Informe 03 - Algoritmos Paralelos

Lorena Xiomara Castillo Galdos

Abril 2017

Contents

1 Regla Trapezoidal en MPI

Podemos usar la regla trapezoidal para aproximar el área entre la gráfica de una función $y = f(x)$, dos líneas verticales y el eje x. Figura 1. La idea básica es dividir el intervalo en el eje x en n subintervalos iguales, luego aproximar el área entre la gráfica y cada subintervalo con un trapezoide cuya base es el subintervalo, cuyos lados verticales son las líneas verticales hasta el final del subintervalo y su cuarto lado es la línea secante que une los puntos donde las líneas verticales se cruzan en la gráfica. Figura 2. Si los puntos finales de los subintervalos son x_i y x_{i+1}, entonces el largo de los subintervalos es $h = x_{i+1} - x_1$. También, si los largos de los dos segmentos verticales son $f(x_i)$ y $f(x_{i+1})$, entonces el área del trapezoide es

$$A = \frac{h}{2}[(x_i) + f(x_{i+1})]$$

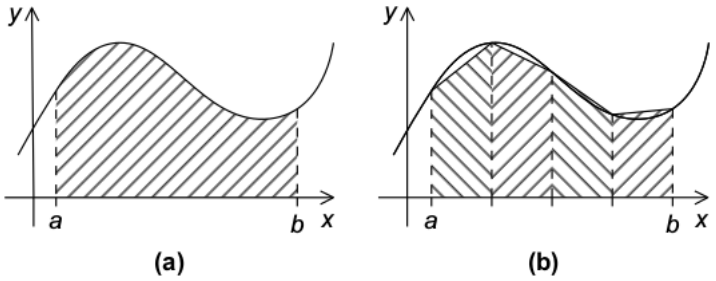

Figure 1: Regla trapezoidal: a) área a ser estimada y b) área aproximada usando trapezoides.

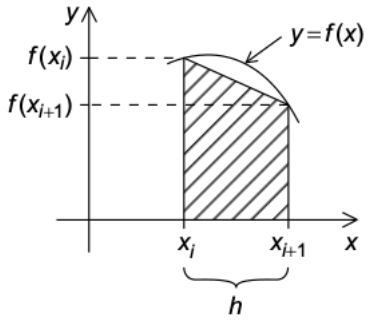

Figure 2: Un trapezoide.

Dado que escogemos los n subintervalos de tal manera que tengan el mismo largo, conocemos que si las líneas verticales son $x = a$ y $x = b$, entonces

$$h = \frac{b-a}{n}$$

Además, si llamamos al punto más a la izquierda x_0, y al punto más a la derecha x_n, tendremos

$$x_0 = a, x_1 = a + h, x_2 = a + 2h, ..., x_{n-1} = a + (n-1)h, x_n = b,$$

y la suma de las áreas de los trapezoides, según aproximación sería

$$Suma = h[f(x_0/2 + f(x_1) + f(x_2) + ... + f(x_{n-1}) + f(x_n)/2].$$

Una forma de paralelizar el código es dividir el intervalo $[a, b]$ en subintervalos y luego aplicar la regla trapexoidal a cada subintervalo, luego tendremos a uno de los procesos, por ejemplo el proceso 0, sumando los valores estimados.[1]

El código del programa se muestra a continuación

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

float trapecio(float local_a, float local_b, int local_n, float h)
{
    float integral;
    float x;
    int i;
    integral = ( ( f(local_a) + f(local_b) ) *1.0 )/2.0;
    x = local_a;

    for( i = 1; i <= (local_n -1); i++ )
    {
        x+=h;
        integral+=f(x);
    }
    integral*=h;
    return integral;
}

float f(float x)
{
  float return_val=x*x;
  return return_val;
}

void Get_data(int my_rank,int p,float *a_ptr,
                float *b_ptr,int *n_ptr)
{
      int source=0 , dest , tag;
```

```
      MPI_Status status;
      if(my_rank==0)
      {
          printf("Enter a, b, and n\n");
          scanf("%f %f %d",a_ptr,b_ptr,n_ptr);
          for(dest=1;dest<p;dest++) {
            tag=30;
            MPI_Send(a_ptr,1,MPI_FLOAT,dest,tag,MPI_COMM_WORLD);
            tag=31;
            MPI_Send(b_ptr,1,MPI_FLOAT,dest,tag,MPI_COMM_WORLD);
            tag=32;
            MPI_Send(n_ptr,1,MPI_INT,dest,tag,MPI_COMM_WORLD);
          }
      }
       else {
          tag=30;
          MPI_Recv(a_ptr,1,MPI_FLOAT,source,
            tag,MPI_COMM_WORLD,&status);
          tag=31;
          MPI_Recv(b_ptr,1,MPI_FLOAT,source,
            tag,MPI_COMM_WORLD,&status);
          tag=32;
          MPI_Recv(n_ptr,1,MPI_INT,source,
            tag,MPI_COMM_WORLD,&status);
        }
}


int main(int argc,char **argv)
{
   int  my_rank, p, n, local_n, source, dest=0, tag=50;
   float a, b, h, local_a, local_b, integral, total;
   //
   MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
   MPI_Comm_size(MPI_COMM_WORLD,&p);
   //
   Get_data(my_rank,p,&a,&b,&n);
   h=(b-a)/n ; local_n=n/p;
   local_a=a+my_rank*local_n*h ;
   local_b=local_a+local_n*h;
   integral=trapecio(local_a,local_b,local_n,h);
   if(my_rank==0) {
     total=integral;
     for(source=1;source<p;source++) {
       MPI_Recv(&integral,1,MPI_FLOAT,source,
        tag,MPI_COMM_WORLD,&status);
       total+=integral;
       printf("proceso %d calculo un total de %f\n",source,total);
     }
      printf("con n= %d trapezoides\n la estimacion",n);
      printf("de la integral entre %f y %f\n es= %f \n",a,b,total);
    }
    else {
      MPI_Send(&integral,1,MPI_FLOAT,dest,tag,MPI_COMM_WORLD);
    }
```

```
    MPI_Finalize();
}
```

El resultado se muestra en la Figura 3.

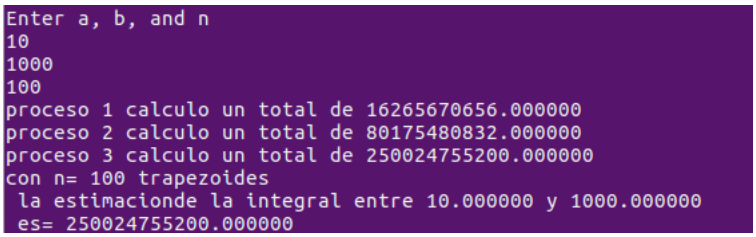

Figure 3: Resultados trapezoide.

2 MPI_Scatter y MPI_Gather

2.1 MPI_Scatter

Para la leer la dimensión de vectores, el proceso 0 puede recibir el valor ingresado por el usuario, y emitir (*broadcast*) este valor a los otros procesos. En este caso podemos hacer algo parecido con los vectores: el proceso 0 puede leerlos y emitirlos a los otros procesos. Sin embargo, esto puede causar mucho desperdicio. Si hay 10 procesos y los vectores tienen 10,000 componentes, entonces caa proceso necesitará asignar almacenamiento para vectores de 10,000 componentes, cuando sólo operará en subvectores con 1000 componentes. Si por ejemplo, cuando usamos una distribución de bloques, sería mejor si el proceso 0 enviara sólo componentes del 1000 al 1999 al proceso 1, componentes 2000 al 2999 al proceso 2 y así sucesivamente. Usando este enfoque, los procesos del 1 al 9 solo necesitarían asignar almacenamiento para los componentes que estén usando realmente.

Para leer todo un vector que está en el proceso 0 pero que sólo envíe los componentes necesarios a cada uno de los procesos MPI provee la función MPI_Scatter.

```
int MPI_Scatter (
      void*         send_buf_p /* in */,
      int           send_count /* in */,
      MPI_Datatype  send_type  /* in */,
      void*         recv_buf_p /* out */,
      int           recv_count /* in */,
      MPI_Datatype  recv_type  /* in */,
      int           src_proc   /* in */,
      MPI_Comm      comm       /* in */);
```

El código del programa para la distribución de datos de un vector se muestra a continuación

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

int main(int argc, char* argv[])
{
  int p, my_rank;
  int max_p = 8;
  int tam_buffer = 24; /* tama o del buffer a enviar,
          tiene que ser divisible por 2, 4, 6 y 8 */
  int tag = 5; //Etiqueta S=5
  MPI_Status status;

  int enviar[tam_buffer]; int recibir[tam_buffer];

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

  if (p > max_p || p % 2 != 0)
  {
    if (my_rank == 0)
    {
      printf("Ingresa un n mero par de procesos");
    }
    MPI_Finalize();
    exit(0);
  }

  if (my_rank == 0)
  {
    for ( int i = 0; i < tam_buffer; i++ )
    {
      enviar[i] = i;
    }

    // dividir el array enviar para todos los procesos,
          ponerlos en recibir
    MPI_Scatter(enviar, tam_buffer/p , MPI_INT, recibir,
            tam_buffer/p, MPI_INT, 0, MPI_COMM_WORLD);

    // imprimir porcion del array dividido
    for ( int i = 0; i < tam_buffer/p; i++ )
    {
      printf(" %d", recibir[i]);
    }
    printf("\n");

    // recibir de todos los procesos
    for ( int i = 1; i < p; i++ )
    {
      MPI_Recv (recibir, tam_buffer/p, MPI_INT, i, tag,
            MPI_COMM_WORLD, &status);
      for ( int j = 0; j < tam_buffer/p; j++ )
      {
        printf(" %d",recibir[j]);
      }
```

```
      printf("\n");
    }

    printf("Fin\n");
  }
  else
  {
    // Recibir de 0
    MPI_Scatter(enviar, tam_buffer/p, MPI_INT, recibir,
            tam_buffer/p, MPI_INT, 0, MPI_COMM_WORLD);
    //Enviar de regreso a 0
    MPI_Send (recibir, tam_buffer/p, MPI_INT, 0,
            tag, MPI_COMM_WORLD);

  }

  MPI_Finalize();
  exit(0);
}
```

El resultado se muestra en la Figura 4.

```
lorena@Lorena-pc:~/Documentos/Paralelos/TareaMPI/scatter$ mpiexec -np
 4 scatter
 0 1 2 3 4 5
 6 7 8 9 10 11
 12 13 14 15 16 17
 18 19 20 21 22 23
Fin
```

Figure 4: Resultados distribución de un vector de 0 a 23 con MPI_Scatter en 4 procesos.

2.2 MPI_Gather

Nuestro programa no estaría completo a menos que podamos ver nuestro vector resultante, así que necesitamos escribir una función para imprimir un vector distribuído. Este programa puede recolectar todos los componentes del vector en el proceso 0, y luego el proceso 0 puede imprimir todos los componentes. La comunicación en esta función puede ser llevada a cabo por la función MPI_Gather,

```
int MPI_Gather (
      void*         send_buf_p /* in */,
      int           send_count /* in */,
      MPI_Datatype  send_type  /* in */,
      void*         recv_buf_p /* out */,
      int           recv_count /* in */,
      MPI_Datatype  recv_type  /* in */,
      int           dest_proc   /* in */,
      MPI_Comm      comm       /* in */);
```

Los datos almacenados en la memoria referida como $send_buf_p$ en el proceso 0 es almacenada en el primer bloque en $recv_buf_p$, los datos almacenados

en la memoria referida como $send_buf_p$ en el proceso 1 es almacenada en el segundo bloque referido como rec_buf_p y así sucesivamente. Así que, si usamos una distribución de bloque, podemos implementar nuestra impresión de un vector distribuído como se muestra en el código a continuación,

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

int main(int argc, char* argv[])
{
  int p, my_rank;
  int max_p = 8;
  int tam_buffer = 24; /* tama o del buffer a enviar,
        tiene que ser divisible por 2, 4, 6 y 8 */
  int tag = 5; //Etiqueta S=5
  MPI_Status status;

  int enviar[tam_buffer]; int recibir[tam_buffer];

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

  if (p > max_p || p % 2 != 0)
  {
    if (my_rank == 0)
    {
      printf("Ingresa un n mero par de procesos
        (al menos %d)\n", max_p);
    }
    MPI_Finalize();
    exit(0);
  }

  for ( int i = 0; i < tam_buffer/p; i++ )
  {
     enviar[i] = (tam_buffer/p) * my_rank + i;
  }

  if (my_rank == 0) {    /* Process 0 does this */

    /* Gather the array x from all proceses, place it in y */
    MPI_Gather(enviar, tam_buffer/p, MPI_INT,
        recibir, tam_buffer/p, MPI_INT, 0,
            MPI_COMM_WORLD);

    /* Print out the gathered array */
    for ( int i = 0; i < tam_buffer; i++)
    {
      printf(" %d", recibir[i]);
    }
    printf("\n\n");

    for ( int i =0; i < tam_buffer/p; i++ )
```

```
    {
      printf (" %d", enviar[i]);
    }
    printf ("\n");

    /* Receive messages with hostname and the original data */
    /* from all other processes */
    for ( int i = 1; i < p; i++ )
    {
      MPI_Recv (&recibir, tam_buffer/p, MPI_INT,
            i, tag, MPI_COMM_WORLD, &status);
      for ( int j = 0; j < tam_buffer/p; j++ )
      {
        printf(" %d", recibir[j]);
      }
      printf("\n");
    }

    printf("Fin\n");
  }
  else
  { /* all other processes do this */

    /* Receive the scattered array from process 0,
        place it in array y */
    MPI_Gather(&enviar, tam_buffer/p, MPI_INT, &recibir,
        tam_buffer/p, MPI_INT, 0, MPI_COMM_WORLD);

    /* Send the received array back to process 0 */
    MPI_Send (&enviar, tam_buffer/p, MPI_INT,
        0, tag, MPI_COMM_WORLD);

  }

  MPI_Finalize();
  exit(0);
}
```

El resultado se muestra en la Figura 5.

```
lorena@Lorena-pc:~/Documentos/Paralelos/TareaMPI/gather$ mpiexec -np 4 gather
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

 0 1 2 3 4 5
 6 7 8 9 10 11
 12 13 14 15 16 17
 18 19 20 21 22 23
Fin
```

Figure 5: Resultados de impresión de un vector de 0 a 23 con MPI_Gather repartido en 4 procesos.

3 Multiplicación de Matriz-Vector con MPI_Allgather

En una función que multiplica una matriz por un vector, si $A = (a_{ij})$ is una matriz de mxn y x es un vector con n componentes, entonces $y = Ax$ es un vector con m componentes, entonces $y = Ax$ es un vector con m componentes y podemos encontrar el elemento i-ésimo de y realizando el producto punto de la i-ésima fila de A con x,

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + ...a_{i,n-1}x_{n-1}.$$

Como se muestra en la Figura 6

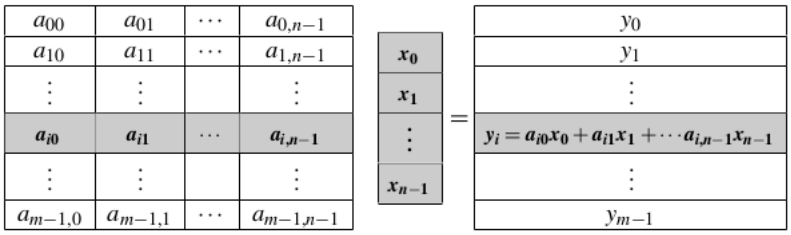

Figure 6: Multiplicación de una matriz por un vector.

Usando comunicación colectiva, podemos ejecutar una llamada a MPI_Gather seguida de una llamada a MPI_Bcast, esto involucra dos comunicaciones de estructura de árbol, y puede ser mejor usando una de mariposa. MPI provee una función para esto,

```
int MPI_Allgather (
      void*         send_buf_p /* in */,
      int           send_count /* in */,
      MPI_Datatype  send_type  /* in */,
      void*         recv_buf_p /* out */,
      int           recv_count /* in */,
      MPI_Datatype  recv_type  /* in */,
      MPI_Comm      comm       /* in */);
```

Esta función concatena los contenidos de cada proceso

A continuación se muestra el código de un programa para multiplicar una matriz por un vector,

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

void Print_vector(double local_b[], int local_n,
       int n, char title[], int my_rank, MPI_Comm comm){
```

```
  double* b = NULL;
  int i;
  if(my_rank == 0){
    b = malloc(n*sizeof(double));
    MPI_Gather(local_b, local_n, MPI_DOUBLE,b,
            local_n, MPI_DOUBLE,0,comm);
    printf("%s\n", title);
    for(i = 0; i < n; i++){
      printf("%f ", b[i]);
    }
    printf("\n");
    free(b);
  }
  else{
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b,
        local_n, MPI_DOUBLE, 0, comm);
  }
}

void Read_vector(double local_a[], int local_n,
        int n, char vec_name[], int my_rank, MPI_Comm comm){
  double* a = NULL;
  int i;

  if(my_rank == 0){
    a = malloc(n*sizeof(double));
    printf("Ingrese el vector %s\n", vec_name);
    for(i = 0; i < n; i++)
      {scanf("%lf", &a[i]);}
     MPI_Scatter(a, local_n, MPI_DOUBLE, local_a,
            local_n, MPI_DOUBLE,0,comm);
     free(a);

  }
    else{
      MPI_Scatter(a,local_n,MPI_DOUBLE,local_a,
            local_n,MPI_DOUBLE,0,comm);
    }
}

void Mat_vect_mult(double local_A[], double local_x[],
        double local_y[], int local_m, int n, int local_n,
            MPI_Comm comm)
{
  double* x;
  int local_i,j;
  int local_ok=1;
  x = malloc(n*sizeof(double));
  MPI_Allgather(local_x, local_n, MPI_DOUBLE, x,
    local_n, MPI_DOUBLE, comm);
  for(local_i = 0; local_i < local_m; local_i++){
    local_y[local_i] = 0.0;
    for(j = 0; j <n; j++){
      local_y[local_i] += local_A[local_i*n+j] * x[j];
    }
  }
  free(x);
```

```
}

int main(int argc,char **argv){
  int comm_sz;
  int my_rank;
  double start,finish;

  int m=4; ///filas
  int n=4; ///columnas
  int local_n,local_m;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  //
  local_n = n/comm_sz;
  local_m = m/comm_sz;
  double local_x[local_n],local_y[local_m],local_A[local_n*local_m];

  Read_vector(local_A,local_n+local_m,n*m,"A",
    my_rank,MPI_COMM_WORLD);
  Read_vector(local_x,local_n,n,"X",my_rank,MPI_COMM_WORLD);

  //
  double tInicio,tFin;
  double MPI_Wtime(void);
  tInicio = MPI_Wtime();
  Mat_vect_mult(local_A, local_x, local_y, local_m,n,
    local_n,MPI_COMM_WORLD);
  tFin = MPI_Wtime();
  Print_vector(local_y,local_m,m,"Y", my_rank,MPI_COMM_WORLD);
  printf("Tiempo: %f \n",tFin-tInicio);
  MPI_Finalize();
  return 0;
}
```

Los resultados se muestran en la Figura 7.

```
lorena@Lorena-pc:~/Documentos/Paralelos/TareaMPI/matriz-vector$ mpiexec -np 4 ma
triz-vector
Ingrese el vector A
1 2 3 4 5 6 7 8 9 7 6 5 4 3 2 1
Ingrese el vector X
1 2 3 4
Tiempo: 0.000167
Tiempo: 0.000108
Tiempo: 0.000014
Y
20.000000 44.000000 68.000000 92.000000
Tiempo: 0.000174
```

Figure 7: Resultados de multiplicación matriz-vector.

4 Parallel-Sorting

A continuación se muestra el código del programa,

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

/* the number of data elements in each process */
#define N 16384


/* initialize the data to random values
    based on rank (so they're different) */
void init(int* data, int rank) {
  int i;
  srand(rank);
  for (i = 0; i < N; i++) {
    data[i] = rand( ) % 100;
  }
}

/* print the data to the screen */
void print(int* data ) {
  int i;

  for (i = 0; i < N; i++) {
    printf("%d ", data[i]);
  }
  printf("\n");
}

/* comparison function for qsort */
int cmp(const void* ap, const void* bp) {
  int a = * ((const int*) ap);
  int b = * ((const int*) bp);

  if (a < b) {
    return -1;
  } else if (a > b) {
    return 1;
  } else {
    return 0;
  }
}

/* find the index of the largest item in an array */
int max_index(int* data) {
  int i, max = data[0], maxi = 0;

  for (i = 1; i < N; i++) {
    if (data[i] > max) {
      max = data[i];
      maxi = i;
    }
  }
  return maxi;
}

/* find the index of the smallest item in an array */
```

```
int min_index(int* data) {
  int i, min = data[0], mini = 0;

  for (i = 1; i < N; i++) {
    if (data[i] < min) {
      min = data[i];
      mini = i;
    }
  }
  return mini;
}


/* do the parallel odd/even sort */
void parallel_sort(int* data, int rank, int size) {
  int i;

  /* the array we use for reading from partner */
  int other[N];

  /* we need to apply P phases where P is the number of processes */
  for (i = 0; i < size; i++) {
    /* sort our local array */
    qsort(data, N, sizeof(int), &cmp);

    /* find our partner on this phase */
    int partener;

    /* if it's an even phase */
    if (i % 2 == 0) {
      /* if we are an even process */
      if (rank % 2 == 0) {
        partener = rank + 1;
      } else {
        partener = rank - 1;
      }
    } else {
      /* it's an odd phase - do the opposite */
      if (rank % 2 == 0) {
        partener = rank - 1;
      } else {
        partener = rank + 1;
      }
    }

    /* if the partener is invalid, we should simply
        move on to the next iteration */
    if (partener < 0 || partener >= size) {
      continue;
    }

    /* do the exchange - even processes send first
        and odd processes receive first
     * this avoids possible deadlock of two processes
        working together both sending */
    if (rank % 2 == 0) {
      MPI_Send(data, N, MPI_INT, partener, 0, MPI_COMM_WORLD);
```

```
      MPI_Recv(other, N, MPI_INT, partener, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
    } else {
      MPI_Recv(other, N, MPI_INT, partener, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
      MPI_Send(data, N, MPI_INT, partener, 0, MPI_COMM_WORLD);
    }

    /* now we need to merge data and other based on
        if we want smaller or larger ones */
    if (rank < partener) {
      /* keep smaller keys */
      while (1) {
        /* find the smallest one in the other array */
        int mini = min_index(other);

        /* find the largest one in out array */
        int maxi = max_index(data);

        /* if the smallest one in the other array is
            less than the largest in ours, swap them */
        if (other[mini] < data[maxi]) {
          int temp = other[mini];
          other[mini] = data[maxi];
          data[maxi] = temp;
        } else {
          /* else stop because the smallest are now in data */
          break;
        }
      }
    } else {
      /* keep larger keys */
      while (1) {
        /* find the largest one in the other array */
        int maxi = max_index(other);

        /* find the largest one in out array */
        int mini = min_index(data);

        /* if the largest one in the other array
        is bigger than the smallest in ours, swap them */
        if (other[maxi] > data[mini]) {
          int temp = other[maxi];
          other[maxi] = data[mini];
          data[mini] = temp;
        } else {
          /* else stop because the largest are now in data */
          break;
        }
      }
    }
  }
}
int main(int argc, char **argv) {
  /* our rank and size */
  int rank, size;
  double MPI_Wtime(void);
```

```
  double start,finish;
  start=MPI_Wtime();
  /* our processes data */
  int data[N];

  /* initialize MPI */
  MPI_Init(&argc, &argv);

  /* get the rank (process id) and size (number of processes) */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  /* initialize the data */
  init(data, rank);

  /* do the parallel odd/even sort */
  parallel_sort(data, rank, size);



  /* quit MPI */
  finish=MPI_Wtime();
  printf("proc %d> Elapsed time = %e seconds\n",
    rank,(finish-start));
  MPI_Finalize( );
  /* now print  our data */
  print(data);
  return 0;
}
```

Los resultados se muestran en la Figura 8.

References

[1] Peter Pacheco. *An introduction to parallel programming.* Elsevier, 2011.

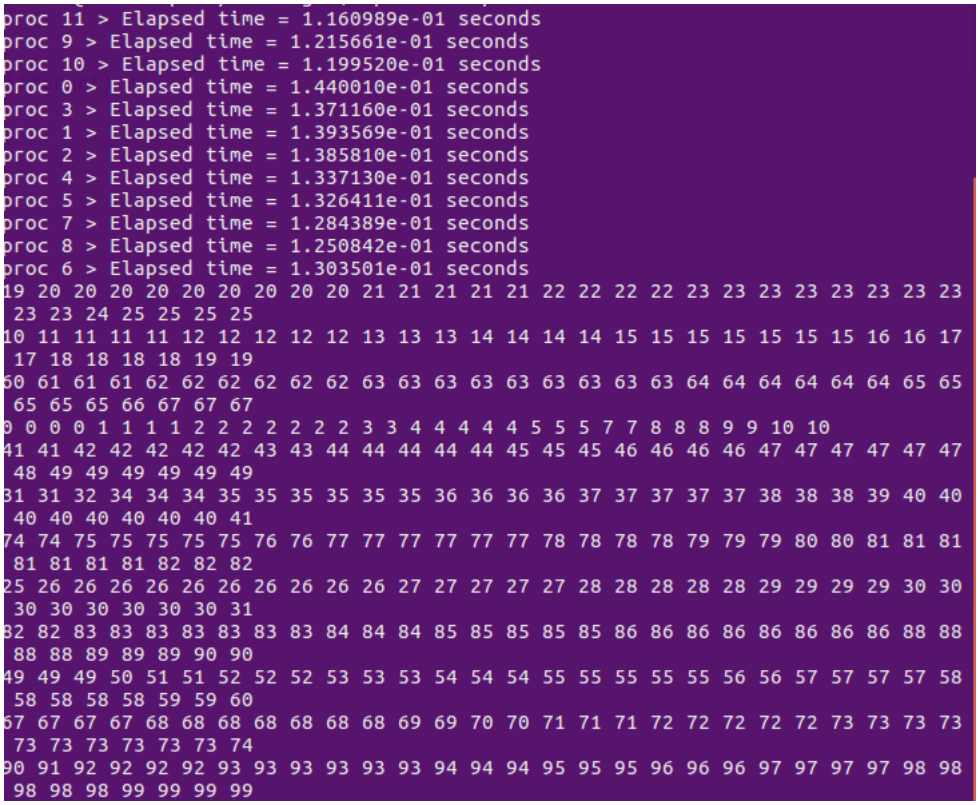

Figure 8: Parallel Sorting.