TEMA 2: PROGRAMACIÓN PARALELA

LECCIÓN 1: HERRAMIENTAS, ESTILOS Y ESTRUCTURAS EN PROGRAMACIÓN PARALELA

1. Problemas que plantea la programación paralela

1.1 Problemas respecto a la programación secuencial

- Dividir en unidades de cómputo independiente -> Tareas
- Agrupar/Asignar las tareas o la carga de trabajo en procesos/threads
- Asignar los procesos a procesadores/núcleos
- Sincronizar y comunicar los procesos

Estos problemas deben ser abordados, tanto por la herramienta de programación, como por el programador o por ambos.

1.2 De donde partir para realizar un programa paralelo

- Versión secuencial del problema
- Descripción o definición de la aplicación

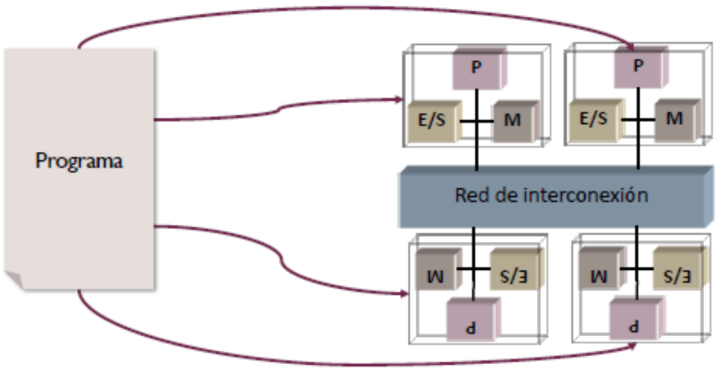
Además, podemos encontrar varios apoyos para desarrollar un programa paralelo:

- Otro programa paralelo que resuelva un problema parecido
- Versiones paralelas u optimizadas de bibliotecas de funciones, como por ejemplo:
  - BLAS (Basic Linear Algebra Subroutine)
  - LAPACK (Linear Algebra PACKage)

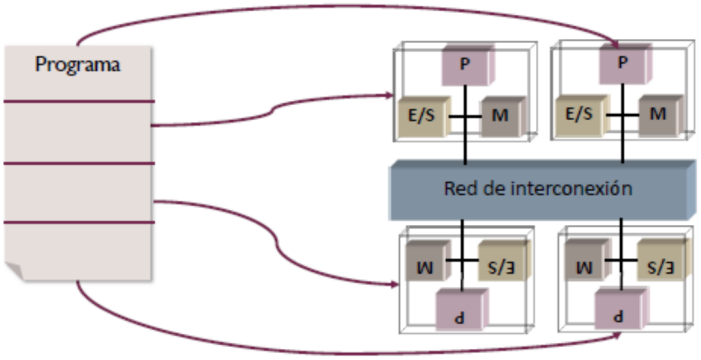
1.3 Modos de programación MIMD

Podemos distinguir dos tipos de programación:

- **SPMD (Single-Program Multiple Data / Paralelismo de datos)**: Todos los códigos que se ejecutan en paralelo se obtienen compilando el mismo programa. Cada una de las copias trabaja con un conjunto de datos distinto, y se ejecuta en un procesador diferente.



- **MPMD (Multiple-Program Multiple Data / Paralelismo de tareas o funciones)**: Los códigos que se ejecutan en paralelo se obtienen compilando programas independientes. En este caso la aplicación a ejecutar (o el código secuencial inicial) se divide en unidades independientes y cada una de ellas trabaja con un conjunto de datos y se asigna a un procesador distinto.



2. Herramientas para obtener código paralelo

2.1. Nivel de abstracción en que sitúan al programador las herramientas

Mientras mayor sea el nivel de abstracción en el que sitúe la herramienta de programación al programador, menos labores tendrá que realizar éste.

Las herramientas de programación paralela se puedes clasificar teniendo en cuenta el nivel de abstracción en que sitúa al programador.

Podríamos clasificarlas de la siguiente manera (de menor a mayor nivel de abstracción):

- **Compiladores paralelos**: Generan el código paralelo a partir de un código secuencial, automáticamente.
- **Lenguajes paralelos** (Occam, Ada, Java) y API funciones + Directivas (OpenMP): Los lenguajes paralelos tienen construcciones particulares y bibliotecas de funciones, que requieren un compilador exclusivo. Las API en este nivel constan de un conjunto de directivas y funciones que se añaden a un compilador de un lenguaje secuencial. En este nivel el programador tiene que detectar el paralelismo implícito.
- **API de funciones**: Las API en este nivel consisten en una biblioteca de funciones que se añade a un compilador de un lenguaje secuencial habitual. El programador debe realizar la asignación de tareas.

Las herramientas para obtener programas paralelos permiten de forma explícita (el trabajo lo haría el programador) o de forma implícita (el trabajo lo haría la propia herramienta) la realización de las siguientes tareas:

- Localizar el paralelismo o descomponer en tareas independientes (*descomposition*)
- Asignar las tareas (carga de trabajo, código+datos) a procesos/threads (*scheduling)*
- Crear y terminar procesos. (o enrolar y desenrolar)
- Comunicarlos y sincronizarlos

La asignación de los procesos a los procesadores (*mapping)* puede realizarla el programador, la herramienta o el SO

2.2 Bibliotecas de funciones

El programador puede usar para programación paralela, un lenguaje secuencial (como C o Fortran) y una biblioteca de funciones. Es decir, el cuerpo de los procesos se escribe con lenguaje secuencial y para crear o gestionar procesos e implementar la comunicación y sincronización se usan funciones de la biblioteca.

Algunas de estas bibliotecas son **Pthread**, **MPI** o **OpenMP**

Ejemplo de código usando la biblioteca OpenMP:

```
#include <omp.h>
#define NUM_THREADS 4
main(int argc, char **argv)  {
  double ancho,x, sum=0;  int intervalos, i;
  intervalos = atoi(argv[1]);
  ancho = 1.0/(double) intervalos;
  omp_set_num_threads(NUM_THREADS);        → Crear y Terminar
#pragma omp parallel
{                                          → Comunicar y sincronizar
  #pragma omp for reduction(+:sum) private(x) \
                  schedule(dynamic)        → Agrupar/Asigna
  for (i=0;i< intervalos; i++) {
      x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
  }
}
  sum* = ancho;
}
```

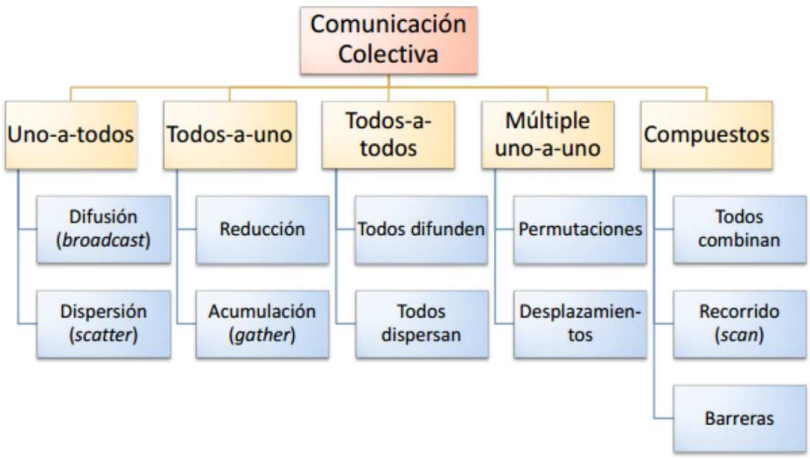
Localizar

2.3 Otras alternativas para comunicación

Las herramientas de programación paralela pueden ofrecer al programador, además de la comunicación entre dos procesos, comunicaciones en las que intervienen múltiples procesos. En algunos casos, estos servicios no se implementan solo para comunicar explícitamente datos, sino para sincronizar entre sí a componentes del grupo.

Usar estos esquemas de comunicación puede suponer un incremento en la eficiencia del programa paralelo.

Funciones colectivas:

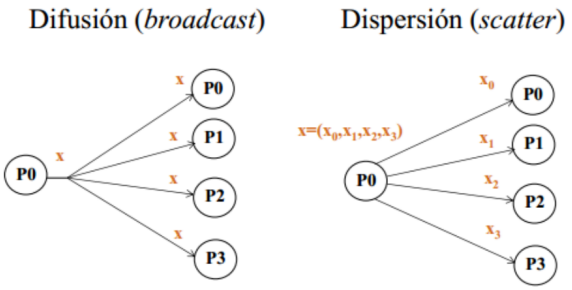


- **Uno-a-todos**

  Un proceso envía, todos reciben.

  - Difusión (*brodcast*):Todos los procesos reciben el mismo mensaje

  - Dispersión (*scatter*):Cada proceso receptor recibe un mensaje diferente

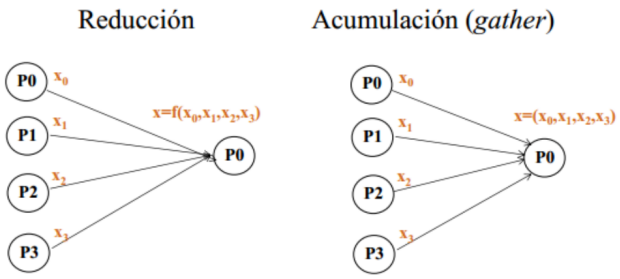


- **Todos-a-uno**

  Todos los procesos en el grupo envían un mensaje a un único proceso.

  - Reducción: Los mensajes enviados se combinan en uno solo mediante algún operador. Normalmente esta combinación es conmutativa y asociativa.

  - Acumulación(*gather*): Los mensajes se reciben de forma concatenada en el receptor en una estructura vectorial.

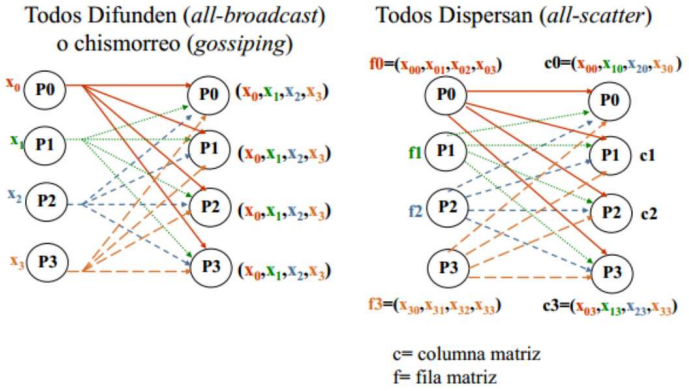


- **Todos-a-todos**

  Todos los procesos del grupo realizan una comunicación uno-a-todos. Cada proceso recibe *n* mensajes, cada uno de un proceso diferente.

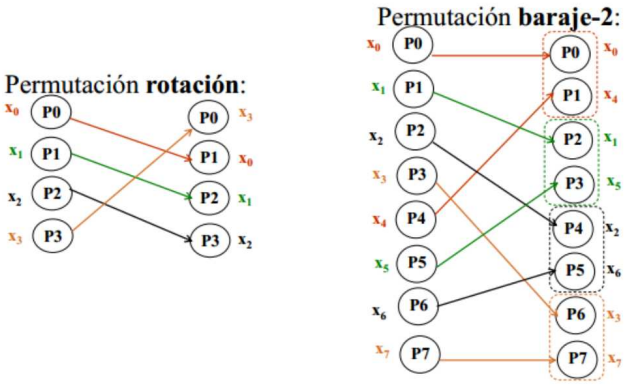
  - Todos-Difunden (*all-broadcast*) o chismorreo (*gossiping*): Todos los procesos difunden. Normalmente las *n* transferencias recibidas por un proceso se concatenan en función del identificador de proceso que envían, de forma que todos los procesos reciben lo mismo.

- Todos-Dispersan *(all-scatter)*: Los procesos concatenan diferentes transferencias.



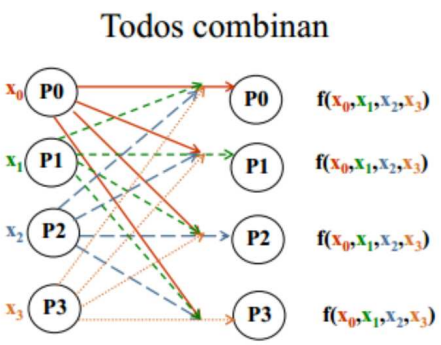
- **Múltiple uno-a-uno**

  Se caracteriza porque hay componentes del grupo que envían un único mensaje y componentes que reciben un único mensaje. Si todos los componentes del grupo envían y reciben, se llama permutación. En caso contrario, se llama desplazamiento.



- **Servicios compuestos**

  Hay servicios que resultan de una combinación de los anteriores



Ejemplos de comunicación colectiva en OpenMP

Uno-a-todos	Difusión (Seminario pract. 2)	✓ Cláusula `firstprivate` (desde thread 0) ✓ Directiva `single` con cláusula `copyprivate` ✓ Directiva `threadprivate` y uso de cláusula `copyin` en directiva `parallel` (desde thread 0)
Todos-a-uno	Reducción (Seminario pract. 2)	✓ Cláusula `reduction`
Servicios compuestos	Barreras (Seminario pract. 1)	✓ Directiva `barrier`

Ejemplos de comunicación colectiva en MPI

Uno-a-uno	Asíncrona	`MPI_Send()/MPI_Receive()`
Uno-a-todos	Difusión	`MPI_Bcast()`
	Dispersión	`MPI_Scatter()`
Todos-a-uno	Reducción	`MPI_Reduce()`
	Acumulación	`MPI_Gather()`
Todos-a-todos	Todos difunden	`MPI_Allgather()`
Servicios compuestos	Todos combinan	`MPI_Allreduce()`
	Barreras	`MPI_Barrier()`
	Scan	`MPI_Scan`

3. Estilos/paradigmas de programación paralela

Cada herramienta de programación paralela ofrece al programador un modelo de programación particular. Estos modelos se encuadran en uno de los siguientes paradigmas de programación: Paso de Mensajes (ej. MPI), variables compartidas (ej. OpenMP) o paralelismo de datos (ej. HPM).

- **Paso de mensajes**: Los flujos de instrucciones no comparten memoria, cada uno tiene su espacio de direcciones.
- **Variables compartidas:** Los flujos de instrucciones comparten memoria, por lo que puede haber comunicación de flujos por medio de variables
- **Paralelismo de datos:** Todos los flujos deben ejecutar a la vez la misma instrucción pero con distintos datos.

3.1 Estilos y arquitecturas

Paso de mensajes -> Multicomputadores

Variables compartidas -> Multiprocesadores

Paralelismo de datos -> Procesadores matriciales, GPU

3.2 Estilos y herramientas de programación

Paso de mensajes -> Lenguajes: Ada y Occam

API (Biblioteca de funciones): MPI, PVM

Variables compartidas -> Lenguajes: Ada y Java

API (directivas del compilador + funciones): OpenMP

API (Bibliotecas de funciones): POSIX Threads, Intel TBB…

Paralelismo de datos-> Lenguajes: HPF y Fortran 95

API (funciones): Nvidia CUDA

4. Estructuras típicas de procesos/threads en códigos paralelos

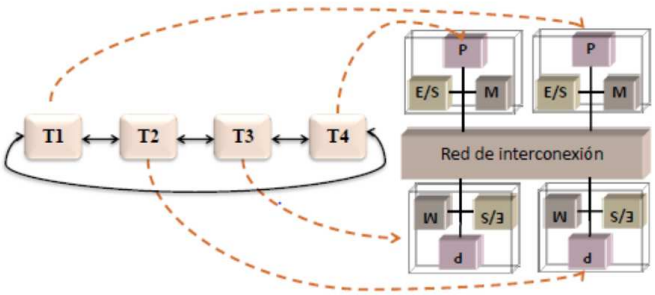
Descomposición de dominio o de datos

El trabajo a realizar por cada proceso se determina dividiendo las estructuras de datos de entrada o de salida (o ambas) en tantos trozos como flujos.

Generalmente, las tareas realizan operaciones similares, aunque hay descomposiciones en las que las tareas no realizan exactamente las mismas operaciones.

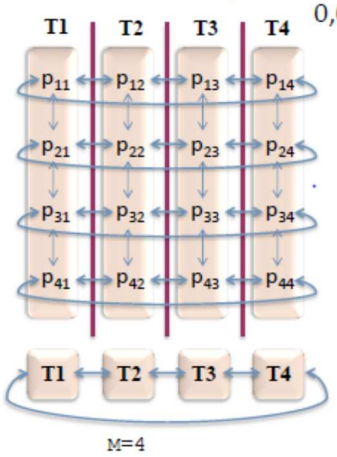
La descomposición de datos es útil para resolver algoritmos con imágenes

Es frecuente encontrar combinaciones de master-slave con descomposición de dominio.



Ejemplo:

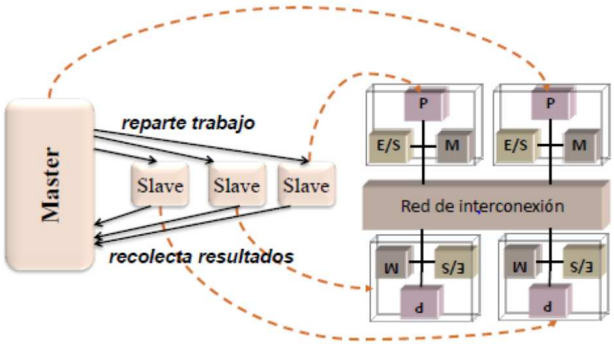
$$p_{ij}(t) = 0{,}75 \times p_{ij}(t-1) + 0{,}0625 \times p_{i-1j}(t-1) + 0{,}0625 \times p_{ij-1}(t-1) + 0{,}0625 \times p_{i+1j}(t-1) + 0{,}0625 \times p_{ij+1}(t-1)$$




```
#include <omp.h>
…
omp_set_num_threads(M)
#pragma omp parallel private(i)
{
  for (i=0;i<N;i++)     //recorre filas
    #pragma omp for
    for (j=0;j<M;j++) //recorre columnas
      pS[i,j] = 0,75*p[i,j] +
        0,0625*(p[i-1,j]+p[i,j-1]+
                p[i+1,j]+p[i,j+1]);

}
…
```

Dueño-Esclavo



El proceso dueño se encarga de distribuir las tareas a un conjunto (granja) de procesos esclavos, y de ir recolectando los resultados finales que van calculando los esclavos

No suele haber comunicación entre esclavos.

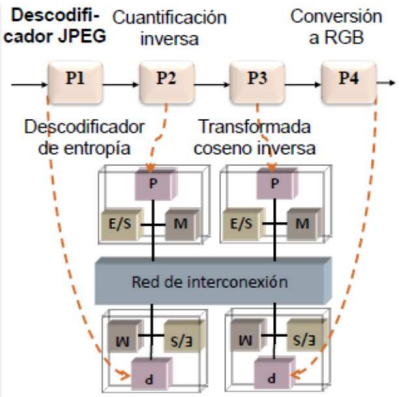
La distribución de tareas a los esclavos puede hacerse tanto de forma dinámica (durante la ejecución) o de forma estática (por el programador). En este último caso, se sabe la tarea que va a ejecutar cada esclavo.

Estructura segmentada o flujo de datos

Se utiliza cuando se aplica a un flujo de datos de entrada las mismas funciones en secuencia. Por tanto, cada proceso ejecuta distinto código.

Ejemplo: **descodificador JPEG**

```
#include <omp.h>
...
omp_set_num_threads(4);
...
#pragma omp parallel
{...
 for (i=0;i<N;i++) {
   ...
  #pragma omp sections
  {
   #pragma omp section
     P1();
   #pragma omp section
     P2();
   #pragma omp section
     P3();
   #pragma omp section
     P4();
  } ...
 }
...
} ...
```



Divide y Vencerás o descomposición recursiva

Se usa cuando el problema a resolver se puede dividir en subproblemas que son instancias más pequeñas del problema original, de forma que se obtiene el resultado final mediante la combinación de los resultados de dichos subproblemas.