

Esta expresión se denomina **ganancia escalable** en J. L. Gustafson, 1988. Mientras que Gene M. Amdahl asume que el tiempo de ejecución secuencial (o tamaño del problema) se mantiene constante conforme se incrementa el número de procesadores, concluyendo que la ganancia en prestaciones está limitada debido a la parte del código no paralelizable, John L. Gustafson mantiene constante el tiempo de ejecución paralelo y muestra que la ganancia puede crecer con pendiente constante conforme se incrementa el número de procesadores.

Para evaluar en qué medida las prestaciones que ofrece un sistema para un código paralelo se acercan a las prestaciones máximas que idealmente debería ofrecer dado los procesadores disponibles en el mismo, se usa la siguiente expresión de **eficiencia** (se ha tenido en cuenta la expresión (1)):

$$E(p) = \frac{\text{Prestacion es}(p, n)}{p \times \text{Prestacion es}(1, n)} = \frac{S(p)}{p} \quad (8)$$

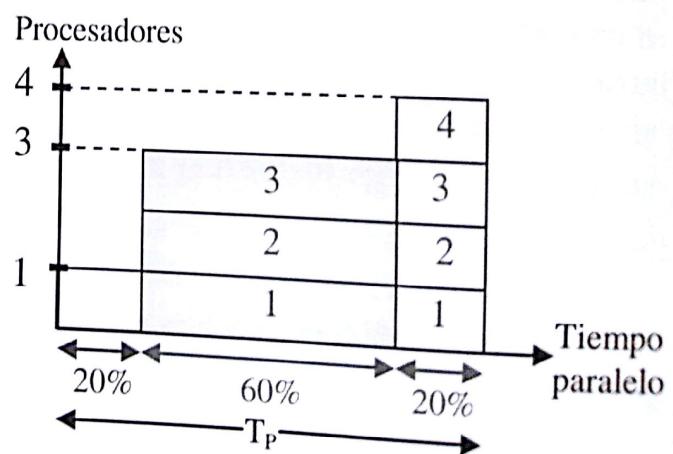
5. Problemas

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

Solución

Datos del ejercicio:

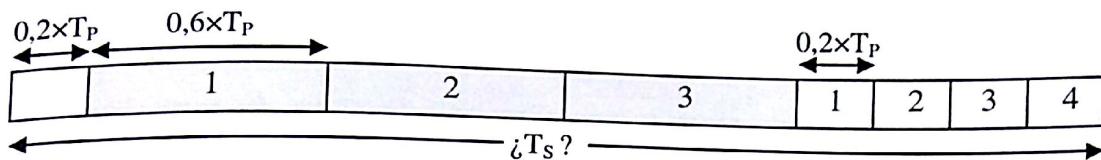
En el gráfico de la derecha se representan los datos del ejercicio. Estos datos son la fracción del tiempo de ejecución paralelo (T_p) que supone el código no paralelizable (20% de T_p , es decir, $0,2 \times T_p$), la fracción que supone el código paralelizable en 3 procesadores ($0,6 \times T_p$) y la que supone el código paralelizable en 4 procesadores ($0,2 \times T_p$). 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.

(a) Tiempo de ejecución secuencial, T_s :

Debido a que sólo se usa un procesador (núcleo), los trozos de código representados en la gráfica anterior se deben ejecutar secuencialmente, es decir, uno detrás de otro (en algún orden), como se ilustra el gráfico siguiente:



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

(b) Ganancia en velocidad, $S(p)$:

$$S(4) = \frac{T_s}{T_p(4)} = \frac{2,8 \times T_p(4)}{T_p(4)} = 2,8$$

(c) Eficiencia, $E(p)$:

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



Problema 2. Un programa tarda 20 s en ejecutarse en un procesador P1, y requiere 30 s en otro procesador P2. Si se dispone de los dos procesadores para la ejecución del programa (despreciamos sobrecarga):

- (a) ¿Qué tiempo tarda en ejecutarse el programa si la carga de trabajo se distribuye por igual entre los procesadores P1 y P2?
- (b) ¿Qué distribución de carga entre los dos procesadores P1 y P2 permite el menor tiempo de ejecución utilizando los dos procesadores en paralelo? ¿Cuál es este tiempo?

Solución

Datos del ejercicio:

$$T^{P1}=20\text{ s}$$

$$T^{P2}=30\text{ s}$$

- (a) Tiempo de ejecución paralelo distribuyendo la carga por igual, T_p :

$$T_p^{P1}(1/2) = 20s/2 = 10 \text{ s} \text{ y } T_p^{P2}(1/2) = 30s/2 = 15 \text{ s}$$

$$T_p^{P1,P2} = \max(20s/2, 30s/2) = 15 \text{ s}$$

En el entorno de trabajo heterogéneo que conforma P1 y P2, si la carga de trabajo se distribuye por igual entre los procesadores, el tiempo de ejecución en paralelo lo determinará el procesador que acabe más tarde de ejecutar su mitad. La distribución de carga de trabajo no se ha hecho de forma que los procesadores empiecen y terminen a la vez (no se ha equilibrado).

(b) Distribución con menor tiempo y tiempo de ejecución paralelo en ese caso, T_p :

El mejor caso se obtiene si se puede distribuir la carga de forma que los dos procesadores empiecen y terminen a la vez, el tiempo de ejecución sería:

$$T_p^{P1}(x) = T_p^{P2}(1-x) \Rightarrow 20s \times x = 30s \times (1-x) \Rightarrow 2 \times x = 3 \times (1-x) \Rightarrow 5 \times x = 3 \Rightarrow x = 3/5$$

$3/5 = 0,6$ para P1 y $2/5 = 0,4$ para P2

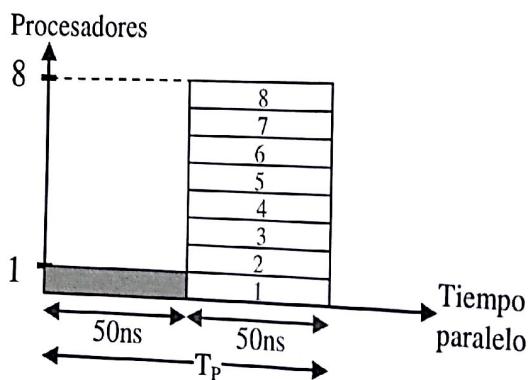
$$T_p^{P1}(3/5) = 20s \times 3/5 = 12 \text{ s}$$



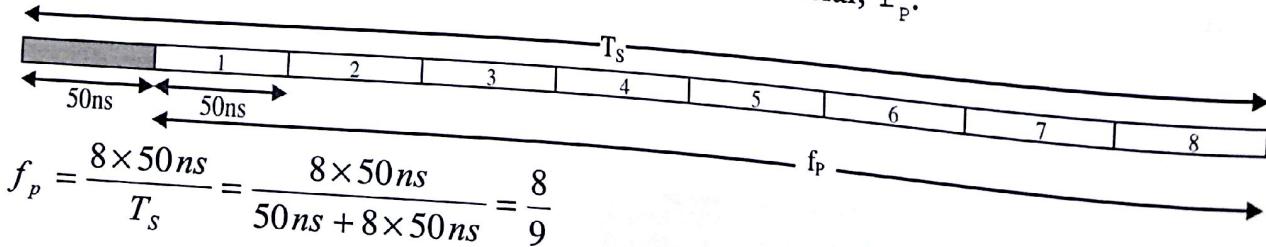
Problema 3. ¿Cuál es fracción de código paralelo de un programa secuencial que, ejecutado en paralelo en 8 procesadores, tarda un tiempo de 100 ns, durante 50 ns utiliza un único procesador y durante otros 50 ns utiliza 8 procesadores (distribuyendo la carga de trabajo por igual entre los procesadores)?

Solución

Datos del programa:



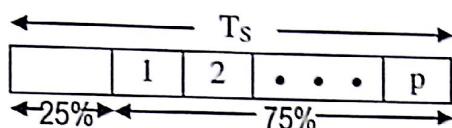
Fracción de código paralelizable del programa secuencial, f_p :



Problema 4. Un 25% de un programa no se puede parallelizar, el resto se puede distribuir por igual entre cualquier número de procesadores. (a) ¿Cuál es el máximo valor de ganancia de velocidad que se podría conseguir al parallelizarlo en p procesadores, y con infinitos? (b) ¿A partir de cuál número de procesadores se podrían conseguir ganancias mayores o iguales que 2?

Solución

Datos del ejercicio:



(a) Ganancia en velocidad con p procesadores y máxima ideal, $S(p)$:

$$S(p) = \frac{T_s}{T_p} = \frac{T_s}{0,25 \times T_s + \frac{0,75 \times T_s}{p}} = \frac{1}{0,25 + \frac{0,75}{p}}$$

$$S(p) = \frac{1}{0,25 + \frac{0,75}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{0,25} = 4$$

(b) Número de procesadores a partir del cual se obtienen ganancias mayores o iguales que 2:

$$S(p) = \frac{1}{0,25 + \frac{0,75}{p}} \geq 2 \Rightarrow 1 \geq 0,5 + \frac{1,5}{p} \Rightarrow 0,5 \geq \frac{1,5}{p} \Rightarrow p \geq \frac{1,5}{0,5} = 3$$

Problema 5. En la Figura 8, se presenta el grafo de dependencia entre tareas para una aplicación. La figura muestra la fracción del tiempo de ejecución secuencial que la aplicación tarda en ejecutar las tareas del grafo. Suponiendo un tiempo de ejecución secuencial de 60 s, que las tareas no se pueden dividir en tareas de menor granularidad y que el tiempo de comunicación es despreciable, y que el tiempo de ejecución en paralelo y la ganancia en velocidad en un computador con:

(a) 4 procesadores.

(b) 2 procesadores.

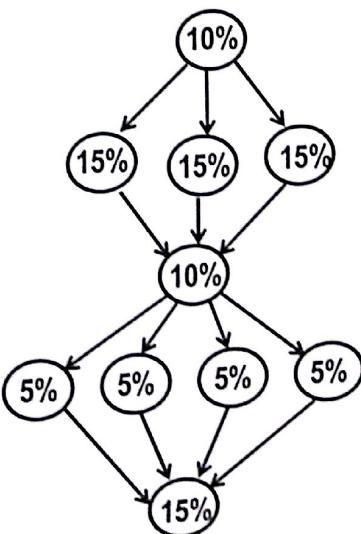
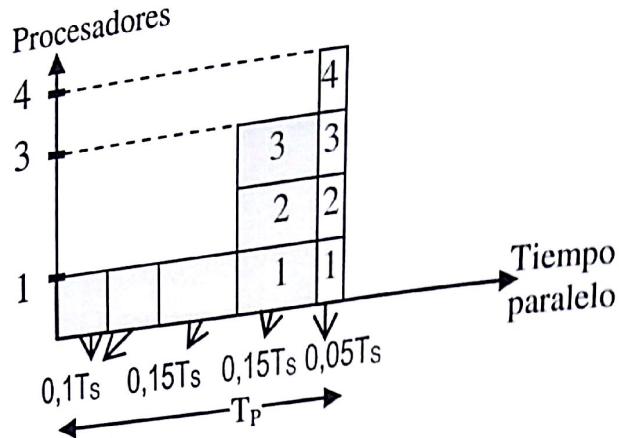


Figura 8. Grafo de tareas del ejercicio 5.

Solución

(a) Tiempo de ejecución paralelo, T_p y ganancia en velocidad, $S(p)$, para $p=4$:

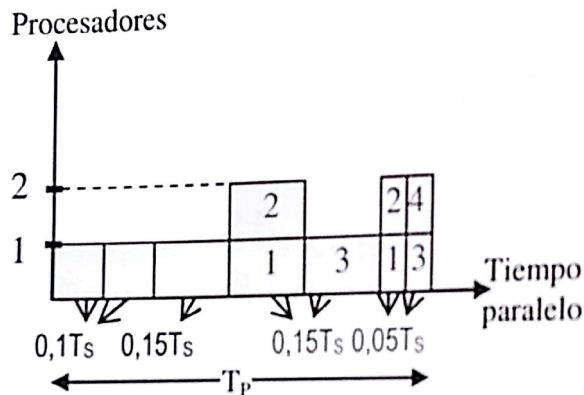


$$T_p(4) = (0,1 + 0,1 + 0,15) \times T_s + 0,15 \times T_s + 0,05 \times T_s = (0,35 + 0,15 + 0,05) \times T_s$$

$$= 0,55 \times T_s = 0,55 \times 60s = 33s$$

$$S(4) = \frac{T_s}{T_p(4)} = \frac{T_s}{0,55 \times T_s} = \frac{1}{0,55} \approx 1,82$$

(b) Tiempo de ejecución paralelo, T_p y ganancia en velocidad, $S(p)$, para $p=2$:



$$T_p(2) = (0,1 + 0,1 + 0,15) \times T_s + 2 \times 0,15 \times T_s + 2 \times 0,05 \times T_s = (0,35 + 0,3 + 0,1) \times T_s$$

$$= 0,75 \times T_s = 0,75 \times 60s = 45s$$

$$S(4) = \frac{T_s}{T_p(4)} = \frac{T_s}{0,75 \times T_s} = \frac{1}{0,75} \approx 1,33$$

Problema 6. Un programa se ha conseguido dividir en 10 tareas. El orden de precedencia entre las tareas se muestra con el grafo dirigido de la Figura 9. La ejecución de estas tareas en un procesador supone un tiempo de 2 s. El 10% de ese tiempo es debido a la ejecución de la tarea 1; el 15% a la ejecución de la tarea 2; otro 15% a la ejecución de 3; cada tarea 4, 5, 6 o 7 supone el 9%; un 8% supone la tarea 8; la tarea 9

un 10%; por último, la tarea 10 supone un 6%. Se dispone de una arquitectura con 8 procesadores para ejecutar la aplicación. Se considera que el tiempo de comunicación se puede despreciar. Conteste a las siguientes preguntas:

- (a) ¿Qué tiempo tarda en ejecutarse el programa en paralelo?

- (b) ¿Qué ganancia en velocidad se obtiene con respecto a su ejecución secuencial?

Solución

Datos del ejercicio: $T_s = 2$ s y $p = 8$

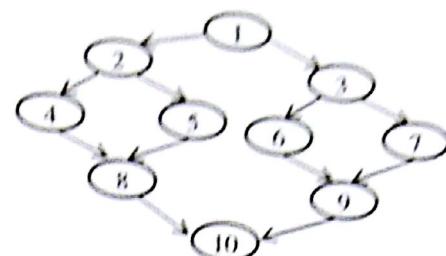
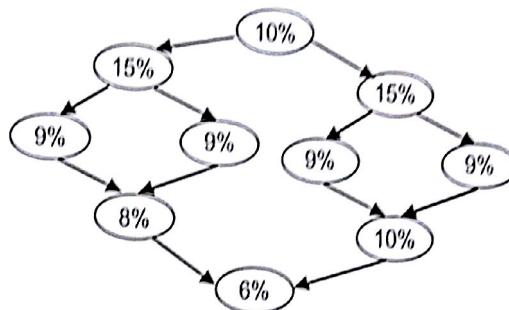
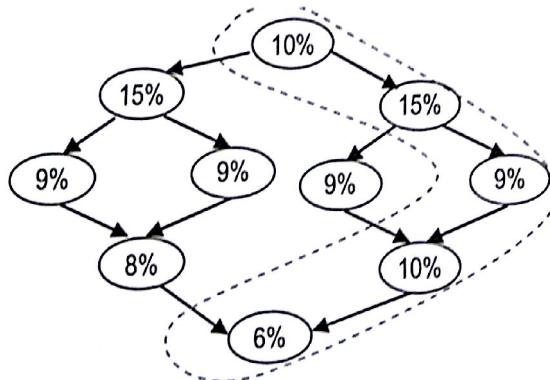


Figura 9. Grafo de dependencia entre tareas del ejercicio 6.



- (a) Tiempo de ejecución paralelo, T_p :

Hay procesadores suficientes como para poder aprovechar todo el grado de paralelismo del programa. En este caso el grado es 4, ya que 4 es el número máximo de tareas que se pueden ejecutar en paralelo. El tiempo de ejecución paralelo dependerá del camino más largo (que supone mayor tiempo de ejecución) en el grafo de dependencia entre tareas. Como suponemos despreciable el tiempo de comunicación, el camino más largo va a depender sólo del tiempo de cálculo de las tareas. En este caso hay dos caminos que llevan al mayor tiempo de ejecución, uno de ellos está englobado en línea discontinua en el siguiente grafo, ambos caminos pasan por la tarea 9.



$$T_p(8) = T_p(4) = (0,1 + 0,15 + 0,09 + 0,1 + 0,06) \times T_s = 0,5 \times T_s = 0,5 \times 2s = 1s$$

(b) Ganancia en velocidad, $S(p)$:

$$S(8) = S(4) = \frac{T_s}{T_p(4)} = \frac{T_s}{0,5 \times T_s} = \frac{1}{0,5} = 2$$

Problema 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}
```

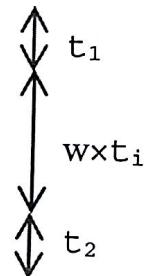
Los cálculos antes y después del bucle suponen un tiempo de t_1 y t_2 , respectivamente. Una iteración del ciclo supone un tiempo t_i . En la ejecución paralela, la inicialización de p procesos supone un tiempo k_1p (k_1 constante), los procesos se comunican y se sincronizan, lo que supone un tiempo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



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 paralelo, 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 ?

$$T_p(p, w) = t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p \quad (9)$$

En la expresión (9), 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 divisor), y el término englobado con línea discontinua o tiempo de sobrecarga ($k \times p$), crece conforme se incrementa p con pendiente k constante (ver ejemplos en Figura 10 y Figura 11). Las oscilaciones en las figuras del tiempo de cálculo paralelo se deben al redondeo al entero superior del cociente w/p , pero la tendencia 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.

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 (9) con el fin de hacer la función continua (después se comentará la influencia del redondeo en el cálculo del mínimo):

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

$$T'_p(p, w) = 0 - \frac{w}{p^2} \times t_i + k \Rightarrow \frac{w}{p^2} \times t_i = k \Rightarrow p = \sqrt{\frac{w \times t_i}{k}} \quad (10)$$

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 (9), se deberían comprobar necesariamente:

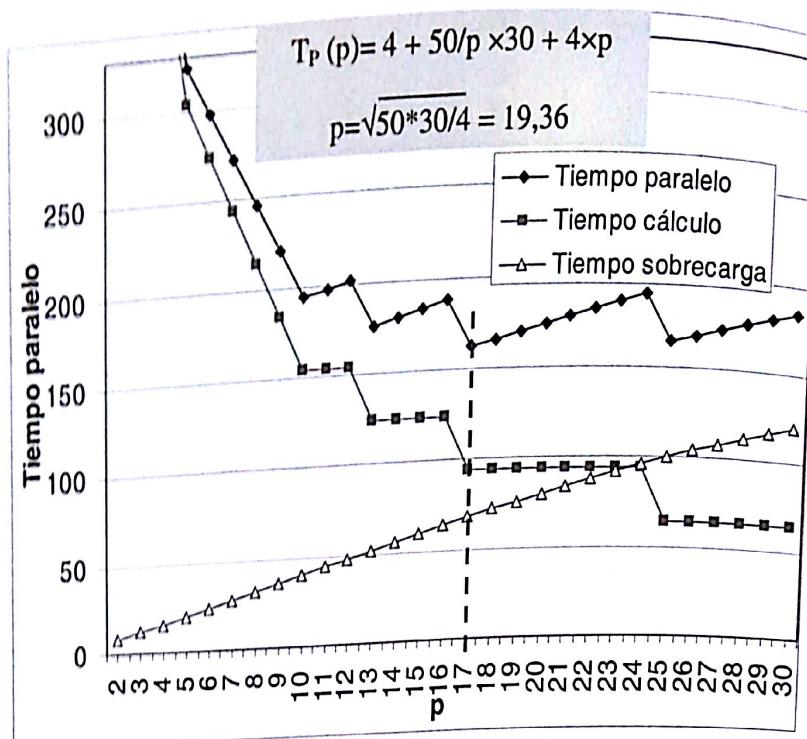


Figura 10. Tiempo de ejecución paralelo para un caso particular en el que $w=50$, t_i es 30 unidades de tiempo, t son 4 unidades de tiempo y k son 4 unidades de tiempo. El mínimo se alcanza para $p=17$.

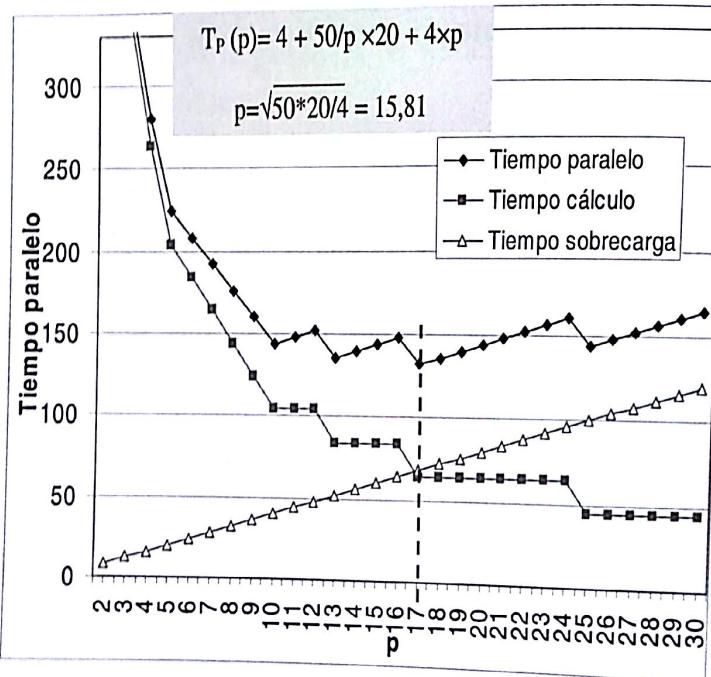


Figura 11. Tiempo de ejecución paralelo para un caso particular en el que $w=50$, t_i es 20 unidades de tiempo, t son 4 unidades de tiempo y k son 4 unidades de tiempo. El mínimo se alcanza para $p=17$.

- El natural p' menor o igual y más alejado al resultado p generado con (10) para el que

$$\left\lceil \frac{w}{p'} \right\rceil = \left\lceil \frac{w}{p} \right\rceil$$

Observar, que en el ejemplo de la Figura 10, aunque de (10) se obtiene 19,36, el p con menor tiempo es 17 debido al efecto del redondeo.

2. El natural p' mayor y más próximo al p generado con (10) para el que

$$\left\lceil \frac{w}{p'} \right\rceil = \left\lceil \frac{w}{p} \right\rceil - 1$$

Observar, que en el ejemplo de la Figura 11, aunque de (10) 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.

Problema 8. Supongamos que se va a ejecutar en paralelo la suma de n números en una arquitectura con p procesadores (p y n potencias de dos) utilizando un grafo de dependencias en forma de árbol (divide y vencerás) para las tareas.

- (a) Dibujar el grafo de dependencias entre tareas para $n = 16$ y p igual a 8. Hacer una asignación de tareas a flujos de instrucciones.
- (b) Obtener el tiempo de cálculo paralelo para cualquier n y p con $n > p$ suponiendo que se tarda una unidad de tiempo en realizar una suma.
- (c) Obtener el tiempo comunicación del algoritmo suponiendo (1) que las comunicaciones en un nivel del árbol se pueden realizar en paralelo en un número de unidades de tiempo igual al número de datos que recibe o envía un proceso en cada nivel del grafo de tareas (tenga en cuenta la asignación de tareas a procesos que ha considerado en el apartado (a)) y (2) que los procesadores que realizan las tareas de las hojas del árbol tienen acceso sin coste de comunicación a los datos que utilizan dichas tareas.
- (d) Suponiendo que el tiempo de sobrecarga coincide con el tiempo de comunicación calculado en (c), obtener la ganancia en prestaciones.
- (e) Obtener el número de procesadores para el que se obtiene la máxima ganancia con n números.

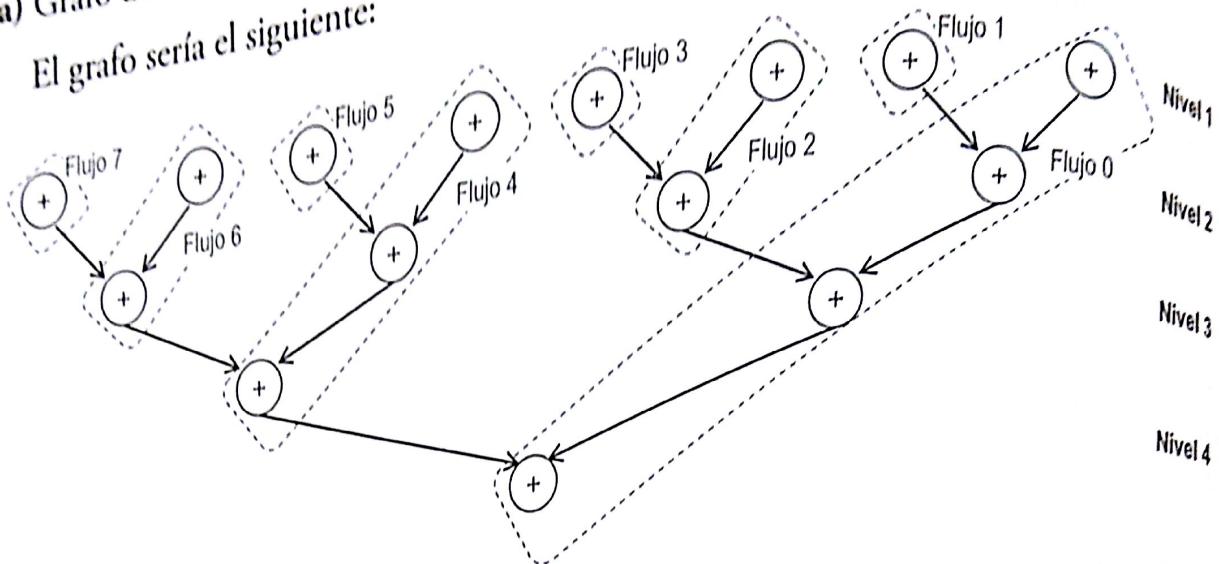
Solución

Datos del ejercicio:

Suma de n números en paralelo con $p=2^v$ y $n=2^u$

(a) Grafo de tareas para $n=16$ y $p=2$ y asignación a procesos.

El grafo sería el siguiente:



Los flujos de instrucciones aparecen en el grafo con línea discontinua. El grado de paralelismo es 8, se puede aprovechar, por tanto, por los 8 procesadores disponibles.

(b) Tiempo de cálculo paralelo, T_c , para $n>p$.

Datos: Cada suma supone una unidad de tiempo.

Primero se va a obtener el tiempo de cálculo paralelo para el ejemplo del grafo del apartado (a). Cada tarea del grafo tiene que realizar una suma, lo que supone una unidad de tiempo. Todas las tareas del nivel 1 del árbol se pueden ejecutar en paralelo al no haber dependencias entre ellas (no hay arcos del grafo entre ellas), esta ejecución supone 1 unidad de tiempo. También todas las tareas del nivel 2 se pueden ejecutar en paralelo en una unidad, al igual que las tareas del nivel 3. La tarea de nivel 4 será la última en ejecutarse y supondrá una unidad. En total se consume 4 unidades:

$$T_c(16,8) = 4 = 1 + \lg_2 8$$

En general, para un $p=n/2$ sería (habrá $1+\lg_2 p$ niveles de tareas en el árbol binario):

$$T_c(p,n; n=2p) = 1 + \lg_2 p$$

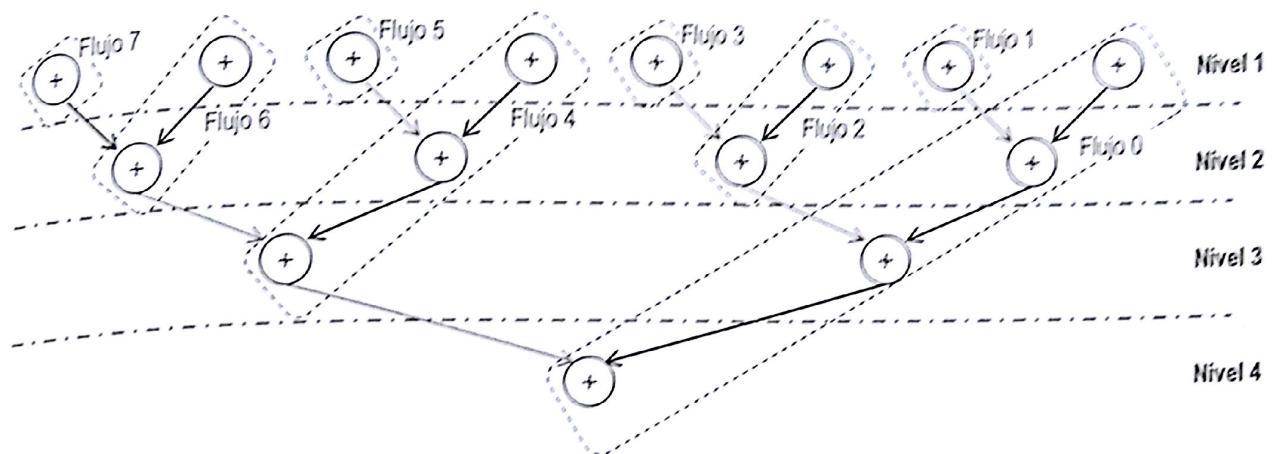
Para un $p>n$ ($n=kp$, con k potencia de 2), teniendo en cuenta que el máximo grado de paralelismo aprovechable sería igual al número de procesadores (flujos), se tendrán tantas tareas en el primer nivel del árbol como procesadores. En total habrá $1+\lg_2 p$ niveles de tareas en el árbol binario. Cada una de las tareas del primer nivel tendría asignada la suma de n/p números (todas el mismo número al ser n divisible por p), por lo que la ejecución en paralelo de estas tareas supondrá $n/p-1$ unidades de tiempo.

Para cada uno de los niveles restantes, la ejecución en paralelo de las tareas supondrá una unidad, ya que únicamente realizan la suma de dos números:

$$T_c(p, n; n = kp) = \frac{n}{p} - 1 + \lg_2 p$$

(c) Tiempo de comunicación/sincronización, $T_{C/S}$, para $n > p$.

La comunicación en cada uno de los niveles del árbol supone 1 unidad de tiempo, teniendo en cuenta el enunciado y que sólo se recibe un dato en cada comunicación. A continuación se han destacado en el grafo los niveles de comunicación trazando líneas discontinuas horizontales, las flechas entre flujos que atraviesan estas líneas (en gris) son comunicaciones a través de la red:



El número de niveles de comunicación en el ejemplo es 3, en general será $\lg_2 p$:

$$T_{C/S}(p) = \lg_2 p$$

(d) Ganancia en prestaciones, $S(p, n)$

$$S(p, n) = \frac{T_s(n)}{T_p(p, n)} = \frac{n - 1}{\frac{n}{p} - 1 + 2 \times \lg_2 p}$$

(e) Número de procesadores para el que se obtiene la máxima ganancia con n números.

Analíticamente se podría encontrar el mínimo de la función para T_p igualando a 0 la primera derivada:

$$T_p(p) = \frac{n}{p} - 1 + 2 \times \lg_2 p$$

$$T'_p(p) = -\frac{n}{p^2} + \frac{2}{p \times L2} = 0 \Rightarrow \frac{n}{p} = \frac{2}{L2} \Rightarrow p = \frac{n \times L2}{2} \approx 1,3863 \times n \approx \frac{n}{2,8854} \quad (11)$$

$$T''_P(p) = \frac{2 \times p \times n}{p^4} - \frac{2}{L2 \times p^2} = \frac{2 \times n}{p^3} - \frac{2}{L2 \times p^2} > 0 \Rightarrow \frac{2 \times n}{p^3} > \frac{2}{L2 \times p^2} \Rightarrow \frac{n}{p} > \frac{1}{L2} \Rightarrow L2 \times n > p$$

El valor de p obtenido, $n \times L2/2$, es menor que $n \times L2$ (L nota logaritmo neperiano), luego se trata de un mínimo y no de un máximo.

En la Figura 12 se ha representado para $n=32$ el tiempo paralelo, T_p , el tiempo de cálculo de las tareas en paralelo, T_c , y el tiempo de sobrecarga, T_o , en función de p (suponiendo que la función es aplicable para cualquier valor de p). En nuestro caso p es potencia de 2. El mínimo que se obtiene de la expresión (11) está entre dos potencias de 2 ($n/2$ y $n/4$), se tendrá que obtener cuál de las dos supone un tiempo menor; en el programa paralelo de este ejercicio, realmente, los dos suponen el mismo tiempo. Este tiempo se puede calcular fácilmente evaluando T_p en $n/2$ y $n/4$:

$$T_p(n/2) = \frac{2 \times n}{n} - 1 + 2 \times \lg_2 \frac{n}{2} = 1 + 2 \times (\lg_2 n - \lg_2 2) = 1 + 2 \times (\lg_2 n - 1) = 2 \times \lg_2 n - 1$$

$$T_p(n/4) = \frac{4 \times n}{n} - 1 + 2 \times \lg_2 \frac{n}{4} = 3 + 2 \times (\lg_2 n - \lg_2 4) = 3 + 2 \times (\lg_2 n - 2) = 2 \times \lg_2 n - 1$$

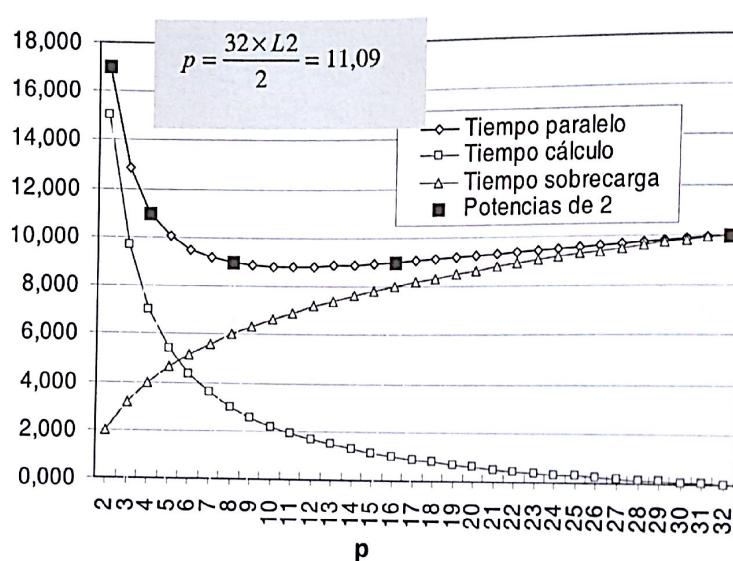
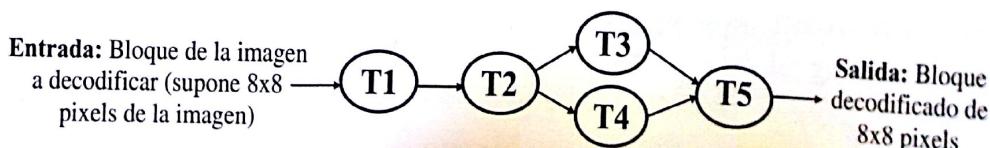


Figura 12. Tiempo de ejecución paralelo suponiendo que la expresión obtenida para el tiempo paralelo se puede aplicar a todo p . Los puntos válidos para este ejemplo son aquellos en los que p es potencia de 2 (representados con cuadrados). El mínimo, para el ejemplo, se alcanza en 8 y en 16.

Problema 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):



Las tareas 1, 2 y 5 se ejecutan en un tiempo igual a t , mientras que 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 8×8 píxeles. Si se procesa una imagen que se puede dividir en n bloques de 8×8 píxeles, a cada uno de esos n bloques se aplica el grafo de tareas de la figura. Obtener 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 8×8 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) y (b), $T_p(p, n)$.

Tiempo de ejecución secuencial $T_s(n)$:

Un procesador tiene que ejecutar las 5 tareas para cada bloque de 8×8 píxeles. El tiempo que dedica el procesador a cada bloque es de $3t$ (tareas 1, 2 y 5) + $3t$ (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 $T_p(5, n)$, $S(5, n)$.

Tabla 1. Asignación de tareas a procesadores (P1, P2, P3, P4 y P5), 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

Asignación de tareas a 5 procesadores P1, P2, P3, P4 y P5

Etapa 1 (t)	Etapa 2 (t)	Etapa 3 (1,5t)	Etapa 4 (t)	Terminado	Tiempo procesamiento
T1 (P1)	T2 (P2)	T3 (P3) y T4 (P4)	T5 (P5)		
0 - Bloque 1 - t					
t - Bloque 2 - 2t	t - Bloque 1 - 2t				Procesamiento Bloque 1 (4,5t)
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					

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 cauce segmentado consta de dos tiempos: el tiempo que tarda en procesarse la primera entrada (ver celdas con fondo gris en la tabla) más el tiempo que tardan cada uno del resto de bloques (entradas al cauce) en terminar, una detrás de otra. 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). El tiempo y la ganancia con la asignación de la Tabla 1 son:

$$T_p(5, n) = 4,5t + (n - 1) \times 1,5t = 3t + n \times 1,5t$$

$$S(5, n) = \frac{T_s(n)}{T_p(5, n)} = \frac{n \times 6t}{3t + n \times 1,5t} \xrightarrow{n \rightarrow \infty} 4$$

La ganancia se aproxima a 4 para n suficientemente grande.

- (b) Tiempo de ejecución paralelo y ganancia en prestaciones para 4 procesadores $T_p(4, n), S(4, n)$.

Tabla 2. Asignación de tareas a procesadores (P1,P2,P3 y P4), 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

Possible asignación de tareas a 4 procesadores P1, P2, P3 y P4

Etapa 1 (2t)	Etapa 2 (1,5t)	Etapa 3 (t)	Terminado	Tiempo procesamiento
T1 y T2 (P1)	T3 (P2) y T4 (P3)	T5 (P4)		
0 - Bloque 1 - 2t				
2t - Bloque 2 - 4t	2t - Bloque 1 - 3,5t			Procesamiento Bloque 1 (4,5t)
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
$T_{\text{entrada_etapa}} - \text{Bloque } x - T_{\text{salida_etapa}}$				

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 cauce segmentado 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 $1,5t$

(Etapa 2) y otra de t (Etapa 3). Si, por ejemplo, se hubieran agrupado T3 y T4 en un procesador la etapa más lenta supondría $3t$. El tiempo y la ganancia con la asignación de la tabla 2 son:

$$T_p(4, n) = 4,5t + (n - 1) \times 2t = 2,5t + n \times 2t$$

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

La ganancia se aproxima a 3 para n suficientemente grande.



Problema 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):

$$P(x) = \sum_{i=0}^n (b_i \cdot L_i(x)),$$

donde

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

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

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, 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 $A_i = (x - a_i)$ $((x - a_i) \text{ se obtiene en } P_i)$	(2) Todos reducen A_i con resultado en B_i : $B_i = \prod_{i=0}^n A_i$
	a_0	x	k_0	b_0		
P0	a_0	x	k_0	b_0	$(x - a_0)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$
P1	a_1	x	k_1	b_1	$(x - a_1)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$
P2	a_2	x	k_2	b_2	$(x - a_2)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$
P3	a_3	x	k_3	b_3	$(x - a_3)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$

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 P_i)	(4) Cálculo en paralelo $C_i = b_i \cdot L_i (b_i, L_i(x) se obtiene en P_i)$	(5) Reducción del contenido de C_i con resultado en P_0 ($P(x)$ se obtiene en P_0)
P0	$(x-a_1).(x-a_2).(x-a_3)/k_0$	$b_0 \cdot (x-a_1).(x-a_2).(x-a_3)/k_0$	$P = \sum_{i=0}^n (C_i) = \sum_{i=0}^n (b_i \times L_i)$
P1	$(x-a_0).(x-a_2).(x-a_3)/k_1$	$b_1 \cdot (x-a_0).(x-a_2).(x-a_3)/k_1$	
P2	$(x-a_0).(x-a_1).(x-a_3)/k_2$	$b_2 \cdot (x-a_0).(x-a_1).(x-a_3)/k_2$	
P3	$(x-a_0).(x-a_1).(x-a_2)/k_3$	$b_3 \cdot (x-a_0).(x-a_1).(x-a_2)/k_3$	

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 $(x-a_i)$. En el paso (5) y último se realiza una operación de reducción para obtener las sumas de todos los productos $(b_i \times L_i)$ en el proceso 0.

Problema 11. (a) Escribir un programa secuencial con notación algorítmica (se podría escribir en C) que determine si un número de entrada, x , es primo o no. El programa debe imprimir si el número es o no primo. Tendrá almacenados en un vector, NP, M números primos. Determine qué números primos se deberán almacenar si el máximo número de entrada x es MAX_INPUT. ¿De qué orden es el número de operaciones que hay que realizar en el código implementado? Justificar respuestas.

(b) Escribir 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. Considerar que la herramienta de programación ofrece funciones send() / receive() para implementar una comunicación uno-a-uno, en particular, con función send(buffer, count, datatype, idproc, group) no bloqueante y receive(buffer, count, datatype, idproc, group) bloqueante. Al ser la función receive() bloqueante, si no han llegado aún los datos cuando el flujo ejecuta la función, éste espera en la función hasta que se reciben, así se consigue la sincronización necesaria para que la comunicación se lleve a cabo. 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, flotante de 32 bits, entero de 64 bits, flotante de 64 bits, ...).
- count: número de datos a transferir de tipo datatype.

¿Cuál es el orden de complejidad del código implementado? (considere sólo cálculos, no tenga en cuenta el orden de complejidad de las comunicaciones o de otro tipo de sobrecarga). Justificar respuesta.

Solución

(a) Programa secuencial que determina si x es o no primo.

Se van a implementar dos versiones, la segunda versión tiene un menor orden de complejidad.

Versión 1

Pre-condición

x : número de entrada $\{2, \dots, \text{MAX_INPUT}\}$. MAX_INPUT : máximo valor de la entrada

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

NP : vector con M componentes (los M nº primos desde 2 hasta MAX_INPUT estarán ubicados desde $\text{NP}[0]$ hasta $\text{NP}[M-1]$)

Pos-condición: Imprime en pantalla si el número de entrada x es primo o no

Código

Versión 1

```
if (x>NP[M-1]) {
    print("%u supera el máximo primo a detectar %u \n", x, NP[M-1]);
    exit(1);
}
for (i=0; x>NP[i]; i++) {};
if (x==NP[i]) printf("%u ES primo\n", x);
else printf("%u NO es primo\n", x);
```

Orden de complejidad:

El número de iteraciones del bucle depende de en qué posición se encuentre el número primo x en NP . Teniendo en cuenta el teorema de los números primos:

$$\lim_{x \rightarrow \infty} \left(\frac{\pi(x)}{x / \ln x} \right)^n = 1 \quad (\pi(x) \text{ es el número de primos menores o iguales que } x)$$

el número de iteraciones y , por tanto, el orden de complejidad se podría aproximar por $O(x/\ln x)$.

Versión 2

Para implementar esta versión se ha tenido en cuenta que, si un número x no es primo, alguno de sus factores (sin contar el 1) debe ser menor o igual que \sqrt{x} . Entonces,

para encontrar si x es primo hay que dividir entre $2, 3, \dots, [\sqrt{x}]$. En realidad, bastaría hacer las divisiones entre los números primos menores o iguales que $[\sqrt{x}]$. Por ejemplo: para probar si 227 es primo sabiendo que $\sqrt{227} = 15'0665\dots$ basta con ver si es divisible entre $2, 3, 5, 7, 11$ y 13 (en este caso no lo es, 227 es primo).

Pre-condición

x: número de entrada $\{2, \dots, \text{MAX_INPUT}\}$, MAX_INPUT: máximo valor de la entrada
M: número de primos entre 2 y $[\sqrt{\text{MAX_INPUT}}]$ (ambos incluidos)
NP: vector con los M nº primos desde 2 hasta $[\sqrt{\text{MAX_INPUT}}]$
xr: raíz cuadrada de x

Pos-condición: Imprime en pantalla si el número x es primo o no

Código

Versión 2

```
xr = sqrt(x); //sqrt(x) devuelve la raíz cuadrada de x
if (xr>NP[M-1]) {
    print("La raíz cuadrada de %u supera la del máximo primo a detectar
    (%u) \n", xr, NP[M-1]);
    exit(1);
}
for ( i=0 ; ((NP[i]<xr) && (x % NP[i])) ; i++) {} // % = módulo
if (x % NP[i]) printf("%u ES primo", x);
else printf("%u NO ES primo", x);
```

Orden de complejidad: El peor caso se obtiene cuando x es primo.

- Si el número es primo, el número de iteraciones del bucle depende de en qué posición se encuentra el número primo mayor que $[\sqrt{x}]$ en NP. Teniendo en cuenta el número de primos entre 2 y $[\sqrt{x}]$ según el teorema de los números primos, el orden de complejidad sería de $O([\sqrt{x}]/L[\sqrt{x}])$.
- Si el número no es primo, el bucle recorre el vector hasta que encuentra el primer primo que divide a x. Lo encontrará antes de llegar al número primo mayor que $[\sqrt{x}]$.

(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 flujos (procesos) ejecutan el mismo código. Se escribirá una versión paralela para cada versión secuencial del apartado (a).

Pre-condición

x: número de entrada $\{2, \dots, \text{MAX_INPUT}\}$; se supone que $x \leq \text{MAX_INPUT}$

xr (versión 2): almacena la raíz cuadrada de x

M (versión 1): número de primos entre 2 y MAX_INPUT (ambos incluidos)

M (versión 2): número de primos entre 2 y $\lfloor \sqrt{MAX_INPUT} \rfloor$ (ambos incluidos)

NP (versión 1): vector con M+num_procesos componentes (los M nº primos desde 2 hasta MAX_INPUT estarán ubicados desde NP[0] hasta NP[M-1]; a NP[M] ... NP[M+num_procesos-1] se asignará $x+1$ en el código)

NP (versión 2): vector con M+num_procesos componentes (los M nº primos entre 2 y $\lfloor \sqrt{MAX_INPUT} \rfloor$ estarán ubicados desde NP[0] hasta NP[M-1]; a NP[M] ... NP[M+num_procesos-1] se asignará $xr+1$ en el código)

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

b, baux: variables que podrán tomar dos valores 0 o 1

Pos-condición: Imprime en pantalla si el número x es primo o no

Código

Versión 1	Versión 2
<pre> // x es menor que MAX_INPUT //Difusión de x y NP if (idproc==0) for (i=1; i<num_procesos; i++) { send(NP,M,tipo,i,grupo); send(x,1,tipo,i,grupo); } else { receive(NP,M,tipo,0,grupo); receive(x,1,tipo,0,grupo); } //Cálculo paralelo, asignación estática i=id_proc; NP[M+i]=x-1; while (x<NP[i]) do { i=i+num_procesos; } b=(x==NP[i])?1:0; //Comunicación resultados if (idproc==0) for (i=1; i<num_procesos; i++) { receive(baux,1,tipo,i,grupo); b = b baux; } else send(b,1,tipo,0,grupo); //Proceso 0 imprime resultado if (idproc==0) if (b) printf("%u ES primo", x); else printf("%u NO ES primo", x); </pre>	<pre> // x es menor que MAX_INPUT //Difusión de x y NP if (idproc==0) for (i=1; i<num_procesos; i++) { send(NP,M,tipo,i,grupo); send(x,1,tipo,i,grupo); } else { receive(NP,M,tipo,0,grupo); receive(x,1,tipo,0,grupo); } //Cálculo paralelo, asignación estática i=id_proc; xr = sqrt(x); NP[M+i]=xr+1; while ((NP[i]<=xr)&&(x % NP[i])) do { i=i+num_procesos; } b=(NP[i]>xr)?1:0; //Comunicación resultados if (idproc==0) for (i=1; i<num_procesos; i++) { receive(baux,1,tipo,i,grupo); b = b & baux; } else send(b,1,tipo,0,grupo); // Proceso 0 imprime resultado if (idproc==0) if (b) printf("%u ES primo", x); else printf("%u NO ES primo", x); </pre>

Orden de complejidad Versión 1:

- Teniendo en cuenta el número de primos entre 2 y x , según el teorema de los números primos, y teniendo en cuenta el reparto de las iteraciones del bucle entre los procesos, el orden de complejidad se puede aproximar por $O(x / (\text{num_procesos} \times Lx))$. Se consigue una distribución equilibrada de las tareas a realizar repartiendo las iteraciones en turno rotatorio (*Round-Robin*). Con esta distribución, el proceso 0 ejecuta las iteraciones para i igual a 0, num_procesos , $2 \times \text{num_procesos}$, $3 \times \text{num_procesos}, \dots$; el proceso $idproc$ para i igual a 1, $\text{num_procesos} + idproc$, $2 \times \text{num_procesos} + idproc$, $3 \times \text{num_procesos} + idproc, \dots$; etc. Algunos procesos pueden ejecutar sólo una iteración más que otros.

Orden de complejidad Versión 2:

- Si el número es primo, el número de iteraciones del bucle depende de en qué posición se encuentra el número primo mayor que $\lfloor \sqrt{x} \rfloor$ en NP. Teniendo en cuenta el número de primos entre 2 y $\lfloor \sqrt{x} \rfloor$ y teniendo en cuenta el reparto de las iteraciones del bucle entre los procesos en turno rotatorio, el orden de complejidad se aproxima por $O(\lfloor \sqrt{x} \rfloor / (\text{num_procesos} \times L \lfloor \sqrt{x} \rfloor))$.
- Si el número no es primo, algunos procesos recorren con el bucle el vector hasta que encuentran un primo que divide a x , el resto (no todos van a encontrar un factor de x) lo recorre hasta encontrar el número primo mayor que $\lfloor \sqrt{x} \rfloor$. $O(\lfloor \sqrt{x} \rfloor / (\text{num_procesos} \times L \lfloor \sqrt{x} \rfloor))$.

Problema 12. (a) Escribir una versión paralela del programa secuencial del ejercicio 11 suponiendo que la herramienta de programación ofrece las funciones colectivas de difusión y reducción. En la función de difusión, `broadcast(buffer, count, datatype, idproc, group)`, se especifica:

- `group`: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo reciben.
- `idproc`: identificador del proceso que envía.
- `buffer`: dirección de comienzo en memoria de los datos que difunde `idproc` y que almacenará, en todos los procesos del grupo, los datos difundidos.
- `datatype`: tipo de los datos a enviar/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`.

En la función de reducción, `reduction(sendbuf, recvbuf, count, datatype, oper, idproc, group)`, se especifica:

- `group`: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo envían.

- idproc: identificador del proceso que recibe.
- recvbuf: dirección en memoria a partir de la cual se almacena el escalar resultado de la reducción de todos los componentes de todos los vectores sendbuf.
- sendbuf: dirección en memoria a partir de la cual almacenan todos los procesos del grupo los datos de tipo datatype a reducir (uno o varios).
- datatype: tipo de los datos a enviar y recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- oper: tipo de operación de reducción. Puede tomar los valores OR, AND, ADD, MUL, MIN, MAX
- count: número de datos de tipo datatype, del buffer sendbuffer de cada proceso, que se van a reducir.

(b) ¿Qué estructura de procesos/tareas implementa el código paralelo del apartado anterior? Justifique su respuesta.

Solución

(a) Código paralelo

Pre-condición

x: número de entrada $\{1, \dots, \text{MAX_INPUT}\}$. Se cumple que x es menor o igual que MAX_INPUT

xr: almacena la raíz cuadrada de x.

MAX_INPUT: máximo valor de la entrada

M: número de primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$ (ambos incluidos)

grupo: identificador del grupo de procesos que intervienen en la comunicación

idproc: identificador del proceso

tipo: tipo de los datos a enviar o recibir

num_procesos: número de procesos en el grupo

NP (versión 2): vector con $M + \text{num_procesos}$ componentes (los M nº primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$ estarán ubicados desde $NP[0]$ hasta $NP[M-1]$; a $NP[M] \dots NP[M+\text{num_procesos}-1]$ se asignará $xr+1$ en el código)

b: variable que podrá tomar dos valores 0 o 1

Pos-condición:

Imprime en pantalla si el número x es primo o no

Código:

```

// x no es mayor que MAX_INPUT
//difusión del vector NP y de x
broadcast(NP, M, tipo, 0, grupo);
broadcast(x, 1, tipo, 0, grupo);

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

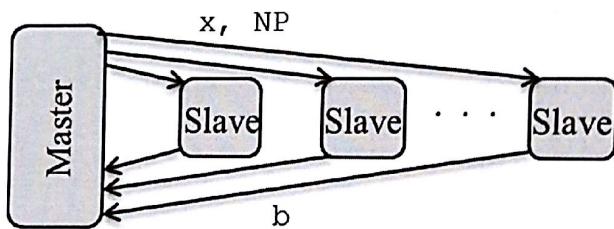
//Comunicación resultados
reduction(b, b, 1, tipo, AND, 0, grupo);

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

```

(b) Estructura de procesos/tareas

Se podría decir que tiene una estructura *Master-Slave* (ver figura más abajo), puesto que el proceso 0 se encarga de distribuir los datos (el trabajo) entre el resto de procesos del grupo y recolectar resultados para, combinándolos, obtener e imprimir el resultado definitivo. Además, se reparte el trabajo haciendo una descomposición de dominio, es decir, dividiendo NP en trozos y asignando el trabajo asociado a cada trozo a un proceso distinto. Los num_procesos esclavos realizan el mismo trabajo.



Problema 13. (a) Escribir una versión paralela del programa secuencial del Problema 11 para un multiprocesador usando el modelo de programación paralela de variables compartidas de OpenMP; en particular usar la directivas de trabajo compartido `for`, `sections/section` y/o `single` para distribuir las tareas entre los *threads* en lugar de realizar una asignación explícita como en los códigos realizados con paso de mensajes. (b) ¿Cuál es el orden de complejidad del código implementado? (considere sólo cálculos, no tenga en cuenta el orden de complejidad de las comunicaciones o de otro tipo de sobrecarga). Razonar respuestas.

Solución

(a) Se van a implementar dos versiones paralelas con OpenMP, teniendo en cuenta las dos versiones secuenciales del Problema 11.

Pre-condición

x : número de entrada $\{2, \dots, \text{MAX_INPUT}\}$. MAX_INPUT : máximo valor de la entrada
 x^2 (versión 2): almacena la raíz cuadrada de x

M (versión 1): número de primos entre 2 y MAX_INPUT (ambos incluidos)

M (versión 2): número de primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$ (ambos incluidos)

NP (versión 1): vector con los M nº primos desde 2 hasta MAX_INPUT

NP (versión 2): vector con los M nº primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$

b : variable que podrá tomar dos valores 0 o 1

Pos-condición: Imprime en pantalla si el número x es primo o no

Para poder usar la directiva `for` en un bucle debe ajustarse a la forma que se indica en las especificaciones de OpenMP (la última versión es la 4.5). Esta directiva no se puede aplicar a los bucles usados en los códigos de los ejercicios anteriores que encuentran si un número es o no primo. Los códigos de este ejercicio se han implementado con un bucle `for` al que se puede aplicar la directiva `for`.

Versión 1

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

b=0;
#pragma omp parallel for
for (i=0;i<M;i++) {
    if (NP[i]==x) b=1; //sólo un thread hace b=1
}

if (b) printf("%u ES primo", x);
else printf("%u NO es primo", x);
```

Versión 2

```
xr = sqrt(x); //sqrt(x) devuelve la raíz cuadrada de x
if (xr>NP[M-1]) {
    print("La raíz cuadrada de %u supera la del máximo primo a
detectar (%u) \n", xr, NP[M-1]);
    exit(1);
}

b=1;
#pragma omp parallel for reduction(&&:b)
for (i=0;i<M;i++) {
    b = b && (x % NP[i]);
}

if (b) printf("%u ES primo", x);
else printf("%u NO es primo", x);
```

(b) Versión 1: $O(\text{MAX_INPUT}/(\text{num_threads} \times \text{LMAX_INPUT}))$. Los threads se reparten las M iteraciones del bucle, como M es el número de primos entre 2 y MAX_INPUT , teniendo en cuenta el teorema de los números primos, M se aproxima por $\text{MAX_INPUT}/\text{LMAX_INPUT}$

Versión 2: $O(\lfloor \sqrt{\text{MAX_INPUT}} \rfloor / (\text{num_threads} \times L[\sqrt{\text{MAX_INPUT}}]))$. Los threads se reparten las M iteraciones del bucle, como M es el número de primos entre 2 y $\sqrt{\text{MAX_INPUT}}$, teniendo en cuenta el teorema de los números primos, M se aproxima por $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor / L[\sqrt{\text{MAX_INPUT}}]$

6. Cuestiones

Cuestión 1. Indique las diferencias entre OpenMP y MPI.

Solución

OpenMP es un estándar de facto para la programación paralela con el estilo de variables compartidas mientras MPI es un estándar de facto para la programación con el estilo de paso de mensajes. OpenMP es una API basada en directivas del compilador y funciones, mientras que MPI es una API basada en funciones de biblioteca, que sitúa al programador en menor nivel de abstracción que OpenMP.

Cuestión 2. Ventajas e inconvenientes de una asignación estática de tareas a flujos (procesos/threads) frente a una asignación dinámica.

Solución

- Ventajas de la asignación estática frente a la dinámica:
 - La asignación estática requiere menos instrucciones extra que la dinámica.
 - Elimina la comunicación/sincronización necesaria para asignar tareas a flujos durante la ejecución.
- Inconvenientes de la asignación estática frente a la dinámica:
 - No se puede utilizar en aquellas aplicaciones en las que no hay un momento ni antes ni durante la ejecución en el que se sepa con seguridad el número de tareas en total que se deben ejecutar; las tareas a realizar van apareciendo en tiempo de ejecución y en distinto momento; es decir, no aparecen todas a la vez.
 - Difícil conseguir un equilibrado de la carga cuando la plataforma (hardware/software) es heterogénea (i.e. el tiempo de cálculo de los núcleos/procesadores no es igual) y/o no uniforme (i.e. el tiempo de comunicación depende de los nodos que se quieren comunicar, no es igual para todos los nodos de la plataforma)

- Difícil conseguir un equilibrado de la carga cuando las tareas a repartir entre flujos requieren distinto tiempo y, más aún, si no se sabe cuánto va a ser ese tiempo.

Cuestión 3. ¿Qué se entiende por escalabilidad lineal y por escalabilidad superlineal (Figura 5)? Indique las causas por las que se puede obtener una escalabilidad superlineal.

Solución

Para obtener una escalabilidad lineal se debe obtener una ganancia en prestaciones conforme se añaden recursos (por ejemplo, procesadores) igual al número de recursos utilizados. Se llama lineal porque la gráfica es una línea recta. Representa la escalabilidad ideal que se esperaría conseguir. Pero en algunos códigos paralelos se observa ganancias por encima de la lineal. Cuando esto ocurre se dice que la escalabilidad es superlineal.

La escalabilidad superlineal se puede deber al hardware y/o al código que se ejecuta. Al añadir un nuevo recurso (por ejemplo, un chip de procesamiento) realmente en la práctica se añaden varios recursos de distinto tipo (por ejemplo, en el caso de añadir un nuevo chip de procesamiento se añade, además de nuevos núcleos de procesamiento, más caché), si la aplicación se beneficia de más de uno de los tipos de recursos añadidos la ganancia puede resultar por encima de la lineal. También se puede explicar por la aplicación que se ejecuta; por ejemplo, hay aplicaciones que consisten en explorar una serie de posibilidades para encontrar una solución al problema. La exploración en paralelo puede hacer que se llegue antes a comprobar la posibilidad que lleva a una Solución

Cuestión 4. Enuncie la ley de Amdahl en el contexto de procesamiento paralelo.

Solución

La ganancia en prestaciones que se puede conseguir al añadir procesadores está limitada por la fracción del tiempo de ejecución secuencial que supone la parte del código no paralelizable.

Cuestión 5. Deduzca la expresión matemática que se suele utilizar para caracterizar la ganancia escalable. Defina claramente y sin ambigüedad el punto de partida que va a utilizar para deducir esta expresión y cada una de las etiquetas que utilice.

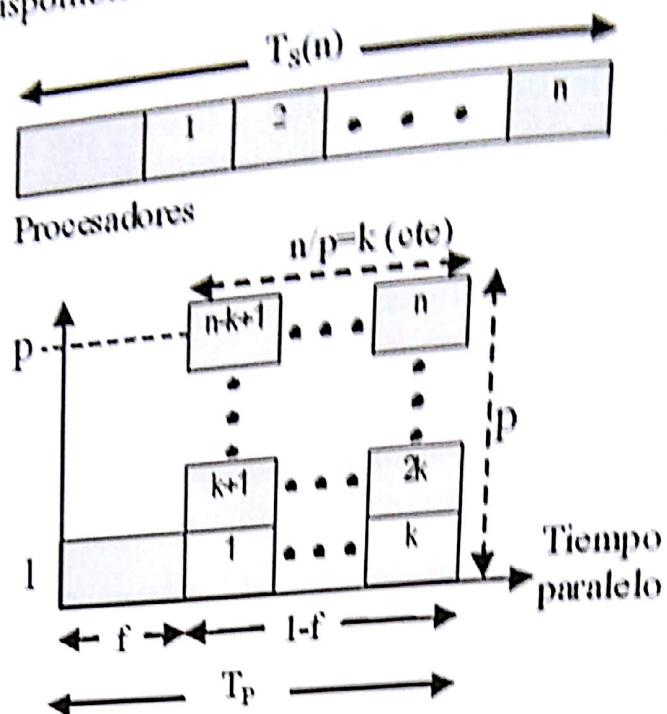
Solución

Punto de partida:

Se parte de un modelo de código en el que el tiempo de ejecución secuencial no permanece constante al variar el número de procesadores, lo que permanece constante es el tiempo de ejecución paralelo y en el que hay (como se representa en la figura de abajo):

- Un trozo no paralelizable (la fracción notada por f en la figura)

- Otro trozo (el resto) que se puede parallelizar repartiéndolo por igual entre los procesadores disponibles.



Las etiquetas de la figura se describen a continuación:

- T_p es una constante que representa el tiempo de ejecución paralelo que permanece constante conforme varía p ya que k es constante
- k es una constante igual a al grado de paralelismo del código secuencial dividido entre el número de procesadores (n/p). Cuando aumenta p aumenta también n para mantener constante k y T_p
- $T_s(n=kp)$ es el tiempo de ejecución secuencial que varía conforme varía p
- f es la fracción del tiempo de ejecución paralelo del código que supone la parte no paralelizable
- $(1-f)$ es la fracción del tiempo de ejecución paralelo del código que supone la ejecución de la parte paralelizable
- p representa el número de procesadores disponibles

Deducción:

La ganancia en prestaciones para este modelo de código ideal sería (se supone 0 el tiempo de sobrecarga):

$$S(p) = \frac{T_s(n=kp)}{T_p} = \frac{f \times T_p + (1-f) \times T_p \times p}{T_p} = f + (1-f) \times p$$

Según esta expresión la ganancia crece de forma lineal conforme varía p con una pendiente constante de $(1-f)$.

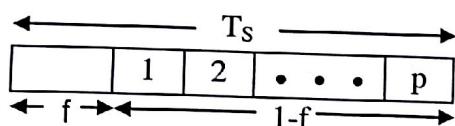
Cuestión 6. Deduzca la expresión que caracteriza a la ley de Amdahl. Defina claramente el punto de partida y todas las etiquetas que utilice.

Solución

Punto de partida:

Se parte de un modelo de código secuencial (como se representa en la figura de abajo) en el que se verifica lo siguiente:

- Tiene una parte no paralelizable (la fracción notada por f en la figura) que permanece constante aunque varíe el número de procesadores p disponibles. El resto se puede parallelizar con un grado de paralelismo ilimitado y, además, repartiéndolo por igual entre los p procesadores disponibles.
- El tiempo de ejecución secuencial permanece constante aunque varíe el número de procesadores disponibles p



Las etiquetas de la figura se describen a continuación:

- T_s constante que representa el tiempo de ejecución secuencial (es independiente de p)
- f constante que representa la fracción del tiempo de ejecución secuencial del código que supone la parte no paralelizable
- $(1-f)$ es la fracción del tiempo de ejecución secuencial del código que supone la ejecución de la parte paralelizable
- p variable que representa el número de procesadores disponibles

Deducción:

La ganancia en prestaciones para este modelo de código ideal sería (suponiendo 0 el tiempo de sobrecarga):

$$S(p) = \frac{T_s}{T_p(p)} = \frac{T_s}{f \times T_s + \frac{(1-f) \times T_s}{p}} = \frac{1}{f + \frac{1-f}{p}} = \frac{p}{fp + (1-f)} = \frac{p}{1 + f(p-1)}$$

En la práctica la ganancia en prestaciones será menor debido a que (1) el grado de paralelismo estará limitado, (2) puede ser difícil equilibrar la carga de trabajo y (3) la parallelización puede añadir un tiempo de sobrecarga (*overhead*) debido a la necesidad de comunicación/sincronización y a las operaciones extras que puede requerir la parallelización. Es decir:

$$S(p) \leq \frac{p}{1 + f(p-1)}$$