SISTEMAS COMPLEJOS EN MÁQUINAS PARALELAS

TRABAJO PRÁCTICO 1

MESSAGE PASSING INTERFACE – PRODUCTO Y SUMA DE VECTORES

Estudiante

FREDY ANDRÉS MERCADO NAVARRO
Pasaporte: 98´773.532
Maestría en Simulación Numérica y Control
Cuatrimestre: I-2012
30 de Abril

Universidad de Buenos Aires
Ciudad Autónoma de Buenos Aires
Argentina
2012

INDICE DE CONTENIDOS

1. PLANTEAMIENTO DEL PROBLEMA

Se requiere implementar un programa en C o C++ que calcule las siguientes funciones:

$$f_1 = a\boldsymbol{X}.\boldsymbol{Y} + b$$

$$\boldsymbol{f_2} = a\boldsymbol{X} + b\boldsymbol{Y}$$

El procesamiento deberá ser en serie y en paralelo. En éste último caso el master se encargará de las operaciones de Input/Output. El resultado deberá ser obtenido en un archivo de texto. NOTA: el tamaño del vector puede ser cualquiera.

2. DETALLES DEL PROGRAMA

El programa vep3.c fue diseñado para calcular los resultados de f_1 y f_2 en forma simultánea e imprimirlos en dos archivos diferentes f1.txt y f2.txt en el mismo directorio que se encuentra el archivo vep3.out.

El usuario del programa deberá indicar el número de procesos, el tamaño de los vectores X e Y que por definición de la suma de vectores y del producto escalar debe ser el mismo, y los valores de los escalares a y b.

A continuación se describirán los aspectos que se consideran relevantes para el objetivo del trabajo a medida que se recorre el código. A modo didáctico, se definió el color anaranjado para el código que ejecutan todos los procesos y azul oscuro para las porciones que solo ejecuta el proceso 0:

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

#define ROOT 0

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

int rank, p, i, m, length;
float a, b, m_local;
double *X, *Y, ppunto, f1, *f2;
double *X_local, *Y_local, ppunto_local=0, *f2_local;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
```

El objetivo es efectuar en forma general, el producto punto de dos vectores y su suma. Para ello lo primero es generar ambos vectores, los cuales se llenarán con números de tipo *float* entre -10 y 10. Lo primero es definir la cantidad de procesos que conforman la máquina de cálculo. Lo segundo, definir el tamaño $length$ de ambos vectores X e Y. Cada uno tendrá entonces m filas por 1 columna. Igualmente, en el código siguiente, se pide el valor de los escalares a y b.

```
//Proceso ROOT genera vectores X e Y
if (rank == ROOT)
```

```
{
 printf("\nPrograma f1 = aX.Y + b\n");
 printf("        f2 = aX + bY\n\n");
 printf("Ingrese tamaño m (X[m,1] Y[m,1]): \n");
 scanf("%d", &length);
 printf("Ingrese a: \n");
 scanf("%f", &a);
 printf("Ingrese b: \n");
 scanf("%f", &b);
```

La siguiente porción de código permite solucionar el problema de que el tamaño de los vectores $length$ no sea divisible en forma exacta por el número de procesos p. Si esto se cumple entonces el tamaño de $length$ es aumentado hasta que la división por p sea exacta. Esto se puede realizar debido a que el resto de los vectores estaría lleno con valores de cero, por lo tanto, no sumarían a los resultados de f_1 ni f_2.

```
 //Si el residuo es != 0 aumento tamaño
 if (length%p != 0)
 {
  m = length + 1;
  while (m%p != 0)
  {
   m = m + 1;
  }
 }
 else
 {
  m = length;
 }

 X = (double *)malloc(m*sizeof(double));
 Y = (double *)malloc(m*sizeof(double));
 f2 = (double *)malloc(m*sizeof(double));
```

Mediante el uso de la función rand() generamos números aleatorios entre -10 y 10. Los vectores X e Y solo se llenan hasta el valor del tamaño seleccionado por el usuario, aunque el tamaño en memoria del vector sea mayor.

```
 //Lleno X e Y hasta X[length] e Y[length]
```

```
 for (i=0; i<length; i++)
 {
 X[i]=(2*((float)rand()/RAND_MAX)-1)*10;
 Y[i]=(2*((float)rand()/RAND_MAX)-1)*10;
 }

 m_local = m/p;
}

MPI_Bcast(&m_local,1,MPI_DOUBLE,ROOT,MPI_COMM_WORLD);
MPI_Bcast(&a,1,MPI_DOUBLE,ROOT,MPI_COMM_WORLD);
MPI_Bcast(&b,1,MPI_DOUBLE,ROOT,MPI_COMM_WORLD);

//Envío a cada proceso la parte que le corresponde
X_local = (double *)malloc(m_local*sizeof(double));
Y_local = (double *)malloc(m_local*sizeof(double));
f2_local = (double *)malloc(m_local*sizeof(double));

MPI_Scatter(X,m_local,MPI_DOUBLE,X_local,m_local,MPI_DOUBLE,ROOT,MPI_COMM_WORL
D);
MPI_Scatter(Y,m_local,MPI_DOUBLE,Y_local,m_local,MPI_DOUBLE,ROOT,MPI_COMM_WORL
D);
```

A partir de este punto todos los procesos computan el producto punto local para f_1 (un escalar) y calculan f_2 local (un vector) directamente. Más tarde, los productos punto locales serán sumados, multiplicados por a y sumados a b, y los vectores f_2 locales serán guardados en orden en f_2 que es el vector global.

```
//Cada proceso computa en paralelo
for (i=0; i<m_local; i++)
{
 ppunto_local += X_local[i] * Y_local[i];
 f2_local[i] = a*X_local[i] + b*Y_local[i];
}
```

Sumo productos punto locales con MPI_Reduce, dado que necesito sumar los valores de ppunto_local de todos los procesos. Esto para f_1. Para f_2 requiero solo guardar cada vector f2_local de cada proceso en orden, así que llamo la función MPI_Gather para formar el vector global f_2 (global).

```
//Sumo productos punto locales con MPI_Reduce
//Armo vector de suma f2 con MPI_Gather
MPI_Reduce(&ppunto_local,&ppunto,1,MPI_DOUBLE,MPI_SUM,ROOT,MPI_COMM_WORLD);
MPI_Gather(f2_local,m_local,MPI_DOUBLE,f2,m_local,MPI_DOUBLE,ROOT,MPI_COMM_WOR
LD);
```

En el siguiente paso se termina de computar el valor de f_1 en el proceso 0 y se imprimen los resultados de f_1 y f_2 en un archivo de texto para cada uno. Adicionalmente, en el programa se incluye la impresión de los vectores X e Y en otro archivo, sin embargo esto solo se hace para que se puedan revisar los cálculos del programa, es decir, para facilitar la revisión. Podría simplemente omitirse. Todo lo descrito lo realiza el proceso 0.

```
if (rank == ROOT)
{
 f1 = a*ppunto + b;

 //Proceso 0 imprime resultados en f1.txt y f2.txt
 FILE *file_pointer;
 file_pointer = fopen("./f1.txt", "w");
 fprintf(file_pointer, "%15.5f\n", f1);
 fclose(file_pointer);

 FILE *file_pointer2;
 file_pointer2 = fopen("./f2.txt", "w");
 for (i=0; i<length; i++)
 {
  fprintf(file_pointer2, "%15.5f\n", f2[i]);
 }
 fclose(file_pointer2);

 FILE *file_pointer3;
 file_pointer3 = fopen("./XY.txt", "w");
 for (i=0; i<length; i++)
 {
 fprintf(file_pointer3, "X[%d]= %15.5f Y[%d]= %15.5f\n",i,X[i],i,Y[i]);
 }
 fclose(file_pointer3);
}
```

Las siguientes líneas de código son para mantenimiento del programa. Imprimen resultados parciales y totales con el objetivo de identificar si las operaciones de comunicación MPI funcionan correctamente. Basta con suprimir los caracteres /* y */ al inicio y al final.

```
/*//Impresión por pantalla - Omitir
if (rank == ROOT)
{
 for (i=0; i<length; i++)
 {
  printf("X %15.3f Y %15.3f\n",X[i],Y[i]);
  printf("rank%d. f2[%d] = %f\n",rank,i,f2[i]);
 }
 printf("rank%d. ppunto = %f\n",rank,ppunto);
 printf("rank%d. f1 = %f\n",rank,f1);
}

MPI_Barrier(MPI_COMM_WORLD);

printf("rank%d. a = %f b = %f\n",rank,a,b);
printf("rank%d. %15.3f %15.3f\n",rank,X_local[0],Y_local[0]);
printf("rank%d. %15.3f %15.3f\n",rank,X_local[1],Y_local[1]);
printf("rank%d. ppunto_local = %f\n",rank,ppunto_local);
*/

MPI_Finalize();

Return 0;

}
```

3. DETALLES DE USO

Las siguientes instrucciones se detallan con el objeto de replicar los mismos pasos realizados por el programador para ejecutar el programa:

- Para compilar el programa, digite en la terminal de Linux (sin $):

$mpicc –o vep3.out vep3.c –lm

- Para ejecutarlo digitar:

$mpirun –np 4 vep3.out

Naturalmente, el número de procesos puede ser diferente de 4. Este número se cambia a gusto del usuario. Al ejecutar, el programa imprimirá en la terminar un mensaje solicitando ingresar el tamaño m de los vectores, donde m es el número de filas de un vector expresado como una matriz de tamaño $m\ x\ 1$ (m filas, 1 columna). Como paso siguiente se piden los valores de a y b.

Realizado esto, el programa se ejecuta. De no haber errores el programa no emite ningún mensaje, solo termina. Allí podemos ir al directorio de ejecución del programa y revisar f1.txt y f2.txt.

Nota: 1. Si asignamos un valor de 1 al número de procesos el programa igual podrá realizar los cálculos. En este caso todas las operaciones serán desarrolladas por el proceso cero (0).

4. CONCLUSIONES

El programa desarrollado calcula satisfactoriamente el producto punto y la suma de vectores utilizando funciones del estándar MPI. Una mejora que se puede introducir al diseño del algoritmo es hallar la manera de dividir la cantidad de datos entre varios procesos aunque la división no sea exacta sin tener que incrementar innecesariamente el tamaño de la memoria asignada a cada vector. Las funciones MPI_Gatherv y MPI_Scatterv podrían ser útiles para éste propósito.

Otro punto a considerar en la elaboración de un programa similar es que generalmente los datos estarían suministrados por una fuente diferente al mismo programa. En este caso los vectores X e Y son generados por el proceso 0 utilizando una función rand() modificada para generar números entre -10 y 10. En un caso más real los datos tendrían que ser leídos de una fuente, como un archivo de texto, y el programa debería adaptarse a ello.

En esta ocasión no se efectuaron mediciones de tiempo comparativas para conocer el verdadero alcance del programa en paralelo con respecto al programa serial. Ese trabajo se deja como una oportunidad a futuro que será implementada en los próximos trabajos prácticos.

El proceso cero tiene a cargo tareas de lectura de datos, generación de vectores y escritura o impresión de datos, mientras que todos los procesos en conjunto solo ejecutan tareas de comunicación y los cálculos que son paralelizables, como:

```
for (i=0; i<m_local; i++)
{
 ppunto_local += X_local[i] * Y_local[i];
 f2_local[i] = a*X_local[i] + b*Y_local[i];
}
```

Esta porción de código ilustrada se pudo ejecutar en todas las máquinas y aprovechar sus resultados para sumarse con los de otros procesos. Como trabajo futuro también se deben tomar medidas de tiempo para conocer qué tan importantes son los tiempos de comunicación de las funciones MPI_Bcast, MPI_Reduce, MPI_Scatter, MPI_Gather, etc, con respecto a los tiempos de cálculo de un ciclo for como el ilustrado en el párrafo anterior.