Informe Deber
"Implementación en paralelo mediante procesos del algoritmo de eliminación de Gauss para resolver $Ax = b$"
Asignatura "Programación en paralelo"

Estudiante doctorado Fausto Fabian Crespo Fernandez

Febrero 2018

1 Resumen del código

El código en lenguaje C usado fue el siguiente:

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

int main(void)
{
    MPI_Init(NULL, NULL);

    int i,j,k;
    const int n=8192;
    int map[n];

    const int total_num_tests=5;

    double sum=0.0;
    double* A;

    double b[n],c[n], x[n];
    A= malloc(n*n*sizeof(double));


    double* A_copia;
    double* b_copia;


    double result_tests[total_num_tests];
    double temp;

```

```
31      int rank;
32      int comm_sz;
33
34      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
35      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
36
37
38      double start, finish, loc_elapsed, elapsed;
39      MPI_Comm comm;
40
41      comm=MPI_COMM_WORLD;
42
43
44
45     for(int num_test=0; num_test <total_num_tests; num_test++) {
46      if (rank==0)
47      {
48
49        A_copia=  malloc(n*n*sizeof(double));
50        b_copia= malloc(n*sizeof(double));
51
52                for (i = 0; i < n; i++){
53                   sum=0.0;
54                   b[i] = (10.0 * (double) rand())/((double) RAND_MAX
      );
55                   b_copia[i]= b[i];
56                   for (j = 0; j < n; j++){
57
58                     A[i*n + j] = (10 * ((double) rand()))/((double)
      RAND_MAX);
59                     A_copia[i*n + j]= A[i*n + j];
60                     sum =sum +A[i*n+j];
61                     if (j == n-1) {
62                        A[i*n+i]=sum ;
63                        A_copia[i*n+i]=A[i*n+i];
64                     }
65
66                   }
67                }
68
69      }
70
71
72
73      MPI_Bcast (A,n*n,MPI_DOUBLE,0,MPI_COMM_WORLD);
74      MPI_Bcast (b,n,MPI_DOUBLE,0,MPI_COMM_WORLD);
75     start = MPI_Wtime();
76      for(i=0; i<n; i++)
77      {
78          map[i]= i % comm_sz;
79      }
80
81      for(k=0;k<n;k++)
82      {
83          MPI_Bcast (&A[k*n + k],n-k,MPI_DOUBLE,map[k],MPI_COMM_WORLD
      );
84      }
```

```
85          MPI_Bcast (&b[k],1,MPI_DOUBLE,map[k],MPI_COMM_WORLD);
86          for(i= k+1; i<n; i++)
87          {
88              if(map[i] == rank)
89              {
90                  c[i]=A[i*n+k]/A[k*n+k];
91                  for(j=k;j<n;j++)
92                  {
93                      A[i*n+j]=A[i*n+j]-( c[i]*A[k*n+j] );
94                  }
95                  b[i]=b[i]-( c[i]*b[k] );
96              }
97          }
98
99      }
100
101
102
103
104
105
106     if (rank==0)
107     {
108         //Si se puede incluir dentro del lazo pero eso solo una
        operacion
109         // y queda m s claro dejarlo fuera del lazo
110         x[n-1]=b[n-1]/A[(n-1)*n + n-1];
111         for(i=n-2;i>=0;i--)
112         {
113             sum=0;
114
115             for(j=i+1;j<n;j++)
116             {
117                 sum=sum+A[i*n+j]*x[j];
118             }
119             x[i]=(b[i]-sum)/A[i*n + i];
120         }
121
122
123     }
124
125     finish = MPI_Wtime();
126     loc_elapsed = finish-start;
127     MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0,
        comm);
128     result_tests[num_test]=elapsed;
129
130     if (rank==0)
131     {
132
133         double res= 0.0;
134         for(i=0;i<n;i++)
135         {   temp=0.0;
136             for(j=0;j<n;j++){
137                 temp=temp+A_copia[i*n + j]*x[j];
138
139             }
```

```
            res+= fabs(b_copia[i]-temp);

        }
        printf("\nEl residuo  es: %.2lf\n", res);


        printf("tamano de la matriz = %d\n", n);
        printf("num procesos = %d\n", comm_sz);
        printf("Tiempo transcurrido experimento= %e(s)\n", elapsed)
    ;

    }


  }

  if(rank==0) {
  sum=0.0;
  for(i=0;i<total_num_tests;i++){
    sum=sum+result_tests[i];
  }

  printf("tamano de la matriz = %d\n", n);
  printf("num procesos = %d\n", comm_sz);
  printf("Tiempo transcurrido promedio = %e(s)\n", sum /
    total_num_tests);
  }

  MPI_Finalize();

  free(A);
  free(A_copia);
  free(b_copia);
  return 0;


}
```

El código análogo en Python es el siguiente:

```
from mpi4py import MPI
import time
import numpy as np

def lectura_datos(file1):
    with open(file1) as f:
        result = f.readlines()
    result  = [x.strip() for x in result]
    return result

# Programa principal
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
print('Inicia proceso {} de {}\n'.format(rank, size))

total_num_tests=5
map=[]
```

```
b=[]
c=[]
x=[]
A=[]
A_temp=[]
b_temp=[]
result_tests=np.array([0]*total_num_tests)
max_random_int=10
file1='datosGauss.txt'
datos=lectura_datos(file1)
datos = filter(None, datos)
for ns in datos:
  n=int(ns)
  map=np.array([0.0]*n)
  c=np.array([0.0]*n)
  x=np.array([0.0]*n)
  b=np.array([0.0]*n)
  A=np.array([0.0]*n*n)

  for num_test in range(0,total_num_tests):
    if rank==0:
      A = max_random_int * np.random.rand(n*n)
      for i in range(0,n):
            A[i*n+i]=np.sum(A[i*n:(i+1)*n-1])

    comm.barrier()
    comm.Bcast(A, root=0)
    comm.Bcast(b, root=0)

    for i in range(n):
      map[i]= i % size

    start= time.time()
    for k in range(0,n):
      comm.Bcast(A[k*n + k:], root=map[k])
      comm.Bcast(b[k:], root=map[k])
      for i in range(k+1,n):
          if map[i]== rank:
             c[i]=A[i*n+k]/A[k*n+k]
             A[i*n+k:i*n+n-1]=A[i*n+k:i*n+n-1]-( c[i]*A[k*n+k:k*n+n
    -1])
             b[i]=b[i]-( c[i]*b[k] );


    if rank==0:
        x[n-1]=b[n-1]/A[(n-1)*n + n-1]
        for i in list(reversed(range(0,n-1))):
            sum=0
            sum=np.sum(A[i*n+i+1:i*n+n-1]*x[i+1:n-1])
            x[i]=(b[i]-sum)/A[i*n + i]

    end = time.time()
    tiempo = end - start

    tiempo = comm.allreduce(tiempo, op=MPI.MAX)
    result_tests[num_test]=tiempo

```

```
75
76  if rank==0:
77        print('tamano de la matrix  =: {:.0f}\n'.format(n))
78        print('num procesos =: {:.0f}\n'.format(size))
79        print('Tiempo transcurrido promedio: {:.8f}\n'.format(np.
    average(result_tests)))
80        temp=A.reshape(n,n)
81        print('suma componentes de residuos: {:.8f}\n'.format(np.
    linalg.norm(np.dot(temp, x)-b)))
```

En ambos códigos, el proceso 0 genera aleatoriamente la matriz A de dimensión $n \times n$ de manera que sea diagonal dominante y para esto, se le asigna a cada elemento de la diagonal principal la suma de todos los elementos de la fila(que son generados aleatoriamente entre 1 al 10).También se genera aleatoriamente el arreglo lineal b y se reserva espacio para el arreglo solución x y un arreglo temporal c de dimensión n.

Luego que se generan la matriz A y el vector b se hacen dos Broadcast desde el proceso 0 a los demás procesos: el primero desde el proceso 0 y en el cual se envía un puntero a la matriz A generada y otro Broadcast desde el proceso 0 con un puntero al vector b generado .

Posteriormente los procesos inicializan un vector local map de dimensión n, que contiene los números del 0 a la cantidad de procesos -1 ($commSize - 1$) y que identifica el número del proceso que debe transformar cada fila.

Luego en cada iteración k del método de Eliminación de Gauss, se envían dos Broadcast desde el proceso que le corresponde transformar la fila $k(map[k])$: el primero con el puntero al primer elemento no 0 de la fila k(fila pivote para el paso k y que incluye $n - k$ elementos distintos de 0) de la matriz A y el otro Broadcast con el puntero al elemento de la fila k del vector b. Estos Broadcasts, a su vez, tienen el efecto de barrera ya que los procesos esperan que le llegue la información de la fila pivote para poder transformar desde la fila $k + 1$ en adelante, las filas que les corresponde modificar según el arreglo map .

Los multiplicadores por los cuales hay que multiplicar las filas pivote, se almacenan en el arreglo c: $c[i] = A[i][k]/A[k][k]$ y luego con ellos, los procesos actualizan las filas de la matriz A: $A[i*n+j] = A[i*n+j] - (c[i]*A[k*n+j])$; y del vector b:$b[i] = b[i] - (c[i]*b[k])$; según que el arreglo map indique que esa fila debe ser actualizada por ese proceso($if(map[i] == rank)$).

Luego que la matriz están en forma triangular superior, el proceso 0 realiza la sustitución hacia atrás.

Cada proceso tiene un tiempo de procesamiento, y posteriormente se calcula el máximo de estos tiempos, y se guarda dicho tiempo en la posición adecuada

de un arreglo de tamaño 5(pues se ejecutan 5 pruebas por cada valor de n).

En las especificación del deber se pedía leer de la consola el tamaño n pero en este caso se optó, para evitar el código de verificación de que el n ingresado sea múltiplo de p, por no implementar dicha opción de entrada y establecer el n y p potencias de 2 de forma tal que n fuera múltiplo de p.

Los valores usados para el código en C fueron $512, 1024, 2048, 4096, 8192$.

En el caso del código en Python los valores de n se leen del archivo de texto datosGauss.txt donde en cada línea está un valor para n. Los valores usados para el código en Python fueron $769, 1024, 2048, 4096$(no se incluyo el valor 512 pues los tiempo era aproximadamente 0.00 lo que hacia imposible estimar la aceleración y la eficiencia. Por otro lado el valor de 8192 pues el tiempo de ejecución era muy elevado)

Un detalle importante en la implementación en Python es que se usó las función Bcast en vez de bcast pues se esta enviando a otros procesos, objetos de la librería numpy en vez de tipos primitivos.

2 Tablas con tiempos de cálculo, aceleración y eficiencia

Se tiene que $T_{paralelo}(n,p) = T_{serial}(n)/p + T_{overhead=comunicación/procesos}(n,p)$ y el tiempo serial del algoritmo de eliminación de Gauss es $O(n^3)$. Cuando n es grande la parte dominante es el primer término de la derecha pues la parte de la comunicación entre los procesos tiende a 0.

La aceleración se mide como:

$S(n,p) = \frac{T_{serial}(n)}{T_{paralelo}(n,p)}$

y lo ideal es que $S(n,p) = p$.

La eficiencia se mide como $E(n,p) = \frac{S(n,p)}{p}$

En las tablas para la implementación en C, se observa que a medida que duplicamos el tamaño del problema, hacia abajo por las filas, el tiempo de ejecución

aumenta en cada columna. En las columnas, el caso ideal: donde la aceleración $S(n,p) = p$ casi nunca se alcanza excepto en: $n = 512, p = 2$ donde la eficiencia es muy cercana a 1 para la implementación en C(para relativamente pocos procesadores) y en ningún caso se alcanza eficiencia de 1 con la implementación en Python. Se observa que a medida que aumenta el tamaño del problema(donde tenemos muchos procesadores y pesa el tiempo de comunicación entre ellos respecto al tiempo total) la aceleración aumenta pero tiende a un valor asintótico que se aleja del valor ideal p. En general para p grande y n pequeña la aceleración es mucho menor que p. Sin embargo para p pequeña y n grande la aceleración es casi la ideal.

Analizando los resultados con el código en Python vemos que en las filas por lo general, para un mismo problema(n fijo), aumentan los tiempos de ejecución lo que sugiere que no es conveniente aumentar la cantidad de procesos ya que se emplea mucho tiempo en la comunicación entre los mismos(se hacen 2 Broadcasts en cada iteración del ciclo for de la linea 52(o sea se hacen $2n$ Broadcast en total). Por el contrario en el código en C, si es conveniente aumentar la cantidad de procesos pues los tiempos de ejecución disminuyen en las filas.

Si comparamos los tiempos entre la implementación en Python y los de la implementación de C, para $n = 1024, 2048, 4096$ con 1 y varios procesos, el tiempo de ejecución del código en Python es siempre mayor al del código en C. Esto es porque no se usa mucho el código optimizado de numpy para el Método de Gauss, excepto en algunas operaciones vectorizadas, como las de las lineas 58 y 66.

Método Gauss resolver Ax=b con C++

Tiempos de ejecución (apollo-1)

	1	2	4	8	16
512	0,23	0,12	0,07	0,04	0,03
1024	1,39	0,84	0,43	0,23	0,13
2048	11,14	6,66	3,32	1,76	0,91
4096	97,79	51,75	26,17	13,53	6,98
8192	770,47	409,40	206,33	106,06	56,19

Aceleración (Speedup:= Tserial / Tparalelo)

	1	2	4	8	16
512	1	1,9	3,5	5,4	9,2
1024	1	1,7	3,2	6,1	10,8
2048	1	1,7	3,4	6,3	12,2
4096	1	1,9	3,7	7,2	14,0
8192	1	1,9	3,7	7,3	13,7

Eficiencia (Efficiency:= Speedup / p)

	1	2	4	8	16
512	1	1,0	0,9	0,7	0,6
1024	1	0,8	0,8	0,8	0,7
2048	1	0,8	0,8	0,8	0,8
4096	1	0,9	0,9	0,9	0,9
8192	1	0,9	0,9	0,9	0,9

Método Gauss resolver Ax=b con Python

Tiempos de ejecución (apollo-1)

	1	2	4	8	16
768	1,00	1,00	1,00	1,00	1,20
1024	3,00	2,00	3,00	3,00	4,00
2048	13,00	12,20	12,00	25,00	33,00
4096	86,00	120,00	199,00	191,20	259,00

Aceleración (Speedup:= Tserial / Tparalelo)

	1	2	4	8	16
512	1	1,0	1,0	1,0	0,8
1024	1	1,5	1,0	1,0	0,8
2048	1	1,1	1,1	0,5	0,4
4096	1	0,7	0,4	0,4	0,3

Eficiencia (Efficiency:= Speedup / p)

	1	2	4	8	16
512	1	0,5	0,3	0,1	0,1
1024	1	0,8	0,3	0,1	0,0
2048	1	0,5	0,3	0,1	0,0
4096	1	0,4	0,1	0,1	0,0

3 Escalabilidad del algoritmo

Un algoritmo es escalable si, el tamaño del problema se puede incrementar a una tasa tal que la eficiencia no disminuya, mientras el número de procesos se incrementa. Si se mantiene una eficiencia constante, sin incrementar el tamaño del problema se dice fuertemente escalable y si el programa mantiene una eficiencia constante, cuando el tamaño del problema aumenta a la misma tasa del aumento del número de procesos, es débilmente escalable.

Al observar la tabla de eficiencia para el código en C, vemos que el programa no es fuertemente escalable, pues la eficiencia no se mantiene constante sin aumentar el tamaño del problema.

Si duplicamos el tamaño del problema y duplicamos los recursos para el código en C, o sea analizamos las diagonales(resaltadas con el mismo color) vemos que la eficiencia no sólo no se mantiene constante, sino que puede disminuir ligeramente. Por tanto el algoritmo no es fuertemente escalable. Pero consideramos que el código en C es débilmente escalable debido a que las variaciones de la eficiencia a lo largo de las diagonales, no es muy grande.

Sin embargo, para el código en Python, la eficiencia disminuye drásticamente en las diagonales y ni siquiera es débilmente escalable, o sea en este caso no es conveniente aumentar el número de procesos a medida que aumenta el tamaño del problema.

Como se menciono anteriormente, el gran número de Broadcasts o comunicación entre procesos en cada iteración, combinado con el débil uso de las optimizaciones de código en la librería numpy, hace que el código en Python sea más lento que en C, y no escalable.

4 Bibliografía

1- Pacheco T. Introduction to Parallel Programming
2- https://stackoverflow.com/questions/25236369/c-parallel-implementation-of-gauss-elimination-with-mpi.