Programación y computación paralelas

Ivan Rodero Castro
Francesc Guim Bernat

PID_00209162

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En este módulo didáctico, estudiaremos los principales modelos de programación de aplicaciones paralelas utilizados en la computación de altas prestaciones, los esquemas básicos de programación de aplicaciones paralelas y los entornos utilizados para la ejecución de este tipo de aplicaciones en el contexto de la computación de altas prestaciones.

En los primeros apartados, nos centraremos en los modelos de programación tanto para memoria compartida como distribuida. En relación con los modelos de memoria compartida, estudiaremos principalmente OpenMP y también pondremos énfasis en la programación gráfica, la cual tiene cada vez más presencia en la computación de altas prestaciones. A continuación estudiaremos los principales modelos para memoria distribuida, tanto de paso de mensajes (*MPI*) como aquellos que proporcionan abstracciones de espacios de direcciones compartidos utilizando memoria distribuida.

Una vez presentados los modelos de programación más relevantes para la computación de altas prestaciones, estudiaremos algunas de las técnicas más populares que permiten el desarrollo de aplicaciones paralelas, tanto para su diseño inicial como para convertir aplicaciones secuenciales en paralelas. En concreto, estudiaremos tanto técnicas para la obtención de paralelismo (por ejemplo, a partir del paralelismo de datos) como varios patrones de algoritmos paralelos.

Objetivos

Los materiales didácticos de este módulo contienen las herramientas necesarias para lograr los objetivos siguientes:

1. Conocer los modelos de programación para memoria compartida y saber programar aplicaciones paralelas con OpenMP.

2. Aprender los conceptos fundamentales para programar dispositivos GPU con los modelos de programación para computación gráfica CUDA y OpenCL.

3. Aprender los conceptos fundamentales para programar aplicaciones paralelas con MPI y conocer modelos de programación para memoria distribuida basados en espacios de direcciones compartidos, como por ejemplo PGAS.

4. Aprender las técnicas y los patrones básicos para el diseño de algoritmos paralelos y ser capaces de desarrollar aplicaciones basadas en estas técnicas y en los modelos de programación estudiados en este módulo didáctico.

5. Conocer el funcionamiento y los componentes que forman los sistemas de gestión de aplicaciones en sistemas paralelos para computación de altas prestaciones.

6. Aprender los conceptos fundamentales de las políticas de planificación de trabajos en entornos de altas prestaciones.

1. Modelos de programación para memoria compartida

Tal y como vimos en el módulo "Introducción a la computación de altas prestaciones" de esta asignatura, los sistemas multiprocesador o multinúcleo de memoria compartida son aquellos en los que cualquiera de los procesadores o los núcleos puede acceder a cualquier espacio de la memoria. Por lo tanto, encontramos un único espacio de direcciones y, de este modo, cada ubicación de memoria tiene una sola dirección dentro de un rango de direcciones. Las principales alternativas para la programación de sistemas de memoria compartida son las siguientes.

- Utilizar procesos convencionales ofrecidos por el sistema operativo.
- Utilizar flujos, como por ejemplo Pthreads.
- Utilizar un nuevo lenguaje de programación. Un ejemplo es el lenguaje Ada, a pesar de que no es una solución demasiado extendida.
- Utilizar una biblioteca con un lenguaje de programación.
- Modificar la sintaxis de un lenguaje de programación secuencial para crear un lenguaje de programación paralela. Un ejemplo es *UPC*[1].
- Utilizar un lenguaje de programación secuencial y complementarlo con directivas de compilación para especificar paralelismo. Un ejemplo es OpenMP.

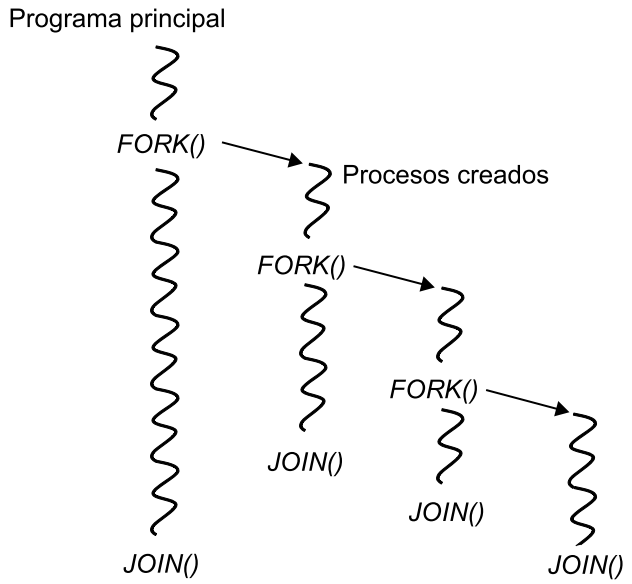
(1) Del inglés *unified parallel C.*

En las siguientes secciones estudiaremos las características básicas de la programación basada en procesadores/flujos y los posibles problemas asociados. A continuación, estudiaremos OpenMP como modelo de programación para memoria compartida adecuado para la computación de altas prestaciones.

1.1. Programación con procesos/flujos

Uno de los posibles mecanismos para implementar paralelismo con memoria compartida es el uso de procesos convencionales utilizando el modelo *fork-join*, tal y como muestra la figura 1.

Figura 1. Modelo *fork-join* utilizado para implementar paralelismo con procesos



Se trata de un modelo *SPMD*[2] en el que, generalmente, el código correspondiente al proceso padre es diferente al que ejecutan los procesos creados, que se acostumbran a denominar esclavos. Además, el proceso padre es el que se encarga de esperar la finalización de los procesos esclavos (*join* en la figura 1). La estructura del código que implementa el modelo *fork-join* se muestra a continuación.

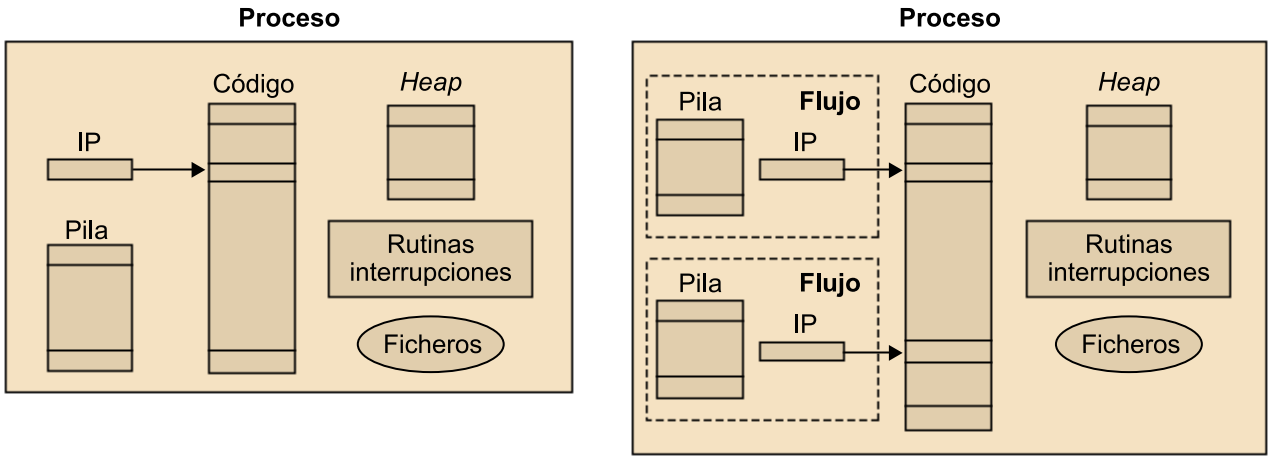
(2)Del inglés *single-program multiple-data.*

```
pid = fork();
if (pid==0) {
// código para que lo ejecute un proceso esclavo
}
else{
   // código para que lo ejecute el proceso padre
}
if (pid==0) exit;
else wait(0);
...
```

El principal problema que hace que esta solución no sea efectiva es la gran sobrecarga asociada a la creación y gestión de los procesos. En cambio, el modelo introducido anteriormente se aplica de manera mayoritaria en la programación para memoria compartida.

La alternativa más evidente a utilizar procesos tradicionales es usar flujos. Mientras que los procesos son muy pesados –pues se trata de códigos completamente independientes, con sus propias variables y estructuras de memoria–, los flujos comparten un espacio global de memoria entre las diferentes rutinas, tal y como muestra la figura 2.

Figura 2. Diferencias entre procesos y flujos

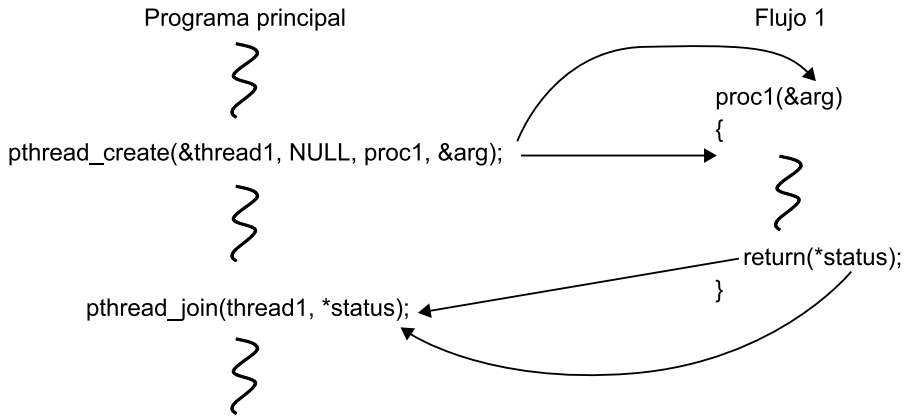


En otro módulo de la asignatura presentamos la biblioteca de flujos Pthreads, que cumple los estándares POSIX. También vimos un código que ejemplificaba el modelo *fork-join*, el cual se puede representar tal y como muestra la figura 3.

Ved también

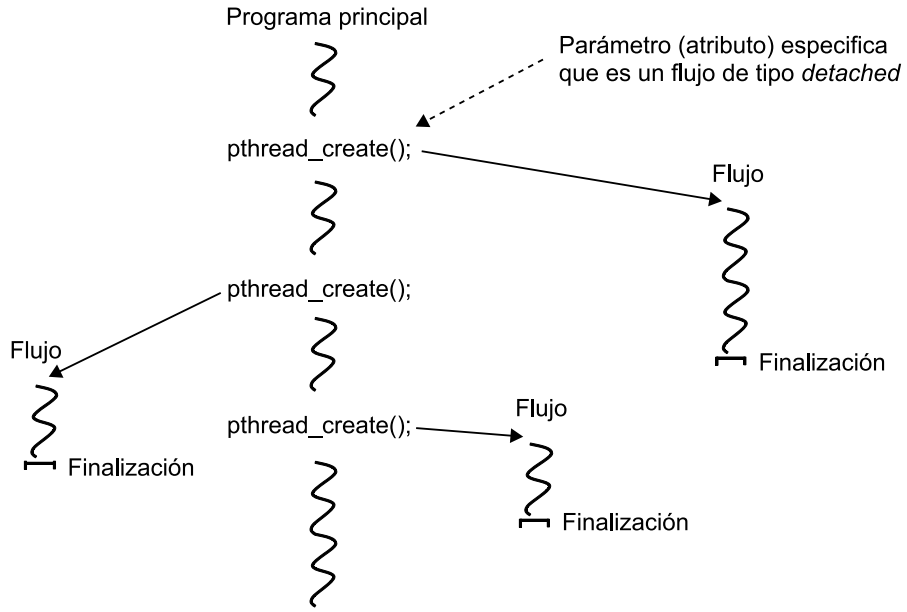
La biblioteca de flujos Pthreads se ha tratado en el módulo "Introducción a la computación de altas prestaciones" de esta asignatura.

Figura 3. Ejecución de un flujo mediante Pthreads



La biblioteca de Pthreads también permite utilizar flujos que no necesitan que el programa principal tenga que esperar su finalización (mediante *join*). Estos tipos de flujos se denominan *detached* y, cuando finalizan, son destruidos y sus recursos, liberados. La figura 4 muestra un esquema de funcionamiento de flujos tipo *detached*.

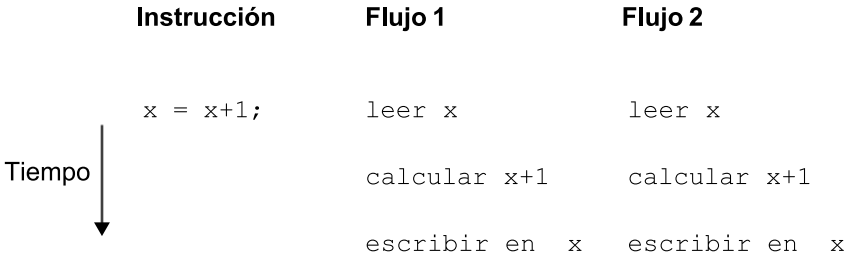
Figura 4. Ejecución de flujos tipo *detached*



Cuando se utilizan flujos, hay que tener en cuenta que el espacio de memoria es compartido y puede haber problemas tanto de consistencia como de rendimiento.

Podemos decir que una rutina es segura si la ejecución de múltiples instancias de la rutina en diferentes flujos produce resultados correctos. De este modo, será necesario certificar que las rutinas son seguras cuando acceden a datos compartidos. En el ejemplo de la figura 5, es posible ver dos flujos que incrementan el valor de la variable compartida `x`. Para llevar a cabo la instrucción hay que leer primero la variable `x`, después calcular `x + 1` y finalmente escribir el resultado en `x`. Puesto que los dos flujos leen el valor original de `x` al mismo tiempo, el valor final de `x` solo reflejará un único incremento.

Figura 5. Ejemplo de acceso a datos compartidos mediante flujos



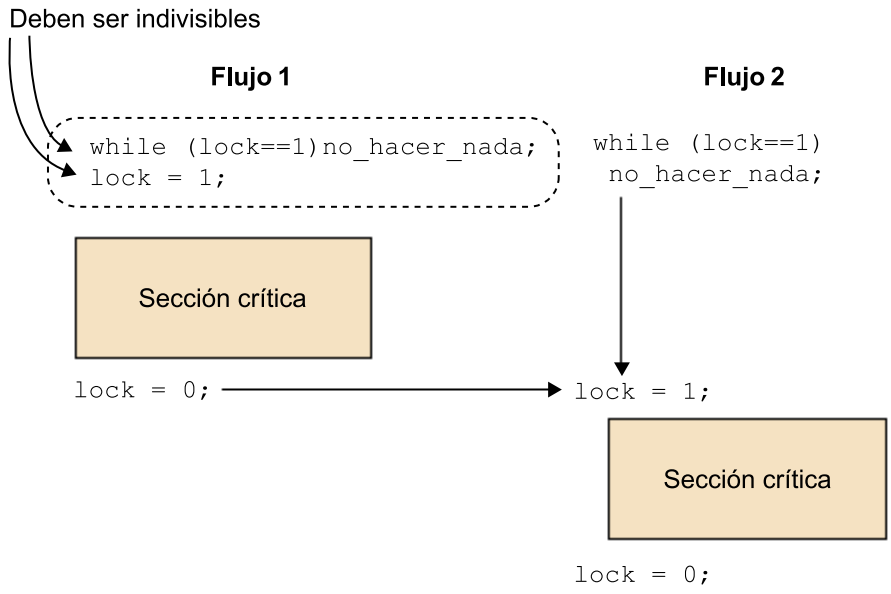
Un mecanismo para asegurar que solo un flujo accede a un recurso específico a la vez es definir secciones del código que únicamente se puedan ejecutar una vez al mismo tiempo. Este tipo de secciones de código se denominan secciones críticas, y el mecanismo asociado a la utilización de las mismas se conoce como exclusión mutua.

El mecanismo más sencillo para asegurar exclusión mutua en secciones críticas es la utilización de *locks*. Un *lock* es una variable de un bit, que cuando es 1 indica que un flujo ha entrado en la sección crítica y, cuando es 0, indica que ningún flujo ha entrado. Una posible implementación de exclusión mutua se produce mediante una espera activa (*busy waiting*), que consiste en los siguientes pasos (también ilustrados en la figura 6):

- Esperar hasta que el *lock* indique que no hay ningún flujo en la sección crítica.
- Bloquear la sección crítica cambiando el valor del *lock* a 1.
- Ejecutar el código de la sección crítica.
- Desbloquear la sección crítica y cambiar el valor del *lock* otra vez a 0.

El principal inconveniente de la espera activa es que consume muchos ciclos de *CPU* que no son realmente útiles.

Figura 6. Esquema de espera activa para acceder a una sección crítica desde 2 flujos



La biblioteca de Pthreads implementa los *locks* directamente con variables *lock* de exclusión mutua que se denominan variables de tipo *mutex*. Las secciones críticas se definen como se muestra a continuación:

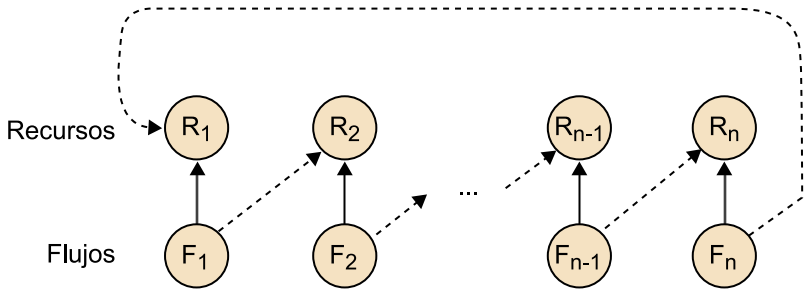
```
pthread_mutex_lock(&mutex1);
   Sección crítica
pthread_mutex_unlock(&mutex1);
```

Si un flujo llega a la llamada `pthread_mutex_lock`, quedará bloqueado hasta que el *lock* indicado esté abierto. Si hay varios flujos bloqueados en este lugar, cuando el *lock* se abre el sistema determinará el flujo que puede acceder a la sección crítica. Hay que tener en cuenta que solo el flujo que cierra un *lock* lo puede volver a abrir.

Uno de los principales problemas de la exclusión mutua es el conocido como interbloqueo[3]. El interbloqueo de dos flujos se produce cuando cada uno de los dos necesita el recurso que bloquea al otro flujo. Este fenómeno también se puede producir de manera circular, involucrando múltiples flujos (como ilustra la figura 7). Este fenómeno se conoce como abrazo mortal.

(3)En inglés, *deadlock*.

Figura 7. Abrazo mortal involucrando *n* flujos



Para prevenir los interbloqueos, la librería de Pthreads ofrece la operación `pthread_mutex_trylock()`, que permite consultar si un *lock* está abierto o cerrado antes de bloquear el flujo. Esta operación cierra el *lock* y retorna 0 si previamente estaba abierto, o bien retorna `EBUSY` si está cerrado.

Actividad

Buscad por vuestra parte otros mecanismos para prevenir interbloqueos, como por ejemplo semáforos o monitores.

1.2. OpenMP

Es posible desarrollar programas paralelos para memoria compartida con Pthreads, pero será necesario gestionar los flujos y tener en cuenta problemas de sincronización como los vistos hasta ahora. Obviamente, esto no favorece la productividad del programador y hace que se necesiten abstracciones de más alto nivel que faciliten la programación y la eficiente ejecución de aplicaciones paralelas, aspectos que proporciona OpenMP.

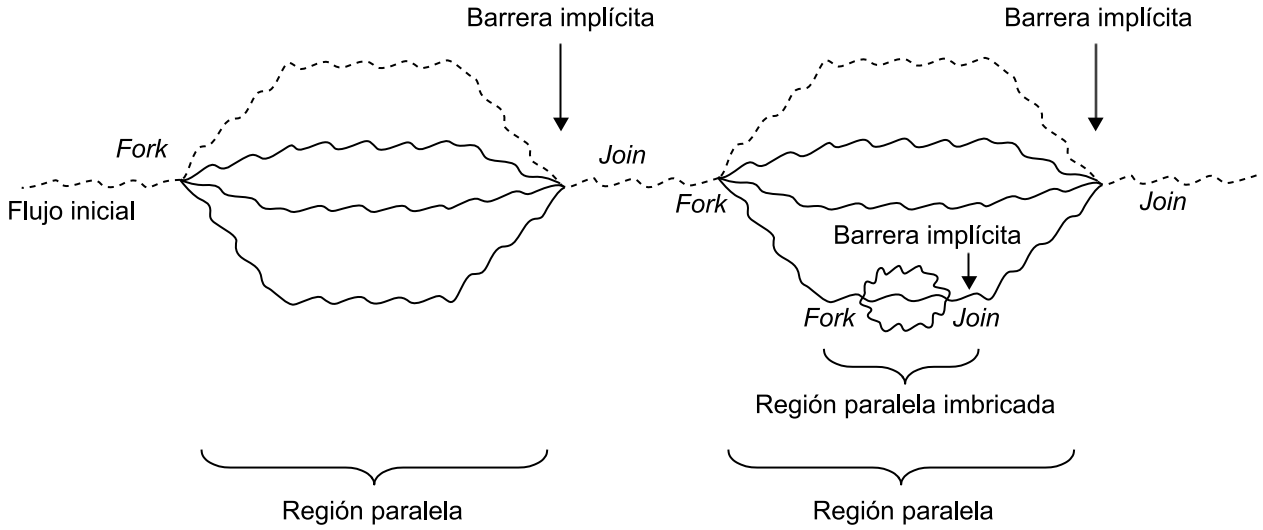
OpenMP para clústeres

El sistema Blacklight del Pittsburgh Supercomputing Center expone 4.096 núcleos con un modelo de memoria compartida.

OpenMP consiste en una interfaz de programación que extiende lenguajes de programación, como por ejemplo C/C++ y Fortran, mediante el uso de directivas o *pragmas*. OpenMP es el estándar actual para la programación de sistemas de memoria compartida, como por ejemplo sistemas multinúcleo y computadores de altas prestaciones basados en multiprocesadores con memoria virtual compartida. De manera típica, OpenMP se utiliza en sistemas con un número no demasiado grande de procesadores, a pesar de que hay implementaciones de sistemas de memoria compartida con centenares de núcleos. También hay versiones de OpenMP para otros tipos de sistemas, como por ejemplo clústeres y aceleradores, pero en estos casos el rendimiento de los programas OpenMP puede ser inferior a los que utilizan entornos de programación específicos para estos sistemas.

El modelo de ejecución de OpenMP es *fork-join*, como ilustra la figura 8. Esto quiere decir que inicialmente el programa trabaja con un único flujo hasta que llega a una región paralela, es decir, cuando llega al constructor `#pragma omp parallel` que pone en marcha un número de flujos esclavos (parte *fork* del modelo). El flujo que trabaja inicialmente es el flujo maestro de este conjunto de esclavos. Los flujos están numerados desde 0 hasta número de flujos -1. El maestro y los esclavos trabajan en paralelo en el bloque que aparece a continuación del constructor. Al acabar la región paralela hay una sincronización de todos los flujos, los esclavos mueren y queda únicamente el flujo maestro (parte *join* del modelo). A partir de este momento, el maestro continúa trabajando de manera secuencial a no ser que empiece otra región paralela. Tal y como ilustra la figura, también es necesario tener en cuenta que puede haber más de un nivel de regiones paralelas (región paralela imbricada).

Figura 8. Modelo *fork-join* de OpenMP



El siguiente código muestra un ejemplo de programa en OpenMP en el que podemos observar algunos elementos necesarios en programas OpenMP. Por ejemplo, incluir el fichero de cabeceras `omp.h` –en el que se definen las funciones de OpenMP– y utilizar directivas (empiezan por `#pragma omp`) para indicar al compilador la manera en que se tiene que distribuir el trabajo o cláusulas para indicar operaciones concretas como, por ejemplo, la reducción. En concreto, el código retorna la suma de la multiplicación de los elementos de dos vectores.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
   int   i, n;
   float a[100], b[100], sum;
```

```
   /* Algunas inicializaciones - región secuencial */
   n = 100;
   for (i=0; i < n; i++)
      a[i] = b[i] = i * 1.0;
sum = 0.0;

#pragma omp parallel for reduction(+:sum)   /* Región paralela */
for (i=0; i < n; i++)
   sum = sum + (a[i] * b[i]);

printf("Suma = %f\n",sum);   /* Región secuencial */
}
```

1.2.1. Directivas OpenMP

En las directivas OpenMP, se siguen las convenciones de los estándares para directivas de compilación en C/C++, se diferencia entre mayúsculas y minúsculas, solo se puede especificar un nombre de directiva por directiva y cada directiva se aplica al menos a la sentencia que sigue, que puede ser un bloque estructurado. En directivas largas, se puede utilizar el carácter `\` al final de la línea.

El formato general es el siguiente:

```
#pragma omp nombre_directiva [cláusulas, ...]
```

Donde

- `#pragma omp` requiere a todas las directivas OpenMP para C/C++.
- `nombre_directiva` es un nombre válido de directiva, y tiene que aparecer después de `pragma` y antes de cualquier cláusula. En nuestro ejemplo, es `parallel for`.
- `[cláusulas, ...]` son opcionales. Las cláusulas pueden ir en cualquier orden y repetirse cuando sea necesario, a menos que haya alguna restricción. En nuestro caso, son `reduction` y `private`.

1.2.2. Creación de flujos

Tal y como se ha indicado anteriormente, la directiva con la que se crean los flujos esclavos es `parallel` o se utiliza de la manera siguiente.

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

Con este `pragma`:

- Se crea un grupo de flujos y el flujo que los pone en marcha toma el rol de flujo maestro.

- El número de flujos que hay que crear se obtiene por la variable de entorno `OMP_NUM_THHREADS` o, de manera explícita, con una llamada a la librería, tal y como veremos más adelante. Si el valor es igual a cero, se ejecuta de manera secuencial.

- Hay una barrera implícita al final de la región, de modo que el flujo maestro espera a que acaben todos los esclavos para continuar con la ejecución secuencial.

- Cuando dentro de una región hay otro constructor `parallel`, cada esclavo crea otro grupo de flujos esclavos de los que sería el maestro. Esto se denomina paralelismo imbricado y, pese a que se pueden programar de esta manera, en algunas implementaciones de OpenMP la creación de grupos de esclavos dentro de una región paralela no está soportada.

- Las cláusulas de compartición de variables que soporta la directiva `parallel` son principalmente `private`, `firstprivate`, `lastprivate`, `default`, `shared`, `copyin` y `reduction`. El significado de estas cláusulas, así como el de otras de compartición, se verá más adelante.

1.2.3. Cláusulas de compartición de variables

Hemos visto en los casos anteriores que las directivas OpenMP tienen cláusulas con las que indicar cómo se lleva a cabo la compartición de variables, las cuales están todas en memoria compartida. Cada directiva tiene una serie de cláusulas que se le pueden aplicar. Estas son las siguientes.

- `private(lista)`: las variables de la lista son privadas a los flujos, lo que quiere decir que cada flujo tiene una variable privada con este nombre (que pueden tener valores diferentes en distintos flujos). Las variables no se inicializan antes de entrar en la región paralela (por lo tanto, no podemos esperar un valor inicial concreto al entrar en la región paralela), y no se guarda su valor al salir (de modo que no podemos esperar que el último valor de una variable privada se conserve al finalizar la región paralela).

- `firstprivate(lista)`: las variables son privadas a los flujos y se inicializan al entrar en la región paralela con el valor que tuviera la variable correspondiente del flujo maestro.

- `lastprivate(lista)`: son privadas a los flujos y al salir de la región paralela quedan con el valor de la última iteración (si estamos en un bucle

`for` paralelo) o sección (veremos más adelante el funcionamiento de las secciones).

- `shared(lista)`: indica las variables compartidas por todos los flujos. Por defecto todas son compartidas, por lo que no es realmente necesario utilizar esta cláusula en la mayoría de los casos.

- `default(shared|none)`: indica cómo serán las variables por defecto. Si se especifica `none`, habrá que señalar de manera explícita con la cláusula `shared` las que se desea que sean compartidas.

- `reduction(operador:lista)`: las variables de la lista se obtienen por la aplicación del operador, que tiene que ser asociativo.

- `copyin(lista)`: se utiliza para asignar el valor de la variable en el maestro a variables del tipo `threadprivate`.

1.2.4. Cláusulas de división del trabajo

Una vez se ha generado con `parallel` un conjunto de flujos esclavos, estos y el maestro pueden trabajar en la resolución de un problema en paralelo. Hay dos maneras de dividir el trabajo entre los flujos: mediante las directivas `for` y `sections`. Mientras que la primera directiva hace referencia al paralelismo de datos, la segunda se refiere al paralelismo funcional.

La directiva `for` tiene la forma siguiente:

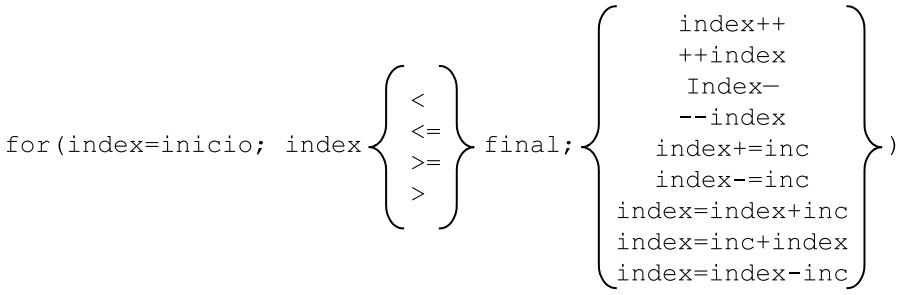
```
#pragma omp for [cláusulas]
   bucle for
```

Donde:

- Las iteraciones se ejecutan en paralelo por los flujos que ya existen, creados de manera previa con `parallel`.

- El bucle `for` debe tener una forma especial (forma canónica), tal y como muestra la figura 9. La parte de inicialización ha de tener una asignación; la parte del incremento, una suma o resta; la de de evaluación es la comparación de una variable entera con signo con un valor, utilizando un comparador mayor o menor (puede incluir igual); y los valores que aparecen en las tres partes de `for` deben ser enteros.

Figura 9. Forma canónica de los bucles en OpenMP



- Hay una barrera implícita al final del bucle, a no ser que se utilice la cláusula `nowait`.

- Las cláusulas de compartición de variables que admite son `private`, `firstprivate`, `lastprivate` y `reduction`.

- Puede aparecer una cláusula `schedule` para indicar de qué manera se dividen las iteraciones de `for` entre los flujos, es decir, la política de planificación.

Las políticas de planificación disponibles en OpenMP son las siguientes.

- `schedule(static, tamaño)`: las iteraciones se dividen según el tamaño de bloque que se indica. Por ejemplo, si el número de iteraciones del bucle es 100, están numeradas de 0 a 99 y el tamaño de bloque es 2, se consideran 50 bloques, cada uno de estos con dos iteraciones (bloques con iteraciones 0-1, 2-3, etc.). Si disponemos de 4 flujos, los bloques de iteraciones se asignan a los flujos de manera cíclica, con lo que al flujo 0 le corresponden las iteraciones 0-1, 8-9, etc.; al flujo 1, las iteraciones 2-3, 10-11, etc.; al 2, las 4-5, 12-13, etc.; y al 3, las 6-7, 14-15, etc. Si no se indica ningún tamaño de bloque, las iteraciones se dividen por igual entre los flujos y los bloques son de tamaño máximo: si tenemos 12 iteraciones y 4 flujos, se asignan al flujo 0 las iteraciones 0-2, al 1 las 3-5, al 2 las 6-8 y al 4, las 9-11.

- `schedule(dynamic, tamaño)`: las iteraciones se agrupan según el tamaño de bloque especificado y se asignan a los flujos de manera dinámica cuando van acabando su trabajo. En el caso anterior de 100 iteraciones, tamaño de bloque 2 y 4 flujos, se asignan inicialmente al flujo *i* las iteraciones `2i` y `2i + 1`, y a partir de aquí el resto de los bloques de dos iteraciones se asignan a los flujos según se vayan quedando sin trabajo. Cuando el volumen de computación de cada iteración no se conoce *a priori*, quizá sea preferible utilizar una asignación dinámica, puesto que permite reducir el desbalanceo.

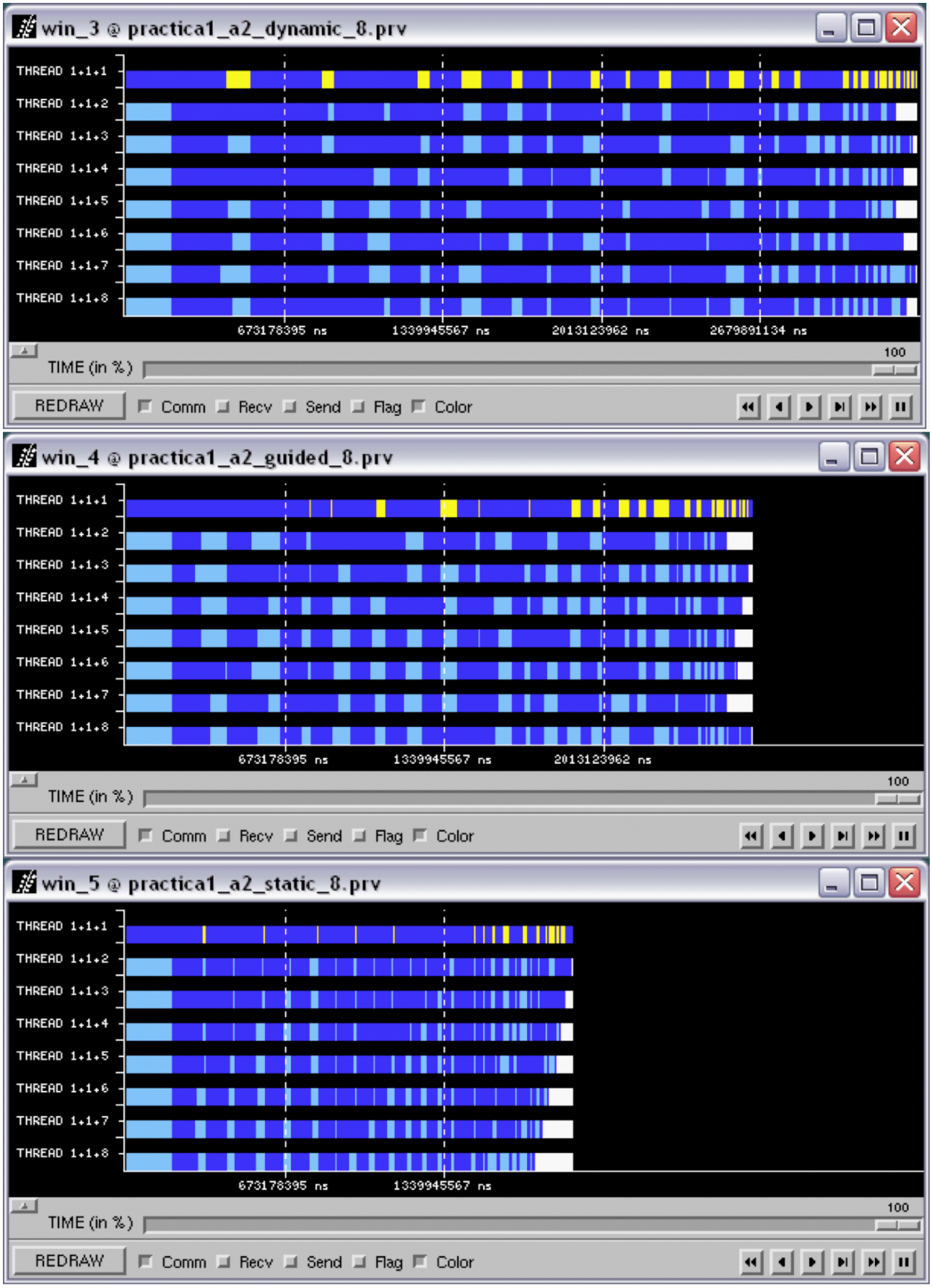
- `schedule(guided, tamaño)`: las iteraciones se asignan dinámicamente a los flujos pero con tamaño de bloque decreciente hasta llegar al tamaño de bloque que se indica (si no se indica ningún valor, es 1 por defecto).

El tamaño inicial de bloque y la forma en que decrece dependen de su implementación.

- `schedule(runtime)`: decide la política de planificación en tiempo de ejecución mediante la variable de entorno `OMP_SCHEDULE`.

La figura 10 muestra los trazos obtenidos de la ejecución de una aplicación OpenMP utilizando las políticas de planificación `static`, `dynamic` y `guided`. Se puede apreciar el hecho de que el tiempo de ejecución (eje de de las *x*) puede variar de manera muy significativa en función de la política de planificación.

Figura 10. Trazos de la ejecución de una misma aplicación OpenMP con diferentes políticas de planificación para la distribución del trabajo



La directiva `sections` tiene la forma siguiente.

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

Donde:

- Cada sección se ejecuta mediante un flujo. La forma en que se distribuyen las secciones entre los flujos depende de la implementación concreta de OpenMP.
- Hay una barrera final, a menos que se utilice la cláusula `nowait`.
- Las cláusulas de compartición de variables que admite son `private`, `firstprivate`, `lastprivate` y `reduction` .

Dado que `for` y `sections` se utilizan con mucha frecuencia inmediatamente después de haber creado los flujos con `parallel`, hay dos directivas combinadas que se usan mucho y que se especifican a continuación:

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

y

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

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

1.2.5. Directivas de sincronización

Dentro de una región paralela, puede ser necesario sincronizar los flujos en el acceso a algunas variables o zonas de código para evitar situaciones de interbloqueo. OpenMP ofrece varias primitivas con esta finalidad.

- `single`: el código afectado por la directiva lo ejecutará un único flujo. Los flujos que no están trabajando durante la ejecución de la directiva esperan al final. Admite las cláusulas `private`, `firstprivate` y `nowait`. No está

permitido hacer bifurcaciones hacia/desde un bloque `single`. Resulta útil para secciones de código que pueden ser no seguras para su ejecución paralela (por ejemplo, para entrada/salida).

- `master`: el código lo ejecuta solo el flujo maestro. El resto de los flujos no ejecutan esta sección de código.

- `critical`: protege una sección de código para que pueda acceder un solo flujo al mismo tiempo. Se le puede asociar un nombre de la forma:

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

  Por lo que puede haber secciones críticas protegiendo zonas diferentes del programa, de modo que varios flujos a la vez pueden acceder a secciones con nombres diferentes. Las regiones que tengan el mismo nombre se tratan como la misma región. Todas las que no tienen nombre se consideran la misma. No está permitido hacer bifurcaciones hacia/desde un bloque `critical`.

- `atomic`: asegura que una posición de memoria se modifique sin que múltiples flujos intenten escribirla de manera simultánea. Se aplica a la sentencia que sigue a la directiva. Solo se asegura en modo exclusivo la actualización de la variable, pero no la evaluación de la expresión.

- `barrier`: sincroniza todos los flujos. Cuando un flujo llega a la barrera espera a que lleguen los otros y, cuando han llegado todos, siguen su ejecución.

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

- `ordered`: asegura que el código se ejecute en el orden en el que las iteraciones se ejecutan en su forma secuencial. Puede aparecer solo una vez en el contexto de una directiva `for` o `parallel for`. No está permitido hacer bifurcaciones hacia/desde un bloque `ordered`. Solo puede haber un único flujo ejecutándose de manera simultánea en una sección `ordered`. Una iteración de un bucle no puede ejecutar la misma directiva `ordered` más de una vez, y no tiene que ejecutar más de una directiva `ordered`. Un bucle con una directiva `ordered` debe contener una cláusula `ordered`.

- `flush`: tiene la forma siguiente.

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

  Asegura que el valor de las variables se actualiza en todos los flujos en los que son visibles. Si no hay lista de variables, se actualizarán todas.

1.2.6. Funciones y variables

Las funciones que típicamente se utilizan más en OpenMP son las relacionadas con el número de flujos, es decir, `omp_set_num_threads`, `omp_get_num_threads` y `omp_get_thread_num`. Otras funciones importantes son las siguientes.

- `omp_get_max_threads`: obtiene la máxima cantidad posible de flujos.
- `omp_get_num_procs`: retorna el número máximo de procesadores que se pueden asignar al programa.
- `omp_in_parallel`: retorna un valor diferente a cero si se ejecuta dentro de una región paralela.

Se puede poner o quitar lo que el número de flujos se pueda asignar de manera dinámica en las regiones paralelas (`omp_set_dynamic`), o se puede comprobar si está permitido el ajuste dinámico con la función `omp_get_dynamic`, que retorna un valor diferente a cero si está permitido el ajuste dinámico del número de flujos.

Es posible desautorizar el paralelismo imbricado con `omp_set_nested`, o comprobar si está permitido con `omp_get_nested`, que retorna un valor diferente a cero si lo está.

También hay funciones para la gestión de *locks*, como las que se muestran a continuación.

- `void omp_init_lock(omp_lock_t *lock)`: para inicializar un *lock*, que se inicializa como no bloqueado.
- `void omp_init_destroy(omp_lock_t *lock)`: para destruir un *lock*.
- `void omp_set_lock(omp_lock_t *lock)`: para bloquear un *lock*.
- `void omp_unset_lock(omp_lock_t *lock)`: para desbloquear un *lock*.
- `void omp_test_lock(omp_lock_t *lock)`: para comprobar si un *lock* está bloqueado o no y así evitar bloqueos indeseados.

1.2.7. Entorno de compilación y ejecución

Para compilar un programa OpenMP, hay que disponer de un compilador que pueda interpretar las directivas que aparecen en el código. OpenMP está soportado por el compilador GCC desde la versión 4.1 y también por otros propietarios, como por ejemplo el ICC de Intel. En este subapartado, utilizaremos el lenguaje C para ejemplificar el uso de OpenMP. La opción de compilación para GCC es `-fopenmp` o `-openmp` y para ICC es `-openmp`. Así pues, la siguiente línea serviría para compilar un programa OpenMP con GCC:

```
gcc -fopenmp -o programa programa.c
```

La ejecución del fichero binario (`programa` en nuestro ejemplo) se lleva a cabo como en cualquier otro programa, pero hay que determinar cuántos flujos intervendrán en la ejecución paralela. Podemos utilizar una variable de entorno (`OMP_NUM_THREADS`), que nos indica el número de flujos que hay que utilizar. Si no se inicializa esta variable, cuando ejecutemos nuestro programa OpenMP tendrá un valor por defecto, que acostumbra a coincidir con el número de núcleos del nodo en el que estamos trabajando.

Antes de la ejecución se pueden establecer los valores de la variable, por ejemplo, desde el intérprete de órdenes. Dado el caso, si establecemos el valor de la variable como a continuación:

```
export OMP_NUM_THREADS=6
```

Estableceremos el número de flujos que hay que utilizar en la región paralela a seis, de manera independiente del número de núcleos del que disponemos. Esto quiere decir que es posible utilizar más flujos que procesadores hay en nuestro sistema. En términos generales, el rendimiento de la ejecución de un programa OpenMP utilizando más flujos que procesadores disponibles en el sistema es inferior al que podemos conseguir utilizando un flujo por procesador, debido a la sobrecarga que genera la creación y gestión de los flujos. De hecho, podemos llegar a observar que el tiempo de ejecución del programa paralelo sea más elevado que el del programa secuencial.

Es muy importante recordar que el tamaño del problema es vital para explotar el paralelismo. Así pues, aunque dispongamos de varios núcleos, el tamaño del problema (el número de intervalos) puede no ser lo suficientemente grande como para que se note el efecto de la paralelización, especialmente si tenemos en cuenta que hay zonas que se ejecutan de manera secuencial.

Aun así, aunque no dispongamos de suficientes procesadores para ejecutar todos los flujos, podría ocurrir que se obtuviera mejor rendimiento debido a unas mejores particiones y localidad de los datos. Por ejemplo, puede suceder que al dividir el problema en varios flujos, los datos que hay que tratar en cada iteración de un bucle quepan en una página de memoria caché y, en consecuencia, se pueda acelerar la ejecución de manera muy significativa.

Actividad

Puesto que actualmente la mayoría de los computadores personales disponen de múltiples núcleos, podéis experimentar y buscar explicación al tiempo que se obtiene de la ejecución de algún programa OpenMP variando el número de flujos, de modo que este número sea tanto menor como mayor que el número de núcleos disponibles.

Hay cuatro variables de entorno. Además de `OMP_NUM_THREADS` también están disponibles:

- `OMP_SCHEDULE` indica el tipo de planificación para `for` y `parallel for`.

- `OMP_DYNAMIC` autoriza o desautoriza el ajuste dinámico del número de flujos.
- `OMP_NESTED` autoriza o desautoriza el paralelismo imbricado. Por defecto, no está autorizado.

2. Modelos de programación gráfica

En los últimos años, la capacidad de cálculo de los procesadores gráficos[4] se ha hecho más elevada que la de los procesadores multinúcleo más avanzados para llevar a cabo ciertas tareas. Esto ha hecho que este tipo de dispositivos resulten muy populares para el cómputo de algoritmos de propósito general y no solo para la generación de gráficos. La computación de propósito general sobre *GPU* se conoce popularmente como *GPGPU*[5].

(4)*GPU*, del inglés *graphics processing unit.*

(5)Del inglés *general-purpose computing on graphics processing unit.*

Las *GPU* proporcionan rendimientos muy elevados solo para ciertas aplicaciones, debido a sus características arquitecturales y de funcionamiento. De manera general, podemos decir que las aplicaciones que pueden aprovechar mejor las capacidades de las *GPU* son aquellas que cumplen las dos condiciones siguientes.

- Trabajan sobre vectores de datos grandes.
- Tienen un paralelismo de grano fino tipo SIMD.

Uno de los principales inconvenientes a la hora de trabajar con *GPU* es la dificultad para el programador de transformar programas diseñados para *CPU* tradicionales en programas que puedan ser ejecutados de manera eficiente en una *GPU*. Por este motivo, se han desarrollado modelos de programación –ya sean propietarios (CUDA) o abiertos (OpenCL)– que proporcionan al programador un nivel de abstracción más cercano a la programación para *CPU*, y que simplifican considerablemente su tarea.

En este apartado, estudiaremos los principales modelos de programación para *GPU* orientadas a aplicaciones de propósito general, de manera específica *CUDA* y OpenCL, que son los estándares actuales para la programación de procesadores gráficos.

2.1. *CUDA*

CUDA[6] es una especificación inicialmente propietaria desarrollada por Nvidia como plataforma para sus productos *GPU*. *CUDA* incluye las especificaciones de la arquitectura y un modelo de programación asociado. En este subapartado, estudiaremos el modelo de programación *CUDA*. En el caso de aquellas *GPU* que implementan la arquitectura y las especificaciones definidas en *CUDA*, hablaremos de dispositivos compatibles con *CUDA*.

(6)Del inglés *compute unified device architecture.*

2.1.1. Arquitectura compatible con *CUDA*

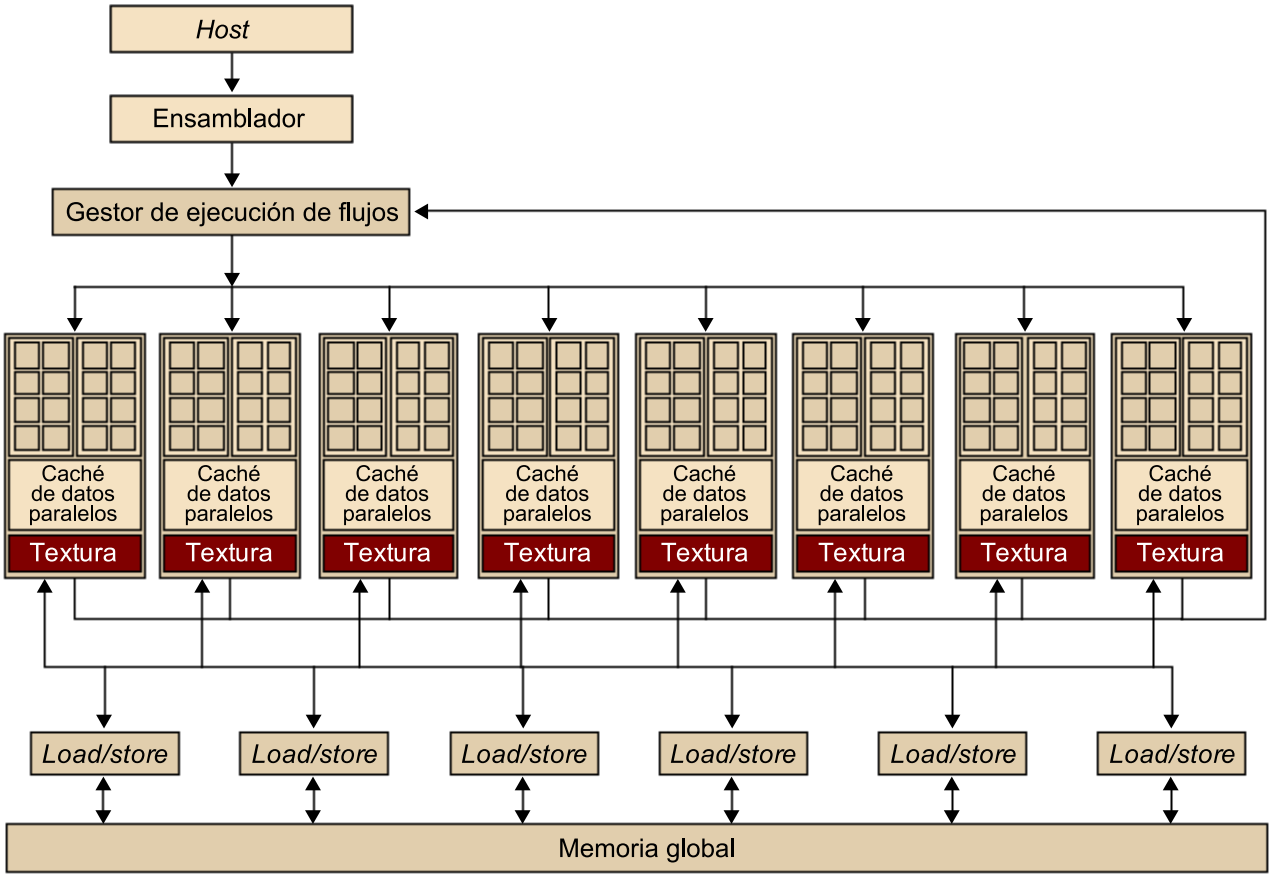
La figura 11 muestra la arquitectura de una *GPU* genérica compatible con *CUDA*. Esta arquitectura está organizada en una serie de multiprocesadores[7] que tienen una cantidad elevada de flujos de ejecución. En la figura, dos *SM* forman un bloque a pesar de que el número de *SM* por bloque depende de la implementación concreta del dispositivo. Además, cada *SM* de la figura tiene un número *de streaming processors* (*SP*) que comparten la lógica de control y la memoria caché de instrucciones.

(7)*SM*, del inglés *streaming multiprocessors.*

La *GPU* tiene una memoria DRAM de tipo *GDDR*[8], la cual está indicada en la figura 11 como memoria global. Esta memoria *GDDR* se diferencia de la memoria DRAM de la placa base del computador en el hecho de que esencialmente se utiliza para gráficos (*framebuffer*). Para aplicaciones gráficas, esta mantiene las imágenes de vídeo y la información de las texturas. En cambio, para cálculos de propósito general, funciona como memoria externa con mucho ancho de banda pero con una latencia algo más elevada que la memoria típica del sistema. Aun así, para aplicaciones masivamente paralelas, el mayor ancho de banda compensa la latencia más elevada.

(8)Del inglés *graphics double data rate.*

Figura 11. Esquema de la arquitectura de una *GPU* genérica compatible con *CUDA*



Observad que esta arquitectura genérica es muy similar a la G80, que soporta hasta 768 flujos por *SM*, lo que suma un total de 12.000 flujos en un único chip.

La arquitectura GT200, posterior a la G80, tiene 240 *SP* y supera el TFlop de pico teórico de rendimiento. Puesto que los *SP* son masivamente paralelos, es posible utilizar todavía más flujos por aplicación que con la G80. GT200 soporta 1.024 flujos por *SM* y, en total, suma en torno a 30.000 flujos por chip. Por lo tanto, la tendencia muestra claramente que el nivel de paralelismo soportado por las *GPU* está aumentando rápidamente. Será muy importante, por lo tanto, intentar explotar este nivel tan elevado de paralelismo cuando se desarrollen aplicaciones de propósito general para *GPU*.

2.1.2. Entorno de programación

CUDA se desarrolló para aumentar la productividad en el desarrollo de aplicaciones de propósito general para *GPU*. Desde el punto de vista del programador, el sistema está compuesto por un *host* que es una *CPU* tradicional, como por ejemplo un procesador de arquitectura Intel, y uno o más dispositivos[9] que son *GPU*.

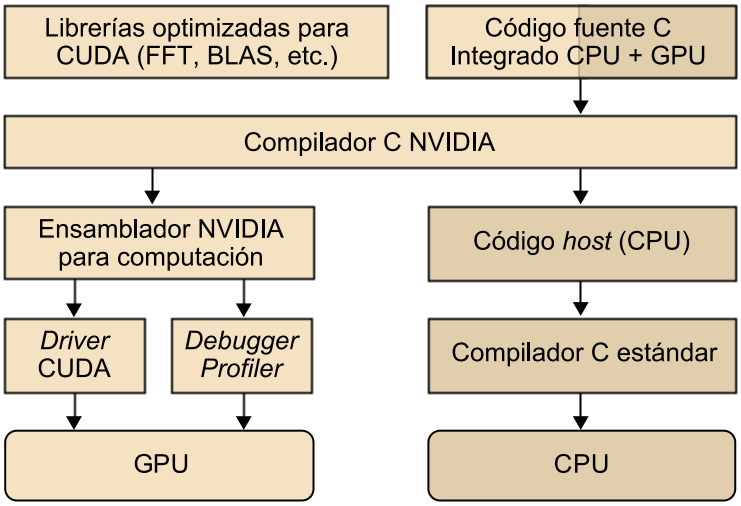
(9)En inglés, *devices*.

CUDA pertenece al modelo *SIMD* y, por lo tanto, está pensada para explotar el paralelismo en un ámbito de datos. Esto significa que un conjunto de operaciones aritméticas se pueden ejecutar sobre un conjunto de datos de manera simultánea. Afortunadamente, muchas aplicaciones tienen partes con un nivel muy elevado de paralelismo en un ámbito de datos.

Un programa en *CUDA* consiste en una o más fases que pueden ser ejecutadas o bien en el *host* (*CPU*) o bien en el dispositivo *GPU*. Las fases en las que hay muy poco o nada de paralelismo en un ámbito de datos se implementan en el código que se ejecutará en el *host*, y las fases con un nivel de paralelismo elevado en el ámbito de datos se implementan en el código que se ejecutará en el dispositivo.

Tal y como muestra la figura 12, el compilador de NVIDIA (`nvcc`) se encarga de proporcionar la parte del código correspondiente al *host* y al dispositivo durante el proceso de compilación. El código correspondiente al *host* es simplemente código ANSI C, que se compila mediante el compilador de C estándar del *host*, como si fuera un programa para *CPU* convencional. El código correspondiente al dispositivo también es ANSI C pero con extensiones que incluyen palabras clave para definir funciones que tratan los datos en paralelo. Estas funciones se denominan *kernels*.

Figura 12. Esquema de bloques del entorno de compilación de *CUDA*



El código del dispositivo se vuelve a compilar con `nvcc` y entonces ya se puede ejecutar en el dispositivo *GPU*. En situaciones en las que no hay ningún dispositivo disponible o bien el *kernel* es más apropiado para una *CPU*, también es posible ejecutar los *kernels* en una *CPU* convencional mediante herramientas de emulación que proporciona la plataforma *CUDA*. La figura 13 muestra todas las etapas del proceso de compilación y la tabla 1 describe cómo el compilador `nvcc` interpreta los diferentes tipos de ficheros de entrada.

Figura 13. Pasos en la compilación de código *CUDA*, desde .cu hasta .cu.c

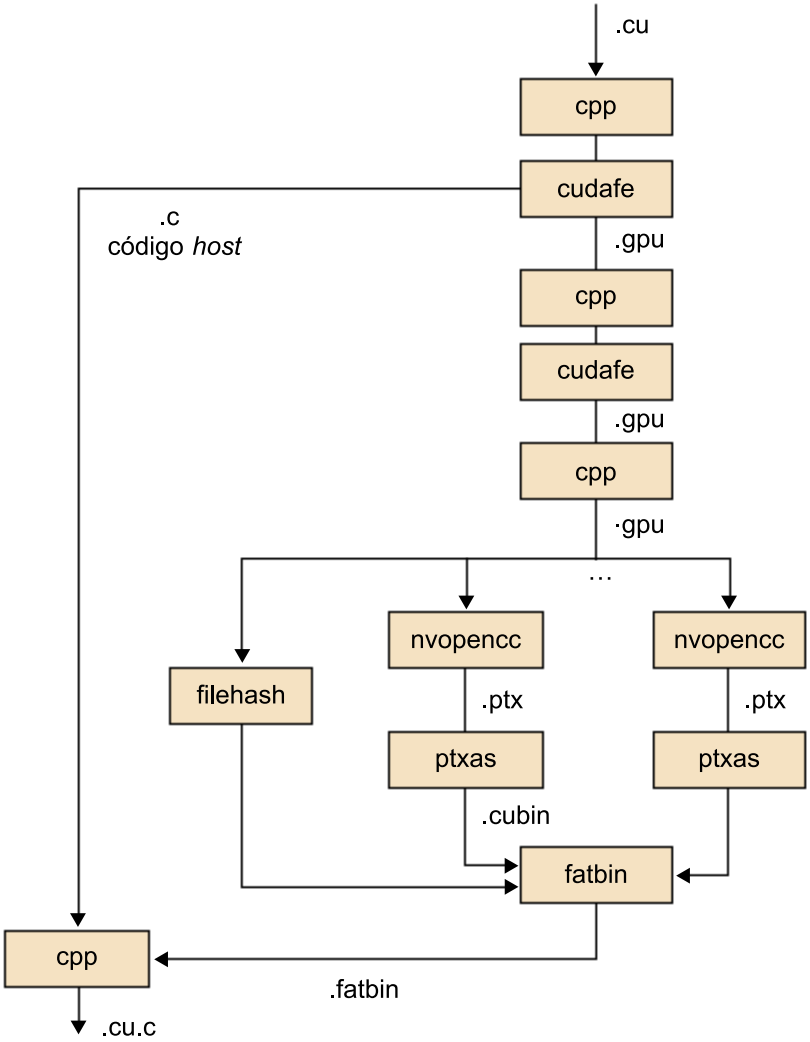


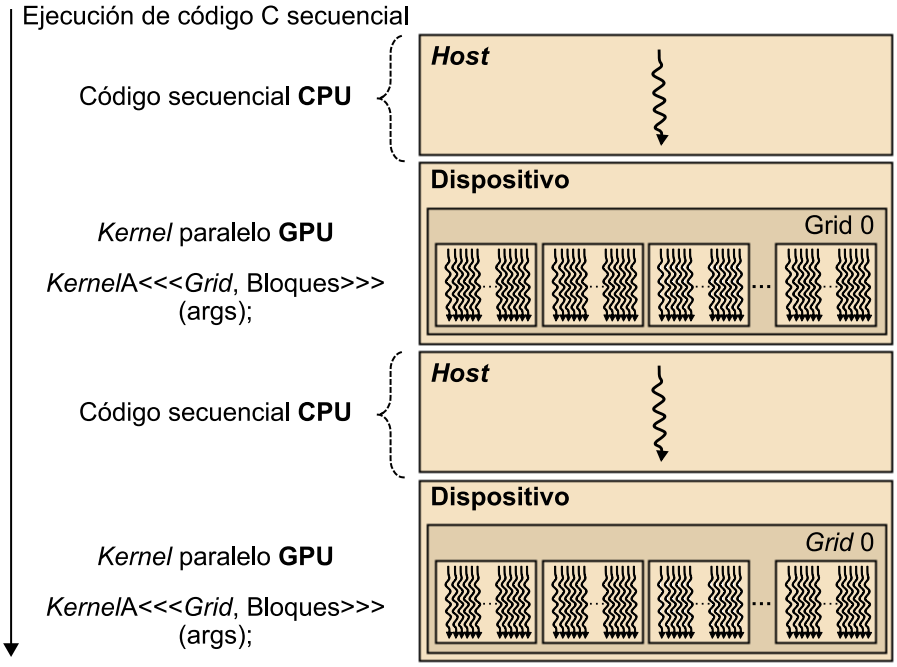
Tabla 1. Interpretación de nvcc de los ficheros de entrada

.cu	Código fuente CUDA, que contiene tanto el código del *host* como las funciones del dispositivo
.cup	Código fuente CUDA preprocesado, que contiene tanto el código del *host* como las funciones del dispositivo
.c	Fichero de código fuente C
.cc, .cxx, .cpp	Fichero de código fuente C++
.gpu	Fichero intermedio gpu
.ptx	Fichero asemblador intermedio ptx
.o, .obj	Fichero de objeto
.a, .lib	Fichero de librería
.res	Fichero de recurso
.so	Fichero de objeto compartido

Para explotar el paralelismo a nivel de datos, los *kernels* tienen que generar una cantidad de flujos de ejecución bastante elevada (por ejemplo, del orden de decenas o centenares de miles de flujos). Debemos tener en cuenta que los flujos de *CUDA* son mucho más ligeros que los flujos de *CPU*. De hecho, podremos asumir que para generar y planificar estos flujos solo necesitaremos unos pocos ciclos de reloj debido al soporte *hardware*, en contraste con los flujos convencionales para *CPU*, que normalmente requieren miles de ciclos.

La ejecución de un programa típico *CUDA* se muestra en la figura 14. La ejecución empieza con ejecución en el *host* (*CPU*). Cuando se invoca un *kernel*, la ejecución se mueve hacia el dispositivo (*GPU*) en el que se genera un número muy elevado de flujos. El conjunto de todos estos flujos que se generan cuando se invoca un *kernel* se denomina *grid*. En la figura 14, se muestran dos *grids* de flujos. Un *kernel* finaliza cuando todos sus flujos finalizan su ejecución en el *grid* correspondiente. Una vez finalizado el *kernel*, la ejecución del programa continúa en el *host* hasta que se invoca otro *kernel*.

Figura 14. Etapas en la ejecución de un programa típico *CUDA*



2.1.3. Modelo de memoria

Es importante remarcar que la memoria del *host* y del dispositivo son espacios de memoria completamente separados. Esto refleja la realidad de que los dispositivos son típicamente tarjetas que tienen su propia memoria DRAM. Para ejecutar un *kernel* en el dispositivo *GPU*, normalmente hay que seguir estos pasos.

- Reservar memoria en el dispositivo (paso 1 de la figura 15).

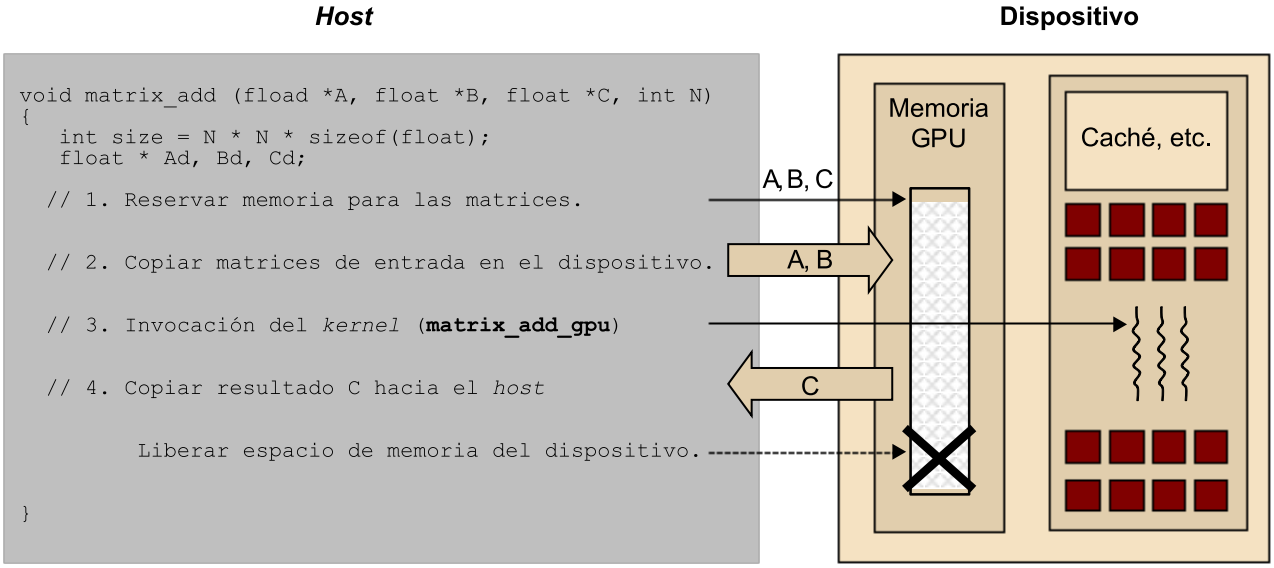
- Transferir los datos necesarios desde el *host* al espacio de memoria asignado al dispositivo (paso 2 de la figura 15).

- Invocar la ejecución del *kernel* en cuestión (paso 3 de la figura 15).

- Transferir los datos con los resultados desde el dispositivo hacia el *host* y liberar la memoria del dispositivo (si ya no es necesaria), una vez finalizada la ejecución del *kernel* (paso 4 de la figura 15).

El entorno *CUDA* proporciona una interfaz de programación que le simplifica estas tareas al programador. Por ejemplo, solo hay que especificar qué datos se deben transferir desde el *host* al dispositivo y viceversa.

Figura 15. Esquema de los pasos para la ejecución de un *kernel* en una *GPU*



Durante todo este apartado, utilizaremos el mismo programa de ejemplo que efectúa la suma de dos matrices. El ejemplo siguiente muestra el código correspondiente a la implementación secuencial en C estándar de la suma de matrices para arquitectura tipo *CPU*.

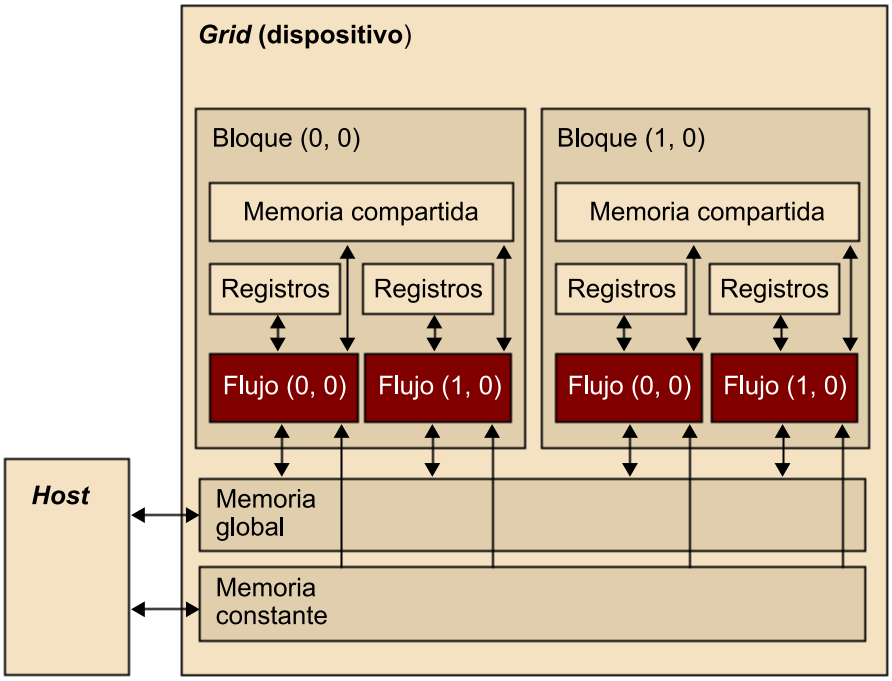
```
void matrix_add_cpu (float *A, float *B, float *C, int N)
{
   int i, j, index;
   for (i=0; i<N; i++){
      for (j=0; j<N; j++){
         index = i+j*N;
         C[index] = C[index] + B[index];
      }
   }
}

int main(){
   matrix_add_cpu(a, b, c, N);
```

```
}
```

La figura 16 muestra el modelo de memoria de *CUDA* expuesto al programador en términos de asignación, transferencia y utilización de los diferentes tipos de memoria del dispositivo. En la parte inferior de la figura, se encuentran las memorias de tipo global y constante. A estas memorias el *host* les puede transferir datos de manera bidireccional, por *grid*. Desde el punto de vista del dispositivo, se puede acceder a diferentes tipos de memoria con los siguientes modos.

- Acceso de lectura/escritura a la memoria global, por *grid*.
- Acceso solo de lectura a la memoria constante, por *grid*.
- Acceso de lectura/escritura a los registros, por flujo.
- Acceso de lectura/escritura a la memoria local, por flujo.
- Acceso de lectura/escritura a la memoria compartida, por bloque.
- Acceso de lectura/escritura.

Figura 16. Modelo de memoria de *CUDA*



La interfaz de programación para asignar y liberar memoria global del dispositivo consiste en dos funciones básicas: `cudaMalloc()` y `cudaFree()`. La función `cudaMalloc()` se puede llamar desde el código del *host* para asignar un espacio de memoria global para un objeto. Como habréis observado, esta función es muy similar a la función `malloc()` de la librería de C estándar, puesto que *CUDA* es una extensión del lenguaje C y pretende mantener las interfaces muy similares a las originales.

La función `cudaMalloc()` tiene dos parámetros. El primer parámetro es la dirección del puntero hacia el objeto una vez se haya asignado el espacio de memoria. Este puntero es genérico y no depende de ningún tipo de objeto, por lo

que se deberá hacer *cast* en tipo (`void **`). El segundo parámetro es el tamaño del objeto que se quiere asignar en *bytes*. La función `cudaFree()` libera el espacio de memoria del objeto indicado como parámetro de la memoria global del dispositivo. El código siguiente muestra un ejemplo de cómo utilizar estas dos funciones. Después de llevar a cabo el `cudaMalloc(), Matriz` apunta a una región de la memoria global del dispositivo que se le ha asignado.

```
float *Matriz;
int tamaño = ANCHURA * LONGITUD * sizeof(float);
cudaMalloc((void **) &Matriz, tamaño);
...
cudaFree(Matriz);
```

Una vez que un programa ha asignado memoria global del dispositivo para los objetos o las estructuras de datos del programa, es posible transferir los datos que serán necesarios para la computación desde el *host* hacia el dispositivo. Esto se hace mediante la función `cudaMemcpy()`, que permite transferir datos entre memorias. Hay que tener en cuenta que la transferencia es asíncrona.

La función `cudaMemcpy()` tiene cuatro parámetros. El primero es un puntero a la dirección de destino donde se tienen que copiar los datos. El segundo parámetro apunta a los datos que se tienen que copiar. El tercer parámetro especifica el número de *bytes* que se deben copiar. Finalmente, el cuarto parámetro indica el tipo de memoria involucrado en la copia, que puede ser uno de los siguientes.

- `cudaMemcpyHostToHost`: de la memoria del *host* hacia la memoria del mismo *host*.
- `cudaMemcpyHostToDevice`: de la memoria del *host* hacia la memoria del dispositivo.
- `cudaMemcpyDeviceToHost`: de la memoria del dispositivo hacia la memoria del *host*.
- `cudaMemcpyDeviceToDevice`: de la memoria del dispositivo hacia la memoria del dispositivo.

Hay que tener en cuenta que esta función se puede utilizar para copiar datos de la memoria de un mismo dispositivo, pero no entre diferentes dispositivos. A continuación, se muestra un ejemplo de transferencia de datos entre *host* y dispositivo basado en el ejemplo de la figura 15.

```
void matrix_add (fload *A, float *B, float *C, int N)
{
   int size = N * N * sizeof(float);
   float * Ad, Bd, Cd;

   // 1. Reservar memoria para las matrices
   cudaMalloc((void **) &Ad, size);
```

```
    cudaMalloc((void **) &Bd, size);
    cudaMalloc((void **) &Cd, size);
    // 2. Copiar matrices de entrada al dispositivo
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    ...
    // 4. Copiar resultado C hacia el host
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    ...
```

Además de los mecanismos de asignación de memoria y transferencia de datos, *CUDA* también soporta diferentes tipos de variables. Las distintas variables que utilizan los diferentes tipos de memoria se utilizan en varios ámbitos y tendrán ciclos de vida distintos, como resume la tabla 2.

Tabla 2. Diferentes tipos de variables en CUDA

Declaración variables	Tipos de memoria	Ámbito	Ciclo de vida
Por defecto (diferentes a vectores)	Registro	Flujo	*Kernel*
Vectores por defecto	Local	Flujo	*Kernel*
__device__, __shared__, int SharedVar;	Compartida	Bloque	*Kernel*
__device__, int GlobalVar;	Global	*Grid*	Aplicación
__device__, __constant__, int ConstVar;	Constante	*Grid*	Aplicación

2.1.4. Definición de *kernels*

El código que se ejecuta en el dispositivo (*kernel*) es la función que ejecutan los diferentes flujos durante la fase paralela, cada uno de estos en el rango de datos que le corresponde. Hay que recordar que *CUDA* sigue el modelo *SPMD* y, por lo tanto, todos los flujos ejecutan el mismo código. A continuación, se muestra la función o *kernel* de la suma de matrices y su llamada.

```
__global__ matrix_add_gpu (fload *A, float *B, float *C, int N)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   int index = i + j*N;
   if (i<N && j<N){
         C[index] = A[index] + B[index];
   }
}
```

```
int main(){
   dim3 dimBlock(blocksize, blocksize);
   dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
   matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

Podemos observar la utilización de la palabra clave específica de *CUDA* `__global__` ante la declaración de `matrix_add_gpu()`. Esta palabra clave indica que esta función es un *kernel* y que hay que llamarlo desde el *host* para generar el *grid* de flujos que ejecutará el *kernel* en el dispositivo. Además de `__global__`, hay otras dos palabras clave que se pueden utilizar ante la declaración de una función:

- La palabra clave `__device__` indica que la función declarada es una función *CUDA* de dispositivo. Una función de dispositivo se ejecuta únicamente en un dispositivo *CUDA* y solo se puede llamar desde un *kernel* o desde otra función de dispositivo. Estas funciones no pueden tener ni llamadas recursivas ni llamadas indirectas a funciones mediante punteros.

- La palabra clave `__host__` indica que la función es una función de *host*, es decir, una función simple de C que se ejecuta en el *host* y, por lo tanto, que puede ser llamada desde cualquier función de *host*. Por defecto, todas las funciones en un programa *CUDA* son funciones de *host* si no se especifica ninguna palabra clave en la definición de la función.

Las palabras clave `__host__` y `__device__` se pueden utilizar de manera simultánea en la declaración de una función. Esta combinación hace que el compilador genere dos versiones de la misma función: una que se ejecuta en el *host* y que solo se puede llamar desde una función de *host*, y otra que se ejecuta en el dispositivo y que solo se puede llamar desde el dispositivo o función de *kernel*.

2.1.5. Organización de flujos

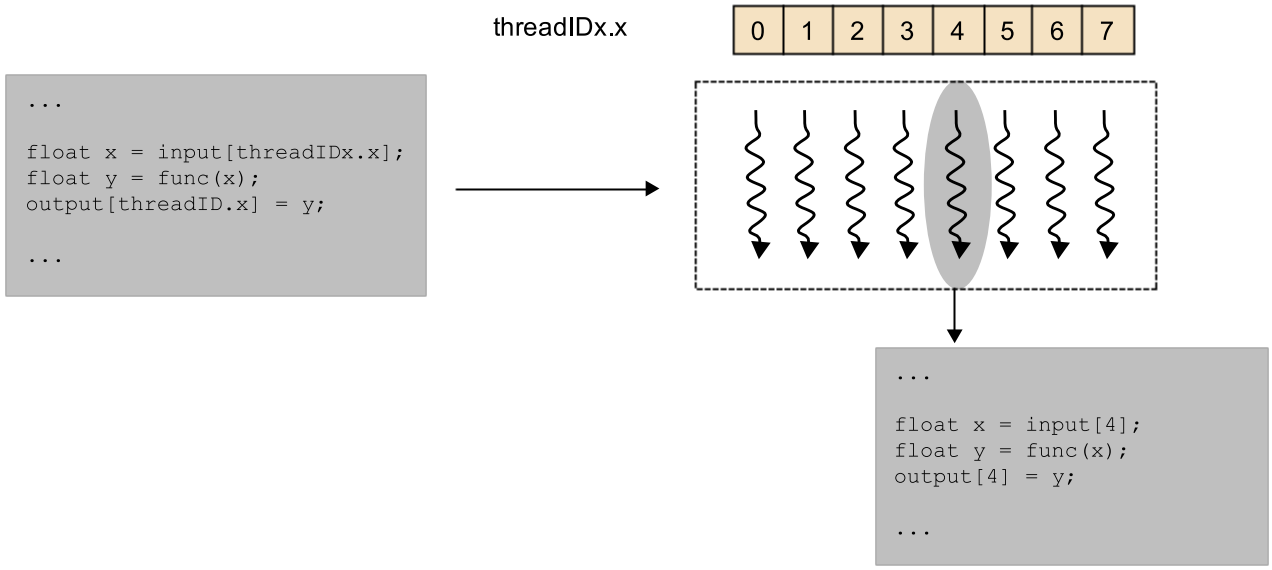
En *CUDA*, un *kernel* se ejecuta mediante un conjunto de flujos (por ejemplo, un vector o una matriz de flujos). Puesto que todos los flujos ejecutan el mismo *kernel* (modelo *SIMT*[10]), se necesita un mecanismo que permita diferenciarlos y, de este modo, asignar la parte correspondiente de los datos a cada flujo de ejecución. *CUDA* incorpora palabras clave para hacer referencia al índice de un flujo (por ejemplo, `threadIdx.x` y `threadIDx.y` si tenemos en cuenta dos dimensiones).

[10]Del inglés *single instruction multiple threads.*

La figura 17 muestra cómo el *kernel* hace referencia al identificador de flujo y que, durante su ejecución, en cada uno de los flujos el identificador se sustituye por el valor que le corresponde. Por lo tanto, las variables `threadIdx.x` y `threadIdx.y` tendrán diferentes valores para cada uno de los flujos de ejecu-

ción. Observad que las coordenadas reflejan la organización multidimensional de los flujos de ejecución, pese a que el ejemplo de la figura 17 solo hace referencia a una dimensión (`threadIdx.x`).

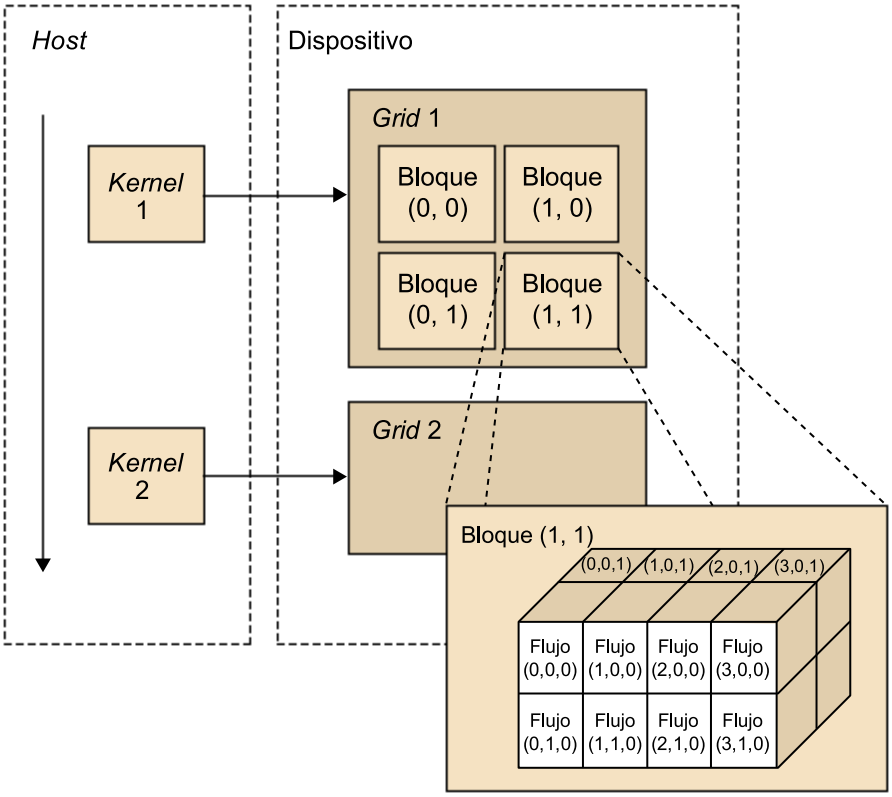
Figura 17. Ejecución de un *kernelCUDA* en un vector de flujos



Normalmente, los *grid* que se utilizan en *CUDA* están formados por muchos flujos (del orden de miles o incluso de millones de flujos). Los flujos de un *grid* están organizados en una jerarquía de dos niveles, como se puede ver en la figura 18. También se puede observar que el `kernel 1` crea el `Grid 1` para su ejecución. El nivel superior de un *grid* consiste en uno o más bloques de flujos. Todos los bloques de un *grid* tienen el mismo número de flujos y tienen que estar organizados del mismo modo. En la figura 18, el `Grid 1` está compuesto de 4 bloques y se organiza como una matriz de 2 × 2 bloques.

Cada bloque de un *grid* tiene una coordenada única en un espacio de dos dimensiones mediante las palabras clave `blockIdx.x` y `blockIdx.y`. Los bloques se organizan en un espacio de tres dimensiones con un máximo de 512 flujos de ejecución. Las coordenadas de los flujos de ejecución en un bloque se identifican con tres índices (`threadIdx.x`, `threadIdx.y` y `threadIdx.z`), a pesar de que no todas las aplicaciones necesitan utilizar las tres dimensiones de cada bloque de flujos. En la figura 18, cada bloque está organizado en un espacio de 4 × 2 × 2 flujos de ejecución.

Figura 18. Ejemplo de organización de los flujos de un *grid* en *CUDA*



Cuando el *host* hace una llamada a un *kernel*, se tienen que especificar las dimensiones del *grid* y de los bloques de flujos mediante parámetros de configuración. El primer parámetro especifica las dimensiones del *grid* en términos de número de bloques, y el segundo especifica las dimensiones de cada bloque en términos de número de flujos. Los dos parámetros son de tipo *dim3*, que esencialmente es una estructura de *C* con tres campos (*x*, *y*, *z*) de tipo entero sin signo. Puesto que los *grids* son grupos de bloques en dos dimensiones, el tercer campo de parámetros de configuración del *grid* se ignora (cualquier valor será válido). A continuación, tenéis un ejemplo en el que dos variables de estructuras de tipo *dim3* definen el *grid* y los bloques del ejemplo de la figura 18.

```
// Configuración de las dimensiones de grid y bloques
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);

// Invocación del kernel (suma de matrices)
matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
```

CUDA también ofrece un mecanismo para sincronizar los flujos de un mismo bloque mediante la función de tipo barrera `__syncthreads()`. Cuando se llama a la función `__syncthreads()`, el flujo que la ejecuta quedará bloqueado hasta que todos los flujos de su bloque lleguen a este mismo punto. Esto sirve para asegurar que todos los flujos de un bloque han completado una fase antes de pasar a la siguiente.

2.2. OpenCL

OpenCL es una interfaz estándar, abierta, libre y multiplataforma para la programación paralela. La principal motivación para el desarrollo de OpenCL fue la necesidad de simplificar la tarea de programación portable y eficiente de la creciente cantidad de plataformas heterogéneas, como por ejemplo *CPU* multinúcleo, *GPU* o incluso sistemas empotrados. OpenCL fue concebida por Apple a pesar de que la acabó desarrollando el grupo Khronos, que es el mismo que impulsó y es responsable de OpenGL.

OpenCL consiste en tres partes: la especificación de un lenguaje multiplataforma, una interfaz en un ámbito de entorno de computación y una interfaz para coordinar la computación paralela entre procesadores heterogéneos. OpenCL utiliza un subconjunto de C99 con extensiones para el paralelismo y utiliza el estándar de representación numérica IEEE 754 para garantizar la interoperabilidad entre plataformas.

Hay muchas similitudes entre OpenCL y *CUDA*, pese a que OpenCL tiene un modelo de gestión de recursos más complejo, puesto que soporta múltiples plataformas y portabilidad entre diferentes fabricantes. OpenCL soporta modelos de paralelismo tanto en un ámbito de datos como de tareas. En este subapartado nos centraremos en el modelo de paralelismo en un ámbito de datos, que es equivalente al de *CUDA*.

2.2.1. Modelo de paralelismo en un ámbito de datos

Del mismo modo que en *CUDA*, un programa en OpenCL está formado por dos partes: los *kernels* que se ejecutan en uno o más dispositivos y un programa *host* que invoca y controla la ejecución de los *kernels*. Cuando se lleva a cabo la invocación de un *kernel*, el código se ejecuta en trabajos elementales[11] que corresponden a los flujos de *CUDA*. Los trabajos elementales y los datos asociados a cada trabajo elemental se definen a partir del rango de un espacio de índices *N*-dimensional[12]. Los trabajos elementales forman grupos de trabajos[13] que corresponden a bloques de *CUDA*. Los trabajos elementales tienen un identificador global que es único. Además, los grupos de trabajos elementales se identifican dentro del rango *N*-dimensional y, para cada grupo, cada uno de los trabajos elementales tiene un identificador local que irá desde 0 hasta tamaño del grupo-1. Por lo tanto, la combinación del identificador del grupo y del identificador local dentro del grupo también identifica de manera única un trabajo elemental. La tabla 3 resume algunas de las equivalencias entre OpenCL y *CUDA*.

(11) En inglés, *work items.*

(12) En inglés, *NDRanges.*

(13) En inglés, *work groups.*

Tabla 3. Algunas correspondencias entre OpenCL y CUDA

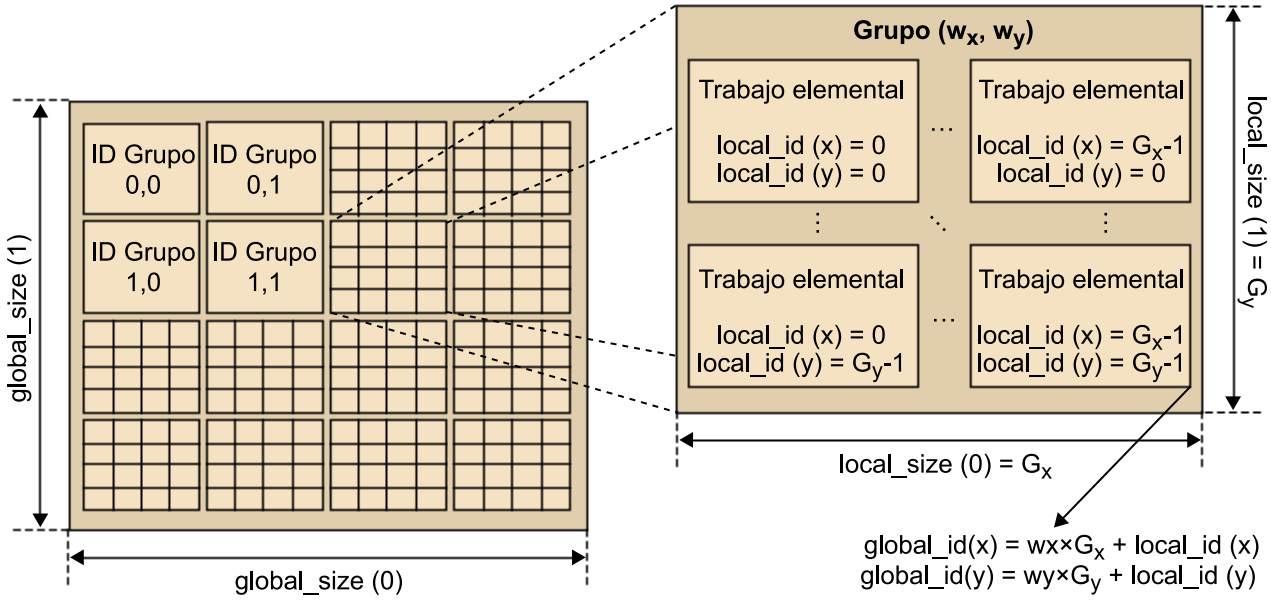
OpenCL	CUDA
Kernel	*Kernel*

OpenCL	CUDA
Programa *host*	Programa *host*
NDRange (rang N-dimensional)	*Grid*
Trabajo elemental (*work item*)	Flujo
Grupo de trabajos (*work group*)	Bloque
`get_global_id(0);`	`blockIdx.x * blockDim.x + threadIdx.x`
`get_local_id(0);`	`threadIdx.x`
`get_global_size(0);`	`gridDim.x*blockDim.x`
`get_local_size(0);`	`blockDim.x`

La figura 19 muestra el modelo de paralelismo en un ámbito de datos de OpenCL. El rango *N*-dimensional (equivalente al *grid* en *CUDA*) contiene los trabajos elementales. En el ejemplo de la figura, el *kernel* utiliza un rango de dos dimensiones. Mientras que en *CUDA* cada flujo tiene unos valores de `blockIdx` y `threadIdx` que se combinan para obtener el identificador del flujo, en OpenCL disponemos de interfaces para identificar los trabajos elementales de las dos maneras que hemos visto. Por un lado, la función `get_global_id()`, dada una dimensión, retorna el identificador único de trabajo elemental en la dimensión especificada. En el ejemplo de la figura, las llamadas `get_global_id(0)` y `get_global_id(1)` retornan el índice de los trabajos elementales en las dimensiones *X* e *Y*, respectivamente. Por otro lado, la función `get_local_id()`, dada una dimensión, retorna el identificador del trabajo elemental dentro de su grupo en la dimensión especificada. Por ejemplo, `get_local_id(0)` es equivalente a `threadIdx.x` en *CUDA*.

OpenCL también dispone de las funciones `get_global_size()` y `get_local_size()` que, dada una dimensión, retornan la cantidad total de trabajos elementales y la cantidad de trabajos elementales dentro de un grupo en la dimensión especificada, respectivamente. Por ejemplo, `get_global_size(0)` retorna la cantidad de trabajos elementales que es equivalente a `gridDim.x*blockDim.x` en *CUDA*.

Figura 19. Ejemplo de rango *N*-dimensional

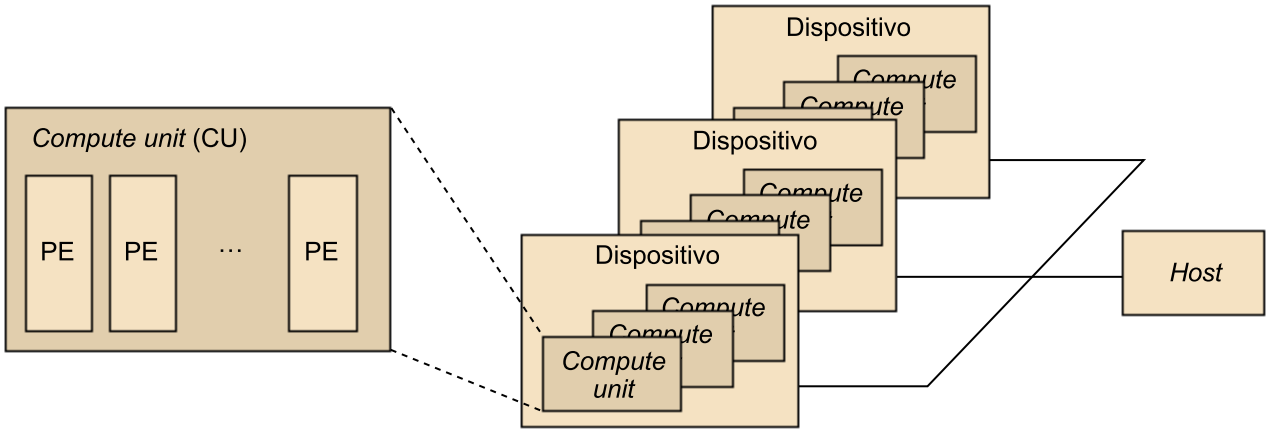


Los trabajos elementales dentro del mismo grupo se pueden sincronizar entre sí utilizando barreras que son equivalentes a `__syncthreads()` de *CUDA*. En cambio, los trabajos elementales de diferentes grupos no se pueden sincronizar entre sí, excepto si es para la terminación del *kernel* o la invocación de uno nuevo.

2.2.2. Arquitectura conceptual

La figura 20 muestra la arquitectura conceptual de OpenCL, la cual está formada por *un host* (de manera típica, una *CPU* que ejecuta el programa *host*) conectado a uno o más dispositivos OpenCL. Un dispositivo OpenCL está compuesto por una o más *compute units* (*CU*) que corresponden a los *SM* de *CUDA*. Finalmente, un *CU* está formado por uno o más *processing elements* (*PE*) que corresponden a los *SP* de *CUDA*. El programa se acabará ejecutando en los *PE*.

Figura 20. Arquitectura conceptual de OpenCL



2.2.3. Modelo de memoria

En un dispositivo OpenCL hay disponible una jerarquía de memoria que incluye varios tipos de memoria diferentes, tal y como muestra la figura 21: global, constante, local y privada.

- La memoria global es la que pueden utilizar todas las unidades de cálculo de un dispositivo.

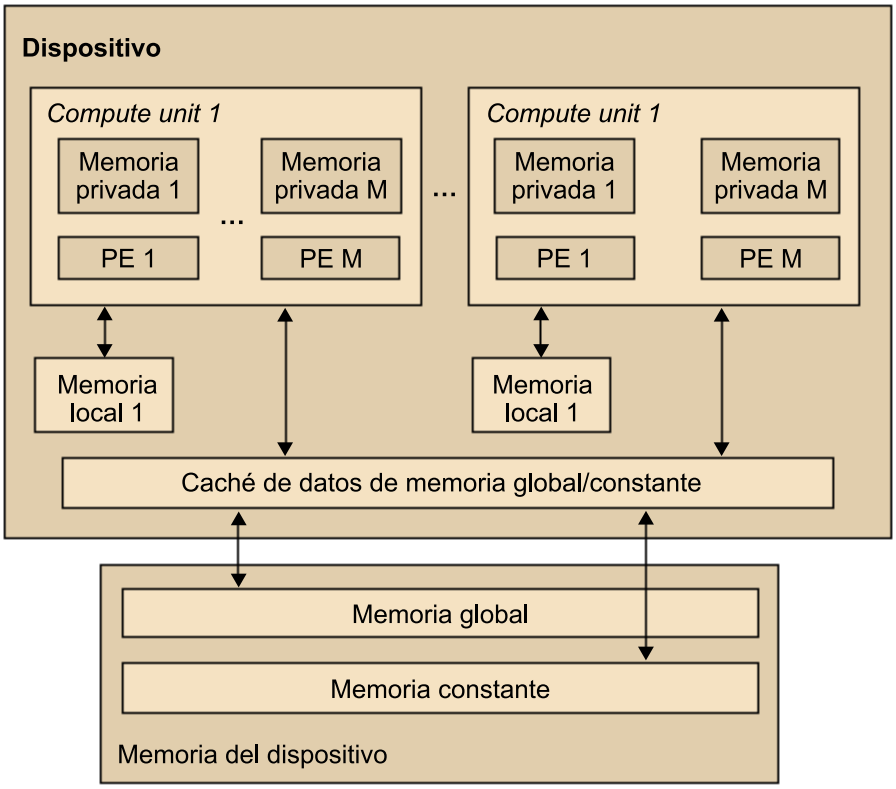
- La memoria constante es la memoria que todas las unidades de cálculo de un dispositivo pueden utilizar para almacenar datos constantes para acceso solo de lectura durante la ejecución de un *kernel*. El procesador *host* es responsable de asignar e iniciar los objetos de memoria que residen en el espacio de memoria.

- La memoria local es la memoria que pueden utilizar los trabajos elementales de un grupo.

- La memoria privada es la que solo puede utilizar una unidad de cálculo. Esto es similar a los registros en una sola unidad de cálculo o un solo núcleo de una *CPU*.

Así pues, tanto la memoria global como la constante corresponden a los tipos de memoria con el mismo nombre de *CUDA*, la memoria local corresponde a la memoria compartida de *CUDA* y la memoria privada, a la memoria local de *CUDA*.

Figura 21. Arquitectura y jerarquía de memoria de un dispositivo OpenCL



Para crear y administrar objetos en la memoria de un dispositivo, la aplicación que se ejecuta en *el host* utiliza la interfaz de OpenCL, puesto que los espacios de memoria del *host* y de los dispositivos son principalmente independientes uno de otro. La interacción entre el espacio de memoria del *host* y de los dispositivos puede ser de dos tipos: copiando datos de manera explícita o bien mapeando/desmapeando regiones de un objeto OpenCL en memoria. Para copiar datos explícitamente, el *host* pone en cola comandos para transferir datos entre la memoria del objeto y la memoria del *host*. Estos comandos de transferencia pueden ser o bien bloqueantes o bien no bloqueantes. Cuando se utiliza una llamada bloqueante, se puede acceder a los datos desde el *host* con seguridad una vez la llamada ha finalizado. En cambio, para hacer una transferencia no bloqueante, la llamada a la función de OpenCL finaliza de manera inmediata y se desencola pese a que no es seguro utilizar la memoria desde el *host*. La interacción mapeando/desmapeando objetos en memoria permite que el *host* pueda tener en su espacio de memoria una región correspondiente a objetos OpenCL. Los comandos de mapeo/desmapeo también pueden ser bloqueantes o no bloqueantes.

2.2.4. Gestión de *kernels* y de dispositivos

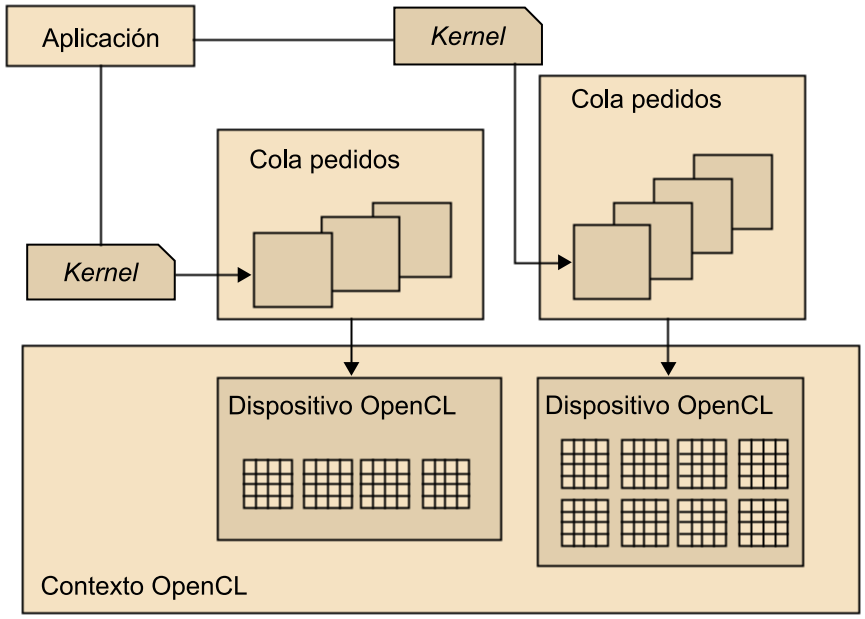
Los *kernels* de OpenCL tienen la misma estructura que los de *CUDA*. Por lo tanto, son funciones que se declaran empezando con la palabra clave `__kernel`, que equivale a `__global` de *CUDA*. A continuación, se muestra la implementación de nuestro ejemplo de suma de matrices en OpenCL. Observad que los argumentos del *kernel* correspondientes a las matrices están declarados como

`__global`, puesto que se encuentran en la memoria global, y las dos matrices de entrada están también declaradas como `const`, ya que solo será necesario hacer accesos en modalidad de lectura. En la implementación del cuerpo del *kernel*, se utiliza `get_global_id()` en las dos dimensiones de la matriz para definir el índice asociado. Este índice se utiliza para seleccionar los datos que corresponden a cada uno de los trabajos elementales que instancie el *kernel*.

```
__kernel void matrix_add_opencl (   __global const float *A,
            __global const float *B,
            __global float *C,
            int N) {
   int i = get_global_id(0);
   int j = get_global_id(1);
   int index = i + j*N;
   if (i<N && j<N){
         C[index] = A[index] + B[index];
   }
}
```

El modelo de gestión de dispositivos de OpenCL es mucho más sofisticado que el de *CUDA*, puesto que OpenCL permite abstraer diferentes plataformas *hardware*. Los dispositivos se gestionan mediante contextos. Como muestra la figura 22, para gestionar uno o más dispositivos (2 en el ejemplo de la figura) primero hay que crear un contexto que contenga los dispositivos, bien mediante la función `clCreateContext()` o bien por medio de la función `clCreateContextFromType()`. Normalmente, hay que indicar a las funciones `CreateContext` la cantidad y el tipo de dispositivos del sistema mediante la función `clGetDeviceIDs()`. Para ejecutar cualquier tarea en un dispositivo, primero hay que crear una cola de comandos para el dispositivo mediante la función `clCreateCommandQueue()`. Una vez se ha creado una cola para el dispositivo, el código que se ejecuta en el *host* puede insertar un *kernel* y los parámetros de configuración asociados. Cuando el dispositivo esté disponible para ejecutar el siguiente *kernel* de la cola, este *kernel* se elimina de la cola y pasa a ser ejecutado.

Figura 22. Gestión de dispositivos en OpenCL mediante contextos



A continuación, se muestra el código correspondiente al *host* para ejecutar la suma de matrices mediante el *kernel* del ejemplo de código anterior. Supongamos que la definición y la inicialización de variables se han llevado a cabo correctamente y que la función del *kernel*`matrix_add_opencl()` está disponible. En el primer paso se crea un contexto y, una vez se han obtenido los dispositivos disponibles, se crea una cola de comandos que utilizará el primer dispositivo disponible de los presentes en el sistema. En el segundo paso, se definen los objetos que necesitaremos en memoria (las tres matrices *A*, *B* y *C*). Las matrices *A* y *B* se definen de lectura, y la matriz *C* se define de escritura. Observad que en la definición de las matrices *A* y *B* se utiliza la opción `CL_MEM_COPY_HOST_PTR`, la cual indica que los datos de las matrices *A* y *B* se copiarán de los punteros especificados (`srcA` y `srcB`). En el tercer paso se define el *kernel*, que se ejecutará posteriormente mediante `clCreateKernel`, y se especifican también los argumentos de la función `matrix_add_opencl`. A continuación, se encola el *kernel* en la cola de comandos previamente definida y, por último, se leen los resultados del espacio de memoria correspondiente a la matriz C a través del puntero `dstC`. En este ejemplo, no hemos entrado en detalle en muchos de los parámetros de las diferentes funciones de la interfaz de OpenCL. Por lo tanto, os recomendamos que repaséis la especificación de los mismos que está disponible en la página web http://www.khronos.org/opencl/.

```
main(){
    // Inicialización de variables, etc.
    (...)

    // 1. Creación del contexto y cola en el dispositivo
    cl_context context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
    // Para obtener la lista de dispositivos GPU asociados al contexto
    size_t cb;
```

```
clGetContextInfo( context,   CL_CONTEXT_DEVICES,   0,   NULL, &cb);
cl_device_id *devices = malloc(cb);
clGetContextInfo( context,   CL_CONTEXT_DEVICES,   cb,   devices,   NULL);
cl_cmd_queue cmd_queue = clCreateCommandQueue(context, devices[0], 0 , NULL);

// 2. Definición de los objetos en memoria (matrices A, B y C)
cl_mem memobjs[3];
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
         sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
         sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float)*n, NULL, NULL);

// 3. Definición del kernel y argumentos
cl_program program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);
cl_int err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "matrix_add_opencl", NULL);
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&memobjs[2]);
err |= clSetKernelArg(kernel, 3, sizeof(int), (void *)&N);

// 4. Invocación del kernel
size_t global_work_size[1] = n;
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size,
   NULL, 0, NULL, NULL);

// 5. Lectura de los resultados (matriz C)
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE, 0, n*sizeof(cl_float),
   dstC, 0, NULL, NULL);
(...)
```

3. Modelos de programación para memoria distribuida

En este apartado, nos centraremos en los principales modelos de programación para memoria distribuida (*MPI*), que es el estándar *de facto* desde hace años, y en los modelos *PGAS*, que intentan dar solución a retos importantes que presentan las características de los futuros sistemas de altas prestaciones.

3.1. *MPI*

MPI[14] es una biblioteca de paso de mensajes que se puede utilizar tanto en programas C como Fortran, y desde los cuales se hacen llamadas a funciones de *MPI* para la gestión y comunicación de procesos.

(14)Del inglés *message passing interface.*

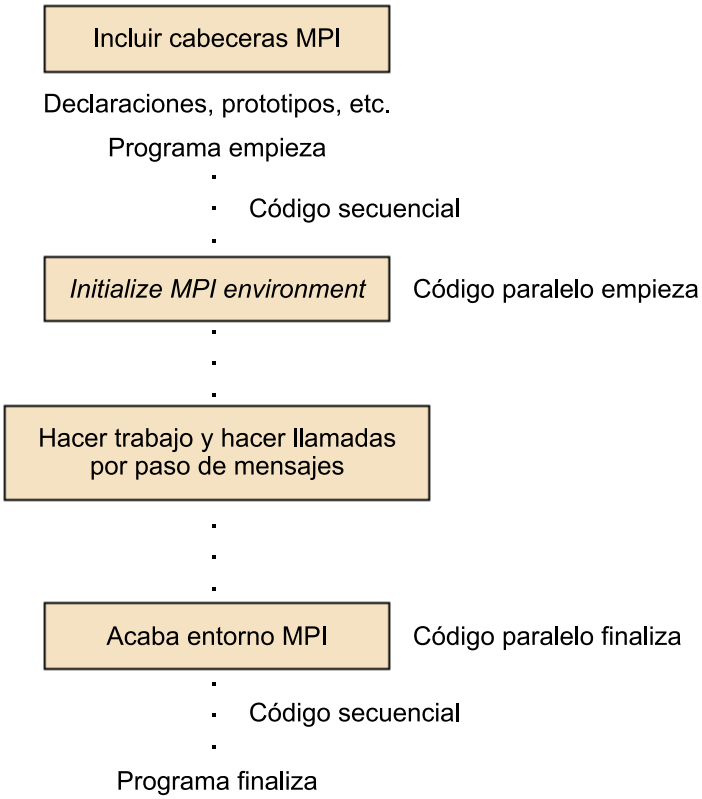
Ved también

La biblioteca *MPI* se ha tratado en el módulo "Introducción a la computación de altas prestaciones" de esta asignatura.

La figura 23 muestra la estructura general de un programa MPI, en el que se puede apreciar cómo inicialmente hay un solo proceso que ejecuta el código de manera secuencial hasta que se inicializa el entorno *MPI* (momento en el que el código paralelo empieza). En la sección de código paralelo, se llevará a cabo la tarea concreta y se utilizarán mensajes para comunicar los diferentes procesos *MPI*. Finalmente, una vez acabado el trabajo que se quiere ejecutar en paralelo, se indica la finalización del código paralelo y se vuelve a ejecutar el resto del código de manera secuencial hasta que acaba la ejecución del programa.

Hay que tener presente que, a diferencia de OpenMP, el número de procesos no se puede establecer en tiempo de ejecución, sino que se especifica cuando se quiere ejecutar el programa, tal y como veremos más adelante. También hay que recordar que todos los procesos *MPI* ejecutarán exactamente el mismo código.

Figura 23. Estructura general de un programa *MPI*



A continuación se muestra el ejemplo básico de programa *MPI* (variante del típico "*Hello world*"), en el que se puede observar la estructura general de un programa *MPI*. Esta es una versión similar a la vista en el primer módulo de la asignatura, pero ejemplificando su utilización en Fortran.

```
    program hello

    use fmpi
!   include "mpif.h"
    call MPI_INIT( ierr )
    call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
    call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

    write (*,*) "Hello from ",myid
    write (*,*) "Numprocs is ",numprocs

    call MPI_FINALIZE(ierr)
    stop
    end
```

El modelo *SPMD* es una de las maneras más habituales de utilizar *MPI*. En este caso, será necesario diferenciar el código que ejecuta el proceso maestro (típicamente el proceso con rango igual a cero) del que ejecutan los procesos esclavos, tal y como se muestra a continuación.

```
main (int argc, char *argv[])
{
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // Consulta el rango del proceso

   if (myrank == 0)
      master(); // código que ejecuta el proceso maestro
   else
      slave();  // código que ejecutan los procesos esclavos

   MPI_Finalize();
}
```

3.1.1. Comunicadores

Un comunicador permite definir una serie de procesos entre los que se puede llevar a cabo comunicaciones. Cada proceso puede estar en varios comunicadores y tendrá un identificador en cada uno de los mismos (también denominado rango). Los identificadores estarán entre 0 y el número de procesos en el comunicador menos 1. Al ponerse en marcha *MPI*, todos los procesos están en `MPI_COMM_WORLD`, que es una constante que identifica un comunicador que incluye en todos los procesos.

Aparte de `MPI_COMM_WORLD` también se pueden definir comunicadores con un número inferior de procesos, por ejemplo para comunicar los datos en cada una de las filas de procesos en una parrilla, con lo que se puede hacer *broadcast* en las diferentes filas de procesos sin que las comunicaciones de unos afecten a los otros. Aun así, también se pueden usar en las comunicaciones punto a punto.

Hay dos tipos de comunicadores: los intracomunicadores y los intercomunicadores. Los intracomunicadores se utilizan para enviar mensajes entre los procesos en este comunicador, y los intercomunicadores, para enviar mensajes entre procesos en diferentes comunicadores. Nosotros nos centramos en comunicaciones entre procesos de un mismo comunicador. Las comunicaciones entre procesos en comunicadores diferentes pueden tener sentido si al diseñar librerías se crean comunicadores y un usuario de la librería quiere comunicar un proceso de su programa con otro de la biblioteca *MPI*.

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

Actividad

Explorad cómo se pueden definir topologías virtuales de comunicadores y cuál puede ser su utilidad.

En *MPI* hay dos tipos básicos de comunicaciones: las comunicaciones punto a punto, que involucran dos procesos que intercambian mensajes solo entre sí, y las comunicaciones colectivas, que involucran un conjunto de procesos que llevan a cabo una tarea concreta de manera colectiva.

3.1.2. Comunicaciones punto a punto

En el ejemplo anterior, los procesos no interaccionan entre sí. Lo normal es que los procesos no trabajen de manera independiente, sino que intercambien información mediante el paso de mensajes. Para enviar mensajes entre dos procesos (uno de origen y uno de destino), se utilizan las comunicaciones punto a punto. Más adelante, se muestra un fragmento de código *MPI* en el que el proceso con rango 0 le envía el entero *x* al proceso con rango 1. El programa muestra la forma normal de trabajar con paso de mensajes. Se ejecuta el mismo programa en todos los procesos, pero procesos diferentes ejecutan distintas partes del código. En el ejemplo se utilizan las funciones `MPI_Send` y `MPI_Recv` para enviar y recibir mensajes, de manera respectiva. Estas dos funciones son bloqueantes y su definición se muestra a continuación.

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

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

Los parámetros se describen a continuación:

- `buf` contiene el inicio de la zona de memoria de la que se tomarán los datos que hay que enviar o bien donde se almacenarán los datos que se reciben.

- `count` indica el número de datos que hay que enviar o el espacio disponible para recibir (no el número de datos del mensaje que se recibe, puesto que el tamaño lo determina quien envía).

- `datatype` es el tipo de datos que hay que transferir y debe ser un tipo `MPI (MPI_Datatype)`, que en nuestro ejemplo es `MPI_CHAR`.

- `dest` y `source` son los identificadores del proceso al que se envía el mensaje y del proceso del que se recibe, de manera respectiva. Se puede utilizar

la constante `MPI_ANY_SOURCE` para indicar que es posible recibirlo desde cualquier proceso.

- `tag` se utiliza para diferenciar entre mensajes, y su valor debe coincidir en el proceso que envía y el que recibe. Se puede utilizar `MPI_ANY_TAG` para indicar que el mensaje es compatible con mensajes con cualquier identificador.

- `comm` es el comunicador dentro del que se hace la comunicación. Es del tipo `MPI_Comm`, y en el ejemplo se utiliza el identificador de comunicador formado por todos los procesos (`MPI_COMM_WORLD`).

- `status` referencia una variable de tipo `MPI_Status`. En el programa no se utiliza, pero contiene información del mensaje que se ha recibido y se puede consultar para identificar alguna característica del mensaje. Por ejemplo, su longitud, el proceso de origen, etc.

A continuación, se muestra un fragmento de código *MPI* en el que el proceso con rango 0 le envía el entero *x* al proceso con rango 1:

```
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */
    if (myrank == 0) {
          int x;
          MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
    }
    else if (myrank == 1) {
          int x;
          MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD,status);
}
```

En los mensajes, hay que utilizar tipos de datos *MPI*. Por razones de portabilidad, *MPI* redefine sus tipos de datos elementales. La tabla siguiente muestra los tipos estándar para C y Fortan:

Tabla 4. Tipos de datos MPI para C y Fortran

Tipos de datos C		**Tipos de datos Fortran**	
MPI_CHAR	signed char	MPI_CHARACTER	character(1)
MPI_WCHAR	wchar_t wide character		
MPI_SHORT	signed short int		
MPI_INT	signed int	MPI_INTEGER MPI_INTEGER1 MPI_INTEGER2 MPI_INTERGER4	integer integer*1 integer*2 integer*4
MPI_LONG	signed		
MPI_LONG_LONG_INT MPI_LONG_LONG	signed long long int		

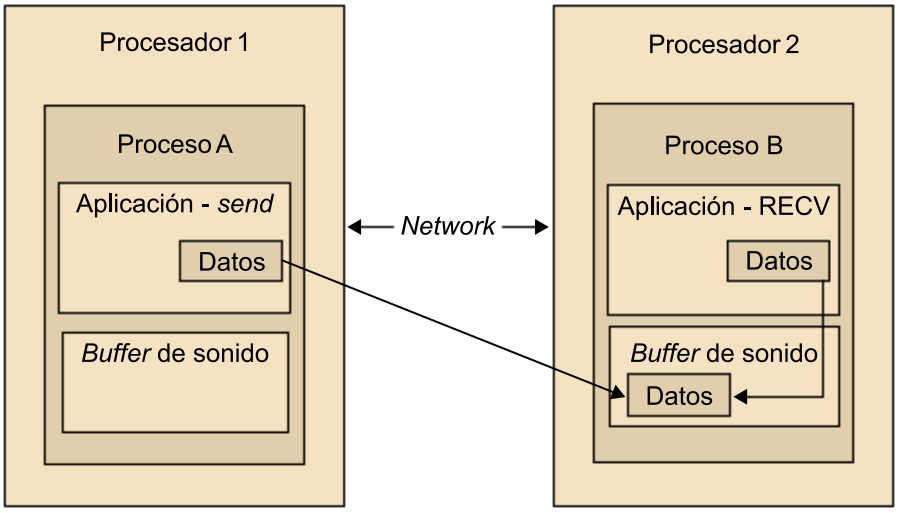
Tipos de datos C		Tipos de datos Fortran	
MPI_SIGNED_CHAR	Signed char		
MPI_UNSIGNED_CHAR	Unsigned char		
MPI_UNSIGNED_SHORT	Unsigned short int		
MPI_UNISGNED	Unsigned int		
MPI_SIGNED_CHAR	Signed char		
MPI_UNSIGNED_CHAR	Unsigned char		
MPI_FLOAT	Float	MPI_REAL MPI_REAL2 MPI_REAL4 MPI_REAL8	Real Real*2 Real*4 Real*8
MPI_DOUBLE	Double	MPI_DOUBLE_PRECISION	Double precision
MPI_LONG_DOUBLE	Long double		
MPI_C_COMPLEX MPI_C_FLOAT_COMPLEX	Float_Complex	MPI_COMPLEX	Complex
MPI_C_DOUBLE COMPLEX	Double_Complex	MPI_DOUBLE_COMPLEX	Double complex
MPI_C_LONG_DOUBLE_COMPLEX	Long double_Complex		
MPI_C_BOOL	_Bool	MPI_LOGICAL	Logical
MPI_C_LONG_DOUBLE_COMPLEX	Long double_Complex		
MPI_INT8_T MPI_INT16_T MPI_INT32_T MPI_INT64_T	Int_8_t Int_16_t Int_32_t Int_64_t		
MPI_UINT8_T MPI_UINT16_T MPI_UINT32_T MPI_UINT64_T	Uint_8_t uint_16_t uint_32_t uint_64_t		
MPI_BYTE	8 binary digits	MPI_BYTE	8 binary digits
MPI_PACKED	Data packed or unpacked with MPI_PACK()/ MPI_UNPACK	MPI_PACKED	Data packed or unpacked with MPI_PACK()/ MPI_UNPACK

La comunicación que hemos visto anteriormente mediante `MPI_Send` y `MPI_Recv` es de tipo bloqueante, puesto que se lleva a cabo una sincronización entre el proceso que envía el mensaje y el que lo recibe. Con `MPI_Send` y `MPI_Recv`, el proceso que envía el mensaje sigue ejecutándose una vez acabado el envío. Si el proceso que recibe el mensaje solicita su recepción cuando aún no ha llegado, se queda bloqueado hasta que el mensaje llegue.

MPI proporciona otras posibilidades para envíos bloqueantes, por ejemplo `MPI_Ssend`, `MPI_Bsend` y `MPI_Rsend`. Estas funciones se diferencian en la forma en que se gestiona el envío, y es posible gestionar el *buffer* de comunica-

ción o determinar que el proceso tome los datos directamente de la memoria. En la figura 24, se ilustra el funcionamiento de paso de mensajes mediante *buffer*.

Figura 24. Funcionamiento de paso de mensaje en *MPI* mediante *buffer*



La función `MPI_Sendrecv` combina en una llamada el envío y la recepción entre dos procesos, pero continúa siendo una función bloqueante.

MPI proporciona también comunicación no bloqueante, en la que el proceso receptor solicita el mensaje y si este no ha llegado, sigue su ejecución. Las funciones son `MPI_Isend` y `MPI_Irecv`. Si el proceso receptor no ha recibido el mensaje, puede llegar un momento en el que obligatoriamente tenga que esperarlo para seguir la ejecución. La definición de estas dos funciones se muestra a continuación.

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

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

También se dispone de la función `MPI_Wait()` para esperar la llegada de un mensaje y de `MPI_Test()` para comprobar si la operación se ha completado.

A continuación se muestra el mismo ejemplo del desarrollo del fragmento de código *MPI*, en el que el proceso con rango 0 le envía el entero *x* al proceso con rango 1, pero ahora utilizando llamadas asíncronas a cambio de llamadas síncronas o bloqueantes.

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
    if (myrank == 0) {
       int x;
```

```
        MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
        compute(); // Efectúa algún cálculo mientras se hace el envío
        MPI_Wait(req1, status);
    }
    else if (myrank == 1) {
        int x;
        MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

3.1.3. Comunicaciones colectivas

Además de las comunicaciones punto a punto, *MPI* ofrece una serie de funciones para llevar a cabo comunicaciones en las que intervienen todos los procesos de un comunicador. Siempre que sea posible llevar a cabo las comunicaciones por medio de estas comunicaciones colectivas, es conveniente hacerlo de este modo puesto que facilita la programación evitando posibles errores. Si las funciones están optimizadas para el sistema en el que estamos trabajando, es posible que se desarrollen programas más eficientes con comunicaciones colectivas que con comunicaciones punto a punto.

Algunas de las comunicaciones colectivas más útiles se presentan a continuación.

a) Barrera

```
int MPI_Barrier(MPI_Comm comm)
```

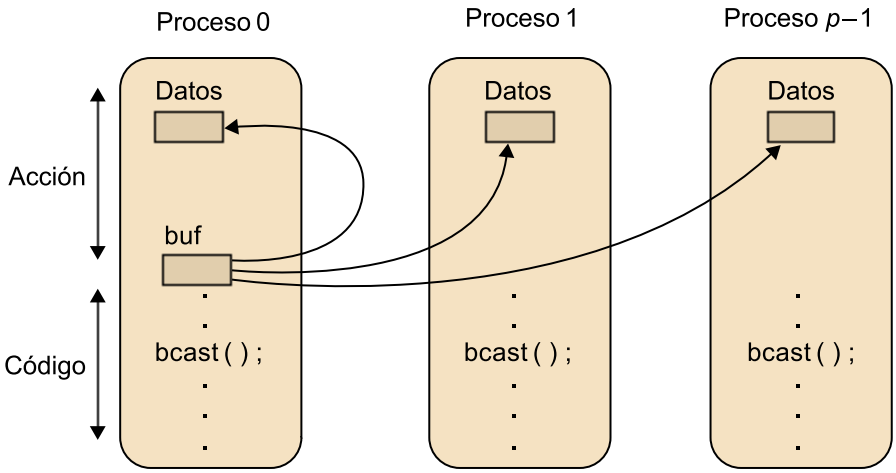
Establece una barrera. Todos los procesos esperan a llegar a la barrera, para continuar la ejecución una vez han llegado. Se utiliza un comunicador que establece el grupo de procesos que se está sincronizando. Todas las funciones de comunicaciones colectivas retornan un código de error.

b) *Broadcast*

```
int MPI_Bcast(void *buffer, int count,
      MPI_Datatype datatype, int root, MPI_comm comm)
```

Lleva a cabo una operación de *broadcast* (comunicación uno a todos), en la que se envían `count` datos del tipo `datatype` desde el proceso raíz (`root`) al resto de los procesos en el comunicador. Todos los procesos que intervienen deben llamar a la función indicando el proceso que actúa como raíz. En la raíz, los datos que se envían se toman de la zona apuntada por *buffer*, y los que reciben almacenan en la memoria reservada en *buffer*. La figura 25 ilustra el funcionamiento de esta operación.

Figura 25. Funcionamiento de la operación *broadcast* de *MPI*

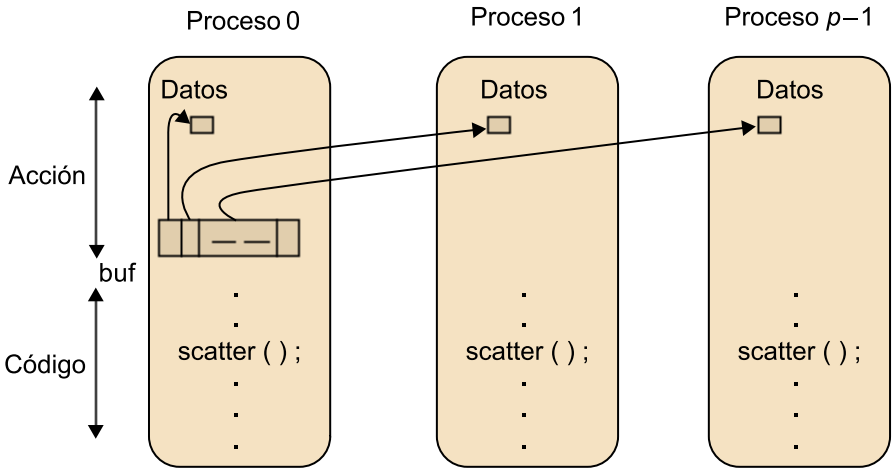


c) *Scatter*

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

Se puede utilizar `MPI_Scatter` para enviar desde un proceso mensajes diferentes al resto de los procesos. En el proceso raíz, el mensaje se divide en segmentos de tamaño `sendcount` y el segmento *i*-ésimo envía el proceso `i`. Si se quiere enviar bloques de tamaños diferentes a los diferentes procesos, o si los bloques que hay que enviar no están seguidos en memoria, se puede utilizar la función `MPI_Scatterv`. La figura 26 ilustra el funcionamiento de `MPI_Scatter`.

Figura 26. Funcionamiento de la operación *scatter* de *MPI*



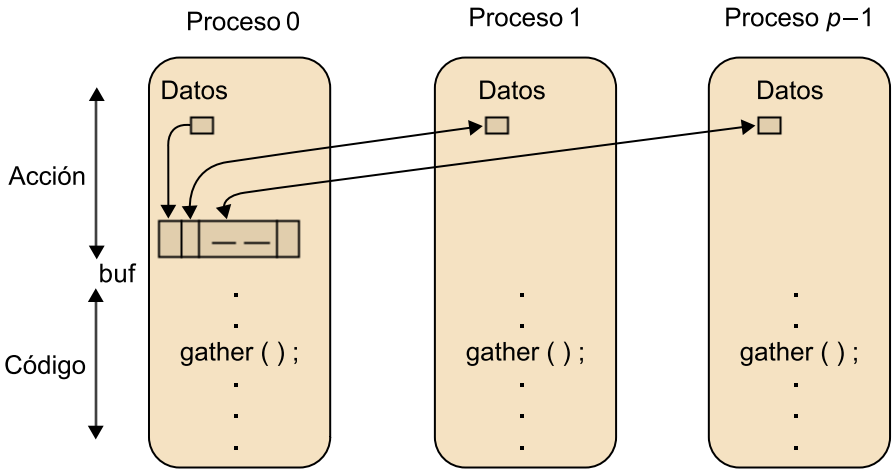
d) *Gather*

```
int MPI_Gather(void *sendbuf, int sendcount,
            MPI_Datatype sendtype, void *recvbuf,
```

```
            int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm)
```

Esta es la función inversa de `MPI_Scatter`. Todos los procesos (incluida la raíz) envían al proceso raíz `sendcount` datos desde `sendbuf`, y la raíz los almacena en `recvbuf` por el orden de los procesos. Si se quiere que cada proceso envíe bloques de tamaños diferentes, se utilizará `MPI_Gatherv`. La figura 27 ilustra el funcionamiento de `MPI_Gather`.

Figura 27. Funcionamiento de la operación *gather* de *MPI*



Para enviar bloques de datos de todos a todos los procesos, se utiliza:

```
int MPI_Allgather (void *sendbuf, int sendcount,
            MPI_Datatype sendtype, void *recvbuf,
            int recvcount, MPI_Datatype recvtype,
            MPI_Comm comm)
```

Donde el bloque enviado por el *i*-ésimo proceso se almacena como bloque *i*-ésimo en `recvbuf` en todos los procesos. Para enviar bloques de tamaños diferentes, se utiliza `MPI_Allgatherv`.

Para enviar bloques de datos diferentes a los diferentes procesos, se utiliza:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
            MPI_Datatype sendtype, void *recvbuf,
            int recvcount, MPI_Datatype recvtype,
            MPI_Comm comm)
```

Donde de cada proceso *i*, el bloque *j* se envía al proceso *j*, que lo almacena como bloque *i* en `recvbuf`. Encontramos el correspondiente `MPI_Alltoallv`.

e) Reducción

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
```

```
            MPI_Datatype datatype, MPI_Op op,
            int root, MPI_Comm comm)
```

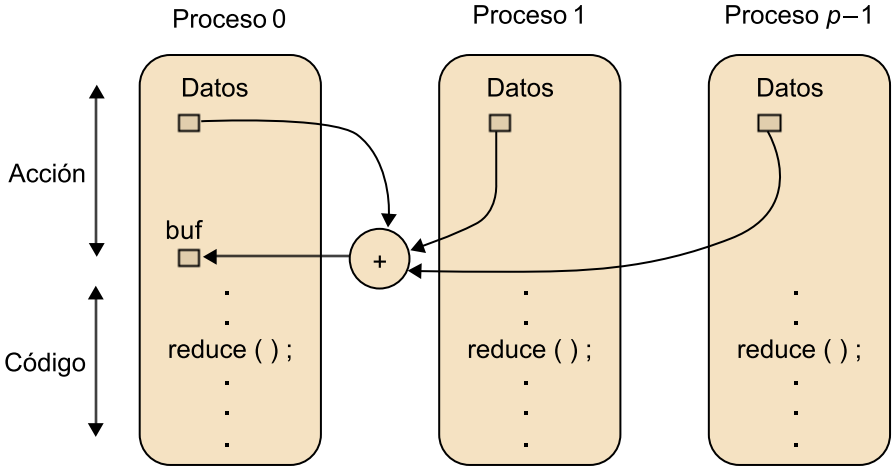
Lleva a cabo una reducción de todos a uno, es decir, se recogen los valores distribuidos en los diferentes procesos y se les aplica una operación a todos. Esto significa que los datos sobre los que hay que hacer la reducción se toman de la zona apuntada por `sendbuf`, y el resultado se deja en la apuntada por `recvbuf`. Si el número de datos (`count`) es mayor que 1, la operación se lleva a cabo elemento a elemento en los del *buffer* por separado. El resultado se deja en el proceso `root`. La operación que se aplica a los datos viene dada por `op`. Las operaciones que se admiten para hacer reducciones se muestran en la tabla siguiente.

Tabla 5. Tipos de datos MPI para C y Fortran

Operación	Significado	TiposC permitidos
MPI_MAX	máximo	Enteros y punto flotante
MPI_MIN	mínimo	Enteros y punto flotante
MPI_SUM	suma	Enteros y punto flotante
MPI_PROD	producto	Enteros y punto flotante
MPI_LAND	AND lógico	Enteros
MPI_LORD	OR lógico	Enteros
MPI_LXORD	XORD lógico	Enteros
MPI_BAND	AND bit a bit	Enteros y bytes
MPI_BOR	OR bit a bit	Enteros y bytes
MPI_BXOR	XOR bit a bit	Enteros y bytes
MPI_MAXLOC	Máximo y localización	Parejas de tipos
MPI_MINLOC	Mínimo y localización	Parejas de tipos

La figura 28 ilustra el funcionamiento de `MPI_Reduce`. A continuación, se muestra un ejemplo en el que esta operación se utiliza para sumar los valores de un intervalo a partir de las sumas parciales de cada uno de los procesos *MPI* y reducción.

Figura 28. Funcionamiento de la operación de reducción de *MPI*




```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
   int MyProc, tag=1, size;
   char msg='A', msg_recpt ;
   MPI_Status *status ;
   int root ;
   int left, right, interval ;
   int number, start, end, sum, GrandTotal;
   int mystart, myend;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   root = 0;
   if (MyProc == root)  /* El proceso raíz lee los límites del intervalo */
   {
      printf("Give the left and right limits of the interval\n");
      scanf("%d %d", &left, &right);
      printf("Proc root reporting : the limits are : %d %d\n", left, right);
   }
   MPI_Bcast(&left, 1, MPI_INT, root, MPI_COMM_WORLD); /*Bcast limites a todos*/
   MPI_Bcast(&right, 1, MPI_INT, root, MPI_COMM_WORLD);
   if (((right - left + 1) % size) != 0)
      interval = (right - left + 1) / size + 1 ;  /*Fija límites locales de suma*/
   else
      interval = (right - left + 1) / size;
   mystart = left + MyProc*interval ;
   myend = mystart + interval ;
   /* establece los límites de los intervalos correctamente */
   if (myend > right) myend = right + 1 ;
   sum = root; /* Suma localmente en cada proceso MPI */
   if (mystart <= right)
```

```
        for (number = mystart; number < myend; number++) sum = sum + number ;
    /* Hace la reducción en el proceso raíz */
    MPI_Reduce(&sum, &GrandTotal, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD) ;
    MPI_Barrier(MPI_COMM_WORLD);
    /* El proceso raíz retorna los resultados */
    if(MyProc == root)
       printf("Proc root reporting : Grand total = %d \n", GrandTotal);
    MPI_Finalize();
}
```

Cuando todos los procesos tienen que recibir el resultado de la operación, se utiliza la función:

```
int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

3.1.4. Compilación y ejecución

En este primer ejemplo, veremos cómo se compila y se ejecuta un código MPI. El modo de hacerlo puede variar de una implementación a otra. Una manera normal de compilar es la siguiente (para *MPICH*, por ejemplo).

```
    mpicc programa.c -o programa
```

Donde `mpicc` llama al compilador de C que esté establecido y lo monta con la librería de *MPI*. Se indica el código que hay que compilar y las opciones que tradicionalmente se pasan al compilador C.

Una vez generado el código, la manera de ejecutarlo también depende de la compilación. Lo más habitual es utilizar `mpirun` pasándole el código que hay que ejecutar y una serie de argumentos que indican los procesos para poner en marcha y la distribución de los procesos en los procesadores. Por ejemplo, si se llama de la manera siguiente:

```
    mpirun -np 4 programa
```

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

Hay varias formas de indicar cómo se asignan los procesos al ejecutar el programa. Por ejemplo, mediante:

```
mpirun -np 4 -machinefile nodes.txt programa
```

se indica que los procesos se asignen según se señala en el fichero `nodos.txt`, el cual indica los servidores que se pueden usar para ejecutar los procesos *MPI*. Se trata de una lista con los nombres de los servidores, como por ejemplo:

```
node1
node2
```

En este ejemplo, y si estamos ejecutando en el `nodo0`, los cuatro procesos se asignarán en el orden siguiente: el proceso 0 en el nodo donde estamos ejecutando el programa (`nodo0`), el proceso 1 en el `nodo1`, el 2 en el `nodo2` y, una vez se haya acabado la lista de máquinas del fichero, se empieza otra vez por el principio, con lo que el proceso 3 se asigna al `nodo1`. El modo de asignación depende de la implementación, y puede que al nodo en el que estamos no se le asigne ningún proceso o que entre en la asignación cíclica una vez acabada la lista de máquinas del fichero. Por ejemplo, con la opción `-nolocal` podemos evitar que se utilice el servidor desde el cual estamos ejecutando el programa (lo que puede ser útil, por ejemplo, en el caso de estar utilizando un nodo *login* que no debería usarse para la computación).

3.2. Lenguajes *PGAS*

La programación de sistemas distribuidos muchas veces se ha equiparado con el paradigma de paso de mensajes. Aun así, es posible implementar abstracciones de más alto nivel por encima del sistema de memoria distribuida. Durante la última década varios grupos de investigación han desarrollado estas abstracciones, lo que ha resultado en una familia de lenguajes conocidos como lenguajes *PGAS*[15].

(15) Del inglés *partitioned global address space.*

El espacio de direcciones global hace referencia al hecho de que el lenguaje proporciona un único espacio de memoria por encima de las memorias virtuales de las máquinas distribuidas. Obviamente, estos lenguajes no proporcionan memoria compartida, puesto que no se puede esperar que el hardware pueda encargarse de su coherencia. Aun así, el espacio de direcciones global permite definir estructuras de datos globales, lo que mejora el modelo de memoria distribuida en el que el programador tiene que recordar un conjunto de estructuras de datos independientes que actúan como una estructura de datos distribuida difícil de mantener. Con los lenguajes *PGAS*, los programadores se pueden centrar en el comportamiento de un único proceso y distinguir los datos locales de los datos no locales (remotos), pero estos lenguajes permiten simplificar la programación, puesto que eliminan los detalles del paso de mensajes y mejoran la productividad del programador. Los compiladores son los que generan todas las llamadas de comunicación cuando hay referencias a direcciones de memoria que no son locales.

Las implementaciones de lenguajes *PGAS* más importantes, y que estudiaremos a continuación, son *UPC*, Co-Array Fortran y Titanium, que extienden C, Fortran y Java, respectivamente.

3.2.1. UPC

UPC[16] se desarrolló en torno al año 2000. *UPC* le proporciona al programador una visión global del espacio de memoria en la que cuando los elementos (por ejemplo, matrices) se definen como compartidos (*shared*), se distribuyen en las memorias de los diferentes procesos, de modo que los elementos compartidos se distribuyen de manera cíclica o con bloques cíclicos. Este tipo de distribución hace que se pueda balancear mejor la carga pero, en general, también hace que se reduzca la localidad de los datos. Aun así, podemos mejorar la localidad de los datos asignando los fragmentos de los datos compartidos directamente a los procesadores, de tal modo que muchos datos contiguos queden en el mismo proceso.

(16)Del inglés *unified parallel C*.

Puesto que *UPC* extiende el lenguaje de programación C, es posible utilizar punteros. Los punteros de *UPC* pueden o bien ser privados (locales al proceso) o compartidos entre todos los procesos. Dado que los punteros pueden ser privados o compartidos y que, además, pueden apuntar a datos privados o compartidos, hay cuatro tipos de punteros tal y como muestra la figura 29.

Figura 29. Tipos de puntero en *UPC*

		Propiedad del puntero	
		Private	***Shared***
Propiedad de la referencia	***Private***	*Private-private*, `p1`	*Private-shared*, `p2`
	Shared	*Shared-private*, `p3`	*Shared-shared*, `p4`

Estas propiedades están asociadas a los tipos de lenguajes, como se muestra a continuación en el ejemplo, en el que pueden verse varias declaraciones de punteros en *UPC*.

```
int *p1;   /* puntero privado apuntando a datos locales */
shared int *p2; /* puntero privado apuntando al espacio compartido */
int *shared p3; /* puntero compartido apuntando a datos locales *;
shared int *shared p4; /* puntero compartido apuntando al espacio compartido */
```

El código de la suma de vectores que se muestra a continuación ilustra el uso del segundo caso (puntero privado apuntando al espacio compartido).

```
shared int v1[N], v2[N], v1v2sum[N];

void main()
{
```

```
    int i;
    shared int *p1, *p2;
    p1=v1;
    p2=v2;
    upc_forall(i=0; i<N; i++; p1++; p2++; i)
    {
       v1v2sum[i]=*p1+*p2;
    }
}
```

El código anterior también ilustra el uso de otro elemento básico de *UPC*, el `upc_forall`. Esta abstracción distribuye las iteraciones de bucles de C normal (`for`) en los procesos utilizando una cláusula de afinidad (que es la cuarta cláusula del `upc_forall`). La cláusula de afinidad del bucle indica dónde se deben ejecutar las iteraciones del bucle. En el ejemplo anterior, el proceso que ejecuta la iteración *i*-ésima es el que está asociado a `i` (el propietario). `upc_forall` es una operación global en la que la mayor parte del código se ejecuta de manera local en un proceso.

3.2.2. Co-Array Fortran

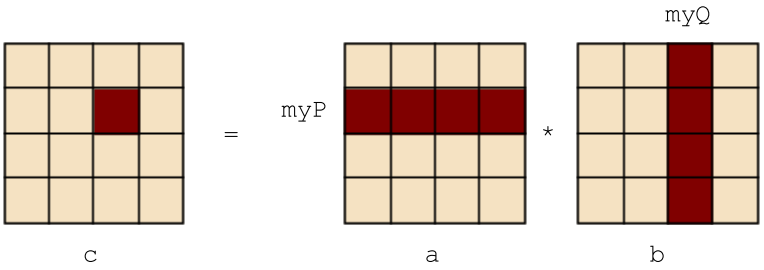
Co-Array Fortran (CAF) es una extensión de Fortran desarrollada a finales de los años noventa. CAF está caracterizado por su elegancia y simplicidad. La idea principal es añadir una extensión al lenguaje denominada `co-array`, que es el mecanismo para la comunicación entre procesadores. Aprovechando que Fortran utiliza paréntesis para las referencias de las matrices, CAF añade corchetes a los nombres de variables para referirse a *co-arrays*. Por ejemplo, el código que se muestra a continuación define tres matrices con *co-arrays*.

```
real, dimension(n,n)[p,*]::a,b,c
...
do k=1,n
   do q=1,p
           c(i,j)[myP,myQ]=c(i,j)[myP,myQ]+a(i,k)[myP,q]*b(k,j)[q,myQ]
   enddo
enddo
```

La especificación de los *co-arrays* indica que la memoria de una variable que se distribuye por todos los procesos se denomina imagen en CAF. En el ejemplo anterior, por lo tanto, en cada proceso hay asignados fragmentos de las matrices `a,b` y `c`. Además, el hecho de que el *co-array* esté definido para tener dos dimensiones hace que los procesos se puedan organizar en la estructura lógica de dos dimensiones de *p* filas y *q* columnas. Hay que tener en cuenta que el símbolo ***** indica que cada fila de procesos se debe llenar lo máximo posible y, por lo tanto, si `num_images()=pq`, entonces hay *p* filas y *q* columnas de procesos. Cada proceso encontrará su ubicación en la organización de dos dimensiones llamando a la rutina `this_image()` y definiendo los dos

parámetros `myP` y `myQ` para indicar un fragmento concreto de la matriz. La gestión del particionado de datos, incluida la inicialización de sus valores, es responsabilidad del programador. En la figura 30, se muestra un ejemplo en el que `myP=2` y `myQ=3`.

Figura 30. Ejemplo de organización de *co-array* con CAF



El código anterior muestra que cada proceso, definido por `[myP,myQ]`, actualiza su fragmento de la matriz cuando calcula el producto escalar. Para indicar las filas se accede a los datos en los procesos `[myP, q]`, y para indicar las columnas se accede a los datos en los procesos `[q, myQ]`. Las llamadas de comunicación para estos accesos remotos las genera el compilador, lo cual hace que la programación se simplifique de manera sustancial.

3.2.3. Titanium

El lenguaje Titanium (Ti) es una extensión de Java que se puede ejecutar en computadores de memoria distribuida. Tiene un modelo de memoria similar a *UPC* y, del mismo modo que otros lenguajes *PGAS*, genera código de comunicación entre nodos basándose en una biblioteca de comunicaciones unidireccional. Una de las cualidades que lo diferencian de los otros lenguajes *PGAS* que hemos visto es el hecho de que es orientado a objetos. También se diferencia de Java por el hecho de añadir las denominadas regiones, que soportan una gestión de memoria de altas prestaciones como alternativa al mecanismo de *garbage collection*. Hay otras funcionalidades de Java que se han restringido o limitado, pero se soportan las funcionalidades más típicas. Por ejemplo, se han añadido matrices de dos dimensiones para hacer Ti más apropiado para la computación científica.

Una prestación de eficiencia importante de Ti es la iteración desordenada, `foreach`. Esta simplifica la tarea tanto del programador como del compilador, y funciona de tal modo que permite iterar sobre el dominio de una variable. A continuación, se muestra su utilización para la multiplicación de matrices.

```
public static void matMul(   double [2d] a,
                double [2d] b,
                double [2d] c)
{
   foreach (ij in c.domain())
   {
```

```
        double [1d] aRowi=a.slice(1, ij[1]);
        couble [1d] bColj=b.slice(2, ij[2]);
        foreach (k in aRowi.domain())
        {
           c[ij]+=aRowi[k]*bColj[k];
        }
     }
  }
```

De este modo, `foreach`, en contra de `forall`, permite concurrencia sobre múltiples índices dentro de un mismo bloque.

Titanium fuerza la sincronización global mediante barreras y el concepto de variable compartida entre todos los procesos que se denominan `single`. Por ejemplo, en una simulación es bastante habitual que cada proceso lleve a cabo sus cálculos sobre datos locales y que de manera periódica coordinen sus acciones. La barrera asegura que todos los procesos detienen sus cálculos al mismo tiempo, para leer de la memoria o actualizarla. La utilización de variables tipo `single` garantiza que todos los procesos están en la misma fase del cómputo. Por ejemplo, una simulación de partículas podría utilizar este concepto tal y como se muestra a continuación.

```
     int single stepCount=0;
     int single endCount=100;
     for (; stepCount<endCount; stepCount++)
     {
        Lee partículas remotas
        Calcula las fuerzas de la mía
        Ti.barrier();
        Escribe mis partículas utilizando nuevas fuerzas
        Ti.barrier();
     }
```

4. Esquemas algorítmicos paralelos

En este apartado, analizaremos algunos de los esquemas algorítmicos más utilizados en la resolución de problemas en entornos paralelos. Estudiaremos un conjunto de esquemas de referencia que constituye la base para la resolución de una gama amplia de problemas reales.

4.1. Paralelismo de datos

Con esta técnica, se trabaja con muchos datos que se tratarán de igual o similar manera. Típicamente, se trata de algoritmos numéricos en los que los datos se encuentran en vectores o matrices y se trabaja con los mismos llevando a cabo un mismo procesamiento. La técnica es apropiada para memoria compartida, y normalmente se obtiene el paralelismo dividiendo el trabajo de los bucles entre los diferentes flujos (por ejemplo, en OpenMP `#pragma omp for`). Los algoritmos en los que aparecen bucles donde no hay dependencias de datos, o estas dependencias se pueden evitar, son adecuados para paralelismo de datos. Si se distribuyen los datos entre los procesos trabajando en memoria distribuida, hablamos de particionado de datos.

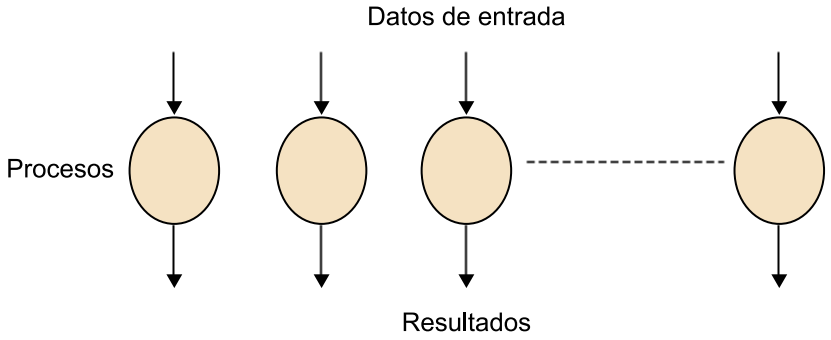
Este tipo de paralelismo es el que hay en la multiplicación de matrices en OpenMP, pues en este caso se paraleliza la multiplicación simplemente indicando con `#pragma omp parallel for` en el bucle más externo que cada flujo calcule una serie de filas de la matriz resultado. La aproximación de una integral también sigue este paradigma.

La suma secuencial de n números se lleva a cabo con un único bucle `for`, el cual se puede paralelizar con `#pragma omp parallel for`, con la cláusula `reduction` para indicar la forma en la que los flujos acceden a la variable en la que se almacena la suma. La paralelización se lleva a cabo de una manera muy simple y se habla de paralelismo implícito, pues el programador no es el responsable de la distribución del trabajo ni del acceso a las variables compartidas.

En algunos casos, nos puede interesar (por ejemplo, para disminuir los costes de gestión de los flujos) poner en marcha un trabajo para cada uno de los flujos, y que el programador se encargue de la distribución y el acceso a los datos, con lo que se complica la programación. En este caso, hablamos de paralelismo explícito.

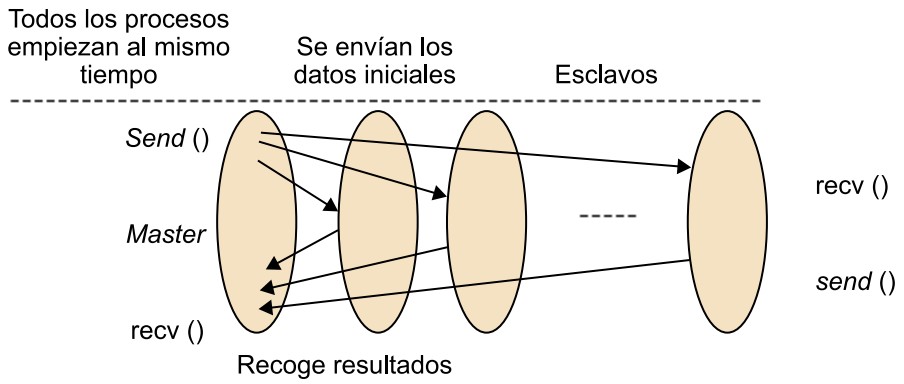
Este tipo de problemas que están compuestos de un conjunto de tareas independientes, como muestra la figura 31, también se conoce como *embarrassingly parallel*. En este tipo de aplicaciones, no hay o hay muy poca comunicación entre procesos y cada proceso puede llevar a cabo su tarea sin necesitar ninguna interacción con los otros procesos.

Figura 31. Modelo de ejecución de múltiples procesos con tareas independientes



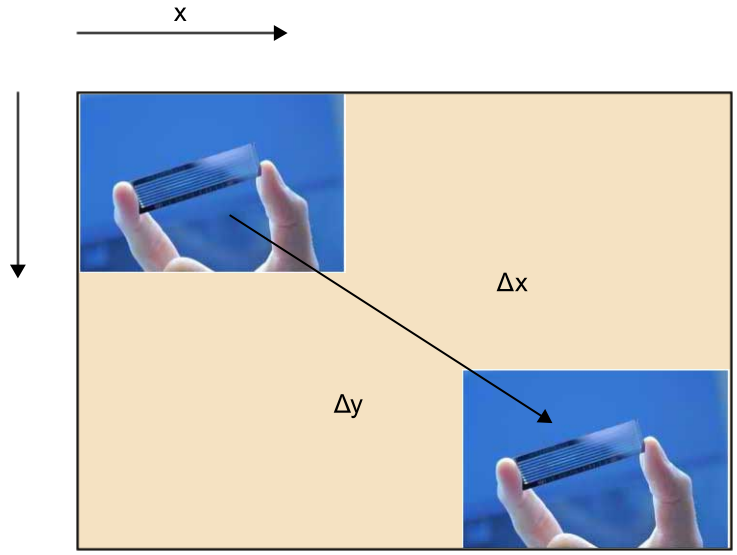
Si pensamos en una implementación estática con *MPI* de este modelo basado en el patrón maestro-esclavo, veremos que el proceso maestro crea un conjunto de procesos esclavos que se sincronizan mediante paso de mensajes al principio, efectúan su tarea independiente y, por último, se vuelven a sincronizar al final, cuando el proceso maestro recoge los resultados de los otros procesos (como ilustra la figura 32).

Figura 32. Implementación estática de tareas independientes con *MPI* basada en maestro-esclavo



Tal y como hemos comentado anteriormente, aplicaciones como por ejemplo el procesamiento de imágenes son típicas de este modelo. Un ejemplo claro es el desplazamiento de una imagen en el que cada píxel se tiene que desplazar un incremento concreto, como muestra la figura 33.

Figura 33. Ejemplo de aplicación de desplazamiento de una imagen



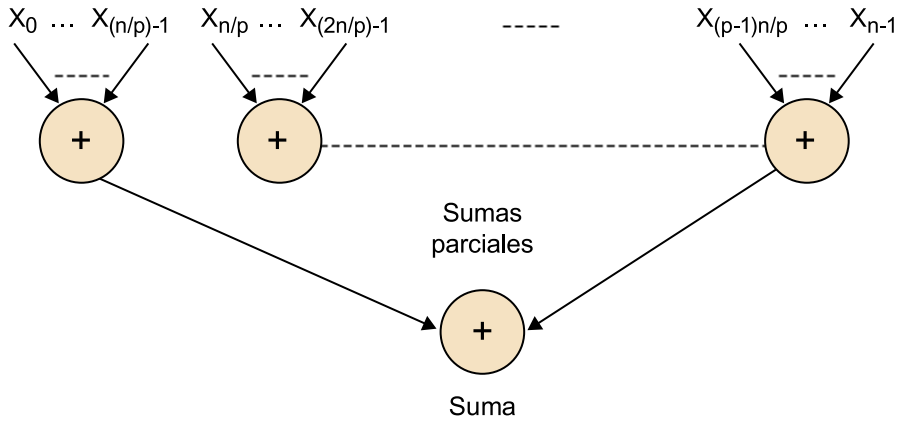
4.2. Particionado de datos

En este tipo de aplicaciones se trabaja con volúmenes de datos grandes que normalmente están en vectores o matrices, y se obtiene paralelismo dividiendo el trabajo en zonas diferentes de los datos entre distintos procesos. A diferencia del paralelismo de datos, en este caso se tienen que distribuir los datos entre los procesos, y esta distribución determina el reparto del trabajo. Lo más normal es dividir el espacio de datos en regiones y que en el algoritmo haya un intercambio de datos entre regiones adyacentes, por lo que hay que asignar regiones adyacentes a los procesadores, intentando que las comunicaciones no sean muy costosas. Algunas comunicaciones comparten datos y están distribuidas en la memoria del sistema, por lo que se produce un coste de comunicaciones que suele ser bastante elevado. De este modo, para obtener buenas prestaciones es necesario que el volumen de computación sea mucho mayor que el de comunicación.

Un tipo concreto de particionado de datos es el de dividir y conquistar. Este tipo de problemas se caracteriza por el hecho de dividir el problema en subproblemas como el problema principal. Normalmente, se divide en todavía más subproblemas de manera recursiva.

La figura 34 muestra un ejemplo de suma de una secuencia de números mediante patrón modelo dividir y conquistar, en el que cada proceso lleva a cabo una suma parcial que finalmente es agregada para obtener el resultado final.

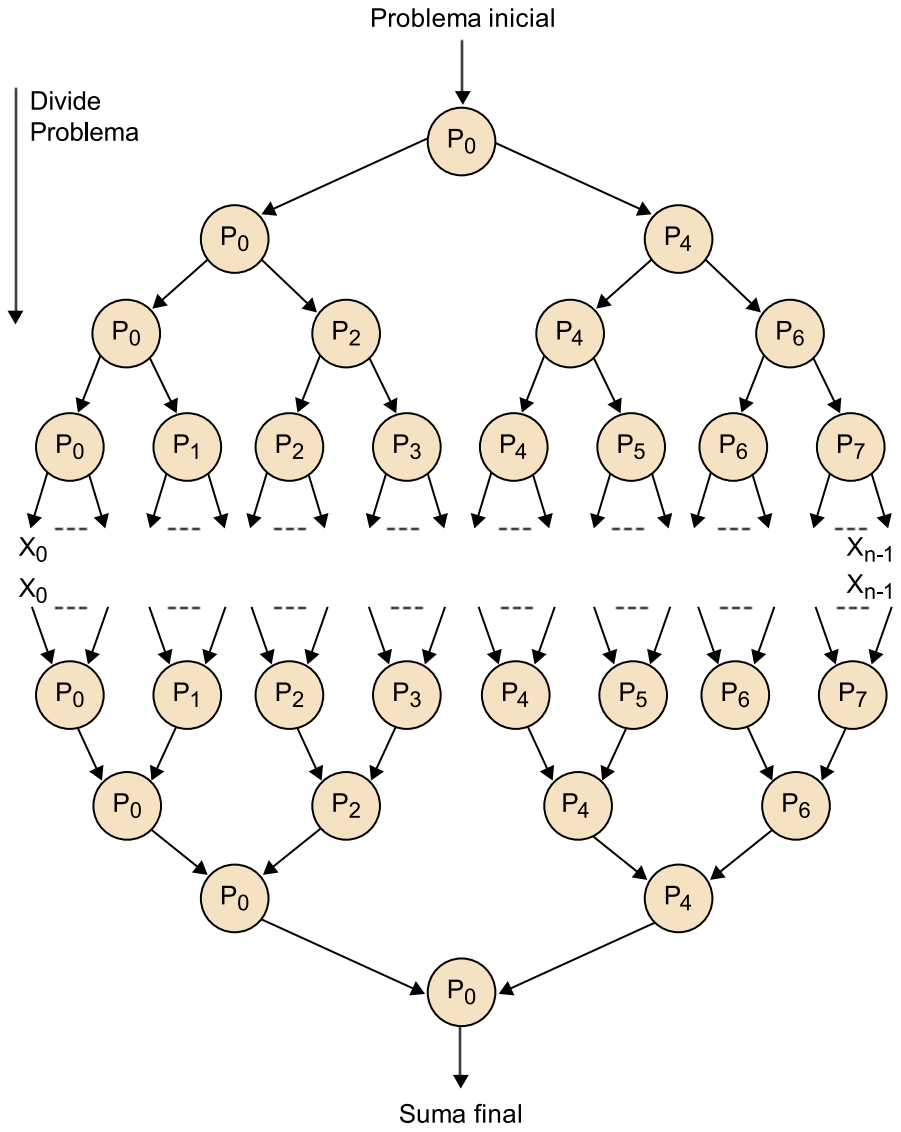
Figura 34. Ejemplo de suma de una secuencia de números mediante patrón modelo dividir y conquistar



4.3. Esquemas paralelos en árbol

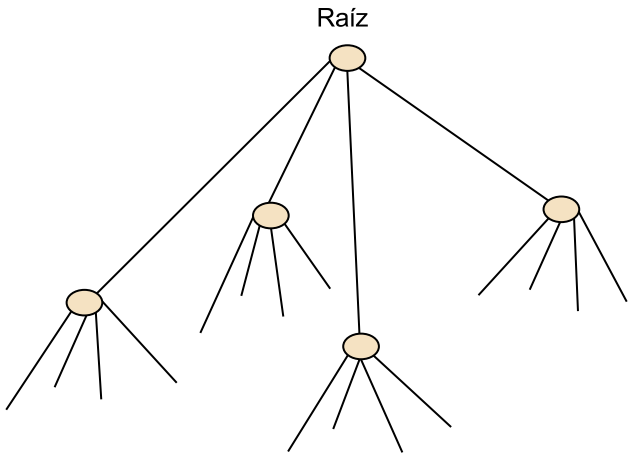
Muchos problemas tienen una representación en forma de árbol o grafo, y su resolución consiste en recorrer el árbol o grafo llevando a cabo computaciones. Una técnica para desarrollar programas paralelos es, dado un problema, obtener el grafo de dependencias que representa las computaciones que hay que llevar a cabo (nodos del grafo) y las dependencias de datos (aristas), y a partir del grafo decidir la asignación de tareas a los procesos y las comunicaciones entre los mismos, que vendrán dadas por las aristas que comunican nodos asignados a procesos diferentes. Un ejemplo es la suma prefija, como muestra la figura 35. Vemos cómo en la primera fase se divide el problema a medida que se van generando hijos y, en la segunda parte, se van recogiendo los resultados mientras se recorre el árbol.

Figura 35. Ejemplo de suma prefija mediante el esquema paralelo en árbol



Los árboles también pueden tener más de dos hijos, como ilustra la figura 36, donde los nodos tienen 4 hijos.

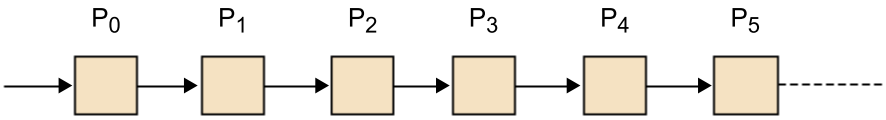
Figura 36. Esquema paralelo en árbol con 4 hijos



4.4. Computación en *pipeline*

Con el patrón *pipeline* se resuelve un problema descomponiéndolo en una serie de tareas sucesivas, de modo que los datos fluyen en una dirección por la estructura de procesos. De hecho, se puede ver como una forma de descomposición funcional en la que el problema se divide en diferentes funciones que se deben llevar a cabo de manera sucesiva, como muestra la figura 37. Posee una estructura lógica de paso de mensajes, pues entre las tareas consecutivas se tienen que comunicar datos. Puede tener interés cuando no hay un único conjunto de datos para tratar, sino una serie de conjuntos de datos que pasan a ser computados uno tras otro; o bien en problemas en los que hay paralelismo funcional, con operaciones de diferente tipo sobre el conjunto de datos y que deben ejecutarse una tras otra.

Figura 37. Esquema *pipeline* sencillo



Por ejemplo, supongamos que tenemos que sumar los elementos de un vector mediante un algoritmo como el siguiente.

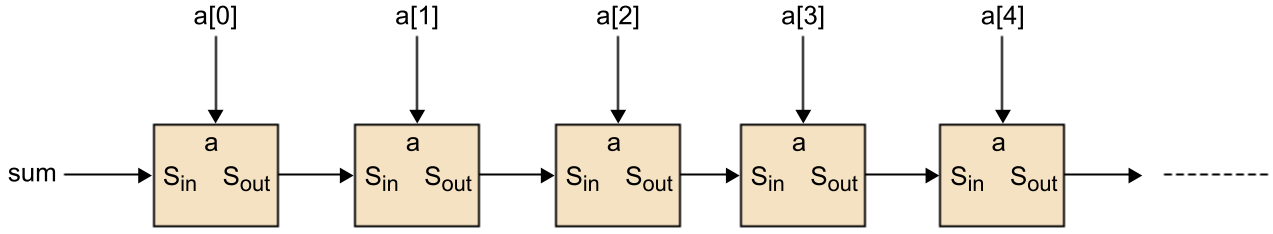
```
for (i = 0; i < n; i++)
   sum = sum + a[i];
```

y podemos desplegar el bucle de la manera siguiente:

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
...
```

De este modo, nos encontramos con el *pipeline* que muestra la figura 38:

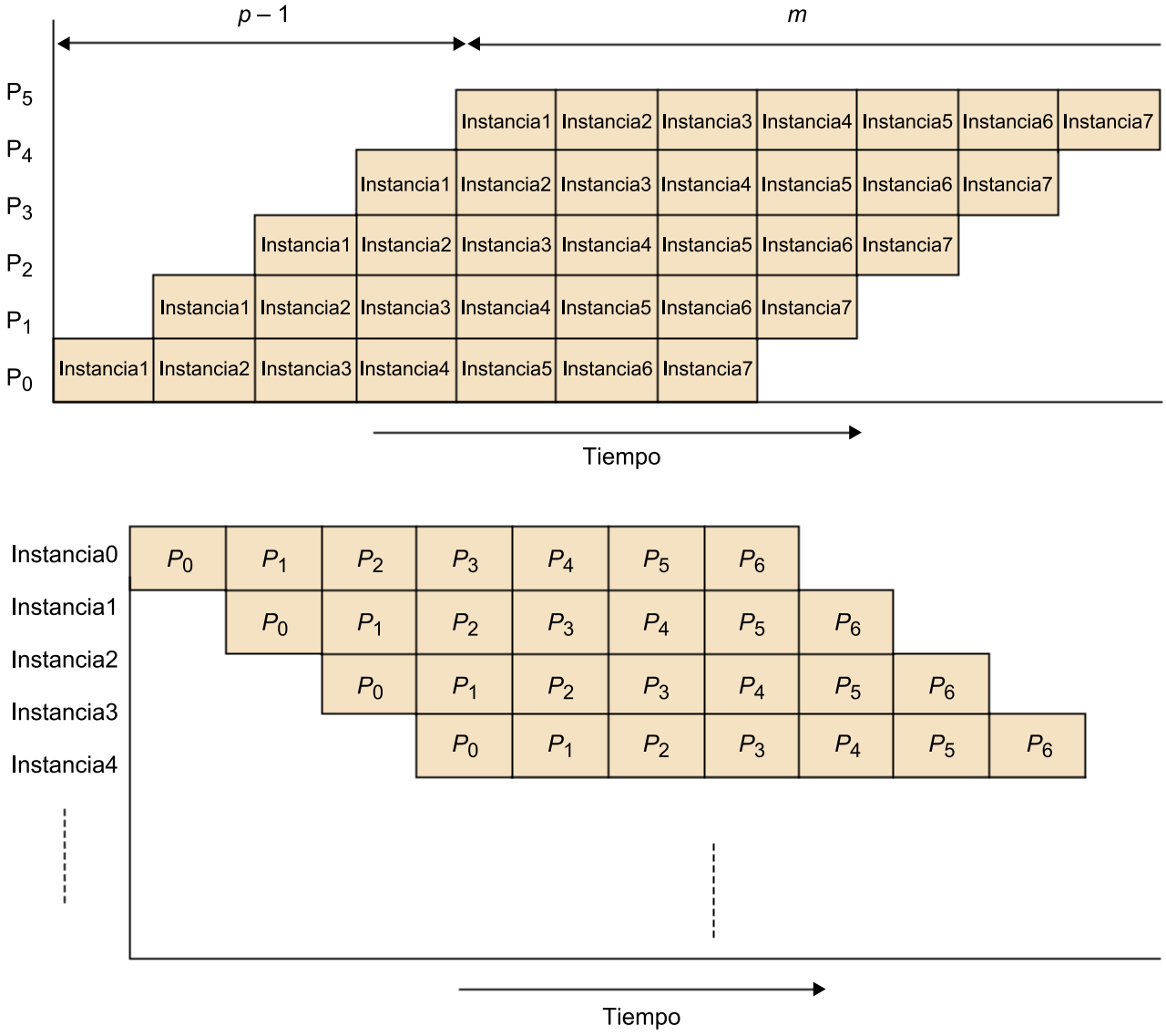
Figura 38. *Pipeline* del ejemplo de la suma de los elementos de un vector



Si suponemos que, en general, el problema se puede dividir en una serie de tareas secuenciales, el esquema de *pipeline* puede proporcionar un tiempo de ejecución más corto en los siguientes tipos de computaciones.

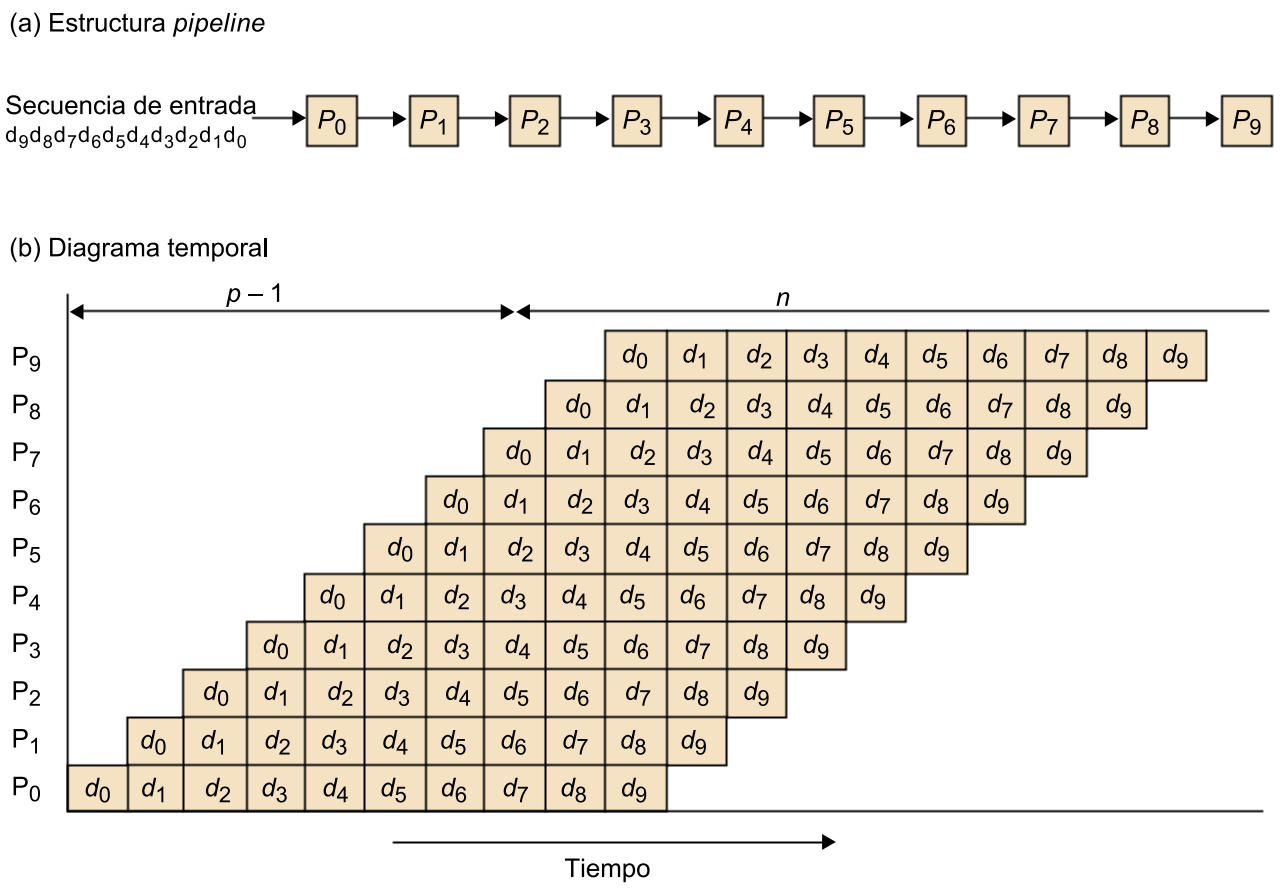
1) Si es posible ejecutar más de una instancia del problema completo al mismo tiempo, tal y como muestran los diagramas de la figura 39.

Figura 39. Diagrama espaciotemporal del primer tipo de *pipeline*



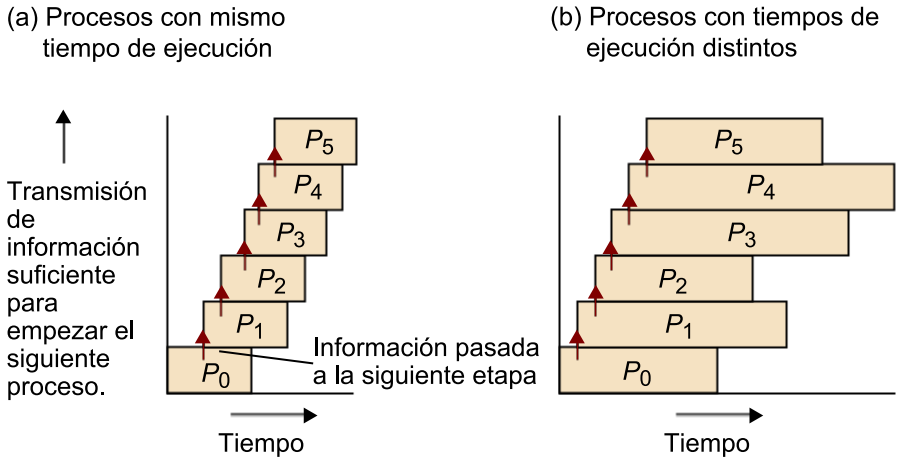
2) Si una serie de elementos de datos se tienen que procesar y cada uno requiere múltiples operaciones, como muestra la figura 40.

Figura 40. Diagrama espaciotemporal del segundo tipo de *pipeline*



3) Si la información necesaria para empezar la ejecución del siguiente proceso se puede pasar al siguiente proceso antes de que el proceso actual haya acabado sus operaciones, tal y como ilustra la figura 41. Un ejemplo de este tipo es la búsqueda de una cierta información en un conjunto de datos.

Figura 41. Diagrama espaciotemporal del tercer tipo de *pipeline*



4.5. Esquema maestro-esclavo

En el paradigma maestro-esclavo, tenemos un flujo/proceso diferenciado (denominado maestro) que pone en marcha los otros procesos (denominados esclavos), les asigna trabajo y recoge las soluciones parciales que generan.

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

Otro paradigma relacionado con estos dos es el de los trabajadores replicados. En este caso, los trabajadores (flujos o procesos) actúan de manera autónoma y resuelven tareas que posiblemente dan lugar a nuevas tareas que serían resueltas por el mismo trabajador u otro distinto. Se gestiona una bolsa de tareas: cada trabajador toma una tarea que se incluye en la bolsa y, una vez acabado el trabajo de la tarea asignada, toma una nueva tarea de la bolsa para trabajar con la misma, y así mientras quedan tareas para resolver. Se plantea el problema de la terminación, pues resulta posible que un trabajador solicite una tarea y no haya más en la bolsa, pero esto no significa que no queden tareas para resolver, porque puede que otro proceso esté computando y genere nuevas tareas. La condición de finalización será que no queden tareas en la bolsa y que no haya trabajadores trabajando. Hay diferentes maneras de abordar el problema de la terminación.

Este es el modo típico de trabajo en OpenMP. Inicialmente trabaja un único flujo y, al llegar a un constructor paralelo, pone en marcha una serie de flujos esclavos y se convierte en su maestro.

En programación por paso de mensajes, también se acostumbra a utilizar este patrón. En *MPI*, al ponerse en marcha los procesos con una orden como `mpirun -np 8 programa`, se inicializan ocho procesos iguales. Sin embargo, la entrada y la salida se suelen llevar a cabo mediante un proceso que actúa como maestro y que genera el problema y distribuye los datos. Tras este proceso empieza la computación, en la que puede intervenir o no el maestro, y al final el maestro recibe los resultados de los esclavos.

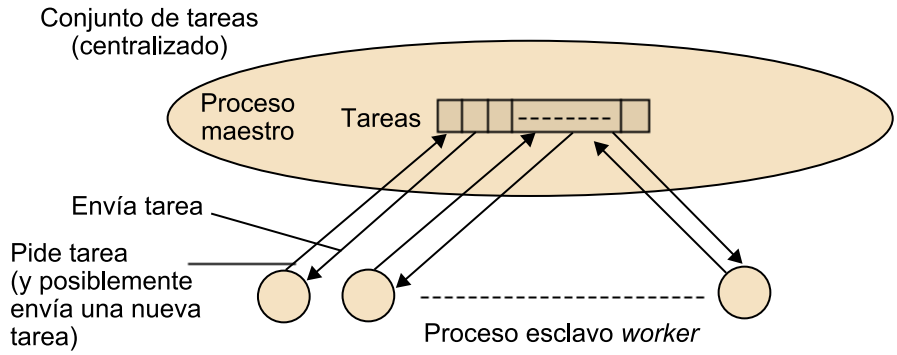
La asignación del trabajo a los esclavos puede efectuarse de manera estática o dinámica:

- En la asignación estática, el maestro decide los trabajos que asigna a los esclavos y lleva a cabo el envío.

- En la asignación dinámica, el maestro genera los trabajos y los almacena en una bolsa de tareas que se encarga de gestionar. Los esclavos van pidiendo trabajo de la bolsa de tareas a medida que van quedando libres

para efectuar nuevas tareas. De este modo, se equilibra la carga de manera dinámica pero puede haber una sobrecarga de la gestión de la bolsa y esta puede convertirse en un cuello de botella. Además, en algunos problemas, al resolver una tarea los esclavos generan nuevas tareas que se deben incluir en la bolsa, por lo que se generan más comunicaciones con el maestro. Un ejemplo de esto se muestra en la figura 42.

Figura 42. Ejemplo de asignación dinámica mediante un conjunto de tareas centralizado



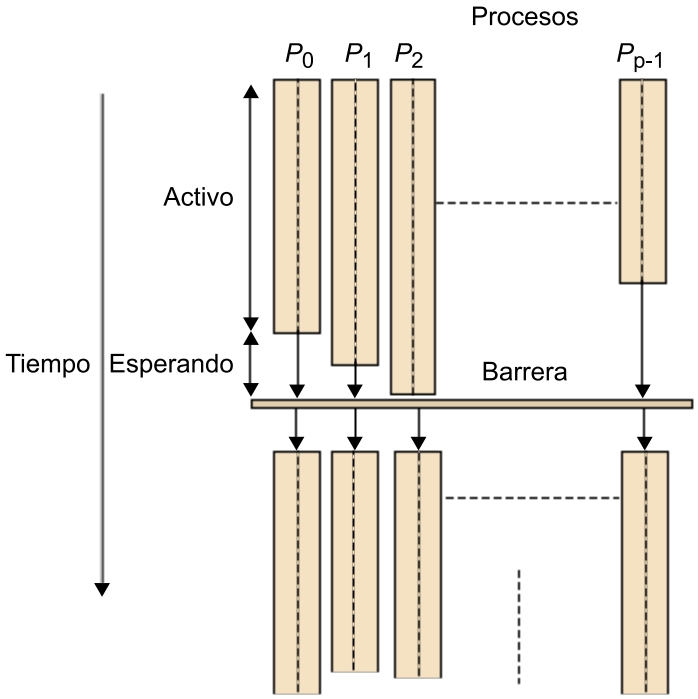
4.6. Computación síncrona

Hablamos de paralelismo síncrono cuando se resuelve un problema por iteraciones sucesivas en las que todos los procesos se sincronizan. Las características generales son las siguientes.

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

Hay muchos problemas que se resuelven con este tipo de patrón. El mecanismo principal es la utilización de barreras allí donde los procesos se tienen que sincronizar. Los procesos pueden continuar su ejecución cuando todos llegan a este punto, como muestra la figura 43.

Figura 43. Procesos llegando a una barrera en diferentes momentos



Actividad

Buscad posibles implementaciones de barreras tanto para memoria compartida como distribuida. Algunos ejemplos típicos de posibles implementaciones son:

a) Centralizado (lineal).

b) En forma de árbol.

c) *Butterfly*.

Bibliografía

Gaster, B.; Howes, L.; Kaeli, D. R.; Mistry, P.; Schaa, D. (2011). *Heterogeneous Computing with OpenCL*. Burlington, Massachusetts: Morgan Kaufmann.

Grama, A.; Karypis, G.; Kumar, V.; Gupta, A. (2003). *Introduction to Parallel Computing*. Reading, Massachusetts: Addison-Wesley.

Hwang, K.; Fox, G. C.; Dongarra, J. J. (2012). *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Burlington, Massachusetts: Morgan Kaufmann.

Jain, R. (1991). *The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience.

Kirk, D. B.; Hwu, W. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, Massachusetts: Morgan Kaufmann.

Lin, C.; Snyder, L. (2008). *Principles of Parallel Programming*. Reading, Massachusetts: Addison Wesley.

Munshi, A. (2009). *The OpenCL Specification*. Khronos OpenCL Working Group.

Munshi, A.; Gaster, B.; Mattson, T. G.; Fung, J.; Ginsburg, D. (2011). *OpenCL Programming Guide*. Reading, Massachusetts: Addison-Wesley Professional.

Nvidia (2007). *Nvidia CUDA Compute Unified Device Architecture*. Technical Report.

Nvidia (2007). *The CUDA Compiler Driver NVCC*.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Burlington, Massachusetts: Morgan Kaufmann.

Sanders, J.; Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Reading, Massachusetts: Addison-Wesley Professional.