PROGRAMACIÓN PARALELA Y DISTRIBUIDA

PRACTICA Nº 5: PROGRAMACIÓN Y RENDIMIENTO

TEMAS: Ejercicios de programación en MPI. Rendimiento de aplicaciones paralelas.
OBJETIVOS: Que el alumno resuelva problemas a través del paralelismo, y desarrolle soluciones en lenguaje C sobre un cluster de computadoras mediante la librería de comunicación MPI. Asimismo, que evalúe y valore el comportamiento obtenido.

Ejercicio Nº1 – Escribir un programa en MPI que implemente el clásico “Hola Mundo!” y además de saludar indique en qué nodo se está ejecutando.

Ejercicio Nº2 – Escribir un programa en MPI que implemente un “ping-pong”, esto es, un programa en el que un proceso envía un mensaje a otro y éste último lo devuelve inmediatamente al primero. Utilizar la función MPI_Wtime para calcular cuánto tiempo se invierte en esta operación.

Ejercicio Nº3 – Dada la malla de procesos que se muestra a continuación, en la que cada proceso aparece con su rango en el comunicador MPI_COMM_WORLD. Escribir una única llamada a MPI_Comm_Split que permita obtener un nuevo comunicador (new_comm) que incluya a los procesos que aparecen con fondo gris en la figura pero cuyos rangos se ordenen de forma inversamente proporcional al rango que tenían en el comunicador original. Utilizar el menor número de órdenes posible.

a)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

b)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Ejercicio Nº4 – Re-implementar el juego de Siete y Medio realizado en la práctica Nº2, pero ahora siguiendo el esquema Master/Worker y comunicación mediante MPI.

Ejercicio Nº5 – Re-implementar el juego de Lotería realizado en la práctica Nº2, considerando con qué modelo paralelo se identifica, y utilizando primitivas de MPI.

Ejercicio Nº6 – Utilizando las operaciones punto-a-punto de MPI, implementar la función `Bcast(int source, char *msg, int tam)` que emulará a la operación de broadcast. ¿Cuántas iteraciones invierte la función?

Ejercicio N°7 – Retomar el ejercicio 4 del TP4, en el que se tiene una función `ejemplo`, donde se supone que las funciones T1, T3 y T4 tienen un coste de *n* y las funciones T2 y T5 de 2*n*, siendo *n* un valor constante:

```
double ejemplo(int i, int j)
{
 double a, b, c, d, e;
 a = T1(i);
 b = T2(j);
 c = T3(a+b, i );
 d = T4(a/c);
 e = T5(b/c);
 return d+e; /*T6*/
}
```

a) Implementar la versión paralela del algoritmo dado usando MPI con dos procesos. Ambos procesos invocan a la misma función con el mismo valor de los argumentos *i* y *j*.

b) Analizar y valorar los resultados obtenidos, en base a un análisis de rendimiento.

Ejercicio N°8 – Queremos medir la latencia de un anillo de n procesos en MPI, entendiendo por latencia el tiempo que tarda un mensaje de tamaño 0 en circular entre todos los procesos. Un anillo de n procesos MPI funciona de la siguiente manera: P0 envía el mensaje a P1, cuando éste lo recibe, lo reenvía a P2, y así sucesivamente hasta que llega a Pn−1 que lo enviará a P0. Escribir un programa MPI que implemente este esquema de comunicación y muestre la latencia. Es recomendable hacer que el mensaje dé varias vueltas al anillo, y luego sacar el tiempo medio por vuelta, para obtener una medida más fiable.

Ejercicio N°9 – Escribir un programa en MPI que encuentre y contabilice la cantidad de números primos en un determinado rango de números. Realizar el diseño experimental para estudiar el rendimiento en relación al rango de búsqueda y la cantidad de nodos involucrados. Evaluar los resultados hallados de acuerdo a las métricas estudiadas.

Ejercicio N°10 – Escribir una función con la siguiente cabecera, la cual debe hacer que los procesos con identificación *proc1* y *proc2* intercambien el vector *x* que se pasa como argumento, mientras que en el resto de los procesos el vector *x* no debe sufrir modificaciones.

```
void intercambiar(double x[N], int proc1, int proc2)
```

No utilizar las funciones MPI_Sendrecv, MPI_Sendrecv_replace ni MPI_Bsend.

Ejercicio N°11 – La siguiente función muestra por pantalla el máximo de un vector *v* de *n* elementos y su posición:

```
void func(double v[], int n) {
  int i, posmax = 0;
  double max = v[0];
for (i=1; i<n; i++) {
    if (v[i]>max) {
      max = v[i];
      posmax = i;
    }
  }
```

```
    printf("Máximo: &lf. Posición: &d\n", max, posmax);
}
```

Escribir la versión paralela MPI con la siguiente cabecera, donde los argumentos *rank* y *np* han sido obtenidos mediante las funciones de MPI.

```
void func_par(double v[], int n, int rank, int np)
```

La función debe asumir que el arreglo *v* del proceso 0 contendrá inicialmente el vector, mientras que en el resto de los procesos dicho arreglo podrá usarse para almacenar la parte local que corresponda. El cálculo del máximo se repartirá de forma equitativa entre los procesos. Se deben usar comunicaciones punto a punto y se puede asumir que *n* es múltiplo del número de procesos.

Ejercicio N°12 – El siguiente programa cuenta el número de ocurrencias de un valor en una matriz.

```
#include <stdio.h>
#include DIM 1000

void leer(double A[DIM][DIM], double *x)
{...}

int main(int argc, char *argv[])
{
  double A[DIM][DIM], x;
  int i, j, cont;
  leer(A,&x);
  cont=0;
  for (i=0; i<DIM; i++)
    for (j=0; j<DIM, j++)
      if (A[i][j]==x) cont++;
  printf("%d ocurrencias\n", cont);
  return 0;
}
```

Hacer una versión paralela MPI del programa usando operaciones de comunicación colectiva cuando sea posible. La función leer sólo será invocada por el proceso 0. Se puede asumir que DIM es divisible entre el número de procesos.

Ejercicio N°13 – Supongamos una matriz de enteros A[M][N]. Escribir el fragmento de código necesario para el envío desde P0 y la recepción en P1 de los datos que se especifican en cada caso, usando para ello un solo mensaje. En caso necesario, definir un tipo de datos derivado MPI.

a) Envío de la tercera fila de la matriz A.
b) Envío de la tercera columna de la matriz A.