Ejercicios de programación paralela con OpenMP y MPI

José E. Román | José Miguel Alonso | Fernando Alvarruiz |
Ignacio Blanquer |David Guerrero | J. Javier Ibáñez |
Enrique Ramos

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

José E. Román (coord.)
José Miguel Alonso
Fernando Alvarruiz
Ignacio Blanquer
David Guerrero
J. Javier Ibáñez
Enrique Ramos

Ejercicios de programación paralela con OpenMP y MPI

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Colección *Académica*

Para referenciar esta publicación utilice la siguiente cita: Román Moltó, José E.; Alonso Ábalos, José Miguel; Alvarruiz Bermejo, Fernando; Blanquer Espert, Ignacio; Guerrero López, David; Ibáñez González, Jacinto Javier; Ramos Peinado, Enrique. *Ejercicios de programación paralela con OpenMP y MPI* Valencia: Editorial Universitat Politècnica de València

distribución: www.lalibreria.upv.es / Ref.: 0721_05_01_01

Imprime: Byprint Percom, sl

ISBN: 978-84-9048-714-3
Impreso bajo demanda

Impreso en España

Resumen

Este libro puede resultar de interés para cualquier persona con conocimientos de programación que quiera adentrarse en el mundo de la programación paralela, es decir, la programación de computadores paralelos, bien sean ordenadores con varios núcleos o computadores formados por varios nodos conectados mediante una red de interconexión (*clusters*). La programación para estos dos tipos de arquitecturas se trata en los capítulos 2 y 3, respectivamente. El libro aborda dicha programación de forma eminentemente práctica, mediante ejercicios que proponen un código secuencial sencillo que debe ser paralelizado. La parte más teórica y descriptiva del libro se restringe al capítulo 1, que introduce brevemente los conceptos más importantes, haciendo referencia a otros libros donde el lector puede encontrar una descripción más detallada.

Para poder seguir el libro es necesario tener conocimientos de programación, en particular en el lenguaje de programación C. Todos los códigos presentados están en lenguaje C, si bien los conceptos son directamente trasladables a otros lenguajes de programación como Fortran. Otros conocimientos previos recomendables son los fundamentos básicos de arquitectura de computadores paralelos y de programación concurrente. Con este libro, el lector adquirirá competencias de programación paralela que son difíciles de obtener con otras obras que abordan la temática desde un punto de vista más teórico.

Valencia, mayo de 2018
Los autores

Índice general

Capítulo 1

Introducción

En este capítulo introductorio describimos brevemente los conceptos más destacados de la computación paralela y de la programación paralela para memoria compartida (mediante OpenMP) y para memoria distribuida (mediante MPI). En lugar de describir en detalle estos conceptos, al tratarse de un libro de ejercicios, simplemente se proporcionan las ideas básicas, citando otras fuentes para que el lector pueda ampliar la información.

1.1. Computación paralela

La computación paralela intenta aprovechar la concurrencia disponible en las arquitecturas de computadores actuales, las cuales permiten ejecutar varias operaciones simultáneamente, procesar varios hilos de ejecución concurrentemente, o incluso ejecutar al mismo tiempo diferentes aplicaciones en varios procesadores integrados en un único computador. Tradicionalmente, un computador paralelo era una máquina extremadamente cara, reservada para los centros de computación más avanzados. Sin embargo, actualmente las arquitecturas paralelas han permeado todos los ámbitos de la computación y están presentes en cualquier computador (desde un portátil hasta los servidores más potentes) o incluso en dispositivos como los teléfonos móviles.

Para llevar a cabo la computación paralela se utiliza la programación paralela, mediante la cual se desarrollan algoritmos paralelos, normalmente a partir de algoritmos secuenciales, y se lleva a cabo la implementación de los mismos me-

diante algún lenguaje o herramienta de programación paralela. Existen muchas de estas herramientas, algunas de ellas apropiadas para un tipo de arquitectura paralela específica. En este libro nos centramos en dos de las más extendidas: OpenMP, para memoria compartida, y MPI, para memoria distribuida.

Una referencia muy recomendable como introducción a la computación paralela es el libro de Grama y col. (2003) o el más reciente de Pacheco (2011), o bien si se prefiere en castellano el libro de Almeida y col. (2008). Otras referencias algo más antiguas son (Wilkinson y Allen 2005) y (Petersen y Arbenz 2004).

El diseño de algoritmos paralelos (Grama y col. 2003, Cap. 3) se basa en identificar tareas que sean susceptibles de ser ejecutadas concurrentemente, y posteriormente asignar dichas tareas a los elementos de proceso (hilos de ejecución o procesos del sistema operativo). Para garantizar que el resultado del algoritmo paralelo es el mismo que el del secuencial, es necesario respetar las dependencias de datos, las cuales obligan a una tarea a esperar la finalización de las tareas precedentes. Las dependencias de datos, que vienen determinadas por las condiciones de Bernstein, establecen pues un orden parcial en el que se han de ejecutar las tareas. Dicho orden se puede representar formalmente mediante el denominado grafo de dependencias.

El grafo de dependencias es una herramienta muy potente para analizar un algoritmo paralelo. La longitud del camino crítico del citado grafo (el camino con mayor coste entre todos los posibles) establece una cota inferior del tiempo de ejecución del algoritmo paralelo. Esto implica que si el camino crítico contiene tareas muy costosas en relación con las demás, la paralelización será ineficiente y, por tanto, sería conveniente considerar una descomposición de tareas alternativa. El grado medio de concurrencia es un buen indicador para saber si el grafo de dependencias permite una buena paralelización o no.

A menudo la computación paralela se aplica a algoritmos de cálculo intensivo, en cuyo caso el objetivo principal suele ser acelerar el cálculo para obtener la respuesta en un tiempo inferior al que se tendría con un algoritmo secuencial. En otras ocasiones, el objetivo es poder abordar problemas de mayor dimensión, los cuales exceden la capacidad de memoria de un único computador, pero sí son factibles con la memoria agregada al utilizar varios computadores a la vez, como ocurre en un *cluster* de computadores. El análisis de costes (especialmente el coste computacional, aunque también el coste de memoria) es muy importante para poder determinar si un algoritmo paralelo será más o menos rápido y eficiente en comparación con su equivalente secuencial, ver (Grama y col. 2003, Cap. 5).

En el análisis del coste computacional *a priori*, el coste del algoritmo secuencial, t_1, se mide en *flops* (número de operaciones en coma flotante), contabilizando el total de operaciones realizadas en el algoritmo. El equivalente en el caso paralelo es el coste (o tiempo) paralelo con p procesos (o hilos de ejecución), t_p. Este coste comprende el intervalo de tiempo desde el inicio de la ejecución de la primera tarea del grafo de dependencias hasta que finaliza la última. El coste aritmético dependerá del número de procesos utilizados, p, siendo como mínimo igual al coste asociado al camino crítico. Al coste aritmético hay que añadir el coste de comunicación en el caso de paralelización por paso de mensajes. La bondad relativa del algoritmo paralelo se puede determinar comparando los tiempos secuencial y paralelo, mediante el *speedup* (o aceleración) y la eficiencia,

$$S_p = \frac{t_1}{t_p}, \qquad E_p = \frac{S_p}{p}.$$

Una eficiencia cercana a 1 indica que se están aprovechando los p procesadores eficientemente. El *speedup* y la eficiencia también se pueden analizar *a posteriori* sobre una implementación particular del algoritmo paralelo, en cuyo caso se mide el tiempo en segundos en lugar de expresarlo en flops.

1.2. Programación en memoria compartida: OpenMP

El modelo de programación paralela en memoria compartida se basa en disponer de varios hilos de ejecución concurrentes que pueden acceder a variables alojadas en zonas de la memoria a las que tienen acceso todos ellos. Esto facilita mucho la programación, ya que se evita el tener que intercambiar explícitamente datos entre los diferentes procesadores.

OpenMP es un sistema de programación de memoria compartida basado en directivas de compilación, es decir, el programador inserta directivas en el código secuencial para indicar al compilador cómo ha de realizar la paralelización, además de incorporar algunas llamadas a funciones específicas. Una descripción del paradigma de memoria compartida y de OpenMP se puede encontrar en (Grama y col. 2003, Cap. 7), y también en las monografías de Chandra y col. (2001) y Chapman y col. (2008). Para aspectos más avanzados de OpenMP no cubiertos en este libro se puede consultar (van der Pas y col. 2017).

El modelo de ejecución de OpenMP establece que determinadas directivas (por ejemplo la directiva `parallel`) crean una región paralela, en la cual hay varios hilos de ejecución activos. Durante la ejecución de un programa se van alternando las regiones paralelas con regiones secuenciales, en las que únicamente

el hilo principal está activo. Al finalizar una región paralela se impone una barrera implícita, es decir, que el hilo principal no podrá continuar hasta que no hayan alcanzado ese punto todos los hilos participantes en la región paralela que termina.

Si bien el uso de directivas de compilación es muy cómodo, el programador debe ser consciente de las implicaciones que tiene sobre el bloque de código al que afecta la directiva. En particular, para cada una de las variables implicadas es necesario determinar su alcance, es decir, determinar si dichas variables deben ser compartidas entre los diferentes hilos o si por el contrario cada hilo debe mantener una copia privada. Un caso particular de este último tipo es el de las variables con reducción, las cuales se comportan como variables privadas a las que se aplica una operación aritmética (o lógica) al finalizar el bloque asociado a la directiva.

Dos esquemas de paralelización ampliamente utilizados son el paralelismo de datos y el paralelismo de tareas. OpenMP da soporte para ambos esquemas de manera muy sencilla.

En el paralelismo de datos, las tareas del algoritmo están muy ligadas a los datos, siendo habitualmente un cálculo muy regular implementado mediante el uso de bucles. En estos casos la paralelización orientada a los bucles suele ser muy efectiva, y se utiliza mucho en problemas como el cálculo matricial. OpenMP ofrece la directiva `parallel for` para repartir la ejecución de las diferentes iteraciones del bucle entre los hilos disponibles. El programador puede especificar la forma de realizar este reparto mediante la cláusula `schedule`, siendo lo más habitual un reparto estático por bloques.

En el paralelismo de tareas es el programador el que explícitamente define cada una de las tareas del algoritmo paralelo, a diferencia del caso citado del paralelismo de bucles en el que las tareas implícitas son cada una de las iteraciones del bucle. En OpenMP el paralelismo de tareas se puede implementar de dos formas: mediante la directiva `task` (no tratada en este libro) o mediante la construcción `parallel sections`. En esta última, el grafo de dependencias de tareas se implementa en el código de la siguiente forma: las tareas que se puedan ejecutar concurrentemente se incluyen dentro de una construcción `parallel sections` (delimitando cada tarea mediante bloques `section`) y las dependencias entre las tareas se fuerzan mediante la barrera implícita existente al finalizar dicha construcción.

En los algoritmos en los que no sea factible ninguno de los dos esquemas anteriores, se puede realizar la paralelización mediante una directiva `parallel` y un reparto manual del trabajo a partir del identificador de hilo.

En cualquiera de los casos anteriores, es importante el uso correcto de las variables compartidas, ya que una condición de carrera podría producir un resultado erróneo del algoritmo paralelo. Para evitar las condiciones de carrera es necesario proteger el acceso concurrente (de lectura y escritura simultáneas por parte de diferentes hilos) a las variables compartidas, de manera que dicho acceso se realice en exclusión mutua. OpenMP dispone de la directiva `critical` para delimitar una sección crítica, y también de la directiva `atomic` para implementar una operación atómica de modificación y escritura de una variable compartida, que solo produce la secuencialización de las operaciones si el área de memoria es la misma.

La programación en OpenMP se aborda a través de ejemplos en el Capítulo 2.

1.3. Programación en memoria distribuida: MPI

El modelo de programación de paso de mensajes es el que se utiliza en computadores de memoria distribuida (por ejemplo, los *clusters*). En este caso, las tareas del algoritmo paralelo son asignadas a procesos del sistema operativo, en lugar de a hilos, los cuales se ejecutan en procesadores (o nodos del *cluster*) con espacios de memoria disjuntos. Se utiliza el envío explícito de mensajes (habitualmente a través de una red de interconexión) tanto para intercambiar datos entre tareas que se ejecutan en diferentes procesos, como para sincronizarse según las dependencias presentes en el grafo.

MPI es un estándar ampliamente utilizado para la programación paralela mediante paso de mensajes. Se basa en extender un lenguaje de programación convencional (como C o Fortran) con funciones de biblioteca, las cuales pueden ser utilizadas únicamente después de haber realizado la inicialización (`MPI_Init`). En (Grama y col. 2003, Cap. 6) se presenta una descripción básica de MPI. Para profundizar en su uso se puede acudir a las monografías de Pacheco (1997) y de Gropp, Lusk y col. (1999). En este libro no se cubren aspectos avanzados como la entrada-salida paralela o las operaciones de acceso a memoria remota, para los cuales se puede consultar (Gropp, Hoefler y col. 2014).

El modelo de ejecución de MPI consiste en lanzar simultáneamente p copias del mismo programa ejecutable en p procesos distintos (mediante el co-

mando `mpiexec`)[1]. Estos p procesos forman el comunicador inicial llamado `MPI_COMM_WORLD`, a partir del cual se pueden crear nuevos comunicadores si fuera necesario. Un comunicador es un grupo de procesos (cada uno de ellos identificado por un número de orden) al que se asocia un contexto de comunicación (el ámbito en el que se envían los mensajes).

La programación más básica en MPI consiste en realizar operaciones de comunicación punto a punto, en las que un proceso envía un mensaje y otro proceso lo recibe. Las operaciones de envío y recepción deben especificar un *buffer* (o zona de memoria) mediante un puntero, el número de elementos y su tipo. Además deben indicar el comunicador en el que se realiza la comunicación, el identificador del proceso destinatario o fuente del envío y una etiqueta que distingue unos mensajes de otros.

Las operaciones de comunicación tienen un coste no despreciable, especialmente si la red de interconexión no es muy eficiente. Por tanto, en el análisis de costes *a priori* es necesario tener en cuenta dicho coste. Esto se hace habitualmente con un modelo sencillo para el envío de un mensaje en el que se suma el tiempo de latencia más el tiempo de transmisión de un elemento por la longitud del mensaje.

A veces los programas MPI resultan ineficientes, no solo por el tiempo que tarda un mensaje en transmitirse del proceso origen al destino, sino por los tiempos de espera en los casos en los que uno de los dos procesos no está preparado aún para realizar la comunicación. En estas circunstancias es conveniente el uso de primitivas de comunicación no bloqueantes. También se puede optimizar la comunicación con el uso de la operación combinada de envío y recepción.

En el contexto de MPI también es pertinente hablar de paralelismo de tareas y paralelismo de datos. En el paralelismo de tareas, cada proceso ejecuta una o más tareas, y las dependencias de datos se garantizan mediante el envío de mensajes entre unas tareas y otras (si dos tareas se han asignado al mismo proceso, no será necesaria la comunicación dado que el espacio de memoria será el mismo). Un esquema de paralelización muy empleado es el de maestro y trabajadores, en el cual existe una bolsa grande de tareas que es gestionada por un proceso denominado maestro, el cual se encarga de repartir las tareas a los procesos trabajadores a medida que van terminando las tareas asignadas anteriormente. Este esquema de asignación dinámica es muy efectivo en determinadas aplicaciones.

[1] También es posible crear nuevos procesos durante la ejecución, pero es menos habitual.

En cuanto al paralelismo de datos, en el contexto de MPI se asocia a la *distribución de datos*, en donde una estructura de datos de gran dimensión (por ejemplo, una matriz) se reparte entre los diferentes procesos implicados, cada uno de los cuales almacena únicamente un fragmento. En el caso de trabajar con matrices, las más utilizadas habitualmente son la distribución por bloques de filas (o de columnas), la distribución cíclica o la cíclica por bloques.

Para la distribución de datos es muy importante el uso de operaciones de comunicación colectiva, así como para cualquier algoritmo paralelo en el que el patrón de comunicación sea más complejo que el simple envío punto a punto. El estándar MPI facilita enormemente la programación proporcionando multitud de operaciones de comunicación colectiva, como la difusión (envío de uno a todos), el reparto y la recogida (en las que un proceso distribuye fragmentos del buffer de datos a los demás o combina los fragmentos recibidos del resto), o la reducción (en la que además de la comunicación se realiza una operación aritmética o lógica sobre los datos enviados). El uso de estas primitivas no solo simplifica la programación, sino que además garantiza que se emplea un algoritmo óptimo de comunicación para dicha operación.

En el contexto de la distribución de datos también es muy importante la posibilidad que ofrece MPI de definir tipos de datos derivados. El objetivo es poder enviar eficientemente datos que no están contiguos en memoria, minimizando las posibles copias intermedias. Un ejemplo de su uso es el envío de una columna de una matriz[2], dado que los elementos de dicha columna están almacenados en posiciones de memoria equiespaciadas con una separación igual al número de columnas de las que consta.

La programación en MPI se aborda a través de ejemplos en el Capítulo 3.

[2]En el lenguaje C los arrays bidimensionales se almacenan por filas.

Capítulo 2

Programación con OpenMP

En este capítulo se recopilan ejercicios de programación paralela para entornos de memoria compartida (multi-hilo) en los que se usa OpenMP. En la sección 2.1 se aborda la paralelización de bucles, utilizando la directiva `parallel for`*. En la sección 2.2 se tratan ejemplos en los que es más apropiado utilizar regiones paralelas (directiva* `parallel`*) de forma general, utilizando en muchos casos la cláusula de reparto de trabajo* `sections`*. Por último, en la sección 2.3 se aborda la problemática de la sincronización necesaria para evitar las condiciones de carrera en el acceso a variables compartidas.*

2.1. Paralelización de bucles

Cuestión 2.1-1. Según las condiciones de Bernstein, indica el tipo de dependencias de datos existente entre las distintas iteraciones en los casos que se presentan a continuación. Justifica si se puede eliminar o no esa dependencia de datos, eliminándola en caso de que sea posible.

(a)
```
for (i=1;i<N-1;i++) {
  x[i+1] = x[i] + x[i-1];
}
```

Solución: Hay una dependencia de datos entre las diferentes iteraciones: incumple la 1ª condición de Bernstein ($I_j \cap O_i \neq \emptyset$), pues, por ejemplo, x[2] es una variable de salida en la iteración i=1 y una variable de entrada en la iteración i=2. No es posible eliminar esa dependencia de datos.

(b)
```
for (i=0;i<N;i++) {
  a[i] = a[i] + y[i];
  x = a[i];
}
```

Solución: Hay una dependencia de datos entre las diferentes iteraciones: incumple la 3ª condición de Bernstein ($O_i \cap O_j \neq \emptyset$), pues `x` es una variable de salida en todas las iteraciones. En este caso sí es posible eliminar la dependencia:

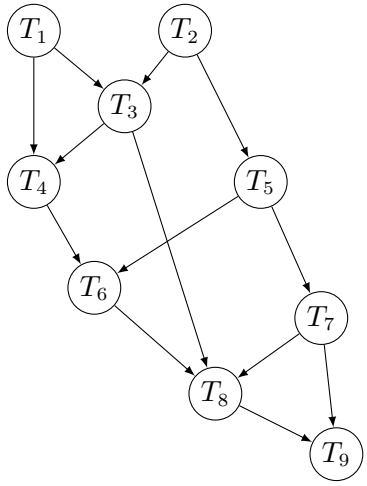
```
for (i=0;i<N;i++) {
  a[i] = a[i] + y[i];
}
x = a[N-1];
```

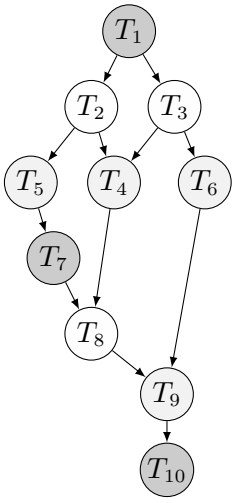
(c)
```
for (i=N-2;i>=0;i--) {
  x[i] = x[i] + y[i+1];
  y[i] = y[i] + z[i];
}
```

Solución: Hay una dependencia de datos entre las diferentes iteraciones: incumple la 1ª condición de Bernstein ($I_j \cap O_i \neq \emptyset$), pues, por ejemplo, y[1] es una variable de salida en la iteración i=1 y de entrada en la iteración i=0. En este caso sí es posible eliminar la dependencia:

```
x[N-2] = x[N-2] + y[N-1];
for (i=N-3;i>=0;i--) {
  y[i+1] = y[i+1] + z[i+1];
  x[i] = x[i] + y[i+1];
}
y[0] = y[0] + z[0];
```

Cuestión 2.1-2. Dados los siguientes grafos de dependencias de tareas:





T_1, T_7, T_{10}	$\frac{1}{3}n^3$
T_2, T_3, T_8	n^3
T_4, T_5, T_6, T_9	$2n^3$

(a) Para el grafo de la izquierda, indica qué secuencia de nodos del grafo constituye el camino crítico. Calcula la longitud del camino crítico y el grado medio de concurrencia. Nota: no se ofrece información de costes, se puede suponer que todas las tareas tienen el mismo coste.

Solución: De entre todos los posibles caminos entre un nodo inicial y un nodo final, el que mayor coste tiene (camino crítico) es $T_1 - T_3 - T_4 - T_6 - T_8 - T_9$ (o de forma equivalente empezando en T_2). Su longitud es $L = 6$. El grado medio de concurrencia es

$$M = \sum_{i=1}^{9} \frac{1}{6} = \frac{9}{6} = 1{,}5$$

(b) Repite el apartado anterior para el grafo de la derecha. Nota: en este caso el coste de cada tarea viene dado en flops (para un tamaño de problema n) según la tabla mostrada.

Solución: En este caso, el camino crítico es $T_1 - T_2 - T_5 - T_7 - T_8 - T_9 - T_{10}$ y su longitud es

$$L = \frac{1}{3}n^3 + n^3 + 2n^3 + \frac{1}{3}n^3 + n^3 + 2n^3 + \frac{1}{3}n^3 = 7n^3 \text{ flops}$$

El grado medio de concurrencia es

$$M = \frac{3 \cdot \frac{1}{3}n^3 + 3 \cdot n^3 + 4 \cdot 2n^3}{7n^3} = \frac{12n^3}{7n^3} = 1{,}71$$

Cuestión 2.1-3. El siguiente código secuencial implementa el producto de una matriz B de dimensión $N \times N$ por un vector c de dimensión N.

```
void prodmv(double a[N], double c[N], double B[N][N])
{
  int i, j;
  double sum;
  for (i=0; i<N; i++) {
    sum = 0;
    for (j=0; j<N; j++)
      sum += B[i][j] * c[j];
    a[i] = sum;
  }
}
```

(a) Realiza una implementación paralela mediante OpenMP del código dado.

(b) Calcula los costes computacionales en flops de las implementaciones secuencial y paralela, suponiendo que el número de hilos p es un divisor de N.

(c) Calcula el speedup y la eficiencia del código paralelo.

Solución:

(a)
```
void prodmvp(double a[N], double c[N], double B[N][N])
{
  int i, j;
  double sum;
  #pragma omp parallel for private(j,sum)
  for (i=0; i<N; i++) {
    sum = 0.0;
    for (j=0; j<N; j++)
      sum += B[i][j] * c[j];
    a[i] = sum;
  }
}
```

(b) Coste secuencial: $t(N) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 2 = 2N^2$ flops

Coste paralelo: $t(N,p) = \sum_{i=0}^{d-1} \sum_{j=0}^{N-1} 2 = 2dN$ flops, donde $d = \frac{N}{p}$

(c) Speedup: $S(N,p) = \frac{t(N)}{t(N,p)} = \frac{2N^2}{2dN} = \frac{N}{d} = p$

Eficiencia: $E(N,p) = \frac{S(N,p)}{p} = \frac{p}{p} = 1$

Cuestión 2.1-4. Dada la siguiente función:

```
double funcion(double A[M][N])
{
  int i,j;
  double suma;
  for (i=0; i<M-1; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = 2.0 * A[i+1][j];
    }
  }
  suma = 0.0;
  for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
      suma = suma + A[i][j];
    }
  }
  return suma;
}
```

(a) Indica su coste teórico (en flops).

Solución: El cálculo tiene dos fases. En la primera se sobrescribe cada fila de la matriz con la fila siguiente multiplicada por 2. En la segunda parte se realiza la suma de todos los elementos de la matriz.

En la primera fase se realiza sólo una operación en el bucle más interior, y por tanto su coste es $\sum_{i=0}^{M-2} \sum_{j=0}^{N-1} 1 = \sum_{i=0}^{M-2} N = (M-1)N \approx MN$. [La aproximación la hacemos en sentido asintótico, es decir, suponiendo que

tanto N como M son suficientemente grandes.] La segunda fase tiene un coste similar, salvo que el bucle `i` hace una iteración más: MN.
Coste secuencial: $t_1 = 2MN$ flops

(b) Paralelízalo usando OpenMP. ¿Por qué lo haces así? Se valorarán más aquellas soluciones que sean más eficientes.

Solución: Planteamos una paralelización con dos regiones paralelas, una por cada fase, ya que la segunda fase no puede empezar hasta que haya terminado la primera. En la primera fase existen dependencias de datos en el índice `i`, que pueden resolverse intercambiando los bucles y paralelizando el bucle `j` (también se podría paralelizar el bucle `j` sin intercambiar los bucles, pero esto sería más ineficiente). La segunda fase requiere una reducción sobre la variable `suma`. En ambas fases tanto `i` como `j` deben ser variables privadas.

```
double funcion(double A[M][N])
{
  int i,j;
  double suma;
  #pragma omp parallel for private(i)
  for (j=0; j<N; j++) {
    for (i=0; i<M-1; i++) {
      A[i][j] = 2.0 * A[i+1][j];
    }
  }
  suma = 0.0;
  #pragma omp parallel for reduction(+:suma) private(j)
  for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
      suma = suma + A[i][j];
    }
  }
  return suma;
}
```

(c) Indica el speedup que podrá obtenerse con p procesadores suponiendo M y N múltiplos exactos de p.

Para seguir leyendo haga click aquí