EXAMEN DE CONCURRENCIA Y PARALELISMO

Bloque II: Paralelismo

APELLIDOS: _____ **NOMBRE:**_____

Convocatoria ordinaria
24 de mayo de 2018

AVISO: *No mezcleis en el mismo folio preguntas del bloque de concurrencia y del bloque de paralelismo. Las respuestas a cada bloque se entregan por separado.*

[2,5p] 1. **Conceptos de paralelismo.** Una aplicación de análisis de datos en tiempo real recibe a través de un conjunto de sensores un flujo de datos constante, y los procesa en un **pipeline** para imprimir periódicamente un valor S que resume los valores registrados por los sensores. La aplicación funciona cíclicamente del siguiente modo: Al comenzar un ciclo, se leen los datos recibidos de los sensores hasta completar un bloque que almacenamos en una matriz M (etapa 1). Ese bloque se procesa en una función A, que calcula un índice numérico I (etapa 2). En función del valor de I, se decide si se ejecuta la función $B1$ o $B2$, las cuales a partir de M generan una segunda matriz $M2$ (etapa 3). Finalmente, con la matriz $M2$ se calcula el valor de S (etapa 4). El siguiente pseudocódigo describe el algoritmo del pipeline, en el que se indica también la duración del procesamiento secuencial de cada una de las funciones.

```
while(true) {
  M = captura_datos()    // 1) lectura de datos: 50 ms
  I = funcion_A(M)       // 2) funcion A: 50 ms
  if (I >= 0)
    M2 = funcion_B1(M)   // 3.1) funcion B1: 200 ms
  else
    M2 = funcion_B2(M)   // 3.2) funcion B2: 100 ms
  S = funcion_C(M2)      // 4) funcion C: 50 ms
}
```

Las funciones $B1$ y $B2$ consisten en la aplicación de una serie de operaciones sobre la matriz M. $B1$ es paralelizable con una eficiencia paralela constante del 80% para cualquier número de procesos $P > 1$. $B2$ tiene una parte estrictamente secuencial que tarda 40ms, mientras que el resto es paralelizable con una eficiencia del 100%. El resto de las funciones no son paralelizables. El tiempo de comunicaciones es despreciable. Nuestro objetivo es idear una paralelización del pipeline en el que cada etapa tarde como máximo 50 ms, de modo que se elimine el tiempo de espera entre cada ejecución de la etapa 1. Responde brevemente a las siguientes cuestiones, justificando tu respuesta.

(a) Dibuja el grafo de dependencias para este problema. Diferencia las dependencias de datos y de control. **(0,5 p.)**

(b) ¿Qué tipo de descomposición o descomposiciones aplicarías al problema? **(0,5 p.)**

(c) Si queremos comprar una máquina dedicada en exclusiva a ejecutar este algoritmo, calcula cuántos procesadores debería tener para conseguir el objetivo propuesto y describe la asignación entre tareas y procesadores. **(1 p.)**

(d) Describe en una línea temporal el funcionamiento del pipeline para el procesamiento de 5 bloques de datos (5 ciclos completos de ejecución). **(0,5 p.)**

[2,5p] 2. **Diseño de algoritmos paralelos.** Debido a su elevado costo en la mayoría de las ejecuciones, se desea paralelizar el algoritmo

```
float f(int r) {
  if (condition(r)) return compute(r);
  else return f(i1(r)) + f(i2(r));
}

main () {
  int input = read_from_file();
  float result = f(input);
  write_to_file(result);
}
```

en donde todas las funciones invocadas dentro de `f` proceden de librerías externas cuya implementación desconocemos. Lo que sí sabemos es que el tiempo de computación de `condition`, `compute`, `i1` e `i2` es fijo independientemente del `r` utilizado y que el coste computacional de `f(i1(r))` y de `f(i2(r))` son idénticos para un `r` dado, con lo que su carga computacional está totalmente equilibrada. Además se sabe que el número de procesos P usado en las ejecuciones siempre será una potencia de 2 y el proceso 0 será el único que puede acceder al sistema de ficheros.

(a) Explica qué tipo de descomposición se ajusta a este algoritmo, justificando la respuesta. **(0,5 p.)**

(b) Explica qué tipo asignación de tareas usarías, justificando la respuesta y explicando a alto nivel cómo funcionaría tu propuesta. **(0,5 p.)**

(c) Explica qué tipo asignación de tareas usarías en el caso de que en general la carga computacional de `f(i1(r))` y de `f(i2(r))` pudiese ser muy distinta, justificando la respuesta y explicando a alto nivel cómo funcionaría tu propuesta. **(0,5 p.)**

(d) Implementa el algoritmo paralelo en MPI siguiendo este pseudocódigo **(1 p.)**

i. El proceso 0 lee la entrada de disco.

ii. El proceso 0 descompone localmente (sin comunicarse con otros procesos) el trabajo hasta obtener tareas para todos los procesos. Puedes asumir que siempre se podrá generar trabajo para todos los procesos.

iii. El proceso 0 envía trabajo a todos los procesos, incluyendo él mismo, mediante una colectiva de MPI.

iv. Cada proceso ejecuta su trabajo local.

v. Obtener el resultado global de la computación en el proceso 0 mediante otra colectiva de MPI.

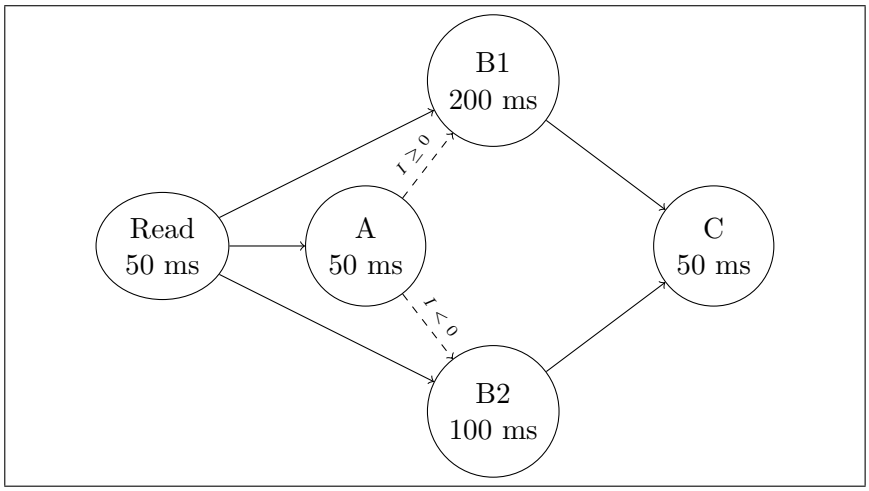
vi. El proceso 0 guarda el resultado en el disco.

```
int MPI_Init(int *argc, char ***argv)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Barrier(MPI_Comm comm)
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
int MPI_Scatterv(const void *sendbuf, const int *sendcounts, const int *displs, MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount,
               MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int *recvcounts,
                const int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

SOLUCIÓN

[2,5p] 1. **Conceptos de paralelismo.**

(a) [0,5p] El grafo quedaría de la siguiente forma:



(b) [0,5p] En primer lugar tenemos una descomposición funcional. La ejecución en paralelo de B1 y B2, dado que no tienen una dependencia real de datos con la función A, podemos considerarlo una descomposición especulativa, aunque por la organización del sistema paralelo como un pipeline no sería necesario, ya que conoceremos el resultado de la ejecución de A antes de ejecutar B1 o B2 sobre esos mismos datos. Por último, para B1 y B2 aplicamos una descomposición de dominio.

(c) [1p] Podemos definir un pipeline de 4 etapas: (1) captura de datos, (2) función A, (3) ejecución de B1 o B2 en función del resultado anterior, y (4) función C. Idealmente cada etapa debe tardar el mismo tiempo (50ms), determinado por lo que tardan las etapas no paralelizables. Por lo tanto, la aceleración máxima de B1 y B2 con las que podremos obtener una mejora en el tiempo de ejecución será 4 y 2 respectivamente, para que ambas tengan también una duración de 50 ms.

En B1, como la eficiencia paralela es del 80% para $P > 1$ idependientemente del valor de P, y la eficiencia paralela es la aceleración dividido por el número de procesos ($E = S/P$) tenemos que $P = S/E$. Entonces, para B1 el número de procesos que necesitamos para conseguir una aceleración de $200ms/50ms = 4$ es:

$$P_{B1} \geq 4/0.8 \geq 5$$

Para B2 con los datos que tenemos podemos aplicar Amdahl para conseguir una aceleración de $100ms/50ms = 2$:

$$\frac{1}{0.4 + 0.6/P_{B2}} \geq 2 \rightarrow P_{B2} \geq \frac{0.6}{1/2 - 0.4} \geq 6$$

Por tanto, para ejecutar B1 o B2 en 50ms o menos necesitamos 6 procesadores.

Dado que el algoritmo lo podemos ejecutar como un pipeline, usaremos 3 procesadores para ejecutar la captura de datos, A y C (uno para cada función) y 6 para ejecutar B1 o B2. En total, lo ideal sería que la máquina dedicada tuviese al menos 9 procesadores.

(d) [0,5p] Cada bloque de datos procesado en el pipeline lo denominamos D_i. Con la descomposición y asignación propuesta, la ejecución de cada una de las tareas tarda como máximo 50 ms, y por lo tanto se pueden capturar todos los datos en tiempo real sin que se pierda ninguna lectura de los sensores.

t (ms)	**captura**	**A**	**B1/B2**	**C**
0	D_0			
50	D_1	D_0		
100	D_2	D_1	D_0	
150	D_3	D_2	D_1	D_0
200	D_4	D_3	D_2	D_1
250		D_4	D_3	D_2
300			D_4	D_3
350				D_4

[2,5p] 2. **Diseño de algoritmos paralelos.**

(a) [0,5p] En esta función el paralelismo procede del hecho de que cada computación para un `r` disinto de 0 que no cumpla `condition` puede subdividirse en dos partes que pueden calcularse en paralelo, tratándose además de un proceso recursivo. Así pues, es un caso de descomposición recursiva o divide-y-vencerás.

(b) [0,5p] El enunciado indica que siempre que se puede descomponer un cómputo de `f`, `f(i1(r))` y `f(i2(r))` tienen el mismo coste, con lo que el trabajo siempre está equilibrado a los distintos niveles de recursividad. Además, como el como el coste de las funciones usadas dentro de `f` es fijo, esto indica que si el coste de `f(i1(r))` y `f(i2(r))` es el mismo, entonces el número de subdivisiones o niveles de recursividad de las dos también es el mismo. Por todo ello la carga puede equilibrarse fácilmente de forma estática.

En cuanto a la propuesta de implementación, una estrategia lógica sería por ejemplo ir dividiendo el trabajo en el procesador 0 hasta obtener una tarea para cada procesador disponible, o al menos para el máximo de procesadores que se puede en el caso de que la descomposición recursiva finalice antes de llegar a ese punto. Entonces repartirlos entre todos los procesadores, tras lo cual cada procesador haría su parte, y finalmente se sumarían los resultados parciales de cada procesador.

(c) [0,5p] Al poder haber mucho desequilibrio de carga entre las distintas subtareas y no poder disponer por adelantado del coste concreto de cada tarea, tendremos que recurrir a un reparto de tareas dinámico.

Una propuesta posible sería usar un esquema maestro-esclavo en el que el proceso 0 hace una descomposición inicial entre los demás procesos y éstos se van quedando con algunos de los subproblemas que generan, mientras que los sobrantes se los mandan al 0, al cual le pedirán trabajo cuando se queden sin él. También le enviarían al proceso 0 los resultados de sus computaciones para que los fuese sumando de cara a obtener el resultado final.

(d) [1p] Una propuesta de implementación que cumple con el pseudocódigo proporcionado es

```
int P, myrank, tmp, offset, *inputs;
float result, rtmp;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &P);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if(myrank == 0) {
  inputs  = (int *)malloc(sizeof(int) * P);

  inputs[0] = read_from_file();
  for(offset = 1; (offset < P); offset = offset * 2) {
    for(i = 0; i < offset; i++) {
      tmp = inputs[i];
      inputs[i] = i1(tmp);
      inputs[i + offset] = i2(tmp);
    }
  }
}

MPI_Scatter(inputs, 1, MPI_INT, &tmp, 1, MPI_INT, 0, MPI_COMM_WORLD);
rtmp = f(tmp);
MPI_Reduce(&rtmp, &result, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

if(myrank == 0) {
  write_to_file(result);
}
MPI_Finalize();
```