EXAMEN DE CONCURRENCIA Y PARALELISMO

Bloque II: Paralelismo

APELLIDOS: _____ **NOMBRE:** _____

Convocatoria extraordinaria

7 de Julio de 2017

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.** Para resolver un determinado problema de forma paralela se usa una distribución bloque bidimensional de una matriz bidimensional como la vista en los apuntes de teoría. Dada una matriz con 500 filas y 500 columnas, y dada una malla de 48 procesos organizada en 6 filas y 8 columnas:

a) [0,5p] ¿Qué cantidad de elementos será asignado a cada uno de los procesos?

b) [0,5p] ¿A qué proceso será asignado el elemento (200, 200) y qué posición ocupa en la submatriz local de ese proceso?

c) [0,5p] Supón que el programa secuencial es paralelizable desde el principio hasta el fin, y se consigue desarrollar una aplicación paralela que no incluye ningún tipo de sobrecarga (ni por comunicaciones ni por sincronizaciones). De acuerdo al resultado del apartado a), si el tiempo de cálculo para cada elemento de la matriz es un segundo, ¿cuál será el tiempo paralelo para la ejecución con 48 procesos? ¿Cuál será el *speedup*?

d) [0,5p] Ahora supón una aplicación diferente que necesita, en un primer momento, unas comunicaciones para que el Proceso 0 distribuya la matriz de entrada entre todos los procesos (a cada proceso le tocarán solo los datos con los que tiene que trabajar). Para ello se usa una aproximación poco eficiente donde este proceso tiene un bucle en el que va mandando el bloque con los datos correspondientes a cada uno de los procesos (desde Proceso 1 a Proceso 47). Un mensaje a cualquier destinatario necesita un tiempo fijo de 100ms y un tiempo variable que depende del tamaño del bloque (1ms por elemento). Asumiendo que el tiempo de recepción es despreciable, ¿cuál es ahora el tiempo paralelo de la aplicación ejecutada con 48 procesos? ¿Y el *speedup*?

e) [0,5p] Si la red de interconexión sigue un modelo de malla 2D (solo los procesos de la misma fila y la misma columna están conectados), describe una distribución más eficiente que la del apartado anterior para la matriz de entrada. No hace falta pseudocódigo.

SOLUCIÓN

a)
- La mayoría de procesos tendrán $84 \times 63(5292)$ elementos.
- Los procesos de la última fila tendrán $80 \times 63(5040)$ elementos.
- Los procesos de la última columna tendrán $84 \times 59(4956)$ elementos.
- El proceso 47 tendrá $80 \times 59(4720)$ elementos.

b) Al Proceso (2,3) o Proceso 11. En la posición (32,11)

c)
- El tiempo paralelo es el de los procesos más costosos: $5292s$

- El tiempo secuencial es $500 \times 500 = 250000s$. Por tanto el speedup es: $\frac{250000}{5292} = 47{,}24$

d)
- El tiempo de comunicación es la suma de los 100ms de envío a cada proceso y el tiempo de enviar los 250000 elementos: $(48 * 100 + 500 * 500) \times 0{,}001 = 254{,}80s$.
- El tiempo paralelo es estonces la suma de este tiempo de comunicaciones ($254{,}80s$) y el tiempo de computación del Proceso 0 ($5292s$): $5796{,}21s$
- El speedup es $\frac{250000}{5796{,}21} = 43{,}13$

e) Hay diferentes variantes. Una aproximación sería que el Proceso 0 empezase distribuyendo las filas completas entre los procesos de la primera columna de la malla y luego estos, en paralelo, distribuyen las columnas entre los procesos de su misma fila de la malla.

[2,5p] 2. **Diseño de algoritmos paralelos.** Se desea paralelizar sobre un clúster conformado por $P > 1$ ordenadores con memoria distribuida el siguiente problema, donde A es una matriz de $N \times N$ elementos, y v y r son vectores de $N \times 1$ elementos:

```
int i, N, P, myrank;
double *A, *v, *r, tmp;

readInputs(&A, &N, &v);
r = (double *)malloc(sizeof(double) * N);

matrixVectorProduct(A, N, N, v, r); /* r = A x v */
tmp = 0.;
for(i = 0; i < N; i++) {
  tmp += r[i] * v[i];
  v[i] = r[i];
}

save(v, N, tmp); /* saves vector v and scalar tmp in disk */
```

El programa asumirá que inicialmente N, A y v son obtenidos de un fichero en el proceso 0 mediante la función `readInputs(double **A, int *N; double **v)`, la cual reserva el espacio de memoria para A y v antes de rellenarlos. Así mismo, el producto matriz vector lo realiza la función `matrixVectorProduct(const double *matrix, int rows, int cols, const double *inputVector, double *resultVector)`, donde el nombre de cada parámetro indica su significado, y el proceso 0 deberá volcar los resultados v y tmp a un fichero al finalizar el bucle mediante la función `save`. Además podemos asumir que $N \gg P$ y N mód $P = 0$

a) [0,5p] Explica qué tipo(s) de descomposición de tareas aplicarías justificando la respuesta.

b) [0,75p] Diseña la paralelización y represéntala mediante un pseudocódigo a alto nivel no basado en código, sino en expresar con palabras qué se hace en cada punto del programa paralelizado. Enumera cada paso de tu pseudocódigo.

c) [1,25p] Implementa tu algoritmo paralelo en MPI haciendo uso de las operaciones colectivas que sea posible (a continuación tienes un cuadro con la sintaxis de varias rutinas MPI).

```
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_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)
int MPI_Allgather(const void *sendbuf, int  sendcount,
               MPI_Datatype sendtype, void *recvbuf, int recvcount,
               MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

SOLUCIÓN

a) Aplicaría una descomposición de dominio al producto matriz-vector (con un bloque consecutivo de filas de la matriz asignado a cada proceso), dado que esta operación se produce aplicando en paralelo la misma operación sobre los distintos puntos de un dominio. Como la matriz es la estructura más grande de la operación, la descomposición está basada en ella. Al tener cada proceso un bloque de filas consecutivos de la matriz, para poder hacer su parte de la computación requerirá todo el vector v y generará la porción del vector r asociada a las filas de la matriz que posee.

Aplicaría una descomposición una recursiva al producto escalar (con un bloque consecutivo de elementos distinto a reducir en cada proceso), ya que el problema se descompondría en problemas más pequeños (producto escalar de cada subbloque en cada proceso) que luego hay que combinar (en este caso, sumando) para obtener el resultado final.

b) Pseudocódigo:

1) el proceso 0 lee A y v
2) se difunde (broadcast) N a todos los procesos
3) los procesos que no son 0 reservan el espacio de memoria para A y v
4) todos los procesos reservan el espacio para r
5) se difunde (broadcast) v a todos los procesos
6) se descompone en grupos de N/P filas A entre los procesos (scatter)
7) cada proceso multiplica sus filas de A por v, obteniendo el subrango asociado de elementos de r
8) cada proceso hace el producto escalar de su porción de r por la porción correspondiente de v
9) los resultados parciales se reducen entre todos los procesadores de forma que el proceso 0 tiene tmp (reduce)
10) el proceso 0 recoge en v las porciones que tiene cada uno de r (gather)
11) el proceso 0 guarda v y tmp en un fichero.

c) Código:

```
int i, N, P, myrank, NLocal, myfirstelem;
double *A, *v, *r, tmp, tmp0;

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

if(!myrank) {
  readInputs(&A, &N, &v);
}

MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
NLocal = N / P;
myfirstelem = NLocal * myrank;
if(myrank) {
    A = (double *)malloc(sizeof(double) * NLocal * N);
    v = (double *)malloc(sizeof(double) * N);
}
r = (double *)malloc(sizeof(double) * NLocal);

MPI_Bcast(v, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(A, NLocal * N, MPI_DOUBLE, A, NLocal * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);


matrixVectorProduct(A, NLocal, N, v, r);
tmp0 = 0.;
for(i = 0; i < NLocal; i++) {
  tmp0 += r[i] * v[myfirstelem + i];
}

MPI_Reduce(&tmp0, &tmp, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Gather(r, NLocal, MPI_DOUBLE, v, NLocal, MPI_DOUBLE, 0, MPI_COMM_WORLD);


if(!myrank) {
  saveVector(v, N, tmp);
}

MPI_Finalize();
```