

2º curso / 2º cuatr.

Grado en
Ing. Informática

Arquitectura de Computadores

Tema 2

Programación paralela

Material elaborado por los profesores responsables de la asignatura:
Mancia Anguita – Julio Ortega

Licencia Creative Commons



ugr

Universidad
de Granada



Lecciones

- Lección 4. Herramientas, estilos y estructuras en programación paralela
 - Problemas que plantea la programación paralela al programador. Punto de partida
 - Herramientas para obtener código paralelo
 - Estilos/paradigmas de programación paralela
 - Estructuras típicas de códigos paralelos
- Lección 5. Proceso de parallelización
- Lección 6. Evaluación de prestaciones en procesamiento paralelo

Objetivos Lección 4

- Distinguir entre los diferentes tipos de herramientas de programación paralela: compiladores paralelos, lenguajes paralelos, API Directivas y API de funciones.
- Distinguir entre los diferentes tipos de comunicaciones colectivas.
- Diferenciar el estilo/paradigma de programación de paso de mensajes del de variables compartidas.
- Diferenciar entre OpenMP y MPI en cuanto a su estilo de programación y tipo de herramienta.
- Distinguir entre las estructuras de tareas/procesos/treads master-slave, cliente-servidor, descomposición de dominio, flujo de datos o segmentación, y divide y vencerás.

Bibliografía Lección 4

➤ Fundamental

- Capítulo 2, Secciones 1-3. M. Anguita, J. Ortega. Fundamentos y problemas de Arquitectura de Computadores, Editorial Técnica Avicam. ESIIT/C.1 ANG fun
- Capítulo 7. Sección 7.4. J. Ortega, M. Anguita, A. Prieto. “Arquitectura de Computadores”. ESII/C.1 ORT arq

➤ Complementaria

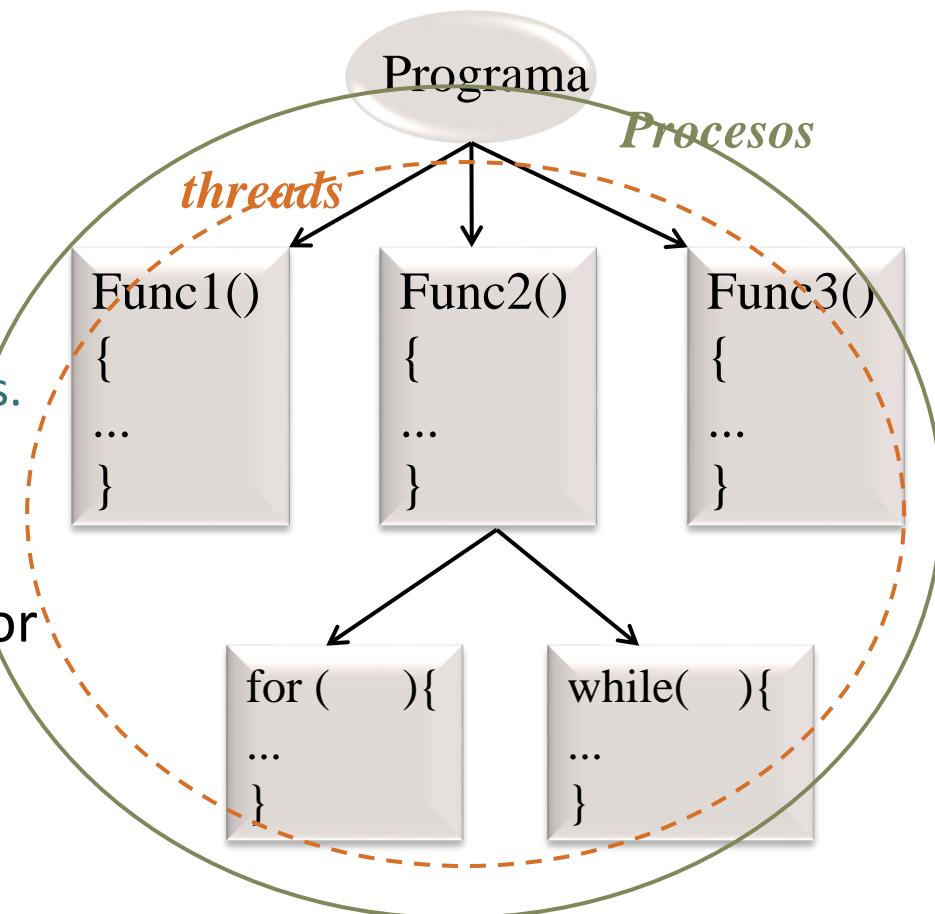
- Thomas Rauber, Gudula Rünger. “Parallel Programming: for Multicore and Cluster Systems.” Springer, 2010. Disponible en línea (biblioteca UGR): <http://dx.doi.org/10.1007/978-3-642-04818-0>
- Barry Wilkinson.“Parallel programming : techniques and applications using networked workstations and parallel computer”, 2005. ESIIT/D.1 WIL par

Contenido Lección 4

- Problemas que plantea la programación paralela al programador. Punto de partida
- Herramientas para obtener código paralelo
- Estilos/paradigmas de programación paralela
- Estructuras típicas de códigos paralelos

Problemas que plantea la programación paralela

- Nuevos problemas, respecto a programación secuencial:
 - División en unidades de cómputo independientes (tareas).
 - Agrupación/asignación de tareas o carga de trabajo (código, datos) en procesos/threads.
 - Asignación a procesadores/núcleos.
 - Sincronización y comunicación.
- Los debe abordar la herramienta de programación o el programador o SO

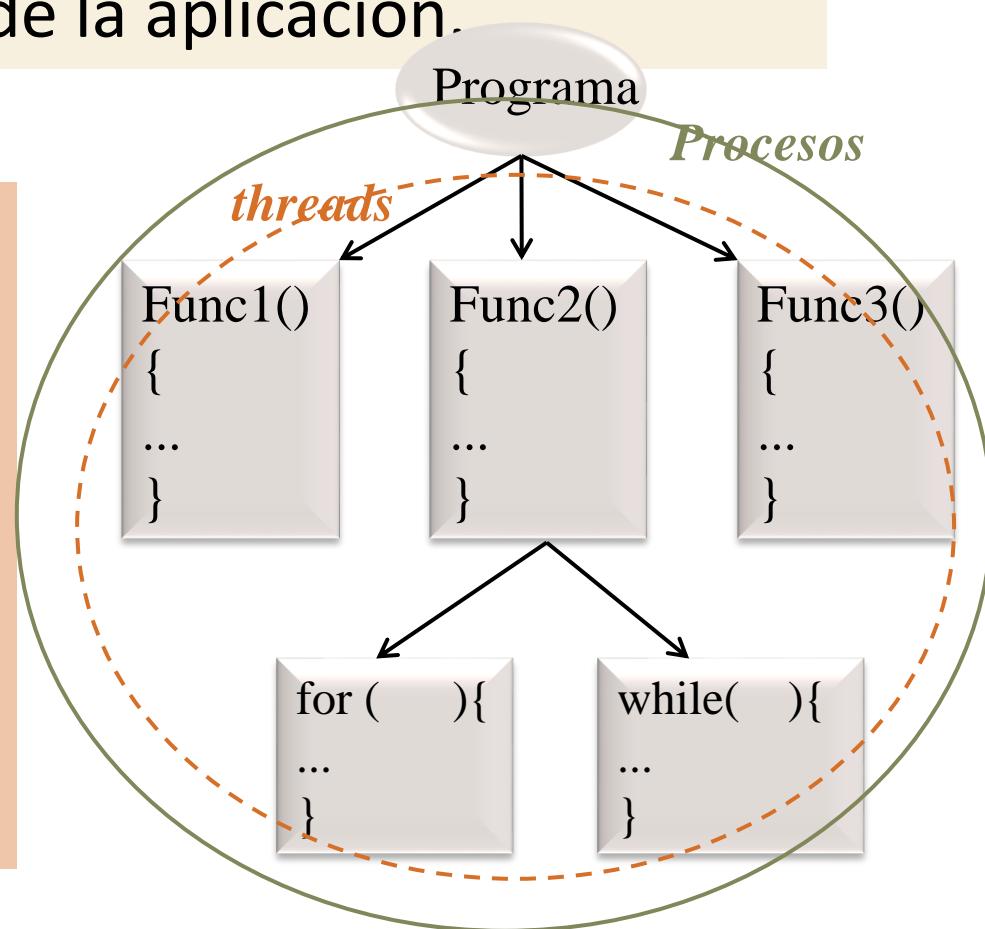


Punto de partida

- Partir de una versión secuencial.
- Descripción o definición de la aplicación.

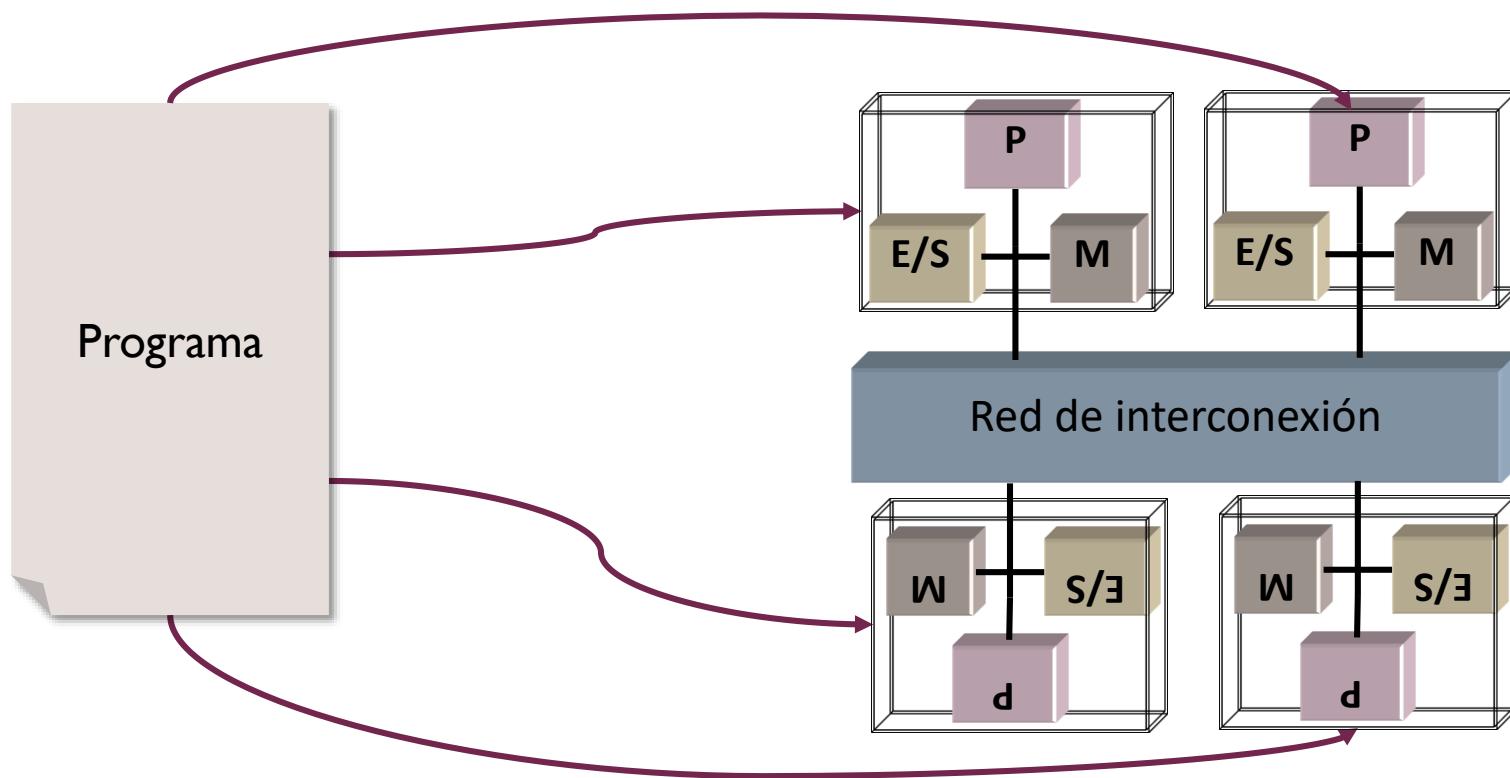
Apoyo:

- Programa paralelo que resuelva un problema parecido.
- Versiones paralelas u optimizadas de bibliotecas de funciones:
BLAS (*Basic Linear Algebra Subroutine*),
LAPACK (*Linear Algebra PACKage*),
...



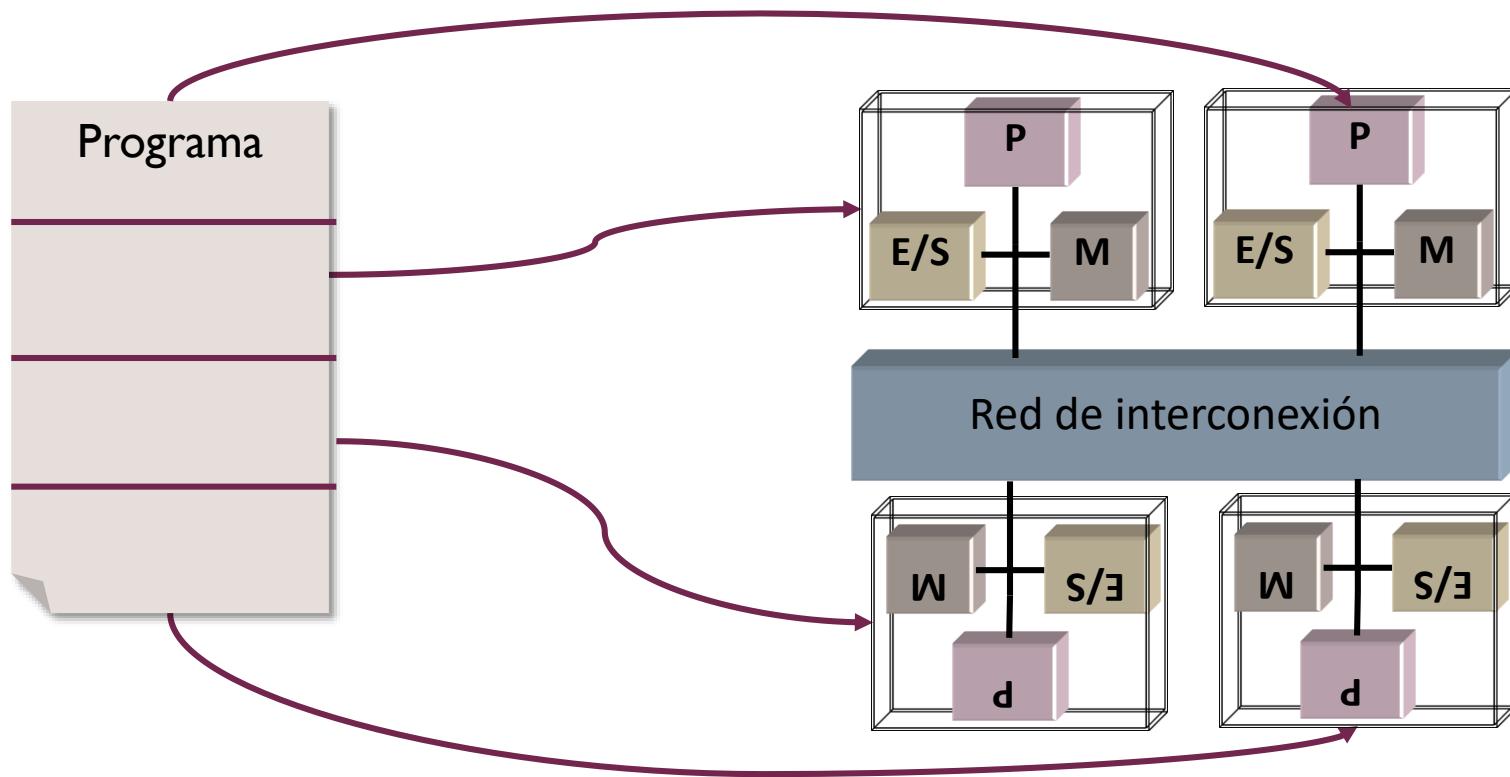
Modos de programación MIMD

- SPMD (*Single-Program Multiple Data*)



Modos de programación MIMD

- MPMD (*Multiple-Program Multiple Data*)



Contenido Lección 4

- Problemas que plantea la programación paralela al programador. Punto de partida
- Herramientas para obtener código paralelo
- Estilos/paradigmas de programación paralela
- Estructuras típicas de códigos paralelos

Herramientas de programación paralela

Compiladores paralelos (paralelización automática)

Extracción automática del paralelismo



Lenguajes paralelos (Occam, Ada, Java) y API funciones + Directivas (OpenMP)

Construcciones del lenguaje + funciones

Lenguaje secuencial + (directivas + funciones)



API funciones (Pthreads, MPI)

Lenguaje secuencial + funciones

Herramientas para obtener programas paralelos

- Las herramientas permiten de forma implícita o explícita (lo hace el programador):
 - **Localizar** parallelismo o descomponer en tareas independientes (*decomposition*)
 - **Asignar** las tareas, es decir, la carga de trabajo (código + datos), a procesos/threads (*scheduling*)
 - **Crear y terminar** procesos/threads (o enrolar y desenrolar en un grupo)
 - **Comunicar y sincronizar** procesos/threads
- El programador, la herramienta o el SO se encarga de
 - **Asignar** procesos/threads a unidades de procesamiento (*mapping*)

Ejemplo: cálculo de PI con OpenMP/C

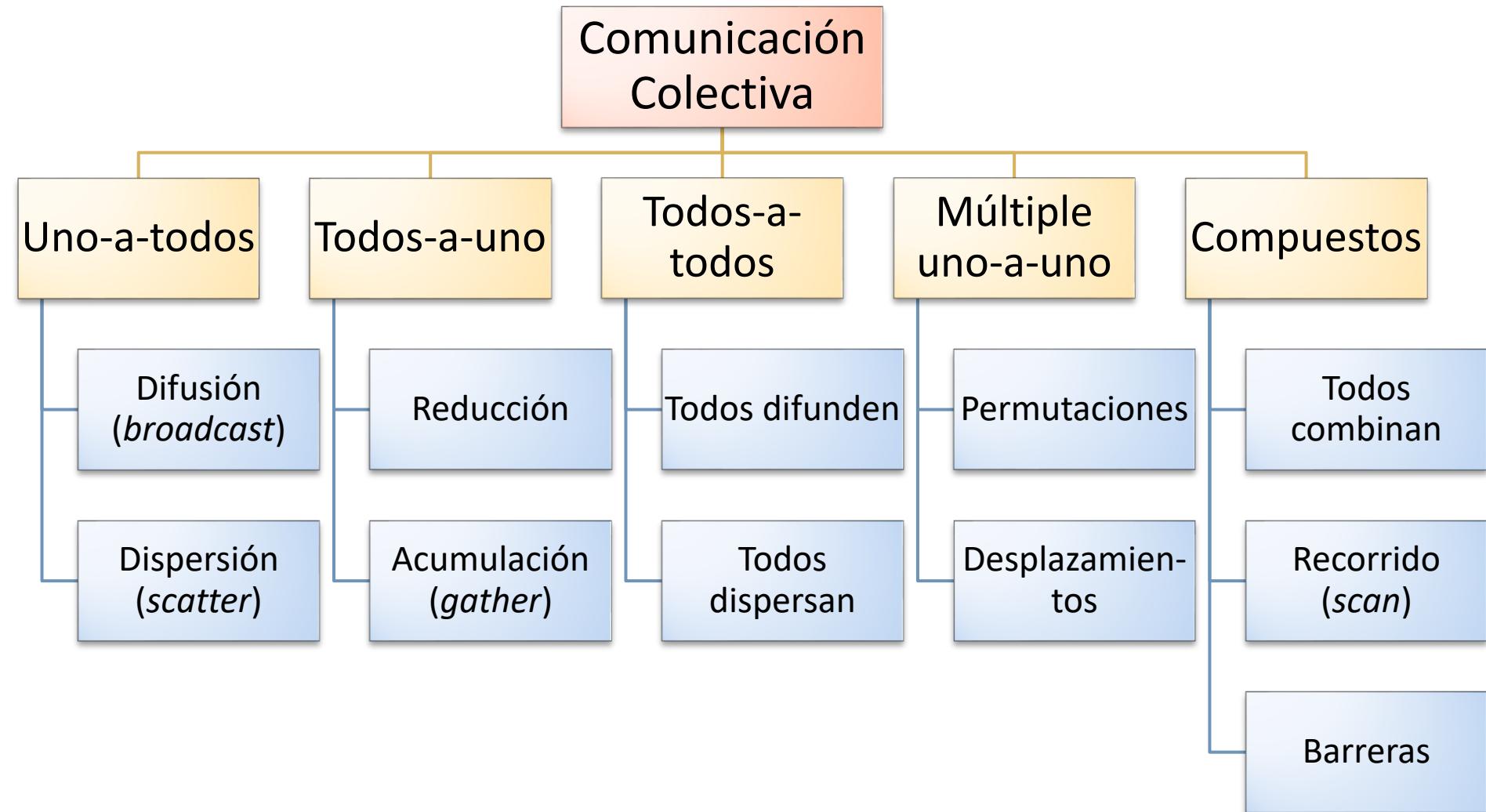
```
#include <omp.h>
#define NUM_THREADS 4
main(int argc, char **argv) {
    double ancho,x, sum=0;    int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    omp_set_num_threads(NUM_THREADS); →Crear y Terminar
#pragma omp parallel →Comunicar y sincronizar
{
    #pragma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

Localizar y Asignar

Ejemplo: cálculo de PI en MPI/C

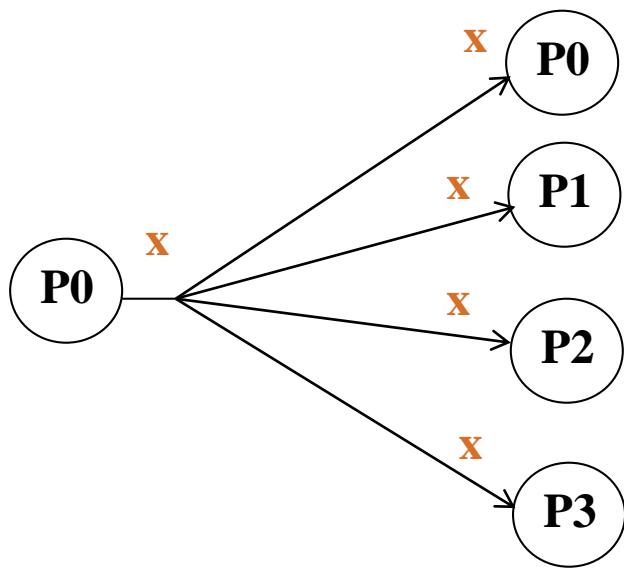
```
#include <mpi.h>
main(int argc, char **argv)  {
    double ancho,x,sum,lsum; int intervalos,i,nproc,iproc;
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);                                →Enrolar
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    intervalos=atoi(argv[1]);                                              →Localizar y Asignar
    ancho=1.0/(double) intervalos; lsum=0;
    for (i=iproc; i<intervalos; i+=nproc) {
        x = (i+0.5)*ancho; lsum+= 4.0/(1.0+x*x);
    }
    lsum*= ancho;                                                        →Comunicar/sincronizar
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
               MPI_SUM,0,MPI_COMM_WORLD);
    MPI_Finalize();                                                 →Desenrolar
}
```

Comunicaciones colectivas

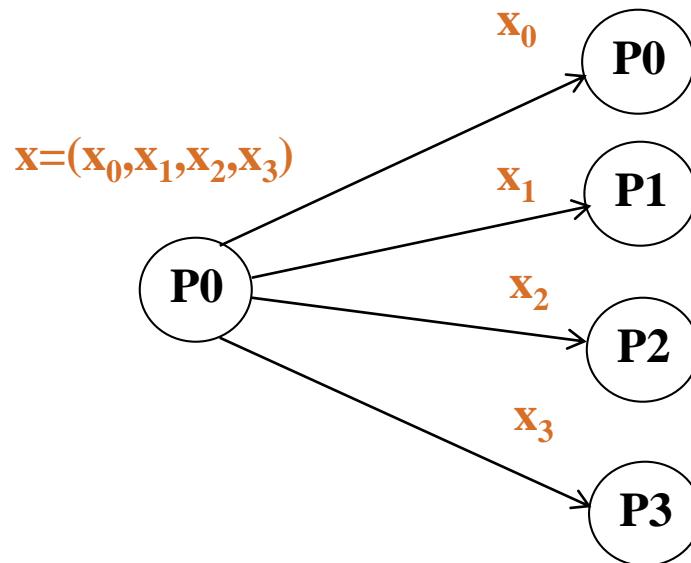


Comunicación uno-a-todos

Difusión (*broadcast*)

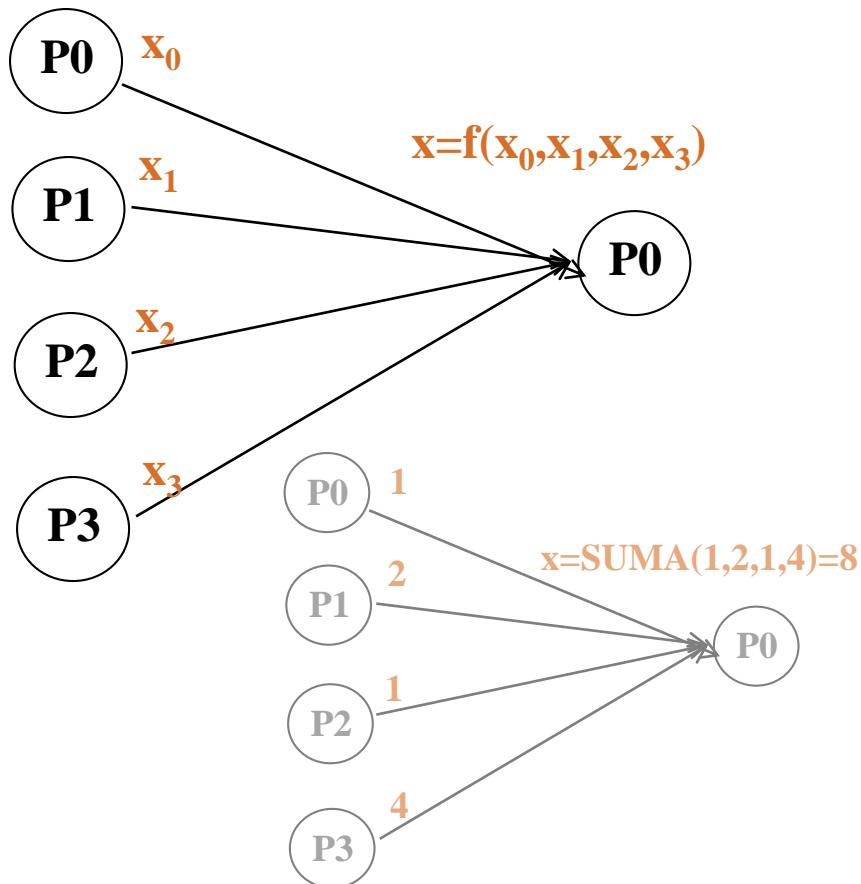


Dispersión (*scatter*)

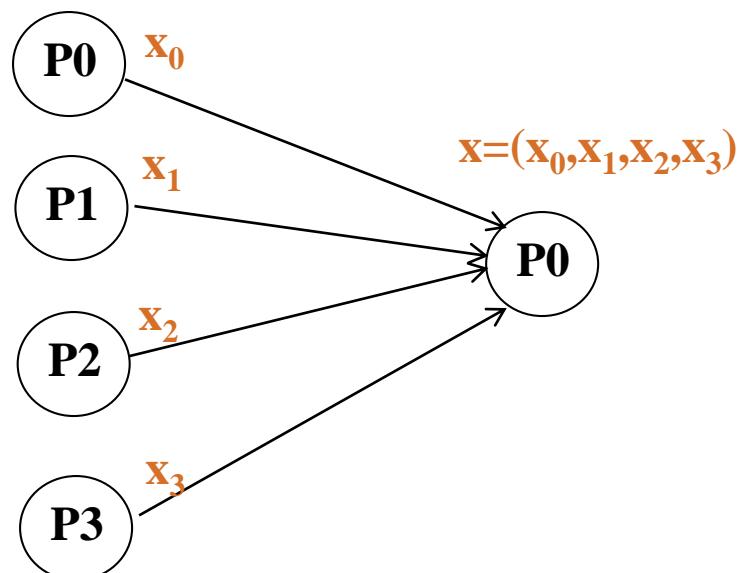


Comunicación todos-a-uno

Reducción

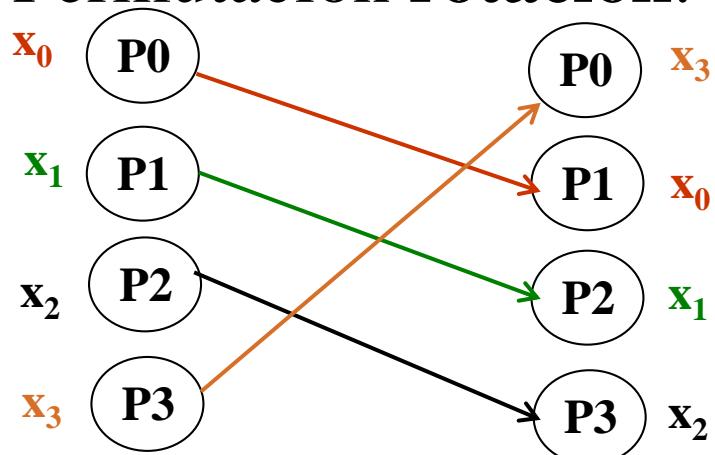


Acumulación (*gather*)

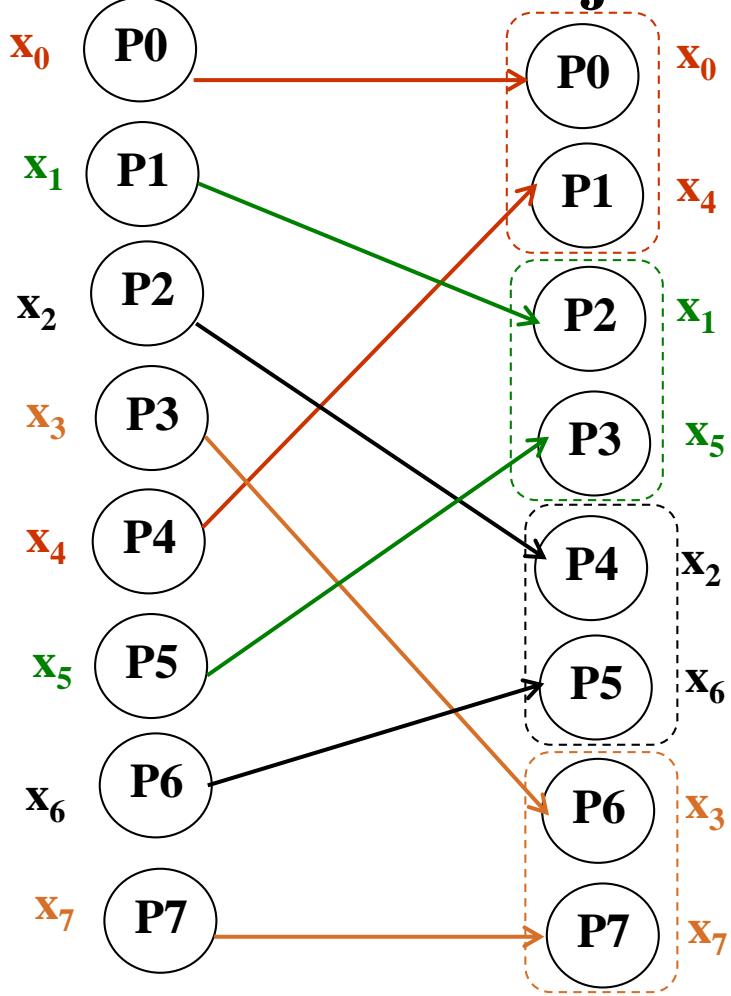


Comunicación múltiple uno-a-uno

Permutación rotación:

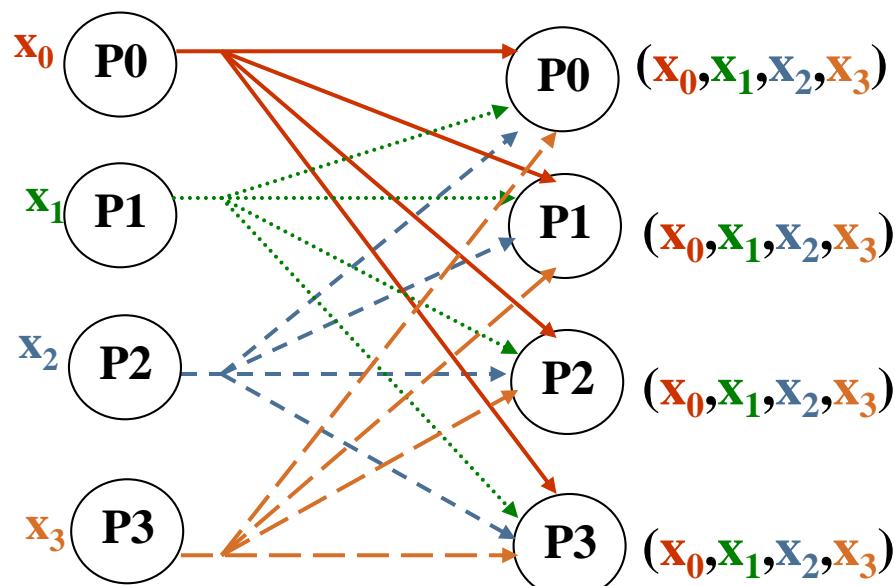


Permutación baraje-2:

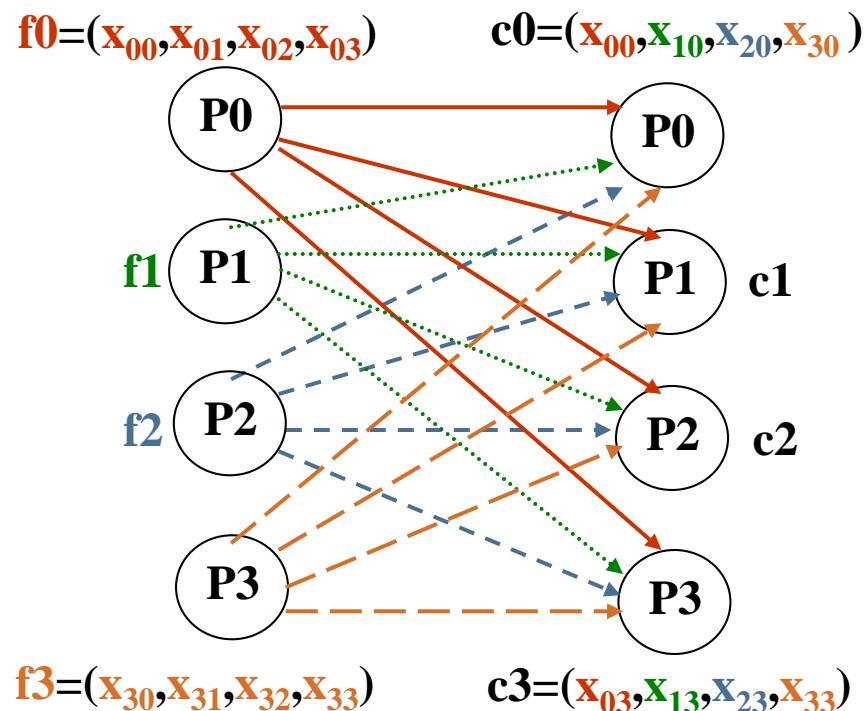


Comunicación todos-a-todos

Todos Difunden (*all-broadcast*)
o chismorreo (*gossiping*)



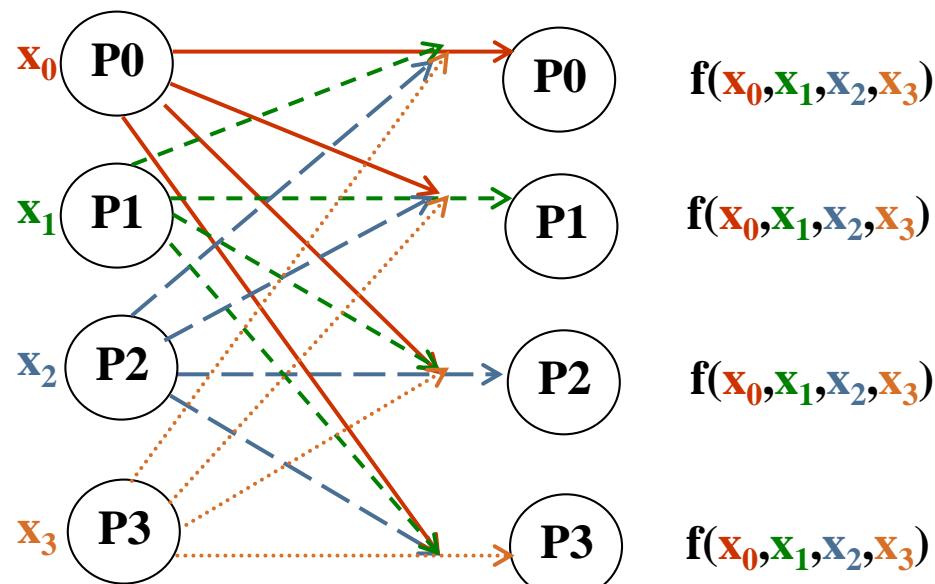
Todos Dispersan (*all-scatter*)



c= columna matriz
f= fila matriz

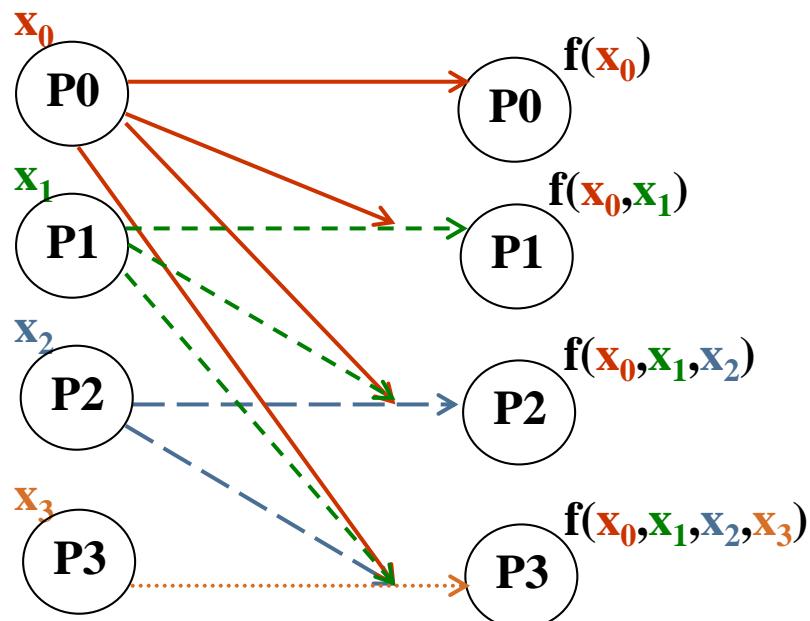
Servicios compuestos

Todos combinan

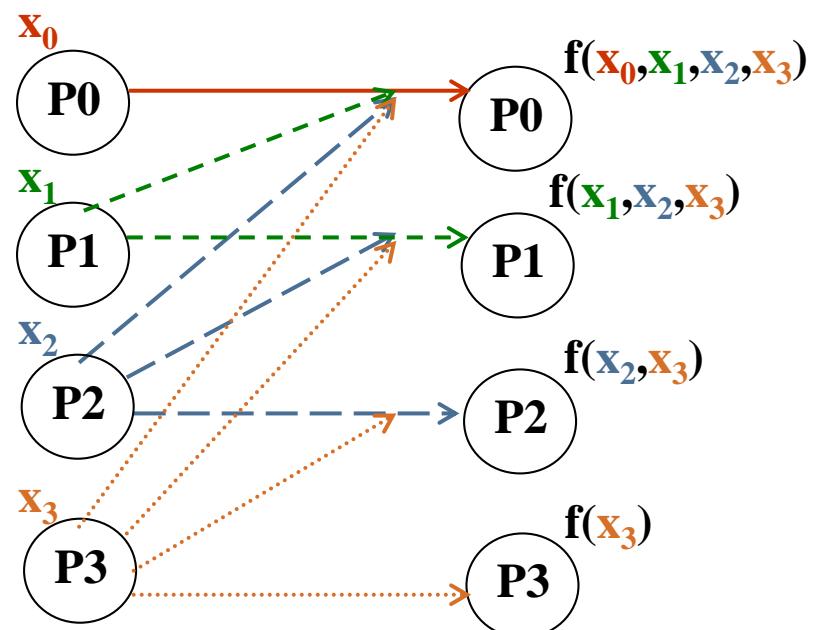


Servicios compuestos

Recorrido (scan) prefijo paralelo



Recorrido sufijo paralelo



Ejemplo: comunicación colectiva en OpenMP

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

Ejemplo: comunicación en MPI

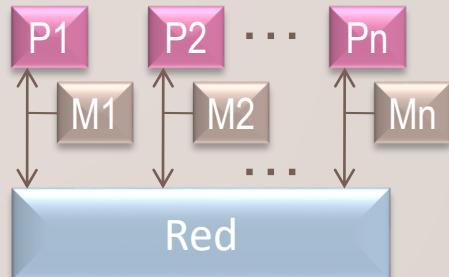
Uno-a-uno	Asíncrona	<code>MPI_Send() /MPI_Receive()</code>
Uno-a-todos	Difusión	<code>MPI_Bcast()</code>
	Dispersión	<code>MPI_Scatter()</code>
Todos-a-uno	Reducción	<code>MPI_Reduce()</code>
	Acumulación	<code>MPI_Gather()</code>
Todos-a-todos	Todos difunden	<code>MPI_Allgather()</code>
Servicios compuestos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

Detalles de la programación con MPI en la asignatura: Arquitecturas y Computación de Altas Prestaciones
(IC.SCAP.ACAP – Especialidad (IC), Materia (SCAP), Asignatura (ACAP))

Contenido Lección 4

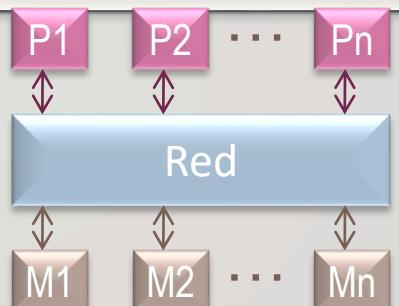
- Problemas que plantea la programación paralela al programador. Punto de partida
- Herramientas para obtener código paralelo
- Estilos/paradigmas de programación paralela
- Estructuras típicas de códigos paralelos

Estilos de programación y arquitecturas paralelas



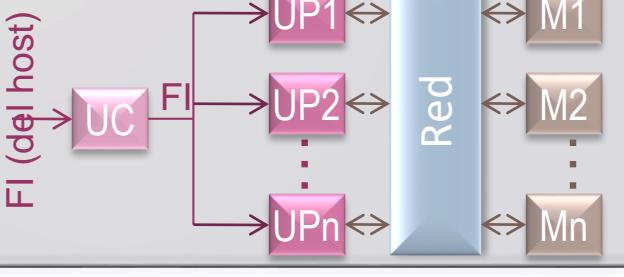
Paso de mensajes

- Multicomputadores



Variables compartidas

- Multiprocesadores



Paralelismo de datos

- Procesadores matriciales, GPU (*stream processing*)

Estilos de programación y herramientas de programación

- Paso de mensajes (*message passing*)
 - Lenguajes de programación: Ada, Occam
 - API (Bibliotecas de funciones): MPI, PVM
- Variables compartidas (*shared memory, shared variables*)
 - Lenguajes de programación: Ada, Java
 - API (directivas del compilador + funciones): **OpenMP**
 - API (Bibliotecas de funciones): POSIX Threads, shmem, Intel TBB (*Threading Building Blocks*), C++ con clases thread, mutex, atomic ...
- Paralelismo de datos (*data parallelism*)
 - Lenguajes de programación + funciones: HPF (*High Performance Fortran*), Fortran 95 (`forall`, operaciones con matrices/vectores), Nvidia CUDA
 - API (directivas del compilador + funciones - *stream processing*): OpenACC

Contenido Lección 4

- Problemas que plantea la programación paralela al programador. Punto de partida
- Herramientas para obtener código paralelo
- Estilos/paradigmas de programación paralela
- Estructuras típicas de códigos paralelos

Estructuras típicas de procesos/threads/tareas

Estructuras típicas de procesos/threads/tareas en código paralelo

Descomposición de dominio o descomposición de datos

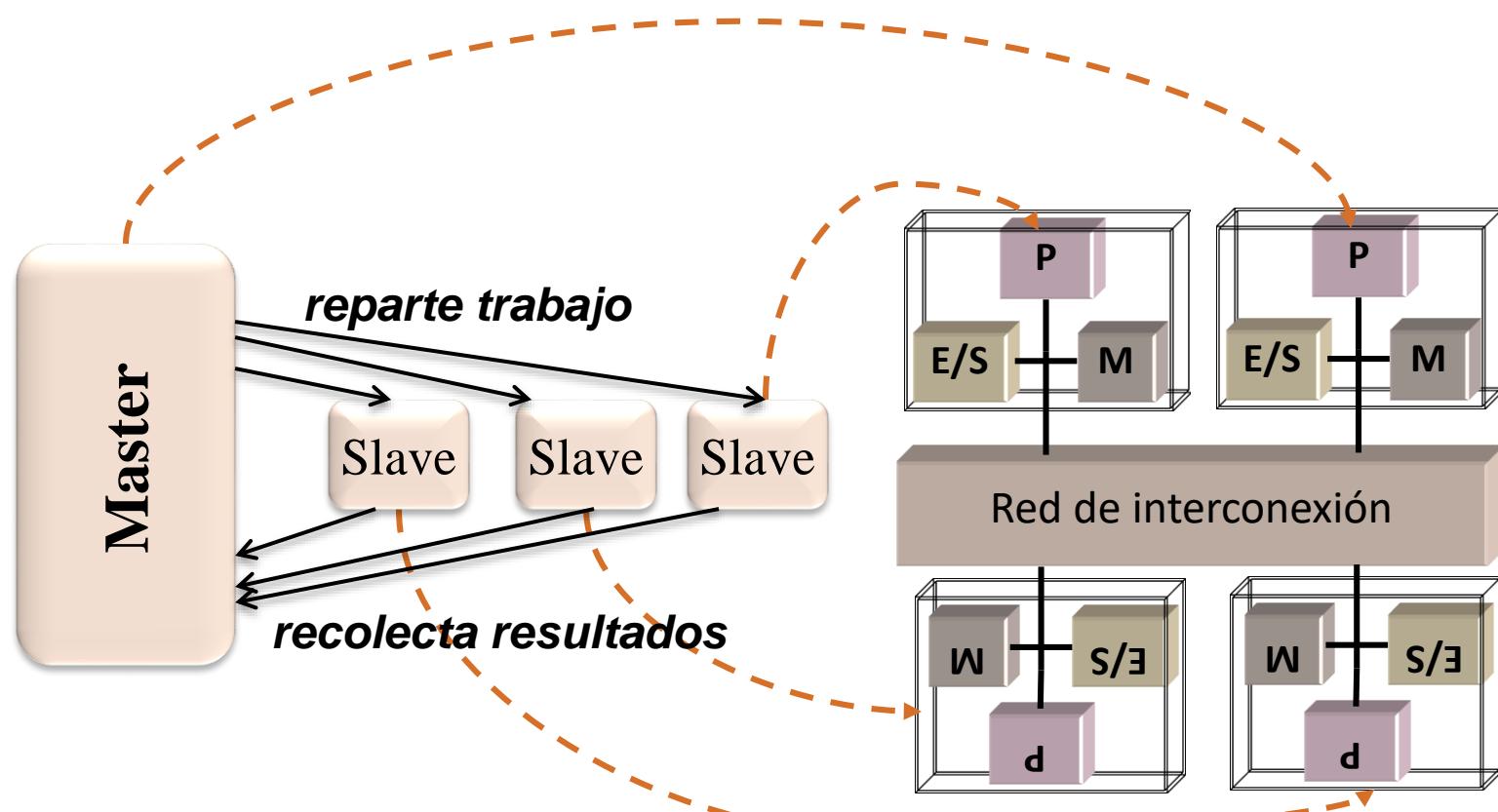
cliente/servidor

Divide y vencerás o descomposición recursiva

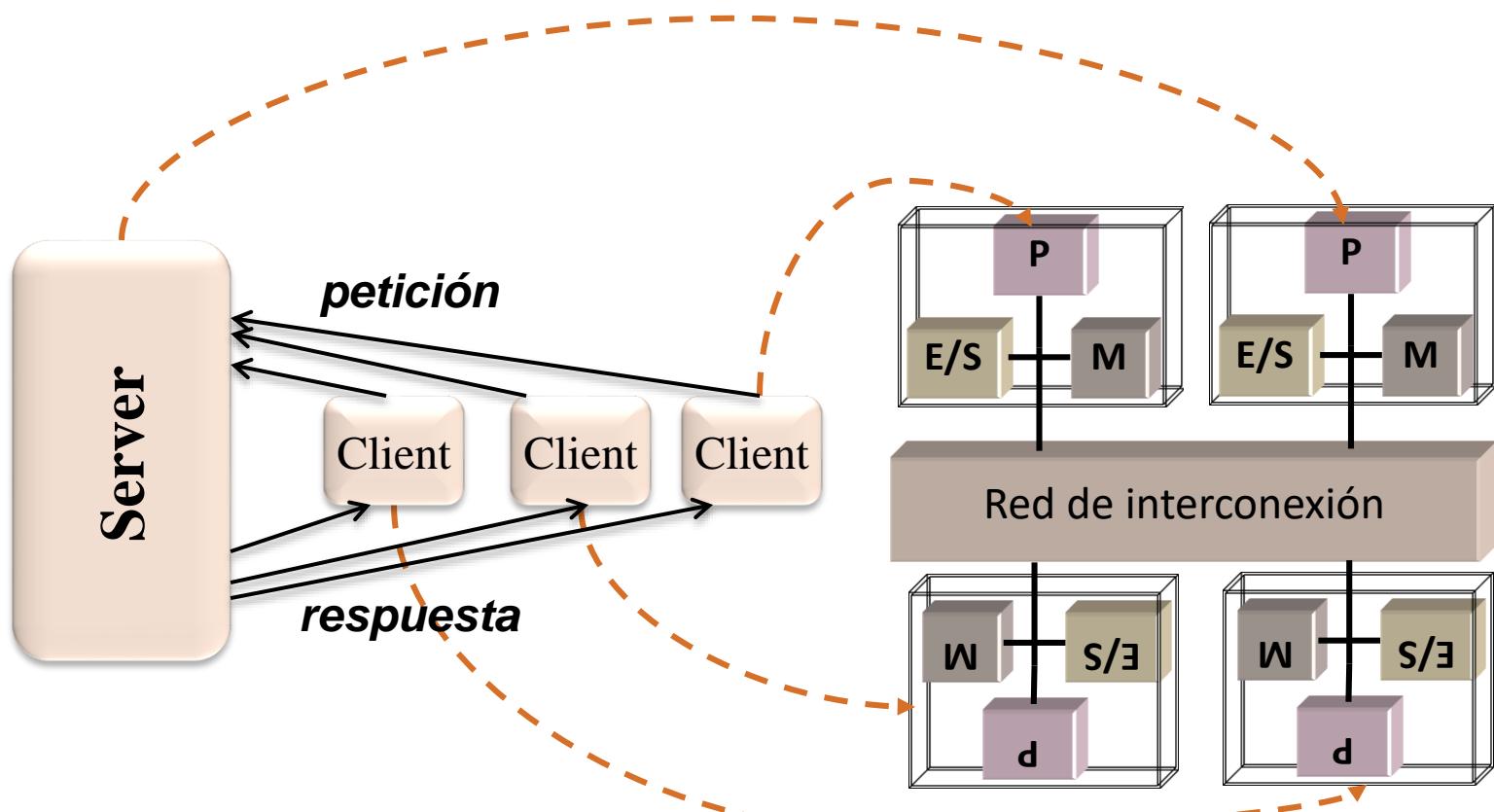
Segmentación o flujo de datos

Master-Slave, o granja de tareas

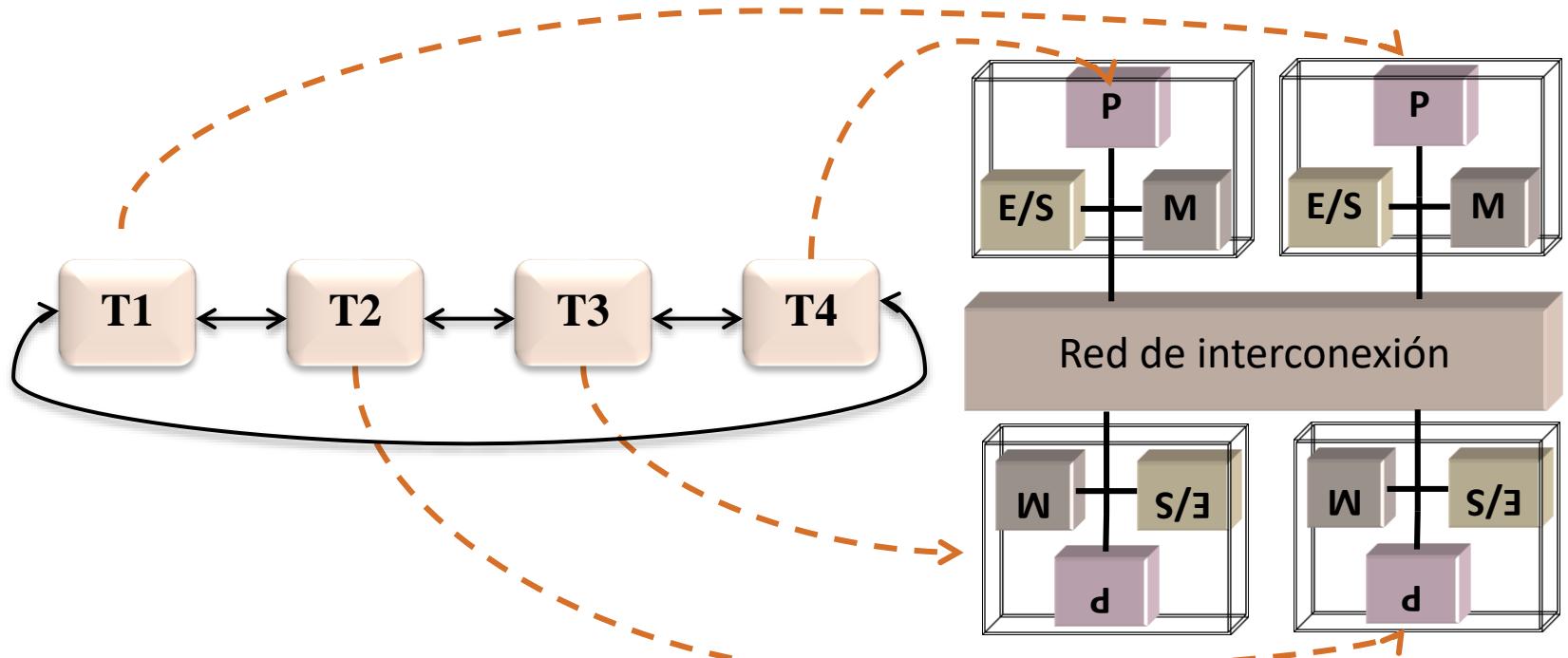
Master-Slave o granja de tareas



Cliente-Servidor

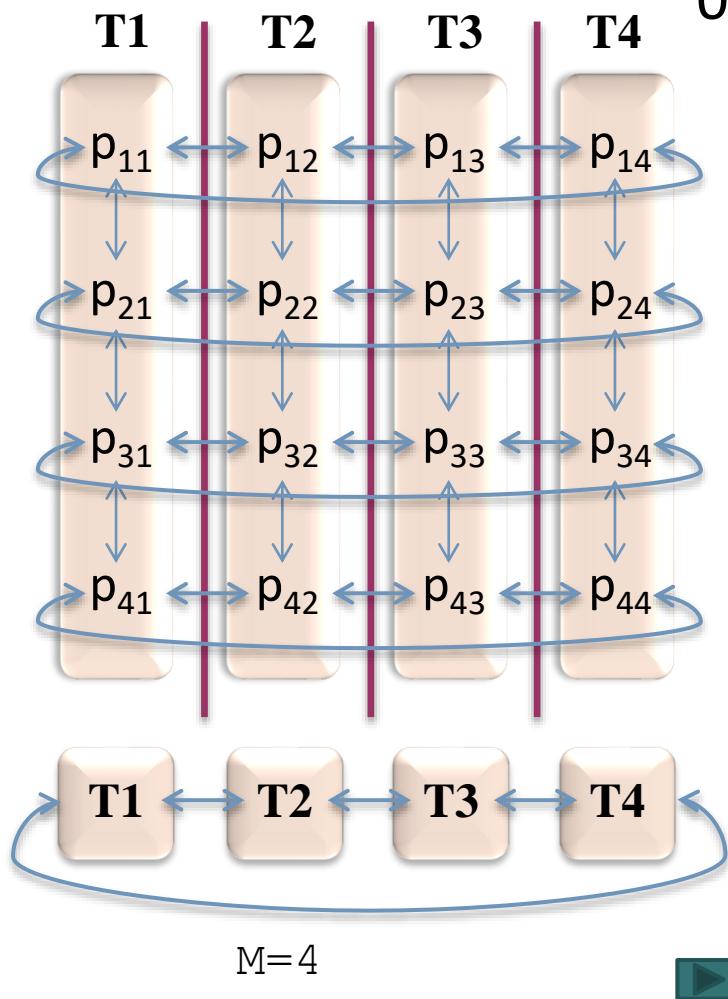


Descomposición de dominio o descomposición de datos I



Descomposición de dominio II. Ej: filtrado imagen

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

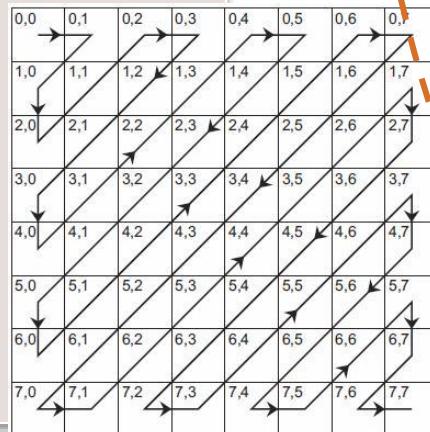


```
#include <omp.h>
...
omp_set_num_threads (M)
#pragma omp parallel private(i)
{
    ...
#pragma omp barrier
    ...
    for (i=0;i<N;i++) //recorre filas
        #pragma omp for schedule(static)
        for (j=0;j<M;j++) //recorre columnas
            ps[i,j] = 0,75*p[i,j] +
                0,0625*(p[i-1,j]+p[i,j-1]+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

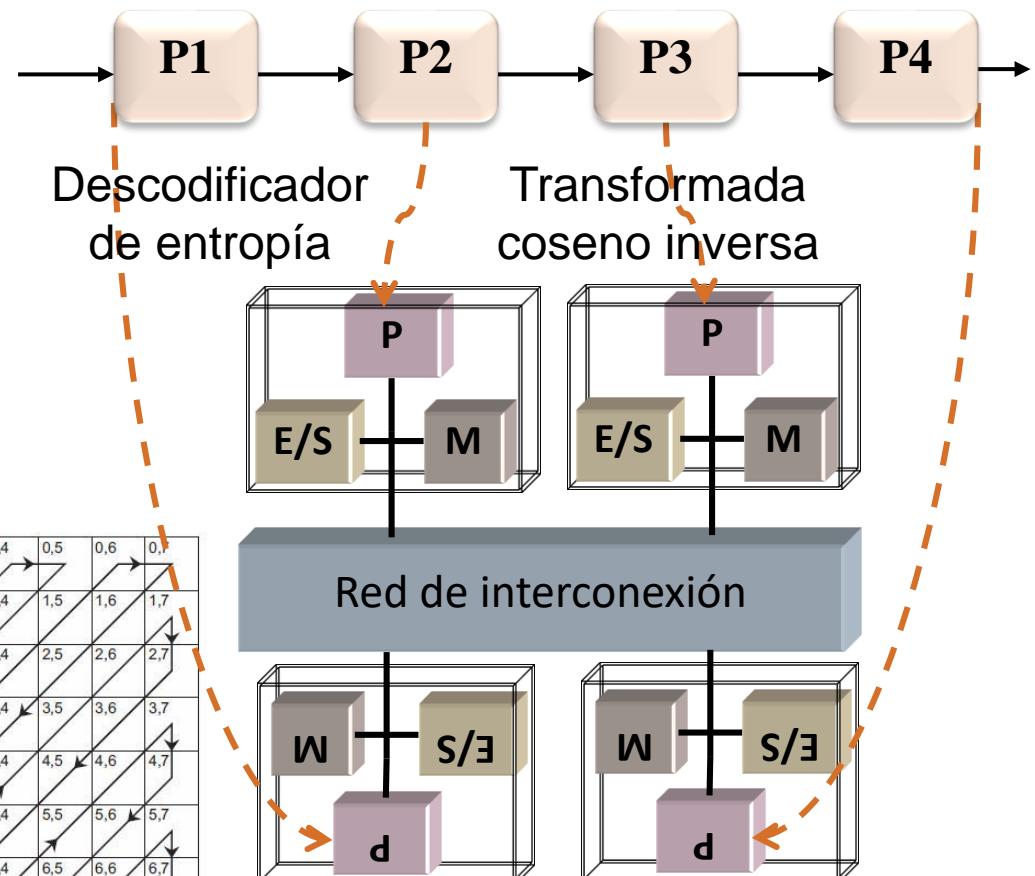


Estructura segmentada o de flujo de datos

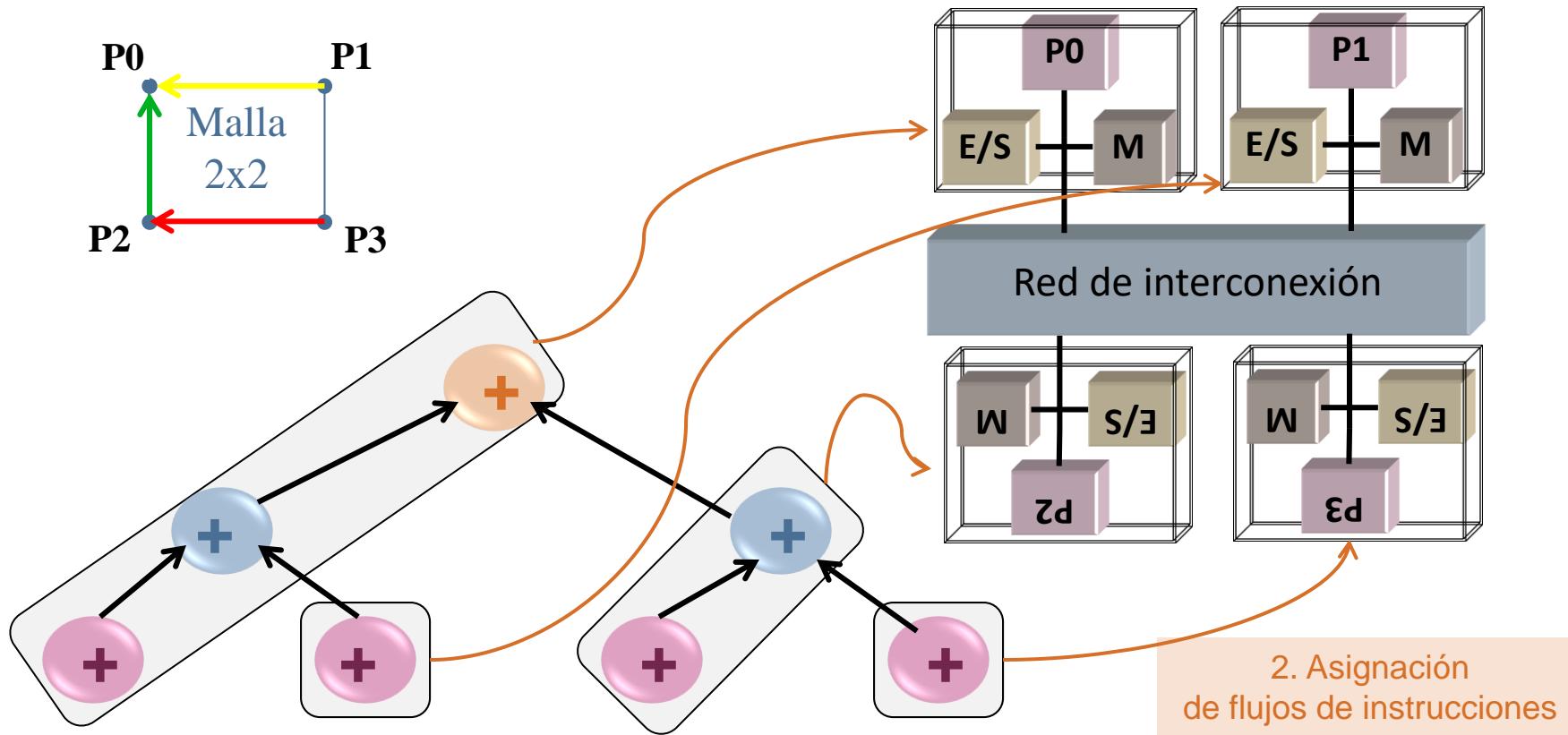
```
#include <omp.h>
...
omp_set_num_threads(4);
...
#pragma omp parallel
{
    ...
    for (i=0;i<N;i++) {
        ...
        #pragma omp sections
        {
            #pragma omp section
            P1();
            #pragma omp section
            P2();
            #pragma omp section
         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



Descodificador JPEG Cuantificación inversa Conversión a RGB



Divide y vencerás o descomposición recursiva



$$(((a_0+a_1) + (a_2+a_3)) + ((a_4+a_5) + (a_6+a_7)))$$

2. Asignación
de flujos de instrucciones
a procesadores

1. Agrupación/Asignación
de tareas a
flujos de instrucciones

2º curso / 2º cuatr.

Grado en
Ing. Informática

Arquitectura de Computadores

Tema 2

Programación paralela

Material elaborado por los profesores responsables de la asignatura:
Mancia Anguita – Julio Ortega

Licencia Creative Commons



ugr

Universidad
de Granada



Lecciones

- Lección 4. Herramientas, estilos y estructuras en programación paralela
- Lección 5. Proceso de parallelización
- Lección 6. Evaluación de prestaciones en procesamiento paralelo

Objetivos Lección 5

- Programar en paralelo una aplicación sencilla.
- Distinguir entre asignación estática y dinámica (ventajas e inconvenientes).

Bibliografía Lección 5

➤ Fundamental

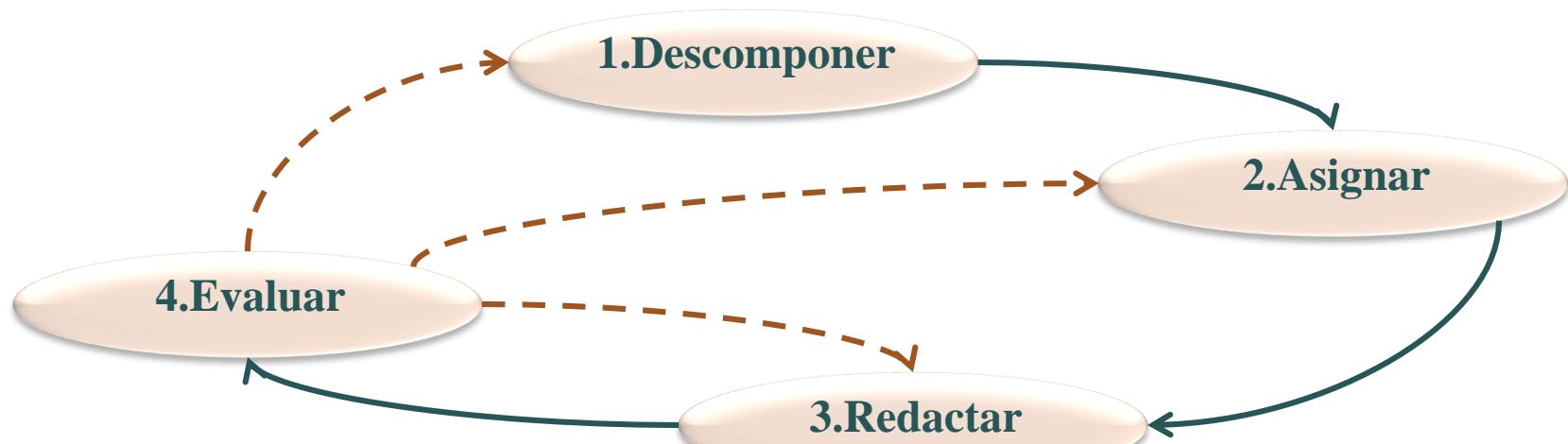
- Capítulo 7. Sección 7.4. J. Ortega, M. Anguita, A. Prieto.
“Arquitectura de Computadores”. Thomson, 2005. ESII/C.1
ORT arq

➤ Complementaria

- Thomas Rauber, Gudula Rünger. “Parallel Programming: for Multicore and Cluster Systems.” Springer, 2010. Disponible en línea (biblioteca UGR): <http://dx.doi.org/10.1007/978-3-642-04818-0>
- Barry Wilkinson.“Parallel programming : techniques and applications using networked workstations and parallel computer”, 2005. ESIIT/D.1 WIL par

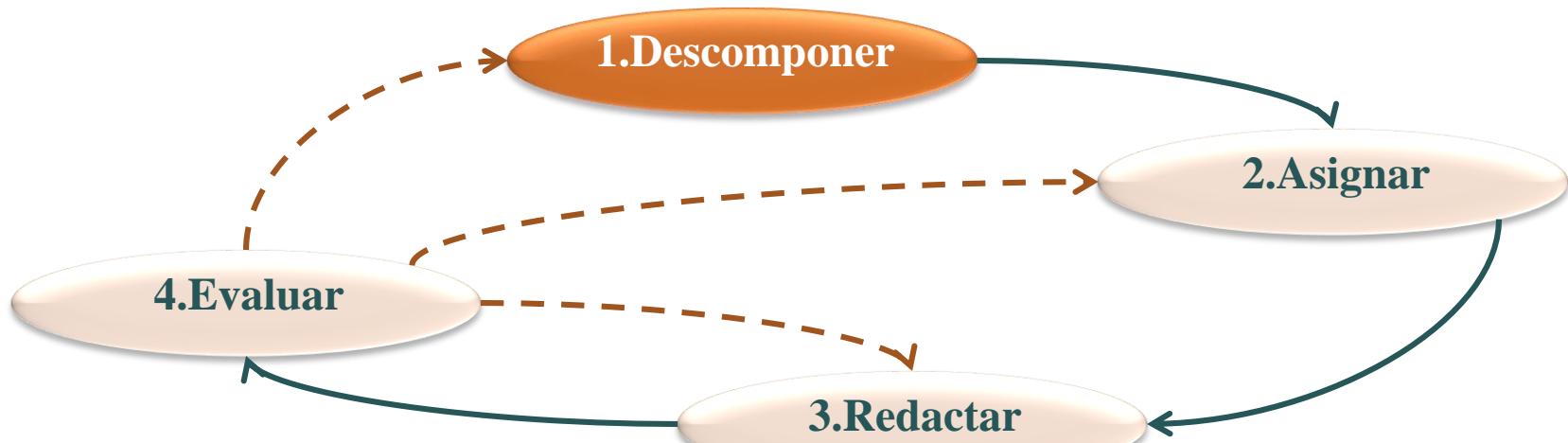
Proceso de paralelización

1. *Descomponer (decomposition)* en tareas independientes o Localizar paralelismo
2. *Asignar (planificar+mapear, schedule+map)* tareas a procesos y/o threads.
3. *Redactar* código paralelo.
4. *Evaluar* prestaciones.



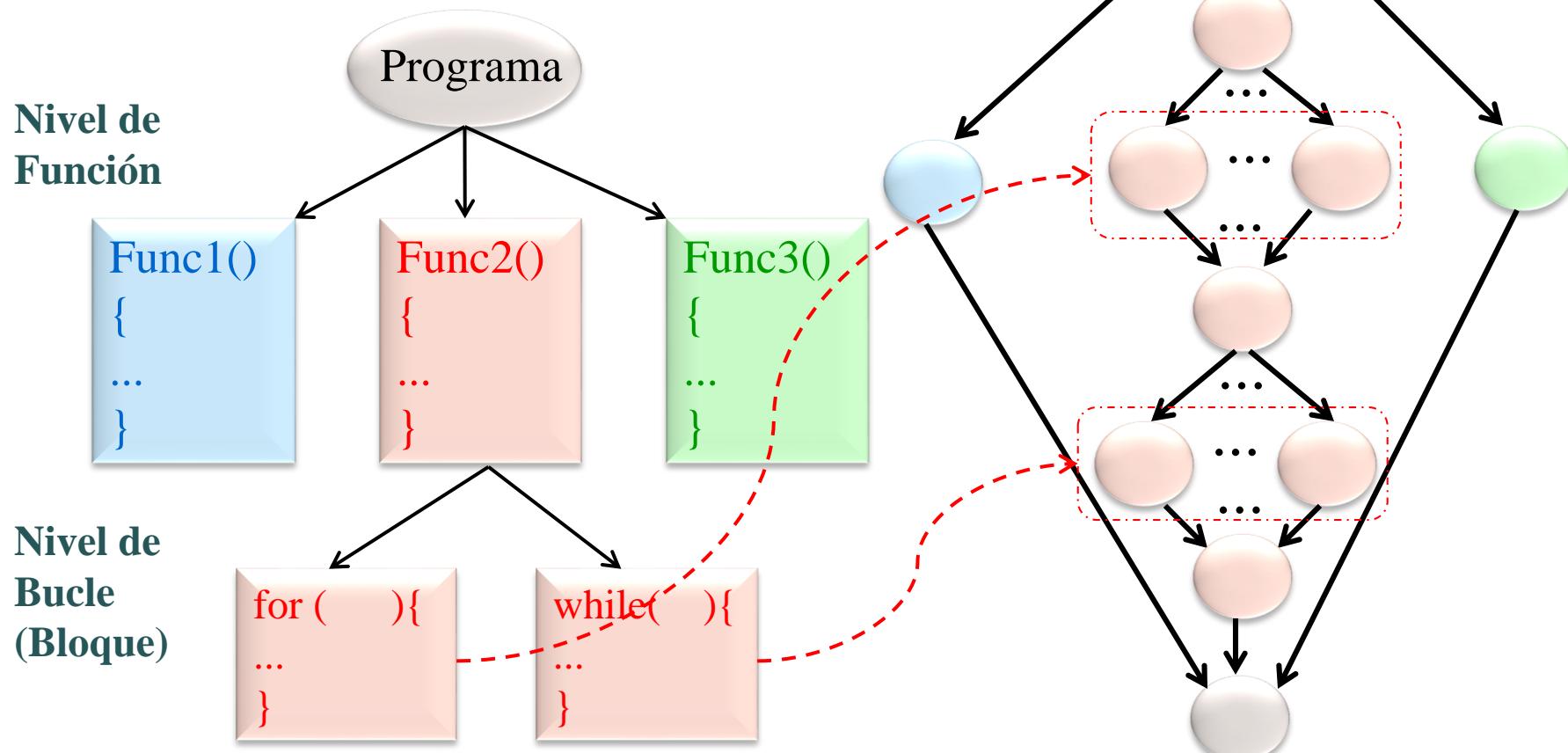
Proceso de paralelización

1. *Descomponer* en tareas independientes.
2. *Asignar* (*planificar+mapear*) tareas a procesos y/o threads.
3. *Redactar* código paralelo.
4. *Evaluar* prestaciones.



Descomposición en tareas independientes

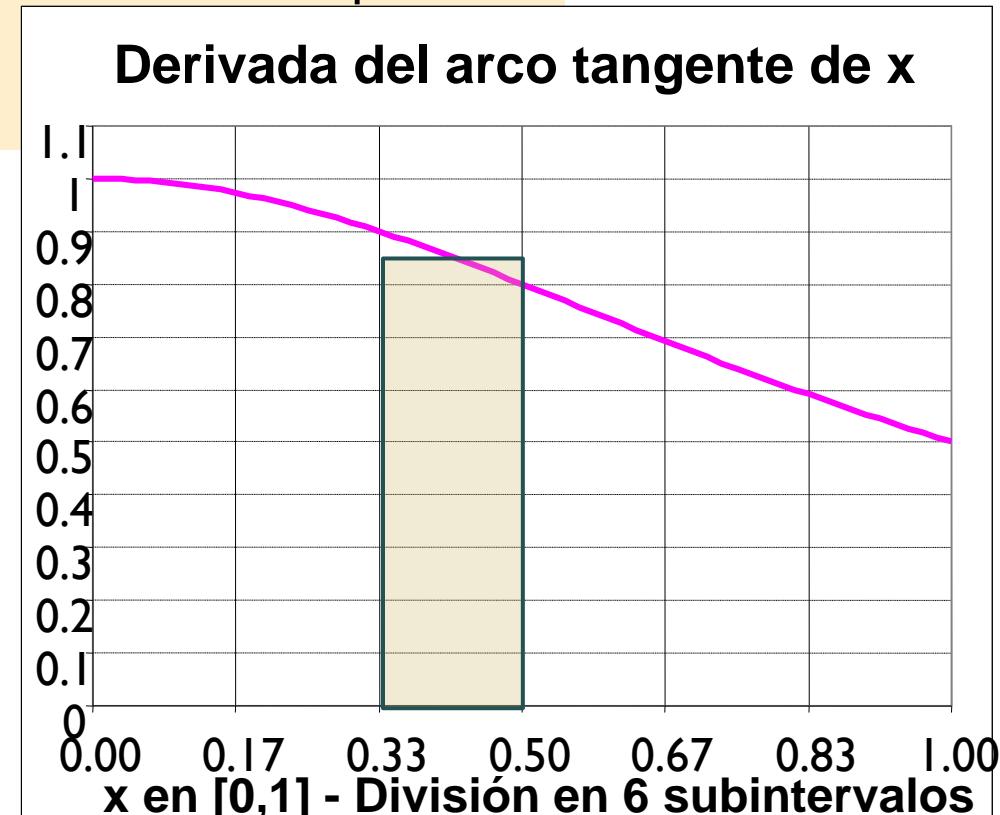
- Análisis de dependencia entre **funciones**.
- Análisis de dependencia entre **iteraciones de bucles**.



Ejemplo de cálculo PI: Descomposición en tareas independientes

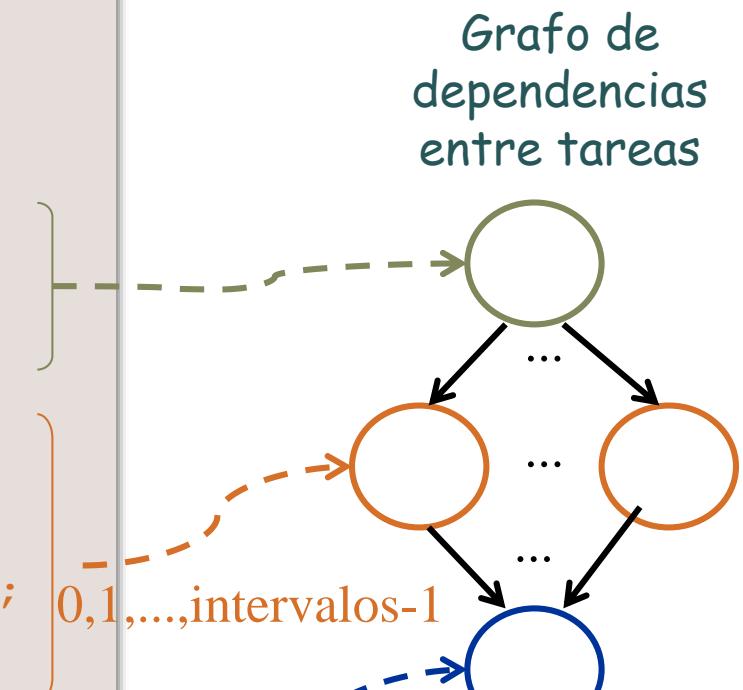
$$\left. \begin{array}{l} \arctg'(x) = \frac{1}{1+x^2} \\ \arctg(1) = \frac{\pi}{4} \\ \arctg(0) = 0 \end{array} \right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} = \arctg(x) \Big|_0^1 = \frac{\pi}{4} - 0$$

- PI se puede calcular por integración numérica.



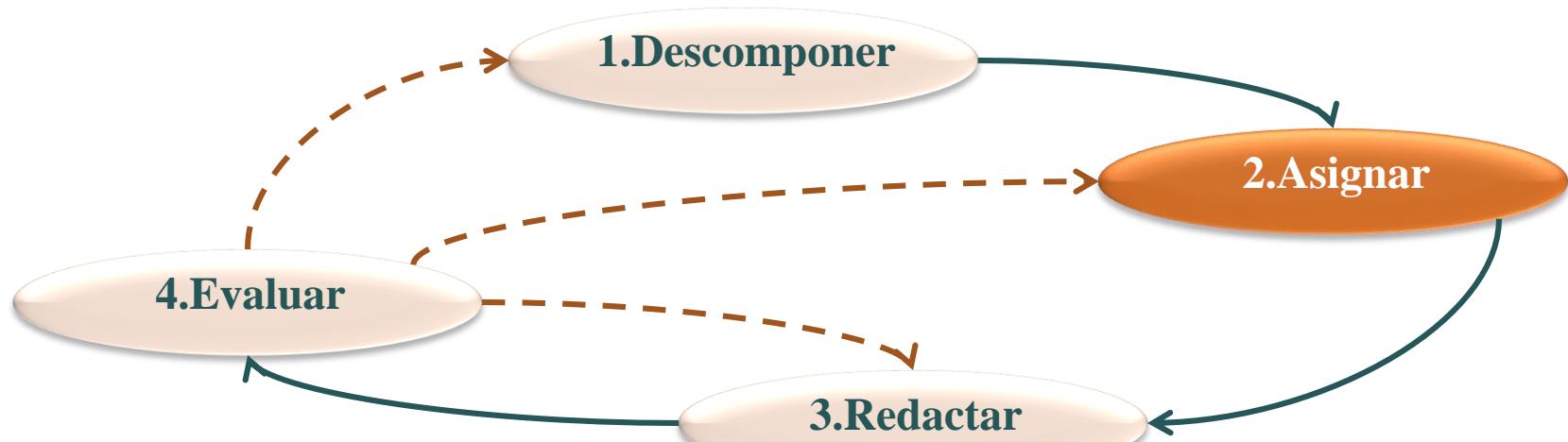
Ejemplo de cálculo PI: Descomposición en tareas independientes

```
main(int argc, char **argv) {  
    double ancho, sum=0;  
    int intervalos, i;  
  
    intervalos = atoi(argv[1]);  
    ancho = 1.0/(double) intervalos;  
  
    for (i=0;i< intervalos; i++) {  
        x = (i+0.5)*ancho;  
        sum = sum + 4.0/(1.0+x*x);  
    }  
  
    sum* = ancho; } }
```



Proceso de paralelización

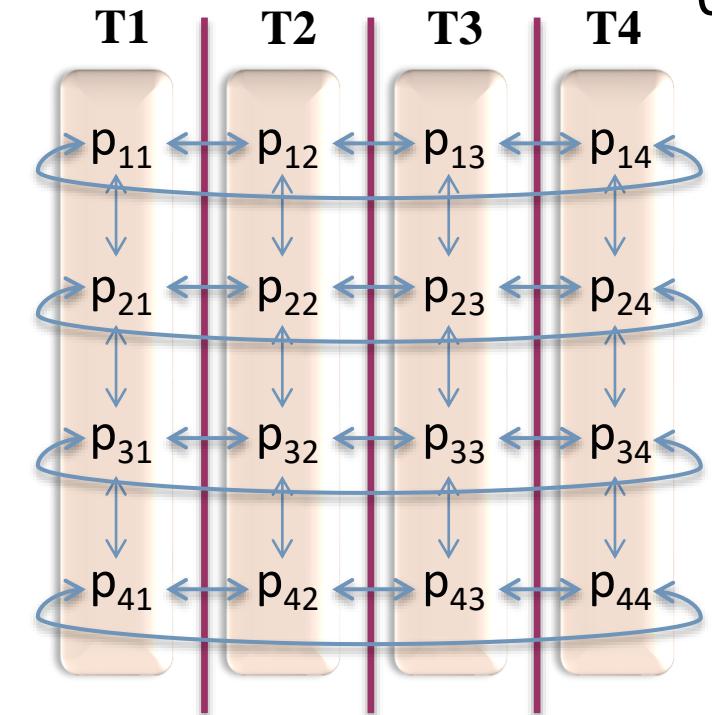
- 1. Descomponer** en tareas independientes.
- 2. Asignar (planificar+mapear)** tareas a procesos y/o threads.
- 3. Redactar** código paralelo.
- 4. Evaluar** prestaciones.



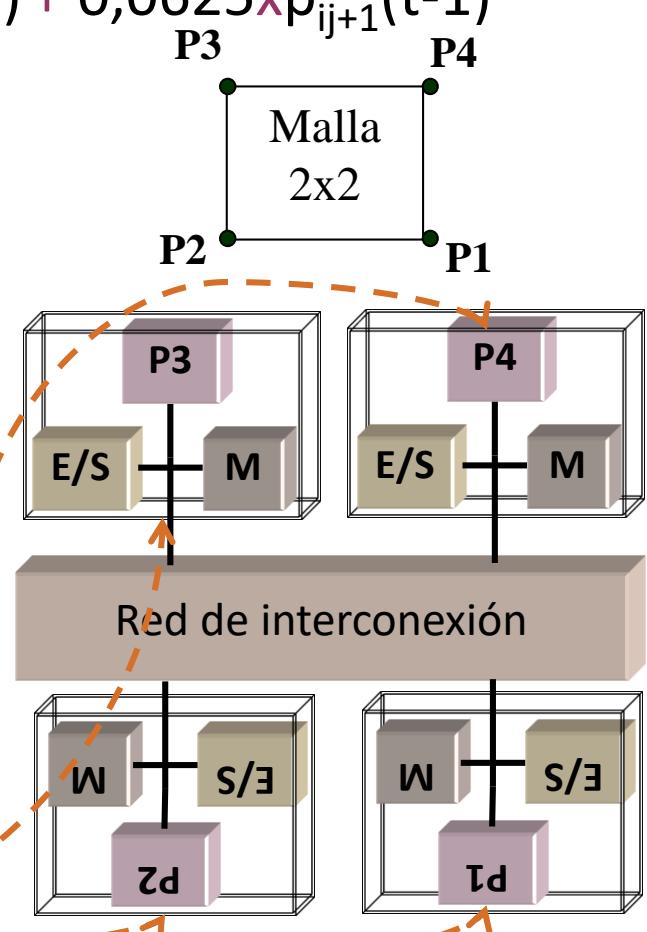
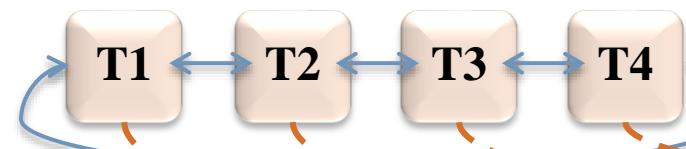
Asignación (planificación + mapeo). Ej.: filtrado de imagen I

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

Planificación: agrupar tareas en threads



Mapeo: asignar threads a cores/procesadores



Asignación de tareas a procesos/threads I

- Incluimos: agrupación de tareas en procesos/threads (*scheduling*) y mapeo a procesadores/cores (*mapping*)
- La **granularidad** de la carga de trabajo (tareas) asignada a los procesos/threads depende de:
 - número de cores o procesadores o elementos de procesamiento
 - tiempo de comunicación/sincronización frente a tiempo de cálculo
- **Equilibrado de la carga** (tareas = código + datos) o *load balancing*:
 - Objetivo: unos procesos/threads no deben esperar a otros



Asignación de tareas a procesos/threads II

➤ ¿De qué depende el equilibrio?

➤ La arquitectura:

- homogénea frente a heterogénea,
- uniforme frente a no uniforme

➤ La aplicación/descomposición

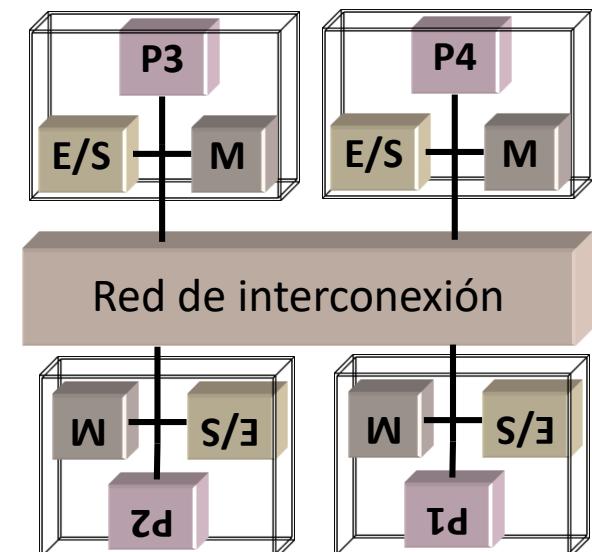
➤ Tipos de asignación:

➤ Estática

- Está determinado qué tarea va a realizar cada procesador o core
- Explícita: programador
- Implícita: herramienta de programación al generar el código ejecutable

➤ Dinámica (en tiempo de ejecución)

- Distintas ejecuciones pueden asignar distintas tareas a un procesador o core
- Explícita: el programador
- Implícita: herramienta de programación al generar el código ejecutable



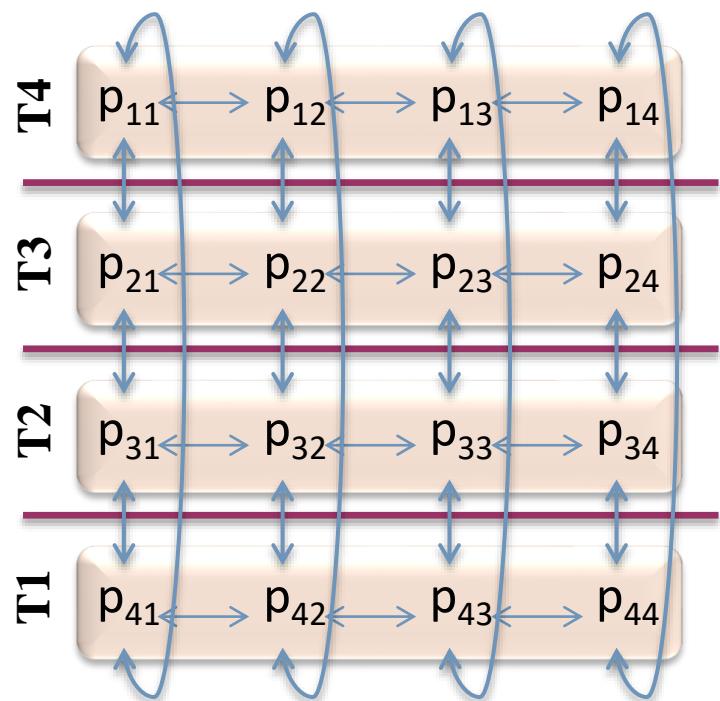
Mapeo de procesos/threads a unidades de procesamiento

- Normalmente se deja al SO el mapeo de threads (*light process*)
- Lo puede hacer el entorno o sistema en tiempo de ejecución (*runtime system* de la herramienta de programación)
- El programador puede influir

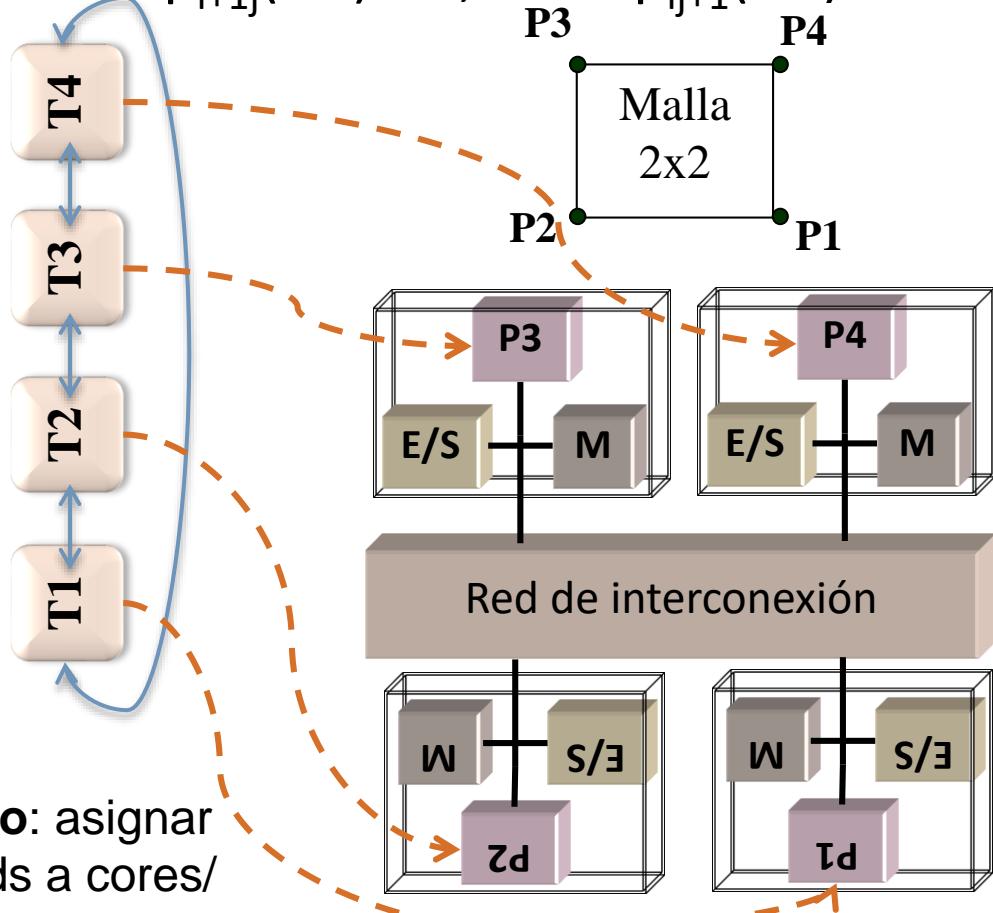
Asignación (planificación + mapeo). Ej.: filtrado de imagen II

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

Planificación: agrupar tareas en threads



Mapeo: asignar threads a cores/procesadores



Códigos filtrado imagen

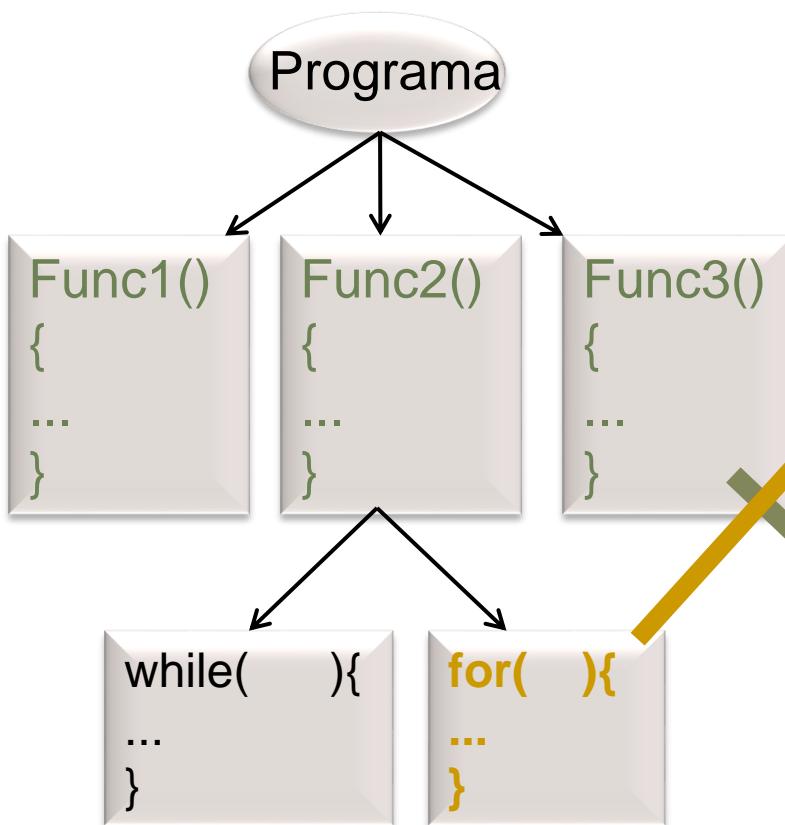
Descomposición por columnas

```
#include <omp.h>
...
omp_set_num_threads(M)
#pragma omp parallel private(i)
{
    for (i=0;i<N;i++) {
        #pragma omp for
        for (j=0;j<M;j++) {
            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]);
        }
    }
}
...
```

Descomposición por filas

```
#include <omp.h>
...
omp_set_num_threads(N)
#pragma omp parallel private(j)
{
    #pragma omp for
    for (i=0;i<N;i++) {
        for (j=0;j<M;j++) {
            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]);
        }
    }
}
...
```

Ejemplo de asignación estática del paralelismo de tareas y datos con OpenMP



```
Func1 () { ... }
Func2 () { ...
    #pragma omp parallel for \
        schedule(static)
    for (i=0;i<N;i++) {
        código para i
    }
}
Func3 () { ... }
Main () {

# pragma omp parallel sections
{ # pragma omp section
    Func1();
# pragma omp section
    Func2();
# pragma omp section
    Func3();
}
```

Asignación estática

- Asignación **estática** y explícita de las iteraciones de un bucle:

Estática Round-Robin

Bucle

```
for (i=0;i<Iter;i++) {  
    código para i  
}
```

```
for (i=idT;i<Iter;i=i+nT) {  
    código para i  
}
```

Estática Continua

```
for (i= idT * Iter / nT ; i < (idT + 1) * Iter / nT ; i++) {  
    código para i  
}
```

Asignación dinámica

- Asignación **dinámica** y explícita de las iteraciones de un bucle:

Bucle

```
for (i=0;i<Iter;i++) {  
    código para i  
}
```



```
lock(k);  
    n=i;    i=i+1;  
unlock(k);  
while (n<Iter) {  
  
    código para n ;  
  
    lock(k);  
    n=i;    i=i+1;  
unlock(k);  
}
```

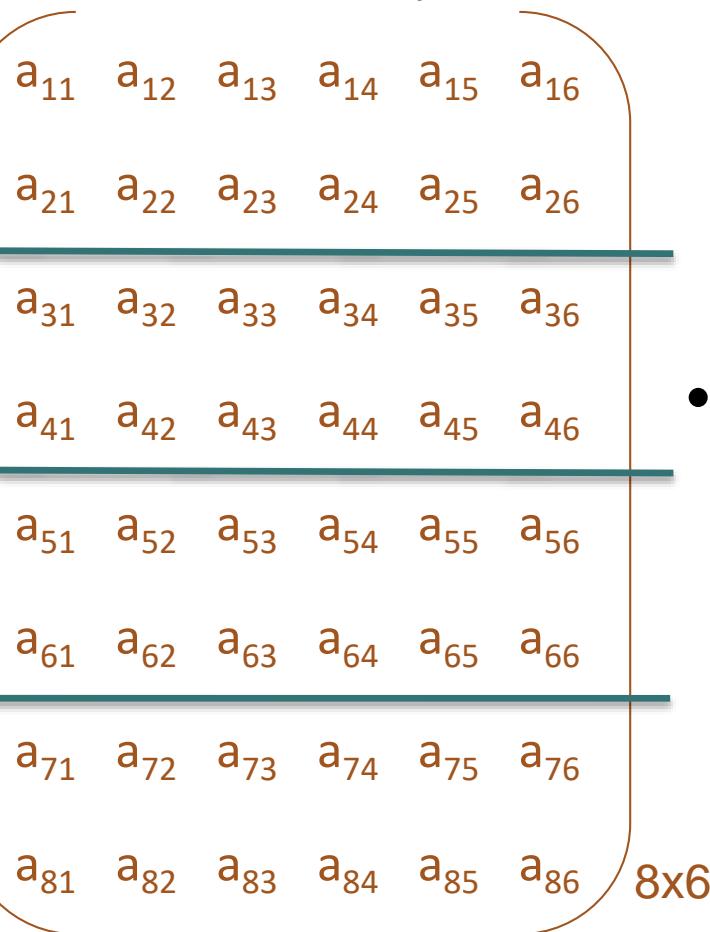
- Dinámica e implícita

Ej. con OpenMP

NOTA: La variable i se supone inicializada a 0

Asignación. Ej.: multiplicación matriz por vector I

$$c = A \bullet b; \quad c_i = \sum_{k=0}^{M-1} a_{ik} \bullet b_k = a_i^T \bullet b, \quad c(i) = \sum_{k=0}^{M-1} A(i,k) \bullet b(k), \quad i = 0, \dots, N-1$$



$$\begin{matrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{matrix} = \begin{matrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \end{matrix}$$

N=8 M=6

0 $c(1) = \sum_{k=0}^{M-1} A(1,k) \bullet b(k)$

1 $c(2) = \sum_{k=0}^{M-1} A(2,k) \bullet b(k)$

2 $c(3) = \sum_{k=0}^{M-1} A(3,k) \bullet b(k)$

3 $c(4) = \sum_{k=0}^{M-1} A(4,k) \bullet b(k)$

4 $c(5) = \sum_{k=0}^{M-1} A(5,k) \bullet b(k)$

5 $c(6) = \sum_{k=0}^{M-1} A(6,k) \bullet b(k)$

6 $c(7) = \sum_{k=0}^{M-1} A(7,k) \bullet b(k)$

7 $c(8) = \sum_{k=0}^{M-1} A(8,k) \bullet b(k)$

Asignación. Ej.: multiplicación matriz por vector II

$$\mathbf{c} = \mathbf{A} \bullet \mathbf{b}; \quad c_i = \sum_{k=0}^{M-1} a_{ik} \bullet b_k = a_i^T \bullet \mathbf{b}, \quad c(i) = \sum_{k=0}^{M-1} A(i,k) \bullet b(k), \quad i = 0, \dots, N-1$$

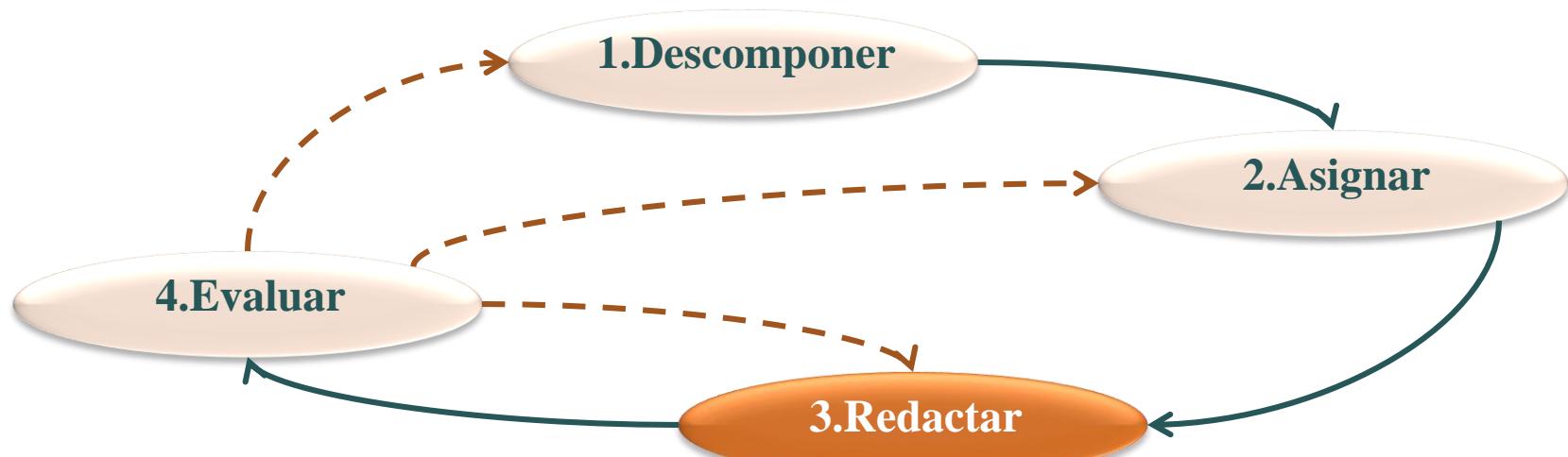
0	1	2
a_{11}	a_{12}	a_{13}
a_{14}	a_{15}	a_{16}
a_{21}	a_{22}	a_{23}
a_{24}	a_{25}	a_{26}
a_{31}	a_{32}	a_{33}
a_{34}	a_{35}	a_{36}
a_{41}	a_{42}	a_{43}
a_{44}	a_{45}	a_{46}
a_{51}	a_{52}	a_{53}
a_{54}	a_{55}	a_{56}
a_{61}	a_{62}	a_{63}
a_{64}	a_{65}	a_{66}
a_{71}	a_{72}	a_{73}
a_{74}	a_{75}	a_{76}
a_{81}	a_{82}	a_{83}
a_{84}	a_{85}	a_{86}

$$\begin{array}{c|c|c} 0 & b_1 & c_1 \\ & b_2 & c_2 \\ \hline 1 & b_3 & c_3 \\ & b_4 & c_4 \\ \hline 2 & b_5 & c_5 \\ & b_6 & c_6 \\ \hline & & c_7 \\ & & c_8 \end{array} = \begin{array}{c|c} N=8 & c_1 \\ M=6 & c_2 \\ & c_3 \\ & c_4 \\ & c_5 \\ & c_6 \\ & c_7 \\ & c_8 \end{array}$$

$c_1 = a_{11}b_1 + a_{12}b_2 + a_{13}b_3 + a_{14}b_4 + a_{15}b_5 + a_{16}b_6$
 $c_2 = a_{21}b_1 + a_{22}b_2 + a_{23}b_3 + a_{24}b_4 + a_{25}b_5 + a_{26}b_6$
 $c_3 = a_{31}b_1 + a_{32}b_2 + a_{33}b_3 + a_{34}b_4 + a_{35}b_5 + a_{36}b_6$
 $c_4 = a_{41}b_1 + a_{42}b_2 + a_{43}b_3 + a_{44}b_4 + a_{45}b_5 + a_{46}b_6$
 $c_5 = a_{51}b_1 + a_{52}b_2 + a_{53}b_3 + a_{54}b_4 + a_{55}b_5 + a_{56}b_6$
 $c_6 = a_{61}b_1 + a_{62}b_2 + a_{63}b_3 + a_{64}b_4 + a_{65}b_5 + a_{66}b_6$
 $c_7 = a_{71}b_1 + a_{72}b_2 + a_{73}b_3 + a_{74}b_4 + a_{75}b_5 + a_{76}b_6$
 $c_8 = a_{81}b_1 + a_{82}b_2 + a_{83}b_3 + a_{84}b_4 + a_{85}b_5 + a_{86}b_6$

Proceso de paralelización

- 1. Descomponer** en tareas independientes.
- 2. Asignar (planificar+mapear)** tareas a procesos y/o threads.
- 3. Redactar código paralelo.**
- 4. Evaluar** prestaciones.



Ejemplo: cálculo de PI con OpenMP/C

AC

ATC

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

Diagrama de ejecución en paralelo:

Annotations:

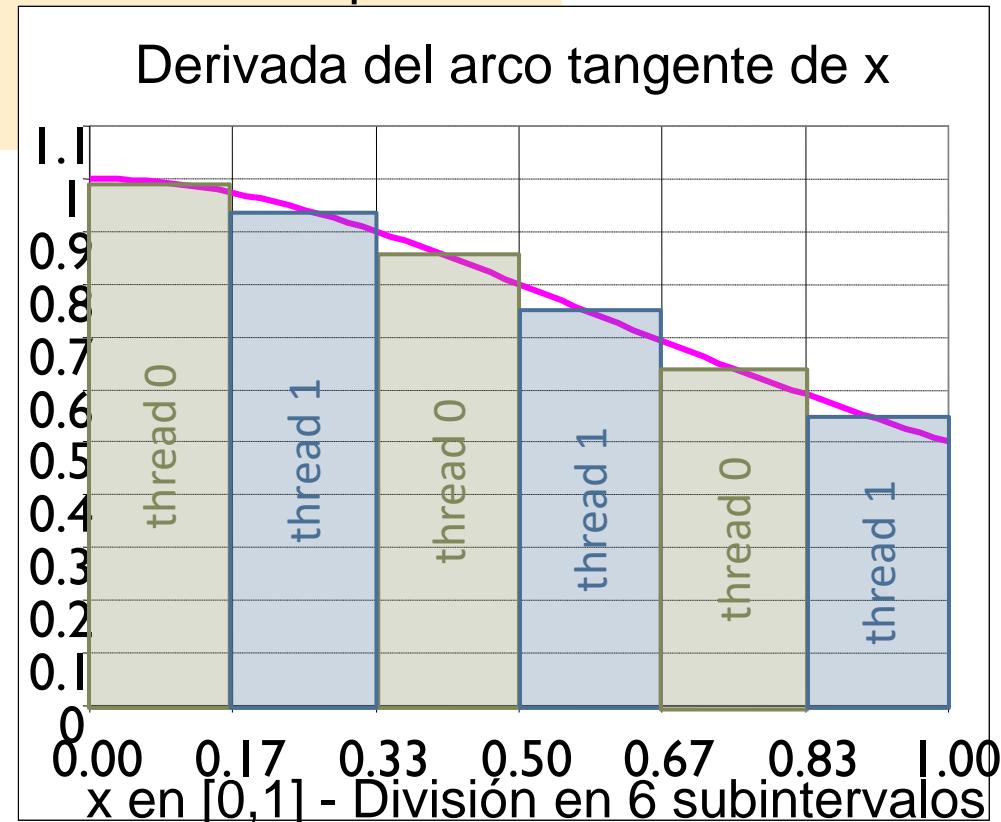
- Localizar → Señala la palabra `omp_parallel`.
- Crear/Terminar → Señala la instrucción `omp_set_num_threads`.
- Comunicar/sincronizar → Señala la instrucción `schedule(dynamic)`.
- Agrupar/Asignar → Señala la instrucción `reduction(+:sum)`.



Asignación de tareas a 2 threads estática por turno rotatorio

$$\left. \begin{array}{l} \arctg'(x) = \frac{1}{1+x^2} \\ \arctg(1) = \frac{\pi}{4} \\ \arctg(0) = 0 \end{array} \right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} = \arctg(x) \Big|_0^1 = \frac{\pi}{4} - 0$$

- PI se puede calcular por integración numérica.



Ejemplo: cálculo de PI en MPI/C

```
#include <mpi.h>
main(int argc, char **argv)  {
    double ancho,x,lsum,sum; int intervalos,i,nproc,iproc;
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    intervalos=atoi(argv[1]);
    ancho=1.0/(double) intervalos; lsum=0;
    for (i=iproc; i<intervalos; i+=nproc) {
        x = (i+0.5)*ancho; lsum+= 4.0/(1.0+x*x);
    }
    lsum*= ancho;
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
               MPI_SUM,0,MPI_COMM_WORLD);
    MPI_Finalize();
}
```

Enrolar → **MPI_Init(&argc, &argv)**

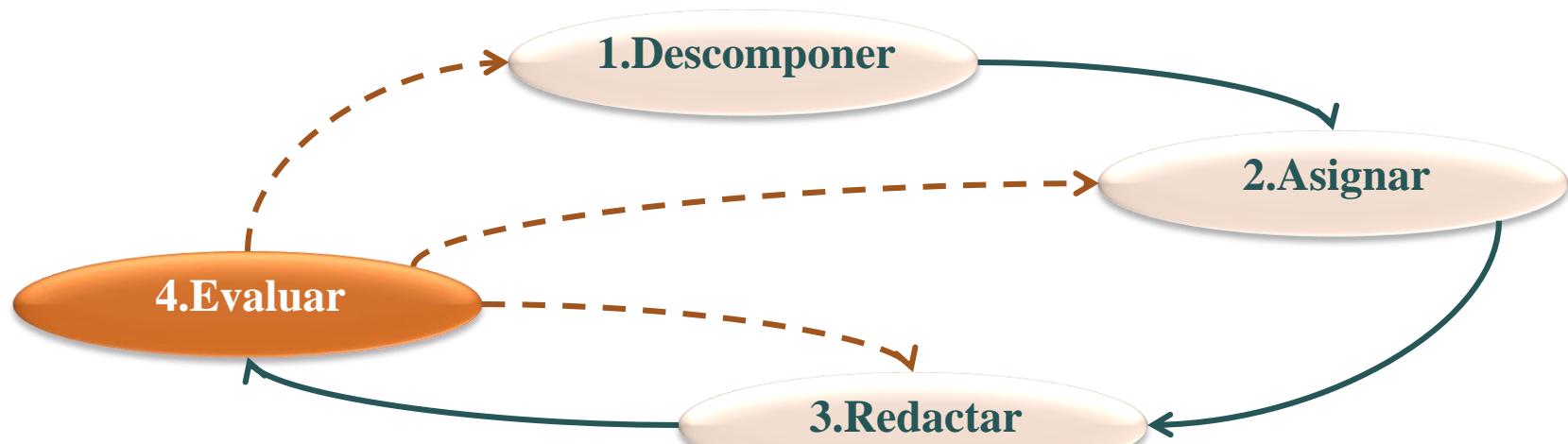
Localizar-Agrupar → **MPI_Comm_size(MPI_COMM_WORLD, &nproc);**
MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
intervalos=atoi(argv[1]);

Comunicar/sincronizar → **MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,**
MPI_SUM,0,MPI_COMM_WORLD);

Desenrolar → **MPI_Finalize();**

Proceso de paralelización

- 1. Descomponer** en tareas independientes.
- 2. Asignar (agrupar+mapear)** tareas a procesos y/o threads.
- 3. Redactar** código paralelo.
- 4. Evaluar** prestaciones.



2º curso / 2º cuatr.

Grado en
Ing. Informática

Arquitectura de Computadores

Tema 2

Programación paralela

Material elaborado por los profesores responsables de la asignatura:
Mancia Anguita – Julio Ortega

Licencia Creative Commons



ugr

Universidad
de Granada



Lecciones

- Lección 4. Herramientas, estilos y estructuras en programación paralela
- Lección 5. Proceso de parallelización
- Lección 6. Evaluación de prestaciones en procesamiento paralelo
 - Ganancia en prestaciones y escalabilidad
 - Ley de Amdahl
 - Ganancia escalable

Objetivos Lección 6

- Obtener ganancia y escalabilidad en el contexto de procesamiento paralela
- Aplicar la ley de Amdahl en el contexto de procesamiento paralela
- Comparar la ley de Amdahl y ganancia escalable.

Bibliografía

➤ Fundamental

- Capítulo 2, Secciones 4. M. Anguita, J. Ortega. Fundamentos y problemas de Arquitectura de Computadores, Editorial Técnica Avicam. ESIIT/C.1 ANG fun
- Secc. 7.5. J. Ortega, M. Anguita, A. Prieto. “Arquitectura de Computadores”. ESII/C.1 ORT arq

Contenido Lección 6

- Ganancia en prestaciones y escalabilidad
- Ley de Amdahl
- Ganancia escalable

Evaluación de prestaciones

- Medidas usuales
 - Tiempo de respuesta
 - Real (*wall-clock time, elapsed time*) (`/usr/bin/time`)
 - Usuario, sistema, CPU time = user + sys
 - Productividad
- Escalabilidad
- Eficiencia
 - Relación prestaciones/prestaciones máximas
 - Rendimiento = prestaciones/nº_recursos
 - Otras: Prestaciones/consumo_potencia,
prestaciones/área_ocupada

Ganancia en prestaciones. Escalabilidad

Ganancia en prestaciones:

$$S(p) = \frac{\text{Prestaciones}(p)}{\text{Prestaciones}(1)} = \frac{T_s}{T_p(p)}$$

Ganancia en velocidad (Speedup)

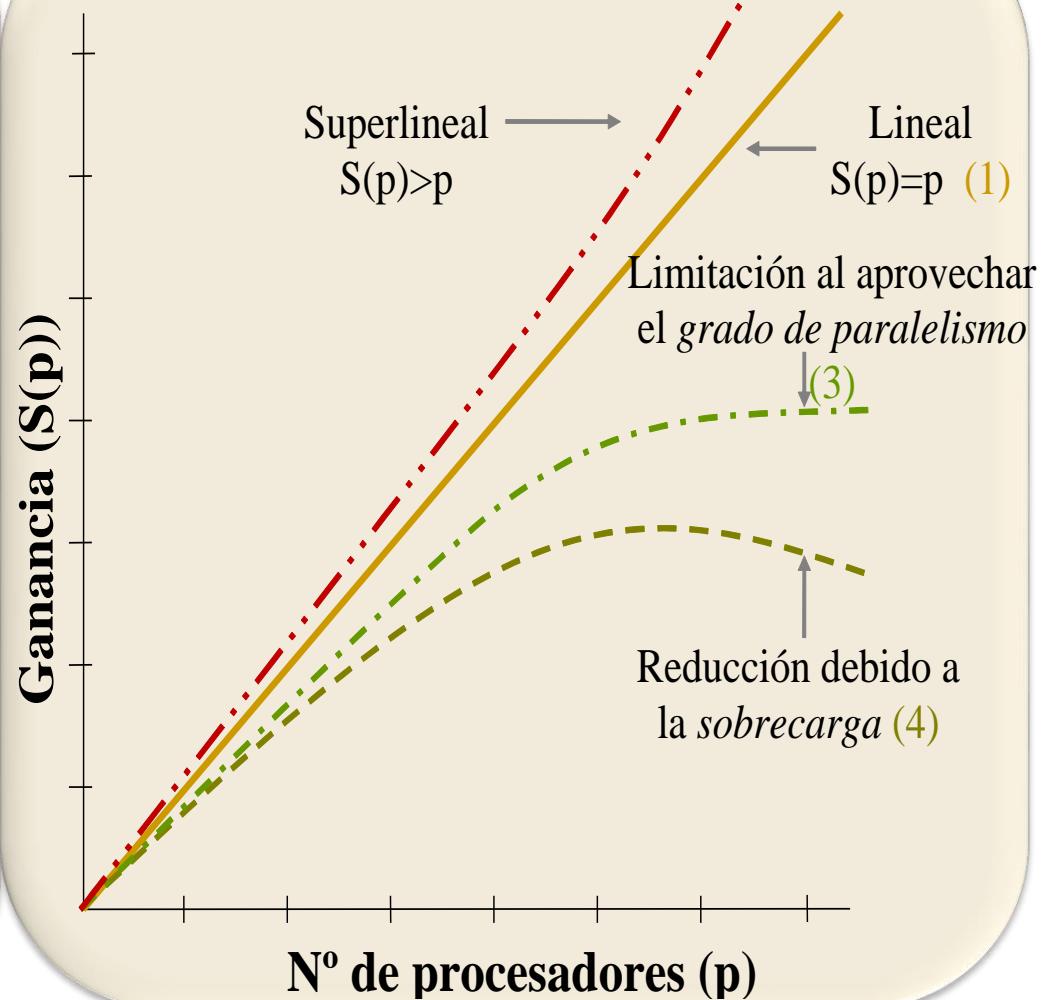
$$T_p(p) = T_c(p) + T_o(p)$$



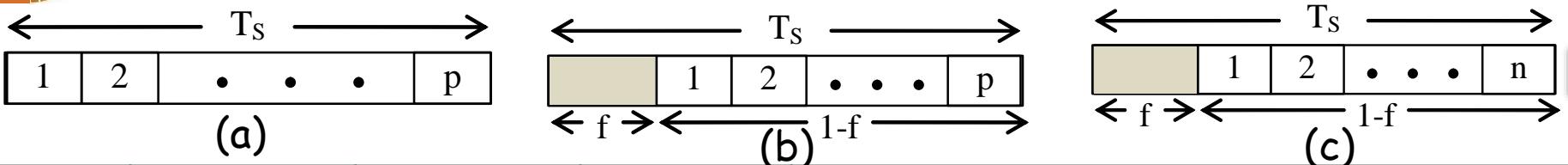
Sobrecarga (Overhead):

- Comunicación/sincronización.
- Crear/terminar procesos/threads.
- Cálculos o funciones no presentes en versión secuencial.
- Falta de equilibrado.

Escalabilidad:

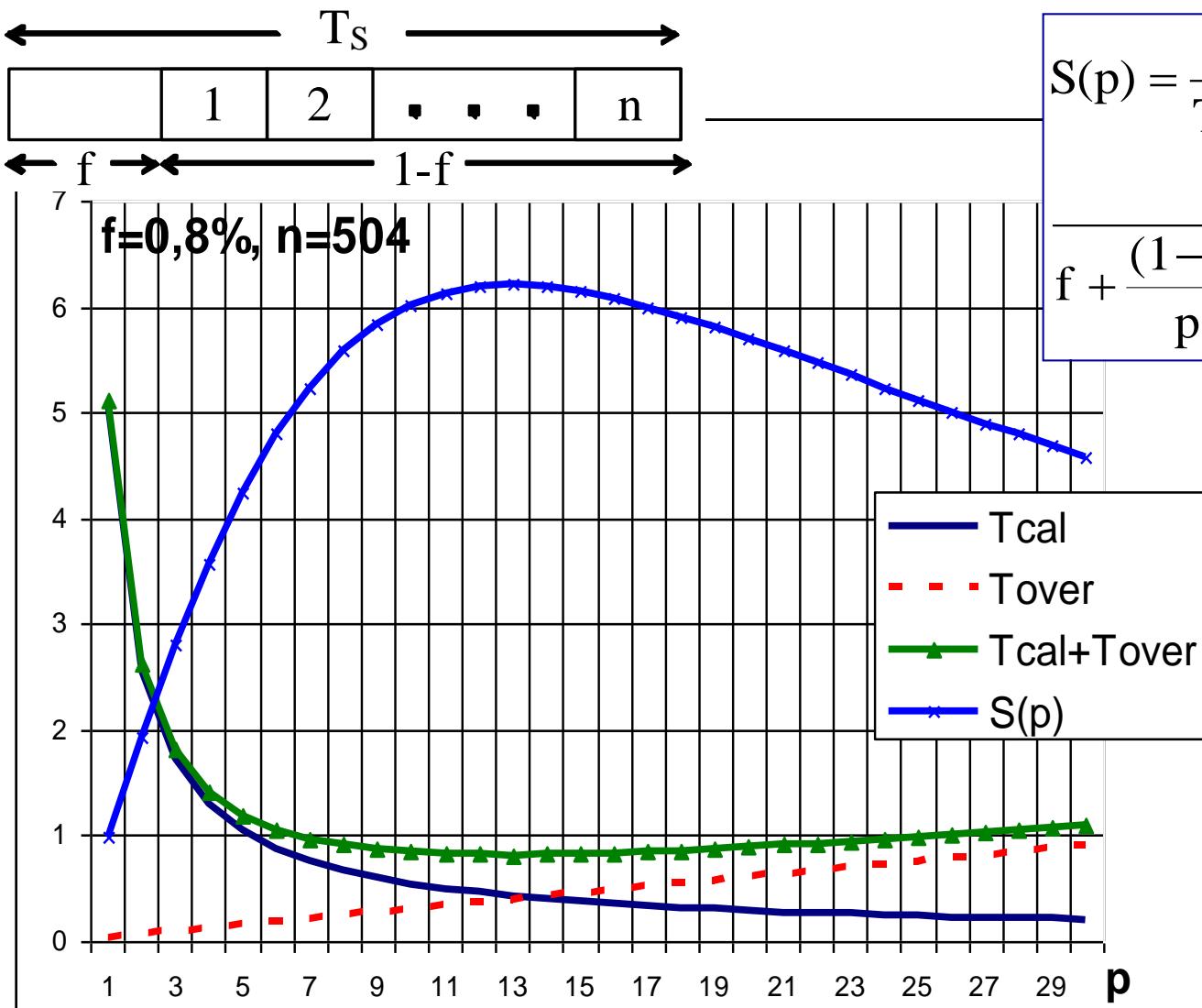


Ganancia en prestaciones. Ganancia máxima



Modelo código	Fracción no paral. en T_S	Grado paralelismo	Overhead	Ganancia en función del número de procesadores p con T_S constante
(a)	0	ilimitado	0	$S(p) = \frac{T_S}{T_P(p)} = p$ Ganancia lineal (1)
(b)	f	ilimitado	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$ (2)
(c)	f	n	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p=n} \frac{1}{f + \frac{(1-f)}{n}}$ (3)
(b)	f	ilimitado	Incrementa linealmente con p	$S(p) = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_O(p)}{T_S}} \xrightarrow{p \rightarrow \infty} 0$ (4)

Número de procesadores óptimo



$$S(p) = \frac{T_S}{T_P(p)} = \frac{T_S}{T_C(p) + T_O(p)} = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_O(p)}{T_S}}$$

$$T_C(p) = O\left(\frac{1}{p}\right)$$

$$T_O(p) = O(p)$$

```
For(i=0;i<n;i++) {  
    Código para i  
}
```

Contenido Lección 6

- Ganancia en prestaciones y escalabilidad
- Ley de Amdahl
- Ganancia escalable

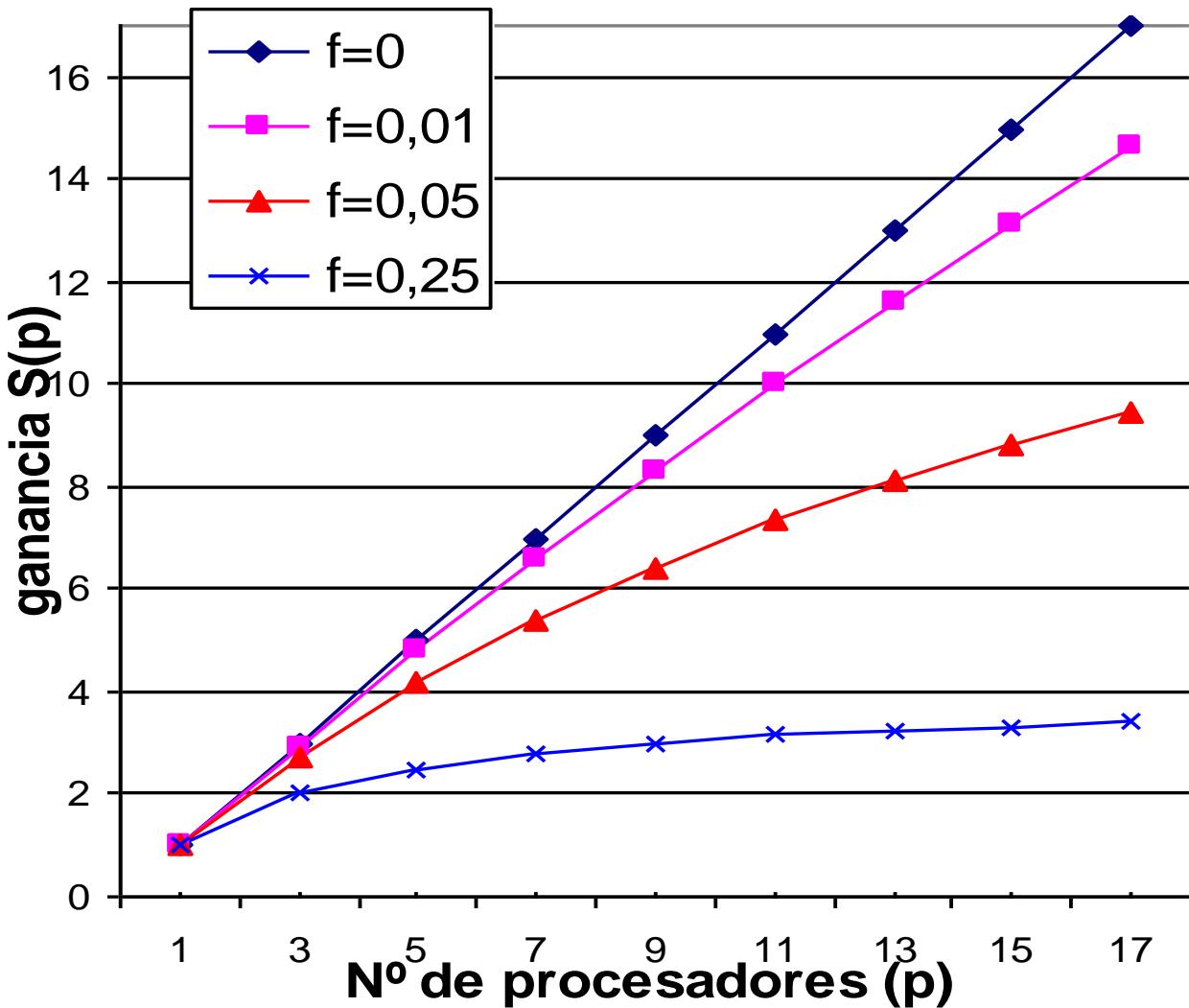
Ley de Amdahl

- Ley de Amdahl: la ganancia en prestaciones utilizando p procesadores está limitada por la fracción de código que no se puede parallelizar (2):

$$S(p) = \frac{T_s}{T_p(p)} \leq \frac{T_s}{f \cdot T_s + \frac{(1-f) \cdot T_s}{p}} = \frac{p}{1 + f(p-1)} \rightarrow \frac{1}{f} \quad (p \rightarrow \infty)$$

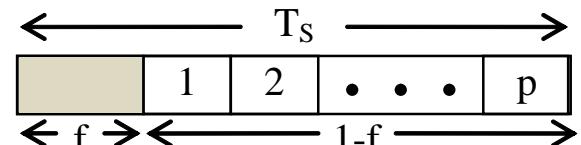
- S : Incremento en velocidad que se consigue al aplicar una mejora. (*parallelismo*)
- p : Incremento en velocidad máximo que se puede conseguir si se aplica la mejora todo el tiempo. (*número de procesadores*)
- f : fracción de tiempo en el que no se puede aplicar la mejora. (*fracción de t. no parallelizable*)

Ley de Amdahl

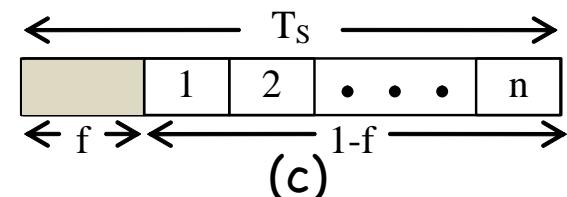
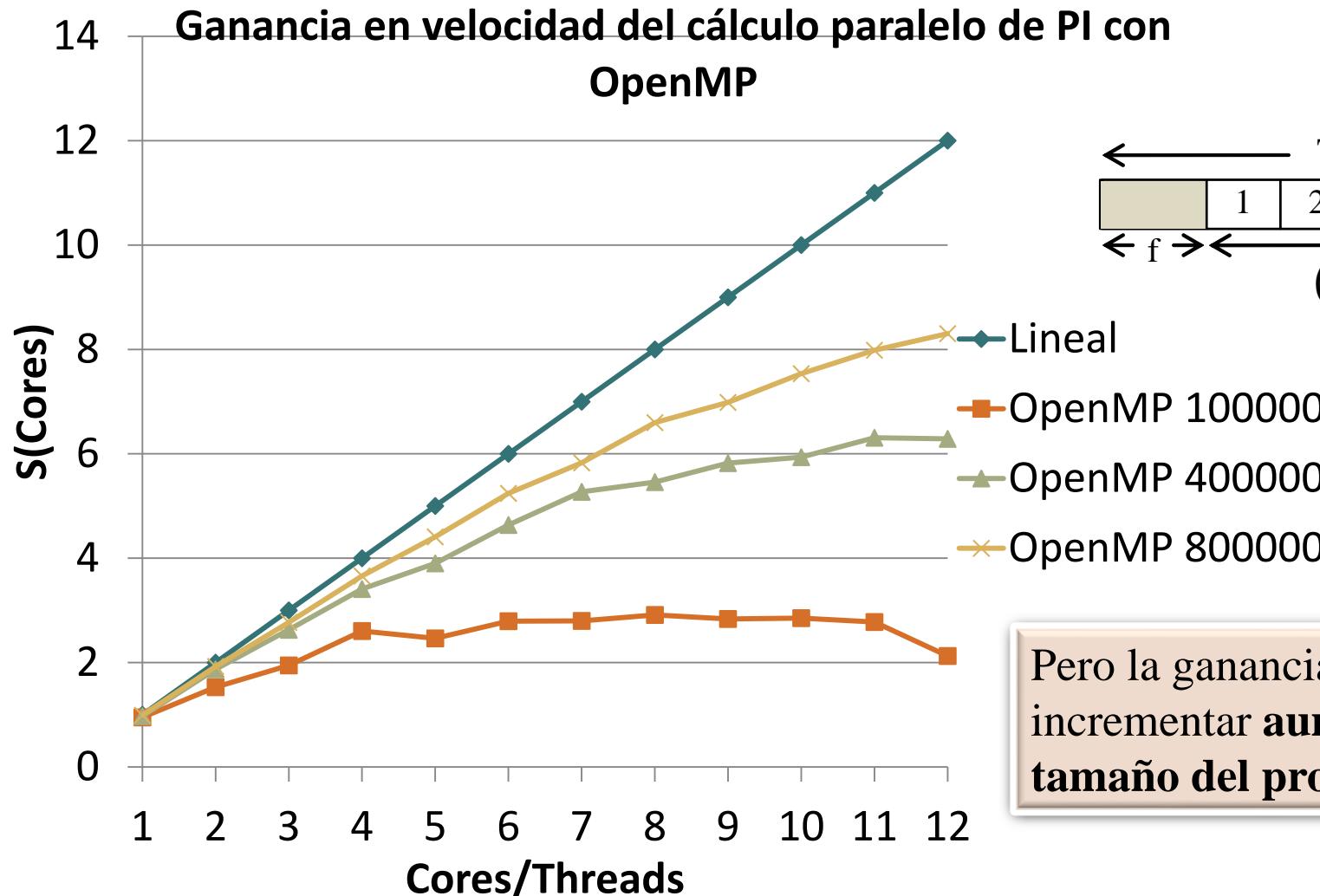


$$S(p) = \frac{p}{1 + f(p - 1)}$$
$$S(p) \rightarrow \frac{1}{f} (p \rightarrow \infty)$$

Es pesimista.
Nos dice que la *escalabilidad* está limitada por f (*fracción de T_s que no se puede paralelizar*)



Ganancia escalable

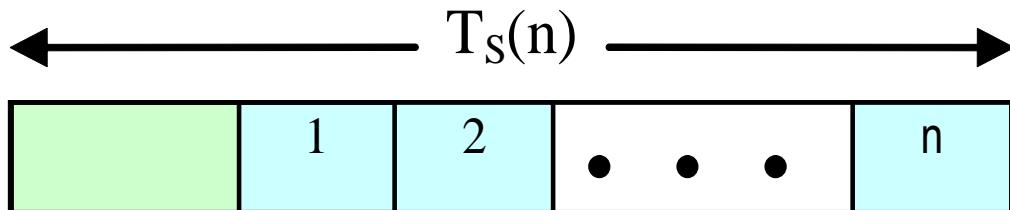


Pero la ganancia se puede incrementar **aumentando el tamaño del problema**

Contenido Lección 6

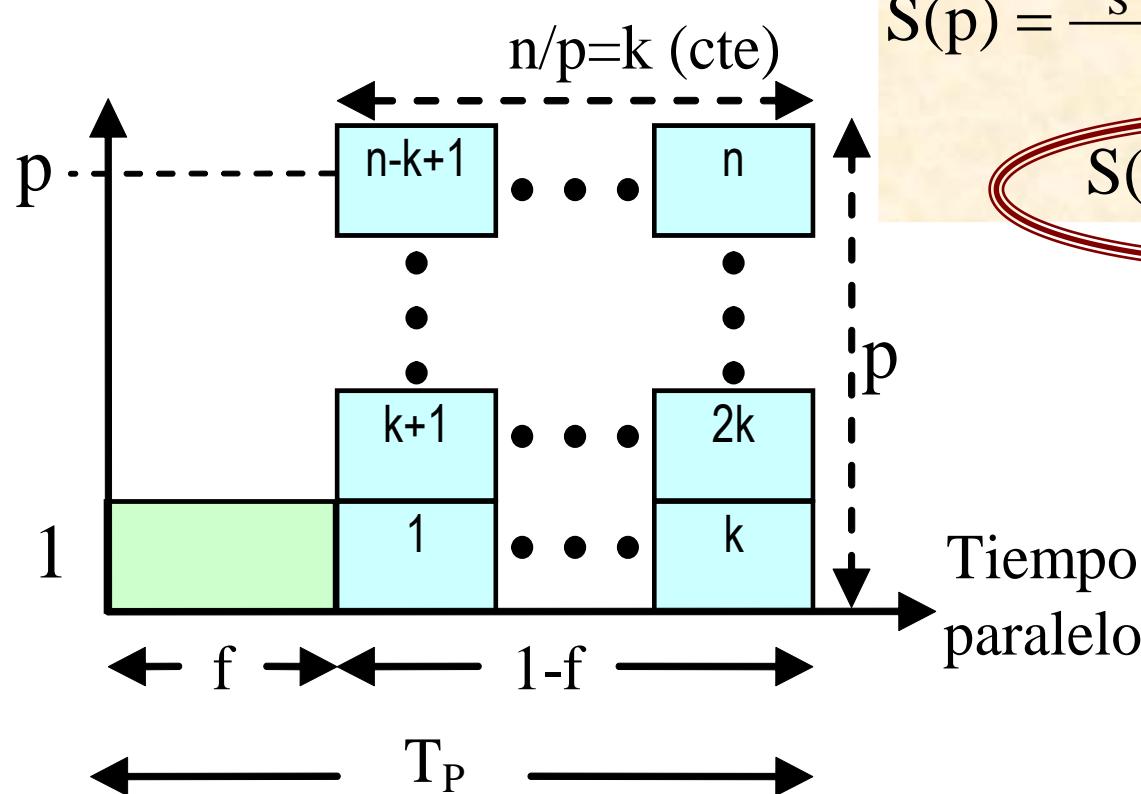
- Ganancia en prestaciones y escalabilidad
- Ley de Amdahl
- Ganancia escalable

Ganancia escalable o Ley de Gustafson



Amdahl mantiene constante T_S ,
Gustafson mantiene constante T_P

Procesadores



$$S(p) = \frac{T_S(n = kp)}{T_P} = \frac{fT_P + p(1-f)T_P}{T_P}$$
$$S(p) = p(1-f) + f$$

Para ampliar ...

- Páginas Web:
 - http://en.wikipedia.org/wiki/Parallel_computing
- Artículos en revistas
 - Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference (AFIPS '67 (Spring))*. ACM, New York, NY, USA. Disponible en línea (biblioteca UGR):
<http://doi.acm.org/10.1145/1465482.1465560>
 - John L. Gustafson. 1988. Reevaluating Amdahl's law. *Commun. ACM* 31, 5 (May 1988), 532-533. Disponible en línea (biblioteca UGR):
<http://doi.acm.org/10.1145/42411.42415>