EXAMEN DE CONCURRENCIA Y PARALELISMO

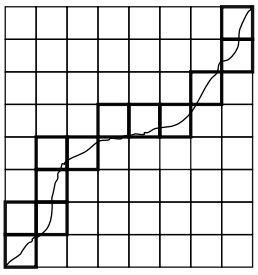
Bloque II: Paralelismo

APELLIDOS: _____ **NOMBRE:** _____

Convocatoria ordinaria

8 de junio de 2021

[**2.5p**] 1. **Conceptos de paralelismo.** A lo largo de una frontera se han instalado cámaras y sensores a fin de detectar posibles intrusiones. La información recogida por estos elementos se mapea sobre una matriz en la que cada celda representa una porción de terreno. En cada celda vigilada se ejecuta un algoritmo que determina si puede haber una intrusión en ella a partir de los datos recogidos por estos elementos. La figura adjunta da una idea abstracta del problema, en donde la frontera es la línea irregular y las celdas que requieren computación tienen un borde grueso.



```
while (1) {
  para cada celda de interés (i,j):
     datos_celda(i,j) = recabar_datos(i,j);
     estimación(i,j) = computación(datos_celda(i,j));
  estimación_global = reducción(estimación);
  si intrusión(estimación_global) :
    emitir_alarma();
}
```

El algoritmo secuencial consiste en el pseudocódigo de arriba, donde `datos_celda` y `estimación` son matrices de $M \times N$ elementos y `estimación_global` es un escalar, mientras que `recabar_datos` recibe los datos de la celda y `computación` los procesa. El coste por celda vigilada de la primera función son 120 ms y el de la segunda 180 ms. La función `reducción` reduce a un escalar la información de si ha habido intrusión en alguna celda y su coste computacional es prácticamente cero, lo cual también ocurre con las funciones `intrusión` y `emitir_alarma`. El sistema no permite solapar comunicaciones y computaciones en cada procesador y necesita $4 + 1 \times n$ ms para comunicar n escalares entre dos procesadores.

Responde las siguientes cuestiones razonando siempre tu respuesta:

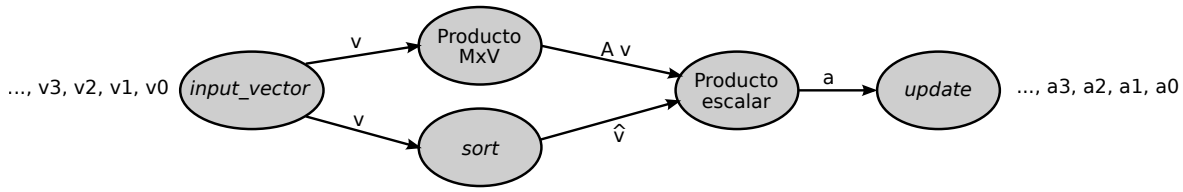
(a) Si la computación de cada celda es independiente, ¿Qué tipo de descomposición y de asignación de tareas aplicarías en este problema? [0,5p]

(b) Si al paralelizar el problema sobre 16 procesos el número mínimo, medio y máximo de celdas vigiladas por proceso es 100, 120,140, respectivamente ¿Cuál es el tiempo de ejecución paralelo de una iteración del bucle, el speedup y la eficiencia? [1p]

(c) Supongamos que se modificase el algoritmo de forma que el cómputo asociado a cada celda vigilada requiriese de 3 escalares obtenidos de cada una de sus 4 celdas vecinas, vigiladas o no, y que paralelizásemos el algoritmo por bloques de filas tras conseguir una media muy estable de 9 celdas vigiladas por fila de la matriz. Si la matriz fuese de 160×160 y usásemos 16 procesadores, ¿cuál sería el tiempo de ejecución de una iteración si los demás parámetros del problema fuesen los mismos? [0,5p]

(d) Si descomponemos la matriz de 160×160 sobre una malla de 4×7 procesadores de forma no cíclica, ¿A qué proceso sería asignado el elemento (100, 100) y cuál sería su posición relativa dentro de dicho proceso? [0,5p]

[2,5p] 2. **Diseño de algoritmos paralelos.** Un sistema dedicado a la monitorización en tiempo real dispone de una sonda que es capaz de proporcionar un bloque de datos en 50 ms, representados en un vector ν de n elementos. El procesamiento de cada bloque consiste en calcular un valor r como el producto escalar entre el resultado de multiplicar una matriz cuadrada A de $n \times n$ elementos por el vector ν, y el propio vector ordenado $\hat{\nu}$. Finalmente una función *update* actualiza el estado del sistema. El siguiente grafo de dependencias ilustra el proceso:



El programa que rige el sistema de monitorización utiliza el siguiente algoritmo, en el que se indica el tiempo que requiere cada una de las secciones:

```
1   double A[N][N], v[N], v_sort[N], Av[N], a;
2
3   input_matrix(A,N,N); // lee la matriz A (15 ms)
4
5   while(1)
6   {
7     input_vector(v,N); // accede a la sonda (50 ms)
8
9     for (i=0; i<N; i++) // producto matriz-vector (50 ms)
10    {
11      Av[i] = 0.0;
12      for (j=0; j<N; j++)
13        Av[i] += A[i][j]*v[j];
14    }
15
16    sort(v, v_sort); // ordena el vector (20 ms)
17
18    a = 0.0;
19    for (i=0; i<N; i++) // producto escalar (30 ms)
20      a += Av[i] * vsort[i];
21
22    update(a); // actualizacion de estado (10 ms)
23  }
```

La sonda produce datos constantemente, y en el momento en que se realiza la lectura se recogen los últimos datos obtenidos. Esta implementación secuencial provoca la pérdida de gran cantidad de información de la sonda, ya que se emplea demasiado tiempo en el procesamiento de cada vector, y desde que se termina la lectura de la sonda hasta que se lee nuevamente se descartan los datos producidos por la sonda.

Queremos realizar una implementación paralela de este código que minimice el tiempo de espera entre lecturas de la sonda. Disponemos para ello de un procesador de 4 núcleos de computación. Las funciones *input_matrix*, *input_vector*, *sort* y *update* no pueden ser paralelizadas, pero cualquier proceso puede realizar cualquiera de ellas. El coste de las comunicaciones en este sistema es despreciable.

Contesta razonadamente a las siguientes cuestiones:

(a) Indica el tipo de descomposición que debemos aplicar al problema, y la asignación de tareas a los procesos. [0,5p]

(b) Para la descomposición y asignación del apartado anterior, determina la latencia de procesamiento de un bloque de datos y la aceleración tras la ejecución de 100 iteraciones del bucle *while*. [0,5p]

(c) Realiza una implementación paralela del algoritmo. A continuación tienes las firmas de las funciones MPI que puedes utilizar. [1,5p]

```
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_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_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, 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

1. (a) Se trata de una computación sobre una matriz en la que el peso de las computaciones se distribuye irregularmente. Una descomposición de dominio por filas, columnas o bloques puede provocar fuertes desequilibrios de carga entre los procesadores. Por otra parte, una distribución cíclica por bloques no es la mejor porque las celdas vigiladas suelen ser adyacentes entre sí al ir siguiendo la frontera, favoreciendo la creación de bloques pesados frente a otros sin cómputos asociados. El hecho de que no se requieran comunicaciones entre las celdas para la computación también apunta en la dirección de hacer una distribución cíclica pura, puesto que ésta estaría penalizada si se necesitasen comunicaciones con las celdas vecinas.

   La mejor opción en esta situación no obstante sería hacer un vector de estructuras con las V celdas vigiladas y sus índices i y j, ya que son usados por la función `recabar_datos`, y dividir este vector por bloques o cíclicamente entre los procesadores, pues esto garantizaría el máximo equilibrio. En el reparto por bloques entre P procesadores cada uno obtendría $\lceil V/P \rceil$ celdas, excepto el último, que obtendría $V - \lceil V/P \rceil \times P$.

   Por otra parte, el hecho de que cada celda vigilada requiera el mismo tiempo de procesado y siempre tengamos el mismo número de celda vigiladas, unidos a los costes de comunicaciones, nos indica que un reparto dinámico no merece la pena. Así pues realizaría una asignación estática.

   (b) El rendimiento estará limitado por los procesadores con más carga, que en cada iteración del bucle principal requerirán $140 \times (120 + 180)$ms $= 42$ segundos para obtener los datos de las celdas y procesarlos. Para ser preciso, habría que añadir el tiempo de la reducción, que si se usa un árbol binario requeriría $\log_2(16) = 4$ pasos de 5 ms cada uno ya que reducimos un escalar, con lo que el tiempo total ascendería a 42.02 s. si suponemos que al menos uno de los procesadores más cargados está en una de las hojas del árbol binario de reducción. Si por el contrario el procesador más cargado fuese sólo el de la raíz de la reducción, los mensajes de reducción se solaparían con sus cómputos y el tiempo quedaría en 42 s.

   Si la media de celdas vigiladas por proceso son 120 y hay 16 procesadores, el total de celdas de este tipo son $120 \times 16 = 1920$, con lo que su tiempo de procesamiento secuencial debe ser $1920 \times (120 + 180)$ms$= 576$ s. Por tanto el speedup es aproximadamente $576/42 = 13.71$

   La eficiencia se calcularía como $13.71/16$ procesadores $= 0.857$.

   (c) Cada procesador procesaría 160/16=10 filas de 9 celdas. Como cada celda sigue requiriendo $120 + 180$ ms para ser procesada en las funciones, eso requeriría $10 \times 9 \times (120 + 180)$ms $=$ 27s. Luego, la reducción entre los 16 procesadores seguiría requiriendo 20 ms, totalizando 27.02 s.

   Ahora bien, ahora hay que añadir comunicaciones verticales de 3 escalares por cada celda adyacente a una celda vigilada. Esto podría hacerse de tres formas. Una sería enviar y recibir los 3 escalares para los 160 elementos de cada fila, lo que requeriría $4 + 160 \times 3$ ms $= 484$ ms por mensaje. Como se enviaría/recibiría uno al norte y otro al sur, serían en total $484 \times 2 = 968$ ms. La segunda forma sería primero enviar un mensaje con los números de columna de las celdas vigiladas, lo cual al ser 9 por fila de media, requeriría $4 + 9 = 13$ms, y luego recibir los 3 escalares de esas 9 celdas de interés, lo que requeriría $4 + 9 \times 3 = 31$ms, totalizando entre las dos direcciones $2 \times (13 + 31) = 88$ms. Como optimización, el primer mensaje con los número de columnas podría hacerse una sola vez al principio del algoritmo, y luego en cada iteración sólo había que mandar el segundo, requiriéndose así sólo $2 \times 31 = 61$ ms en esta tercera estrategia.

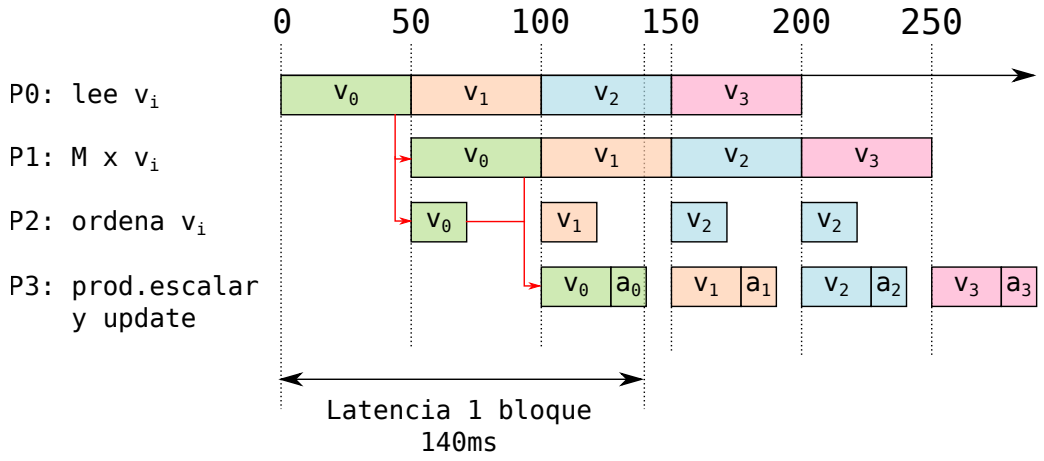
   Así, el tiempo total estaría entre $27.02 + 0.968 = 27.988$ s., $27.02 + 0.088 = 27.108$ s. o incluso $27.02 + 0.061 = 27.081$ s.

(d) En una descomposición no cíclica de una matriz $M \times N$ sobre una malla de procesadores $P \times Q$ cada procesador recibe un bloque de $\lceil M/P \rceil \times \lceil N/Q \rceil$ excepto los de la última fila y columna. En este caso serían bloques de $(160/4 = 40) \times (\lceil 160/7 \rceil = 23)$. Para determinar el procesador que posee la celda (i, j) haríamos pues el cálculo $(\lfloor i/40 \rfloor, \lfloor j/23 \rfloor)$, lo que para $(100, 100)$ da $(2, 4)$.

La posición relativa dentro de un procesador se obtiene haciendo la operación de módulo del índice de cada dimensión con respecto al tamaño del bloque en dicha dimensión. Así pues, en este caso el valor sería $(100 \bmod 40 = 20, 100 \bmod 23 = 8)$.

2. (a) La mejor opción es aplicar una descomposición funcional para organizar el funcionamiento del sistema como un pipeline. Una posible asignación de tareas es la siguiente: P0: input_vector (50 ms), P1: Producto matriz-vector (50 ms), P2: sort (20 ms), P3: producto escalar + update (40 ms). De esta forma cada etapa del pipeline tendrá una duración máxima de 50 ms.

(b) La latencia está determinada por el camino crítico del grafo de dependencias, y serían $50+50+30+10 = 140$ ms. 100 bloques de datos se procesan secuencialmente en $160 \times 100 = 16000$ ms, mientras que organizando la ejecución como un pipeline se tardará $50 \times 99 + 140 = 5090$ ms. La aceleración es por tanto $16000/5090 = 3.14$.



(c) Para la implementación paralela establecemos una dependencia entre procesos de la siguiente forma: $P0 \rightarrow \{P1, P2\} \rightarrow P3$, donde las flechas indican las comunicaciones entre procesos, de modo que P0 se encarga de leer la sonda y enviar los datos a P1 y P2, estos 2 procesos ejecutan sus respectivas tareas y envían el resultado a P3, que realiza el producto escalar y actualiza el estado del sistema. Las comunicaciones bloqueantes mantienen el correcto funcionamiento del pipeline.

```
1   double A[N][N], v[N], v_sort[N], Av[N], a;
2
3   // solo el proceso 1 necesita la matriz A
4   if (mpi_rank == 1)
5     input_matrix(A,N,N);
6
7   while(1)
8   {
9     if (mpi_rank == 0)
10    {
11      input_vector(v,N);
12
13      MPI_Send(v, N, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
14      MPI_Send(v, N, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);
15    }
16    else if (mpi_rank == 1)
17    {
18      MPI_Recv(v, N, MPI_DOUBLE, 0,
19               0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
20      for (i=0; i<N; i++)
21      {
22        Av[i] = 0.0;
23        for (j=0; j<N; j++)
24          Av[i] += A[i][j]*v[j];
25      }
26      MPI_Send(Av, N, MPI_DOUBLE, 3, 0, MPI_COMM_WORLD);
27    }
28    else if (mpi_rank == 2)
29    {
30      MPI_Recv(v, N, MPI_DOUBLE, 0,
31               0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
32      sort(v, v_sort);
33      MPI_Send(v_sort, N, MPI_DOUBLE, 3, 1, MPI_COMM_WORLD);
34    }
35    else if (mpi_rank == 3)
36    {
37      MPI_Recv(Av, N, MPI_DOUBLE, 1,
38               0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
39      MPI_Recv(v_sort, N, MPI_DOUBLE, 2,
40               1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
41
42      a = 0.0;
43      for (i=0; i<N; i++)
44        a += Av[i] * vsort[i];
45      update(a);
46    }
47  }
```