Universidad Internacional de Valencia

Paralelismo
G23GIIN

UC-4: Herramientas de Programación Paralela

Yudith Cardinale

Diciembre 2022

Introducción

Modelos de comunicación en programación paralela:

- Por memoria compartida
  - Sistemas centralizados: pipes, semáforos, mutex, señales, hilos,…
  - Sistemas **paralelos con memoria compartida:** hilos con OpenMP
- Cliente-servidor: para redes y sistemas distribuidos (e.g., *sockets*, RPC, RMI).
- Pase de mensajes: para sistemas centralizados, distribuidos y **paralelos** (MPI, CUDA).

Introducción

Modelos de comunicación en programación paralela:

- Basada en *streams:* para sistemas distribuidos en escenarios en los que el tiempo y orden de los mensajes es vital (transferencia de audio y vídeo).

- Comunicación en grupo: para sistemas centralizados, distribuidos y **paralelos**, para enviar mensajes entre múltiples procesos (e.g., *multicast, broadcast*)

Introducción

Comunicación por memoria compartida en HPC:

- Tiene algunas semejanzas con la programación concurrente:
  - Se basa en el uso de múltiples hilos;
  - Es el subsistema de comunicación, librería o sistema operativo quien realiza la distribución de los datos y cómputo y controla la comunicación;
  - Requiere de mecanismos de control de sincronizaciones y exclusión mutua;
  - Funciona solo en arquitecturas de una única memoria.

Introducción

Comunicación por pase de mensajes en HPC:

- Tiene las siguientes características:
  - Funciona tanto en arquitecturas paralelas de memoria compartida como distribuida;
  - Se puede usar combinado con el modelo de memoria compartida;
  - Es soportado por llamadas al sistema operativo o a través de librerías especiales;
  - Las primitivas principales de comunicación son:
    - **send(destino, mensaje, long_mensaje)**: para enviar un **mensaje** de tamaño **long_mensaje** a un proceso **destino**;
    - **receive(fuente, mensaje, &long_mensaje)**: para recibir un **mensaje** de tamaño **long_mensaje** de un proceso **fuente**.

Introducción

- **Comunicación en grupo en HPC**: Consiste en usar primitivas de comunicación que permitan enviar/recibir mensajes a/desde múltiples procesos.

- Independiente del modelo de comunicación usado:
  - la transferencia de datos y mensajes se realiza a través de un **enlace físico** (memoria, buses, cable coaxial, microondas, fibra óptica).

  - a nivel de los procesos se definen **enlaces lógicos** que el sistema operativo traduce en accesos a tales enlaces físicos:
    - son provistos a nivel de software y sirven para administrar la comunicación;
    - ¿Cómo se establecen? ¿A cuántos procesos está asociado? ¿Cuántos enlaces son posibles entre cada par de procesos? ¿Cuál es la capacidad del enlace y tamaño del mensaje soportado? ¿Es bidireccional o unidireccional?

Introducción

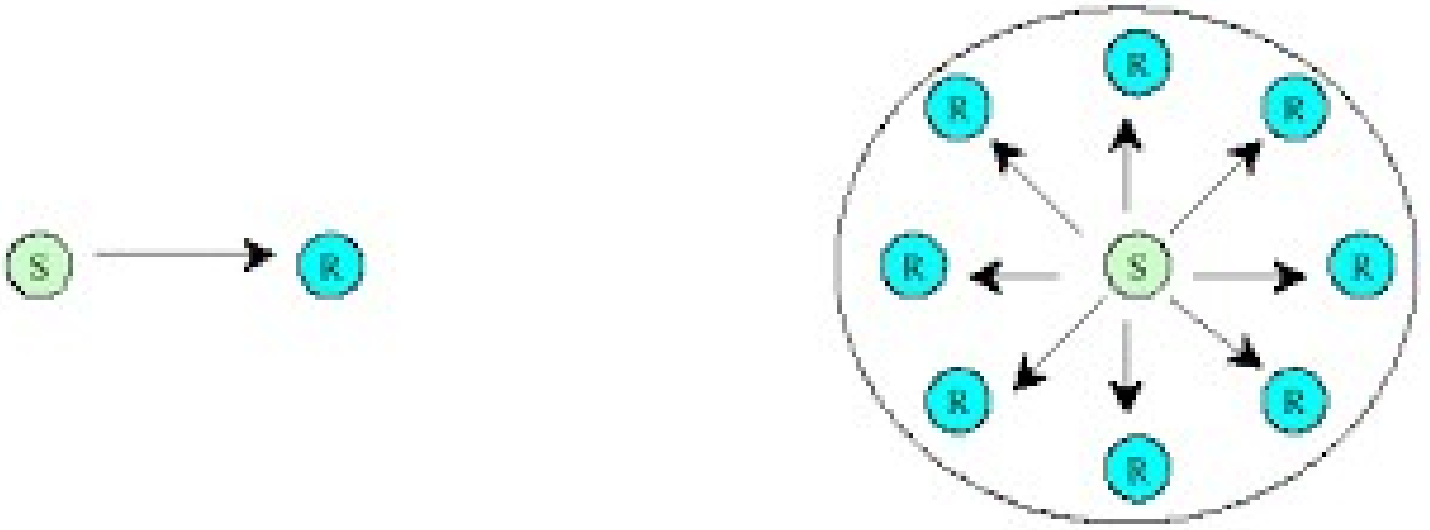
De acuerdo a las características de los enlaces lógicos, se define:

- **Comunicación directa o indirecta**
- **Comunicación simétrica o asimétrica**
- **Comunicación síncrona o asíncrona** → **Comunicación bloquente o no bloqueante**
- **Comunicación transitoria o persistente**

Introducción

Comunicación simétrica o asimétrica:

- La comunicación **simétrica**, se establece sólo entre dos procesos: **comunicación uno-a-uno**;
- La comunicación **asimétrica**, permite comunicar más de dos procesos en una sola comunicación al estilo: **uno-a-muchos, muchos-a-uno, muchos-a-muchos**;



Comunicación punto a punto y comunicación uno a muchos.

Introducción

Comunicación directa:

- En la comunicación **directa**, se nombran explícitamente a los procesos destino y fuente;
- No hay entidades intermediarias durante la comunicación;
- Los procesos enviador y receptor **deben conocer la identidad** del proceso destino o fuente, respectivamente, y colocarlo explícitamente en las primitivas de comunicación;
- El **enlace lógico** se establece automáticamente y se asocia sólo con dos procesos;
- Entre cada par de procesos existe sólo un enlace;
- Para este tipo de comunicación, los enlaces pueden ser unidireccionales o bidireccionales.

Introducción

Comunicación directa simétrica y directa asimétrica:

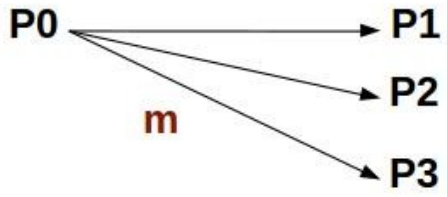
- **Directa simétrica**: los dos procesos que se comunican, conocen sus identificadores. La comunicación se establece mientras los dos procesos **se ejecutan en paralelo**;

P0	P1
…	…
`send(P1,message)`	`receive(P0,message)`
…	…

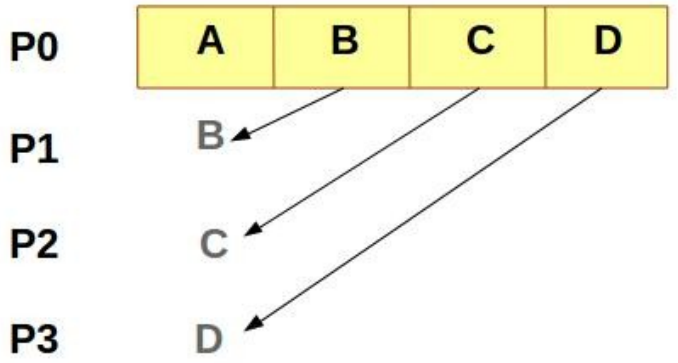
- **Directa asimétrica**: los procesos que se comunican, conocen sus identificadores. La comunicación se establece mientras los procesos **se ejecutan en paralelo**.

Introducción

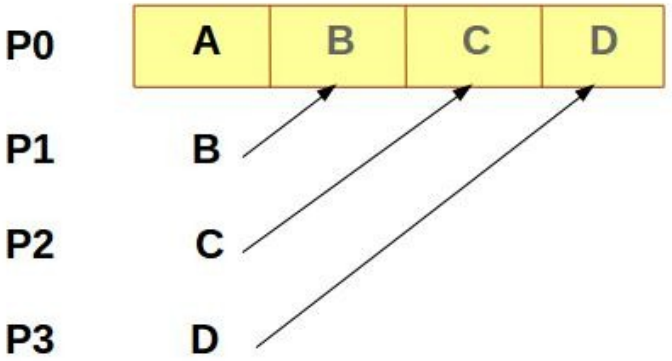
Ejemplos de comunicación directa asimétrica:



(a) *broadcast(P0,m) o multicast(P0,m)*



(b) *scatter(P0,m[i])*



(c) *gather(P0,m)*

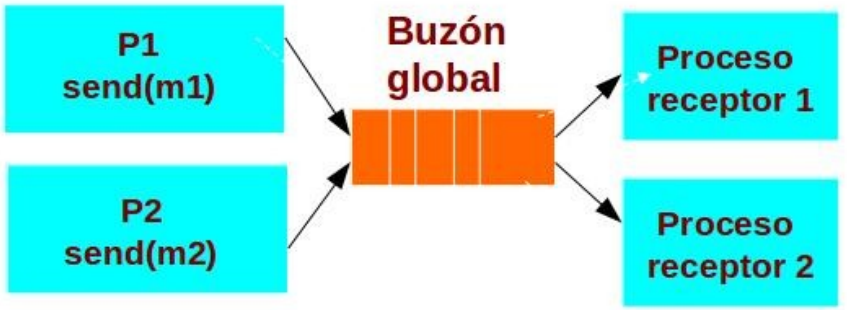
Introducción

Comunicación indirecta:

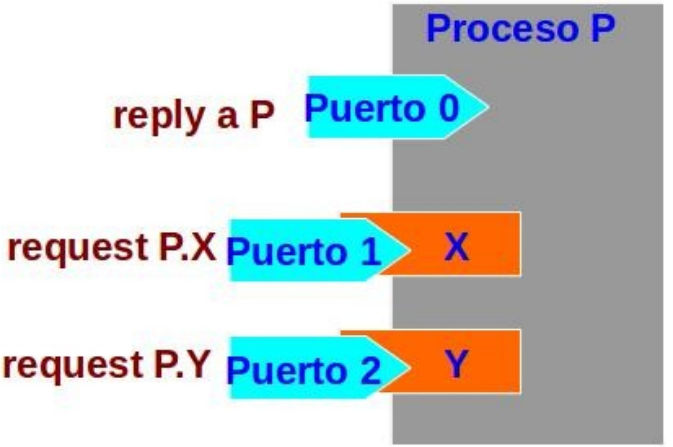
- En la comunicación indirecta, un **enlace lógico** se establece entre un par de procesos sólo si comparten un **buzón (*mailbox*), un puerto (*socket*) o un pipe**;
- Por lo tanto, un enlace se puede asociar con más de un par de procesos;
- Entre cada par de procesos puede haber más de un enlace lógico;
- Los enlaces pueden ser **unidireccionales o bidireccionales**.

Introducción

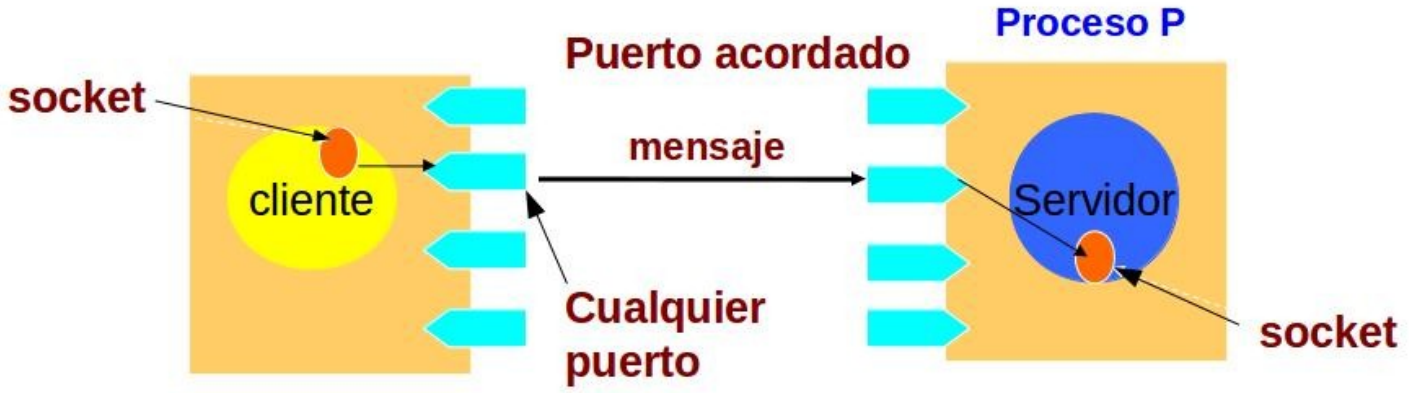
Ejemplos de comunicación indirecta:



(a) Comunicación indirecta por buzón



(b) Comunicación indirecta por puertos



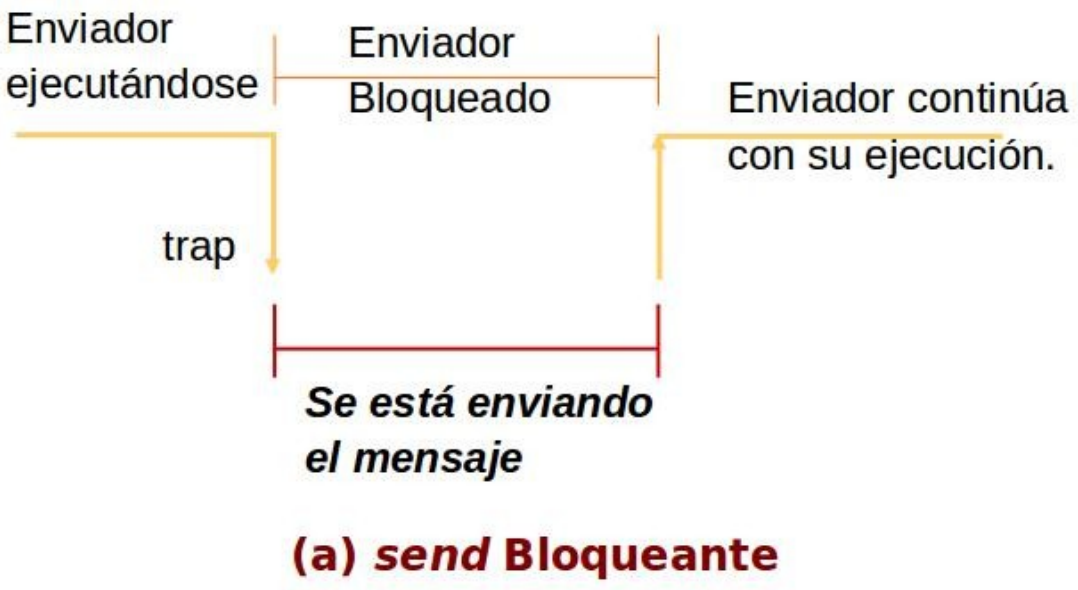
(c) Comunicación indirecta por sockets

Introducción

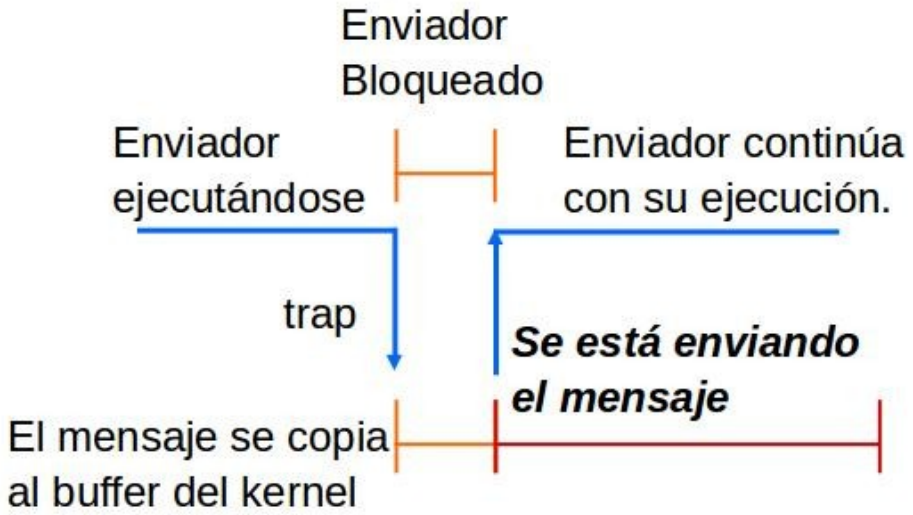
Comunicación síncrona o asíncrona:

- La implementación de las primitivas *send* y *receive* pueden ser implementadas como **síncronas (bloqueantes)** o **asíncronas (no bloqueantes)**:
  - Un *send* con bloqueo espera o se suspende hasta que el receptor haya recibido;
  - Un *send* sin bloqueo continúa aunque el receptor no haya recibido, retorna el control al llamador inmediatamente antes que el mensaje sea enviado;
  - Un *receive* con bloqueo espera hasta que el enviador haya enviado;
  - Un *receive* sin bloqueo continúa aunque el enviador no haya enviado, le indica al kernel un *buffer* donde se dejará el mensaje y la llamada retorna inmediatamente.

Introducción



(a) *send* Bloqueante



(b) *send* No-bloqueante

Introducción

Comunicación transitoria o permanente:

- En la comunicación **transitoria:**
  - los mensajes son almacenados por el sistema de comunicaciones sólo mientras los procesos enviador y receptor se están ejecutando;
  - si alguno de los dos procesos falla, el mensaje es eliminado.
- La comunicación **directa** permite comunicación **transitoria:** los mensajes son mantenidos por el subsistema de comunicación sólo mientras se ejecutan enviador y receptor.
- En la comunicación **persistente:**
  - los mensajes son almacenados por el sistema de comunicaciones hasta que el receptor los tome;
  - No requiere la ejecución simultánea de enviador y receptor.
- La comunicación **indirecta** permite comunicación **persistente**: el mensaje se mantiene en el subsistema de comunicaciones aún cuando ninguno de los procesos enviador/receptor estén activos.

Paralelismo con MPI

Programación paralela con memoria distrubuida:

- HPC ha sido dominado desde sus inicios por el **paradigma paralelo de memoria distribuida**, pues permite distribuir el trabajo entre varios procesos con su propio espacio de memoria;
- El modelo de comunicación usado en este paradigma requiere de mecanismos de **pase de mensaje** entre los procesos;
- Este modelo se popularizó con las arquitecturas multi-núcleos, multiprocesadores y los *cluster* de computación;
- En respuesta a este avance, Gropp, W. et al., (1994) propusieron un sistema de pase de mensajes estandarizado y portable, llamado **MPI** por *Message Passing Interface*;
- Desde entonces, MPI ha sido el estándar de-facto para la comunicación entre procesos en sistemas de memoria distribuida y así el sistema de programación dominante en el mundo de HPC.

Paralelismo con MPI

Librerías MPI

- El estándar MPI ha sido la base de muchos proyectos de desarrollo capaces de ejecutarse en diferentes plataformas, redes y sistemas operativos; así como sobre diferentes lenguajes de programación (Fortran 77, HPF, F90, C y Java).
- Otros proyectos, proveen implementaciones de MPI para redes específicas como Myrinet, InfiniBand y Quadrics.
- Existen muchas implementaciones disponibles libres y de dominio público, soportadas en los lenguajes de programación más populares actualmente (Fortran, C, Java, Phyton): MPI-LAM, MPICH y OpenMPI.
- En esta asignatura usaremos OpenMPI (http://www.open-mpi.org/).

Paralelismo con MPI

Librerías MPI

- Independientemente de la implementación del estándar MPI que se use, las características principales de los APIs no varían:
  - Proveen soporte de **librerías de comunicación** con facilidades para desarrollar **programas paralelos**;
  - Permite modelar programas paralelos con el enfoque **maestro-esclavos**;
  - Permite modelar **topologías lógicas**;
  - Soporta el modelo **SPMD** (*Single Process Multiple Data*);
  - Incluye facilidades de comunicación **uno-a-uno** (punto-a-punto), **comunicación en grupo** (colectiva), definición de grupo de procesos, definición de topologías de procesos, interfaz de **perfilamiento** y operaciones de memoria compartida con **soporte de hilos**.

Paralelismo con MPI

OpenMPI: una implementación de MPI

⭐ Operaciones básicas:

```
#include <stdio.h>
#include <unistd.h>
#include "mpi.h"
Int main(int argc, char *argv[])
{
  int npes, myrank,i,j,hlen;
  char myhostname[255];
  gethostname(myhostname,hlen);
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&npes);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
     printf("From process: %d out of %d (on %s), Hello World!  \n",
                                    myrank,npes,myhostname);
  MPI_Finalize();
}
```

Paralelismo con MPI

OpenMPI: una implementación de MPI

- Todas las funciones de MPI empiezan por **MPI_** y deben incluirse en el programa a través de la directiva: **#include "mpi.h"**
- Todas las funciones de MPI del programa deberán estar comprendidas entre las líneas **MPI_Init(&argc, &argv);** (inicializa el proceso en su respectivo procesador) y **MPI_Finalize();** (finaliza el proceso).
- En MPI, los procesos que se comunican deben pertenecer a un grupo o **comunicador**. Por defecto se puede utilizar **MPI_COMM_WORD**, en cuyo caso el grupo de procesos es el conjunto de procesos lanzados conjuntamente para resolver un problema.
- Para conocer cuántos procesos se iniciaron con el programa paralelo se usa la función:
  MPI_COMM_SIZE(MPI_COMM_WORLD,&numprocess)

Paralelismo con MPI

OpenMPI: compilación y ejecución

- Un programa que usa la librería MPI, debe compilarse usando el *script* disponible para ello. En OpenMPI es:
  - **mpicc**: se usa para compilar y enlazar programas en C que hagan uso de MPI. Puede utilizarse con las mismas opciones que el compilador de C/C++ usual, por ejemplo:

```
$ mpicc -c mpi1.c
$ mpicc -o mpi1 mpi1.o
```

  - o simplemente:

```
$ mpicc mpi1.c -o mpi1
```

- Para la ejecución también se usa un *script* especial: **mpirun**. La forma más usual de ejecutar un programa MPI es con la orden:

```
$ mpirun -np 2 mpi1
```

El argumento **-np** sirve para indicar cuántos procesos ejecutarán el programa **mpi1**. En este caso, OpenMPI lanzará dos procesos **mpi1**. Como no se indica nada más en la orden de ejecución, OpenMPI iniciará dichos procesos en la máquina local.

Paralelismo con MPI

OpenMPI: compilación y ejecución

- Para tener un mayor control sobre qué proceso se lanza en cada máquina, se puede especificar en la orden de ejecución, el conjunto de máquinas que se usarán.
- En un *cluster* de computadoras, la forma más simple es especificando una lista de máquinas en un archivo de texto que podrá llamarse **machines.txt**.
- La asignación de procesos se hará por orden: el proceso número 0 a la primera máquina de la lista, el proceso 1 a la segunda máquina y así sucesivamente. Si hay más procesos que máquinas, se continúa la asignación empezando otra vez por el principio de la lista.
- La llamada a **mpirun** ejecuta una copia del programa en cada uno de los procesadores especificados.
- La ejecución de un programa OpenMPI asumiendo un archivo local de máquinas sigue el formato:

```
$ mpirun [ -nolocal ] -machinefile <maquinas> -np <num_proc> <prog>
```

Paralelismo con MPI

- **Ejemplo**, si el archivo de máquinas **machines.txt**, contiene las siguientes líneas:

  159.90.9.187
  159.90.9.188
  159.90.9.189
  159.90.9.190

y se ejecuta el programa **mpi1** sobre 2 procesos con la sentencia:

```
$ mpirun -machinefile machines.txt -np 2 mpi1
```

Esto significa que la primera réplica del programa ejemplo (el proceso 0) se ejecutará sobre la máquina 159.90.9.187 y la segunda (el proceso 1) se lanzará en la máquina 159.90.9.188.

Si se incluye el modificador **-nolocal**, no se asignará ninguna réplica a la la máquina local (desde la que se llama a mpirun), aunque esté en la lista de máquinas.

Paralelismo con MPI

- **Discusión para la clase:**
  - Intentar acceder al cluster de 4 PCs en el laboratorio virtual:
    - verificar que mpi funciona:

```
$> mpicc hello_world.c -o hello
$> mpirun -np 4 hello
```

  - Instalar OpenMPI en sus ordenadores:
    - Usando aptget:

```
$> sudo apt install libopenmpi-dev
```

    - verificar que openmpi funciona:

```
$> mpicc hello_world.c -o hello
$> mpirun -np 4 hello
```

Paralelismo con MPI

⭐ **Comunicación uno-a-uno bloqueante**:

```
#include “mpi.h”
int main(int argc, char **argv) {
    int myId, numprocess, i;
    MPI_Status status;
    //Inicializaciones
       MPI_Init(&argc,&argv);
       MPI_Comm_size(MPI_COMM_WORLD,&numprocess);
       MPI_Comm_rank(MPI_COMM_WORLD,&myId);
       if (myId==0) {
              printf(“Soy el proceso %d”,myId);
              for (i=1;i<numprocess;i++)
                  MPI_Send(&i,1,MPI_INT,i,0, MPI_COMM_WORLD);
        } else {
             MPI_Recv(&i,1,MPI_INT,0,0, MPI_COMM_WORLD, &status);
             printf(“Soy el proceso %d, recibi el nro. %d”,myId,i);
        }
        MPI_Finalize();
}
```

Paralelismo con MPI

int MPI_Send (void *buffer, int count, MPI_Datatype type, int destino, int tag, MPI_Comm comunicador)

int MPI_Recv (void *buffer, int count, MPI_Datatype type, int origen, int tag, MPI_Comm comunicador , MPI_Status *estado)

Los primeros argumentos de estas funciones son inmediatos:

- Un puntero a los datos (**void *buffer**) a enviar o al *buffer* donde recibir (**&i** en el programa **mpi1.c**, tanto para *send* como *receive*.
- La longitud de los datos (**int count**) a enviar (**MPI_Send**) y la longitud máxima de los datos a recibir (**MPI_Recv**), en **mpi1.c** es **1** para *send* y *receive*.
- El tipo de los datos (**MPI_Datatype type**), en el programa **mpi1.c** es **MPI_INT**. MPI proporciona una serie de tipos predefinidos que comienzan todos por la cadena MPI_.
- El identificador o rango del proceso al que se le envían los datos (**int destino**) (en **MPI_Send** es **i** en el programa **mpi1.c**) y del que se esperan los datos (**int fuente**) (en **MPI_Recv** es **0** en el programa **mpi1.c**).

Paralelismo con MPI

- En el programa **mpi1.c**, el proceso 0, envía al proceso **i** (con **1<= i < numprocess**) y en contraparte, los procesos **i** reciben del 0.
- En el caso de **MPI_Recv**, se puede especificar la constante **MPI_ANY_SOURCE**, que indica que puede recibir de cualquier proceso.
- El argumento **int tag** en una etiqueta que funciona como una marca al mensaje; puede ser usada por ejemplo, para diferenciar mensajes de distintos tipos. Hay un comodín llamado **MPI_ANY_TAG** para MPI_Recv. En **mpi1.c**, el tag es **0** para *send* y *receive*.
- El siguiente argumento es lo que se denomina el comunicador (**MPI_Comm comunicador**). Permite crear grupos de procesos que intercambian información. El programa **mpi1.c** usa el comunicador predefinido **MPI_COMM_WORLD**, que incluye a todos los procesos que están en ejecución.
- El último argumento de MPI_Recv (**MPI_Status *estado**) es un puntero a una estructura que contiene información sobre el mensaje. Por ejemplo, en caso de que se use **MPI_ANY_TAG** en una llamada a MPI_Recv, se puede identificar el proceso que ha enviado el mensaje examinando **status -> MPI_SOURCE**.

Paralelismo con MPI

Tipos de datos de MPI

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

Paralelismo con MPI

- **Discusión para la clase:**

- Compile y ejecute el programa mpi1.c con varios procesos:

```
$> mpicc mpi1.c -o mpi1
$> mpirun -np 2 mpi1
$> mpirun -np 4 mpi1
$> mpirun -maqLocal maquinas -np 8 mpi1
```

- Discuta sobre los diferentes resultados

Paralelismo con MPI

- **Comunicación colectiva (en grupo)**:
  - Las funciones colectivas en MPI, son invocadas por todos los procesos en un comunicador.
  - Dos de las funciones de comunicación en grupo más usadas en MPI son:

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

Distribuye datos de un proceso (el proceso con **ID = root**) a todos los demás procesos que pertenecen al mismo comunicador.

MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Combina datos desde todos los procesos que pertenecen a un comunicador y retorna el resulltado final a un solo proceso.

Paralelismo con MPI

Comunicación colectiva (en grupo):

```
#include “mpi.h”
#include <math.h>
int main(int argc, char **argv) {
   int myId, numprocess, n, i, rc; double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocess);
   MPI_Comm_rank(MPI_COMM_WORLD,&myId);
   while (1) {
      if (myId==0) {
         printf(“Introduzca el nro. de intervalos” );
         scanf(“%d”,&n);
      }
      MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
      if (n==0) break;
      else {
         for (i=myId+1;i <= n;i+=nunprocess) {
            realizar calculos de h y sum
          }
         mypi=h*sum;
         MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
         if (myid==0)
            printf(“pi es aprox. %.16f “,pi);
      } //else
   }//while
   MPI_Finalize();
}
```

Paralelismo con MPI

- **Comunicación colectiva (en grupo)**:
  - La instrucción **MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);** del ejemplo, indica que el proceso **0** enviará el valor de n (**&n**), que es es **1 MPI_INT**, a todos los demás procesos que pertenecen al comunicador por defecto **MPI_COMM_WORLD**.

  - Mientras que la instrucción:

MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

indica que el proceso **0** recibirá en la variable **&pi** la suma (**MPI_SUM**) de todos los valores parciales (**&mypi**), de todos los demás procesos que pertenecen al comunicador por defecto **MPI_COMM_WORLD**.

Paralelismo con MPI

Operaciones *Reduce* de MPI

- **MPI_MAX**: retorna el elemento máximo;
- **MPI_MIN**: retorna el elemento mínimo;
- **MPI_SUM**: retorna la suma de los elementos;
- **MPI_PROD**: retorna la multiplicación de los elementos;
- **MPI_LAND**: retorna el *and* lógico aplicado a todos los elementos;
- **MPI_LOR**: retorna el *or* lógico aplicado a todos los elementos;
- **MPI_BAND**: retorna el *and* a nivel de bits (*bitwise and*) aplicado a todos los elementos;
- **MPI_BOR**: retorna el *or* a nivel de bits (*bitwise or*) aplicado a todos los elementos.

Paralelismo con MPI

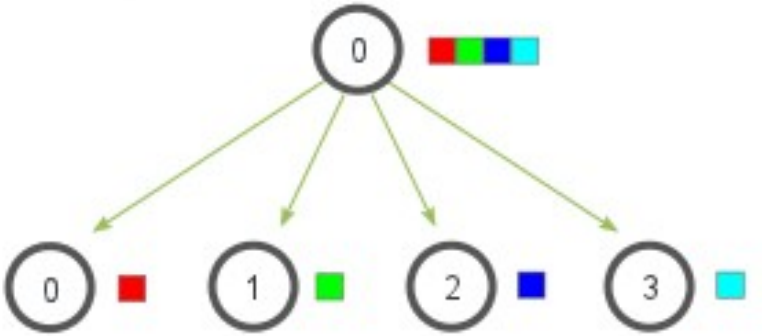
Más operaciones colectivas de MPI

- **MPI_Barrier( )**: bloquea los procesos hasta que todos la invocan;
- **MPI_Gather( )**: recibe valores de un grupo de procesos;
- **MPI_Scatter( )**: distribuye un *buffer* en partes a un grupo de procesos;
- **MPI_Alltoall( )**: envía datos de todos los procesos a todos;
- **MPI_Reduce_scatter( )**: combina valores de todos los procesos y distribuye;
- **MPI_Scan( )**: reducción prefija (0,...,i-1 a i).

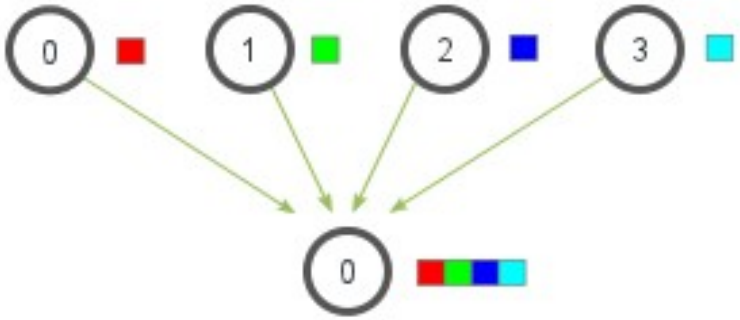
Paralelismo con MPI

Más operaciones colectivas de MPI

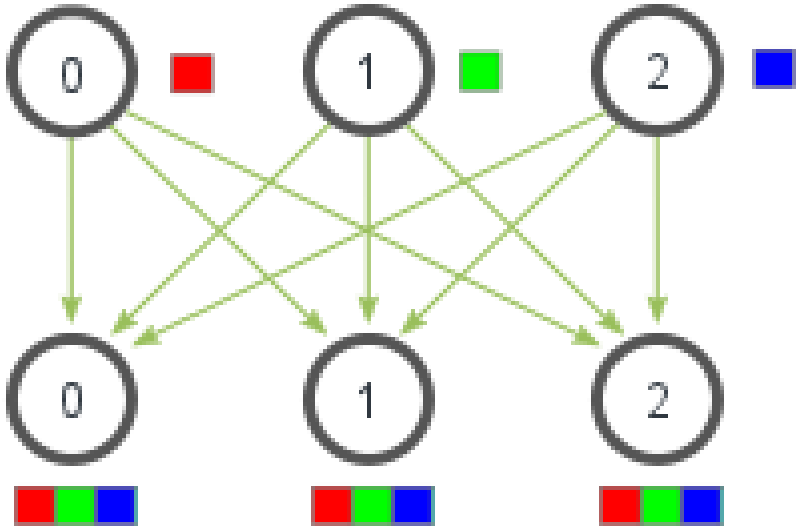
MPI_Scatter



MPI_Gather



MPI_Allgather



Paralelismo con MPI

Operaciones asíncronas (no bloqueantes) de MPI

- Para evitar problemas de interbloqueo o poder solapar comunicación con cómputo, MPI ofrece primitivas send y receive no-bloqueantes (asíncronas):
  - envío no-bloqueante:
    MPI_Isend(buf, count, datatype, dest, tag, comm, request):
  - recepción no-bloqueante:
    MPI_Irecv(buf, count, datatype, source, tag, comm, request):

    El parámetro **request** se usa para saber si la operación de envío correspondiente ya se realizó;
  - **MPI_Wait( )**: espera hasta que se complete la operación de envío correspondiente;
  - **MPI_Test( )**: devuelve un *flag* diciendo si la operación de envío correspondiente se ha completado.

Paralelismo con MPI

Medidas de rendimiento en MPI

- Para medir el tiempo de ejecución de una aplicación, MPI ofrece varias alternativas: usando rutina de librería llamada **MPI_WTIME** o usando la librería **MPE**.

- **MPI_WTIME** retorna la hora de ese momento.

- La librería **MPE** es una herramienta de perfilamiento que provee un amplio rango de funcionalidades para medir el rendimiento de programas MPI. Incluye funcionalidades para recolectar información sobre la ejecución de los programas y almacenarla en archivos *logs* tanto para realizar visualizaciones *post-mortum* como animación en tiempo real, a través de un sistema gráfico de ventanas.

Paralelismo con MPI

Universidad Internacional de Valencia

```
#include “mpi.h” // EJEMPLO DEL USO DE MPI_Wtime
#include <math.h>
int main(int argc, char **argv) {
   int myId, numprocess, n, i, rc; double mypi, pi, h, sum, x, a;
   double t1, t2;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocess);
   MPI_Comm_rank(MPI_COMM_WORLD,&myId);
   t1= MPI_Wtime();
   while (1) {
      if (myId==0) {
         printf(“Introduzca el nro. de intervalos” );
         scanf(“%d”,&n);
      }
      MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
      if (n==0) break;
      else {
         for (i=myId+1;i <= n;i+=nunprocess) {
            realizar calculos de h y sum
          }
         mypi=h*sum;
         MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
         if (myid==0)
            printf(“pi es aprox. %.16f “,pi);
      } //else
   }//while
   t2= MPI_Wtime();
   printf(“El tiempo total fue:  %.16f “,t2-t1);
   MPI_Finalize();
}
```

Paralelismo con MPI

- **Discusión para la clase:**
  - Investigar sobre MPE:
    - qué es y cómo se usa con MPI
    - verificar si se puede usar con OpenMPI.
    - encuentre ejemplos del uso y resultados de MPE
  - Revisar, compilar y ejecutar los programas
    - `primosec.c, primopar1.c y primopar2.c`
    - `piparallel.c`

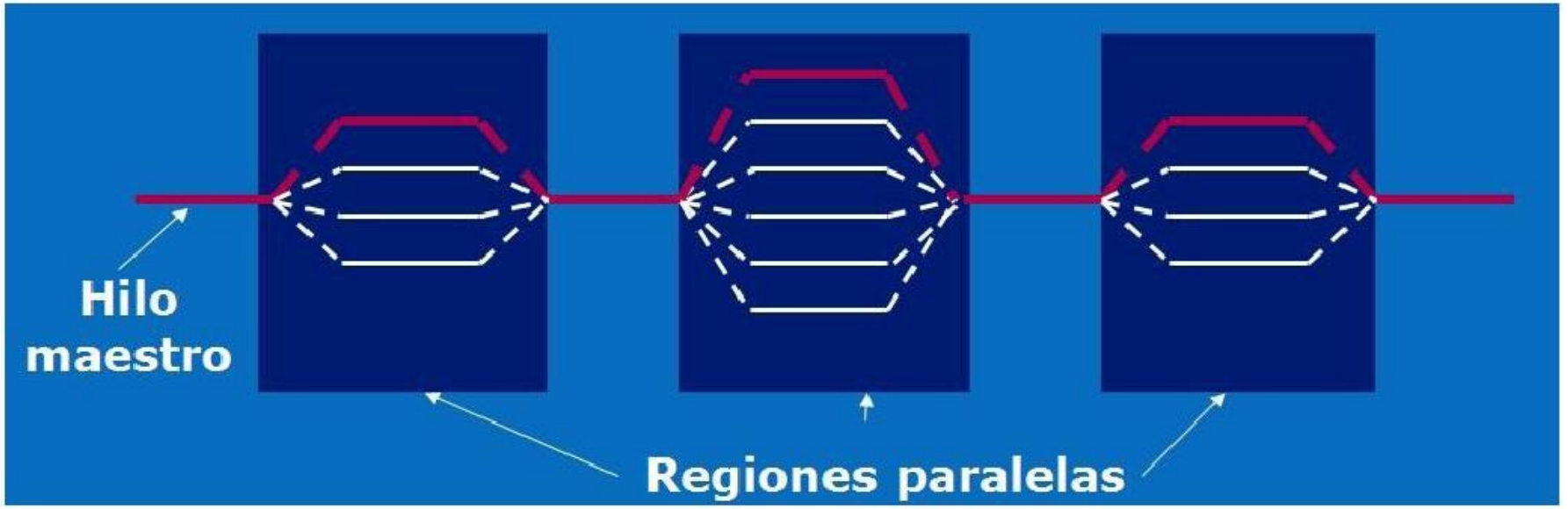
Paralelismo con OpenMP

- **OpenMP** es una especificación, un API, para implementaciones portables de paralelismo en FORTRAN y C/C++, a través del cual se puede expresar paralelismo de memoria compartida, basado en múltiples hilos.

- El sitio oficial de OpenMP es http://openmp.org/.

- Combina código serial y paralelo en un solo código fuente, en base al modelo llamado *fork-join***:** a partir de un único hilo de ejecución se crean N hilos paralelos (*fork* o división). Estos hilos terminan su ejecución conjuntamente en un punto posterior llamado punto de reencuentro o *join*. En ese momento, la ejecución del programa continúa de nuevo con un solo hilo hasta un nuevo punto de *fork* y así sucesivamente.

Paralelismo con OpenMP



Paralelismo con OpenMP

- Maneja el paralelismo basado en hilos de manera implícita y transparente al programador. El programador no tiene que preocuparse de la creación y eliminación de los hilos, ni de la sincronización entre ellos;
- Soporta el modelo de paralelismo de datos e incremental;
- El código es portable y escalable, ya que OpenMP encapsula las llamadas a rutinas de manejo de hilos específicas de cada plataforma;
- No usa el pase de mensajes, lo que minimiza la aparición de errores asociados a las transmisiones;
- El código paralelo se mantiene igual al secuencial, lo que permite ejecuciones tanto en uno como en muchos procesadores sin necesidad de cambiar el programa ni recompilar.

Paralelismo con OpenMP

Los tres principales componentes del API de OpenMP son:

- **Directiva de compilador**: línea especial de código fuente que tiene significado sólo para el compilador; indica al compilador que realice algo. Las directivas de OpenMP son:
  En Fortran: **!$OMP** En C/C++: **#pragma omp**
  Esto significa que las directivas de OpenMP son ignoradas si el código es compilado como un programa secuencial regular en Fortran/C/C++.

- **Funciones de librería**: rutinas que pueden ser invocadas desde el programa para que realicen una función específica.

- **Variables de entorno**: variables de ambiente que se definen antes de la ejecución del programa y que los hilos podrán acceder. Una de las más importante es **OMP_NUM_THREADS** que indica el número de hilos que se crearán al iniciarse una sección paralela (un punto *fork*). Ejemplo de cómo inicializarla en el *shell* de Linux: `$ export OMP_NUM_THREADS=8`.

Paralelismo con OpenMP

Instalación de OpenMP:

```
$ sudo apt-get update
$ sudo apt-get install codeblocks

$ sudo apt-get install gcc-multilib
```

Compilación:

Formato: gcc –fopenmp {source.c} opciones -o ejecutable

Ejemplo:

```
$ gcc -fopenmp e1.c -o e1
```

Ejecución:

```
$ ./e1
```

Paralelismo con OpenMP

Ejemplo 1:

```
/* Programa e1.c */
#include “stdio.h”
#include “stdlib.h”
#include “omp.h”

int main() {
 #pragma omp parallel  //Diectiva Openmp
   printf(“Hola mundo\n”);
  exit (0);
}
```

Compilación:

```
$ gcc -fopenmp e1.c -o e1
```

Ejecución: `$ ./e1`

- **Compile y ejecute. ¿Cuántos hilos se ejecutan?**
- **Elimine #include “omp.h”. Compile y ejecute. ¿Sigue funcinando? ¿Por qué?**
- **Ejecute desde la terminal el siguiente comando:**
  `$ export OMP_NUM_THREADS=8 ==> Variable de ambiente`
- **Ejecute de nuevo e1. ¿Cuántos hilos se ejecutan?**

Paralelismo con OpenMP

Ejemplo 2:

```
/* Programa e2.c */
#include “stdio.h”
#include “stdlib.h”
#include “omp.h”

int main() {
 int cantidad_hilos=6;
 omp_set_num_threads(cantidad_hilos); // Función de librería
 #pragma omp parallel  // Diectiva Openmp
   printf(“Hola mundo\n”);
  exit (0);
}
```

- Compile y ejecute. ¿Cuántos hilos se ejecutan?
- Elimine **`#include “omp.h”`**. Compile y ejecute. ¿Sigue funcinando? ¿Por qué?
- ¿Cuál es la diferencia con e1.c?
- Sustituya la función de librería **`omp_set_num_threads(cantidad_hilos)`** y la directiva **`#pragma omp parallel`** por la directiva **`#pragma omp parallel num_threads(6).`** Compile y ejecute. ¿qué sucede?

Paralelismo con OpenMP

Ejemplo 3: Variables compartidas y privadas

```
/* Programa e3.c */
#include "stdio.h"
#include "stdlib.h"

int main() {
 int comp=0; //Variable global a todos los hilos
 #pragma omp parallel {
    int priv=0; //Variable privada en cada hilo
    priv++;
    comp++;
    printf("Hola mundo! Valor priv = %d; valor compartido=%d\n",priv,
            comp);
   }
   exit (0);
}
```

La variable **comp** es compartida entre los hilos durante la sección paralela, mientras que la variable **priv** es privada para cada hilo. Así, invariablemente la impresión de los hilos dará como resultado 1 para todas las variables priv, mientras que el resultado de las variables **comp** podría ser imprevisible. **Explique por qué.**

Paralelismo con OpenMP

Variables compartidas y privadas (cont.):

Con OpenMP se pueden declarar de manera explícita variables privadas y compartidas que se acceden en las regiones paralelas:

```
#pragma omp parallel private(x0,y0) {
  ...
  x0 = xarray[mi_pos];
  y0 = yarray[mi_pos];
  f[mi_pos] = foo1(x0,y0);
  ...
}
```

En este ejemplo **x0** y **y0** son variables privadas a cada hilo, tomadas de los arreglos compartidos **xarray** y **yarray**, que son usados para calcular alguna variable que luego se almacena en el arreglo compartido **f**.

Paralelismo con OpenMP

Variables compartidas y privadas (cont.):

También es posible especificar qué variables son compartidas:

```
#pragma omp parallel private(x0,y0) shared(xarray,yarray,f)
  ...
  x0 = xarray[mi_pos];
  y0 = yarray[mi_pos];
  f[mi_pos] = foo1(x0,y0);
  ...
}
```

Paralelismo con OpenMP

Paralelización de ciclos:

En las regiones paralelas, todos los hilos ejecutan el mismo código, sin embargo, OpenMP provee directivas que indican que el trabajo se va a dividir entre los hilos, de manera que los datos se reparten entre los hilos, en lugar de replicarse. Esto es **paralelismo de datos**. Este tipo de problema es el más común cuando se trata de ejecutar instrucciones sobre elementos de arreglos o sobre listas numeradas del tipo:

```
for (i=inicio;i<fin;i++)
    x[i] = función(a[i])
```

Es responsabilidad del programador asegurar que las iteraciones de un ciclo paralelo sean independientes para que tenga sentido la paralelización.

Paralelismo con OpenMP

Paralelización de ciclos:

Ejemplo:

```
#define max 300
double vec[max];
#pragma omp parallel for
   for (i =0; i <max; ++i) {
         vec[ i ] = generate(i) ;
   }
```

la directiva **`#pragma omp parallel for`**, implica que el ciclo **for** se paraleliza y **automáticamente las iteraciones se dividen equitativamente** entre los hilos.

Paralelismo con OpenMP

Paralelización de ciclos (cont.):

La cláusula **if** en la directiva **parallel for**, permite decidir si la paralelización se realiza de acuerdo a una condición. En el siguiente ejemplo, el cálculo paralelo de **pi** se hace únicamente si el número de iteraciones es mayor a 5000, ya que para valores menores se considera que el trabajo de crear y gestionar los hilos es mayor que la ganancia de tiempo.

```
#pragma omp parallel for if ( n > 5000 )
for ( i= 0 ; i< n ; i++ ) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Paralelismo con OpenMP

Secciones críticas:

Observe el siguiente ejemplo de suma sobre una variable compartida (**sum**):

```
float producto(float* a, float* b, int N) {
   float sum = 0.0;
   #pragma omp parallel for shared(sum)
     for  (int i=0; i<N; i++)
        { sum += a[i] * b[i]; }
   return sum;
 }
```

- Este código funciona correctamente en el caso secuencial, sin embargo al ejecutarse de forma paralela, es posible que frecuentemente los valores resultantes sean erróneos. **¿De dónde surge el error?**
- Al analizar el código de forma exhaustiva, se observa que en la línea: **sum += a[i] * b[i];** no hay garantía que mientras uno de los hilos actualice la variable compartida **sum**, otro no esté alterándola. Si N es grande y hay muchos hilos, la probabilidad de que tales interferencias ocurran es muy alta. Para evitar este tipo de casos, una idea correcta es hacer que en el momento de la actualización, ***un solo hilo pueda acceder a la variable sum***.

Paralelismo con OpenMP

Secciones críticas (cont.):

OpenMP provee una cláusula que permite el acceso a bloques de código a uno solo de los hilos. Este bloque de código se denomina **sección crítica** y puede ser accedida solamente por uno de los hilos en cualquier momento, aunque otras partes del código pueden seguirse ejecutando de forma concurrente. El siguiente ejemplo ilustra la implementación correcta del código anterior definiendo una sección crítica:

```
float producto(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
       for (int i=0; i<N; i++)   {
           #pragma omp critical
               sum += a[i] * b[i];
           }
      return sum;
   }
```

Ahora sólo un hilo a la vez podrá acceder a sum, evitando condiciones de carrera.

Paralelismo con OpenMP

Operaciones con reducción:

OpenMP permite tener variables privadas que hagan acumulaciones parciales y, al final de la sección paralela, sumar todos los valores parciales en una variable totalizadora, mediante la directiva: **reduction(operador:lista-de-variables).** Esta agrupación puede ser una suma (como en el caso del producto interno) o también otros operadores aritméticos y lógicos.

```
float producto(float* a, float* b, int N) {
   float sum = 0.0;
   #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<N; i++)
        { sum += a[i] * b[i];}
 return sum;
}
```

Dentro del bloque de trabajo en paralelo se crea una copia privada de **sum** y se inicializa de acuerdo al elemento neutro de cada operación (0 para el caso de la suma). Estas copias son actualizadas localmente por los hilos de acuerdo al cómputo que se realice; en este caso cada hilo sumará el subvector que le correspondió de acuerdo a la división de la directiva **parallel for**. Al final del bloque paralelo, las copias locales se combinan de acuerdo al operador (**+:sum**, en el ejemplo) y se actualiza la variable compartida original (**sum**).

Paralelismo con OpenMP

Balance de carga:

El tiempo que tomará la ejecución de las tareas y la distribución de dichos tiempos puede generar que unos hilos tomen cargas más pesadas (lentas) que otros. Una forma de contrarrestar este problema es haciendo ***scheduling*** diferentes de acuerdo a los tiempos de ejecución. Para ofrecer balance de carga, OpenMP ofrece la directiva: **`schedule (<tipo>[,<tamaño>])`**

- **`schedule(static) o schedule(static,n)`**: planifica bloques de iteraciones equitativos (o de tamaño n) para cada hilo.
- **`schedule(dynamic) o schedule(dynamic,K)`**: cada hilo toma un bloque de 1 o K iteraciones, respectivamente, de una cola hasta que se han procesado todas.
- **`schedule(guided) o schedule(guided,K)`**: Cada hilo toma un bloque de iteraciones hasta que se han procesado todas. Se comienza con un tamaño de bloque grande y se va reduciendo exponencialmente hasta llegar a un tamaño K.
- **`schedule(runtime)`**: Se usa lo indicado en **`OMP_SCHEDULE`** o por la rutina de librería en tiempo de ejecución.

Paralelismo con OpenMP

Balance de carga (cont.):

¿Cúando usar cada una de estas opciones de planificación?

- **STATIC**: cuando todas las tareas a ejecutar son de complejidad similar. En este caso, no se pierde tiempo en la planificación y la ejecución será eficiente;

- **DYNAMIC**: cuando las tareas tienen tiempos impredecibles, cuando la complejidad en tiempo es desconocida y además es altamente variable;

- **GUIDED**: es útil cuando las cargas de tiempo son variables pero no demasiado, y se quiere minimizar el costo de planificación. Particularmente, cuando N es demasiado grande (>106).

En el siguiente ejemplo, cada hilo probará 8 números (8 iteraciones) y tomará siempre muestras de tamaño 8 hasta que se termine la ejecución. Sin embargo, todos los hilos tomarán la misma cantidad de muestras.

```
#pragma omp parallel for schedule (static, 8)
for( int i = start; i <= end; i += 2 )  {
       if ( TestForPrime(i) )
            gPrimesFound++;
}
```

Paralelismo con OpenMP

Balance de carga (cont.):

Si se cambia la cláusula por **dynamic**, los hilos irán tomando grupos de 8 iteraciones. En la medida que cada hilo vaya terminando, va tomando otros 8 valores para probar. Es posible que al final de la ejecución, un hilo haya ejecutado muchas más iteraciones que otros, sin embargo los tiempos serán relativamente uniformes entre todos:

```
#pragma omp parallel for schedule (dynamic, 8)
for ( int i = start; i <= end; i += 2 ) {
       if ( TestForPrime(i) )
           gPrimesFound++;
}
```

Paralelismo con OpenMP

Medidas de rendimiento:

Para medir el tiempo de ejecución de una aplicación, OpenMP ofrece una rutina de librería llamada **omp_get_wtime()** que retorna la hora de ese momento y se usa como muestra el siguiente ejemplo:

```
int main() {
      double *A, sum;
      double startTime, endTime, runtime;
      int flag = 0;
      A = (double *)malloc(N*sizeof(double));
      startTime= omp_get_wtime();
          #pragma omp parallel for
                for (i=0;i< MAX; i++) {
                     res[i] = huge();
                }
     endTime = omp_get_wtime();
     runtime = endTime – startTime;
          printf("  In  %lf  seconds,  The  sum  is  %lf  \
n",runtime,sum);
}
```

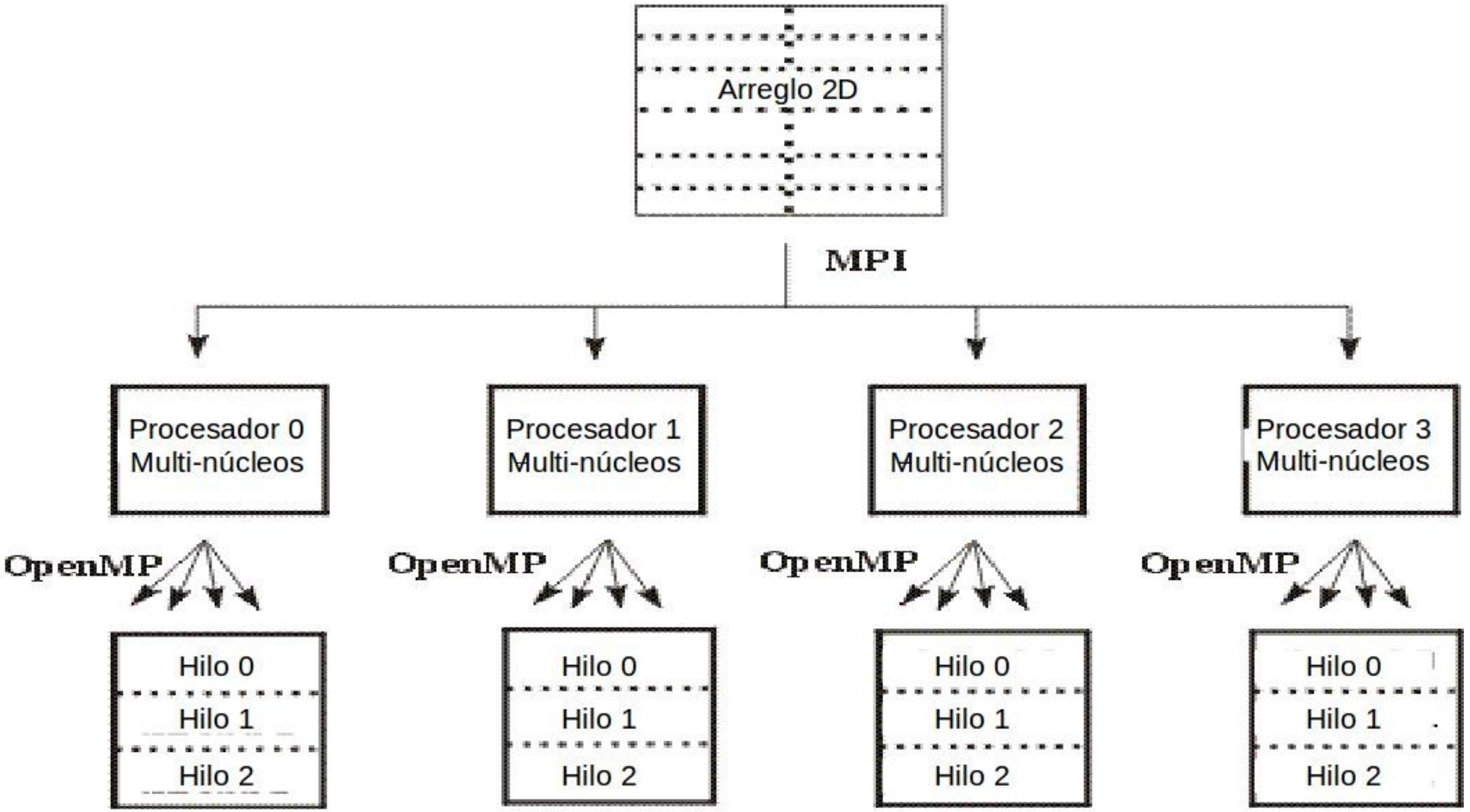
Paralelismo con OpenMP

Otras rutinas:

- **omp_set_num_threads**: Fija el número de hilos para el proceso que la invoca e ignora la variable de ambiente **OMP_NUM_THREADS**;
- **omp_get_num_threads**: Devuelve el número de hilos en ejecución en un momento determinado;
- **omp_get_max_threads**: Devuelve el número máximo de hilos que lanzará el programa en las zonas paralelas;
- **omp_get_thread_num**: Devuelve el identificador del hilo dentro de la sección de código parelelo (automáticamente a cada hilo se le asigna un identificador entre 0 y **omp_get_num_threads() – 1**). Este valor puede ser utilizado para diferenciar tareas entre los procesadores.

Paralelismo con OpenMP

Combinando MPI y OpenMP:



Paralelismo con OpenMP

Combinando MPI y OpenMP (cont.):

```
/* Programa e7.c */
#include <mpi.h>
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc,char **argv){
   int i; int nodo,numnodos;
   int tam=32;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&nodo);
   MPI_Comm_size(MPI_COMM_WORLD,&numnodos);
   MPI_Bcast(&tam, 1, MPI_INT, 0, MPI_COMM_WORLD);
     #pragma omp parallel
        printf("Soy el hilo %d de %d hilos dentro del procesador %d de %d
               procesadores\n", omp_get_thread_num(),
               omp_get_num_threads(), nodo,numnodos);
   MPI_Finalize();
}
```

Para compilar el programa e7.c, se hace con **mpicc** y se agrega la opción de OpenMP **-fopenmp**: **`$ mpicc -fopenmp e07.c -o e7`**

Para ejecutarlo se usa mpirun:

`$ mpirun --disable-hostname-propagation –machinefile ./maquinas.txt -np 8 ./e7`

Paralelismo con OpenMP

Combinando MPI y OpenMP (cont.):

```
$ mpirun --disable-hostname-propagation --machinefile ./maquinas.txt -np 4 ./e07

Soy el hilo 0 de 2 hilos dentro del procesador 0 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 0 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 1 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 1 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 2 de 8 procesadores
Soy el hilo 0 de 2 hilos dentro del procesador 3 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 2 de 8 procesadores
Soy el hilo 1 de 2 hilos dentro del procesador 3 de 8 procesadores

$
```