AC - Relacion 2 Algunos Ejercicios Resueltos.pdf

Ejercicios Resueltos

2º curso / 2º cuatr.

Grado en Ing. Informática

Arquitectura de Computadores. Algunos ejercicios resueltos

Tema 2. Programación paralela

Profesores responsables: Mancia Anguita, Julio Ortega

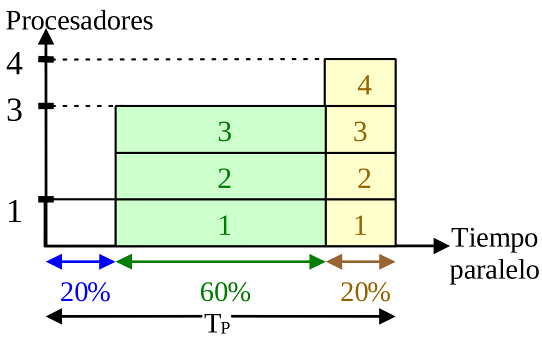
1 Ejercicios

Ejercicio 1. Un programa tarda 40 s en ejecutarse en un multiprocesador. Durante un 20% de ese tiempo se ha ejecutado en cuatro procesadores (core); durante un 60%, en tres; y durante el 20% restante, en un procesador (consideramos que se ha distribuido la carga de trabajo por igual entre los procesadores que colaboran en la ejecución en cada momento, despreciamos sobrecarga). ¿Cuánto tiempo tardaría en ejecutarse el programa en un único procesador?¿Cuál es la ganancia en velocidad obtenida con respecto al tiempo de ejecución secuencial?¿Y la eficiencia?

Solución

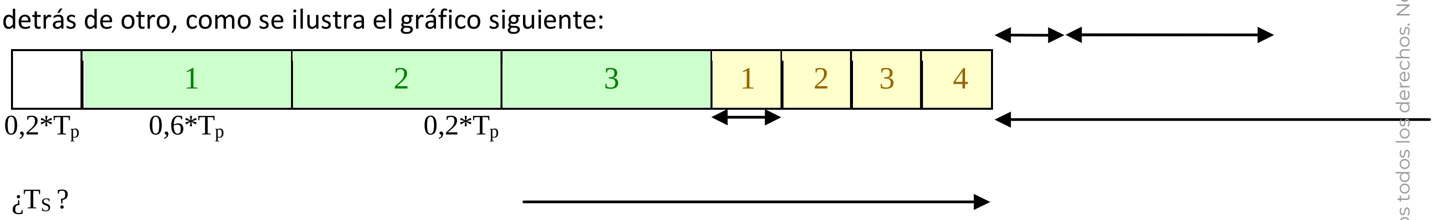
Datos del ejercicio:

En el gráfico de la derecha se representan los datos del ejercicio. Estos datos son la fracción del tiempo de ejecución paralelo (T_p) que supone el código no paralelizable (20% de T_p, es decir, $0,2*T_p$), la fracción que supone el código paralelizable en 3 procesadores ($0,6*T_p$) y la que supone el código paralelizable en 4 procesadores ($0,2*T_p$). Al distribuirse la carga de trabajo por igual entre los procesadores utilizados en cada instante, los trozos asignados a 3 procesadores suponen todos el mismo tiempo (por eso se han dibujado en el gráfico de igual tamaño) e, igualmente, los trozos asignados a 4 procesadores también suponen todos el mismo tiempo.



¿Tiempo de ejecución secuencial, T_S?:

Los trozos de código representados en la gráfica anterior se deben ejecutar secuencialmente, es decir, uno detrás de otro, como se ilustra el gráfico siguiente:



$T_S = 0,2 \times T_P + 3 \times 0,6 \times T_P + 4 \times 0,2 \times T_P = (0,2 + 1,8 + 0,8) \times T_P = 2,8 \times T_P = 2,8 \times 40s = 112\ s$

<u>¿Ganancia en velocidad, S(p)?:</u>

$$S(4) = \frac{T_S}{T_P(4)} = \frac{2,8 \times T_P(4)}{T_P(4)} = 2,8$$

<u>¿Eficiencia, E(p)?:</u>

$$E(4) = \frac{S(p)}{p} = \frac{S(4)}{4} = \frac{2,8}{4} = 0,7$$

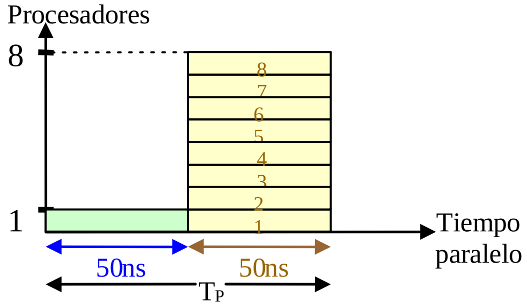
◄◄◄◄◄◄

Ejercicio 2. ◄

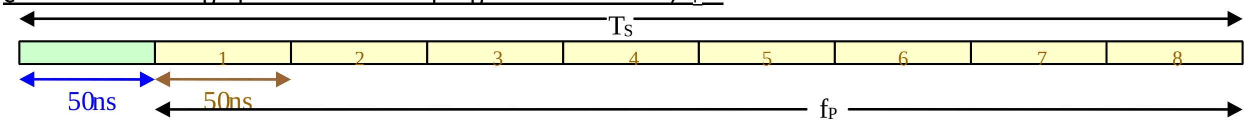
Ejercicio 3. ¿Cuál es fracción de código paralelo de un programa secuencial que, ejecutado en paralelo en 8 procesadores, tarda un tiempo de 100 ns, durante 50ns utiliza un único procesador y durante otros 50 ns utiliza 8 procesadores (distribuyendo la carga de trabajo por igual entre los procesadores)?

Solución

<u>Datos del programa:</u>



<u>¿Fracción de código paralelizable del programa secuencial, f_P?:</u>



$$f_p = \frac{8 \times 50\,ns}{T_S} = \frac{8 \times 50\,ns}{50\,ns + 8 \times 50\,ns} = \frac{8}{9}$$

◄◄◄◄◄◄

Ejercicio 4. ◄

Ejercicio 5. ◄

Ejercicio 6. ◄

Ejercicio 7. Se quiere paralelizar el siguiente trozo de código:

```
{Cálculos antes del bucle} for(
i=0; i<w; i++) {
                Código para i
```

```
}
{Cálculos después del bucle}
```

Los cálculos antes y después del bucle suponen un tiempo de t_1 y t_2, respectivamente. Una iteración del ciclo supone un tiempo t_i. En la ejecución paralela, la inicialización de p procesos supone un tiempo k_1p (k_1 constante), los procesos se comunican y se sincronizan, lo que supone un tiempo k_2p (k_2 constante); $k_1p+ k_2p$ constituyen la sobrecarga.

(a) Obtener una expresión para el tiempo de ejecución paralela del trozo de código en p procesadores (T_p).

(b) Obtener una expresión para la ganancia en velocidad de la ejecución paralela con respecto a una ejecución secuencial (S_p).

(c) ¿Tiene el tiempo T_p con respecto a **p** una característica lineal o puede presentar algún mínimo? ¿Por qué? En caso de presentar un mínimo, ¿para qué número de procesadores p se alcanza? **Solución**

Datos del ejercicio:

```
{Cálculos antes del bucle}
for( i=0; i<w; i++) {
       Código para i

} {cálculos después del
bucle}
```

t_1

t_i

t_2

Sobrecarga: $T_O = k_1p + k_2p = (k_1 + k_2)p$

Llamaremos t a $t_1 + t_2$ (= t) y k a $k_1 + k_2$ (=k) (a)

¿Tiempo de ejecución paralela , T_p?

$$T_P(p,w) = t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p$$

w/p se redondea al entero superior (b)

¿Ganancia en prestaciones , $S(p,w)$?

$$S(p,w) = \frac{T_S}{T_P(p,w)} = \frac{t + w \times t_i}{t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p}$$

(c) ¿Presenta mínimo? ¿Para qué número p?

$$T_P(p,w) = t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p \quad (1)$$

En la expresión (1), el término englobado en línea continua, o tiempo de cálculo paralelo, tiene tendencia a decrecer con pendiente que va disminuyendo conforme se incrementa p (debido a que p está en el p divisor), y el término englobado con línea discontinua o tiempo de sobrecarga (kxp), crece conforme se incrementa p con pendiente k constante (ver ejemplos en Figura 3, Figura 4, Figura 5 y Figura 6). Las oscilaciones en el tiempo de cálculo paralelo se deben al redondeo al entero superior del cociente w/p, pero la tendencia, como se puede ver en las figuras, es que T_p va decreciendo. Dado que la pendiente de la sobrecarga es constante y que la pendiente del tiempo de cálculo decrece conforme se incrementa p, llega un momento en que el tiempo de ejecución en paralelo pasa de decrecer a crecer.

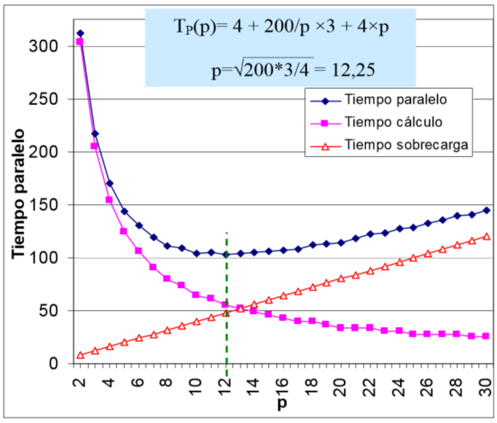


Figura 3. Tiempo de ejecución paralelo para un caso particular en el que *w*=200, t_i es 3 unidades de tiempo, *t* son 4 unidades de tiempo y *k* son 4 unidades de tiempo. El mínimo se alcanza para *p*=12

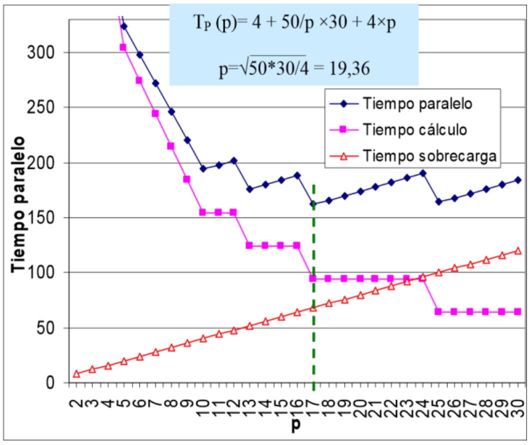


Figura 4. Tiempo de ejecución paralelo para un caso particular en el que *w*=50, t_i es 30 unidades de tiempo, *t* son 4 unidades de tiempo y *k* son 4 unidades de tiempo. El mínimo se alcanza para *p*=17. Aquí el efecto del redondeo se ve más claramente que en la Figura 3

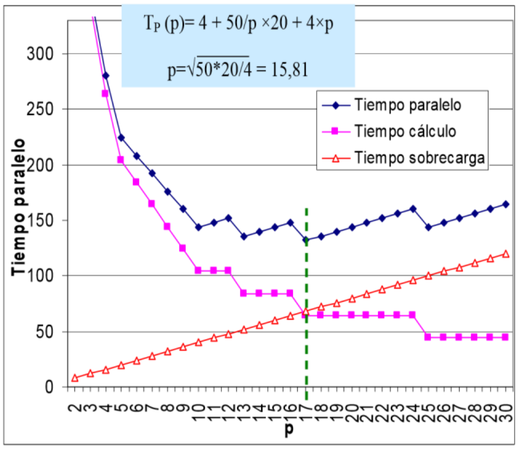


Figura 5. Tiempo de ejecución paralelo para un caso particular particular en el que *w*=50, t_i es 20 unidades de 4 tiempo, *t* son 4 unidades de tiempo y *k* son 4 unidades de tiempo. El mínimo se alcanza para *p*=17.

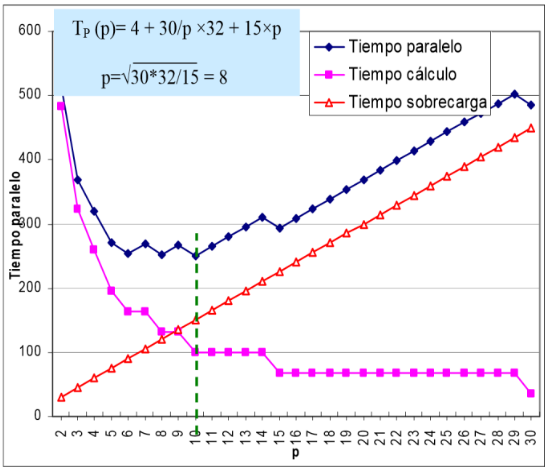


Figura 6. Tiempo de ejecución paralelo para un caso en el que *w*=30, t_i es 32 unidades de tiempo, *t* son unidades de tiempo y *k* son 15 unidades de tiempo. El mínimo se alcanza para *p*=10.

Se puede encontrar el mínimo analíticamente igualando a 0 la primera derivada de T_p. De esta forma se obtiene los máximos y mínimos de una función continua. Para comprobar si en un punto encontrado de esta forma hay un máximo o un mínimo se calcula el valor de la segunda derivada en ese punto. Si, como resultado de este cálculo, se obtiene un valor por encima de 0 hay un mínimo, y si, por el contrario, se obtiene un valor por debajo de 0 hay un máximo. Para realizar el cálculo se debe eliminar el redondeo de la expresión (1) (más abajo se comentará la influencia del redondeo en el cálculo del mínimo):

$$T_P(p,w) = t + \frac{w}{p} \times t_i + k \times p$$

$$T'_P(p,w) = 0 - \frac{w}{p^2} \times t_i + k \Rightarrow \frac{w}{p^2} \times t_i = k \Rightarrow p = \sqrt{\frac{w \times t_i}{k}} \quad (2)$$

El resultado negativo de la raíz se descarta. En cuanto al resultado positivo, debido al redondeo, habrá que comprobar para cuál de los naturales próximos al resultado obtenido (incluido el propio resultado si es un número natural) se obtiene un menor tiempo. Debido al redondeo hacia arriba de la expresión (1), se deberían comprobar necesariamente:

1. El natural p' menor o igual y más alejado al resultado p generado con (2) para el que $\left\lceil \frac{w}{p'} \right\rceil = \left\lceil \frac{w}{p} \right\rceil$ (obsérvese, que en el ejemplo de la Figura 4, aunque de (2) se obtiene 19,36, el p con menor tiempo es 17 debido al efecto del redondeo) y
2. El natural p' mayor y más próximo al p generado con (2) para el que $\left\lceil \frac{w}{p'} \right\rceil = \left\lceil \frac{w}{p} \right\rceil - 1$ (obsérvese, que en el ejemplo de la Figura 5, aunque de (2) se obtiene 15,81, el p con menor tiempo es 17 debido al redondeo).

En cualquier caso, el número de procesadores que obtengamos debe ser menor que w (que es el grado de paralelismo del código).

La segunda derivada permitirá demostrar que se trata de un mínimo:

$$T_P(p,w) = t + \frac{w}{p} \times t_i + k \times p$$

$$T''_P(p,w) = + \frac{2 \times p \times w \times t_i}{p^4} + 0 = \frac{2 \times w \times t_i}{p^3} > 0$$

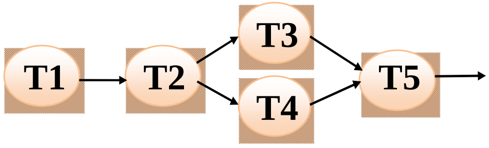
La segunda derivada es mayor que 0, ya que w, p y t_i son mayores que 0; por tanto, hay un mínimo.

◄◄◄◄◄◄

Ejercicio 8. ◄

Ejercicio 9. Se va a paralelizar un decodificador JPEG en un multiprocesador. Se ha extraído para la aplicación el siguiente grafo de tareas que presenta una estructura segmentada (o de flujo de datos):

Entrada: igual a t, mientras que (supone 8x8 pixels de pixels la imagen)



La tareas 1, 2 y 5 se ejecutan Bloque de la **Salida:** Bloque en un tiempo imagen a decodificar decodificado de 8x8 las tareas 3 y 4 suponen 1,5t

El decodificador JPEG aplica el grafo de tareas de la figura a bloques de la imagen, cada uno de 8x8 píxeles. Si se procesa una imagen que se puede dividir en **n** bloques de 8x8 píxeles, a cada uno de esos **n** bloques se aplica el grafo de tareas de la figura. Obtenga la mayor ganancia en prestaciones que se puede conseguir paralelizando el decodificador JPEG en (suponga despreciable el tiempo de comunicación/sincronización): **(a)** 5 procesadores, y **(b)** 4 procesadores. En cualquier de los dos casos, la ganancia se tiene que calcular suponiendo que se procesa una imagen con un total de **n** bloques de 8x8 píxeles. **Solución**

Para obtener la ganancia se tiene que calcular el tiempo de ejecución en secuencial para un tamaño del problema de n bloques $T_s(n)$ y el tiempo de ejecución en paralelo para un tamaño del problema de n y los p procesadores indicados en (a), (b) y (c) $T_p(p,n)$.

Tiempo de ejecución secuencial $T_s(n)$. Un procesador tiene que ejecutar las 5 tareas para cada bloque de 8x8 píxeles. El tiempo que dedica el procesador a cada bloque es de 3t (tareas 1, 2 y 5) + 3 t (tareas 2 y 4) = 6t, luego para n bloques el tiempo de ejecución secuencial será el siguiente:

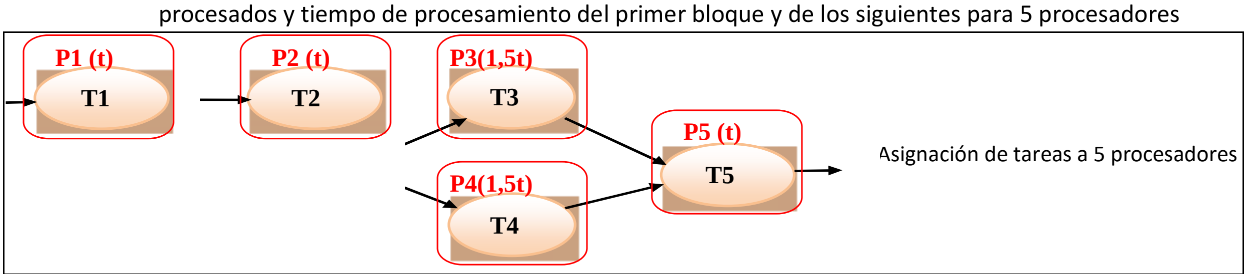
$$T_S = n \times (\ )6t$$

(a) Tiempo de ejecución paralelo y ganancia en prestaciones para 5 procesadores $T_p(5,n)$, $S(5,n)$. Cada tarea se asigna a un procesador distinto, por tanto todas se pueden ejecutar en paralelo (ver asignación de tareas a procesadores en la Tabla 1). El tiempo de ejecución en paralelo en un pipeline consta de dos tiempos: el tiempo que tarda en procesarse la primera entrada (ver celdas con fondo verde en la tabla) + el tiempo que tardan cada uno del resto de bloques (entradas al cauce) en terminar. Este último depende de la etapa más lenta, que en este caso es la etapa 3 (1,5t frente a t en el resto de etapas).

$$T_P(5,n) = 4{,}5t + (n-1)\times 1{,}5t = 3t + n\times 1{,}5t$$

$$S(5,n) = \frac{T_s(n)}{T_p(5,n)} = \frac{n\times 6t}{3t + n\times 1{,}5t} \xrightarrow{n\to\infty} 4$$ La ganancia se aproxima a 4 para n suficientemente grande.

Tabla 1. Asignación de tareas a procesadores, ocupación de las etapas del cauce para los primeros bloques procesados y tiempo de procesamiento del primer bloque y de los siguientes para 5 procesadores



Etapa 1 (t)	Etapa 2 (t)	Etapa 3 (1,5t)	Etapa 4 (t)	Terminado	Tiempo procesa.

T1 (procesador P1)	T2 (procesador P2)	T3 (procesador P3) y T4 (procesador P4)	T5 (procesador P5)		
0 - Bloque 1 - t					Procesamiento Bloque 1 (4,5t)
t - Bloque 2 - 2t	t - Bloque 1 - 2t				
2t - Bloque 3 - 3t	2t- Bloque 2 - 3t	2t - Bloque 1 - 3,5t			
3t - Bloque 4 - 4t	3t - Bloque 3 - 4t	3,5t - Bloque 2 - 5t	3,5t - Bloque 1 - 4,5t		
4t - Bloque 5 - 5t	4t - Bloque 4 - 5t	5t - Bloque 3 - 6,5t	5t - Bloque 2 - 6t	Bloque 1	t + t + 1,5t + t = 4,5t
5t - Bloque 6 - 6t	5t - Bloque 5 - 6t	6,5t - Bloque 4 - 8t	6,5t - Bloque 3 - 7,5t	Bloque 2	4,5t + 1,5t = 6t
6t - Bloque 7 - 7t	6t - Bloque 6 - 7t	8t - Bloque 5 - 9,5t	8t - Bloque 4 - 9t	Bloque 3	6t + 1,5t = 7,5t
…	…	…	…	Bloque 4	7,5t + 1,5t = 9t
T_entrada_etapa – Bloque x – T_salida_etapa					

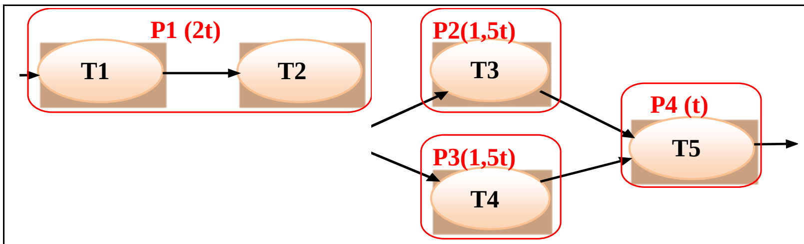
(b) Tiempo de ejecución paralelo y ganancia en prestaciones para 4 procesadores $T_p(4,n)$, $S(4,n)$. Se deben asignar las 5 tareas a los 4 procesadores de forma que se consiga el mejor tiempo de ejecución paralelo, es decir, el menor tiempo por bloque una vez procesado el primer bloque. Una opción que nos lleva al menor tiempo por bloque consiste en unir la Etapa 1 y 2 del pipeline del caso (a) en una única etapa asignando T1 y T2 a un procesador (ver asignación de tareas a procesadores en la Tabla 2). Con esta asignación la etapa más lenta supone 2t (Etapa 1); habría además una etapa de 1,5 t (Etapa 2) y otra de t (Etapa 3). Si, por ejemplo, se hubieran agrupado T3 y T4 un procesador la etapa más lenta supondría 3t.

$$T_P(4,n) = 4{,}5t + (n-1)\times 2t = 2{,}5t + n\times 2t$$

$$S(4,n) = \frac{T_s(4,n)}{T_p(4,n)} = \frac{6\times n\times t}{2{,}5t + n\times 2t} \xrightarrow[n\to\infty]{} 3$$

La ganancia se aproxima a 3 para n suficientemente grande.

Tabla 2. Asignación de tareas a procesadores, ocupación de las etapas del cauce para los primeros bloques procesados y tiempo de procesamiento del primer bloque y de los siguientes para 4 procesadores



Posible asignación de tareas a 4 procesadores

Etapa 1 (2t)	Etapa 2 (1,5t)	Etapa 3 (t)	Terminado	Tiempo procesamiento
T1 y T2 (procesador P1)	T3 (procesador P2) y T4 (procesador P3)	T5 (procesador P4)		
0 - Bloque 1 - 2t				Procesamiento Bloque 1 (4,5t)
2t - Bloque 2 - 4t	2t - Bloque 1 - 3,5t			
4t - Bloque 3 - 6t	4t- Bloque 2 - 5,5t	3,5t - Bloque 1 - 4,5t		
6t - Bloque 4 - 8t	6t - Bloque 3 - 7,5t	5,5t - Bloque 2 - 6,5t	Bloque 1	2t + 1,5t + t = 4,5t
8t - Bloque 5 - 10t	8t - Bloque 4 - 9,5t	7,5t - Bloque 3 - 8,5t	Bloque 2	4,5t + 2t = 6,5t
10t - Bloque 6 - 12t	10t - Bloque 5 - 11,5t	9,5t - Bloque 4 - 10,5t	Bloque 3	6,5t + 2t = 8,5t
12t - Bloque 7 - 14t	12t - Bloque 6 - 13,5t	11,5t - Bloque 5 - 12,5t	Bloque 4	8,5t + 2t = 10,5t
…	…	…	Bloque 5	10,5t + 2t = 12,5t
T_entrada_etapa – Bloque x – T_salida_etapa				

Ejercicio 10. Se quiere implementar un programa paralelo para un multicomputador que calcule la siguiente expresión para cualquier x (es el polinomio de interpolación de Lagrange): $P(x) = \sum_{i=0}^{n} \left(b_i \cdot L_i(x)\right)$, donde

$$L_i(x) = \frac{(x-a_0)\cdot\ldots(x-a_{i-1})\cdot(x-a_{i+1})\cdot\ldots(x-a_n)}{k_i} = \frac{\prod_{j=0,\, j\neq i}^{n}(x-a_j)}{k_i} \quad i = 0,1,\ldots,n$$

$$k_i = (a_i-a_0)\cdot\ldots(a_i-a_{i-1})\cdot(a_i-a_{i+1})\cdot\ldots(a_i-a_n) = \prod_{j=0,\, j\neq i}^{n}(a_i-a_j) \quad i = 0,1,\ldots,n$$

Inicialmente k_i, a_i y b_i se encuentra en el nodo i y x en todos los nodos. Sólo se van a usar funciones de comunicación colectivas. Indique cuál es el número mínimo de funciones colectivas que se pueden usar, cuáles serían y en qué orden se utilizarían y para qué se usan en cada caso.

Solución

Los pasos ((1) a (5)) del algoritmo para n=3 y un número de procesadores de n+1 serían los siguientes:

Pr.	Situación Inicial				(1) Resta paralela $A_i=(x-a_i)$ *($(x-a_i)$ se obtiene en Pi)*	(2) Todos reducen A_i con resultado en B_i: $B_i = \prod_{i=0}^{n}(A_i)$
P0	**a_0**	**x**	**k_0**	**b_0**	$(x-a_0)$	$(x-a_0).(x-a_1).(x-a_2).(x-a_3)$
P1	**a_1**	**x**	**k_1**	**b_1**	$(x-a_1)$	$(x-a_0).(x-a_1).(x-a_2).(x-a_3)$
P2	**a_2**	**x**	**k_2**	**b_2**	$(x-a_2)$	$(x-a_0).(x-a_1).(x-a_2).(x-a_3)$
P3	**a_3**	**x**	**k_3**	**b_3**	$(x-a_3)$	$(x-a_0).(x-a_1).(x-a_2).(x-a_3)$

Pr.	(3) Cálculo de todos los $L_i(x)$ en paralelo $L_i= B_i./(A_i.\ k_i)$ *($L_i(x)$ se obtiene en Pi)*	(4) Cálculo en paralelo $C_i = b_i\ .\ L_i$ *($b_i\ .\ L_i(x)$ se obtiene en Pi)*	(5) Reducción del contenido de C_i con resultado en P0 *(P(x) se obtiene en P0)*
P0	$(x-a_1).(x-a_2).(x-a_3)/\ k_0$	$b_0\ .(x-a_1).(x-a_2).(x-a_3)/\ k_0$	$P=\sum_{i=0}^{n}(C_i)=\sum_{i=0}^{n}(b_i \times L_i)$
P1	$(x-a_0).(x-a_2).(x-a_3)/\ k_1$	$b_1\ .(x-a_0).(x-a_2).(x-a_3)/\ k_1$	
P2	$(x-a_0).(x-a_1).\ (x-a_3)/\ k_2$	$b_2\ .(x-a_0).(x-a_1).\ (x-a_3)/\ k_2$	
P3	$(x-a_0).(x-a_1).(x-a_2)/\ k_3$	$b_3\ .(x-a_0).(x-a_1).(x-a_2)/\ k_3$	

Como se puede ver en el trazado del algoritmo para n=3 mostrado en las tablas, se usan un total de 2 funciones de comunicación colectivas (pasos (2) y (5) en la tabla). En el paso (2) del algoritmo se usa una operación de "todos reducen" para obtener en todos los procesadores los productos de todas las restas (xa_i). En el paso (5) y último se realiza una operación de reducción para obtener las sumas de todos los productos (b_i x L_i) en el proceso 0.

◀◀◀◀◀◀

Ejercicio 11. **(a)** Escriba un programa secuencial con notación algorítmica (podría escribirlo en C) que determine si un número de entrada, `x`, es primo o no. El programa imprimirá si es o no primo. Tendrá almacenados en un vector, `NP`, los `M` números primos entre 1 y el máximo valor que puede tener tener un número de entrada al programa.

(b) Escriba una versión paralela del programa anterior para un multicomputador usando un estilo de programación paralela de paso de mensajes. El proceso 0 tiene inicialmente el número `x` y el vector `NP` en su memoria e imprimirá en pantalla el resultado. Considere que la herramienta de programanción ofrece funciones send()/receive() para implementar una comunicación uno-a-uno asíncrona, es decir, con función `send(buffer,count,datatype,idproc,group)` no bloqueante y `receive(buffer,count,datatype,idproc,group)` bloqueante. En las funciones send()/receive() se especifica:

- `group`: identificador del grupo de procesos que intervienen en la comunicación.
- `idproc`: identificador del proceso al que se envía o del que se recibe.
- `buffer`: dirección a partir de la cual se almacenan los datos que se envían o los datos que se reciben.
- `datatype`: tipo de los datos a enviar o recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, …).
- `count`: número de datos a transferir de tipo `datatype`.

Solución

(a) Programa secuencial que determina si `x` es o no primo Versión 1 Pre-condición `x`:

número de entrada {2,…,`MAX_INPUT`}. `MAX_INPUT`: máximo valor de la entrada `M`:

número de primos entre 2 y `MAX_INPUT` (ambos incluidos).

`NP`: vector con M+1 componentes (los M nº primos desde 2 hasta `MAX_INPUT` en `NP[0]` hasta `NP[M-1]`, a `NP[M]` se asignará en el código `x+1`)

Versión 1 Pos-condición

Imprime en pantalla si el número `x` es primo o no

Código

Versión 1 A	Versión 1 B

```
if (x>NP[M-1]) {                           if (x>NP[M-1]) {
   print("%u supera el máximo primo a    print("%u supera el máximo primo a detectar
%u \n", x, NP[M-1]); detectar %u \n", x, NP[M-1]);
   exit(1);                                   exit(1);
}                                          }
NP[M]=x+1;                                 NP[M]=x+1;
i=0;
while (x<NP[i]) do i++;                    for (i=0; x<NP[i];i++) {}

if (x==NP[i])                              if (x==NP[i])
     printf("%u ES primo\n", x);         printf("%u ES primo\n", x); else
printf("%u NO ES primo\n", x);      else printf("%u NO es primo\n", x);
```

Versión 1: El número de instrucciones de comparación depende de en qué posición se encuentre el número primo `x` en NP. En el peor caso (cuando `x` no es primo) el bucle recorre todo el vector realizándose M+1 comparaciones en el bucle y 1 comparaciones en cada `if/else` (segundo `if` en el código). Resumiendo: M+1 comparaciones en el bucle (Orden de complejidad de `M`). Versión 2 Pre-condición `x`: número de entrada {2,…,`MAX_INPUT`}. `MAX_INPUT`: máximo valor de la entrada

`M`: número de primos entre 2 y `MAX_INPUT` (ambos

incluidos) `NP`: vector con los `M` nº primos desde 2 hasta

`MAX_INPUT` `xr`: raíz cuadrada de `x`

Versión 2 Pos-condición

Imprime en pantalla si el número `x` es primo o no Código

Versión 2

```
 if (x>NP[M-1]) {    print("%u supera el máximo primo a detectar
%u \n", x, NP[M-1]);    exit(1);
} xr = sqrt(x); for ( i=0 ;  (NP[i]<=xr) && (x % NP[i]!=0); i++)
{} ; // % = módulo

if (NP[i]<=xr) printf("%u ES primo", x);
else printf("%u NO ES primo", x);
```

Versión 2: El número de instrucciones de comparación depende de en qué posición, `nr`, de `NP` se encuentre el primer número primo mayor que `sqrt(x)`. En el peor caso (cuando `x` no es primo) se recorre hasta la posición `nr`, realizándose `nr+1` comparaciones y `nr` módulos/comparaciones en el bucle (se supone que el bucle se implementa con dos saltos condicionales y que el primero evalúa `NP[i]<=xr` y el segundo si el resultado del módulo ha sido o no cero) y 1 comparación en cada `if/else`. Resumiendo: Orden de complejidad de `nr`.

(b) Programa paralelo para multicomputador que determina si `x` es o no primo

Se van a repartir las iteraciones del bucle entre los procesadores del grupo. Todos los procesos ejecutan el mismo código

Pre-condición `x`: número de entrada {2,…,`MAX_INPUT`}; `MAX_INPUT`: máximo valor de la entrada

xr: almacena la raíz cuadrada de x (sólo se usa en la versión 2) `M`: número de primos entre 2 y `MAX_INPUT` (ambos incluidos) `grupo`: identificador del grupo de procesos que intervienen en la comunicación. `num_procesos`: número de procesos en `grupo`.
`idproc`: identificador del proceso (dentro del grupo de `num_procesos` procesos) que ejecuta el código
`tipo`: tipo de los datos a enviar o recibir
`NP`: vector con los `M` nº primos entre 2 y `MAX_INPUT` (en la versión 1 tendrá M+1 componentes) `b`, `baux`: variables que podrán tomar dos valores 0 (=false) o 1 (=true, si se localiza `x` en la lista de números primos)

Pos-condición

Imprime en pantalla si el número `x` es primo o no

Código

Versión 1 **Versión 2**

```
if (x>NP[i]) {                                 if (x>NP[i]) {
   print("%u supera el máximo primo a    print("%u supera el máximo primo a detectar
%u \n", x, NP[M-1]); detectar %u \n", x, NP[M-1]);
   exit(1);                                       exit(1);
}                                              }
NP[M]=x+1;

//Difusión de x y NP    //Difusión de x y NP if (idproc==0)
      if (idproc==0)
   for  (i=1;  i<num_procesos)  {           for  (i=1;  i<num_procesos)  {
send(NP,M+1,tipo,i,grupo);                          send(NP,M,tipo,i,grupo);
send(x,1,tipo,i,grupo);        send(x,1,tipo,i,grupo);
   }                                              }
else {                                         else {
   receive(NP,M+1,tipo,0,grupo);                       receive(NP,M,tipo,0,grupo);
receive(x,1,tipo,0,grupo);    receive(x,1,tipo,0,grupo);
}                                              }

//Cálculo paralelo, asignación estática //Cálculo paralelo, asignación estática
i=id_proc;                                     i=id_proc; xr = sqrt(x);
while (x<NP[i]) do {                            while ((NP[i]<=xr)&&(x % NP[i]!=0)) do {
 i=i+num_procesos;                                i=i+num_procesos;
}                                              }
b=(x==NP[i])?1:0;                              b=(NP[i]<=xr)?1:0;

//Comunicación resultados                      //Comunicación resultados
if (idproc==0)                                 if (idproc==0)
   for   (i=1;   i<num_procesos)   {                 for   (i=1;   i<num_procesos)   {
receive(baux,1,tipo,i,grupo);       receive(baux,1,tipo,i,grupo);
       b = b | baux;                                  b = b | baux;
   }                                              }
else  send(b,1,tipo,0,grupo);                  else send(b,1,tipo,0,grupo);

//Proceso 0 imprime resultado                  // Proceso 0 imprime resultado
if (idproc==0)    if (idproc==0)     if (b!=0)           if
(b!=0)
        printf("%d ES primo", x);           printf("%d ES primo", x);    else
printf("%d NO es primo", x);     else printf("%d NO es primo", x);
```

<u>Versión 1:</u> El número de instrucciones de comparación en la parte de cálculo paralelo depende de en qué posición se encuentre el número primo en NP. En el peor caso (cuando `x` no es primo) se recorre todo el vector entre todos los procesos, pero cada uno accede a un subconjunto de componentes del vector; en particular, a `M/num_procesos` componentes si `num_procesos` divide a `M`. Entonces, si `num_procesos` divide a `M`, cada proceso realizará `M/num_procesos+1` comparaciones y `M/num_procesos` en el bucle y 1 comparación más para obtener `b`. Se realiza ese número de comparaciones en el bucle porque cada proceso actualiza el índice `i` del `while` en cada iteración sumándole el número de procesos `num_procesos`. Por tanto, se asignan a los procesos las posiciones de NP con las que comparar en Round-Robin; por ejemplo, el proceso 0 ejecutará las iteraciones para `i` igual a 0, `num_procesos`, `2*num_procesos`, `3*num_procesos`,… En total, todos los procesos en conjunto harían (`M/num_procesos+1+1)*num_procesos` comparaciones en paralelo `(M+2*num_procesos)`. Todos los procesos realizarán el mismo número de comparaciones si `num_procesos` divide a `M` (como se ha comentado más arriba), en caso contrario, debido a la asignación Round-Robin, algunos harán una iteración más del bucle que otros y, por tanto, una comparación más.

<u>Resumiendo</u>:

- Si `num_procesos` divide a `M`: cada proceso realiza `M/num_procesos+1` comparaciones en el bucle y 1 comparación más para obtener `b (Orden de M/num_procesos)`

- Si `num_procesos` NO divide a `M`: algunos procesos realizan una comparación más que el resto, estos procesos realizan Truncado(`M/num_procesos)+2` comparaciones en el bucle y 1 comparación más para obtener `b (Orden de M/num_procesos)`

Versión 2: El número de instrucciones de comparación en la parte de cálculo paralelo depende de en qué posición, `nr`, de `NP` se encuentre el primer número primo mayor que `sqrt(x)`. En el peor caso (cuando `x` no es primo) se recorre hasta `nr` entre todos los procesos, pero cada uno accede a un subconjunto de componentes del vector entre 0 y `nr`; en particular, a `nr/num_procesos` componentes si `num_procesos` divide a `nr`. Entonces, si `num_procesos` divide a `nr`, cada proceso realizará `nr/num_procesos +1` comparaciones y `nr/num_procesos` módulos/comparaciones en el bucle y 1 comparación más para obtener `b`. Se realiza ese número de operaciones en el bucle porque cada proceso actualiza el índice `i` del `while` en cada iteración sumándole el número de procesos `num_procesos`. Por tanto, se asignan a los procesos las posiciones de NP con las que comparar en Round-Robin; por ejemplo, el proceso 1 ejecutará las iteraciones para `i` igual a 1, `num_procesos+1`, `2*num_procesos+1`, `3*num_procesos+1`,…. En total, todos los procesos en conjunto harían `2*nr+2*num_procesos` operaciones (operación comparación y operación módulo/comparación). Todos los procesos realizarán el mismo número de operaciones si `num_procesos` divide a `nr`, en caso contrario, debido a la asignación Round-Robin, algunos harán una iteración más del bucle que otros y, por tanto, dos operaciones más (una operación comparación más una operación módulo/comparación).

Resumiendo:

- Si `num_procesos` divide a `nr`: cada proceso realiza `2*nr+2*num_procesos` operaciones (operación comparación y operación módulo/comparación). El orden de complejidad es de `nr/num_procesos`.
- Si `num_procesos` NO divide a `nr`: algunos procesos realizan dos comparaciones y un módulo más que el resto, estos procesos realizan 2*Truncado(`nr /num_procesos)+3` operaciones en el bucle y 1 operación de comparación para obtener `b`. El orden de complejidad es de `nr/num_procesos`.

◄◄◄◄◄◄

Ejercicio 12.

◄

Ejercicio 13.

◄

Ejercicio 14.

◄