Universidad Nacional de San Agustin

Escuela Profesional de
Ciencia de la Computación

Algoritmos Paralelos

MPI

Alumna :
Rosa Yuliana Gabriela
Paccotacya Yanque
Profesor:
Mg.Alvaro Henry Mamani
Aliaga

Índice

1. Regla Trapezoidal

Este programa usa MPI para implementar la versión paralela de la regla trapezoidal, estima la integral de a a b de una funcion $f(x)$ usando n trapezoides.

1.1. Algoritmo

1. Cada proceso calcula su intervalo de integración
2. Cada proceso estima la integral de $f(x)$ sobre su intervalo usando la regla trapezoidal
3. Todos los procesos diferentes de rango 0 mandan su integral al proceso 0.
4. El proceso 0 suma todo lo recibido e imprime el resultado.

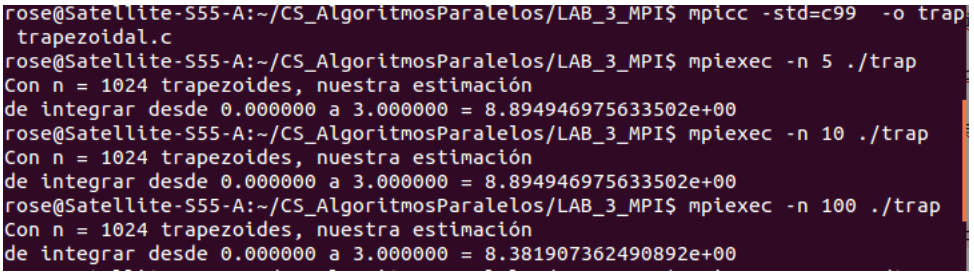
1.2. Código

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

int main(void) {
   int my_rank, comm_sz, n = 1024, local_n;
   double a = 0.0, b = 3.0, h, local_a, local_b;
   double local_int, total_int;
   int source;

   MPI_Init(NULL, NULL);

   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   h = (b-a)/n;
   local_n = n/comm_sz;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   local_int = Trap(local_a, local_b, local_n, h);


```

```
23     if (my_rank != 0) {
24        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
25              MPI_COMM_WORLD);
26     } else {
27        total_int = local_int;
28        for (source = 1; source < comm_sz; source++) {
29           MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
30              MPI_COMM_WORLD, MPI_STATUS_IGNORE);
31           total_int += local_int;
32        }
33     }
34
35
36     if (my_rank == 0) {
37        printf("Con n = %d trapezoides, nuestra estimación\n", n);
38        printf("de integrar desde %f a %f = %.15e\n",
39            a, b, total_int);
40     }
41
42
43     MPI_Finalize();
44
45     return 0;
46  }
47  double Trap(
48        double left_endpt
49        double right_endpt
50        int    trap_count
51        double base_len
52     double estimate, x;
53     int i;
54
55     estimate = (f(left_endpt) + f(right_endpt))/2.0;
56     for (i = 1; i <= trap_count-1; i++) {
57        x = left_endpt + i*base_len;
58        estimate += f(x);
59     }
60     estimate = estimate*base_len;
61
62     return estimate;
```

```
63 }
64
65
66 double f(double x) {
67    return x*x;
68 }
```

1.3. Output



2. Distribución y lectura de los datos de un vector

Estos programas leen e imprimen un vector entre los procesos usando una distribución de bloques.

2.1. Scatter

MPIscatter puede ser usada en una función que lee un vector entero en el proceso 0, pero solo manda los componentes necesarios a los otros procesos

```
167 void Read_vector(
168       char      prompt[]     /* in  */,
169       double    local_vec[]  /* out */,
170       int       n            /* in  */,
171       int       local_n      /* in  */,
172       int       my_rank      /* in  */,
173       MPI_Comm  comm         /* in  */) {
174    double* vec = NULL;
175    int i, local_ok = 1;
```

```
176
177     if (my_rank == 0) {
178        vec = malloc(n*sizeof(double));
179        if (vec == NULL) local_ok = 0;
180        Check_for_error(local_ok, "Read_vector",
181              "No se puede alojar temporary vector", comm);
182        printf("Ingrese vector %s\n", prompt);
183        for (i = 0; i < n; i++)
184           scanf("%lf", &vec[i]);
185        MPI_Scatter(vec, local_n, MPI_DOUBLE,
186              local_vec, local_n, MPI_DOUBLE, 0, comm);
187        free(vec);
188     } else {
189        Check_for_error(local_ok, "Read_vector",
190              "No se puede alojar temporary vector", comm);
191        MPI_Scatter(vec, local_n, MPI_DOUBLE,
192              local_vec, local_n, MPI_DOUBLE, 0, comm);
193     }
194  }  /* Read_vector */
195
```

2.2. Gather

Recolecta todos los componentes del vector sobre el proceso 0.

```
231  void Print_vector(
232        char      title[]     /* in */,
233        double    local_vec[] /* in */,
234        int       n           /* in */,
235        int       local_n     /* in */,
236        int       my_rank     /* in */,
237        MPI_Comm  comm        /* in */) {
238     double* vec = NULL;
239     int i, local_ok = 1;
240
241     if (my_rank == 0) {
242        vec = malloc(n*sizeof(double));
243        if (vec == NULL) local_ok = 0;
244        Check_for_error(local_ok, "Print_vector",
245              "No se puede alojar temporary vector", comm);
```

```
246         MPI_Gather(local_vec, local_n, MPI_DOUBLE,
247               vec, local_n, MPI_DOUBLE, 0, comm);
248         printf("\nEl vector %s\n", title);
249         for (i = 0; i < n; i++)
250            printf("%f ", vec[i]);
251         printf("\n");
252         free(vec);
253      }  else {
254         Check_for_error(local_ok, "Print_vector",
255               "Can't allocate temporary vector", comm);
256         MPI_Gather(local_vec, local_n, MPI_DOUBLE,
257               vec, local_n, MPI_DOUBLE, 0, comm);
258      }
259   }  /* Print_vector */
260
```

2.3. Output

```
Ingrese vector x
1
2
3
4
5
6
7
8
9
10
11
12

El vector x
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
 10.000000 11.000000 12.000000
```

3. Multiplicación de matriz con vector

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  void Check_for_error(int local_ok, char fname[], char message[],
6        MPI_Comm comm);
```

```
7  void Get_dims(int* m_p, int* local_m_p, int* n_p, int* local_n_p,
8        int my_rank, int comm_sz, MPI_Comm comm);
9  void Allocate_arrays(double** local_A_pp, double** local_x_pp,
10       double** local_y_pp, int local_m, int n, int local_n,
11       MPI_Comm comm);
12 void Read_matrix(char prompt[], double local_A[], int m, int local_m,
13       int n, int my_rank, MPI_Comm comm);
14 void Read_vector(char prompt[], double local_vec[], int n, int local_n,
15       int my_rank, MPI_Comm comm);
16 void Print_matrix(char title[], double local_A[], int m, int local_m,
17       int n, int my_rank, MPI_Comm comm);
18 void Print_vector(char title[], double local_vec[], int n,
19       int local_n, int my_rank, MPI_Comm comm);
20 void Mat_vect_mult(double local_A[], double local_x[],
21       double local_y[], int local_m, int n, int local_n,
22       MPI_Comm comm);
23
24 /*--------------------------------------------------------------------*/
25 int main(void) {
26    double* local_A;
27    double* local_x;
28    double* local_y;
29    int m, local_m, n, local_n;
30    int my_rank, comm_sz;
31    MPI_Comm comm;
32
33    MPI_Init(NULL, NULL);
34    comm = MPI_COMM_WORLD;
35    MPI_Comm_size(comm, &comm_sz);
36    MPI_Comm_rank(comm, &my_rank);
37
38    Get_dims(&m, &local_m, &n, &local_n, my_rank, comm_sz, comm);
39    Allocate_arrays(&local_A, &local_x, &local_y, local_m, n, local_n, comm);
40    Read_matrix("A", local_A, m, local_m, n, my_rank, comm);
41 #  ifdef DEBUG
42    Print_matrix("A", local_A, m, local_m, n, my_rank, comm);
43 #  endif
44    Read_vector("x", local_x, n, local_n, my_rank, comm);
45 #  ifdef DEBUG
46    Print_vector("x", local_x, n, local_n, my_rank, comm);
```

```
47  #  endif
48
49     Mat_vect_mult(local_A, local_x, local_y, local_m, n, local_n, comm);
50
51     Print_vector("y", local_y, m, local_m, my_rank, comm);
52
53     free(local_A);
54     free(local_x);
55     free(local_y);
56     MPI_Finalize();
57     return 0;
58  }  /* main */
59
60  void Check_for_error(
61        int        local_ok   /* in */,//1 si hay error
62        char       fname[]    /* in */,
63        char       message[]  /* in */,
64        MPI_Comm   comm       /* in */) {
65     int ok;
66
67     MPI_Allreduce(&local_ok, &ok, 1, MPI_INT, MPI_MIN, comm);
68     if (ok == 0) {
69        int my_rank;
70        MPI_Comm_rank(comm, &my_rank);
71        if (my_rank == 0) {
72           fprintf(stderr, "Proc %d > In %s, %s\n", my_rank, fname,
73                 message);
74           fflush(stderr);
75        }
76        MPI_Finalize();
77        exit(-1);
78     }
79  }  /* Check_for_error */
80
81
82  //Matriz A de mxn
83  void Get_dims(
84        int*       m_p        /* out */,
85        int*       local_m_p  /* out */,
86        int*       n_p        /* out */,
```

```
87        int*      local_n_p  /* out */,
88        int       my_rank    /* in  */, //proceso llamante
89        int       comm_sz    /* in  */,// numero de procesos
90        MPI_Comm  comm       /* in  */) {
91     int local_ok = 1;
92
93     if (my_rank == 0) {
94        printf("Ingrese numero de filas\n");
95        scanf("%d", m_p);
96        printf("Ingrese numero de columnas\n");
97        scanf("%d", n_p);
98     }
99     MPI_Bcast(m_p, 1, MPI_INT, 0, comm);
100    MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
101    if (*m_p <= 0 || *n_p <= 0 || *m_p % comm_sz != 0
102          || *n_p % comm_sz != 0) local_ok = 0;
103    Check_for_error(local_ok, "Get_dims",
104       "m y n deben ser positivos y divisibles por comm_sz",
105       comm);
106
107    *local_m_p = *m_p/comm_sz;
108    *local_n_p = *n_p/comm_sz;
109 }  /* Get_dims */
110
111
112 void Allocate_arrays(
113       double**  local_A_pp  /* out */, //matriz (m/comm_sz , n )
114       double**  local_x_pp  /* out */, //x (n/comm_sz)
115       double**  local_y_pp  /* out */, //y (n/comm_sz)
116       int       local_m     /* in  */, //A y
117       int       n           /* in  */,   //x
118       int       local_n     /* in  */,
119       MPI_Comm  comm        /* in  */) {
120
121    int local_ok = 1;
122
123    *local_A_pp = malloc(local_m*n*sizeof(double));
124    *local_x_pp = malloc(local_n*sizeof(double));
125    *local_y_pp = malloc(local_m*sizeof(double));
126
```

```
127     if (*local_A_pp == NULL || local_x_pp == NULL ||
128           local_y_pp == NULL) local_ok = 0;
129     Check_for_error(local_ok, "Allocate_arrays",
130           "No se puede asignar local arrays", comm);
131  }  /* Allocate_arrays */
132
133
134  void Read_matrix(
135        char      prompt[]   /* in  */,
136        double    local_A[]  /* out */,
137        int       m          /* in  */,
138        int       local_m    /* in  */,
139        int       n          /* in  */,
140        int       my_rank    /* in  */,
141        MPI_Comm  comm       /* in  */) {
142     double* A = NULL;
143     int local_ok = 1;
144     int i, j;
145
146     if (my_rank == 0) {
147        A = malloc(m*n*sizeof(double));
148        if (A == NULL) local_ok = 0;
149        Check_for_error(local_ok, "Read_matrix",
150              "No se puede alojartemporary matrix", comm);
151        printf("Ingrese matriz %s\n", prompt);
152        for (i = 0; i < m; i++)
153           for (j = 0; j < n; j++)
154              scanf("%lf", &A[i*n+j]);
155        MPI_Scatter(A, local_m*n, MPI_DOUBLE,
156              local_A, local_m*n, MPI_DOUBLE, 0, comm);
157        free(A);
158     } else {
159        Check_for_error(local_ok, "Read_matrix",
160              "No se puede alojar temporary matrix", comm);
161        MPI_Scatter(A, local_m*n, MPI_DOUBLE,
162              local_A, local_m*n, MPI_DOUBLE, 0, comm);
163     }
164  }  /* Read_matrix */
165
166
```

```
167 void Read_vector(
168       char      prompt[]     /* in  */,
169       double    local_vec[]  /* out */,
170       int       n            /* in  */,
171       int       local_n      /* in  */,
172       int       my_rank      /* in  */,
173       MPI_Comm  comm         /* in  */) {
174    double* vec = NULL;
175    int i, local_ok = 1;
176 
177    if (my_rank == 0) {
178       vec = malloc(n*sizeof(double));
179       if (vec == NULL) local_ok = 0;
180       Check_for_error(local_ok, "Read_vector",
181             "No se puede alojar temporary vector", comm);
182       printf("Ingrese vector %s\n", prompt);
183       for (i = 0; i < n; i++)
184          scanf("%lf", &vec[i]);
185       MPI_Scatter(vec, local_n, MPI_DOUBLE,
186             local_vec, local_n, MPI_DOUBLE, 0, comm);
187       free(vec);
188    } else {
189       Check_for_error(local_ok, "Read_vector",
190             "No se puede alojar temporary vector", comm);
191       MPI_Scatter(vec, local_n, MPI_DOUBLE,
192             local_vec, local_n, MPI_DOUBLE, 0, comm);
193    }
194 }  /* Read_vector */
195 
196 
197 void Print_matrix(
198       char      title[]    /* in */,
199       double    local_A[]  /* in */,
200       int       m          /* in */,
201       int       local_m    /* in */,
202       int       n          /* in */,
203       int       my_rank    /* in */,
204       MPI_Comm  comm       /* in */) {
205    double* A = NULL;
206    int i, j, local_ok = 1;
```

```
207
208     if (my_rank == 0) {
209        A = malloc(m*n*sizeof(double));
210        if (A == NULL) local_ok = 0;
211        Check_for_error(local_ok, "Print_matrix",
212              "No se puede alojar temporary matrix", comm);
213        MPI_Gather(local_A, local_m*n, MPI_DOUBLE,
214              A, local_m*n, MPI_DOUBLE, 0, comm);
215        printf("\nLa matriz %s\n", title);
216        for (i = 0; i < m; i++) {
217           for (j = 0; j < n; j++)
218              printf("%f ", A[i*n+j]);
219           printf("\n");
220        }
221        printf("\n");
222        free(A);
223     } else {
224        Check_for_error(local_ok, "Print_matrix",
225              "No se puede alojar temporary matrix", comm);
226        MPI_Gather(local_A, local_m*n, MPI_DOUBLE,
227              A, local_m*n, MPI_DOUBLE, 0, comm);
228     }
229  }  /* Print_matrix */
230
231  void Print_vector(
232        char      title[]     /* in */,
233        double    local_vec[] /* in */,
234        int       n           /* in */,
235        int       local_n     /* in */,
236        int       my_rank     /* in */,
237        MPI_Comm  comm        /* in */) {
238     double* vec = NULL;
239     int i, local_ok = 1;
240
241     if (my_rank == 0) {
242        vec = malloc(n*sizeof(double));
243        if (vec == NULL) local_ok = 0;
244        Check_for_error(local_ok, "Print_vector",
245              "No se puede alojar temporary vector", comm);
246        MPI_Gather(local_vec, local_n, MPI_DOUBLE,
```

```
247             vec, local_n, MPI_DOUBLE, 0, comm);
248       printf("\nEl vector %s\n", title);
249       for (i = 0; i < n; i++)
250          printf("%f ", vec[i]);
251       printf("\n");
252       free(vec);
253    }  else {
254       Check_for_error(local_ok, "Print_vector",
255             "Can't allocate temporary vector", comm);
256       MPI_Gather(local_vec, local_n, MPI_DOUBLE,
257             vec, local_n, MPI_DOUBLE, 0, comm);
258    }
259 }  /* Print_vector */
260
261
262 void Mat_vect_mult(
263       double    local_A[]  /* in  */,
264       double    local_x[]  /* in  */,
265       double    local_y[]  /* out */,
266       int       local_m    /* in  */,
267       int       n          /* in  */,
268       int       local_n    /* in  */,
269       MPI_Comm  comm       /* in  */) {
270    double* x;
271    int local_i, j;
272    int local_ok = 1;
273
274    x = malloc(n*sizeof(double));
275    if (x == NULL) local_ok = 0;
276    Check_for_error(local_ok, "Mat_vect_mult",
277          "No se puede alojar temporary vector", comm);
278    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
279          x, local_n, MPI_DOUBLE, comm);
280
281    for (local_i = 0; local_i < local_m; local_i++) {
282       local_y[local_i] = 0.0;
283       for (j = 0; j < n; j++)
284          local_y[local_i] += local_A[local_i*n+j]*x[j];
285    }
286    free(x);
```

```
287 }  /* Mat_vect_mult */
```