Ejercicio práctico IV

Trabajo realizado por: Antonio Gómez Giménez

Ejercicios:

1. Multiplicación de dos matrices según algoritmo descrito en el último apartado del tema “Introducción - Generalidades Sistemas Multiprocesador”.

2. Desarrollar un programa en MPI que realice la ordenación de un vector de “n” elementos (configurable) en “p” computadores.

Como en el laboratorio no se podía implementar en varios ordenadores, se va a implementar en mi propio ordenador, por tanto, para cada ejercicio se usarán varios procesos para un único ordenador usando MPI. Si fuera en el laboratorio, sería 1 proceso para cada ordenador.

Para el ejercicio uno se ha realizado el siguiente código:

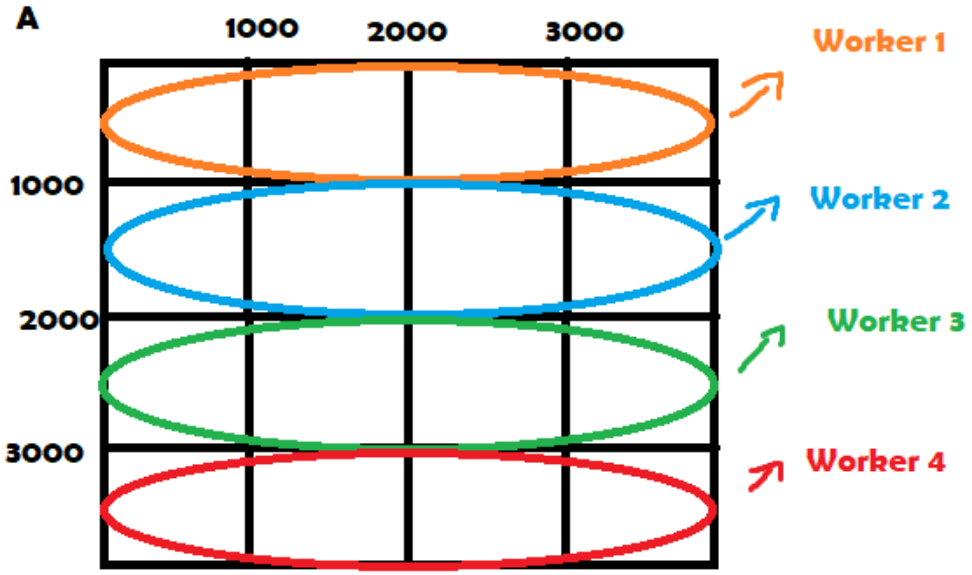
```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define TAMMATRIZ 4000
#define ITERACIONES_MATRICES 4000


float A[TAMMATRIZ][TAMMATRIZ];
float B[TAMMATRIZ][TAMMATRIZ];
float Cparalelo[TAMMATRIZ][TAMMATRIZ];

MPI_Status status;

int main (int argc, char *argv[]) {

    srand (time(NULL));
    double start_t, end_t, total_t;

    int numeroTareas, idtarea, numtrabajadores, TamSeg, offset, dest;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &idtarea);
    MPI_Comm_size(MPI_COMM_WORLD, &numeroTareas);

    numtrabajadores = numeroTareas - 1;
```

Primero creamos las matrices e inicializamos todas las variables que se necesitarán a posteriori. Inicializamos MPI, usamos MPI_Comm_rank para saber el rango o proceso dentro del comunicador (es el comunicador por defecto MPI_COMM_WORLD) y MPI_Comm_size para conocer el número de procesos presentes en la ejecución de un programa.
Una vez realizado lo anterior creamos dos partes del código, una para el master y otra para los workers:

```
29      /*----------------------------maestro-----------------------------------*/
30
31      if (idtarea == 0) {
32
33          //relleno las matrices con valores aleatorios
34          for(int i=0;i<ITERACIONES_MATRICES;i++){
35              for(int j=0;j<ITERACIONES_MATRICES;j++){
36                  A[i][j]= rand() % 10000;
37                  B[i][j]= rand() % 10000;
38                  Cparalelo[i][j]= 0;
39              }
40          }
41
42          //paralelo
43          printf("Realizando multiplicacion en paralelo: \n");
44
45          //enviamos la matriz de datos a los esclavos para realizar las tareas
46          TamSeg = ITERACIONES_MATRICES/numtrabajadores;
47          offset = 0;
48
49          start_t = clock();
50
51          for (dest=1; dest <= numtrabajadores; dest++){
52              MPI_Send(&offset, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
53              MPI_Send(&TamSeg, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
54              MPI_Send(&A[offset][0], TamSeg*ITERACIONES_MATRICES, MPI_FLOAT,dest,1, MPI_COMM_WORLD);
55              MPI_Send(&B, ITERACIONES_MATRICES*ITERACIONES_MATRICES, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
56              offset = offset + TamSeg;
57          }
58
59          /* espera los resultados de las tareas */
60          for (int i=1; i<=numtrabajadores; i++){
61              MPI_Recv(&offset, 1, MPI_FLOAT, i, 2, MPI_COMM_WORLD, &status);
62              MPI_Recv(&TamSeg, 1, MPI_FLOAT, i, 2, MPI_COMM_WORLD, &status);
63              MPI_Recv(&Cparalelo[offset][0], TamSeg*ITERACIONES_MATRICES, MPI_FLOAT, i, 2, MPI_COMM_WORLD, &status);
64          }
65
66          end_t = clock();
67
68          total_t = (end_t - start_t) / CLOCKS_PER_SEC;
69
70          printf("C[0][0]: %f\n", Cparalelo[0][0] );
71          printf("Tiempo total de CPU para paralelizacion: %f\n", total_t  );
72
73      }
```

Primero vemos el idtarea, si es 1 significa que es el master, en el caso contrario significa que ese proceso es un worker. En el caso de ser el master, rellena las matrices con valores aleatorios y se comienza a realizar la división de los datos para cada worker, en este apartado se contará el tiempo que transcurre, este tiempo será la paralelización total.
A cada worker se le va a asignar un tamaño de segmento, este viene dado por el número de iteraciones de las matrices dividido entre la cantidad de trabajadores que tenemos, por tanto si las matrices son de 4000 y tenemos 4 trabajadores, cada trabajador tendrá un segmento de 1000.

Creamos un bucle for que irá desde 1 hasta 4(es el número de trabajadores que especificamos), de esta forma le enviaremos los datos con MPI_Send.
Primero enviamos offset que indicará dónde se empieza, después enviamos el tamaño de segmento, posteriormente se envía la matriz A, pero únicamente pasándose los datos necesarios de cada worker, esto se consigue indicando dónde comienza y especificando el tamaño que ocupa. Esto se ve más sencillo en la siguiente imagen:



Respecto a la matriz B, se pasa completa, no se realiza paralelización. Entonces lo que se busca es paralelizar respecto a la variable i del bucle, se verá más claro al explicar el worker.
No se nos puede olvidar, que cada vez que demos los datos a un worker hay que incrementar el offset, sino todos cogerán los mismos datos.

Una vez enviados los datos se esperará a recibir datos de los workers, esto se consigue gracias a la función MPI_Recv (lógicamente tendremos que crear un bucle para recoger todos los procesos).
De cada proceso recibiremos el tamaño y el offset, aparte de la matriz C (donde se encontrarán los resultados), esto se debe a que al regresar del worker hay que especificar donde se guardarán esos datos dentro de la matriz C, así evitamos que se machaquen entre sí los procesos.

Finalmente se muestra el tiempo que se tarda en ejecutar la paralelización.

En la siguiente imagen observamos como se ha implementado cada worker:

```
75      /*-----------------------------trabajador----------------------------------*/
76
77      if (idtarea > 0) {
78          MPI_Recv(&offset, 1, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
79          MPI_Recv(&TamSeg, 1, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
80          MPI_Recv(&A, TamSeg*ITERACIONES_MATRICES, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
81          MPI_Recv(&B, ITERACIONES_MATRICES*ITERACIONES_MATRICES, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
82
83          //multiplicacion de matrices
84          for(int i=0;i<TamSeg;i++){
85              for(int k=0;k<ITERACIONES_MATRICES;k++){
86                  for(int j=0;j<ITERACIONES_MATRICES;j++){
87                      Cparalelo[i][j]+= A[i][k]*B[k][j];
88                  }
89              }
90          }
91
92          MPI_Send(&offset, 1, MPI_FLOAT, 0, 2, MPI_COMM_WORLD);
93          MPI_Send(&TamSeg, 1, MPI_FLOAT, 0, 2, MPI_COMM_WORLD);
94          MPI_Send(&Cparalelo, TamSeg*ITERACIONES_MATRICES, MPI_FLOAT, 0, 2, MPI_COMM_WORLD);
95      }
96
97      MPI_Finalize();
98
99      return(0);
100 }
```

Como podemos observar, se espera a recibir los datos enviados por el máster, una vez tiene esos datos se realiza la multiplicación como vimos en prácticas anteriores. La única diferencia es que el bucle i, solo llegará hasta el tamaño del segmento que nosotros hemos indicado, para que así cada worker haga su parte.
Una vez calculado, estos datos que se encuentran en la matriz C se devuelven en su posición correcta al master, donde allí se recibe.

Tras realizar todo el código se lanzó el programa con cuatro procesos, con el bucle ikj y matrices de tamaño 4000 y el tiempo obtenido fue el siguiente:

```
antoniogg@antoniogg-SATELLITE-L50D-C:~/Escritorio/AP/P4$ mpiexec -n 4 ./a.out
Realizando multiplicacion en paralelo:
C[0][0]: 99626532864.000000
Tiempo total de CPU para paralelizacion: 175.580868
```

Si lo comparamos con la práctica anterior, al paralelizar ikj usando openMP con 4 hilos el resultado obtenido es de 145.713793.
Por tanto si lo comparamos con MPI, el tiempo que obtenemos es peor. Esto es lógico ya que crear un proceso lleva un costo computacional superior al costo computación que lleva crear un hilo, por ello, el tiempo es un poco más elevado, aparte MPI se centra en el paso de mensajes entre computadores por ello es más costos (enviar datos, recibir datos, etc), aún así es un tiempo muy bueno sobre todo si lo comparamos con el serie que era un tiempo de 407.826484.

Respecto al apartado 2 simplemente realice MPI para ordenación burbuja, lo que hacía era pasarle a cada proceso una parte de los datos y lo ordenaba. Finalmente lo devolvieron al master y este aplicaba ordenación burbuja de nuevo. Como considero que no es una forma eficiente ni correcta no voy a poner ni el código, ya que lo correcto sería implementar que al regresar los datos se almacenarán de forma correcta o implementando otro método mucho más eficiente.