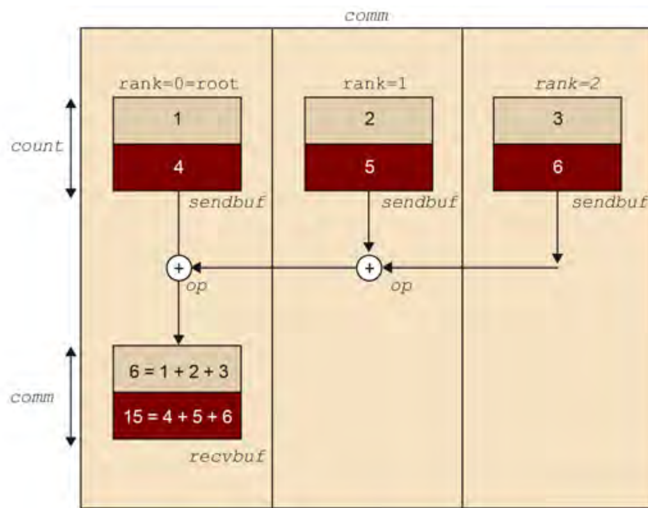
La sintaxis del `MPI_Reduce` se muestra en el código 5.9. La función aplica una operación de reducción determinada por `op` al vector `sendbuf` sobre un conjunto de procesos que están dentro del comunicador `comm`. El resultado lo deja en el vector `recvbuf` en el proceso `root`.

```
1  int MPI_Reduce( void* sendbuf,          /* in */
2                  void* recvbuf,          /* out */
3                  int count,              /* in */
4                  MPI_Datatype datatype,  /* in */
5                  MPI_Op op,              /* in */
6                  int root,               /* in */
7                  MPI_Comm comm);         /* in */
```

Código 5.9: Sintáxi del `MPI_Reduce`

En lo que hace referencia a la reducción, si lo que queremos reducir es un vector de elementos, entonces lo que se efectuará es la reducción del valor de cada posición del vector, tal y como muestra la figura 37.

Figura 37. Operación colectiva `MPI_Reduce` aplicada a un vector de elementos con una operación de `MPI_SUM`.

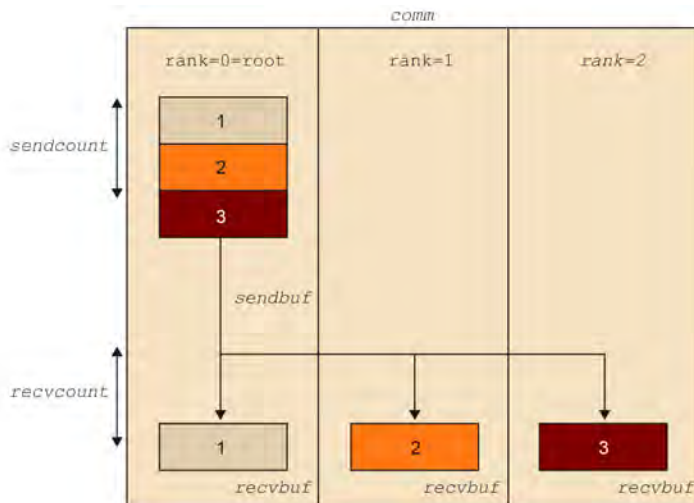


Las operaciones permitidas en la reducción son: `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, `MPI_MIN`, `MPI_MAXLOC`, `MPI_MINLOC`, `MPI_LAND`, `MPI_LOR`, `MPI_LXOR`, `MPI_BAND`, `MPI_BOR`, `MPI_BXOR`, y también otras operaciones que pueden ser definidas por el programador.

Colectivas de repartición y agrupación de datos

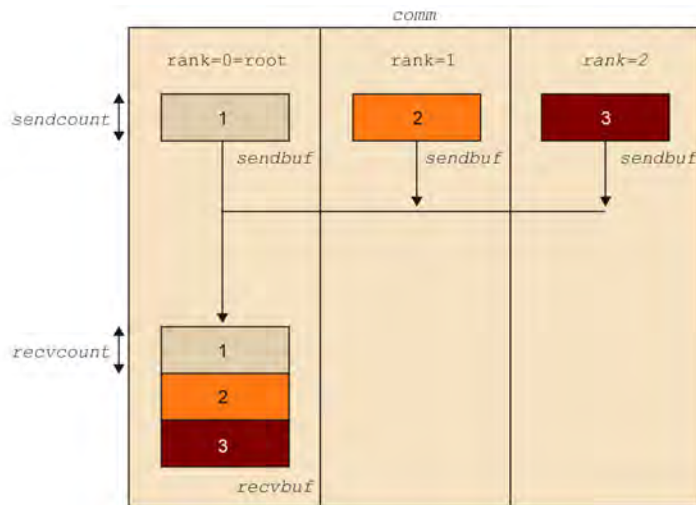
Otras colectivas muy usadas son las *scatter* y *gatther* para la repartición y recogida de datos. La figura 38 muestra la forma en la que un procesador realiza el *scatter* de un vector de tres enteros sobre tres procesos (éste incluido).

Figura 38. Operación colectiva *scatter*.



La figura 39 muestra el *gather* de tres enteros en tres procesadores.

Figura 39. Operación colectiva *gatther*.



En el código 5.10 vemos un ejemplo de dónde se hace el *scatter* de un vector de *nproc**100* elementos, siendo 100 elementos para cada proceso. El proceso `root` envía los elementos que tiene en el `rootbuf` y los procesos, incluido el proceso `root`, lo recibirán en el `localbuf`. Posteriormente se realiza algún trabajo y se hace el *gatther* de todos los `localbuf` sobre el `rootbuff` del proceso `root`.

```
int gsize, localbuf[100];
int root=0, rank, *rootbuf;

...

MPI_Comm_size( MPI_COMM_WORLD, &nproc );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
if (rank == root)
  rootbuf = (int *)malloc(nproc*100*sizeof(int));
  /* MATRIX INITIALIZED IN ROOT */
  MPI_Scatter (rootbuf, 100, MPI_INT, localbuf, 100,
               MPI_INT, root, comm);

  /* DO WORK WITH DATA */

  MPI_Gather (localbuf, 100, MPI_INT, rootbuf, 100,
              MPI_INT, root, comm);

  /* RESULTS BACK IN ROOT */
```

Código 5.10: Ejemplo de las comunicaciones colectivas `MPI_Scatter` y `MPI_Gatther`.

La sintaxis del `MPI_Scatter` se muestra con el código 5.11. El vector `sendbuf` del proceso `root` se distribuirá en partes de `sendcount` elementos al resto de procesos que están dentro del comunicador `comm`. Cada proceso recibirá la parte que le corresponde, de `recvcount` elementos de tipo `recvtype` en el vector `recvcount`.

```
int MPI_Scatter( void* sendbuf,          /* in */
                 int sendcount,          /* in */
                 MPI_Datatype sendtype,  /* in */
                 void* recvbuf,          /* out */
                 int recvcount,          /* in */
                 MPI_Datatype recvtype,  /* in */
                 int root,               /* in */
                 MPI_Comm comm);         /* in */
```

Código 5.11: Sintáxi del `MPI_Scatter`

La sintaxis del `MPI_Gather` se muestra en el código 5.12. Es la operación inversa al *scatter*. Ahora cada proceso envía una parte de un vector `sendbuf` de `sendcount` elementos que el proceso `root` almacenará en `recvbuf`.

```
int MPI_Gather( void* sendbuf,          /* in */
                int sendcount,          /* in */
                MPI_Datatype sendtype,  /* in */
                void* recvbuf,          /* out */
                int recvcount,          /* in */
                MPI_Datatype recvtype,  /* in */
                int root,               /* in */
                MPI_Comm comm );        /* in */
```

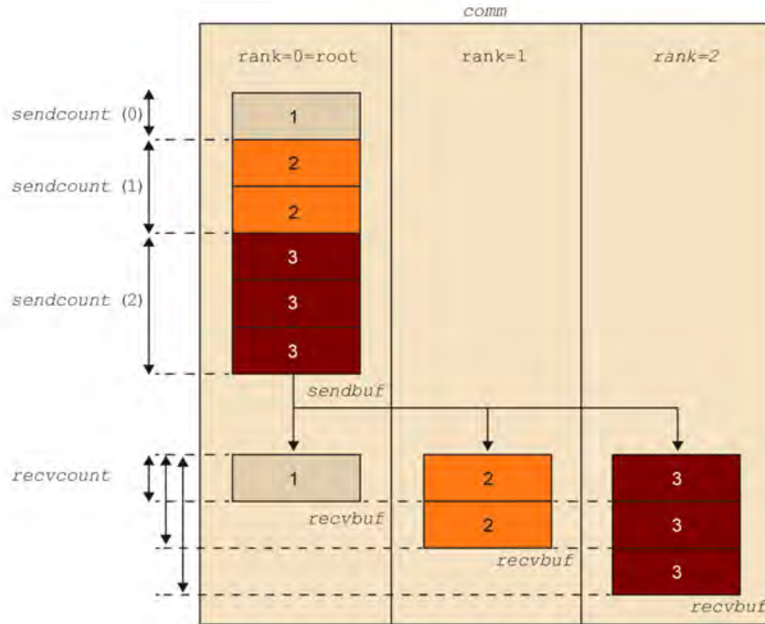
Código 5.12: Sintaxis del gather

La última llamada colectiva que veremos en detalle es la del *scatterv*. Esta colectiva es como la del *scatter* pero en este caso el número de elementos que puede recibir cada proceso puede ser diferente. Esta operación es muy útil cuando sabemos que hay desbalanceo en el número de elementos a recibir por parte de los procesos. En este caso, para distribuir los datos debemos indicar la posición inicial del primer elemento que va a cada proceso (`displ`), y la cantidad de elementos que le debemos enviar (`sendcount`). La figura 40 muestra gráficamente cómo se deben inicializar estos dos vectores.

En el código 5.13 se hace que el proceso cero (`root`) envíe un elemento al proceso `0`, dos elementos al proceso `1` y cuatro elementos al proceso `2`, suponiendo que hay tres procesos ejecutando el código.

```

   ....
   /* Cada proceso hace el cálculo de SendCount y Displacement
      pero podría hacerse únicamente en el proceso root       */

   SendCount[0]=1; Displacement[0] = 0;
   SendCount[1]=2; Displacement[1] = 1;
   SendCount[2]=4; Displacement[2] = 1+2;

   RecvCount = SendCount[myRank];

   MPI_Scatterv(Vector_A, SendCount, Displacement,
                  MPI_FLOAT,Mybuffer_A, RecvCount, MPI_FLOAT,
                  Root, MPI_COMM_WORLD);

```

Figura 40. Operación colectiva `MPI_Scatterv`.




```
16      ...
```

Código 5.13: Ejemplo de la comunicación colectiva `MPI_Scatterv`

La sintaxis del `MPI_Scatterv` se muestra en el código 5.14. Los parámetros los dividiremos entre los que debe rellenar el proceso `root` y los que deben aportar todos los procesos. Los primeros indican el vector a repartir (`sendbuf`), cuántos elementos para cada proceso (`sendcounts`), a partir de qué elemento se envía a cada proceso (`displs`), el tipo de los elementos a enviar, y son los que debe suministrar correctamente el proceso `root` en el comunicador `comm`. El resto son, en orden, el vector de recepción, el número de elementos que se reciben y el tipo de estos elementos.

```
1  int MPI_Scatterv( void *sendbuf,            /* in */
2                    int *sendcounts,          /* in */
3                    int *displs,              /* in */
4                    MPI_Datatype sendtype,    /* in */
5                    void *recvbuf,            /* out */
6                    int recvcount,            /* in */
7                    MPI_Datatype recvtype,    /* in */
8                    int root,                 /* in */
9                    MPI_Comm comm);           /* in */
```

Código 5.14: Sintáxis del `MPI_Scatterv`

Otras colectivas que pueden ser interesantes pero que no detallaremos en este módulo son: `MPI_GATHERV`, `MPI_ALLGATHER`, `MPI_ALLGATHERV`, `MPI_ALLTOALL`, `MPI_ALLTOALLV`.

5.3. OpenMP

OpenMP consiste en una extensión API de los lenguajes C, C++ y Fortran para escribir programas paralelos para memoria compartida. OpenMP se encuentra incluido en el compilador gcc, al igual que otros muchos compiladores propietarios como el icc de Intel, el xlc de IBM, etc. Básicamente incluye soporte para crear automáticamente *threads*, compartir trabajo entre éstos, y sincronizar los *threads* y la memoria.

OpenMP

En julio del 2011 apareció la especificación de la versión 3.1, que incorpora nuevas funcionalidades como la *reduction* min, max, extensiones del *atomici*, etc.

En este subapartado solo daremos algunos ejemplos sin entrar en un gran detalle, ya que no es objetivo exclusivo de este módulo.

5.3.1. Un programa simple

El código 5.15 muestra un "hola mundo" escrito en OpenMP, donde todos los *threads* creados escribirán ese mensaje.

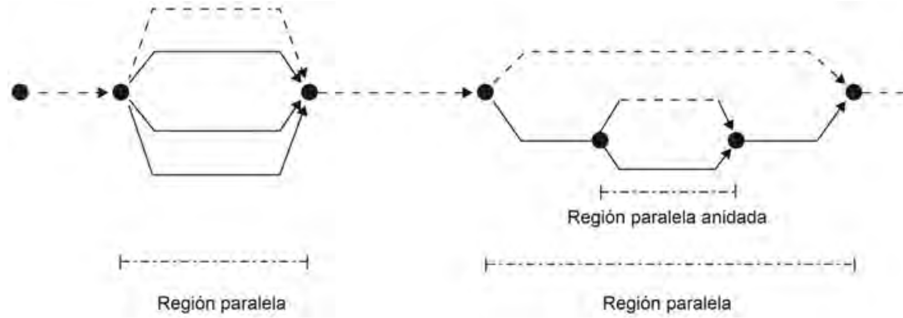
```
1  #include <omp.h>
2
3  int rank;
4  int nproc;
5
6  int main( int argc, char* argv[] ) {
7
8
9      #pragma omp parallel
10     {
11        int thread_id;
12        int num_threads;
13
14        /* Nothing to do */
15
16        thread_id   = omp_get_thread_num();
17        num_threads = omp_get_num_threads();
18        printf("thread: %d de un total de %d: Hola mundo!\n",thread_id,
               num_threads);
19
20     }
21
22 }
```

Código 5.15: Programa simple en OpenMP

La directiva `omp parallel` hace que el compilador inserte código para generar un conjunto de *threads*, tantos como indique la variable de entorno `OMP_NUM_THREADS`. De esta forma, si compilamos y ejecutamos este código con `OMP_NUM_THREADS=4 ./omp-program`, deberían aparecer cuatro mensajes de "hola mundo". Hay otras formas de indicar el número de *threads*. Una es utilizando la función de OpenMP `omp_set_num_threads(number)`. La otra es en el momento de poner la directiva `parallel`, indicar la cláusula `num_threads(number)`.

El modelo de programación OpenMP es un modelo *Fork/Join*, donde podríamos tener directivas `parallel` anidadas, tal y como observamos en la figura 41.

Figura 41. Modelo *Fork/Join* del modelo de programación OpenMP.



En la directiva `parallel` se le puede especificar si las variables del programa, declaradas fuera del contexto del `parallel` son privadas (`private(var1,var2, ...)`), compartidas (`shared(var1,var2, ...)`), o son privadas pero se tiene que copiar el valor que tienen en ese momento (`firstprivate(var1,var2, ...)`).

5.3.2. Sincronización y *locks*

El hecho de tener varios *threads* que pueden estar compartiendo una misma posición de memoria puede llevar a tener condiciones de carrera. Para ello, necesitamos algún tipo de sincronización o *locks* para conseguir un orden en los accesos.

Las tres directivas OpenMP de sincronización son:

- `barrier`: Los *threads* no pueden pasar el punto de sincronización hasta que todos lleguen a la barrera y todo el trabajo anterior se haya completado. Algunas construcciones tienen una barrera implícita al final, como por ejemplo el `parallel`.

- `critical`: Crea una región de exclusión mútua donde solo un *thread* puede estar trabajando en un instante determinado. Se le puede asignar un nombre a la zona de exclusión con `critical (name)`. Por defecto todas tienen el mismo nombre, y por consiguiente, cualquier punto del programa donde nos encontramos un `critical` querrá decir que solo un *thread* puede entrar.

- `atomic`: Implica que la operación simple a la que afecte (operación del tipo leer y actualizar) se haga de forma atómica.

El código 5.16 muestra la versión OpenMP del cálculo de `pi` en el que se usa el `critical`. Para el `atomic` sería cambiar la directiva `critical` por `atomic`.

```
1
2 void main ()
3 {
4    int i, id;
5    double x, pi, sum=0.0;
6
7    step = 1.0/(double) num_steps;
```

```
8     omp_set_num_threads(NUM_THREADS);
9     #pragma omp parallel private(x, i, id)
10    {
11       id = omp_get_thread_num();
12       for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
13          x = (i-0.5)*step;
14          #pragma omp critical
15          sum = sum + 4.0/(1.0+x*x);
16       }
17    }
18    pi = sum * step;
19 }
20 }
```

Código 5.16: Ejemplo de sincronización con `OpenMP`

En este código, cada *thread* realiza las iteraciones del bucle `for` en función de su identificador dentro del conjunto de *threads*, creado con la directiva `parallel`. La variable `sum`, que es la variable donde los *threads* realizarán el cálculo de `pi`, está compartida por defecto. Para evitar una condición de carrera en su cálculo, se tiene que actualizar vía `critical` o `atomic`. Por otro lado, las variables `id, x, i` también serían compartidas por defecto si no se hubieran privatizado explícitamente. En caso de compartirse, el código no sería correcto, ya que todos estarían actualizando la variable inducción.

Otra forma de realizar el cálculo de la variable `sum`, sin necesidad de realizar una exclusión mutua es realizando una reducción con la cláusula `reduction`. El código 5.17 muestra cómo quedaría.

```
1  void main ()
2  {
3     int i, id;
4     double x, pi, sum;
5
6     step = 1.0/(double) num_steps;
7     omp_set_num_threads(NUM_THREADS);
8     #pragma omp parallel private(x, i, id) reduction(+:sum)
9     {
10       id = omp_get_thread_num();
11       for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
12          x = (i-0.5)*step;
13          sum = sum + 4.0/(1.0+x*x);
14       }
15    }
16    pi = sum * step;
17 }
```

Código 5.17: Ejemplo de reducción con `OpenMP`

La cláusula `reduction` crea una copia privada de la variable, de tal forma que se actualiza esta variable local, y después se actualiza de forma sincronizada la variable compartida `sum`, realizando así la reducción. En la cláusula `reduction` se indica qué tipo de operación se debe realizar.

Otra forma de sincronizarse es utilizando *locks*. OpenMP dispone de primitivas *lock* para sincronizaciones de bajo nivel. Las funciones son las siguientes:

- `omp_init_lock`: inicializa el *lock*.
- `omp_set_lock`: adquiere el *lock*, pudiéndose quedar bloqueado.
- `omp_unset_lock`: libera el *lock*, pudiendo desbloquear a *threads* que estuvieran bloqueados.
- `omp_test_lock`: prueba a adquirir un *lock*, pero no se bloquea.
- `omp_destroy_lock`: libera los recursos del *lock*.

El código 5.18 muestra un ejemplo de uso de las operaciones de *lock*.

```
1  #include <omp.h>
2  void foo ()
3  {
4     omp_lock_t lock;
5
6     omp_init_lock(&lock);
7     #pragma omp parallel
8     {
9         omp_set_lock(&lock);
10        // Región de exclusión mútua.
11        omp_unset_lock(&lock);
12    }
13    omp_destroy_lock(&lock);
14 }
```

Código 5.18: Ejemplo de sincronización con `locks` con `OpenMP`.

5.3.3. Compartición de trabajo en bucles

Las construcciones de compartición de trabajo dividen la ejecución de una región de código entre los *threads* de un equipo. De esta forma los *threads* cooperan para hacer un trabajo sin necesidad de tener que identificarse, tal y como pasaba en el código 5.16.

OpenMP tiene cuatro construcciones *worksharing: loop worksharing*, *single*, *section* y *master*. De éstas veremos ejemplos de *loop* y *single*. Las otras dos no se suelen usar.

En el código 5.19 mostramos un ejemplo de uso del *loop worksharing* (`omp for`). Las iteraciones del bucle asociadas a esta directiva se dividen entre los *threads* del conjunto de *threads* creado, y se realizan totalmente en paralelo. Es responsabilidad del usuario que las iteraciones sean independientes, o que de alguna forma se sincronicen si es necesario. Además, el bucle debe ser un bucle `for` donde se pueda determinar cuántas iteraciones hay. La variable de inducción (la `i` en el caso del ejemplo) debe ser de tipo entero, puntero o iteradores (C++). Esta variable es automáticamente privatizada. El resto de variables son *shared* por defecto.

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2

void main ()
{
   int i, id;
   double x, pi, sum;

   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
   #pragam omp parallel
   {
     #pragma omp for private(x) reduction(+:sum)
     for (i=1; i<=num_steps; i++) {
         x = (i-0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
     }
   }
   pi = sum * step;
}
```

Código 5.19: Ejemplo de la directiva `omp for` de `OpenMP`.

La forma de distribuir los *threads* es, por defecto, *schedule* (*static*), de tal forma que en tiempo de compilación se determina qué iteraciones le corresponden a los *threads* existentes. Este tipo de distribución consiste en dar, en el caso del ejemplo, `num_steps/#threads` iteraciones consecutivas a cada *thread*. Hay otros tipos de *schedule* que se le pueden especificar a la directiva `for`. Estos tipos son: *dynamic* y *guided*.

Dynamic* y *guided

El *schedule* (*dynamic*) o *schedule* (*guided*) son tipos de distribución dinámica de las iteraciones, e intentan reducir el desbalanceo de carga entre los *threads*.

El *loop worksharing*, por defecto, tiene un *barrier* implícito al final. Si quisiéramos que esta barrera desapareciera tendríamos que indicarle la cláusula `nowait` en el `omp for`. Otra cláusula interesante es la de `collapse`, que es para cuando tenemos dos bucles anidados perfectos (el cuerpo del bucle externo es únicamente el bucle interno). En este caso, el espacio de iteraciones se colapsa como si fuera un único *loop* con tantas iteraciones como el producto de las iteraciones de los dos bucles.

Finalmente, otra construcción a destacar para el caso de tener que compartir trabajo es la de indicar que solo queremos que uno de los *threads* haga una determinada cosa. Esto lo podemos hacer con la directiva `single`. Veremos un ejemplo en el siguiente subapartado.

5.3.4. Tareas en OpenMP

Las *tasks* son unidades de trabajo que pueden ser ejecutadas inmediatamente, o bien dejadas en una cola para que sean posteriormente ejecutadas. Son los *threads* del conjunto de *threads* creados con el `parallel` los que cooperan en la ejecución de estas tareas.

En realidad, cuando se realiza un `parallel` se crean tareas implícitas a las que se le asigna trabajo a realizar. En OpenMP se puede crear tareas explícitas con la directiva *task*. Con las tareas *tasks*, cuando un *thread* se la encuentra, empaqueta el código y los datos, y crea una nueva tarea para que sea ejecutada por un *thread*.

Normalmente se utiliza este *pragma* cuando se realizan tareas que no están dentro de un *loop* con una variable inducción que permita utilizar `omp for`.

El código 5.20 muestra una forma de utilizar la directiva `task`.

```
1 ...
2 ...
3 #pragma omp parallel
4 #pragma omp single
5 traverse_list ( l );
6 ...
7 ...
8 void traverse_list ( List l )
9 {
10   Element e ;
11   for ( e = l->first ; e ; e = e->next )
12        #pragma omp task
13          process (e );
14 }
```

Código 5.20: Ejemplo de creación de tareas explícitas en `OpenMP`.

En este código se crean un conjunto de *threads* con el `omp parallel`. Para evitar que todos los *threads* ejecuten el código `traverse_list`, y por consiguiente dupliquen el trabajo, se debe utilizar la directiva `omp single`, tal y como mostramos en el código 5.20. De esta forma solo un *thread* se recorrerá la lista y creará las tareas.

Para cada llamada a la función `process(e)` se crea una tarea explícita que alguno de los *threads* del `parallel` ejecutará. ¿Y cómo nos esperamos a que acaben? Hay dos construcciones que nos pueden ayudar a esperarlas:

1) `#pragma omp barrier`: con esta directiva hacemos que todos los *threads* del `parallel` se esperen a que todo el trabajo anterior, incluidas las tareas, se complete.

2) `#pragma omp taskwait`: en este caso, hace suspenderse la tarea actual hasta que todas las tareas hijo (directo, no descendientes) se hayan completado.

pragma omp task

Cuando utilicemos el `pragma omp task` debemos vigilar si queremos que todos los *threads* de un `omp parallel` creen las tareas o no. En caso de que no, deberíamos usar el `pragma omp single`.

El código 5.21 muestra cómo podemos esperar las tareas.

```
1
2 void traverse_list ( List l )
3 {
4   Element e ;
5   for ( e = l->first ; e ; e = e->next )
6        #pragma omp task
7          process (e );
8
9   #pragma omp taskwait
10   //  En este punto todas las tareas han acabado.
11 }
```

Código 5.21: Ejemplo de sincronización de tareas en `OpenMP`.

5.4. OmpSs

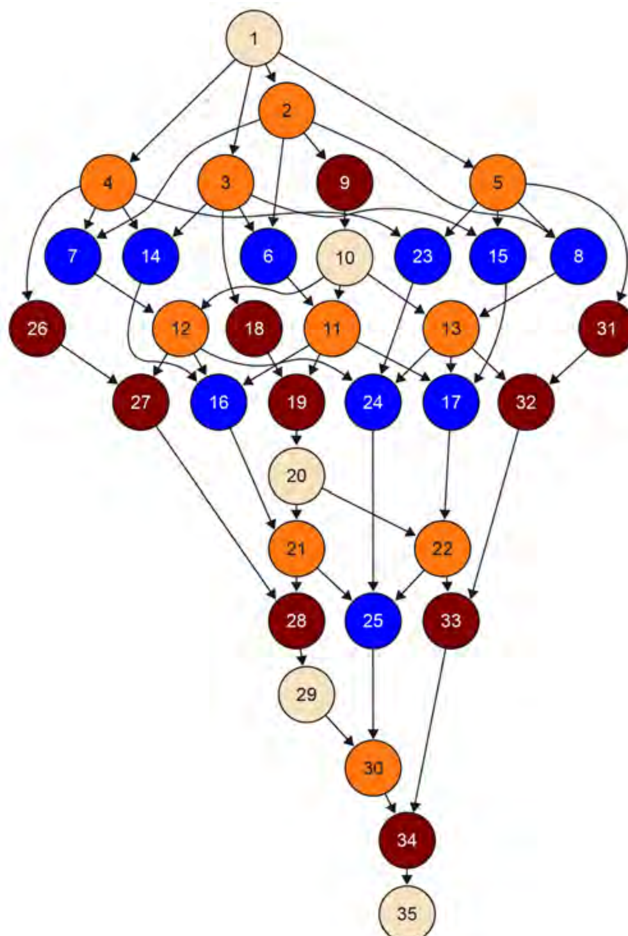
OmpSs es una extensión del modelo de programación paralela OpenMP. Estas extensiones consisten en la introducción de una serie de cláusulas que ayudan a la sincronización en tiempo de ejecución de las tareas que tienen dependencias.

Las cláusulas son básicamente:

- `input(expr-list)`: indican los datos de entrada que necesita la tarea, y por consiguiente, no se ejecutará hasta que éstas estén ya calculadas.
- `output(expr-list)`: indican los datos de salida que genera la tarea, y por consiguiente, el sistema de ejecución OmpSs podrá determinar cuándo otras tareas se pueden ejecutar.
- `inout(expr-list)`: indican los datos que son tanto de entrada (la tarea los necesita) como de salida, por lo que otras tareas pueden depender de ellas.

Un ejemplo típico para mostrar cómo se puede sacar provecho de este modelo de programación es la ejecución del `blocked_cholesky` (código 5.22). La figura 42 muestra el grafo de ejecución de las tareas definidas en el código, para una $NB = 5$. Las tareas más claras son ejecuciones de la función `spotrf`, las tareas naranjas corresponden a la función `strsm`, las tareas azules a la función `ssyrk` y finalmente, las tareas granates a la función `sgemm`.

Figura 42. Tareas ejecutadas en el `blocked_cholesky`.



Lectura complementaria

Eduard Ayguade; Nawal Copty; Alejandro Duran; Jay Hoeflinger; Yuan Lin; Federico Massaioli; Xavier Teruel; Priya Unnikrishnan; Guansong Zhang (2009). *The Design of* `OpenMP` *Tasks, IEEE transactions on parallel and distributed systems* (vol. 20, núm. 3).

```
#pragma omp task inout([BS][BS]A)
void spotrf (float *A);
#pragma omp task input([BS][BS]A) inout([BS][BS]C)
void ssyrk (float *A, float *C);
#pragma omp task input([BS][BS]A, [BS][BS]B)
                  inout([BS][BS]C)
void sgemm (float *A, float *B, float *C);
#pragma omp task input([BS][BS]T) inout([BS][BS]B)
void strsm (float *T, float *B);
void blocked_cholesky( int NB, float *A ) {
   int i, j, k;
   for (k=0; k<NB; k++) {
      spotrf (A[k*NB+k]) ;
      for (i=k+1; i<NB; i++)
         strsm (A[k*NB+k], A[k*NB+i]);
      for (i=k+1; i<NB; i++) {
         for (j=k+1; j<i; j++)
            sgemm( A[k*NB+i], A[k*NB+j], A[j*NB+i]);
         ssyrk (A[k*NB+i], A[i*NB+i]);
      }
   }
}
```

Código 5.22: Ejemplo de las directivas de OmpSs.

En este código se han definido los *input*, *output* y *inout* en las cabeceras de las funciones. Eso significa que cada vez que se llame una de estas funciones se creará una tarea explícita con el código de la función. Además, para cada entrada, salida, entrada y salida, se especifica la forma que tiene ese dato, indicándolo con dos dimensiones entre corchetes delante de cada identificador. En este caso se indica que son matrices de $BS \times BS$ elementos.

Otro ejemplo de definición de tareas con OmpSs lo tenemos en el código 5.23, que resuelve un `gauss_seidel`.

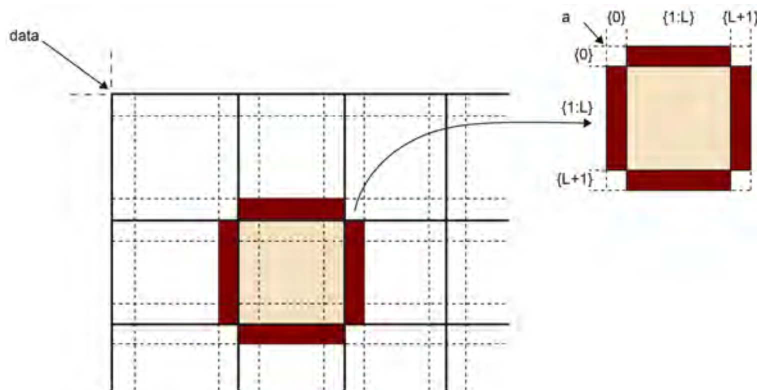
```

pragma omp task input(a[0][1;L],a[L+1][1;L])
                input(a[1;L][0],a[1;L][L+1])
                inout(a[1;L][1;L])
void gauss_seidel(double a[N][N])
{
   for (int i=1; i<=L; i++)
      for (int j=1; j<=L; j++)
         a[i][j] = 0.2 * ( a[i][j] + a[i-][j] + a[i+1][j]
                         + a[i][j-1] + a[i][j+1]);
}

double data[N][N];
...

for (int it=0; it < NITERS; it++)
   for (int i=0; i<N-2; i+=L)
      for (int j=0; j<N-2; j+=L)
         gauss_seidel(&data[i][j]);
```

Código 5.23: Ejemplo de definición de entradas y salidas con OmpSs.

Cada tarea realiza el cálculo de un bloque de $L \times L$ elementos de la matriz *data*. Para ese cálculo necesitamos los bordes superior, inferior, derecha e izquierda. Estos bordes pueden ser calculados por otras tareas, y por eso se definen esas posibles dependencias con

las directivas `input` y `inout`. La figura 43 muestra los bordes definidos en la tarea del código.

Figura 43. Bordes de los que depende cada tarea.



Por ejemplo, `input(a[0][1;L])`, indica que de la fila 0 de la matriz de entrada `a`, se necesita el elemento 1 y L elementos más; es decir, desde 1 a $L+1$. También se podría haber indicado `input(a[0][1:L+1])`.

5.5. Caso de estudio

En este apartado veremos cómo paralelizar un mismo código (contar cuántas instancias de una misma clave, entero, hay en un vector) con OpenMP, MPI y con OpenMP + MPI.

5.5.1. Código secuencial

El código secuencial de la búsqueda de una clave en un vector es el siguiente:

```
1  long search_serial(long Nlen, long *a, long key)
2  {
3    long count = 0;
4    int i;
5
6    for(i=0; i<Nlen; i++)
7      if (a[i]==key) count++
8
9    return count;
10
11 }
12
13 ...
14 int main()
15 {
16   long a[N];
17   long key;
18   long nkey;
19   ...
20
21   nkey = search_serial(N, a, key);
22 }
```

Código 5.24: Código de la búsqueda de una clave en un vector de forma secuencial.

5.5.2. OpenMP

La versión OpenMP, utilizando la estrategia de `loop parallelism` es la que aparece en el código 5.25. Lo que estamos haciendo es la división de las iteraciones del bucle a realizar entre los *threads* creados con el `parallel`. Cada uno hará el cálculo local del número de claves encontradas, y gracias a la reducción se obtendrá el valor total de claves en el vector.

```
1  long search_loop(long Nlen, long *a, long key)
2  {
3    long count = 0;
4    int i;
5
6  #pragma omp parallel
7  {
8    #pragma omp for reduction(+:count)
9    for(i=0; i<Nlen; i++)
10     if (a[i]==key)
11         count++
12
13 }
14   return count;
15
16 }
17
18 ...
19 int main()
20 {
21   long a[N];
22   long key;
23   long nkey;
24   ...
25   nkey = search_loop(N, a, key);
26 }
```

Código 5.25: Ejemplo de `loop parallelism` con `OpenMP`

La resolución del mismo problema la podríamos plantear con un solución secuencial recursiva. Esto daría pie a una estrategia de paralelización distinta: *Divide & Conquer*. El código 5.26 muestra una solución recursiva al mismo código. Se divide la búsqueda en dos partes, y después se suma el resultado de estas dos búsquedas.

A partir de esta solución podemos realizar una paralelización utilizando OpenMP. En este caso, sin embargo, no es posible realizar una paralelización *loop*, ya que no tenemos tal bucle. Aquí es más adecuado realizar una paralelización usando las *tasks*.

```
1  long search_serial_rec(long Nlen, long *a, long key)
2  {
3    long count = 0;
4    long Nlen2 = Nlen / 2;
5    int i;
6
7    if (Nlen<2)
8    {
9       count = (a[0]==key);
10   }
```

```
11   else
12   {
13     count1 = search_serial_rec(Nlen2, a, key);
14     count2 = search_serial_rec(Nlen-Nlen2, a+Nlen2, key);
15     count = count1+count2;
16   }
17
18   return count;
19 }
20
21 ...
22 int main()
23 {
24   long a[N];
25   long key;
26   long nkey;
27   ...
28   nkey = search_serial_rec(N, a, key);
29 }
```

Código 5.26: Implementación secuencial de la solución recursiva.

En el momento de paralelizar las tareas podemos pensar en dos posibilidades, dejar que se hagan en paralelo solamente los casos base de la recursividad secuencial (es decir, las hojas del árbol de búsqueda) o bien realizar también en paralelo el árbol de búsqueda. En lo que estamos tratando aquí, el primer caso significaría poner un `#pragma omp task` en la parte del `if`. Pero debido a que el trabajo a realizar es tan pequeño, el `overhead` de creación y espera de la tarea no vale la pena. Por consiguiente, optaremos por el segundo caso. Eso significa que en cada llamada a la función `search_serial_rec` vamos a crear una *task*. El código 5.27 muestra la solución adoptada.

```
1 long search_task_rec(long Nlen, long *a, long key)
2 {
3   long count = 0;
4   long Nlen2 = Nlen / 2;
5   int i;
6
7   if (Nlen<2)
8   {
9      count = (a[0]==key);
10  }
11  else
12  {
13    #pragma omp task shared(count1) firstprivate(Nlen2,key)
14    count1 = search_task_rec(Nlen2, a, key);
15
16    #pragma omp task shared(count2) firstprivate(Nlen2,key)
17    count2 = search_task_rec(Nlen-Nlen2, a+Nlen2, key);
18
19    #pragma omp taskwait
20    count = count1+count2;
21  }
22
23  return count;
24 }
25
26 ...
27 int main()
28 {
29   long a[N];
30   long key;
31   long nkey;
32   ...
```

```
33   #pragma omp parallel
34   #pragma omp single
35   nkey = search_task_rec(N, a, key);
36 }
```

Código 5.27: Ejemplo de `Divide & Conquer` con tasks de `OpenMP`.

Notad cómo creamos las *tasks* y posteriormente, cuando se tiene que hacer la suma de los dos resultados, nos sincronizamos con el `taskwait`. Si no lo hicieramos así, no estaríamos sumando los valores adecuados. Otras cosas importantes a remarcar son:

- En el programa principal creamos los *threads* con el `parallel`, pero inmediatamente después, para que solo un *thread* cree las tareas, insertamos el `pragma single`.

- En los `pragmas tasks` se introduce la cláusula `shared` para que las variables `count1` y `count2`, privadas por defecto por estar declaradas dentro del `parallel`, puedan ser compartidas y sumadas correctamente para obtener el `count` acumulado después del `taskwait`.

- El `firstprivate` no es del todo necesario aquí porque las variables afectadas no son modificadas, pero es importante notar que aquellas variables que siendo compartidas se pudieran modificar en la ejecución de las tareas, deberíamos declararlas así si el valor que nos interesa es justo el que tiene en el momento de hacer la llamada a la función (tarea).

Finalmente, en una estrategia paralela *Divide & Conquer*, es normal que se quiera limitar el nivel recursivo del árbol en el que se siguen creando tareas. Esto es lo que se conoce como `cutoff`. La idea sería la de establecer con un parámetro o una constante cuántas recursividades paralelas permitimos.

El código 5.28 muestra un ejemplo de cómo realizarlo con la versión paralela que acababamos de analizar.

```
1  long search_task_rec(long Nlen, long *a, long key, int d)
2  {
3    long count = 0;
4    long Nlen2 = Nlen / 2;
5    int i;
6
7    if (Nlen<2)
8    {
9       count = (a[0]==key);
10   }
11   else
12   {
13    if (d < CUTOFF)
14    {
15     #pragma omp task shared(count1) firstprivate(Nlen2,key,d)
16     count1 = search_task_rec(Nlen2, a, key, d+1);
17
18     #pragma omp task shared(count2) firstprivate(Nlen2,key,d)
19     count2 = search_task_rec(Nlen-Nlen2, a+Nlen2, key, d+1);
20
```

```
21       #pragma omp taskwait
22       count = count1+count2;
23      }
24      else {
25         count1 = search_task_rec(Nlen2, a, key, d);
26         count2 = search_task_rec(Nlen-Nlen2, a+Nlen2, key, d);
27         count = count1+count2;
28      }
29    }
30
31    return count;
32  }
33
34  ...
35  int main()
36  {
37    long a[N];
38    long key;
39    long nkey;
40    ...
41    #pragma omp parallel
42    #pragma omp single
43    nkey = search_task_rec(N, a, key, 0);
44  }
```

Código 5.28: Ejemplo de `Divide & Conquer` con tasks y `cutoff`.

De esta forma, cuando llegamos a un nivel de recursividad igual a `cutoff`, pasaríamos a una versión serie de la recursividad, sin crear más tareas.

5.5.3. MPI

En este subapartado veremos una paralelización del mismo código de búsqueda, pero utilizando una estructura SPMD y el modelo para memoria distribuida MPI. La estructura que seguiremos es *Master/Worker* pero tanto `master` como `workers` realizarán trabajo de búsqueda. En particular, los pasos serán los siguientes:

1) El proceso `master` distribuirá el vector y la clave a buscar. La inicialización la realizará el `master` y pasará los datos del vector al resto de procesos. El `master` se quedará una parte del vector.

2) Todos los procesos realizan la búsqueda en la porción del vector que le ha tocado.

3) Todos los procesos comunican la cantidad de elementos encontrados en su porción al `master`.

Veremos primero dos soluciones con comunicación punto a punto (códigos 5.29 y 5.30), y después una solución con comunicaciones de tipo colectiva (código 5.31). En todas las soluciones vamos a suponer que el número de elementos del vector es múltiplo del número de procesadores que tenemos para realizar la búsqueda. Para el punto a punto mostraremos dos versiones, una en la que todos alocatan todo el vector y después solo trabajan con una

parte, y la otra, más eficiente, en la que cada procesador solo alocata una parte del vector, a excepción del `master` que alocata todo también.

La primera versión punto a punto se muestra en el código 5.29. En este código se pueden ver claramente los tres pasos comentados más arriba: inicialización y comunicación de la parte del vector, proceso y obtención de los resultados parciales para obtener el resultado final. En el proceso de los datos (búsqueda local), queremos hacer notar la forma en que se recorre el vector: los índices tienen que ir del inicio al final de la parte del vector en la que cada proceso trabaja.

```
1
2 #include <mpi.h>
3
4 ...
5
6 void main (int argc, char *argv[])
7 {
8   long *a, *my_a;
9   long key;
10  long nkey;
11  int i, my_id, my_n, numprocs;
12  int iproc, count, my_count;
13  MPI_Status status;
14  ...
15  MPI_Init(&argc, &argv) ;
16  MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
17  MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
18
19  my_n= N/numprocs;
20  a = (long *) malloc(N*sizeof(long));
21
22  /* Inicializacíon y Envío */
23  if (my_id==0) /* Master */
24  {
25    init(N, a);
26    for (iproc=1, count=; iproc<numprocs; iproc)
27     MPI_Send( &a[iproc*N/numprocs], N/numprocs , MPI_INT, iproc, 0,
          MPI_COMM_WORLD)
28  }
29  else  /* Worker */
30     MPI_Recv( &a[my_id*N/numprocs], N/numprocs, MPI_INT, 0, 0,
          MPI_COMM_WORLD, &status);
31
32  /* Búsqueda local */
33  my_count = 0
34  for(i=my_id*N/numprocs; i<(my_id+1)*N/numprocs; i++)
35    if (my_a[i]==key)
36        my_count++
37
38  /* Envío del count local para la obtención de los resultados
        parciales */
39  if (my_id==0) /* Master */
40  {
41    int count_proc;
42    count = my_count;
43    for (iproc=1, count=; iproc<numprocs; iproc)
44    {
45     MPI_Recv( &count_proc, 1, MPI_INT, iproc, 0, MPI_COMM_WORLD, &
          status);
46     count += count_proc;
47    }
48  }
49  else  /* Worker */
50     MPI_Send( &my_count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD)
```

```
51
52   free(a);
53   printf("El total de claves es:%d\n", count);
54 }
```

Código 5.29: Ejemplo con comunicaciones punto a punto de MPI

Otra versión del mismo código punto a punto, pero solo alocatando la parte del vector que le corresponde, la mostramos en el código 5.30).

```
1
2 #include <mpi.h>
3
4 ...
5
6 void main (int argc, char *argv[])
7 {
8   long *a, *my_a;
9   long key;
10  long nkey;
11  int i, my_id, my_n, numprocs;
12  int iproc, count, my_count;
13  MPI_Status status;
14  ...
15  MPI_Init(&argc, &argv) ;
16  MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
17  MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
18
19  my_n= N/numprocs;
20
21  /* Inicializacíon y Envío */
22  if (my_id==0) /* Master */
23  {
24    a = (long *) malloc(N*sizeof(long));
25    init(N, a);
26    my_a = a;
27  }
28  else {
29    my_a = (long *) malloc(my_n*sizeof(long));
30  }
31
32  if (my_id==0) /* Master */
33  {
34    for (iproc=1, count=; iproc<numprocs; iproc)
35     MPI_Send( &a[iproc*N/numprocs], N/numprocs , MPI_INT, iproc, 0,
           MPI_COMM_WORLD)
36  }
37  else  /* Worker */
38     MPI_Recv( my_a, N/numprocs, MPI_INT, 0, 0, MPI_COMM_WORLD, &status
           );
39
40  /* Búsqueda local */
41  my_count = 0
42  for(i=0; i<N/numprocs; i++)
43    if (my_a[i]==key)
44        my_count++
45
46  /* Envío del count local para la obtención de los resultados
        parciales */
47  if (my_id==0) /* Master */
48  {
49    int count_proc;
50    count = my_count;
51    for (iproc=1, count=; iproc<numprocs; iproc)
52    {
```

```
53      MPI_Recv( &count_proc, 1, MPI_INT, iproc, 0, MPI_COMM_WORLD, &
            status);
54      count += count_proc;
55     }
56   }
57   else  /* Worker */
58      MPI_Send( &my_count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD)
59
60   if (my_id==0) /* Master */
61     free(a);
62   else          /* Worker */
63     free(my_a);
64
65   printf("El total de claves es: %d\n", count);
66 }
```

Código 5.30: Ejemplo con comunicaciones punto a punto de MPI, pero solo alocatando una parte del vector.

En este caso, el recorrido que se realiza en cada proceso para la búsqueda local se hace desde el índice 0 al total de elementos que le corresponde a cada proceso. Notad que ahora solo se ha pedido memoria para el fragmento del vector que recibimos.

Cualquiera de las dos versiones punto a punto se podría resolver mediante la utilización de comunicaciones colectivas, que seguramente son mucho más eficientes que la versión punto a punto. El código 5.31 muestra la versión con colectivas del segundo código punto a punto.

```
1
2 #include <mpi.h>
3
4 ...
5
6 void main (int argc, char *argv[])
7 {
8   long *a, *my_a;
9   long key;
10  long nkey;
11  int i, my_id, my_n, numprocs;
12  int iproc, count, my_count;
13  MPI_Status status;
14  ...
15  MPI_Init(&argc, &argv) ;
16  MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
17  MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
18
19  my_n= N/numprocs;
20
21  if (my_id==0) /* Master */
22  {
23    a = (long *) malloc(N*sizeof(long));
24    init(N, a);
25    my_a = a;
26  }
27  else {
28    my_a = (long *) malloc(my_n*sizeof(long));
29  }
30
31  MPI_Scatter (a, N/numprocs, MPI_INT, my_a, N/numprocs, MPI_INT, 0,
        MPI_COMM_WORLD);
32
33
```

```
34   /* Búsqueda local */
35   my_count = 0
36   for(i=0; i<N/numprocs; i++)
37     if (my_a[i]==key)
38         my_count++
39
40   MPI_Reduce(&my_count, &count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
         ;
41
42
43   if (my_id==0) /* Master */
44     free(a);
45   else          /* Worker */
46     free(my_a);
47
48   printf("El total de claves es:%d\n", count);
49 }
```

Código 5.31: Ejemplo con comunicaciones colectivas de `MPI`.

5.5.4. Híbrido: MPI + OpenMP

En este subapartado acabaremos combinando MPI y OpenMP en el código 5.32, con tal de poder aprovechar las características de una máquina con memoria distribuida, pero que en cada nodo tiene varios procesadores con memoria compartida. En este código primero distribuimos el trabajo entre nodos, y después, en cada nodo, paralelizaremos la búsqueda tal y como hacíamos en el apartado de OpenMP de los casos de estudio.

```
1
2 #include <mpi.h>
3 #include <omp.h>
4
5 ...
6
7 void main (int argc, char *argv[])
8 {
9   long *a, *my_a;
10  long key;
11  long nkey;
12  int i, my_id, my_n, numprocs;
13  int iproc, count, my_count;
14  MPI_Status status;
15  ...
16  MPI_Init(&argc, &argv) ;
17  MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
18  MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
19
20  my_n= N/numprocs;
21
22  if (my_id==0) /* Master */
23  {
24    a = (long *) malloc(N*sizeof(long));
25    init(N, a);
26    my_a = a;
27  }
28  else {
29    my_a = (long *) malloc(my_n*sizeof(long));
30  }
31
32  MPI_Scatter (a, N/numprocs, MPI_INT, my_a, N/numprocs, MPI_INT, 0,
        MPI_COMM_WORLD);
33
```

```
34
35   /* Búsqueda local */
36   my_count = 0
37   #pragma omp parallel for reduction(+:my_count)
38   for(i=0; i<N/numprocs; i++)
39     if (my_a[i]==key)
40         my_count++
41
42   MPI_Reduce(&my_count, &count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
         ;
43
44
45   if (my_id==0) /* Master */
46     free(a);
47   else          /* Worker */
48     free(my_a);
49
50   printf("El total de claves es: %d\n", count);
51 }
```

Código 5.32: Ejemplo de implementación híbrida con `OpenMP` y `MPI`.

Como podemos observar, en la parte de búsqueda hemos incorporado el `pragma omp parallel for` para poder generar los *threads* y distribuir las iteraciones de forma estática entre los *threads*.

Resumen

En este módulo hemos realizado un repaso de los diferentes niveles de paralelismo que se han incorporado al *hardware* de un uniprocesador con la intención de mejorar el rendimiento de las aplicaciones. Sin embargo, también hemos visto que, debido a que las ganancias de rendimiento no eran las esperadas y además el incremento del consumo energético era significativo, la estrategia seguida hasta el año 2000 de mejorar los uniprocesadores no se podía mantener. A partir de este punto, hemos descrito la clasificación de las máquinas paralelas según Flynn, centrándonos en ver cómo se pueden paralelizar las aplicaciones y cómo se mide el rendimiento y eficiencia de éstas.

En particular, hemos visto que para paralelizar una aplicación necesitamos mecanismos de creación de tareas, y también tener en cuenta los problemas de concurrencia que surgen como consecuencia de esta paralelización. Como parte de los mecanismos de creación de tareas, se ha realizado una introducción a los modelos de programación paralela más conocidos para memoria distribuida o paso de mensajes (MPI), y memoria compartida (OpenMP); pero también hemos explicado otro modelo de programación más avanzado para ayudar al programador en el control automático de las dependencias entre tareas (OmpSs).

También hemos detallado los pasos para analizar el paralelismo potencial existente en una aplicación, y diferentes estrategias de distribución de trabajo en tareas y/o en datos, con tal de realizar una paralelización eficiente y adecuada de las aplicaciones. Como parte de este análisis, se ha descrito un modelo básico de comunicación para paso de mensajes, con el que hemos podido estudiar la paralelización de dos programas paralelos para poder maximizar la eficiencia de éstos, usando o no la técnica de *blocking*.

Finalmente, hemos realizado un caso de estudio sencillo con el objetivo de ver cómo se puede paralelizar un mismo programa utilizando diferentes estrategias y modelos de programación paralela.

Ejercicios de autoevaluación

1. Tenemos que realizar la suma de los elementos de un vector $X[0]$ ... $X[n-1]$. El algoritmo secuencial se muestra en el código 5.33.

```
1 sum = 0;
2 for (i=0; i<n ; i++)
3    sum += X[i]
```

Código 5.33: Suma de elementos de un vector.

Dibujad el grafo de dependencias de tareas, calculad T_1, T_∞ y el paralelismo potencial.

2. Para solucionar el mismo problema del ejercicio anterior, es decir, la suma de los elementos de un vector $X[0]$ ... $X[n-1]$, se podría plantear una solución en árbol, de tal forma que primero se suma por pares, después grupos de dos pares, etc. Plantead una solución iterativa y una solución recursiva para solucionar el problema de esta forma. Después dibujad el grafo de dependencias de tareas, calculad T_1, T_∞ y el paralelismo potencial.

3. Realizad las versiones paralelas con OpenMP de las soluciones del ejercicio anterior.

4. Tenemos que hacer un algoritmo que calcule el producto matriz por vector, que se muestra en el código 5.34.

```
1 for (i=0; i<n; i++)
2     y[i]=0;
3
4 for (i=0; i<n ; i++)
5   for (j=0; j<n ; j++)
6     for (k=0; k<n ; k++)
7       y[i]+= A[i][k] * b[k];
```

Código 5.34: Producto matriz por vector.

Para las siguientes granularidades en la paralelización, calculad T_1, T_∞ y el paralelismo.

- Grano muy fino: cada iteración de los bucles más internos es una tarea.
- Grano fino: el cálculo de todo un elemento $y[i]$ es una tarea.
- Grano grueso: el cálculo de tres elementos consecutivos de $y[i]$ es una tarea.

5. Programad una solución paralela con paso de mensajes en MPI del código 5.31 en la que el número de elementos en el vector no tenga que ser múltiplo del número de procesos, haciendo que el desbalanceo de carga entre los procesos sea mínimo.

6. Programad una solución paralela con paso de mensajes en `MPI` del código 5.35, realizando una descomposición geométrica de la matriz u y $uhelp$. La inicialización y la distribución de los datos las hará el proceso `master`. Después de realizarse todo el cálculo, el proceso `master` deberá recibir el resultado y escribir la matriz resultante. Cada proceso se encargará del cálculo de N/P filas consecutivas, donde N es el número de filas y P el número de procesos. N puede no ser múltiplo de P.
Escribid también el programa principal donde se vea cómo obtenéis memoria para la matriz, se hace la inicialización, la distribución de los datos, y la recepción de los datos de los procesos.

```
1 #include <math.h>
2 void compute( int N, double *u, double *uhelp) {
3     int i, k;
4     double tmp;
5
6     for ( i = 1; i < N-1; i++ ) {
7        for ( k = 1; k < N-1; k++ ) {
8           tmp = u[N*(i+1) + k] + u[N*(i-1) + k] + u[N*i + (k+1)] + u[N*
                 i + (k-1)] - 4 * u[N*i + k];
9           uhelp[N*i + k] = tmp/4;
10       }
11    }
12 }
```

Código 5.35: Código de Stencil

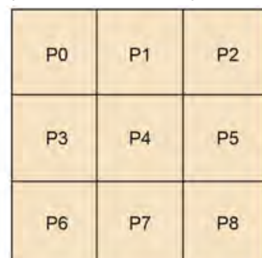
7. Programad una solución paralela con paso de mensajes en MPI del código 5.36, realizando una descomposición geométrica de la matriz u y aplicando la técnica de *blocking*. La inicialización y la distribución de los datos las hará el proceso `master`. Después de realizarse todo el cálculo, el proceso `master` deberá recibir el resultado y escribir la matriz resultante. Cada proceso se encargará del cálculo de N/P filas consecutivas, donde N es el número de filas y P el número de procesos. N puede no ser múltiplo de P. Escribid también el programa principal donde se vea cómo obtenéis memoria para la matriz, se hace la inicialización, la distribución de los datos y la recepción de los datos de los procesos.

```
1  #include <math.h>
2  void compute( int N, double *u) {
3      int i, k;
4      double tmp;
5
6      for ( i = 1; i < N-1; i++ ) {
7         for ( k = 1; k < N-1; k++ ) {
8            tmp = u[N*(i+1) + k] + u[N*(i-1) + k] + u[N*i + (k+1)] + u[N*
                 i + (k-1)] - 4 * u[N*i + k];
9            u[N*i + k] = tmp/4;
10        }
11     }
12 }
```

Código 5.36: Código de Stencil

8. Dado el código 5.36 del ejercicio anterior. Para las siguientes cuestiones suponed que (1) estamos trabajando sobre una máquina de memoria distribuida con paso de mensajes, donde cada mensaje de n elementos tiene un coste de $t_{comm} = t_s + n \times t_w$, (2) que la matriz u está distribuida en bloques por los procesadores (P^2) tal y como se muestra en la figura 44 para el caso de $P = 3$, y (3) que el cuerpo de bucle tiene un coste de t_c y que la matriz tiene n filas y n columnas.

Figura 44. Distribución de los bloques de la matriz u por los P^2 procesadores.



- ¿Qué tiempo de cálculo tiene que realizar cada procesador?

- Indicad qué datos tiene que recibir y enviar a otros procesadores cada procesador. También indicad cuándo debería hacerlo.

- Suponiendo que cada procesador puede empezar el cálculo de sus datos cuando tiene ya todos los datos en la memoria local:

– Dibujad un diagrama temporal de ejecución (cálculo y comunicación) de los procesadores.

– Obtened el modelo de ejecución de tiempo (cálculo y comunicación).

9. Analizad los siguientes códigos e indicad si hay dependencia de datos. Para cada dependencia de datos indicad de qué tipo es, su distancia y de dónde a dónde va. Finalmente, una vez analizadas las dependencias de datos, indicad qué códigos son vectorizables y escribid su código vectorial. N es una constante que puede tener cualquier valor.

```
a)
char *a, *b, i;
...
for (i=16; i<N; i++)
    a[i] = a[i-16] + b[i];
```

```
b)
double *a, i;
...
for (i=N-2; i>=3; i--)
     a[i] = a[i+2] + a[i-3];
```

```
c)
int *a, *b, i;
...
for (i=1; i<(N-1); i++)
     a[i] = a[i-1] + b[i];
```

Bibliografía

Chapman, Barbara; Jost, Gabriele; Pas, Ruud van der (2008). *Using `OpenMP`*. The MIT Press.

Grama, Ananth; Gupta, Anshul; Karypis, George; Kumar, Vipin (2003). *Introduction to Parallel Computing. Second Edition.*. Pearson: Addison Wesley.

Jost, Gabriele; Jin, H.; Mey, D.; Hatay, F. (2003). *Comparing `OpenMP`, `MPI`, and Hybrid Programming*. Proc. Of the 5th European Workshop on OpenMP.

Mattson, Timothy G.; Sanders, Beverly A.; Massingill, Berna L. (2009). *Patterns for Parallel Programming*. Pearson: Addison Wesley.

Pacheco (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

Quinn, M. J. (2008). *Parallel Programming in C with `MPI` and `OpenMP`*. McGraw-Hill.

Tanenbaum, Andrew S. (2006). *Structured Computer Organization*. Pearson, Addison Wesley.