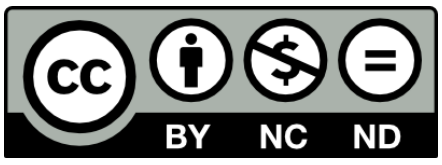
Arquitectura de Computadores

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Arquitectura de Computadores

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2023-2024

Índice general

1. Arquitecturas Paralelas: Clasificación y Prestaciones

1.1. Clasificación del paralelismo implícito en una aplicación

1.1.1. Objetivos

Como items a conocer en esta sección, destacamos:

- Conocer las clasificaciones usuales del paralelismo implícito en una aplicación. Distinguir entre paralelismo de tareas y paralelismo de datos.
- Distinguir entre las dependencias RAW, WAW y WAR.
- Distinguir entre *thread* y proceso.
- Relacionar el paralelismo implícito en una aplicación con el nivel en el que se hace explícito para que se pueda utlizar (instrucción, thread, proceso) y con las arquitecturas paralelas que lo aprovechan.

1.1.2. Niveles y tipos de paralelismo implícito en una aplicación

En una aplicación, podemos encontrar distintos niveles de paralelismo. Para facilitar su comprensión, trataremos de clasificarlos en esta parte inicial de la asignatura. Comenzaremos por marcar varias capas de abstracción que se siguen a la hora de desarrollar la aplicación, lo que los facilitará marcar el paralelismo dentro de esta.

Podemos considerar que un programa está compuesto de funciones, las cuales a su vez están compuestas de bloques de código en la que abundan los bucles (para simplificar esto, diremos que las funciones están compuestas de bucles). Los cuales están basados en operaciones. Asimismo, puede que nuestra aplicación esté compuesta por distintos programas (como en el caso de LibreOffice de LibreOffice Writer, LibreOffice Calc, ...). Por todo esto, nos es natural tratar de clasificar el paralelismo de una aplicación en función de distintos niveles, los cuales serán:

- Nivel de programas.
- Nivel de funciones.
- Nivel de bucles (de bloques).

- Nivel de operaciones.

En general, el paralelismo lo podremos encontrar en distinta granularidad (en mayor o menor medida) en relación al nivel en el que nos encontremos. Para detectar mejor este grade de paralelismo, es cómodo tener una clara distinción del tipo de paralelismo (como estamos haciendo), lo que facilita la tarea del programador y del compilador. Destacamos la ventaja de poder manipular el código secuencial (que ya sabemos manejar) en código con funcionalidades paralelas, lo que nos libra de tener que conocer tecnologías nuevas para poder implementar paralelismo en nuestras aplicaciones.

A continuación, justificamos los niveles ya elegidos, junto con ejemplos de paralelismo en cada uno de ellos:

Nivel de programas: Los diferentes programas que intervienen en una aplicación (o incluso en diferentes aplicaciones) se pueden ejecutar en paralelo, debido a que es poco probable que existan dependencias entre ellos.

Nivel de funciones: Un nivel de abstracción más bajo; las funciones llamadas en un programa se pueden ejecutar en paralelo, siempre que no haya dependencias (riesgos) inevitables entre ellos, como dependencias de datos verdaderas (RAW). Como ejemplo, recomendamos la familiarización de la directiva `#pragma omp parallel sections` de OpenMP de la Sesión 1 de Prácticas, donde descubrimos el paralelismo a nivel de funciones de forma explícita.

Nivel de bucles (de bloques): Una función puede estar basada en la ejecución de uno o varios bucles. En muchas ocasiones, el código que se encuentra dentro de un bucle no está íntegramente asociado con la iteración en sí; sino que deseamos que una cierta tarea se ejecute un cierto número de veces. Se pueden ejecutar en paralelo las iteaciones de un bucle, siempre que eliminen los problemas derivados de las dependencias de datos verdaderas (RAW).

Nivel de operaciones: En este nivel se extra el paralelismo disponible entre operaciones. Las operaciones independientes se pueden ejecutar en paralelo. Por otra parte, podemos encontrar instrucciones compuestas de varias operaciones que se aplican en secuencial mismo tipo de datos de entrada. Por ejemplo, la instrucción `mac` nos permite realizar una suma tras una multiplicación. En este nivel se puede detectar la posiblidad de usar instrucciones compuestas como la ya mencionada.

A esta clasificación del paralelismo que se puede detectar en distintos niveles de un código secuencial se le denomina *paralelismo funcional*. Por otra parte, podemos hablar de *paralelismo de tareas* y de *paralelismo de datos*.

Paralelismo de tareas

En inglés, Task Level Parallelism (TLP). Este paralelismo se encuentra extrayendo la estructura lógica de funciones de la aplicación. Esta estructura está formada por las funciones, siendo las conexiones entre ellas el flujo de datos entre funciones. El paralelismo a nivel de funciones antes descrito en el paralelismo funcional equivale al paralelismo de tareas.

Paralelismo de datos

En inglés, Data Level Parallelism (DLP). El paralelismo de datos se encuentra implícito en las operaciones con estructuras de datos (como vectores y matrices). Las operaciones vectoriales y matriciales engloban operaciones con diversos escalares, las cuales se pueden realizar en paralelo. Como estas operaciones se suelen implementar por bucles, decimos que el paralelismo de datos es equivalente al paralelismo a nivel de operaciones en el paradigma del paralelismo funcional. Por ejemplo, contamos con las instrucciones SIMD (se desarrollarán próximamente), que con una instrucción puede manipular múltiples datos. Un ejemplo de instrucción SIMD es la implementación de una instrución que pueda sumar dos vectores de datos enteros.

Por ejemplo, si tenemos una aplicación que nos permite decodificar el formato de imagen JPEG a formato RGB para imprimir en pantalla, podemos encontrar paralelismo de tareas al tener distintos módulos que realizan cada uno de los pasos intermedios para realizar dicha transformación; mientras que disponemos de paralelismo a nivel de datos en las operaciones, al tener instrucciones que nos permitan sumar (con una sola instrucción) dos vectores.

Granularidad

El paralelismo también puede clasificarse en función de la granularidad de la tarea a realizar. Esto es, de la magnitud del número de operaciones a realizar. Esta se suele hacer corresponder con los distintos niveles de paralelismo funcional anteriormente desarrollado. Ilustramos esta relación uno a uno en la siguiente enumeración:

- Grano grueso: Nivel de programas.
- Grano medio: Nivel de funciones.
- Grano fino-medio: Nivel de bucles.
- Grano fino: Nivel de operaciones.

Dependencias de datos

Constantemente estamos haciendo alusión a las dependencias de datos, pero no nos hemos parado a plantear cuando una sección de código B_2 presenta dependencias de datos con respecto a un bloque de código B_1. Para que se produzca una dependencia de datos entre ellos:

- Deben hacer referencia a una misma variable (una misma posición de memoria).
- Un bloque de código debe aparecer en la secuencia de código antes que el otro.

Una vez que conocemos que existe una dependencia de datos entre dos bloques de código nos surge la cuestión de si cualquier dependencia es igual de importante, de si hay dependencias evitables y de si hay otras que no lo son. Respondemos a todo tipando las dependencias de datos:

RAW (Read After Write): También llamada dependencia verdadera, sucede cuando tratamos de leer una variable (equivalentemente, posición de memoria) después de haberla modificado (de haberla escrito). Recordemos que nos encontramos en el paradigma de la paralelización: tratamos de hacer esto de forma paralela, luego puede que un ente encargado de leer la variable lo haga antes que el encargado de modificarla, haciendo invisible dicha modificación (no existente cuando se leyó) y causando condiciones de carrera junto con un posible mal funcionamiento del programa (así como de romper el esquema determinista de este). Podemos ver un ejemplo de RAW en el siguiente ejemplo:

```
int a = b * c;
int d = a + c;
```

Tenemos en la segunda línea el uso (lectura) de la variable `a`, tras modificarla (escribir en ella) en la primera línea. Si empleamos paralelismo puede suceder que se ejecute la segunda línea antes que la primera, provocando condiciones catastróficas. De hecho, cuando esto se haga, la variable `a` no estará ni siquiera inicializada.

WAW (Write after Write): También llamada anti-dependencia, sucede cuando tratamos de modificar una variable por segunda vez (después de haberla modificado ya). Esto puede plantear, al igual que explicábamos en RAW, condiciones de carrera. Mostramos un ejemplo a continuación:

```
a = b * c;
// se lee a
a = d + e;
```

Donde en la primera y tercera linea modificamos el valor de `a`. Sin embargo, esta dependencia es evitable, ya que si cambiamos el nombre de la variable (empleamos una dirección de memoria distinta), evitamos la dependencia. Por tanto, a esta dependencia también se le llama dependencia de nombre, al no ser una dependencia de datos real.

WAR (Write after Read): También llamda dependencia de salida, sucede cuando tratamos de escribir en una variable tras leer de ella. Esto también puede provocar condiciones de carrera, tal y como vemos en el siguiente ejemplo:

```
b = a + 1;
a = d + e;
```

Donde en la primera línea leemos `a` y en la segunda modificamos su valor. Sin embargo, esta dependencia también recibe el nombre de dependencia de nombre, ya que puede solucionarse con un sencillo cambio de nombre, por lo que no se trata de una dependencia de datos real. Cabe destacar que esto lo suele realizar de forma automática el compilador.

1.1.3. Unidades de ejecución: instrucciones, hebras y procesos

El hardware es el encargado de la administración y ejecución de las instrucciones, mientras que a nivel superior nos encontramos con el SO, haciéndolo (no en el sentido que estás pensando) con las hebras y los procesos. Cada proceso en ejecución tiene su propia asignación de memoria. Los SO multihebra permiten que un proceso se conforme por una o varias hebras (o hilos). Cada hebra tiene su propia pila y banco de registros, mientras que comparte con sus hermanas la memoria que les oferta el proceso. Esto permite que las hebrs puedan crearse, destruirse y comunicarse entre ellas de una forma más rápida que los procesos. Todo esto permite que las hebras dispongan de una menor granularidad que estos.

Esta sección nos ha servido para repasar entes que nos permiten hacer explícito el paralelismo, los cuales simplificarán el diseño de las aplicaciones, al ser las hebras y procesos automáticamente gestionadas por el sistema operativo; y las instrucciones por la arquitectura.

1.1.4. Relación entre paralelismo implícito, explícito y arquitecturas paralelas. Implementación del paralelismo

A lo largo de este documento hemos hecho referencia en varias ocasiones al paralelismo implícito y explícito, sin nunca pararnos a desarrollar de qué estamos hablando. Es ahora la ocasión de hacerlo.

Paralelismo implícito. Se trata de aquellas acciones que automáticamente se llevan a cabo (ya sea gracias al hardware, sistema operativo o compilador) de forma paralela.

Paralelismo explícito. Se trata de aquellas acciones que deseamos que se hagan de forma paralela, y que obligamos a ello de forma explícita, como por ejemplo, con la ayuda de una API en el caso de las prácticas con OpenMP.

Esta diferencia la comentaremos en la siguiente subsección, que será fácil de comprender junto con el desarrollo de las prácticas.

Hecha esta distinción, comenzamos ahora sí con esta subsección, en la que podemos cómo se implementa el paralelismo implícito, así como el explícito, de una forma superficial. Además, será necesario indicar las especificaciones hardware requeridas para llevar esto a cabo. Usaremos el paralelismo funcional, ya que para eso lo hemos desarrollado al inicio.

Nivel de programas El paralelismo entre programas se implementa mediante diversos procesos: en el momento que se ejecuta un nuevo programa, se crea el programa asociado a él, y ya sólo dependerá del sistema operativo el llevar a cabo su paralelización con el resto de procesos[1]. Para poder implementar este tipo de paralelismo, es necesario disponer de un multiprocesador o multicomputador.

[1]Mediante técnicas ya vistas en la asignatura de Sistemas Operativos

Nivel de funciones El paralelismo a este nivel puede extraerse para realizarse a nivel de procesos (si la función realmente lo require) o de hebras, de forma que cada hebra (o proceso) ejecute una o varias funciones. Para ello, necesitaremos de un multiprocesador y, en caso de requerir hebras, será conveniente que este sea multihebra (o en su defecto, contar con una biblioteca de hebras, aunque esto es menos recomendable).

Nivel de bucles Este se puede realizar a nivel de procesos o hebras, tal y como se hacía en el nivel anterior. Sin embargo, el paralelismo a este nivel también puede implementarse con instrucciones en el caso de, por ejemplo, sumas de vectores. Para ello, debemos contar con un multiprocesador (a poder ser, multihebra) y para el último caso considerado, una arquitectura SIMD que permite realizar trabajos similares con vectores y, en general, estructuras de datos.

Nivel de operaciones El paralelismo entre operaciones se puede aprovechar en arquitecturas con operaciones a nivel de instrucción (ILP), ejecutando en paralelo las instrucciones asociadas a operaciones independientes. Para ello, es claro que necesitamos arquitecturas ILP.

1.1.5. Detección, utilización y extracción del paralelismo

En los procesadores ILP superescalares o segmentados la arquitectura en sí misma extrae paralelismo (o como nosotros hemos llamado, implementa paralelismo implícito). Para ello, eliminan dependencias de datos falsas (no del tipo RAW) entre instrucciones y evitan problemas debidas a dependencias de datos, de control y de recursos.

Además, el grado de paralelismo de las instrucciones se puede incrementar con las ayudas del compilador y del programador. En general, se puede definir el grado de paralelismo de un conjunto de entradas a un sistema como el número máximo de entradas del conjunto que se pueden ejecutar en paralelo.

A continuación, para cada tipo de paralelismo, tratamos de explicar la extracción del paralelismo. Esto es, explicar qué ente lo lo detecta, cómo se implemente y en qué unidad se ejecuta. En este caso, la granularidad es inversamente proporcional a la facilidad de extracción del paralelismo.

Nivel de operaciones Puede ser detectado por la arquitectura del hardware, por herramientas de programación (como IDEs o compiladores) y por el programador. Se implementa o aprovecha principalmmente por arquitecturas ILP, que lo hacen usando instrucciones dedicadas a ello.

Nivel de bucles La arquitectura ya escapa a este nivel de abstracción, por lo que sólo podemos detectarlo mediante herramientas de programación o por la destreza del programador. Se implementa a nivel de arquitecturas SIMD mediante intrainstrucciones en el caso de aquellas paralelizaciones vectoriales ya comentadas; mientras que paralelizaciones del estio TLP se implementan mediante multiprocesador multihebra o multicomputadores, usando threads o procesos.

Nivel de funciones A este nivel ya sólo disponemos del programador para llevar la detección a cabo, quien puede hacer el paralelismo explícito mediante multiprocesadores multihebra, multiprocesadores y multicomputadores; mediante hebras y/o procesos.

Nivel de programas El programador puede hacer explícito el paralelismo si dispone de un multiprocesador o multicomputador, mediante el uso de procesos.

1.2. Clasificación de arquitecturas paralelas

1.2.1. Objetivos

Una vez terminada la sección que acabamos de comenzar, tratamos de que el lector sea capaz de:

- Distinguir entre procesamiento o computación paralela y distribuida.
- Clasificar los computadores según segmento del mercado.
- Distinguir entre las diferentes clases de arquitecturas de la clasificación de Flynn.
- Diferenciar un multiprocesador de un multicomputador.
- Distinguir entre NUMA y SMP.
- Distinguir entre arquitecturas DLP, ILP y TLP.
- Distinguir entre arquitecturas TLP con una instancia de SO y TLP con varias instancias de SO.

1.3. Evaluación de prestaciones

1.3.1. Objetivos

En esta sección, aprenderemos a:

- Distinguir entre tiempo de CPU (sistema y usuario) de Unix y el tiempo de respuesta.
- Distinguir entre productividad y tiempo de respuesta.
- Obtener, de forma aproximada mediante cálculos, el tiempo de CPU, GFLOPS y los MIPS del código ejecutado en un núcleo de procesamiento.
- Calcular la ganancia en prestaciones/velocidad.
- Aplicar la ley de Amdahl.

2. Relaciones de Problemas

2.1. Arquitecturas Paralelas

Ejercicio 2.1.1. En el código de prueba (benchmark) que ejecuta un procesador no segmentado que funciona a 300 MHz, hay un 20 % de instrucciones `LOAD` que necesitan 4 ciclos, un 10 % de instrucciones `STORE` que necesitan 3 ciclos, un 25 % de instrucciones con operaciones de enteros que necesitan 6 ciclos, un 15 % de instrucciones con operandos en coma flotante que necesitan 8 ciclos por instrucción, y un 30 % de instrucciones de salto que necesitan 3 ciclos.

1. ¿Cuál es la ganancia que se puede obtener por reducción a 3 ciclos de las instrucciones con enteros?

   Resumimos los datos del enunciado en la siguiente tabla:

I_i	CPI_i^b	NI_i
`LOAD`	4 ciclos	0,2 NI
`STORE`	3 ciclos	0,1 NI
`FX. POINT`	6 ciclos	0,25 NI
`FLT. POINT`	8 ciclos	0,15 NI
`BRANCH`	3 ciclos	0,3 NI

   donde I_i es el tipo de instrucción, CPI_i^b es el número de ciclos por instrucción y NI_i es el número de instrucciones de ese tipo.

   El tiempo base T_b que tardaría en ejecutarse el programa sin mejoras sería:

$$\begin{aligned} T_b &= NI \cdot CPI \cdot T_c = T_c \cdot \sum_i NI_i \cdot CPI_i = \\ &= T_c \cdot NI \cdot \left(\underbrace{0{,}2 \cdot 4}_{LD} + \underbrace{0{,}1 \cdot 3}_{ST} + \underbrace{0{,}25 \cdot 6}_{FP} + \underbrace{0{,}15 \cdot 8}_{FLT\ POINT} + \underbrace{0{,}3 \cdot 3}_{BRANCH} \right) = \\ &= T_c \cdot NI \cdot 4{,}7 \end{aligned}$$

   donde T_c representa el tiempo de ciclo. Respecto al tiempo mejorado T_p, sabiendo ahora que en caso de los números enteros el número de ciclos se reduce

a 3, tendríamos:

$$T_p = NI \cdot CPI \cdot T_c = T_c \cdot \sum_i NI_i \cdot CPI_i =$$

$$= T_c \cdot NI \cdot \left(\underbrace{0{,}2 \cdot 4}_{LD} + \underbrace{0{,}1 \cdot 3}_{ST} + \underbrace{0{,}25 \cdot 3}_{FP} + \underbrace{0{,}15 \cdot 8}_{FLT\ POINT} + \underbrace{0{,}3 \cdot 3}_{BRANCH} \right) =$$

$$= T_c \cdot NI \cdot 3{,}95$$

La expresión de la ganancia, por tanto, es:

$$S = \frac{T_b}{T_p} = \frac{\cancel{T_c} \cdot \cancel{NI} \cdot 4{,}7}{\cancel{T_c} \cdot \cancel{NI} \cdot 3{,}95} = \frac{4{,}7}{3{,}95} \approx 1{,}1898$$

2. ¿Cuál es la ganancia que se puede obtener por reducción a 3 ciclos de las instrucciones en coma flotante?

   Tenemos que:

$$T_p = NI \cdot CPI \cdot T_c = T_c \cdot \sum_i NI_i \cdot CPI_i =$$

$$= T_c \cdot NI \cdot \left(\underbrace{0{,}2 \cdot 4}_{LD} + \underbrace{0{,}1 \cdot 3}_{ST} + \underbrace{0{,}25 \cdot 6}_{FP} + \underbrace{0{,}15 \cdot 5}_{FLT\ POINT} + \underbrace{0{,}3 \cdot 3}_{BRANCH} \right) =$$

$$= T_c \cdot NI \cdot 4{,}25$$

   La expresión de la ganancia, por tanto, es:

$$S = \frac{T_b}{T_p} = \frac{\cancel{T_c} \cdot \cancel{NI} \cdot 4{,}7}{\cancel{T_c} \cdot \cancel{NI} \cdot 4{,}25} = \frac{4{,}7}{4{,}25} \approx 1{,}10599$$

Ejercicio 2.1.2. Un circuito que implementaba una operación en un tiempo de $Top = 450$ ns se ha segmentado mediante un cauce lineal con cuatro etapas de duración $T1 = 100$ ns, $T2 = 125$ ns, $T3 = 125$ ns y $T4 = 100$ ns respectivamente, separadas por un registro de acoplo que introduce un retardo de 25 ns.

1. ¿Cuál es la máxima ganancia de velocidad posible? ¿Cuál es la productividad máxima del cauce?

   Tenemos que el ciclo de reloj es de $T_c = 125ns + 25ns = 150ns$. La ganancia de velocidad es:

$$S(N) = \frac{T^b(N)}{T^s(N)} = \frac{N \cdot T_R}{TLI + (N-1)T_c} = \frac{N \cdot T_R}{4 \cdot T_c + (N-1)T_c}$$

   donde el 4 se debe a que es el número de etapas del cauce.

   La ganancia máxima es:

$$S(N \ggg) = \lim_{N \to \infty} S(N) = \lim_{N \to \infty} \frac{N \cdot T_R}{4 \cdot T_c + (N-1)T_c} = \frac{T_R}{T_c}$$

2. ¿A partir de qué número de operaciones ejecutadas se consigue una productividad igual al 90 % de la productividad máxima?

Ejercicio 2.1.3. En un procesador sin segmentación de cauce, determine cuál de estas dos alternativas para realizar un salto condicional es mejor:

- ALT1: Una instrucción `COMPARE` actualiza un código de condición y es seguida por una instrucción `BRANCH` que comprueba esa condición. Se usan dos intrucciones.
- ALT2: Una sola instrucción incluye la funcionalidad de las instrucciones `COMPARE` y `BRANCH`. Se usa una única instrucción.

Hay que tener en cuenta que hay un 20 % de instrucciones `BRANCH` para ALT1 en el conjunto de programas de prueba; que las instrucciones `BRANCH` en ALT1 y `COMPARE+BRANCH` en ALT2 necesitan 4 ciclos mientras que todas las demás necesitan sólo 3; y que el ciclo de reloj de la ALT1 es un 25 % menor que el de la ALT2, dado que en este caso la mayor funcionalidad de la instrucción `COMPARE+BRANCH` ocasiona una mayor complejidad en el procesador.

Como el tiempo de ciclo de reloj depende de la ejecución más lenta, es normal que este cambie (como se especifica en el enunciado). La relación entre estos es la siguiente:

$$T_c^1 = T_c^2 - 0{,}25T_c^2 = 0{,}75T_c^2$$

Resumimos los datos del enunciado en la siguiente tabla:

I_i^1	CPI_i^1	NI_i^1	I_i^2	CPI_i^2	NI_i^2
br	4 ciclos	$0{,}2 \cdot NI^1$	cmpbr	4 ciclos	$0{,}2 \cdot NI^1$
cmp	3 ciclos	$0{,}2 \cdot NI^1$			
Demás	3 ciclos	$0{,}6 \cdot NI^1$	Demás	3 ciclos	$0{,}6 \cdot NI^1$
		NI^1			$0{,}8 \cdot NI^1 = NI^2$

Tenemos que ver qué alternativa nos da un tiempo de ejecución menor (tengamos en cuenta que el tiempo de ciclo de cada uno no es el mismo, por lo que tenemos que pasarlo todo al mismo tiempo de ciclo):

$$T_{CPU}^1 = NI^1 \cdot (\underbrace{0{,}2 \cdot 4c}_{br} + \underbrace{0{,}8 \cdot 3c}_{cmp+Resto}) \cdot T_c^1 = NI^1 \cdot 3{,}2 \cdot 0{,}75 \cdot T_c^2 = NI^1 \cdot 2{,}4 \cdot T_c^2$$

$$T_{CPU}^2 = NI^1 \cdot (\underbrace{0{,}2 \cdot 4c}_{cmpbr} + \underbrace{0{,}6\dot{3}c}_{Resto}) \cdot T_c^2 = NI^1 \cdot 2{,}6 \cdot T_c^2$$

Por ser $T_{CPU}^1 < T_{CPU}^2$, concluimos que la opción ALT1 es la mejor, en cuanto a tiempos de ejecución.

Ejercicio 2.1.4. ¿Qué ocurriría en el problema del ejercicio anterior (Ejercicio 2.1.3) si el ciclo de reloj fuese únicamente un 10 % mayor para la ALT2?

Ejercicio 2.1.5. Considere un procesador no segmentado con una arquitectura de tipo `LOAD/STORE` en la que las operaciones sólo utilizan como operandos registros de la CPU. Para un conjunto de programas representativos de su actividad se tiene que el 43 % de las instrucciones son operaciones con la ALU (3 CPI), el 21 % `LOAD`s (4 CPI), el 12 % `STORE`s (4 CPI) y el 24 % `BRANCH`s (4 CPI). Se ha podido comprobar que un 25 % de las operaciones con la ALU utilizan operandos en registros que no se vuelven a utilizar. Compruebe si mejorarían las prestaciones si, para sustituir ese 25 % de operaciones, se añaden instrucciones con un dato en un registro y otro en memoria. Tengan en cuenta en la comprobación que para estas nuevas instrucciones el valor de CPI es 4 y que añadirlas ocasiona un incremento de un ciclo en el CPI de los `BRANCH`, pero no afectan al ciclo de reloj.

Resumimos los datos del enunciado en la siguiente tabla:

I_i^1	CPI_i^1	NI_i^1	I_i^2	CPI_i^2	NI_i^2
Instrucción `ALU`	3 ciclos	$0{,}43 \cdot NI$	`ALU r,r`	3 ciclos	$0{,}2325 \cdot NI^1$
			`ALU r,m`	4 ciclos	$0{,}1075 \cdot NI^1$
`LOAD`s	4 ciclos	$0{,}21 \cdot NI^1$	`LOAD`s	4 ciclos	$0{,}1225 \cdot NI^1$
`STORE`s	4 ciclos	$0{,}12 \cdot NI^1$	`STORE`s	4 ciclos	$0{,}12 \cdot NI^1$
`BRANCH`s	4 ciclos	$0{,}24 \cdot NI^1$	`BRANCH`s	5 ciclos	$0{,}24 \cdot NI^1$
					$0{,}8925 \cdot NI^1 = NI^2$

Donde para completar la columna NI_i^2 hamos usado que:

$$NI^1{}_{\texttt{ALU r,r}} = 0{,}75(0{,}43 \cdot NI^1) = 0{,}2325 \cdot NI^1$$
$$NI^1{}_{\texttt{ALU r,m}} = 0{,}25(0{,}43 \cdot NI^1) = 0{,}1075 \cdot NI^1$$
$$NI^1{}_{\texttt{LOADs}} = 0{,}21 \cdot NI^1 - 0{,}1075 \cdot NI^1 = 0{,}1225 \cdot NI^1$$

Calculamos los tiempos en CPU:

$$\begin{aligned} T_{CPU}^1 &= NI^1(\overbrace{0{,}43 \cdot 3c}^{\texttt{ALU r,r}} + (0{,}21 + 0{,}12 + 0{,}24) \cdot 4c) \cdot T_c \\ &= NI^1 \cdot 3{,}57 \cdot T_c \end{aligned}$$

$$\begin{aligned} T_{CPU}^2 &= NI^1(0{,}3235 \cdot 3c + (0{,}1075 + 0{,}1225 + 0{,}12) \cdot 4c + 0{,}24 \cdot 4c) \cdot T_c \\ &= NI^1 \cdot 3{,}4875 \cdot T_c \end{aligned}$$

Y tenemos que $T_{CPU}^2 < T_{CPU}^1$, luego sí que mejorarían las prestaciones.

Ejercicio 2.1.6. Se ha diseñado un compilador para la máquina `LOAD/STORE` del problema anterior (Ejercicio 2.1.5). Ese compilador puede reducir en un 50 % el número de operaciones con la ALU, pero no reduce el número de `LOAD`s, `STORE`s, y `BRANCH`s. Suponiendo que la frecuencia de reloj es de 50 Mhz, ¿Cuál es el número de MIPS y el tiempo de ejecución que se consigue con el código optimizado? Compárelos con los correspondientes del código no optimizado.

Ejercicio 2.1.7. En un programa que se ejecutan en un procesador no segmentado que funciona a 100 MHz, hay un 20 % de instrucciones `LOAD` que necesitan 4 ciclos,

un 15 % de instrucciones `STORE` que necesitan 3 ciclos, un 40 % de instrucciones con operaciones en la ALU que necesitan 6 ciclos, y un 25 % de instrucciones de salto que necesitan 3 ciclos.

1. Si en las instrucciones que usan la ALU el tiempo en la ALU supone 4 ciclos, determine cuál es la máxima ganancia que se puede obtener si se mejora el diseño de la ALU de forma que se reduce su tiempo de ejecución a la mitad de ciclos.

2. Con qué porcentaje de instrucciones con operaciones en la ALU se podría haber obtenido en los cálculos del apartado 1 una ganancia mayor que 2? Razone su respuesta.

Ejercicio 2.1.8. Suponga que en los programas que constituyen la carga de trabajo habitual de un procesador las instrucciones de coma flotante consumen un promedio del 13 % del tiempo del procesador.

1. Ha aparecido en el mercado una nueva versión del procesador en la que la única mejora con respecto a la versión anterior es una nueva unidad de coma flotante que permite reducir el tiempo de las instrucciones de coma flotante a tres cuartas partes del tiempo que consumían antes. ¿Cuál es la máxima ganancia de velocidad que puede esperarse en los programas que constituyen la carga de trabajo si se utiliza la nueva versión del procesador?

   La p de la ley sería $3/4$ y $f = 0{,}87$. Para calcular la ganancia adicional:

   $$S = \frac{T_b}{T_p} = \frac{\cancel{T_b}}{0{,}87\cancel{T_b} + 0{,}13\cancel{T_b} \cdot 3/4} = 1{,}033$$

2. ¿Cuál es la máxima ganancia de velocidad con respecto a la versión inicial del procesador que, en promedio, puede esperarse en los programas debido a mejoras en la velocidad de las operaciones en coma flotante?

   Lo que esperamos en el mejor caso es llevar a 0 el tiempo que tarda en ejecutarse la parte de punto flotante.

   $$S = \frac{T_b}{T_p} = \frac{\cancel{T_b}}{0{,}87\cancel{T_b} + 0} = 1{,}149$$

3. ¿Cuál debería ser el porcentaje de tiempo de cálculo con datos en coma flotante en los programas para esperar una ganancia máxima de 4 en lugar de la obtenida en el apartado 2?

   Buscamos la f para obtener 4:

   $$\frac{\cancel{T_b}}{f\cancel{T_b} + \underbrace{(1-f)\cancel{T_b}}_{p}} = 4 \Rightarrow f = \frac{1}{4}$$

4. ¿Cuánto debería reducirse el tiempo de las operaciones en coma flotante con respecto a la situación inicial para que la ganancia máxima sea 2 suponiendo que en la versión inicial el porcetaje de tiempo de cálculo con como flotante es el obtenido en el apartado 3?

   Se trata de buscar p, suponiendo que tenemos un 75 % de operaciones de coma flotante. Simplemente hay que despejar p.

Ejercicio 2.1.9. Suponga que, en el código siguiente, `a[]` es un array de números de 32 bits en coma flotante y b un número de 32 bits en coma flotante y que debería ejecutarse en menos de 0,5 segundos para $\texttt{N} = 10^9$:

```
for (i=0; i<N; i++)
    a[i+2]=(a[i+2]+a[i+1]+a[i])*b;
```

1. ¿Cuántos GFLOPS se necesitan para poder ejecutar el código en menos de 0,5 segundos?

$$T(10) < 0{,}5seg$$

$$GFLOPS = \frac{NumFFP}{T_{CPU} \cdot 10^9} = \frac{3 \cdot N}{T_{CPU} \cdot 10^9} > \frac{3 \cdot N}{0{,}5 \cdot 10^9}$$

2. Suponiendo que este código en ensamblador tiene 7N instrucciones y que se ha ejecutado en un procesador de 32 bits a 2 GHz. ¿Cual es el número medio de instrucciones que el procesador tiene que poder completar por ciclo para poder ejecutar el código en menos de 0,5 segundos?

   Sabemos que el tiempo en CPU lo podemos calcular de la forma:

$$T_{CPU} = NI \cdot CPI \cdot T_c = \frac{NI}{IPC \cdot F}$$

   Con:

$$IPC = \frac{1}{CPS} =$$

   Sabemos que $NI = 7 \cdot 10^9$, buscamos el valor de IPC y $F = 2 \cdot 10^9$ c/s:

$$\frac{7 \cdot 10^9}{IPC \cdot 2 \cdot 10^9} < 0{,}5$$

$$IPC > 7$$

3. Estimando que el programa pasa el 75 % de su tiempo de ejecución realizando operaciones en coma flotante, ¿cuánto disminuiría como mucho el tiempo de ejecución si se redujesen un 75 % los tiempos de las unidades de coma flotante?

Sea T_p el tiempo de mejora y T_b el tiempo base:

$$Porcreduccion = \frac{T_b - T_p}{T_b} \cdot 100$$

$$T_p = 0{,}25T_p + 0{,}75T_b \cdot 0{,}25 = 0{,}4375T_b$$

$$\left(1 - \frac{T_p}{T_b}\right) \cdot 100 = \left(1 - \frac{0{,}4375\cancel{T_b}}{\cancel{T_b}}\right) \cdot 100 = 56{,}25\,\%$$

$p = 4$ $f = 0{,}25$

Ejercicio 2.1.10. Un compilador ha generado un código máquina optimizado para el siguiente programa

```
par=0; impar=0;
for (i=0;i<N;i++)
    if ((i%2) == 0)
        par=par+c*x[i];
    else
        impar=impar-c*x[i];
```

sin utilizar instrucciones de salto dentro de las iteraciones del bucle (porque se ha usado la técnica de desenrollado el bucle que veremos en el Seminario 4): el código tiene un número de iteraciones de $N/2$, 7 instrucciones fuera del bucle (2 de almacenamiento en memoria, 5 instrucciones para inicializar registros), 9 instrucciones dentro del bucle (4 instrucciones para implementar el bucle for: incremento de la variable de control i, comparación, salto condicional y un salto incondicional; 4 instrucciones coma flotante y 2 instrucciones de carga desde memoria a registro -se leen dos componentes de x).El computador donde se ejecuta dispone de:

- Un procesador superescalar de 32 bits a 2 GHz capaz de terminar dos instrucciones de coma flotante por ciclo y dos instrucciones de cualquier otro tipo por ciclo, excepto instrucciones de carga, cuyo tiempo depende de si hay o no fallo de cache (si no hay fallo de cache suponen 1 ciclo), y las instrucciones de almacenamiento que suponen 1 ciclo.
- Dos caches integradas en el chip de procesamiento (una para datos y otra para instrucciones) de 512 KBytes cada una, mapeo directo, política de actualización de postescritura, líneas de 32 bytes, y latencia de un ciclo de reloj.
- Una memoria principal con latencia de 30 ns. y ciclos burst 6-1-1-1 a través de un bus de memoria de 200 MHz con 64 bits.

Conteste a las siguientes cuestiones:

1. ¿Cuál es la velocidad pico del procesador (en GFLOPS)?
2. ¿Cuál es el tiempo mínimo que tarda en ejecutarse el programa para `N` = 211?

3. ¿Cuántos MFLOPS alcanza el programa?

Observación. Considere que el vector `x` se almacena en memoria en una dirección múltiplo del tamaño de una línea de cache y que ningún componente está en cache cuando se referencia; `N`, `i` estarán en registros de enteros, `par`, `impar`, `c`, y `x[]` son números de 32 bits en coma flotante; dentro del bucle `c`, `par` e `impar` estarán en registros.

Cuestión 2.1.1. Indique cuál es la diferencia fundamental entre una arquitectura CC-NUMA y una arquitectura SMP.

Cuestión 2.1.2. ¿Cuándo diría que un computador es un multiprocesador y cuándo que es un multicomputador?

Cuestión 2.1.3. ¿Un CC-NUMA escala más que un SMP? ¿Por qué?

Cuestión 2.1.4. Indique qué niveles de paralelismo implícito en una aplicación puede aprovechar un PC con un procesador de 4 cores, teniendo en cuenta que cada core tiene unidades funcionales SIMD (también llamadas unidades multimedia) y una microarquitectura segmentada y superscalar. Razone su respuesta.

Cuestión 2.1.5. Si le dicen que un ordenador es de 20 GIPS ¿puede estar seguro que ejecutará cualquier programa de 20000 instrucciones en un microsegundo?

Cuestión 2.1.6. ¿Aceptaría financiar/embarcarse en un proyecto en el que se plantease el diseño e implementación de un computador de propósito general con arquitectura MISD? (Justifique su respuesta).

Cuestión 2.1.7. Deduzca la expresión que se usa para representar la ley de Amdahl suponiendo que se mejora un recurso del procesador, que hay una probabilidad f de no utilizar dicho recurso y que la mejora supone un incremento en un factor de p dela velocidad de procesamiento del recurso.

Cuestión 2.1.8. ¿Es cierto que si se mejora una parte de un sistema (por ejemplo, un recurso de un procesador) se observa experimentalmente que, al aumentar el factor de mejora, llega un momento en que se satura el incremento de velocidad que se consigue? (Justifique la respuesta)

Cuestión 2.1.9. ¿Es cierto que la cota para el incremento de velocidad que establece la ley de Amdahl crece a medida que aumenta el valor del factor de mejora aplicado al recurso o parte del sistema que se mejora? (Justifique la respuesta).

Cuestión 2.1.10. ¿Qué podría ser mejor suponiendo velocidades pico, un procesador superescalar capaz de emitir cuatro instrucciones por ciclo, o un procesador vectorial cuyo repertorio permite codificar 8 operaciones por instrucción y emite una instrucción por ciclo? (Justifique su respuesta).

Cuestión 2.1.11. En la Lección 2 de AC se han presentado diferentes criterios de clasificación de computadores y en el Seminario 0 de prácticas se ha presentado atcgrid. Clasifique atcgrid, sus nodos, sus encapsulados y sus núcleos dentro de la clasificación de Flynn y dentro de la clasificación que usa como criterio el sistema de memoria. Razone su respuesta.

Cuestión 2.1.12. En la Lección 1 de AC se han presentado diferentes criterios de clasificación del paralelismo implícito en una aplicación y en el Seminario 0 de prácticas se ha presentado atcgrid. ¿Qué tipos de paralelismo aprovecha atcgrid? Razone su respuesta.

2.2. Programación paralela

Ejercicio 2.2.1. Un programa tarda 40 s en ejecutarse en un multiprocesador. Durante un 20 % de ese tiempo se ha ejecutado en cuatro procesadores; durante un 60 %, en tres; y durante el 20 % restante, en un procesador (consideramos que se ha distribuido la carga de trabajo por igual entre los procesadores que colaboran en la ejecución en cada momento, despreciamos sobrecarga).

1. ¿Cuánto tiempo tardaría en ejecutarse el programa en un único procesador?

2. ¿Cuál es la ganancia en velocidad obtenida con respecto al tiempo de ejecución secuencial?

3. ¿Cuál es la ganancia en eficiencia obtenida con respecto al tiempo de ejecución secuencial?

Ejercicio 2.2.2. Un programa tarda 20 s en ejecutarse en un procesador P_1, y requiere 30 s en otro procesador P_2. Si se dispone de los dos procesadores para la ejecución del programa (despreciamos sobrecarga):

1. ¿Qué tiempo tarda en ejecutarse el programa si la carga de trabajo se distribuye por igual entre los procesadores P_1 y P_2?

2. ¿Qué distribución de carga entre los dos procesadores P_1 y P_2 permite el menor tiempo de ejecución utilizando los dos procesadores en paralelo ? ¿Cuál es este tiempo?

Ejercicio 2.2.3. ¿Cuál es fracción de código paralelo de un programa secuencial que, ejecutado en paralelo en 8 procesadores, tarda un tiempo de 100 ns, durante 50ns utiliza un único procesador y durante otros 50 ns utiliza 8 procesadores (distribuyendo la carga de trabajo por igual entre los procesadores)?

Ejercicio 2.2.4. Un 25 % de un programa no se puede paralelizar, el resto se puede distribuir por igual entre cualquier número de procesadores. ¿Cuál es el máximo valor de ganancia de velocidad que se podría conseguir al paralelizarlo en p procesadores, y con infinitos? ¿A partir de cuál número de procesadores se podrían conseguir ganancias mayores o iguales que 2?

Ejercicio 2.2.5. En la Figura 2.1, se presenta el grafo de dependencia entre tareas para una aplicación. La figura muestra la fracción del tiempo de ejecución secuencial que la aplicación tarda en ejecutar grupos de tareas del grafo. Suponiendo un tiempo de ejecución secuencial de 60 s, que las tareas no se pueden dividir en tareas de menor granularidad y que el tiempo de comunicación es desprecible, obtener el tiempo de ejecución en paralelo y la ganancia en velocidad en un computador con:

1. 4 procesadores.

2. 2 procesadores.

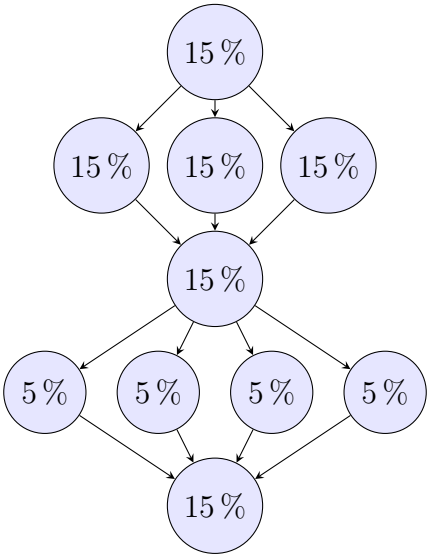


Figura 2.1: Grafo de tareas del Ejercicio 2.2.5

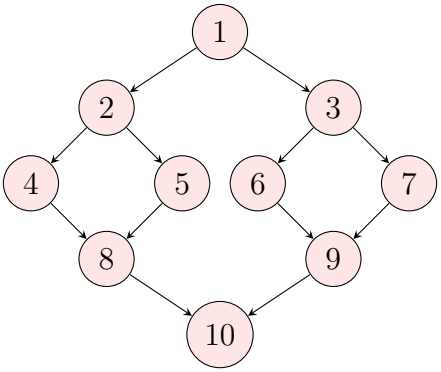


Figura 2.2: Grafo de tareas del Ejercicio 2.2.6

Ejercicio 2.2.6. Un programa se ha conseguido dividir en 10 tareas. El orden de precedencia entre las tareas se muestra con el grafo dirigido de la Figura 2.2. La ejecución de estas tareas en un procesador supone un tiempo de 2 sg. El 10 % de ese tiempo es debido a la ejecución de la tarea 1; el 15 % a la ejecución de la tarea 2; otro 15 % a la ejecución de 3; cada tarea 4, 5, 6 o 7 supone el 9 %; un 8 % supone la tarea 8; la tarea 9 un 10 %; por último, la tarea 10 supone un 6 %. Se dispone de una arquitectura con 8 procesadores para ejecutar la aplicación. Consideramos que el tiempo de comunicación se puede despreciar.

1. ¿Qué tiempo tarda en ejecutarse el programa en paralelo?
2. ¿Qué ganancia en velocidad se obtiene con respecto a su ejecución secuencial?

Ejercicio 2.2.7. Se quiere paralelizar el siguiente trozo de código:

```
//  {Cálculos antes del bucle}
for( i=0; i<w; i++) {
    // Código para i
}
// {cálculos después del bucle}
```

Los cálculos antes y después del bucle suponen un tiempo de t_1 y t_2, respectivamente. Una iteración del ciclo supone un tiempo t_i. En la ejecución paralela, la inicialización de p procesos supone un tiempo $k_1 p$ (k_1 constante), los procesos se comunican y se sincronizan, lo que supone un tiempo $k_2 p$ (k_2 constante); $k_1 p + k_2 p$ constituyen la sobrecarga.

1. Obtener una expresión para el tiempo de ejecución paralela del trozo de código en p procesadores (T_p).

2. Obtener una expresión para la ganancia en velocidad de la ejecución paralela con respecto a una ejecución secuencial (S_p).

3. ¿Tiene el tiempo T_p con respecto a p una característica lineal o puede presentar algún mínimo? ¿Por qué? En caso de presentar un mínimo, ¿para qué número de procesadores p se alcanza?

Ejercicio 2.2.8. Supongamos que se va a ejecutar en paralelo la suma de n números en una arquitectura con p procesadores o cores (p y n potencias de dos) utilizando un grafo de dependencias en forma de árbol (divide y vencerás) para las tareas.

1. Dibujar el grafo de dependencias entre tareas para $n = 16$ y $p = 8$. Hacer una asignación de tareas a procesos.

2. Obtener el tiempo de cálculo paralelo para cualquier n y p con $n > p$ suponiendo que se tarda una unidad de tiempo en realizar una suma.

3. Obtener el tiempo comunicación del algoritmo suponiendo:

   a) Que las comunicaciones en un nivel del árbol se pueden realizar en paralelo en un número de unidades de tiempo igual al número de datos que recibe o envía un proceso en cada nivel del grafo de tareas (tenga en cuenta la asignación de tareas a procesos que ha considerado en el apartado 1)

   b) Que los procesadores que realizan las tareas de las hojas del árbol tienen acceso sin coste de comunicación a los datos que utilizan dichas tareas.

4. Suponiendo que el tiempo de sobrecarga coincide con el tiempo de comunicación calculado en el apartado 3, obtener la ganancia en prestaciones.

5. Obtener el número de procesadores para el que se obtiene la máxima ganancia con n números.

Ejercicio 2.2.9. Se va a paralelizar un decodificador JPEG en un multiprocesador. Se ha extraído para la aplicación el siguiente grafo de tareas que presenta una estructura segmentada (o de flujo de datos):

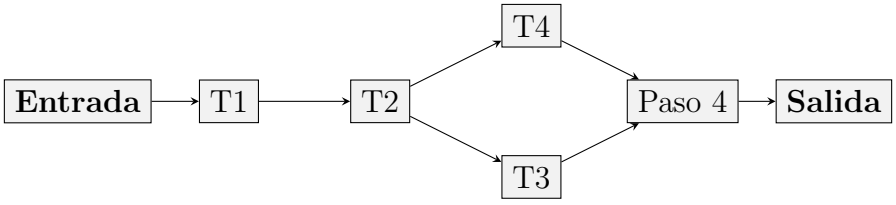


Figura 2.3: Segmentación del Ejercicio 2.2.9

La entrada tenemos que es el bloque de la imagen a decodificar (supone 8x8 pixels de la imagen). La salida será el bloque decodificado de 8x8 pixel. Las tareas 1, 2 y 5 se ejecutan en un tiempo igual a t, mientras que las tareas 3 y 4 suponen 1,5t. El decodificador JPEG aplica el grafo de tareas de la figura a bloques de la imagen, cada uno de 8x8 píxeles. Si se procesa una imagen que se puede dividir en n bloques de 8x8 píxeles, a cada uno de esos n bloques se aplica el grafo de tareas de la figura. Obtenga la mayor ganancia en prestaciones que se puede conseguir paralelizando el decodificador JPEG en (suponga despreciable el tiempo de comunicación/sincronización):

1. 5 procesadores.
2. 4 procesadores.

En cualquier de los dos casos, la ganancia se tiene que calcular suponiendo que se procesa una imagen con un total de n bloques de 8x8 píxeles.

Ejercicio 2.2.10. Se quiere implementar un programa paralelo para un multicomputador que calcule la siguiente expresión para cualquier x (es el polinomio de interpolación de Lagrange): $P(x) = \sum_{i=0}^{n} (b_i \cdot L_i(x))$, donde:

$$L_i(x) = \frac{(x-a_0)\ldots(x-a_{i-1})(x-a_{i+1})\ldots(x-a_n)}{k_i} = \frac{\prod_{\substack{j=0 \\ j\neq i}}^{n}(x-a_j)}{k_i} \qquad i = 0, 1, \ldots, n$$

$$k_i = (a_i-a_0)\ldots(a_i-a_{i-1})(a_i-a_{i+1})\ldots(a_i-a_n) = \prod_{\substack{j=0 \\ j\neq i}}^{n}(a_i-a_j) \qquad i = 0, 1, \ldots, n$$

Inicialmente k_i, a_i y b_i se encuentran en el nodo i y x en todos los nodos. Sólo se van a usar funciones de comunicación colectivas. Indique cuál es el número mínimo de funciones colectivas que se pueden usar, cuáles serían, en qué orden se utilizarían y para qué se usan en cada caso.

Ejercicio 2.2.11.

1. Escriba un programa secuencial con notación algorítmica (podría escribirlo en C) que determine si un número de entrada, x, es primo o no. El programa imprimirá si es o no primo. Tendrá almacenados en un vector, `NP`, los M números primos entre 1 y el máximo valor que puede tener un número de entrada al programa.

2. Escriba una versión paralela del programa anterior para un multicomputador usando un estilo de programación paralela de paso de mensajes. El proceso 0 tiene inicialmente el número x y el vector `NP` en su memoria e imprimirá en pantalla el resultado. Considere que la herramienta de programanción ofrece funciones `send()/receive()` para implementar una comunicación uno-a-uno asíncrona, es decir, con función `send(buffer,count,datatype,idproc,group)` no bloqueante y `receive(buffer,count,datatype,idproc,group)` bloqueante. En las funciones `send()/receive()` se especifica:

- `group`: identificador del grupo de procesos que intervienen en la comunicación.
- `idproc`: identificador del proceso al que se envía o del que se recibe.
- `buffer`: dirección a partir de la cual se almacenan los datos que se envían o los datos que se reciben.
- `datatype`: tipo de los datos a enviar o recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- `count`: número de datos a transferir de tipo `datatype`.

Ejercicio 2.2.12. Escribir una versión paralela del programa secuencial del ejercicio 2.2.11 para un multicomputador usando un estilo de programación paralela de paso de mensajes y suponiendo que la herramienta de programación ofrece las funciones colectivas de difusión y reducción (escribir primero la versión secuencial). Sólo el proceso 0 imprimirá en pantalla. En la función de difusión, `broadcast(buffer,count,datatype,idproc` se especifica:

- `group`: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo reciben.
- `idproc`: identificador del proceso que envía.
- `buffer`: dirección de comienzo en memoria de los datos que difunde `idproc` y que almacenará, en todos los procesos del grupo, los datos difundidos.
- `datatype`: tipo de los datos a enviar/recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- `count`: número de datos a transferir de tipo `datatype`.

En la función de reducción, `reduction(sendbuf,recvbuf,count,datatype,oper,idproc,group)`, se especifica:

- `group`: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo envían.
- `idproc`: identificador del proceso que recibe.
- `recvbuf`: dirección en memoria a partir de la cual se almacena el escalar resultado de la reducción de todos los componentes de todos los vectores `sendbuf`.
- `sendbuf`: dirección en memoria a partir de la cual almacenan todos los procesos del grupo los datos de tipo `datatype` a reducir (uno o varios).
- `datatype`: tipo de los datos a enviar y recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- `oper`: tipo de operación de reducción. Puede tomar los valores `OR`, `AND`, `ADD`, `MUL`,`MIN`,`MAX`
- `count`: número de datos de tipo `datatype`, del buffer `sendbuffer` de cada proceso, que se van a reducir.

Ejercicio 2.2.13.

1. Escribir una versión paralela del programa paralelo del ejercicio 2.2.12 suponiendo que, además de las dos funciones colectivas anteriores, se dispone de dispersión y que M es divisible entre el número de procesos (escribir primero la versión secuencial). Sólo el proceso 0 imprimirá en pantalla. La función `scatter(sendbuf,sendcnt,recvbuf,recvcnt,datatype,idproc,group)` especifica:
   - `group`: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo envían.
   - `idproc`: identificador del proceso que envía.
   - `recvbuf`: dirección en memoria a partir de la cual se almacenan los datos recibidos.
   - `sendbuf`: dirección en memoria a partir de la cual almacena el proceso `idproc` los datos a enviar.
   - `sendtype`: tipo de los datos a enviar y recibir.
   - `recvcnt`: número de datos de tipo `datatype` a recibir en `recvbuf`.
   - `sendcnt`: número de datos de tipo `datatype` a enviar.
2. ¿Qué estructura de procesos/tareas implementa el código paralelo del apartado 1? Justifique su respuesta.

Ejercicio 2.2.14. Escribir una versión paralela del programa secuencial del ejercicio 2.2.11 para un multiprocesador usando el estilo de programación paralela de variables compartidas; en particular, use OpenMP (escribir primero la versión secuencial).

Cuestión 2.2.1. Indique las diferencias entre OpenMP y MPI.

Cuestión 2.2.2. Ventajas e inconvenientes de una asignación estática de tareas a procesos/threads frente a una asignación dinámica.

Cuestión 2.2.3. ¿Qué se entiende por escalabilidad lineal y por escalabilidad superlineal? Indique las causas por las que se puede obtener una escalabilidad superlineal.

Cuestión 2.2.4. Enuncie la ley de Amdahl en el contexto de procesamiento paralelo.

Cuestión 2.2.5. Deduzca la expresión matemática que se suele utilizar para caracterizar la ley de Gustafson. Defina claramente y sin ambigüedad el punto de partida que va a utilizar para deducir esta expresión y cada una de las etiquetas que utilice. ¿Qué nos quiere decir Gustafson con esta ley?

Cuestión 2.2.6. Deduzca la expresión que caracteriza a la ley de Amdahl. Defina claramente el punto de partida y todas las etiquetas que utilice.