EXAMEN DE CONCURRENCIA Y PARALELISMO

Bloque II: Paralelismo

APELLIDOS: _____ **NOMBRE:** _____

Convocatoria extraordinaria

10 de julio de 2020

IMPORTANTE: Al comienzo del documento de respuesta, cada alumno debe indicar su nombre completo y número del documento de identidad o pasaporte.

De cara a contestar las preguntas que figuran a continuación, los datos concretos serán diferentes para cada alumno, dependiendo de su número de DNI (pasaporte para alumnos sin nacionalidad española). Considera tu DNI como $C_1C_2C_3C_4C_5C_6C_7C_8L$ (en caso de alumnos sin nacionalidad española, $C_1C_2C_3C_4C_5C_6C_7$ son las siete cifras que aparecen en el número de pasaporte).

[1p] 1. **Conceptos de paralelismo.** Disponemos de un algoritmo que tiene como estado inicial una matriz S bidimensional de M filas y N columnas de tipo entero. Iterativamente, el algoritmo genera nuevos estados a partir del propio contenido de la matriz. El estado siguiente (k+1) para cada una de las celdas de la matriz se calcula mediante una función f que tiene como parámetros el valor de esa celda y las celdas colindantes del estado actual (k), según la siguiente fórmula:

$$S_{i,j}^{k+1} = f(S_{i-1,j}^{k}, S_{i,j-1}^{k}, S_{i,j+1}^{k}, S_{i+1,j}^{k})$$

Las posiciones con índices que coinciden fuera de rango de la matriz se corresponden con las posiciones del extremo opuesto. Por ejemplo, $S_{-1,j}^{k} = S_{M-1,j}^{k}$, y $S_{i,N}^{k} = S_{i,0}^{k}$.

El siguiente pseudocódigo resume el funcionamiento del algoritmo implementado, donde $S0$ es el estado actual, S^k, y $S1$ es el estado siguiente, S^{k+1}. El algoritmo itera hasta que no se producen cambios entre dos estados consecutivos. La función f se encapsula dentro de otra función, $evoluciona(S^k, i, j)$, que recibe como parámetros el estado actual y los índices de la posición de la matriz que queremos calcular.

```
1  S1 = estado_inicial
2  S0 = estado_vacio
3
4  while (S1 != S0)
5  {
6    S0 = S1
7    for (i = 0..M-1)
8      for (j = 0..N-1)
9        S1[i,j] = evoluciona(S0, i, j)
10 }
```

Queremos paralelizar el algoritmo para ejecutarlo en un sistema dedicado en exclusiva a resolver este problema, que cuenta con 16 procesadores. El coste de ejecutar una vez la función *evoluciona* es de T_f ns. Por las características de la red de interconexión, realizar una comunicación entre procesos tiene un coste fijo de 2^{20} ns, y adicionalmente tarda T_c ns por cada elemento de la matriz enviado. Es decir, un envío de k elementos tardará $2^{20} + kT_c$ ns. Las variables M, N, T_f y T_c adoptan los siguientes valores:

- $M = 2^{C_1+C_2+4}$
- $N = 2^{C_3+C_4+4}$
- $T_f = 2^{C_5}$ ns
- $T_c = 2^{C_6}$ ns

Para este ejercicio ignoraremos el coste de las comunicaciones necesarias para repartir el estado inicial entre los procesos.

Responde justificadamente las siguientes preguntas. Una respuesta sin razonamiento correcto no será válida aunque el resultado o conclusión sean correctos.

(a) ¿Qué tipo de descomposición debemos aplicar al problema? ¿Sería eficiente utilizar una estrategia de asignación dinámica centralizada? [0,1p]

(b) Para una asignación estática por bloques, consideramos los 16 procesos estructurados lógicamente en mallas bidimensionales de 16x1, 8x2 y 4x4. Para cada una de estas tres configuraciones, indica cuántas comunicaciones y de cuántos elementos debe efectuar cada proceso en cada iteración del algoritmo. [0,3p]

(c) Teniendo en cuenta los tiempos mencionados en el enunciado (T_c y T_f), ¿qué configuración del apartado anterior es la más adecuada? [0,3p]

(d) Si distribuimos los procesos en malla bidimensional de 4x4, ¿cuál será el speedup esperado? [0,3p]

[1p] 2. **Diseño de algoritmos paralelos.** Hemos paralelizado el algoritmo descrito en el ejercicio anterior para una distribución de tareas estática por **bloques de filas**.

El tamaño de la matriz ($M \times N$) es el indicado en el ejercicio anterior. En esta ocasión, el algoritmo paralelo diseñado asume también un número constante de procesadores P=2^{C_1+2}.

El proceso 0 se encarga de leer el estado inicial y a continuación ese estado inicial se reparte entre todos los procesos. Como M, N y P son valores constantes, podemos calcular estáticamente el tamaño de las matrices locales que corresponden a cada proceso: $M\_local \times N\_local$. M_local y N_local también serán constantes en nuestro código.

El pseudocódigo de la paralelización es el siguiente:

```
... // inicializacion de MPI

if (rango == 0) {
  lee_estado_inicial(S_inicial, M, N)
}
reparte_estado_inicial(S_inicial, S1)

S0 = estado_vacio

while (continua_bucle(S1, S0))
{
  S0 = S1
  intercambia_datos(S0, rango)

  for (i = 1 .. M_local  )
    for (j = 0 .. N_local-1 )
      S1[i,j] = evoluciona(S0, i, j)
}
```

Como es una distribución por bloques de filas, para el estado local de cada proceso reservamos 2 filas adicionales, $M\_local + 2$ filas en total, para recoger las filas de los procesos vecinos. De esa

forma, las filas propias del proceso se localizarán entre los índices 1 y M_local, la fila 0 queda reservada para recibir la última fila del proceso superior, y la fila $M\_local + 1$ se reserva para recibir la primera fila del proceso inferior.

El bucle principal debe reorganizarse para realizar el trabajo local en cada proceso. Además de los índices del bucle for, se necesitan 2 modificaciones. En primer lugar, en la condición de salida del bucle no es suficiente con que cada proceso compruebe de manera independiente si sus estados S1 y S0 son iguales. El bucle debe terminar si y solo si los estados S1 y S0 son iguales en **todos** los procesos. Para ello contamos con una función denominada "**continua_bucle**", que devolverá 0 (false) si las matrices S1 y S0 son iguales en todos los procesos, o 1 (true) en caso contrario. En segundo lugar, en el bucle es necesario establecer comunicaciones entre los procesos vecinos para enviar la primera y última filas, que se colocarán en las filas adicionales reservadas en los estados locales. Esta tarea está contenida dentro de la función "**intercambia_datos**".

(a) Determina los valores de M_local y N_local. [0,1p]

(b) Proporciona una implementación concreta en lenguaje C de la función "intercambia_datos". Dentro de la función puedes hacer uso de las constantes descritas en el enunciado del ejercicio. [0,45p]

(c) Proporciona una implementación concreta en lenguaje C de la función "continua_bucle". Dentro de la función puedes hacer uso de las constantes descritas en el enunciado del ejercicio. [0,45p]

```
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. Ejercicio 1

   (a) Debemos aplicar una descomposición de dominio. El coste fijo de establecer una comunicación es muy alto (2^{20} ns), y en una asignación dinámica centralizada (maestro-esclavo), en cada iteración el proceso maestro debe gestionar las comunicaciones con los procesos esclavos secuencialmente. Como la evaluación de la función *evoluciona* tiene un coste fijo y la carga puede balancearse bien, las comunicaciones generarían un cuello de botella. Por lo tanto, una estrategia de asignación dinámica no sería necesaria ni eficiente.

   (b)16x1 Es una distribución por bloques de filas. Cada proceso mantiene una matriz de $\frac{M}{16} \times N$ elementos. Cada proceso realiza 2 envíos, a los procesos superior e inferior, cada uno de ellos de N elementos.

   8x2 Cada proceso mantiene una matriz de $\frac{M}{8} \times \frac{N}{2}$ elementos. Cada proceso realiza 3 envíos. En vertical debe enviar $\frac{N}{2}$ elementos a los procesos superior e inferior, y en horizontal enviará $\frac{2M}{8}$ elementos al otro proceso de la misma fila. También se puede considerar que en horizontal son necesarios 2 envíos al mismo proceso en lugar de uno solo (4 en total), de $\frac{M}{8}$ elementos cada uno de ellos.

   4x4 Cada proceso mantiene una matriz de $\frac{M}{4} \times \frac{N}{4}$ elementos. Cada proceso realiza 4 envíos, a los procesos colindantes en las 4 direcciones. En vertical debe enviar $\frac{N}{16}$ elementos en cada mensaje a los procesos superior e inferior, y en horizontal enviará $\frac{M}{16}$ elementos en cada mensaje a los procesos a su izquierda y derecha.

   (c) La respuesta a este apartado dependerá de los valores de M, N, T_c. El tiempo de ejecutar la función, T_f, no influye porque la carga computacional de cada proceso es la misma independientemente de la configuración de los procesos. Para cada una de las configuraciones debemos calcular el tiempo empleado en las comunicaciones por iteración (en nanosegundos) de la siguiente forma:

$$T_{comm} = n\_envios \times 2^{20} + n\_elementos \times T_c$$

   Donde n_envios es el número de envíos que hace cada proceso y n_elementos, el número total de elementos que envían. Entonces:

$$T_{comm}(16x1) = 2 \times 2^{20} + 2NT_c$$
$$T_{comm}(8x2) = 3 \times 2^{20} + (\frac{2M}{8} + 2\frac{N}{2})T_c$$
$$T_{comm}(4x4) = 4 \times 2^{20} + (2\frac{M}{4} + 2\frac{N}{4})T_c$$

   Seleccionaremos la distribución de procesos que implique el menor tiempo de comunicación.

   (d) En función de los valores de las variables, la aceleración resultado variará desde valores muy próximos a cero hasta valores cercanos a 16 (aceleración lineal). La carga computacional de cada proceso es siempre $\frac{MN}{16}$, porque está perfectamente balanceada, y por lo tanto el tiempo de computación es $T_f \frac{MN}{16}$.

$$t_{seq} = T_f MN$$
$$t_{par} = T_{comm}(4x4) + T_f \frac{MN}{16}$$
$$speedup = \frac{t_{seq}}{t_{par}}$$

2. Ejercicio 2

(a) $M\_local = \frac{M}{P} = \frac{M}{2^{C_1+2}}$, $N\_local = N$

(b) Para declarar la función podemos definir un estado S como "int *" o "int **". Como el intercambio se produce por filas individuales, es indiferente si las filas se encuentran reservadas consecutivamente en memoria o no. Por tanto, hemos elegido "int **" por comodidad.

Si la matriz S estuviese declarada como "int *", la dirección de la fila i de la matriz en lugar de ser S[i] sería &(S[i*N_local]), o bien S + i*N_local.

```
void intercambia_datos(int **S, rango)
{
  int inferior = (rango + 1) % P;
  int superior = (rango + P - 1)%P;

  if (rango % 2)
  {
    MPI_Send(S[1], N, MPI_INT, superior, 0, MPI_COMM_WORLD);
    MPI_Send(S[M_local], N, MPI_INT, inferior, 0, MPI_COMM_WORLD);

    MPI_Recv(S[M_local+1], N, MPI_INT, inferior, 0, MPI_COMM_WORLD);
    MPI_Recv(S[0], N, MPI_INT, superior, 0, MPI_COMM_WORLD);
  }
  else
  {
    MPI_Recv(S[M_local+1], N, MPI_INT, inferior, 0, MPI_COMM_WORLD);
    MPI_Recv(S[0], N, MPI_INT, superior, 0, MPI_COMM_WORLD);

    MPI_Send(S[1], N, MPI_INT, superior, 0, MPI_COMM_WORLD);
    MPI_Send(S[M_local], N, MPI_INT, inferior, 0, MPI_COMM_WORLD);
  }
}
```

(c) Igual que en el ejercicio anterior, hemos escogido declarar las matrices S1 y S0 como "int **". Cada proceso debe comprobar si existen diferencias entre sus estados S1 y S0, y a continuación se hace una reducción de tipo LOR (OR lógico) para verificar si existen diferencias entre los estados locales de alguno de los procesos, y un broadcast para que la función retorne el mismo valor en todos los procesos. Dado que la variable *distintas* solo toma valores 0 o 1, también funcionaría correctamente la operación BOR (bitwise OR), y además, como C interpreta cualquier valor distinto de 0 como "true", también es válida una reducción de tipo suma.

```
int continua_bucle(int **S1, int **S0)
{
  int distintas = 0;
  int debe_continuar;

  for (int i=1; i <= M_local; i++)
    for (int j=0; j < M_local; j++)
      if (S1[i][j] != S0[i][j])
      {
        distintas = 1;
        break;
      }

  MPI_Reduce(&distintas, &debe_continuar, 1, MPI_INT,
             MPI_LOR, 0, MPI_COMM_WORLD);
  MPI_Bcast(&debe_continuar, 1, MPI_INT, 0, MPI_COMM_WORLD);

  return debe_continuar;
}
```