

Tema-2-AC.pdf



patrivc



Arquitectura de Computadores



2º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada



Descarga la APP de Wuolah.
Ya disponible para el móvil y la tablet.





**KEEP
CALM
AND
ESTUDIA
UN POQUITO**



Descarga la APP de Wuolah.

Ya disponible para el móvil y la tablet.

Available on the
App Store

GET IT ON
Google Play

TEMA 2: Programación paralela

Lección 4: Herramientas, estilos y estructuras en programación paralela

4.1. Problemas que plantea la programación paralela al programador. Punto de partida.

La programación paralela plantea ciertos problemas que no aparecían en la realización de un programa secuencial. **Ejemplo:** en un programa que es una suma de números, tenemos que sumar 8 números (0,1,2,3,4...). Tenemos que ver qué cosas podemos hacer, como sumar los números de dos en dos (0+1, 2+3, 4+5...). Después hay que agrupar las sumas según el número de procesadores que tengamos y una vez que tengan esas sumas, se deberían comunicar entre ellas. Es decir los **pasos que tenemos que seguir** **son:** identificar que elementos tenemos en paralelo, asignarlos a distintas hebras y distintos procesos y sumar los resultados de cada una de esas hebras (para obtener el resultado final), es decir, comunicar las hebras

La **programación paralela** plantea **problemas** inherentes a la misma que no se dan en la programación secuencial, como la división del cómputo total en tareas independientes, la agrupación de dichas tareas en procesos o hebras, la asignación de estos procesos y hebras a los procesadores y la sincronización y comunicación entre estos procesos. Estos problemas deben ser abordados tanto por la herramienta de programación como por el programador o por ambos.

Punto de partida

Para hacer un programa paralelo podemos partir de distintas situaciones: en primer lugar, podemos suponer que tenemos solo una versión secuencial de la que tenemos que partir. En otros casos tenemos una descripción de la aplicación, entonces hay que partir de estas definiciones/descripciones. Como apoyo tenemos programas paralelos que resuelven problemas con estructura similar al que tenemos que resolver. También se utilizan llamadas a funciones paralelizadas u optimizadas para su ejecución en máquinas paralelas (BLAS, LAPACK...)

Modos de programación MIMD

Al trabajar con programación paralela en arquitecturas MIMD podemos hacerlo de forma SPMD, parallelizando un solo programa, (Single Program Multiple Data) y MPMD (Multiple Program Multiple Data), parallelizando la ejecución de varios programas que a su vez están programados de forma paralela.

SPMD (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. Se ejecuta en todos los nodos el mismo programa. El programa puede ser el mismo, pero el conjunto de instrucciones puede ser diferente (ejemplo: el identificador siempre va a ser distinto). Tenemos un mismo programa, pero múltiples conjuntos de datos.

MPMD (Paralelismo de tareas o funciones): Los códigos que se ejecutan en paralelo se obtienen compilando programas independientes. En este caso, la aplicación a ejecutar se divide en unidades independientes y cada una de ellas trabaja con un conjunto de datos y se asigna a un procesador distinto. El programa se divide en distintos procedimientos, distintas partes, que se ejecutan en distintos nodos de la máquina. Son programas distintos que no generan comunicaciones entre los distintos nodos. Los datos pueden ser diferentes.

4.2. Herramientas para obtener el código paralelo

Para crear programas de ejecución paralela podemos utilizar las tres técnicas ordenadas de mayor a menor abstracción:

Compiladores paralelos: Extraen automáticamente el paralelismo de los programas que compilan, de forma que el programador no tiene qué explicitarla. Generan un código paralelo, extraen el paralelismo que está implícito en los programas que queremos ejecutar.

Lenguajes paralelos y API de directivas: Existen lenguajes paralelos como Occam, Ada o Java que tienen una serie de construcciones del lenguaje que expresan explícitamente el paralelismo y se permite usar una serie de funciones que permiten la sincronización, la comunicación, generación... Dentro de este nivel, tenemos los lenguajes secuenciales que se acompañan de directivas o funciones para realizar o completar algunas de las funciones de comunicación o generación de hebras o procesos. Esto es las directivas de OpenMP permiten indicar cómo paralelizar el programa en el código a gusto del programador.

API de funciones: Las APIs de alto nivel como MPI o Pthreads que permiten parallelizar la programación mediante paso de mensajes y otras técnicas de abstracción alta. Son lenguajes secuenciales que se acompañan de funciones.

Las herramientas de parallelización permiten de forma implícita o explícitamente, localizar el paralelismo de los programas dividiéndolos en tareas independientes, asignar tareas (la carga de trabajo) a los procesos y hebras, crear y terminar estos procesos y hebras y comunicarlos y sincronizarlos.

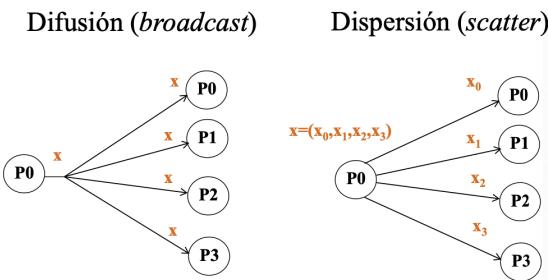
El mapping, la asignación de las diferentes tareas a procesos y threads, puede hacerlo el programador, la herramienta de programación paralela o el propio sistema operativo (fundamentalmente es este último el que lo realiza).

Comunicaciones colectivas

Distinguimos entre diferentes tipos de comunicaciones entre procesos y hebras:

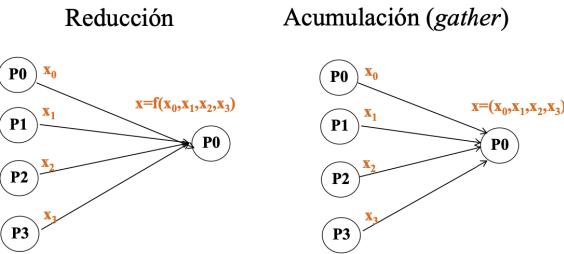
Comunicación uno a todos

Se caracterizan porque un único proceso envía un mensaje a varios procesos al mismo tiempo. Esto puede conseguirse mediante una difusión (broadcast) del mensaje, que envía un mensaje x a todos los procesos a la vez, o mediante una dispersión (scatter), que envía un mensaje x_i a cada proceso i .



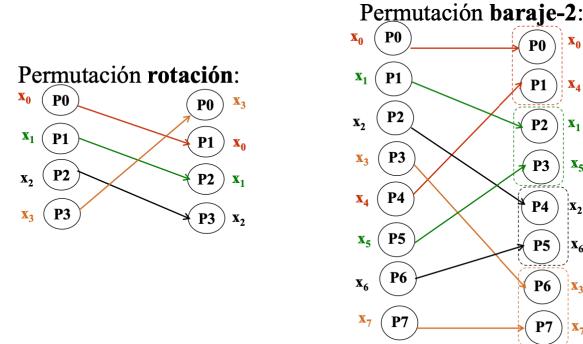
Comunicación todos a uno

Se caracterizan porque un único proceso recibe un mensaje a partir de los mensajes enviados por varios procesos. Esta recepción puede hacerse mediante reducción cuando los mensajes recibidos son argumentos de una función o mediante acumulación (gather), cuando se recogen indistintamente todos los mensajes.



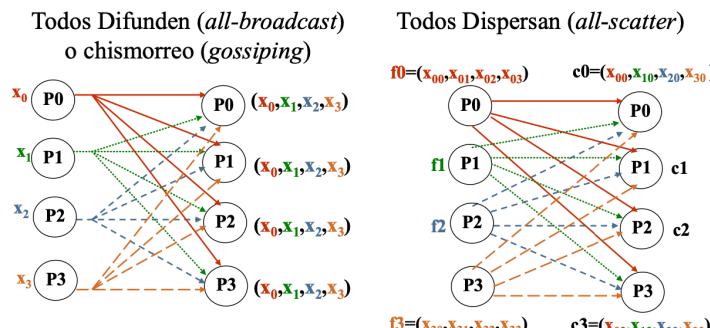
Comunicación múltiple uno a uno

Se produce cuando varios procesos ejecutándose paralelamente se comunican uno a uno entre sí. Esta comunicación puede hacerse por permutaciones de rotación o por permutaciones de baraje-x, en las que cada proceso i manda su mensaje al proceso $i \cdot k \bmod x$ para un total de k procesos.



Comunicación todos a todos

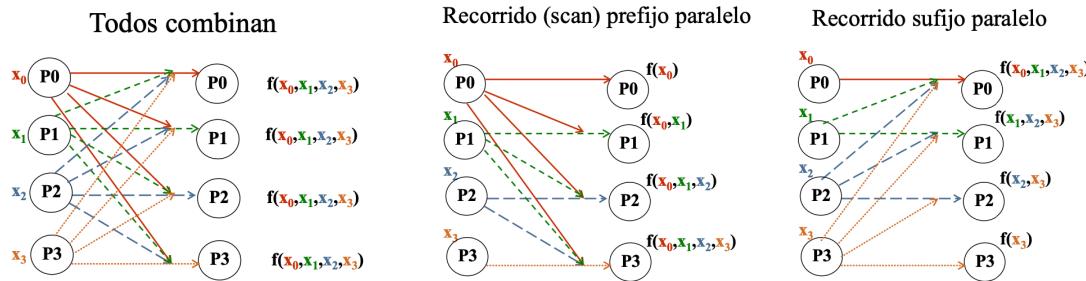
En este tipo de comunicación todos los procesos se comunican con todos. Esto se puede hacer mediante un sistema en el que todos difunden, cada proceso realiza un Broadcast a los demás, tenemos el dato que envía cada proceso de partida, al final tendremos el conjunto de datos que estaba distribuido en los procesadores (all-broadcast), también conocido como chismorreo (gossiping), o mediante un sistema en el que todos dispersan (all-scatter) cada hebra envía un dato diferente a cada proceso.



c = columna matriz
f = fila matriz

Comunicaciones compuestas

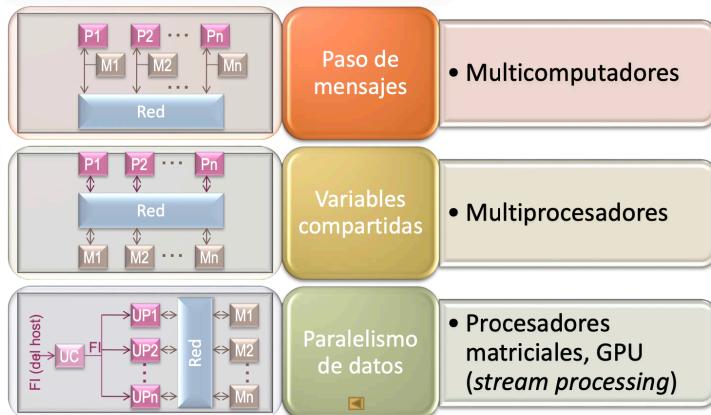
Las comunicaciones compuestas se definen por diferentes modos de reducción de los mensajes enviados por los procesos en los que todos combinan, en cada acción se realiza una combinación de los procesos (lo que provoca una reducción) o se realizan por un recorrido (scan) o prefijo paralelo; cada proceso manda a todos los que tienen un índice mayor o igual que el proceso que envía y el recorrido sufijo paralelo; todos reciben del proceso que tienen un índice mayor o igual del proceso que reciben en función de si los mensajes se recogen en orden de índice ascendente o descendente respectivamente.



En OpenMP podemos utilizar las siguientes directivas y cláusulas para crear sistemas de comunicación colectiva:

Uno-a-todos	Difusión (Seminario pract. 2)	✓ Cláusula <code>firstprivate</code> (desde thread 0)	Uno-a-uno	Asíncrona	<code>MPI_Send()</code> / <code>MPI_Receive()</code>
		✓ Directiva <code>single</code> con cláusula <code>copyprivate</code>		Difusión	<code>MPI_Bcast()</code>
		✓ Directiva <code>threadprivate</code> y uso de cláusula <code>copyin</code> en directiva <code>parallel</code> (desde thread 0)		Dispersión	<code>MPI_Scatter()</code>
Todos-a-uno	Reducción (Seminario pract. 2)	✓ Cláusula <code>reduction</code>	Todos-a-uno	Reducción	<code>MPI_Reduce()</code>
Servicios compuestos	Barreras (Seminario pract. 1)	✓ Directiva <code>barrier</code>		Acumulación	<code>MPI_Gather()</code>
			Todos-a-todos	Todos difunden	<code>MPI_Allgather()</code>
				Todos combinan	<code>MPI_Allreduce()</code>
			Servicios compuestos	Barreras	<code>MPI_Barrier()</code>
				Scan	<code>MPI_Scan</code>

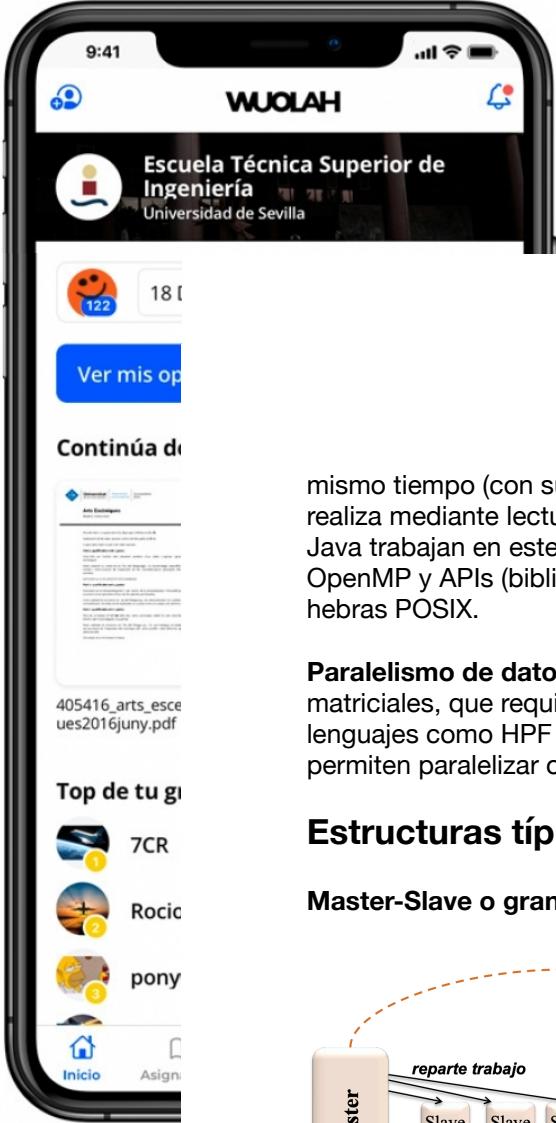
Estilos/paradigmas de programación paralela



Paso de mensajes (message passing): Es el paradigma utilizado en las arquitecturas multicomputador, que necesitan que el computador emisor envíe al receptor los datos con los que necesita trabajar. Estos sistemas se pueden programar con lenguajes como Ada u Occam y APIs (bibliotecas de funciones) como MPI o PVM.

Variables compartidas (shared memory, shared variables):

En los sistemas multiprocesador las variables compartidas pueden alojarse en la memoria compartida y ser accesible por todos los procesadores al



Descarga la APP de Wuolah.

Ya disponible para el móvil y la tablet.

Available on the
App Store

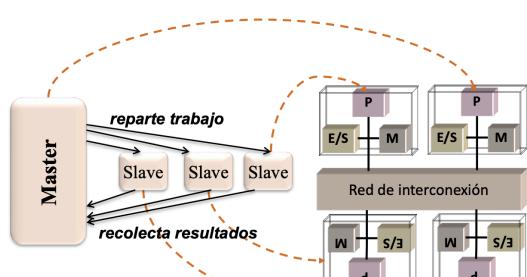
GET IT ON
Google Play

mismo tiempo (con sus consecuentes problemas de concurrencia). La comunicación se realiza mediante lecturas y escrituras en la memoria compartida. Lenguajes como Ada o Java trabajan en este paradigma, así como la API (directiva del compilador + funciones) OpenMP y APIs (bibliotecas de funciones) Intel TBB (Threading Building Blocks) y las hebras POSIX.

Paralelismo de datos (data parallelism): Es la forma de trabajar de los procesadores matriciales, que requieren una gran potencia de procesamiento. Se implementa con lenguajes como HPF (High Performance Fortran) o Fortran 95, en el que los bloques forall permiten paralelizar operaciones con matrices y vectores, y con APIs como Nvidia CUDA.

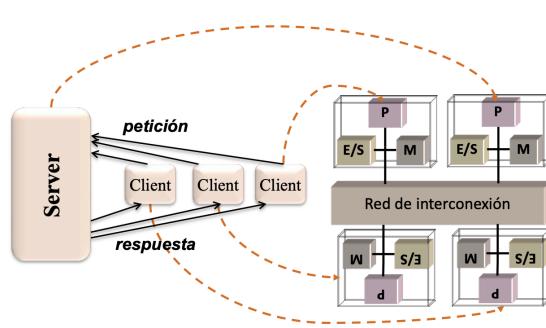
Estructuras típicas de códigos paralelos

Master-Slave o granja de tareas



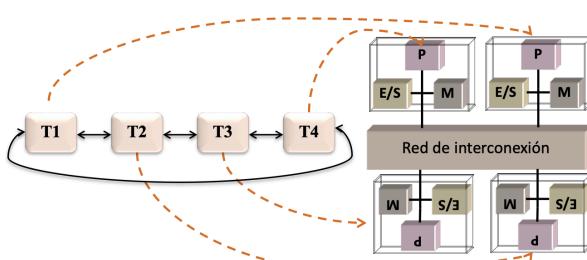
El proceso maestro reparte el trabajo a varios procesos esclavos, que realizan su cómputo individualmente y envían el resultado al proceso maestro para que este haga un cómputo final con todos los resultados recolectados. Tenemos una comunicación de uno a todos.

Cliente-Servidor



Varios procesos cliente mandan peticiones a un proceso servidor, que las gestiona y envía respuestas a los procesos cliente con el resultado de los cómputos.

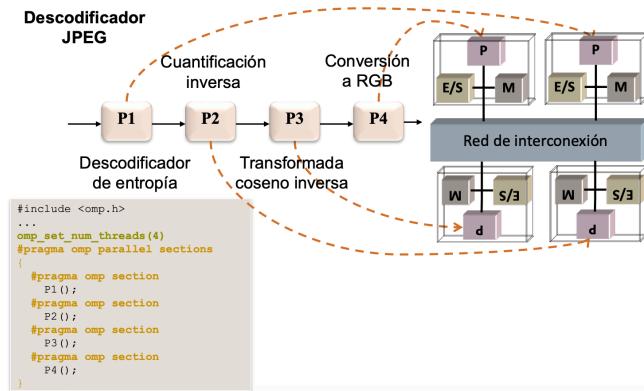
Descomposición de dominio o descomposición de datos I



Cada proceso adquiere una parte de la computación a realizar y entre todos resuelven el problema dividiéndolo en partes equitativas. Un ejemplo de esto sería el uso de la directiva *omp for*. La comunicación se realizará a través de lecturas o

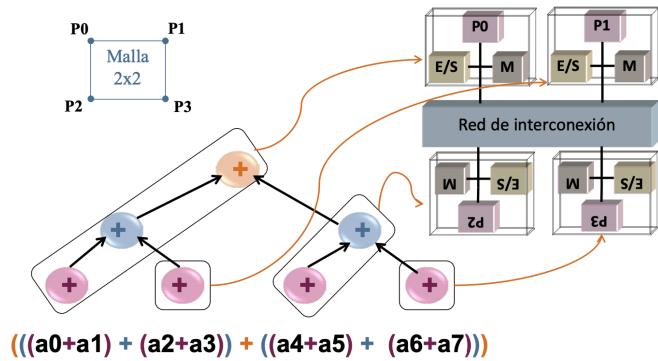
escrituras o por paso de mensajes.

Estructura segmentada o de flujo de datos



Los procesos se organizan de la misma forma que un procesador segmentado de forma que, para cada segmento i , éste pueda ejecutarse mientras el resto de segmentos están procesando otra información que llegará al procesador i o de éste. Estos procesos son diferentes y cada uno ejecuta sus tareas. Podemos crear una estructura donde los procesos van pasando de un segmento a otro.

Divide y vencerás o descomposición recursiva



El problema se divide en subproblemas recursivos más pequeños y cada uno de los procesos ejecuta los cálculos de cada uno de los subproblemas. Cuando dos subproblemas llegan a su caso ancestro común, uno de los procesos se elimina y se continúan los cálculos con el proceso restante.

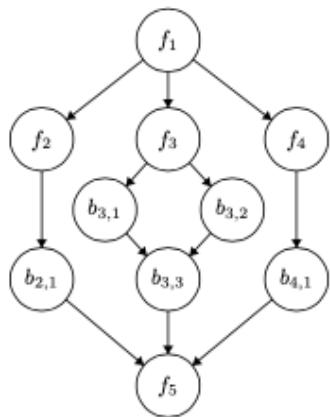
Proceso de parallelización

A la hora de parallelizar un programa seguimos cuatro pasos:

- **Descomponer (descomposición)** el programa en tareas independientes. Buscamos que tareas independientes son las que se pueden realizar en el código que vamos a implementar. Es decir, buscamos el paralelismo implícito.
- **Asignar (planificar+mapear)** tareas a procesos y hebras. Una vez encontradas las tareas, las agrupamos generando distintos procesos y hebras y lo mapeamos en los correspondientes procesadores.
- **Redactar** el código paralelo. Después de esto, definimos la estructura del código.
- **Evaluar** las prestaciones que ofrece la parallelización. Y por último evaluamos las prestaciones y vemos si hemos alcanzado los objetivos que teníamos

Estos cuatro pasos deben seguirse secuencialmente. De la evaluación podemos volver a cualquiera de los tres anteriores en función de los errores que hayamos detectado o dar por finalizada la paralelización del programa.

1. Descomponer en tareas independientes: Para descomponer un programa secuencial en tareas independientes debemos llevar a cabo un análisis de dependencias de datos entre las diferentes funciones en las que se divide y, dentro de éstas, entre los diferentes bloques que las componen. Al hacer esto podemos crear un grafo de dependencias que ilustre que funciones y bloques pueden ejecutarse paralelamente en cada momento. Tenemos que ver qué dependencia hay entre las distintas funciones.



```
int main (int argc, char ** argv) {
    double ancho,
    sum = 0,
    x;
    int intervalos;

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

Por ejemplo, tengamos el siguiente código para aproximar el número pi. En él, identificamos tres bloques principales:

- Declaración e inicialización de variables.
- Cálculo de la aproximación en bucle.
- Ajuste final de la aproximación.

Mientras que el primer y último bloque son indivisibles, el segundo podemos dividirlo en n tareas que se ejecuten paralelamente para n intervalos usados en la aproximación, de forma que cada uno de los valores de i en el bucle se ejecute de forma paralela a los otros.

2. Asignar (planificar + mapear) tareas a procesos y/o hebras: nos surge la asignación de tareas a procesos o hebras de manera que la granularidad de las hebras sea adecuado, la que nos permite el mejor rendimiento. Hay que tener en cuenta tanto el numero de procesadores como el coste de la comunicación en relación con la computación. Incluimos: agrupación de tareas en procesos/threads (scheduling) y mapeo a procesadores/cores (mapping). La granularidad de la carga asignada a los procesos/threads depende de: número de cores o procesadores o elementos de procesamiento y el tiempo de comunicación/sincronización frente a tiempo de cálculo.

También hay que tener en cuenta el equilibrado de la carga (tareas = código + datos) que todo el trabajo que se realicen en paralelo sean similar, de manera que vayan disminuyendo hasta que el tiempo de espera en ellas sea similar, el objetivo es entonces que unos procesos/hebras no deben esperar a otros.

En esta fase distinguimos dos tipos de asignación de las tareas:

- Planificación: Agrupación de las tareas en hebras.

```

void F1 () {
    #pragma omp parallel for schedule(static)
    for (int i=0; i<N; i++)
        // Código para i
}
void F2 () { /* ... */ }
void F3 () { /* ... */ }

int main () {
    #pragma omp parallel section
    {
        #pragma omp section
        F1();
        #pragma omp section
        F2();
        #pragma omp section
        F3();
    }
}

```

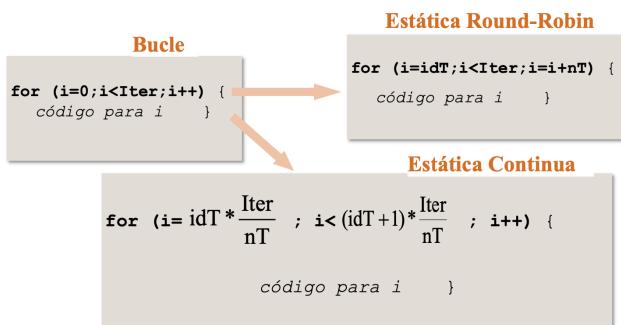
- Mapping: Asignación de las hebras a núcleos o procesadores

En este ejemplo de asignación estática con OpenMP², cada función F[1-3] se ejecuta paralelamente con las otras y, dentro de ellas, los bucles for se ejecutan paralelamente en sus N iteraciones.

El equilibrado de la máquina depende de la arquitectura; tenemos una arquitectura homogénea frente a heterogénea (que nodos son mas rápidos, más lentos...) También hay que ver si la arquitectura es uniforme frente a no uniforme (ver si el numero de nodos es siempre el mismo o no). También depende de la aplicación/descomposición a ejecutar.

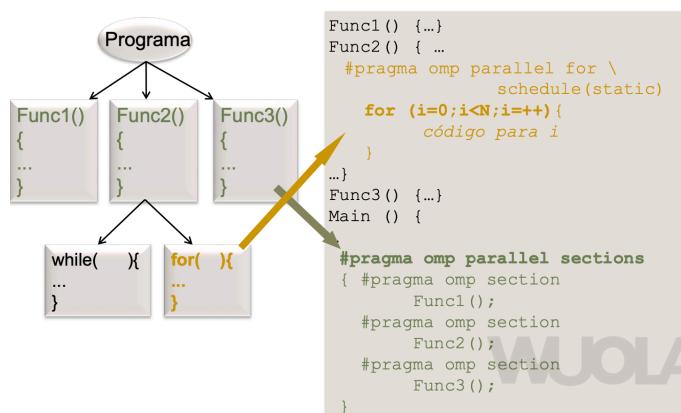
La asignación de las tareas a cada una de las hebras se puede hacer de forma **estática** (determina que volumen de trabajo va a realizar cada procesador o core a partir del código), asignando las hebras en tiempo de compilación, o **dinámica**, en cuyo caso la asignación se hace en tiempo de ejecución. En esta última diferentes ejecuciones del programa pueden dar lugar a asignaciones de las tareas sobre diferentes hebras o procesadores. Ambas asignaciones se pueden hacer de forma explícitas por el programador o de forma implícita por la herramienta de programación utilizada

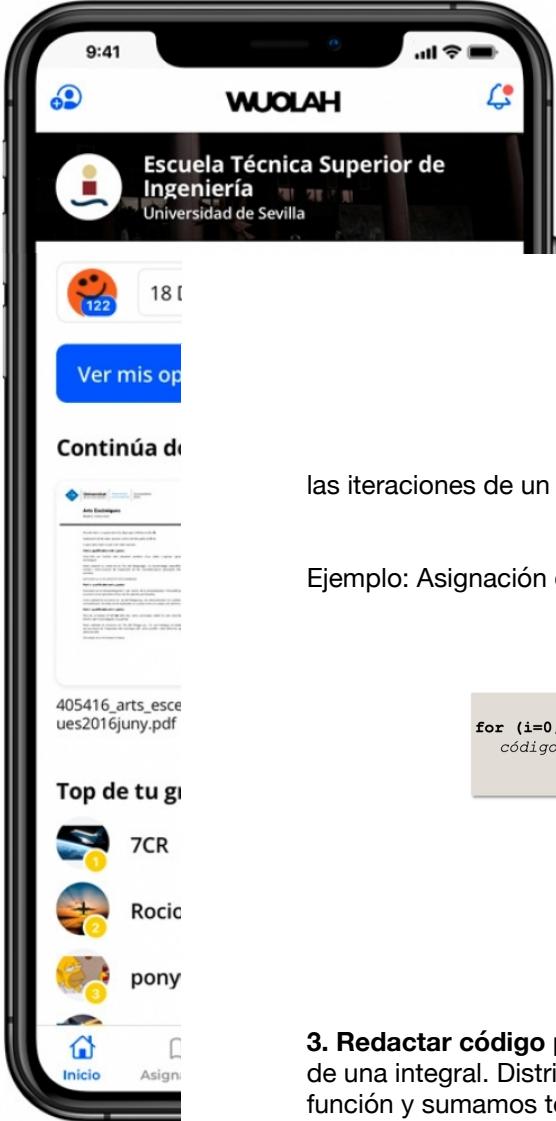
El mapping de las hebras se suele dejar al SO, que lo implementa mediante un sistema llamado light process. También puede hacerse por el entorno o el sistema en tiempo de ejecución (runtime system) de la herramienta de programación o explicitarse por el programador.



El mapping de las hebras se suele dejar al SO, que lo implementa mediante un sistema llamado light process. También puede hacerse por el entorno o el sistema en tiempo de ejecución (runtime system) de la herramienta de programación o explicitarse por el programador.

Ejemplo: Asignación estática y explícita de





Descarga la APP de Wuolah.

Ya disponible para el móvil y la tablet.

Available on the
App Store

GET IT ON
Google Play

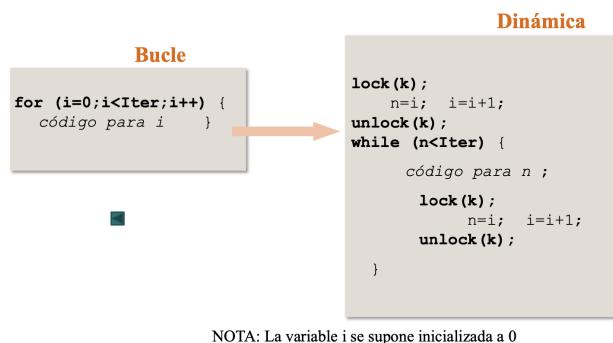


Ver mis op

Continúa d

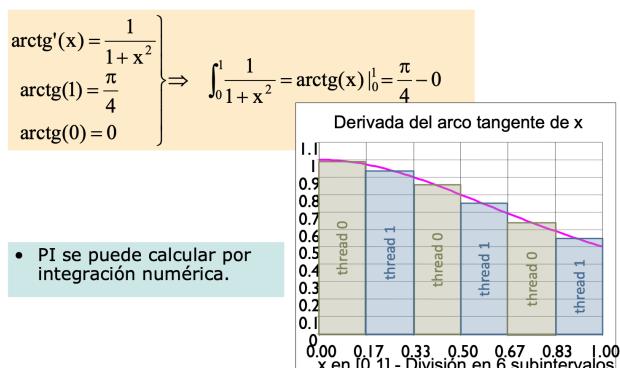
las iteraciones de un bucle:

Ejemplo: Asignación dinámica y explícita de las iteraciones de un bucle:



Reservados todos los derechos. Queda permitida la impresión en su totalidad. No se permite la explotación económica ni la transformación de esta obra.

3. Redactar código paralelo: retomamos el ejemplo del calculo del numero pi a través de una integral. Distribuimos el trabajo entre las hebras y a partir de esto calculamos la función y sumamos todo. Se esta haciendo una asignación estática. En este caso sería Round-Robin. Hemos asignado mas carga a una hebra que a otro. El código que obtendríamos para OpenMP/C sería el siguiente:



```
#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); →Crear/Terminar
#pragma omp parallel →Comunicar/sincronizar
{
    #pragma omp for reduction(+:sum) private(x) \
    schedule(dynamic) →Agrupar/Asignar
    for (i=0;i< intervalos; i++) {
        x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
    }
    sum* = ancho;
}
```

Y el código para MPI/C sería:

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

4. Evaluar prestaciones: ver el programa que hemos hecho, ejecutarlo, valorarlo y ver cómo reacciona, ver si reúne las prestaciones que necesitamos y si no las reúne podemos volver a cualquier fase anterior para solucionarlo.

Evaluación de prestaciones en procesamiento paralelo

Medidas usuales: podemos evaluar las prestaciones de un sistema de procesamiento paralelo en función de su tiempo de respuesta [real (wall-clock time, elapsed time) (/usr/bin/time) - usuario, sistema, CPU time= user + sys] y productividad

Escalabilidad: aumento de las prestaciones que se tiene cuando se varía el número de procesadores que tenemos que utilizar o estamos utilizando.

Eficiencia: relación entre las prestaciones y las prestaciones máximas que se pueden conseguir. En muchas ocasiones se expresan utilizando el número de procesadores que se puede conseguir.

$$\text{Eficiencia} = \frac{\text{Prestaciones}}{\text{Prestaciones máximas}} \quad \text{Rendimiento} = \frac{\text{Prestaciones}}{\text{Nº recursos}} \quad \frac{\text{Prestaciones}}{\text{Consumo potencia}} \quad \frac{\text{Prestaciones}}{\text{Área ocupada}}$$

Ganancia en prestaciones S(p) (speed-up): para p procesadores como la razón entre el tiempo de ejecución secuencial T_s del programa y el tiempo de ejecución paralela $T_p(p)$ para p procesadores. Este tiempo de ejecución paralela podemos calcularlo como la suma del tiempo de cómputo paralelo $T_c(p)$ y el tiempo de sobrecarga (overhead) $T_o(p)$ introducido por el coste de parallelizar el programa para p procesadores. Esta sobrecarga se debe al tiempo sumado por la sincronización y comunicación de las hebras, así como su creación y finalización, los cómputos añadidos necesariamente en la versión paralela y el déficit de equilibrado de la versión paralela.

$$S(p) = \frac{T_s}{T_p(p)} \quad T_p(p) = T_c(p) + T_o(p)$$

Distinguimos cuatro tipos de escalabilidad de un programa paralelo:

Ganancia lineal

Se da cuando todos los bloques son paralelizables ($T_s = 0$) y la sobrecarga es nula, de forma que $S(p) = p$. Si tenemos un bloque secuencial que ocupa una fracción $s \leq 1$ no paralelizable en el código, debemos tener en cuenta que la escalabilidad se ve reducida por la presencia del mismo:

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

Existe el caso de que la ganancia sea superlineal, de forma que $S(p) > p$.
Limitada en el aprovechamiento del grado de paralelismo

El grado de paralelismo es ilimitado y no existe overhead. El sistema esta limitado por f ($ts=f$).

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

Similar a limitada en el aprovechamiento del grado de paralelismo

Se produce cuando el sistema únicamente puede aprovechar hasta un número n de prestaciones, de forma que la ganancia queda máxima. No hay overhead.

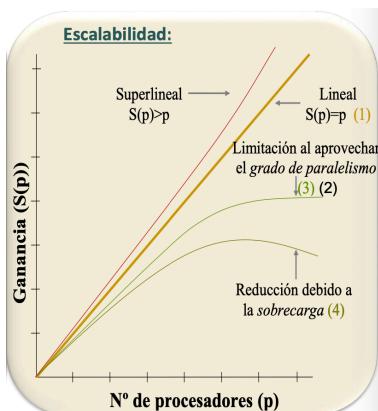
$$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p=n} \frac{1}{f + \frac{(1-f)}{n}}$$

Reducida debido a sobrecarga

Se produce cuando la sobrecarga incrementa linealmente con p , de forma que en algún punto comienza a afectar negativamente al tiempo de ejecución paralelo. El overhead incrementa linealmente con p . El grado de paralelismo es ilimitado.

$$S(p) = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_o(p)}{T_s}} \xrightarrow{p \rightarrow \infty} 0$$

Podemos calcular el número máximo de prestaciones igualando tiempo de cálculo $T_c(p) = O(p)$ al tiempo de sobrecarga $T_o(p) = O(p)$ añadido para una función O de variación del tiempo de ejecución.



Modelo código	Fracción no paral. en T_s	Grado paralelismo	Overhead	Ganancia en función del número de procesadores p con T_s constante	
				(a)	(b)
(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)

Ley de Amdahl

La ganancia en prestaciones utilizando p procesadores está limitada por la fracción de código que no se puede parallelizar. Como consecuencia lógica de esto, cuanto más pequeña es esta fracción, mayor es la ganancia en prestaciones al parallelizar el programa. Sin embargo, esta ley no tiene en cuenta la sobrecarga introducida en la parallelización del programa. Dado que esta sobrecarga es inversamente proporcional al tamaño del problema ejecutado por un determinado número de núcleos, podemos incrementar la ganancia aumentando el número de cómputos.

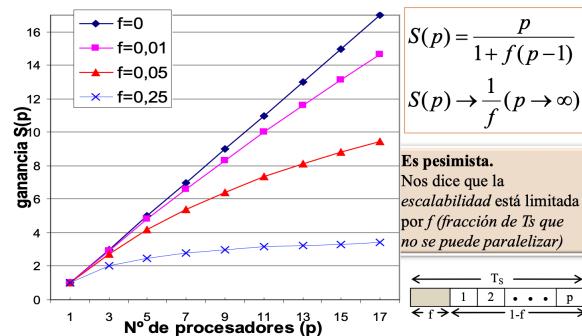
$$S(p) = \frac{T_s}{T_p(p)} \leq \frac{T_s}{f \cdot T_s + (1-f) \cdot T_s} = \frac{p}{1 + f(p-1)} \rightarrow \frac{1}{f} (p \rightarrow \infty)$$

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)

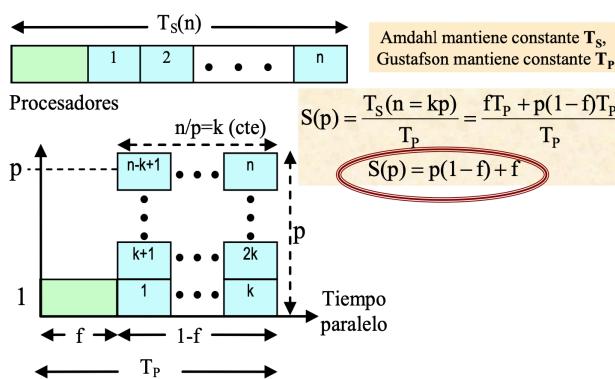
f : fracción de tiempo en el que no se puede aplicar la mejora. (fracción de t. no parallelizable)

En la ley de Amdahl no hay que tener en cuenta el overhead ni la limitación en el parallelismo que pueda tener la parte parallelizable. Pero si consideramos que que se pueden distribuir las tareas. El parallelismo no va a ser eficiente.



Ganancia escalable o Ley de Gustafson

De forma contraria a la ley de Amdahl, la ley de Gustafson plantea que cualquier programa suficientemente grande puede ser eficientemente parallelizado. Mientras que Amdahl no escala la disponibilidad del poder de cómputo junto con el aumento del número de máquinas, Gustafson propone que, con la mayor potencia de cómputo disponible, se podrán resolver problemas mayores en el mismo tiempo debido a la reducción de la parte secuencial no parallelizable. Para un programa con p prestaciones y una porción secuencial s , Gustafson define la escalabilidad de la siguiente manera:



La ganancia se puede incrementar aumentando el tamaño del problema.

Amdahl mantiene constante T_s y Gustafson mantiene constante T_p .