EXAMEN DE CONCURRENCIA Y PARALELISMO

Bloque II: Paralelismo

APELLIDOS: _____ **NOMBRE:** _____

Convocatoria extraordinaria

12 de Julio de 2018

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

[2,5p] 1. **Conceptos de paralelismo.** Una casa de apuestas está interesada en realizar análisis exhaustivos de resultados históricos de partidos para el cálculo de las contrapartidas de las apuestas realizadas por los usuarios. Por desgracia, en los primeros tests de implementación se detecta que el sistema es demasiado lento debido al gran tamaño de los datos. Para solucionarlo, se decide paralelizar el sistema. Para ello, se trocea la base de datos original entre N nodos, cada uno con su propio sistema gestor local, y se proporciona una interfaz a un nodo adicional con un gestor global que gestiona el lanzamiento de operaciones paralelas sobre el conjunto. Las operaciones que deben ser paralelizadas son operaciones clásicas de consulta de bases de datos (búsquedas, reducciones, selecciones, etc.).

   Contesta de forma razonada a las siguientes preguntas:

   a) ¿Qué estrategia o estrategias de descomposición en tareas se han empleado al diseñar el sistema paralelo? ¿Y de asignación de tareas?

   b) Explica cómo se implementaría sobre la arquitectura diseñada una operación "resumen" que calcule cuántas veces ha ganado, empatado y perdido cada equipo en un enfrentamiento determinado entre los equipos A y B.

   c) Supongamos que la base de datos original consta de 2^{40} filas, que la operación en su versión secuencial tarda 64 ciclos de computación por fila, y que una comunicación sobre la red entre dos procesos lleva 2^{16} ciclos de computación. ¿Cuál es el speedup obtenible por la aproximación propuesta para la operación "resumen" en función del número de procesos utilizado? ¿Cuál es la eficiencia paralela utilizando 16 procesos? ¿Para qué tamaño de N se obtiene un speedup de 1.0 trabajando con 256 procesos?

   d) Considera la operación de búsqueda de un partido entre dos equipos A y B en una fecha concreta en dos situaciones: cuando tanto la base de datos original como cada trozo resultante están ordenados por fecha, y cuando están totalmente desordenados. ¿Cómo se implementaría en cada caso? ¿En cuál de las dos situaciones se puede realizar la búsqueda de forma más eficiente? ¿En cuál se obtiene un mayor beneficio de la paralelización, asumiendo que las comunicaciones tienen coste cero? Ten en cuenta que una búsqueda sobre un conjunto ordenado de datos puede implementars como una búsqueda binaria, en lugar de exhaustiva, de forma que en cada iteración del bucle de búsqueda podemos descartar una mitad completa de la tabla.

   e) ¿Qué tipo de escalabilidad presenta la operación "resumen"? ¿Y la operación de búsqueda sobre la base de datos desordenada? ¿Y la operación de búsqueda sobre la base de datos ordenada?

Solución

a) [0,25p] Lo normal es que lo vean como una descomposición de dominio, dado que se trata de ejecutar la misma operación de forma simultánea sobre conjuntos de datos diferentes, aunque podrían justificarse otras alternativas. La asignación de tareas, con los descrito en el enunciado, parecería dinámica, con el gestor global actuando como maestro y los gestores locales como esclavos.

En todo caso, cualquier respuesta razonablemente justificada me vale.

b) [0,5p] Se trataría simplemente de aplicar dicha operación sobre cada uno de los fragmentos locales y, a continuación, realizar una reducción sobre cada uno de los tres valores extraidos.

c) [0,75p] El número de ciclos de computación original sería de:

$$T_s = 2^{40} \times 2^6 = 2^{46} ciclos$$

El tiempo paralelo, asumiendo un reduce binomial:

$$T_p = \frac{2^{46}}{N} + 3 \times 2^{16} \times log_2(N)$$

Por tanto, para 16 procesos:

$$T_{16} = \frac{2^{46}}{2^4} + 3 \times 2^{16} \times log_2(2^4) = 2^{42} + 3 \times 2^{20} ciclos$$

$$sp_{16} \simeq \frac{2^{46}}{2^{42} + 3 \times 2^{20}} = 15{,}99$$

$$eff_{16} \simeq \frac{15{,}99}{16} = 0{,}99$$

Trabajando con 256 procesos:

$$T_s = T_{256} \Rightarrow$$

$$N \times 2^6 = \frac{N \times 2^6}{2^8} + 3 \times 2^{16} \times log_2(2^8) \Rightarrow$$

$$N = \frac{N}{2^8} + 3 \times 2^{10} \times 2^3 \Rightarrow$$

$$2^8 N = N + 3 \times 2^{21} \Rightarrow$$

$$N \simeq = 3 \times 2^{13}$$

d) [0,75p] En el caso en que la tabla se encuentra desordenada, debemos buscar la tabla completa de forma secuencial. El tiempo de ejecución es por tanto $O(N)$, y el beneficio ideal de la paralelización es lineal. Cuando la tabla se encuentra ordenada, sin embargo, se puede usar una búsqueda binaria, cuya complejidad es $O(log(N))$. Al dividir el tamaño de los datos por P, el beneficio de la paralelización es un término constante, pues $\Theta(log(N/P)) \in O(log(N))$, por lo que el speedup será muy inferior a 1.

e) [0,25p] Tanto "resumen" como la búsqueda sobre la base de datos desordenada presenta escalabilidad fuerte (al menos idealmente), dado que "resumen" sólo precisa reducciones (realizables en tiempo logarítmico), mientras que la búsqueda sólo precisa una comunicación del gestor local que encuentra el dato buscado al gestor central. Dado que la operación de búsqueda sobre la base de datos ordenada no es eficiente en ningún caso (excepto para resolver problemas que no pueden ser ejecutados por falta de recursos, p. ej., espacio en disco), no tiene sentido hablar de escalabilidad.

Como en el apartado a), otras respuestas pueden ser aceptables si vienen razonablemente motivadas.

[2,5p] 2. **Diseño de algoritmos paralelos.** Se desea paralelizar un producto más suma ($X = A*B+X$) en el que participan tres matrices de dos dimensiones teniendo en cuenta que:

- la matriz A es solo de entrada con dimensiones MxK.
- la matriz B es solo de entrada con dimensiones KxN.
- la matriz X es de entrada y salida con dimensiones MxN.
- los elementos de la matriz X al terminar el programa deben ser: $x_{i,j} = \sum_{l=0...K-1} a_{i,l} * b_{l,j} + x_{i,j}$

La máquina sobre la que vamos a trabajar solo permite a un procesador leer y escribir las matrices de disco, y además NO permite alternar de forma eficiente entre comunicaciones y computaciones. Por tanto, los algoritmos paralelos diseñados para ella solo deben enviar datos antes y/o después de que todos los procesos hayan realizado su cómputo.

a) Explica qué tipo de descomposición y asignación de tareas usarías, justificando la respuesta. ¿En alguno de los casos tu decisión dependería de los valores de M, N y K? ¿Por qué?

b) Implementa tu algoritmo paralelo en MPI haciendo uso de las operaciones colectivas que sea posible, teniendo en cuenta que $M > N$. A continuación tienes un cuadro con la sintaxis de varias rutinas MPI.

c) Supongamos que la red tiene forma de anillo tal que cada proceso p solo puede enviar mensajes a $p+1$ de forma unidireccional (obviamente el último proceso puede enviar mensajes al proceso 0). Explica de palabra cómo implementarías las colectivas que has usado en el apartado anterior.

d) Si tenemos una ejecución con $M = 1024$, $K = 512$ y $N = 256$ sobre cuatro procesos en un sistema como el descrito en el apartado anterior. ¿Cuál sería el speedup obtenido si el cálculo de cada elemento de X necesita K ciclos y cada mensaje entre dos procesos necesita $S + 1$ ciclos, siendo S el tamaño del mensaje?

```
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

a) [0,5p] La descomposición será de dominio, creando una tarea por cada elemento de X a calcular. En cuanto a la asignación, en principio no hay una gran diferencia entre usar una distribución por bloques o cíclica, ya que los cálculos son homogéneos y no se depende de los vecinos. Usaremos una distribución por bloques porque permite simplificar el algoritmo y en muchas ocasiones hacer más eficientes las comunicaciones. Como no se deben realizar comunicaciones en medio de la computación debemos garantizar que los procesos tienen todos los datos necesarios para trabajar al principio. Habría dos opciones:

- Una distribución for filas de X. Esto llevaría a hacer la misma distribución de datos en A y a replicar B. Es la más adecuada si $M > N$ porque así minimizamos la cantidad de datos a replicar.
- Una distribución for columnas de X. Esto llevaría a hacer la misma distribución de datos en B y a replicar A. Es la más adecuada si $N > M$ porque así minimizamos la cantidad de datos a replicar.

b) [1p] Como $M > N$ aplicamos a distribución por filas:

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

int dims[3]; // Dimensiones M, N, K
float *A, *B, *X;
if(myrank == 0) // Esta funcion reserva memoria y lee los valores
  leeMatrices(A, B, X, &dims[0], &dims[1], &dims[2]);

// Envia M, N y K
MPI_Bcast(dims, 3, MPI_INT, 0, MPI_COMM_WORLD);

if(myRank > 0)
  B = (float *)malloc(sizeof(float)*dims[1]*dims[2]);

// Replica B
MPI_Bcast(B, dism[1]*dims[2], MPI_FLOAT, 0, MPI_COMM_WORLD);

filas = ceil(dims[0]/P);
float *miA = (float *)malloc(sizeof(float)*filas*dims[2]);
float *miX = (float *)malloc(sizeof(float)*filas*dims[1]);

// En caso de que M no sea multiplo de P se reserva
// memoria adicional en A y X para usar Gather y Scatter
if((dims[0]%P != 0) && (myRank == 0)){
  A = (float *)realloc(A, sizeof(float)*P*filas*dims[2]);
  X = (float *)realloc(X, sizeof(float)*P*filas*dims[1]);
}

// Reparte A y X
MPI_Scatter(A, filas*dims[2], MPI_FLOAT, miA, filas*dims[2],
            MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Scatter(X, filas*dims[1], MPI_FLOAT, miX, filas*dims[1],
            MPI_FLOAT, 0, MPI_COMM_WORLD);

// Trabaja
for(i=0; i<filas; i++){
  for(j=0; j<dims[1]; j++){
    for(k=0; k<dims[2]; k++){
      miX[i*dims[1]+j] += miA[i*dims[2]+k]*B[k][j];
    }
  }
}

// Recoge X
MPI_Gather(miX, filas*dims[1], MPI_FLOAT, X, filas*dims[1],
           MPI_FLOAT, 0, MPI COMM WORLD );

if(myRank == 0)
  escribeMatriz(X, dims[0], dims[1]);

MPI_Finalize();
```

3. [0,5p]

   El broadcast sería la implementación más sencilla. Cada proceso empezaría por calcular su sucesor ($myrank + 1$, salvo para el último proceso cuyo sucesor sería el 0) y su predecesor

($myrank - 1$, salvo para el proceso 0 cuyo predecesor sería el último). Habría que distinguir entre tres tipos de procesos:

- Proceso *root* solo enviaría los datos a su sucesor con MPI_Send.
- El proceso cuyo sucesor es el *root* solo tiene que recibir los datos del predecesor con MPI_Recv.
- Los demás procesos tienen que recibir primero los datos de su predecesor y dierctamente enviarlos a su sucesor.

En el caso del scatter sería de forma similar pero cada vez el mensaje se hace más pequeño. Por tanto los procesos deben calcular el tamaño del mensaje que reciben y el que envían. Además, a la hora de reenviar el mensaje hay que hacerlo no desde la posición 0 de los recibido sino desde más adelante.

En el gather los tamaños de los mensajes también varían, pero en este caso van incrementando. Cada proceso debe tener un buffer donde la primera parte son sus datos iniciales, el MPI_Recv alojará los datos recibidos justo después de los datos del proceso. El MPI_Send enviará todo el buffer junto: primero los datos propios y concatenados los datos recién recibidos.

4. [0,5p] El tiempo secuencial es el de calcular toda la matriz: $M * N * K = 134217728$ ciclos. El tiempo paralelo se divide en tiempo de computación y tiempo de comunicación. El tiempo de computación es el que tarda cada proceso (en este caso los bloques tienen el mismo número de filas): $filas * N * K = 33554432$ ciclos. El tiempo de comunicación es la suma de:

   - Broadcast de B. Eso son tres mensajes de tamaño $K * N$. Es decir, $3 * (K * N + 1) = 393219$ ciclos.
   - Scatter de A. Eso es un mensaje entre P0 y P1 de 768 filas, otro entre P1 y P2 de 512 filas, y un último entre P2 y P3 de 256 filas. En total es $(768 * K + 1) + (512 * K + 1) + (256 * K + 1) = 786435$ ciclos.
   - Scatter de X. Muy similar al scatter de A pero ahora las filas son de tamaño N. En total es $(768 * N + 1) + (512 * N + 1) + (256 * N + 1) = 393219$ ciclos.
   - Gather de X. El tamaño y número de mensajes es igual que para el scatter pero ahora en sentido contrario: 393219 ciclos.

   En total son 1966092 ciclos de comunicaciones, lo que hace un tiempo paralelo total de: 35520524 ciclos. El speedup es $3,78$.