Esquemas paralelos

Metodología de la Programación Paralela

Jesús Sánchez Cuadrado (jesusc@um.es)

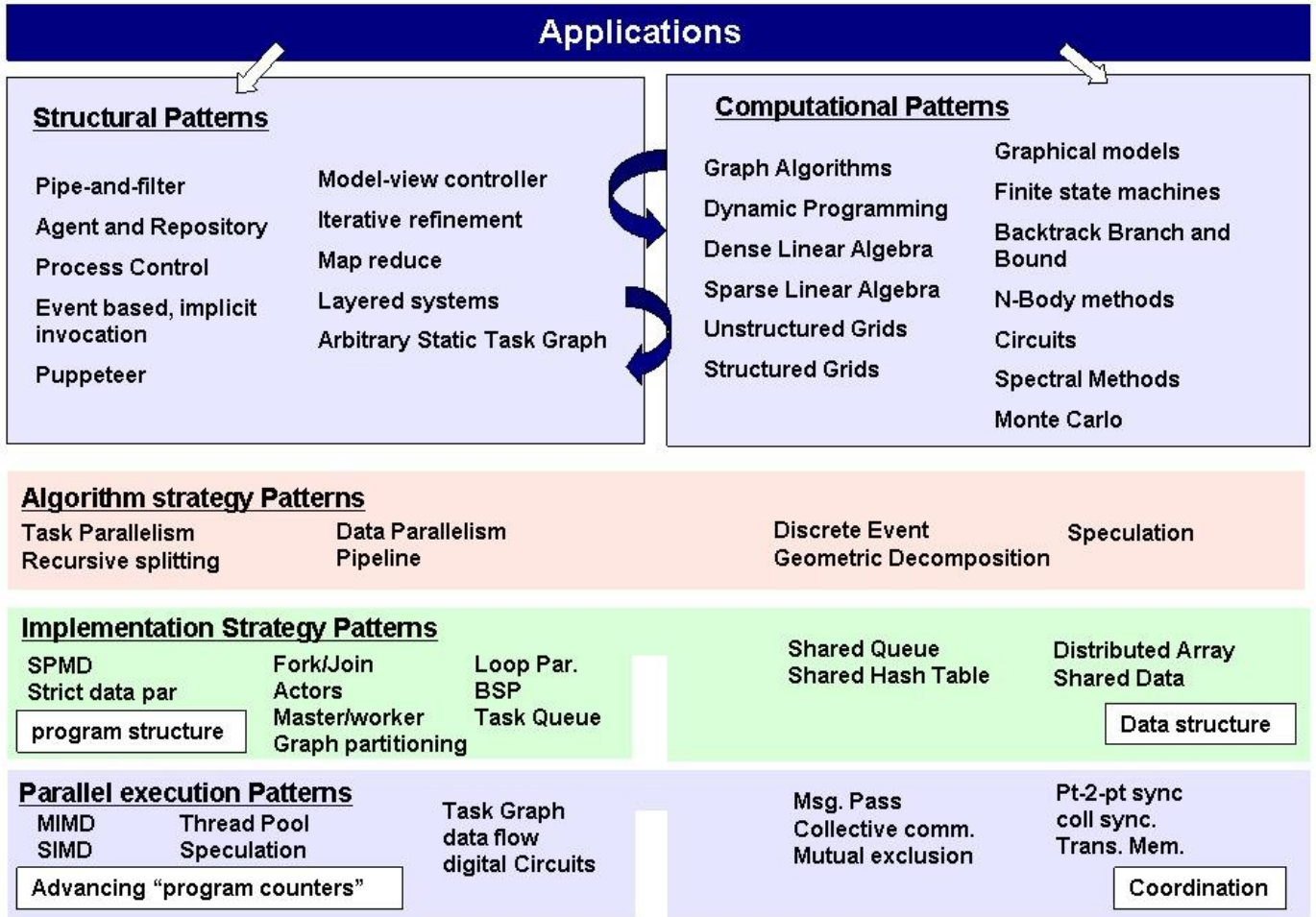
Curso 2020/21

Tipos de esquemas

- Podemos considerar esquemas de distintos tipos:
- Descomposición del trabajo:
  - Paralelismo de datos.
  - Particionado de datos.
  - Algoritmos relajados.
- De paralelismo basado en dependencias de datos:
  - Paralelismo síncrono.
  - Dependencias en árbol o grafo.
  - Pipeline.
- De paralelización de esquemas secuenciales:
  - Divide y Vencerás.
  - Programación Dinámica.
- Recorridos de árboles: Backtracking y Branch and Bound.
- De múltiples tareas o trabajadores:
  - Bolsa de tareas.
  - Granja de procesos.
  - Maestro-Esclavo.

Otras fuentes de patrones

- Our pattern language: https://patterns.eecs.berkeley.edu/



Particionado y paralelismo de datos

Paralelismo y particionado de datos

- Aplicable cuando todos los datos se tratan de igual manera
  - Aplicable a GPUs
- Ejemplo:
  - Algoritmos numéricos donde los datos están en vectores o matrices
  - Posible procesamiento vectorial
  - Paralelismo asignando partes distintas del array a distintos elementos de proceso (directivas OpenMP)
- Memoria Compartida (Paralelismo de datos):
  - Distribución del trabajo entre los hilos
  - Paralelización automática o implícita
- Memoria Distribuida (Particionado de datos):
  - Distribución de los datos a los procesos
  - Paralelización explícita

Ejemplo – Suma de N números

- Suma de N números

```
int s = 0;
for(int i = 0; < n; i++) {
  s = s + a[i];
}
```

- Posible vectorización con simd.
- Paralelización automática: con opción de compilación si no hay dependencia de datos o el compilador detecta que se puede resolver

Ejemplo – Suma de N números

- Paralelismo implícito
  - El programador especifica cuáles son los datos
  - El sistema de decide cómo se reparten
- En OpenMP
  - Clausula `schedule` permite indicar cómo distribuir el trabajo: bloques contiguos, asignación cíclica, dinámica

```
int s = 0;
#pragma omp parallel for private(i) reduction(+:s)
for(int i = 0; < n; i++) {
  s = s + a[i];
}
```

Ejemplo – Suma de N números

- Paralelismo explícito

```
void sumaparcial(double *a, int n, int p) {
  int s=0;
  for(j=0; j<n/p; j++)
    s=s+a[j];
  a[0]=s;

void sumatotal(double *a, int n, int p) {
  double s=0;
  for(j=0; j<p; j+=n/p)
    s = s + a[j];
  return s;
}
```

$$t(n,p) = \frac{n}{p} + p$$

```
#pragma omp parallel for private(i,s)
for(i=0; i<p; i++)
  sumaparcial(&a[(i*n)/p],n,p);
S = sumatotal(a,n,p);
```

Ejemplo – Ordenación por rango

- Para cada elemento, contar cuántos hay menores que el
  - El rango nos dice la posición que ocupa el elemento ordenado
  - Coste $O(n^2)$
  - Fácilmente paralelizable

```
for(i=0; i<n; i++)
  for(j=0;j<n;j++)
    if (a[i]>a[j] || ((a[i] == a[j]) && (i > j)))
      r[i]+=1;

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

Ejemplo – Ordenación por rango

- Paralelismo implícito
  - Se divide el trabajo de cada bucle

```
#pragma omp parallel for private(i,j)
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    if (a[i]>a[j] || ((a[i] == a[j]) && (i > j)))
       r[i] += 1;

#pragma omp parallel for private(i)
for(i = 0; i < n; i++)
  b[r[i]] = a[i];
```

$$t(n,p) = \frac{n^2}{p} + \frac{n}{p}$$

Ejemplo – Ordenación por rango

- Paralelismo implícito
  - Se divide el trabajo de cada bucle

```
#pragma omp parallel for private(i,j)
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    if (a[i]>a[j] || ((a[i] == a[j]) && (i > j)))
       r[i] += 1;

#pragma omp parallel for private(i)
for(i = 0; i < n; i++)
  b[r[i]] = a[i];
```

¿Es posible colapsar los dos bucles?

Ejemplo – Ordenación por rango

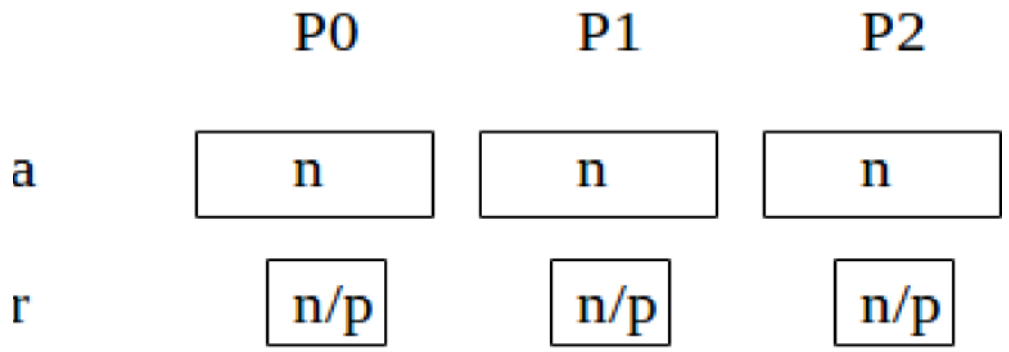
- Paralelismo implícito
  - Se divide el trabajo de cada bucle

```
#pragma omp parallel for private(i,j)
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    if (a[i]>a[j] || ((a[i] == a[j]) && (i > j)))
       r[i] += 1;

  // En este punto la posición del elemento i
  // ya ha sido calculada
  b[r[i]] = a[i];
}
```

Ejemplo – Ordenación por rango

- Paralelismo explícito
  - Asignación manual del trabajo a los hilos



Ejemplo – Ordenación por rango

- Paralelismo explícito

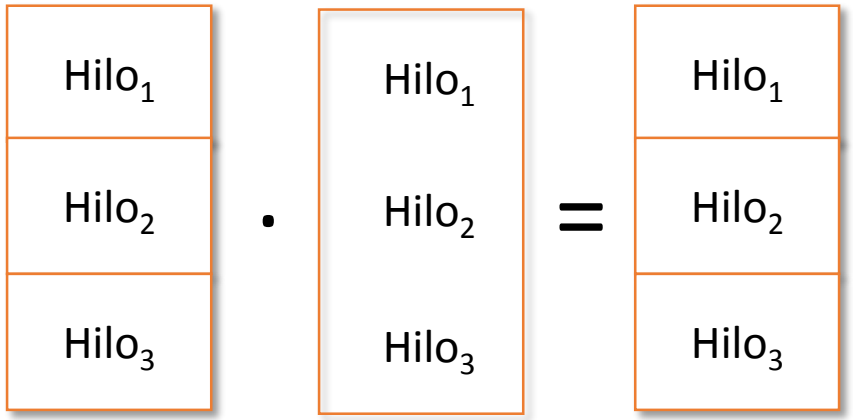
```
void rangoparcial(int *a, int *r, int *b,
                  int hilo, int n, int p) {
   for(int i = hilo * n / p; i < (hilo + 1) * n / p; i++) {
     for(j = 0; j < n; j++)
       if (a[i]>a[j] || ((a[i] == a[j]) && (i > j)))
         r[i] += 1;
     b[r[i]] = a[i];
   }
}

#pragma omp parallel for private(i)
for(i = 0; i < THREADS; i++)
  rangoparcial(a, r, b, n, i, THREADS)
```

Ejemplo – Multiplicación de matrices

- Con paralelismo implícito

```
#pragma parallel for private(i,j,k)
for(i=0; i<n; i++)
  for(j=0; j<n; j++) {
    c[i,j] = 0;
    for(k=0; k<n; k++)
      c[i,j]=c[i,j]+a[i,k]*b[k,j];
}
```



Ejemplo – Multiplicación de matrices

- Con paralelismo explícito

```
#pragma omp parallel for private(i)
for(i=0;i<p;i++)
  multiplicar(c,a,b,i)

void multiplicar(c,a,b,i) {
  for(j=(i*n)/p; j < ((i+1)*n)/p; j++)
    for(k=0;k<n;k++) {
      c[j,k]=0;
      for(l=0;l<n;l++)
        c[j,k]=c[j,k]+a[j,l]*b[l,k];
  }
}
```

Otros ejemplos de paralelismo de datos

- Producto escalar de dos vectores x e y de tamaño n:
  - Similar a la suma de n datos.
  - Asignación de bloques de tamaño n/p de x e y a los p hilos.
  - Suma (secuencial o paralela) de los productos parciales.
- Producto matriz-vector:
  - Dos bucles, paralelización del más externo.
  - Posible trabajo por bloques.

Particionado de datos

- Idea similar al paralelismo de datos para Memoria Distribuida
  - Además de dividir el trabajo hay que distribuir los datos
  - Implementación típica usando Paso de Mensajes
- El espacio de datos se divide en regiones adyacentes:
  - Se asignan a procesos distintos
  - Posible intercambio de datos entre regiones adyacentes
  - Más semejante al Paralelismo de Datos explícito
- Para obtener buenas prestaciones hay que intentar que el volumen de computación entre comunicaciones sea grande.
  - Paralelismo de grano grueso, efecto volumen-superficie.

Particionado de datos

- Ejemplo – Multiplicación de matrices
  - Volumen de computación n^3
  - Volumen de distribución de datos n^2
  - Cada hilo realiza el mismo cálculo que en la versión OpenMP

Ordenación por rango

```
void ordenrango(int *a, int n, int *r, int pid, int numprocs) {
  int i, j, k, t1 = n / numprocs; *b = NULL;

  MPI_Bcast(a, n, MPI_INT, 0, MPI_COMM_WORLD);

  for(k = 0, i = hilo * n / p; i < (hilo + 1) * n / p; i++, k++) {
    for(j = 0; j < n; j++)
      if (a[i]>a[j] || ((a[i] == a[j]) && (i > j)))
        r[k] += 1;
  }

  ...
```

Ordenación por rango

```
void ordenrango(int *a, int n, int *r, int pid, int numprocs) {
  ...
  if (pid == 0) {
    for(i = 1; i < numprocs; i++) {
      MPI_Recv(&r[tl * i], tl, MPI_INT, i, 10, …);
      b = (int *) malloc(sizeof(int) * n);
      for(i = 0; i < n; i++)
         b[r[i]] = a[i];
      for(i = 0; i < n; i++)
         a[i] = b[i];
      free(b);
    }
  } else {
    MPI_Send(r, tl, MPI_INT, 0, 10, MPI_COMM_WORLD);
  }
```

Ordenación por rango

- Coste
  - Volumen de computación n^2
  - Volumen de distribución de datos n
    - Todos los procesos necesitan el array que se va a ordenar completo

$$\underbrace{(p-1)(ts+tw\cdot n)}_{\text{Distribución inicial}}+\underbrace{tc\frac{n^2}{p}}_{\text{Cómputo}}+\underbrace{(p-1)\cdot(ts+\frac{tw\cdot n}{p})}_{\text{Distribución final}}$$

Algoritmos relajados

Algoritmos relajados

- Cada elemento de proceso trabaja de manera independiente.
- No hay sincronización ni comunicación, salvo las de distribuir datos y recoger resultados.
- Buenas prestaciones en Memoria Compartida y Paso de Mensajes.
- A veces a costa de no utilizar el mejor algoritmo paralelo.
- Fáciles de programar.
- Difícil encontrar algoritmos que se adecuen estrictamente al esquema.

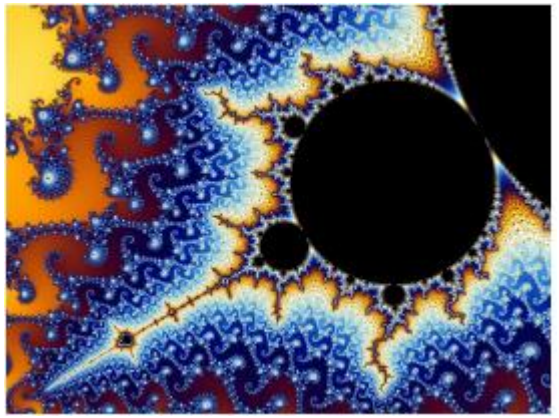
mandel.zip

Ejemplo – Fractales: el conjunto de Mandelbrot

- Algoritmo “embarrassingly parallel”
  - Se crear un número N de tareas, totalmente independientes
  - Particionado estático
- Para cada punto **c** en el plano complejo
  - Calcular la función color(c)

$$\begin{cases} z_0 = 0 \in \mathbb{C} & (\text{término inicial}) \\ z_{n+1} = z_n^2 + c & (\text{sucesión recursiva}) \end{cases}$$

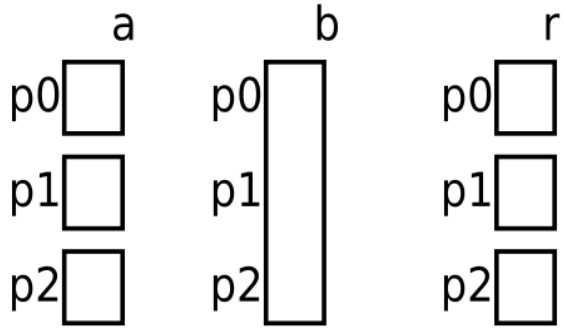
Los puntos cuya distancia al origen es superior a 2 (x2 + y2 > 2) no pertenecen al conjunto



* https://es.wikipedia.org/wiki/Conjunto_de_Mandelbrot
* https://www.dais.unive.it/~calpar/

Ordenación por rango

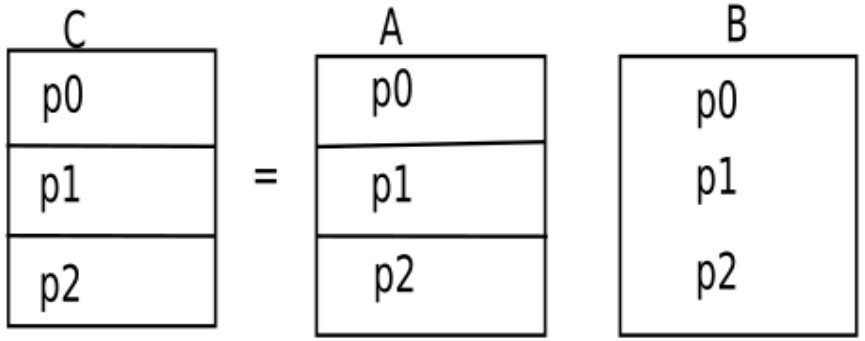
- Memoria Compartida: cada hilo calcula el rango de una parte de los elementos.
- Paso de Mensajes. Primero distribuir en la forma siguiente, y cada proceso calcula el rango de una parte de los elementos.



- Hay duplicación de datos pero se simplifica la programación y se obtienen buenas prestaciones.

Multiplicación de matrices

- Memoria Compartida: cada hilo calcula un bloque de filas de la matriz resultado.
- Paso de Mensajes, con distribución:



- cada procesador calcula las filas de C correspondientes a las filas de A que contiene.
- No es necesaria sincronización ni comunicación
  - salvo la distribución y acumulación (¿qué funciones se utilizarían?),
  - pero es más costoso el envío inicial al repetirse B en cada proceso.

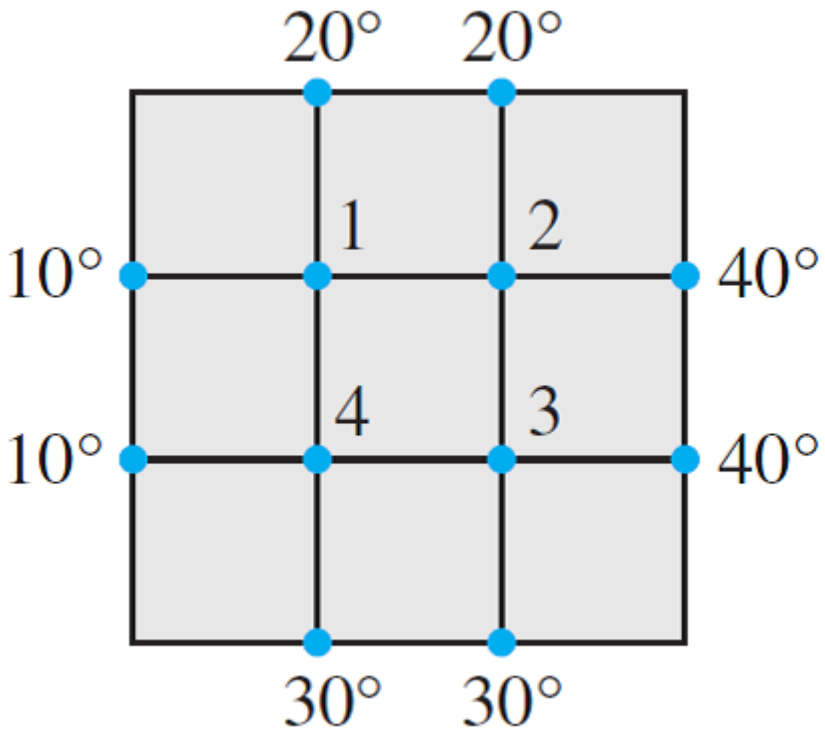
Descomposición geométrica/ Paralelismo síncrono

Características

- Se realizan iteraciones sucesivas:
  - Cada elemento de proceso realiza el mismo trabajo sobre una porción distinta de los datos.
  - Datos de una iteración se utilizan en la siguiente.
  - Tras cada iteración hay sincronización (local o global).
  - La iteración finaliza cuando se cumple algún criterio de convergencia o cuando se alcanza un número fijo de iteración (o tiempo límite).
- Las prestaciones están afectadas por la sincronización:
  - En Memoria Compartida buenas prestaciones.
  - En Paso de Mensajes bajan prestaciones por el coste de las comunicaciones.

Ejemplo – Iteración de Jacobi

- Por ejemplo, f(x, y) el calor de un punto de una placa de metal
  - Dada una placa de metal para la que conocemos la temperatura de los bordes, calcular cuál es la temperatura en los puntos del interior.



Ejemplo – Iteración de Jacobi

- Se calcula la ecuación de diferencias:

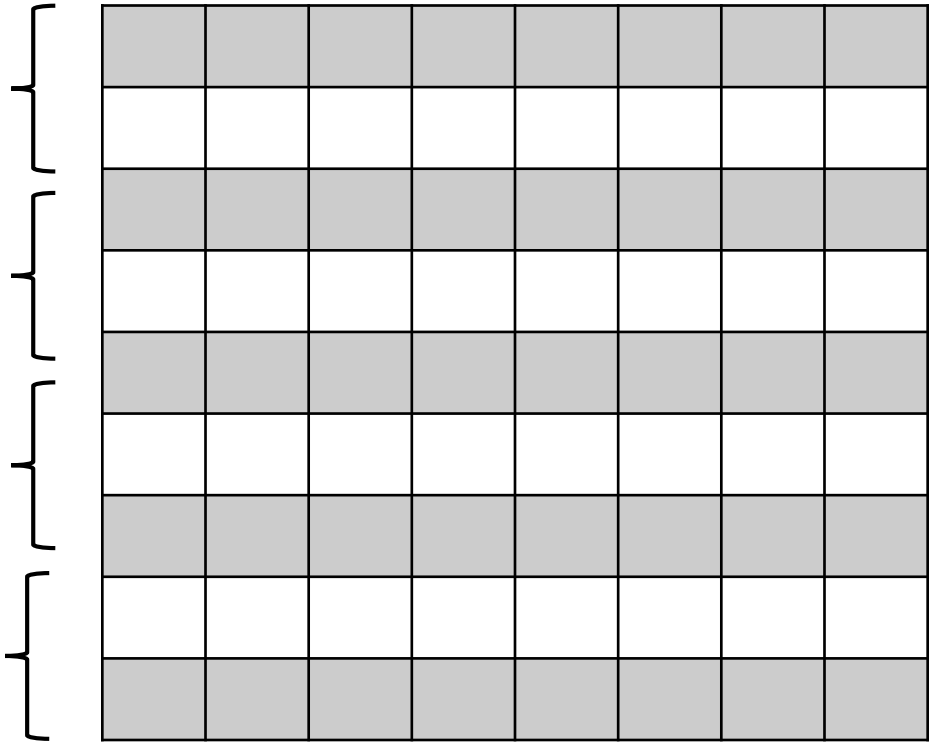
$$V^t(i,j) = \frac{V^{t-1}(i-1,j) + V^{t-1}(i+1,j) + V^{t-1}(i,j-1) + V^{t-1}(i,j+1)}{4}$$

- Converge gradualmente a una solución cada vez más precisa.
- Para obtener una solución más precisa aumentar el número de puntos.
- Una iteración tras otra secuencialmente, pero dentro de cada iteración paralelismo.
- Entre iteraciones se necesita comunicación local.
- Para comprobar condición de fin comunicación global.

Paralelismo implícito

jacobi_omp.c

- Entre cada bucle paralelo, la sincronización es implícita
- Hay paralelismo de datos en los bucles
- División del trabajo

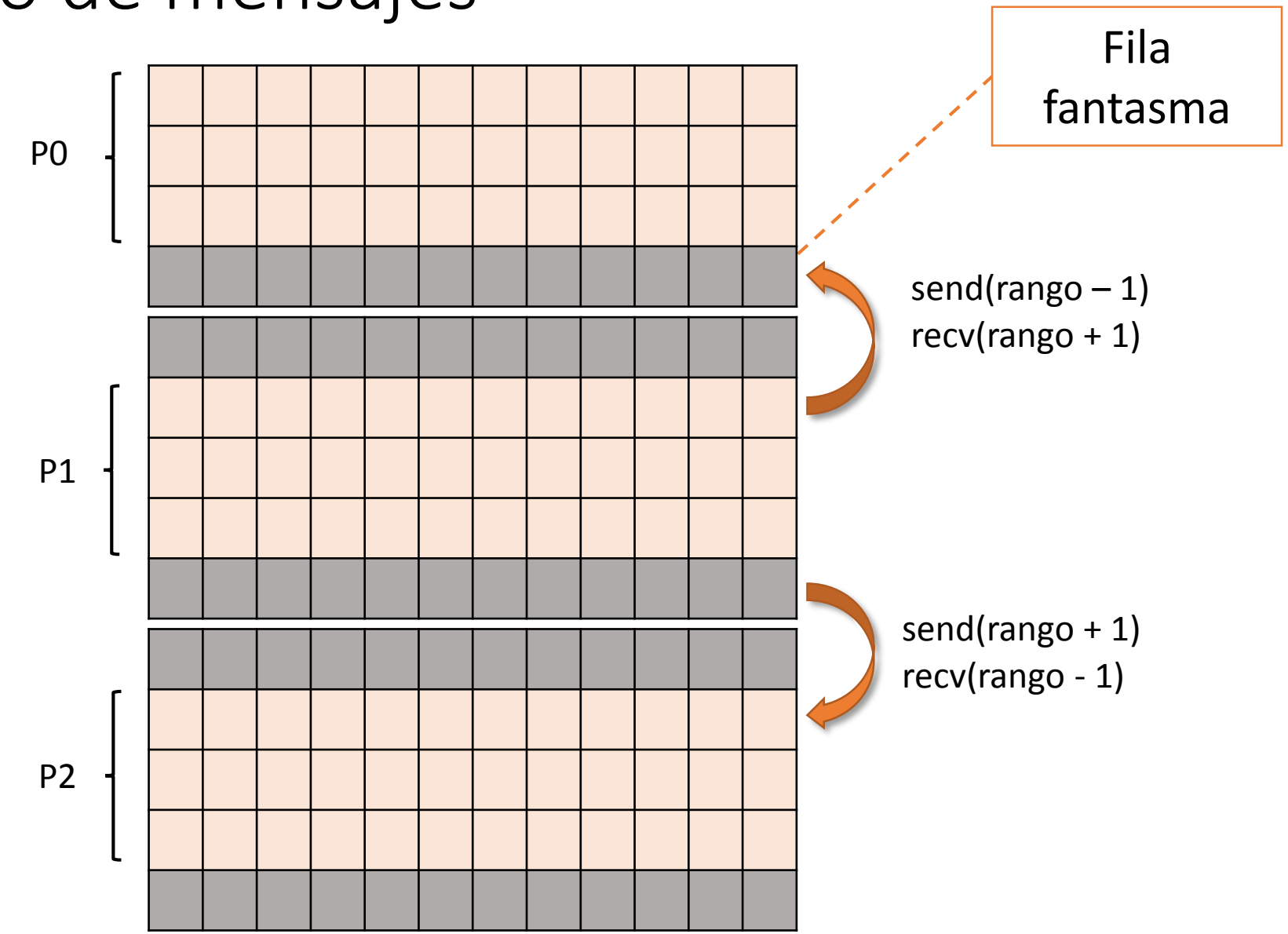


Paso de mensajes

- El array de datos *n* filas
- Cada proceso tiene un array con tl = n/p + *2* filas
- Las dos filas extras son para almacenar “los bordes”
  - Filas fantasma
- Los intercambios consisten en comunicarse los resultados del borde

Paso de mensajes

jacobi_mpi.c



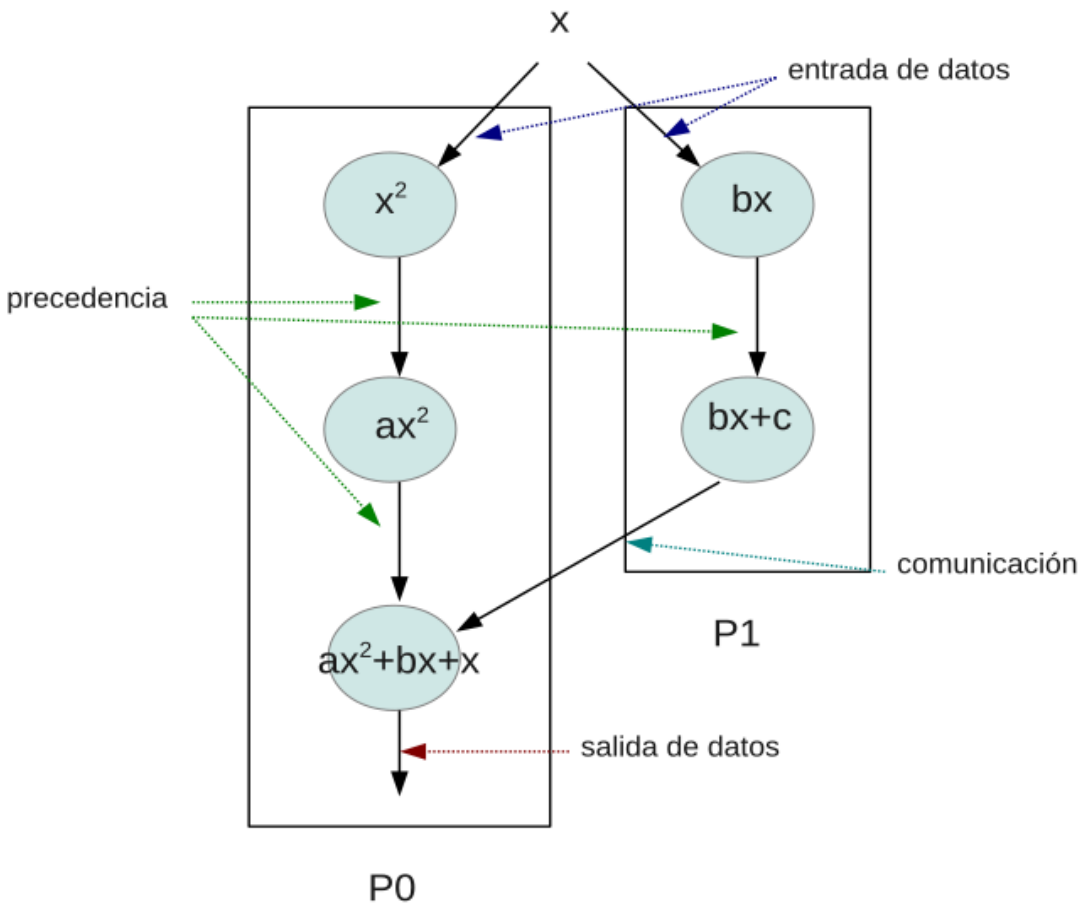
Paso de mensajes

- ¿Cómo se puede optimizar introduciendo comunicaciones asíncronas?

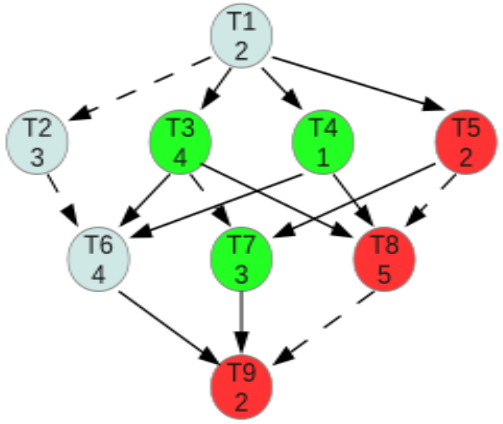
Dependencias en árbol o grafo

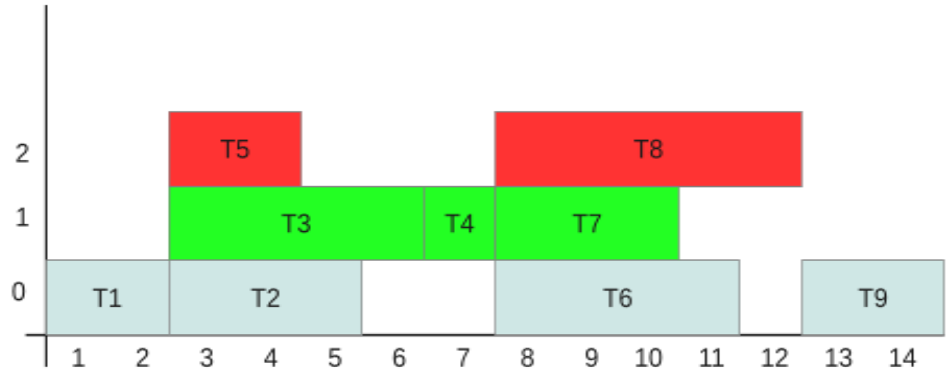
Grafos de dependencias

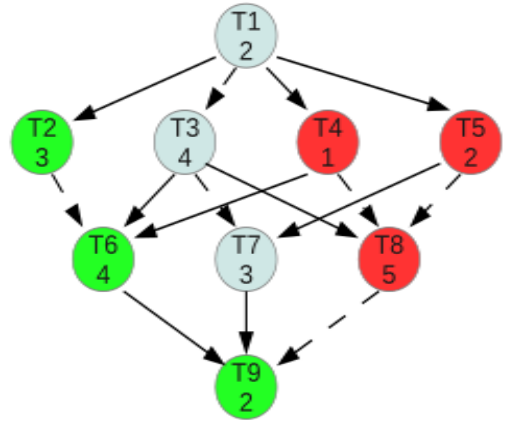
- Un Grafo de Dependencias es un grafo dirigido acíclico
- Nodos representan tareas
- Una arista de un origen a un destino representa que para poder realizarse la tarea destino tiene que haberse ejecutado la origen (posiblemente por una dependencia de datos).
- Los nodos pueden etiquetarse con un valor que representa el coste de la tarea. Las aristas pueden etiquetarse con valor que representa el coste de la comunicación

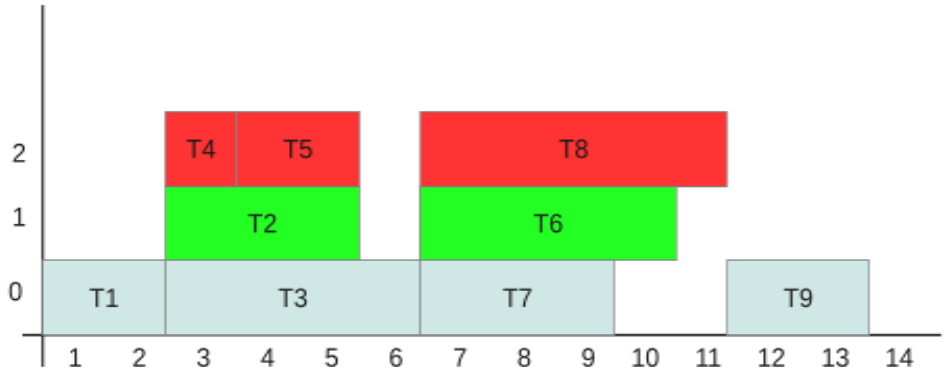


Asignación de tareas









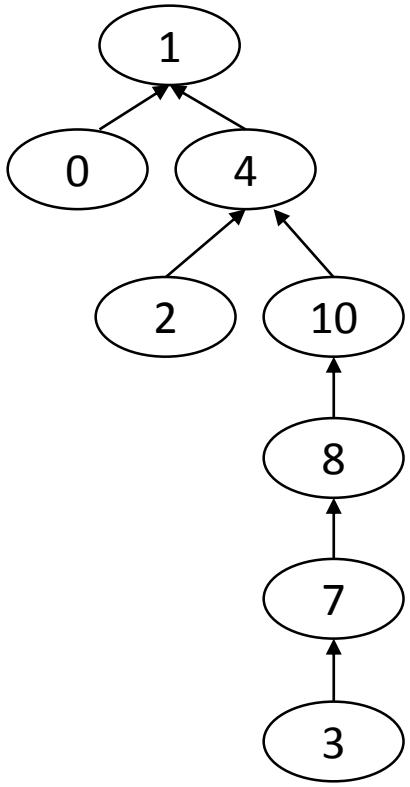
- Hay que asignar las tareas a los elementos de proceso.
- Distintas asignaciones pueden originar tiempos distintos.
- El problema de la asignación óptima es NP.

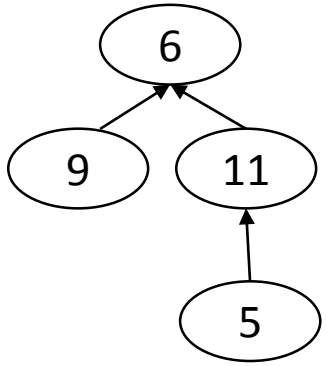
Asignación de tareas

- Dentro de un elemento de proceso hay que elegir el orden de ejecución de las tareas.
- Desbalanceo de la carga: balancear cálculo y comunicaciones.
- Tiempos de espera: por dependencias entre tareas.
- Si granularidad fina: muchas tareas, lo que posibilita el balanceo pero genera más comunicaciones.
- Si granularidad gruesa menos comunicaciones pero más difícil el balanceo.

Clases de equivalencia

- Relación de equivalencia expresada como árboles
  - Elementos de un conjunto que comparten cierta propiedad





Problema: encontrar el representante de la clase a la que pertenece cada nodo

Nodo	0	1	2	3	4	5	6	7	8	9	10	11
Padre	1	1	4	7	1	11	6	8	10	6	4	6

Clases de equivalencia

- Algoritmo secuencial
  - Sustituir en cada iteración el padre de un nodo por el padre de su padre

```
int cambio = 1;
while (cambio) {
   cambio = 0;
   for(int i = 0; i < n; i++) {
      if (array[i] != array[array[i]]) {
         array[i] = array[array[i]];
         cambio = 1;
      }
   }
}
```

Clases de equivalencia

- Algoritmo paralelo

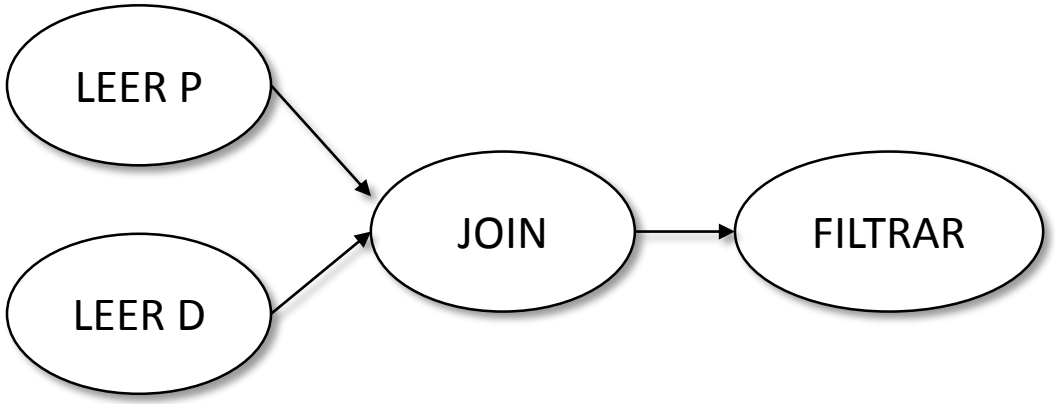
```
int cambio = 1;
while (cambio) {
    cambio = 0;
    memcpy(tmp, array, n);

    #pragma omp parallel for private(i)
    for(int i = 0; i < n; i++) {
        if (array[i] != array[array[i]]) {
            tmp[i] = array[array[i]];
            cambio = 1;
        }
    }
}
```

Ejemplo

- Procesador de consultas (SQL) paralelo

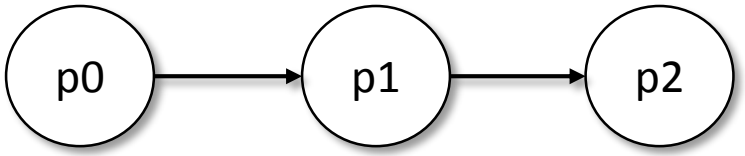
```
SELECT nombre, apellidos, teléfono, direccion
FROM persona P JOIN direcciones D ON P.dni = D.dni_persona
WHERE edad < 18
```



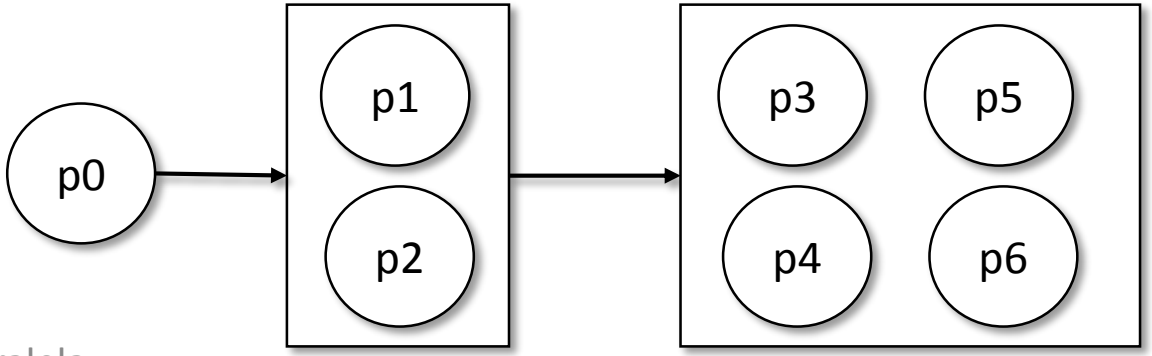
Computación pipeline

Computación pipeline

- Descomponer un problema en una serie de tareas sucesivas
  - Los datos fluyen por la estructura de tareas
  - Cuando una tarea termina procesando un dato se lo pasa al siguiente

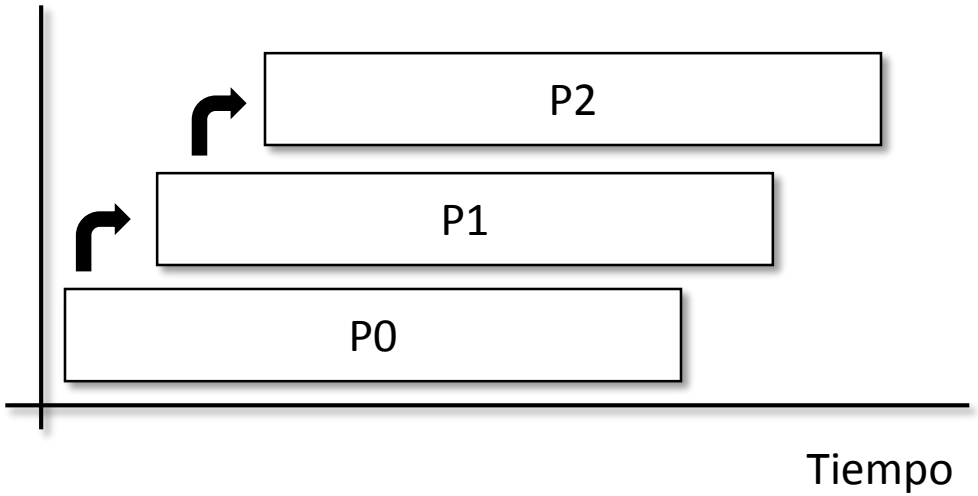


  - Cada tarea puede tener un peso diferente y ser preferible dedicar distinto número de procesadores a cada tarea



Computación pipeline

- Útil cuando:
  - No hay un único conjunto de datos a tratar sino una seria de conjuntos de datos, que entran a ser computados uno tras otro
  - Una tarea puede empezar antes que la tarea previa haya terminado. La tarea previa debe enviar los datos necesarios tan pronto como estén disponibles

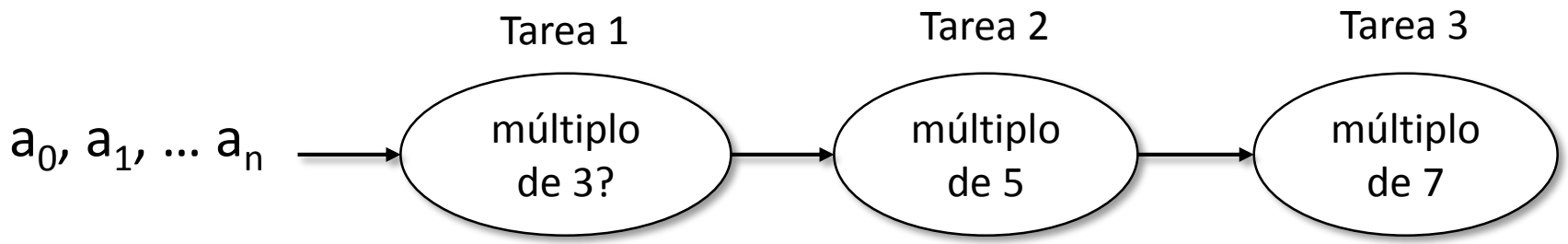


- El coste es mayor que el de la tarea más costosa

Ejemplo – Divisibilidad por P primos

- Lista de números primos P_0, P_1, … P_{m-1}
- Secuencia de enteros a_0, a_1, … a_n
- ¿Qué enteros son múltiplos de todos los primos?

- Estructura de tareas
  - T2 puede comenzar tan pronto como T1 ha procesado a0



Resolución sistema de ecuaciones triangular inferior

- Resolución por sustitución progresiva

$$
\begin{aligned}
3x_1 + 2x_2 - 2x_3 + 4x_4 &= -5,\\
3x_2 - 5x_3 - 3x_4 &= 0,\\
4x_3 + x_4 &= -3,\\
2x_4 &= 6.
\end{aligned}
$$

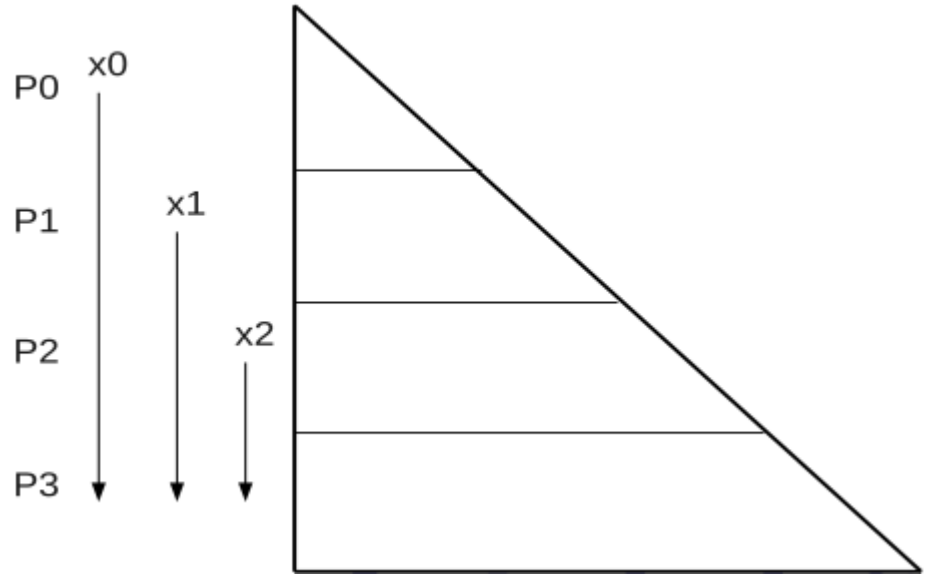
$$
\begin{aligned}
x_4 &= \frac{6}{2} = 3,\\
x_3 &= \frac{-3 - x_4}{4} = -\frac{3}{2},\\
x_2 &= \frac{5x_3 + 3x_4}{3} = \frac{1}{2},\\
x_1 &= \frac{-5 - 2x_2 + 2x_3 - 4x_4}{3} = -7.
\end{aligned}
$$

Resolución sistema de ecuaciones triangular inferior

- Resolución por sustitución progresiva

$$
\begin{array}{lllllll}
a_{00}x_0 & & & & & = & b_0 \\
a_{10}x_0 & + & a_{11}x_1 & & & = & b_1 \\
 & & & \cdots & & & \\
a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + \cdots + & a_{n-1,n-1}x_{n-1} & = & b_{n-1}
\end{array}
$$

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{ij}x_j}{a_{jj}}$$



Resolución sistema de ecuaciones triangular inferior

```
if (rank == 0) {
  x = b / a[0];
  MPI_Send(&x, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
} else if ( rank == n – 1 ) {
  for(i = 0; i < n – 1; i++) {
    MPI_Recv(&x, 1, MPI_DOUBLE, n – 2, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    r += a[i] * x;
  }
  x = (b – r) / a[n – 1];
} else if ( (rank != 0) && (rank != n - 1) ) {
  for(i = 0; i < rank; i++) {
    MPI_Recv(&x, 1, MPI_DOUBLE, rank - 1, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&x, 1, MPI_DOUBLE, rank + 1, i, MPI_COMM_WORLD);
    r += a[i] * x;
  }
  x = (b – r) / a[n – 1];
  MPI_Send(&x, 1, MPI_DOUBLE, rank + 1, rank, MPI_COMM_WORLD);
}
```

Divide y vencerás

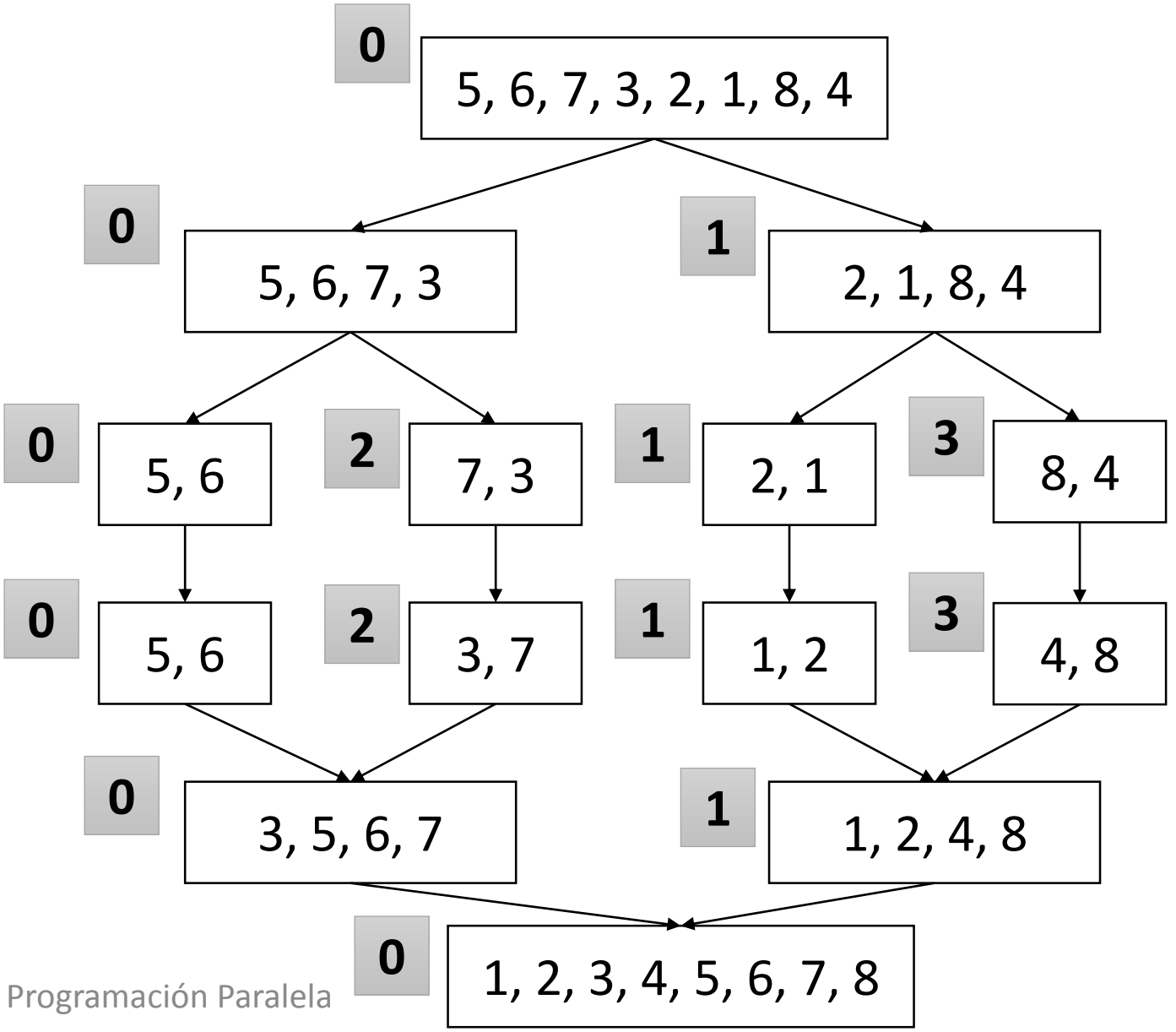
Divide y vencerás

- Esquema algorítmico que consiste en:
  - Resolver un problema dividiéndolo en subproblemas de las mismas características
  - Se resuelven los subproblemas
  - Combinar las soluciones
  - División recursiva en subproblemas

Divide y vencerás – Paralelismo

- La solución de los subproblemas s_i se puede hacer en paralelo.
- La división debe producir subproblemas de coste balanceado.
- La división y la combinación implican comunicaciones y sincronización.
- Si consideramos un árbol de llamadas recursivas, el Grafo de Dependencias viene determinado por el árbol, y podemos considerar que seguimos el paradigma de programación paralela en árbol.

Ordenación por mezcla



Ordenación por mezcla

- OpenMP – con tareas

```
#pragma omp parallel
{
 #pragma omp single
 mergesort(data, n, tmp);
}
```

```
void mergesort(int * X, int n, int * tmp)
{
  if (n < MAX_SIZE) {
    mergesort_sec(X, n, tmp);
    return;
  }
  #pragma omp task firstprivate (X, n, tmp)
  mergesort(X, n/2, tmp);

  #pragma omp task firstprivate (X, n, tmp)
  mergesort(X+(n/2), n-(n/2), tmp);

  #pragma omp taskwait
  merge(X, n, tmp);
}
```

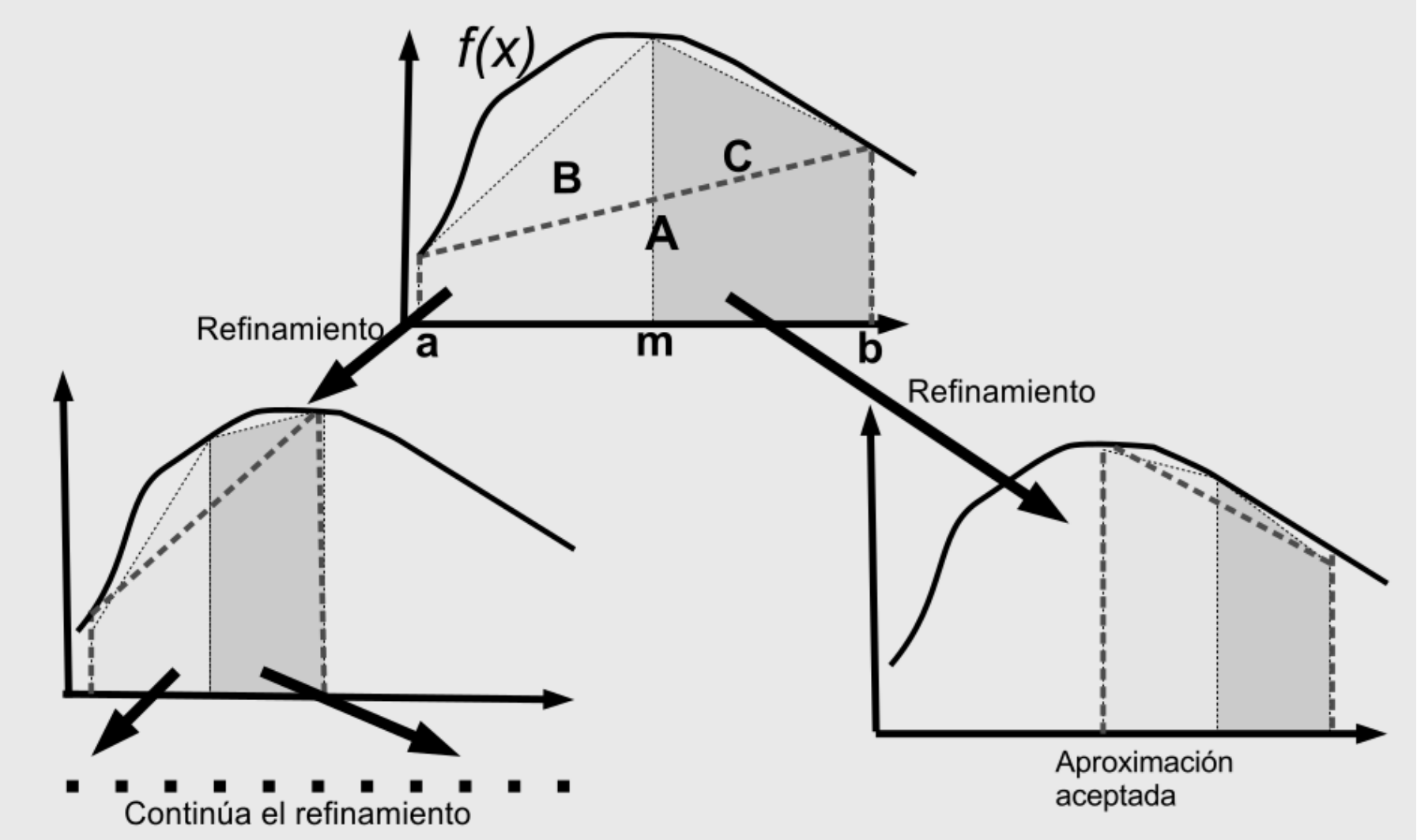
Integración numérica por cuadratura adaptativa

- Se desea aproximar:

$$I = \int_a^b f(x)dx$$

- **Cuadratura adaptativa**:
  - Número variable de intervalos.
  - Dividir ([a,b]) en dos mitades con áreas trapezoidales B y C .
  - Si A − (B + C) < Tolerancia , Aproximación=B + C .
  - Sino, aplica el procedimiento recursivamente a cada mitad.

Integración numérica por cuadratura adaptativa



Integración numérica por cuadratura adaptativa

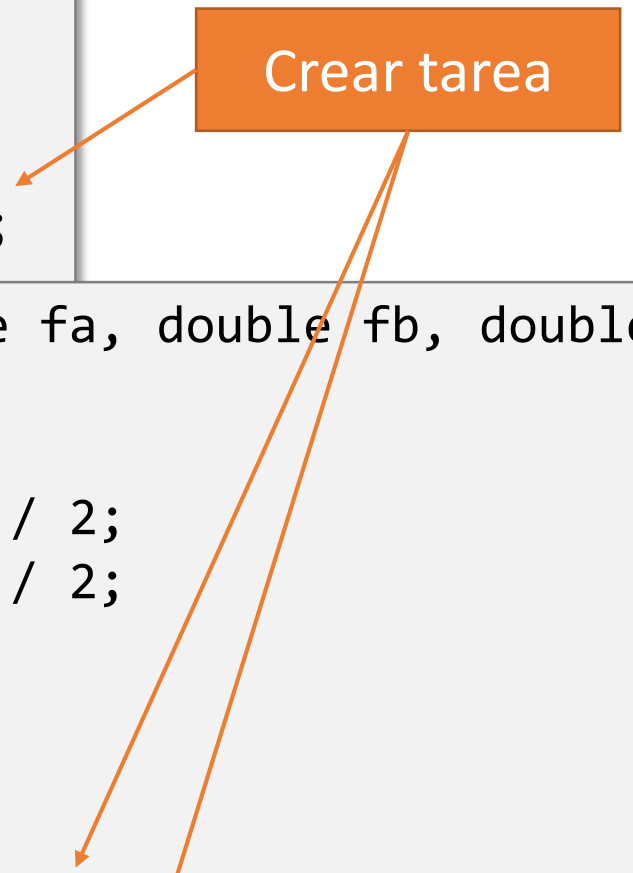
```
void integrar(double a, double b) {
  fa = f(a);
  fb = f(b);
  area = (fa + fb) * (b – a) / 2.0;
  return computa_intervalo(a, b, fa, fb, area,);
}
```

```
void intervalo(double a, double b, double fa, double fb, double aprox) {
  double m = (a + b) / 2;
  double fm = f(m);
  double area_izq  = (fa + fm) * (m – a) / 2;
  double area_dcha = (fm + fb) * (b – m) / 2;
  double aprox2 = area_izq + area_dcha;
  if (aprox – aprox2 < umbral)
     return aprox;
  else {
    return intervalo(a, m, fa, fm, area_izq) +
           intervalo(m, b, fm, fb, area_dcha);
  }
}
```

Integración numérica por cuadratura adaptativa

```
void integrar(double a, double b) {
  fa = f(a);
  fb = f(b);
  area = (fa + fb) * (b – a) / 2.0;
  return computa_intervalo(a, b, fa, fb, area);
}
```

```
void intervalo(double a, double b, double fa, double fb, double aprox) {
  double m = (a + b) / 2;
  double fm = f(m);
  double area_izq  = (fa + fm) * (m – a) / 2;
  double area_dcha = (fm + fb) * (b – m) / 2;
  double aprox2 = area_izq + area_dcha;
  if (aprox – aprox2 < umbral)
     return aprox;
  else {
    return intervalo(a, m, fa, fm, area_izq) +
           intervalo(m, b, fm, fb, area_dcha);
  }
}
```



Recorridos de árboles

Recorridos de árboles

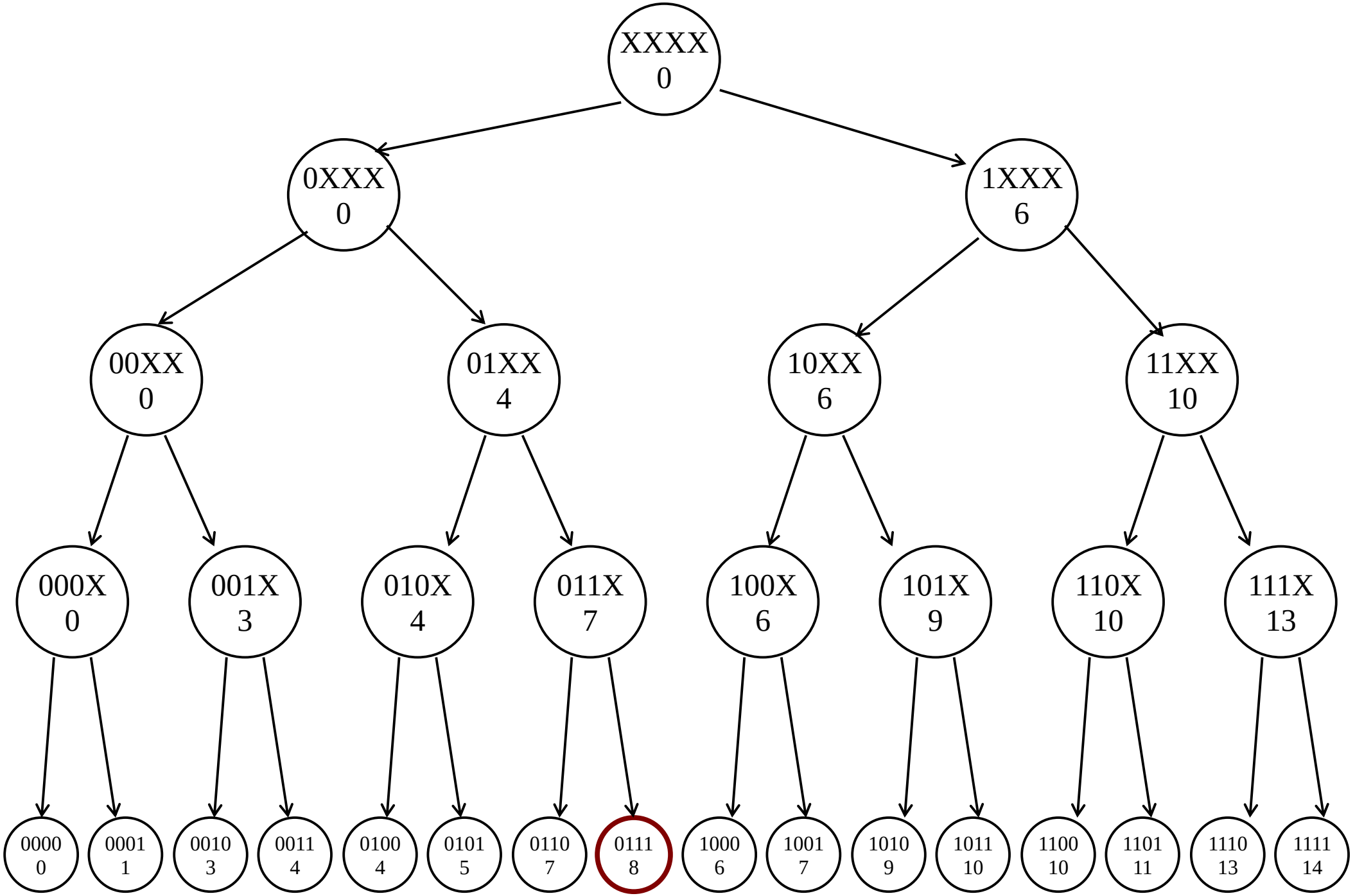
- Problemas de búsqueda en un espacio de posibles soluciones.
- Problemas de distintos tipos: búsqueda de una única solución, de todas las soluciones, de la solución optima, etc.
- Esquemas típicos de recorrido del árbol:
  - Backtracking, Branch and Bound, etc.
- Se descompone el espacio de búsqueda en subespacios más pequeños y se asignan a distintos componentes computacionales(hilos, procesos).
- La generación de subproblemas puede ser estática o dinámica:
  - Todas las tareas se generan inicialmente, o se generan nuevas tareas durante el recorrido del árbol.
- La asignación de tareas puede ser estática o dinámica:
  - Se decide inicialmente qué subproblemas resuelve cada elemento de proceso, o se asigna trabajo según la velocidad de cada elemento de proceso

Suma de un subconjunto

- Dado un conjunto de números *C* y un entero *v*, se quiere encontrar el subconjunto cuyos valores suman *v*.

C = 6, 4, 3, 1

V = 8


XXXX
0
0XXX
0
1XXX
6
00XX
0
01XX
4
10XX
6
11XX
10
000X
0
001X
3
010X
4
011X
7
100X
6
101X
9
110X
10
111X
13
0000
0
0001
1
0010
3
0011
4
0100
4
0101
5
0110
7
0111
8
1000
6
1001
7
1010
9
1011
10
1100
10
1101
11
1110
13
1111
14

Suma de un subconjunto

- ¿De qué maneras podemos paralelizar el recorrido del espacio de soluciones?

Suma de un subconjunto

- Generación secuencial inicial de un número fijo de tareas.
- Distribución de tareas:
  - Estática. Balancear la distribución.
    - Si se sabe el coste de cada tarea se puede calcular distribución óptima (problema NP).
    - Si no se sabe, se pueden generar más tareas que procesos y asignarlas aleatoriamente o cíclicamente.
  - Dinámica. Más difícil la gestión.
    - Memoria Compartida, acceso en exclusiva a descriptores de las tareas.
    - Paso de Mensajes, proceso que gestiona la bolsa de tareas (Maestro), y coste alto de la gestión por comunicaciones.
- Generación dinámica de tareas: se generan tareas y al tratarlas se generan nuevas tareas.
  - Coste de gestión de la bolsa de tareas.
  - La granularidad de las tareas debe ser alta

Maestro/Esclavo y Granja de procesos

Introducción

- Una serie de elementos de proceso que constituyen una Granja de procesos y que pueden ser del mismo tipo (Trabajadores replicados).
- Normalmente trabajan resolviendo tareas que se encuentran en una estructura que contiene la definición de las tareas a realizar: Bolsa de tareas.
- Y la gestión de la bolsa la hace un proceso distinguido (Maestro), que atiende las peticiones y los envíos de otros procesos (Esclavos).
- Se tiene así el paradigma Maestro-Esclavo.

Bolsa de tareas

- Bolsa de tareas: conjunto de descriptores de tareas; cada descriptor especifica una computación.
- En Memoria Compartida:
  - una estructura centralizada de la que los trabajadores toman trabajos y posiblemente depositan otros nuevos.
  - Acceso en exclusión mutua.
- En Paso de Mensajes:
  - la estructura está en la memoria de uno o varios procesos,
  - la petición y depósito de tareas conllevan comunicación.

- Contención:
  - Bolsa centralizada. Cuantos más procesos mayor contención.
- Balanceo:
  - Si se descentraliza la bolsa hay mayor desbalanceo y menor contención ⇒ compromiso entre contención y balanceo.
- Terminación:
  - El test de terminación es global ⇒ sincronización.

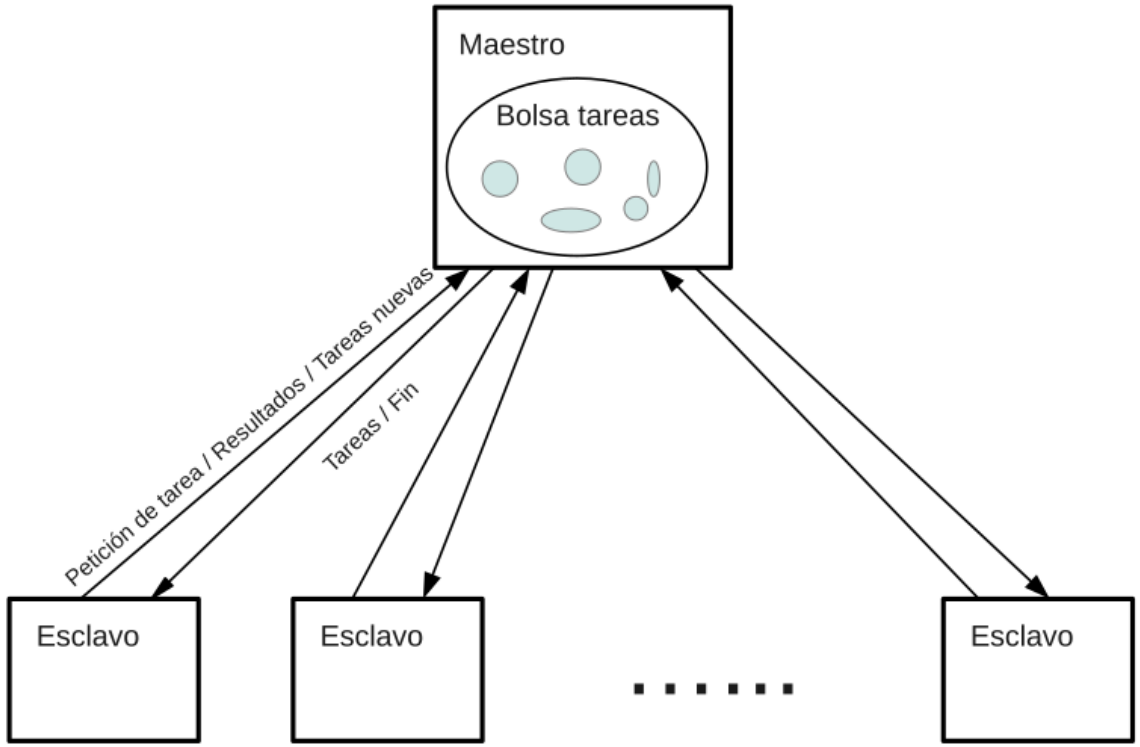
Esquema OpenMP

- Bolsa de tareas y variables tareas y acabados son globales.
- Inicialmente 1 tarea y todos los trabajadores inactivos
  - (acabados = trabajadores).
- Los hilos trabajan mientras queden tareas u otros hilos trabajando
  - (pueden generar nuevas tareas).
- No es necesario un Maestro. Todos los hilos hacen el mismo trabajo.
- Es necesario acceder en exclusión mutua a las variables compartidas.

```
omp set num threads(trabajadores );
acabados = trabajadores ;
#pragma omp parallel private(fin,n)
{
  fin=FALSO;
  Mientras no fin hacer
    #pragma omp critical
    {
      Si tareas = 0 y acabados = trabajadores entonces
         fin=VERDADERO;
      sino Si tareas != 0 entonces
         acabados = acabados – 1;
         tareas = tareas – 1;
         tomar tarea de la bolsa;
      finsi
    }
    Si tomada tarea entonces
      resolver tarea y generar n nuevas tareas;
      #pragma omp critical
      {
         tareas = tareas + n;
         acabados = acabados + 1;
         insertar las n tareas en la bolsa;
      }
    finsi
  finmientras
}
```

Maestro/Esclavo

- Comunicaciones entre el maestro y los esclavos.
- El maestro conoce la condición de fin y la comunica a los esclavos.



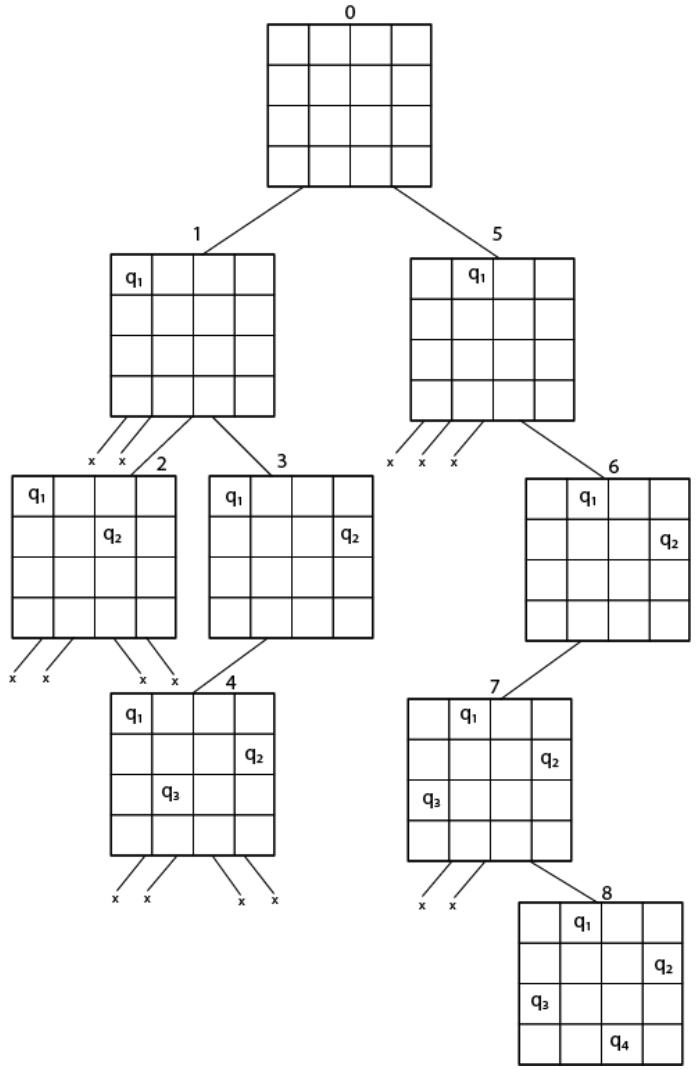
Esquema “MPI”

- Suponemos que inicialmente hay tareas para todos los esclavos.
- El Maestro recibe resultados y envía nuevos trabajos.
- Cuando no le quedan más trabajos manda señales de fin a todos los esclavos.
- Los Esclavos reciben tareas, las resuelven, mandan el resultado y reciben nuevas tareas, hasta que reciben la señal de fin.

```
// Se dispone de p procesos esclavos
// y de t tareas
Si proceso maestro entonces
  asignar un trabajo a cada esclavo;
  t = t – p;
  Mientras t != 0 hacer
    Recibe( resultado );
    Envia( trabajo , esclavo del que ha recibido );
    t = t – 1;
  finmientras
  Mientras p != 0 hacer
    Recibe( resultado );
    Envia(marca de fin, esclavo del que ha recibido);
    p = p – 1;
  finmientras
sino
  Recibe( trabajo , maestro );
  resolver trabajo ;
  Envia( resultado , maestro );
  Recibe( trabajo o marca de fin, maestro );
  Mientras no recibida marca de fin hacer
    Resolver trabajo ;
    Envia( resultado , maestro );
    Recibe( trabajo o marca de fin, maestro );
  finmientras
finsi
```

Problema de las N reinas

- ¿Cómo colocar *n* reinas en un tablera de *n* x *n* sin que sea maten entre sí?
- Soluciones:
  - Recursiva
  - Iterativa



* Visualizador: https://haseebq.com/n-queens-visualizer/

Ejercicios

- Problema de las reinas
  - Versión con tareas
  - Versión asíncrona
  - Programación híbrida
- Mandelbrot en MPI