

Algoritmos Paralelos

Grimaldo José Dávila Guillén

April 10, 2017

Contents

1	Regla del trapecio	2
2	MPI Scatter y MPI Gather	3
2.1	MPI Scatter	3
2.2	MPI Gather	3
3	Multiplicación Matriz-vector con MPI Allgather	4

1 Regla del trapecio

La regla del trapecio sirve para calcular el área bajo la curva de una función. Para realizar este informe lo que se realizó fue una implementación de este método en forma paralelizada utilizando MPI, las funciones implementadas para este método fueron:

- float f(float x)
- float trapecio(float local_a, float local_b, int local_n, float h)
- void Get_data(int my_rank, int p, float *a_ptr, float *b_ptr, int *n_ptr)

Donde la función f viene a ser la función que vamos a integrar, trapecio es la función que calculará el área bajo un rango de la curva dividiendo el área en trapecios y Get_data es la función que se encargará de enviar y recibir los rangos o límites de integración de cada sub área que recibirá cada proceso.

En la figura 1 se ve la paralelización del algoritmo de la regla del trapecio. A continuación describiremos el código utilizado.

Figure 1: Regla del Trapecio paralelizada.

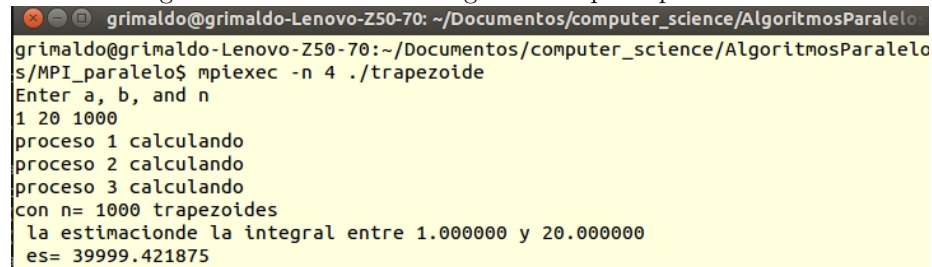
```
64 int main(int argc, char **argv)
65 {
66     int my_rank, p, n, local_n, source, dest=0, tag=50;
67     float a, b, h, local_a, local_b, integral, total;
68     MPI_Status status;
69     MPI_Init(&argc, &argv);
70     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
71     MPI_Comm_size(MPI_COMM_WORLD, &p);
72     Get_data(my_rank, p, &a, &b, &n);
73     h = (b - a) / n; local_n = n / p;
74     local_a = a + my_rank * local_n * h;
75     local_b = local_a + local_n * h;
76     integral = trapecio(local_a, local_b, local_n, h);
77     if(my_rank == 0) {
78         total = integral;
79         for(source = 1; source < p; source++) {
80             MPI_Recv(&integral, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);
81             total += integral;
82             printf("proceso %d calculo un total de %f\n", source, total);
83         }
84         printf("con n= %d trapezoides\n la estimacion", n);
85         printf("de la integral entre %f y %f\n es= %f \n", a, b, total);
86     }
87     else {
88         MPI_Send(&integral, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
89     }
90     MPI_Finalize();
91 }
```

De la línea 69 a 71 se inicializa y obtiene los valores tanto del número de procesos como los identificadores de los procesos que se crearán en el programa,

Línea 73 a 76 obtenemos el ancho de los trapecios y calculamos el área de los trapecios que se repartirán en los procesos creados, Línea 77 a 90 si es proceso 0 recibirá todas las áreas calculadas de los trapecios por los demás procesos. Si es diferente de 0 lo único que hará será enviar el área que calculo al proceso 0.

En la figura 2 podemos ver como diferentes procesos calculan una parte del área de la función.

Figure 2: resultados de la regla del Trapecio paralelizada.



```
grimaldo@grimaldo-Lenovo-Z50-70: ~/Documentos/computer_science/AlgoritmosParalelos
grimaldo@grimaldo-Lenovo-Z50-70:~/Documentos/computer_science/AlgoritmosParalelos$ mpiexec -n 4 ./trapezoide
Enter a, b, and n
1 20 1000
proceso 1 calculando
proceso 2 calculando
proceso 3 calculando
con n= 1000 trapecios
la estimacion de la integral entre 1.000000 y 20.000000
es= 39999.421875
```

2 MPI Scatter y MPI Gather

2.1 MPI Scatter

MPI Scatter lo que hace es repartir un vector entre todos los procesos de tal manera que las tareas asociadas a ese vector se pueden repartir ahorrando tiempo a diferencia de otros métodos este solo utiliza memoria del bloque que utilizara por lo tanto no habrá desperdicio de espacio en la memoria.

En la figura 3 podemos ver el código del programa que describiremos a continuación.

Lo importante en esta función lo encontramos en la línea 20 que podemos ver como MPI_Scatter lo que es repartir el vector en bloques y los enviar en la variable local_a, todo esto es si es el proceso 0 debido que en esta parte en la línea 16 y 16 recién llenamos el vector original. En las líneas 27 a 32 podemos ver como a todos los procesos que no son 0 se encargan de recibir todos los bloques del vector.

2.2 MPI Gather

MPI Gather lo que hace es unir bloques separados de un vector en un solo vector. Esto puede ser utilizado junto a MPI Scatter para poder realizar operaciones mas rápido a vectores

En la figura 4 podemos ver el código del programa que describiremos a continuación.

Figure 3: Lectura de un vector con MPI Scatter

```

9 void Read_vector(double local_a[], int local_n, int n, char vec_name[], int my_rank, MPI_Comm comm){
10     double* a = NULL;
11     int i;
12
13     if(my_rank == 0){
14         a = malloc(n*sizeof(double));
15         printf("Ingrese los valores del vector %s\n", vec_name);
16         for(i = 0; i < n; i++){
17             scanf("%lf", &a[i]);
18         }
19         printf("Vector dividido en \n");
20         MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
21         for(i = 0; i < local_n; i++){
22             printf("%f ", local_a[i]);
23         } printf("\n ");
24         free(a);
25     }
26
27     else{
28         MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
29         for(i = 0; i < local_n; i++){
30             printf("%f ", local_a[i]);
31         } printf("\n ");
32     }
33 }

```

En la figura 5 podemos ver los resultados de ejecutar MPI Scatter y MPI Gather. Podemos notar que con MPI Scatter el vector inicial fue dividido en 4 bloques de 2 elementos cada uno como se muestra en la imagen y con MPI Gather podemos notar que dichos bloques fueron unidos nuevamente en un solo vector.

3 Multiplicación Matriz-vector con MPI Allgather

Esta función nos permite almacenar valores de operaciones realizadas en distintos procesos como multiplicación de una matriz con un vector como en el ejemplo de la figura 6 , 7

En la figura 8 los resultados de ejecutar el programa. Podemos ver como las multiplicación se repartieron en los 4 procesos creados y después los resultados de la multiplicación en un vector resultante.

Figure 4: Recolección de un vector con MPI Scatter

```

35 void Print_vector(double local_b[], int local_n, int n, char title[], int my_rank, MPI_Comm comm){
36     double* b = NULL;
37     int i;
38     if(my_rank == 0){
39         b = malloc(n*sizeof(double));
40         MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
41         printf("%s\n", title);
42         for(i = 0; i < n; i++){
43             printf("%f ", b[i]);
44         }
45         printf("\n");
46         free(b);
47     }
48     else{
49         MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
50     }
51 }

```

Figure 5: Resultados con MPI Scatter y MPI Gather

```

grimaldo@grimaldo-Lenovo-Z50-70:~/Documentos/computer_science/AlgoritmosParalelos/MPI_paralelos$
mpitexec -n 4 ./distrib_vec
Ingrese los valores del vector Vector inicial
1 2 3 4 5 6 7 8 9 0
Vector dividido en
1.000000 2.000000
3.000000 4.000000
7.000000 8.000000
5.000000 6.000000
Vector reconstruido
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000 0.000000

```

Figure 6: Código de multiplicación Matriz-vector con MPI Allgather

```

47 ~ void Mat_vect_mult(double local_A[], double local_x[], double local_y[], int local_m, int n, int local_n, MPI_Comm comm){
48     double* x;
49     int local_i, j;
50     int local_ok=1;
51     x = malloc(n*sizeof(double));
52     MPI_Allgather(local_x, local_n, MPI_DOUBLE, x, local_n, MPI_DOUBLE, comm);
53 ~ for(local_i = 0; local_i < local_m; local_i++){
54     local_y[local_i] = 0.0;
55 ~ for(j = 0; j < n; j++){
56     local_y[local_i] += local_A[local_i*n+j] * x[j];
57     }
58 }
59 free(x);
60 }
61

```

Figure 7: Código de multiplicación Matriz-vector con MPI Allgather en la función main

```

62 int main(int argc, char **argv){
63     int comm_sz;
64     int my_rank;
65     double start, finish;
66
67     int m=4; ///filas
68     int n=4; ///columnas
69     int local_n, local_m;
70     MPI_Init(&argc, &argv);
71     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
72     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
73     local_n = n/comm_sz; /*Divide Matriz en bloques para los n procesos*/
74     local_m = m/comm_sz;
75     double local_x[local_n], local_y[local_m], local_A[local_n*local_m];
76     Read_vector(local_A, local_n+local_m, n*m, "A", my_rank, MPI_COMM_WORLD);
77     Read_vector(local_x, local_n, n, "X", my_rank, MPI_COMM_WORLD);
78
79
80     ///tomar tiempo de ejecucion
81     double tInicio, tFin;
82     double MPI_Wtime(void);
83     tInicio = MPI_Wtime();
84     Mat_vect_mult(local_A, local_x, local_y, local_m, n, local_n, MPI_COMM_WORLD);
85     tFin = MPI_Wtime();
86     Print_vector(local_y, local_m, m, "Y", my_rank, MPI_COMM_WORLD);
87     printf("el tiempo transcurrido es %f \n", tFin-tInicio);
88     MPI_Finalize();
89     return 0;
90 }

```

Figure 8: Resultados de la multiplicación Matriz-vector con MPI Allgather en la función main

```

grimaldo@grimaldo-Lenovo-Z50-70: ~/Documentos/computer_science/AlgoritmosParalelo
s/MPI_paralelo$ mpirun -np 4 ./multi_matr_vec
Enter the vector A
12 43 6 5
54 6 8 7
23 67 90 6
32 34 5 67
Enter the vector X
23 5 9 12
Y
4910.000000 1630.000000 12720.000000 2190.000000
el tiempo transcurrido es 0.000043
el tiempo transcurrido es 0.000042
el tiempo transcurrido es 0.000191
el tiempo transcurrido es 0.000183

```