



Universidad Nacional de Entre Ríos

FACULTAD DE INGENIERÍA

## TRABAJO PRÁCTICO N° 2

*Computación de Alto Rendimiento*

Autor:

Justo Garcia

Septiembre 2023

## Tabla de contenidos

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Ejercicio 1</b>	<b>3</b>
2.1	Consigna . . . . .	3
2.2	Resolución . . . . .	3
2.2.1	Implementación con MPI . . . . .	3
2.2.2	Ejecución en cluster . . . . .	8
2.2.3	Análisis . . . . .	9
<b>3</b>	<b>Ejercicio 2</b>	<b>11</b>
3.1	Consigna . . . . .	11
3.2	Resolución . . . . .	11
3.2.1	Implementación con MPI . . . . .	11
3.2.2	Ejecución y análisis . . . . .	14
<b>4</b>	<b>Ejercicio 3</b>	<b>16</b>
4.1	Consigna . . . . .	16
4.2	Resolución . . . . .	16
4.2.1	Implementación con MPI . . . . .	16
4.2.2	Ejecución . . . . .	20
4.2.3	Análisis . . . . .	22
<b>5</b>	<b>Ejercicio 4</b>	<b>23</b>
5.1	Consigna . . . . .	23
5.2	Resolución . . . . .	23
5.2.1	Implementación con MPI . . . . .	23
5.2.2	Ejecución en cluster . . . . .	26
5.2.3	Análisis . . . . .	26

# 1 Introducción

En este informe desarrollaré las actividades propuestas para el trabajo práctico número 2 de Computación de Alto Rendimiento. Iré adjuntando los códigos necesarios, sin embargo recomiendo revisar el repositorio de GitHub [**repositorio**] de este trabajo para obtener una mejor visualización de las soluciones propuestas.

## 2 Ejercicio 1

### 2.1 Consigna

Escribir una rutina *mybcast(...)* con la misma signatura que *MPI\_Bcast(...)* mediante el uso de send/receive, primero en forma secuencial y luego en forma de árbol. Comparar los tiempos en función del número de procesadores.

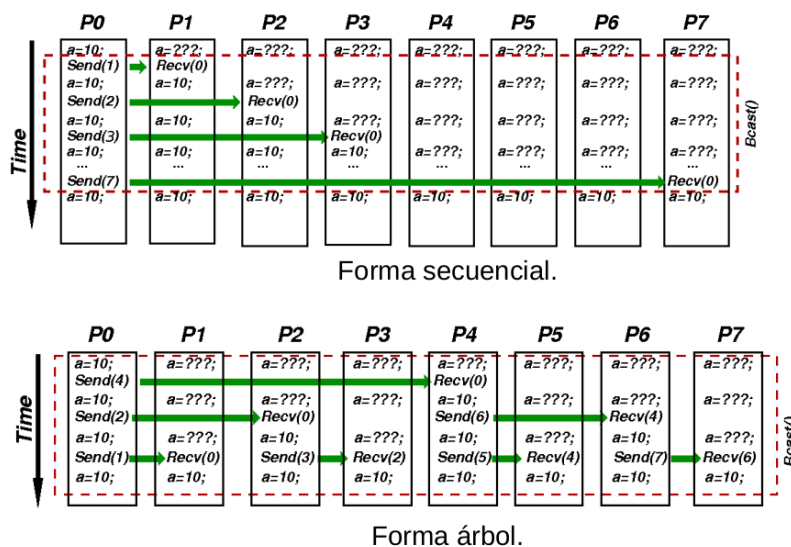


Fig. 1: Formas de hacer el Bcast

## 2.2 Resolución

### 2.2.1 Implementación con MPI

Para resolver este problema decidí declarar dos funciones donde hiciese la difusión de un dato, en el primero con comunicación punto a punto secuencial y en el segundo siguiendo una estructura de árbol. Ambas funciones fueron implementadas respetando los parámetros de *MPI\_Bcast()*.

```
1 #ifndef FUNCIONES_H
```

```
2 #define FUNCIONES_H
3
4 #include <mpi.h>
5
6 /// @brief Funci n que permite hacer el broadcast con comunicaci n punto
    a punto
7 /// @param sendbuff Direcci n del buffer a enviar
8 /// @param largo Largo del buffer a enviar
9 /// @param tipoDeDato Tipo de dato a enviar
10 /// @param origen Rank del proceso que tiene el dato
11 /// @param communicator Communicator de MPI dentro del cual se da la
    comuniaci n
12 void My_BcastPtoPto(void *sendbuff, int largo, MPI_Datatype tipoDeDato,
    int origen, MPI_Comm communicator);
13
14
15 /// @brief Funci n que permite hacer el broadcast con un rbol binario
16 /// @param sendbuff Direcci n del buffer a enviar
17 /// @param largo Largo del buffer a enviar
18 /// @param tipoDeDato Tipo de dato a enviar
19 /// @param origen Rank del proceso que tiene el dato
20 /// @param communicator Communicator de MPI dentro del cual se da la
    comuniaci n
21 void My_BcastTree(void *sendbuff, int largo, MPI_Datatype tipoDeDato, int
    origen, MPI_Comm communicator);
22
23 #endif

```

```
1 #include "funciones.h"
2
3 void My_BcastPtoPto(void *sendbuff, int largo, MPI_Datatype tipoDeDato,
    int origen, MPI_Comm communicator)
4 {
5     int tam;
6     MPI_Comm_size(communicator, &tam);
7     MPI_Status status;
8     int mtag = 0;
9     int rank;
10    MPI_Comm_rank(communicator, &rank);

```

```
11     if(rank == origen)
12     {
13         for(int i = 0 ; i < tam ; i++)
14         {
15             if(i!=origen)
16                 MPI_Send(sendbuff, largo, tipoDeDato, i, mtag,
communicator);
17         }
18     }
19     else
20     {
21         MPI_Recv(sendbuff, largo, tipoDeDato, origen, mtag, communicator,
&status);
22     }
23 }
24
25 void My_BcastTree(void *sendbuff, int largo, MPI_Datatype tipoDeDato, int
origen, MPI_Comm communicator)
26 {
27     int tam;
28     MPI_Comm_size(communicator, &tam);
29     MPI_Status status;
30     int mtag = 0;
31     int rank;
32     MPI_Comm_rank(communicator, &rank);
33
34     int n1 = 0, n2 = tam;
35
36     while(true)
37     {
38         int medio = (n1+n2)/2;
39         if(rank == n1)
40         {
41             MPI_Send(sendbuff, largo, tipoDeDato, medio, mtag,
communicator);
42         }
43         else if (rank == medio)
44         {
45             MPI_Recv(sendbuff, largo, tipoDeDato, n1, mtag, communicator,
```

```
        &status);  
46     }  
47  
48  
49     if (rank < medio)  
50     {  
51         n2 = medio;  
52     }  
53     else  
54     {  
55         n1 = medio;  
56     }  
57  
58     if((n2-n1)==1)  
59     {  
60         break;  
61     }  
62  
63 }  
64 }
```

Luego, en un cpp aparte importé mis implementaciones y las utilicé, midiendo los tiempos que llevó cada una de ellas. Además realicé el mismo proceso con el broadcast implementado en mpich.

```
1 #include <stdio.h>  
2 #include <mpi.h>  
3 #include <iostream>  
4 #include <vector>  
5 #include "modulos/funciones.h"  
6  
7 using namespace std;  
8  
9  
10  
11 int main(int argc, char **argv) {  
12  
13
```

```
14     int ierror, rank, size, entrada = 8;
15     MPI_Init(&argc, &argv);
16     MPI_Status status;
17     MPI_Comm_rank(MPI_COMM_WORLD, &rank); //Paso por referencia rank y la
modifica
18     MPI_Comm_size(MPI_COMM_WORLD, &size);
19     double tInicio, tFin;
20
21     int megas = 400;
22     int N = megas * 1024 * 1024 / sizeof(int);
23     vector<int> mensaje(N, 0);
24     if (rank == 0)
25     {
26         for (int i = 0; i < mensaje.size() ; i++)
27             mensaje[i] = i;
28     }
29     if (rank == 0)
30         printf("Broadcast,Tiempo,Nro Procesadores\n");
31
32     // if(!rank) printf("N = %d", size);
33     MPI_Barrier(MPI_COMM_WORLD);
34     if(rank == 0)
35         tInicio = MPI_Wtime();
36
37     My_BcastPtoPto(&mensaje[0], N, MPI_INT, 0, MPI_COMM_WORLD);
38
39     if(rank == 0)
40     {
41         tFin = MPI_Wtime();
42         printf("Punto a punto,%f,%d\n", tFin-tInicio, size);
43     }
44
45     MPI_Barrier(MPI_COMM_WORLD);
46     if(rank == 0)
47         tInicio = MPI_Wtime();
48     My_BcastTree(&mensaje[0], N, MPI_INT, 0, MPI_COMM_WORLD);
49     if(rank == 0)
50     {
51         tFin = MPI_Wtime();
```



```
52     printf("Arbol,%f,%d\n", tFin-tInicio, size);
53 }
54
55 MPI_Barrier(MPI_COMM_WORLD);
56 if(rank == 0)
57     tInicio = MPI_Wtime();
58 MPI_Bcast(&mensaje[0], N, MPI_INT, 0, MPI_COMM_WORLD);
59 if(rank == 0)
60 {
61     tFin = MPI_Wtime();
62     printf("MPICH,%f,%d\n", tFin-tInicio, size);
63 }
64
65
66
67 MPI_Finalize();
68
69
70 }
```

Como el objetivo era comparar los tiempos en función del número de procesadores, decidí desarrollar un script muy simple en bash que compile el programa y lo corra con diferentes  $N$ s. Sin embargo, no consideré que esto no permitía que se corra con el sistema de colas que utilizan los clusters.

### 2.2.2 Ejecución en cluster

Por la baja disponibilidad de nodos en el cluster de la facultad tuve que enviárselo al profesor para que lo ejecute en Santa Fe.

Luego me devolvió la salida de la ejecución con diferentes cantidades de nodos por el problema que expliqué previamente con el script de bash.

### 2.2.3 Análisis

Lo primero que hice fue convertir los archivos de tipo *.dat* a csvs que pueda manejar con facilidad con la librería Pandas [1].

Con los datos cargados en un DataFrame, estructura de Pandas, utilicé la librería seaborn para graficar los tiempos de los distintos tipos de broadcasts, diferenciandolos también según el número de procesadores. Así obtuve la siguiente gráfica 2:

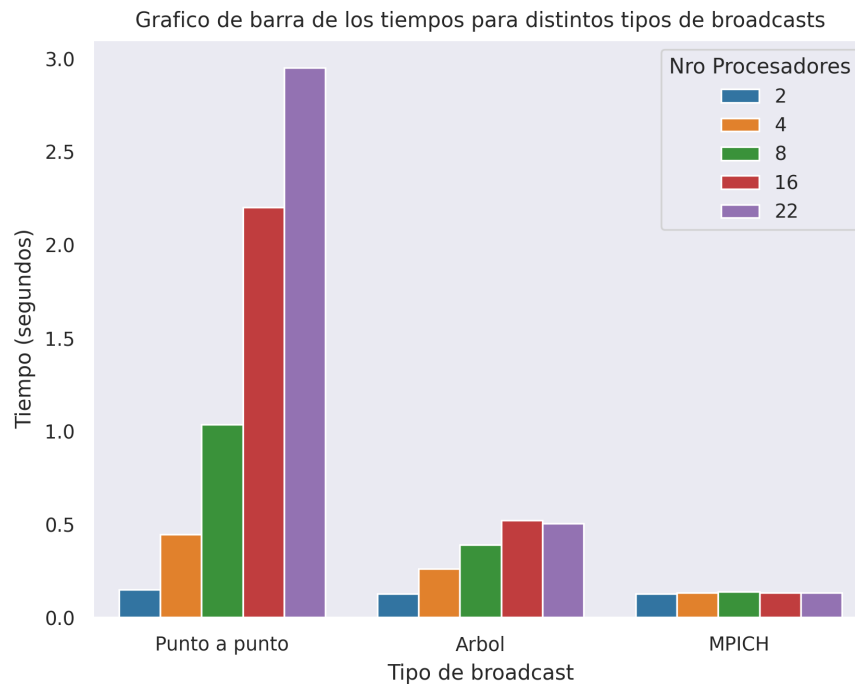


Fig. 2: Tiempos para distintos tipos de broadcasts

Además calcule media y varianza para cada tipo de broadcast:

---

Tipo de broadcast	Media	Desvío
Punto a punto	1,357	1,188
Árbol	0,361	0,168
MPICH	0,133	0,003

---

Tabla 1: Media y varianza por tipo de broadcast

Como era de esperarse, la más lenta es la comunicación punto a punto porque se va a ir realizando secuencialmente a cada una de las unidades de cómputo. Luego, con una reducción significativa del tiempo, lo sigue la implementación de un algoritmo de tipo árbol y por último se encuentra el broadcast implementado en MPICH. Además, tanto con la varianza [1] como con la gráfica [2] podemos observar que en la punto a punto crece mucho el tiempo cuando aumenta el número de procesadores, lo que tiene lógica porque implica un mayor número de iteraciones para entregar un mensaje a todos ellos.

## 3 Ejercicio 2

### 3.1 Consigna

Escribir un programa haciendo uso de MPI que dado una variable definida en todo los procesos busque el valor máximo y que proceso lo contiene. Dicho programa debe funcionar de manera similar a  $MPI_{Reduce}(\dots)$  utilizando una estructura tipo árbol para tal fin.

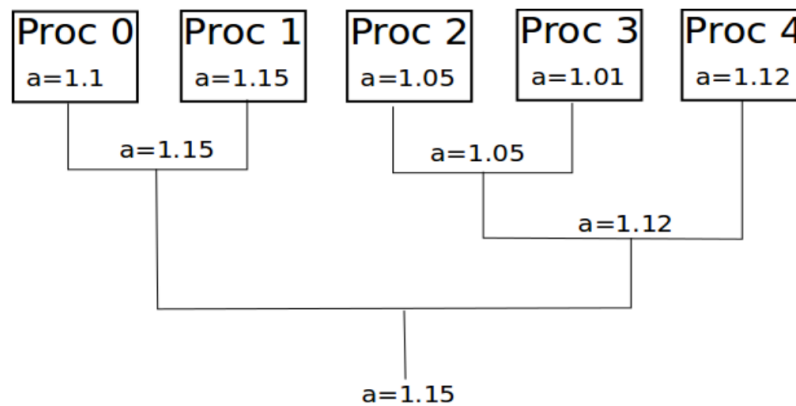


Fig. 3: Forma de hacer el Reduce

### 3.2 Resolución

#### 3.2.1 Implementación con MPI

Para la solución del enunciado procedí a implementar un código que me permita encontrar el valor máximo y el poseedor de él, pensando en el desarrollo de una posterior validación decidí que los datos que tiene cada uno los lea de un archivo *.txt*. El código implementado es el siguiente:

```
1 #include <stdio.h>
2 #include <mpi.h>
3 #include <iostream>
4 #include <vector>
5 #include <fstream>
```

```

6 // #include "modulos/funciones.h"
7
8 using namespace std;
9
10
11
12 int main(int argc, char **argv) {
13
14     freopen("datos/salida.csv", "a", stdout);
15
16     int ierror, rank, size, entrada = 8;
17     MPI_Init(&argc, &argv);
18     MPI_Status status;
19     MPI_Comm_rank(MPI_COMM_WORLD, &rank); //Paso por referencia rank y la
    modifica
20     MPI_Comm_size(MPI_COMM_WORLD, &size);
21
22     double valores[size];
23
24     if(!rank)
25     {
26         ifstream archivo("datos/datosInit.txt");
27         if(!archivo.is_open()) printf("\nAbrio\n");
28         for(int i = 0; i<size ; i++)
29             archivo>>valores[i];
30     }
31
32
33     MPI_Bcast(valores, size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
34
35     double valorMax = valores[rank];
36     double rankMaxV = rank;
37
38     double paqueteMax[2] = {valorMax, rankMaxV};
39     double paqueteCompar[2];
40
41
42
43     for(int paso = 1 ; paso < size ; paso *=2)

```

```
44     {
45         if(rank % (2*paso) == 0)
46         {
47             int rankCompar = rank + paso;
48             if(rankCompar < size)
49             {
50                 double valorCompar;
51                 // printf("%d va a recibir de %d\n", rank, rankCompar);
52                 MPI_Recv(&paqueteCompar, 2, MPI_DOUBLE, rankCompar, 0,
MPI_COMM_WORLD, &status);
53                 if(paqueteCompar[0] >= paqueteMax[0])
54                 {
55                     paqueteMax[0] = paqueteCompar[0];
56                     paqueteMax[1] = paqueteCompar[1];
57                     // printf("Nuevo valor maximo %f y rank %f en %d\n",
paqueteMax[0], paqueteMax[1], rank);
58                 }
59             }
60         }
61         else
62         {
63             int rankEnviar = rank - paso;
64             if (rankEnviar >= 0)
65             {
66                 // printf("%d va a enviar a %d\n", rank, rankEnviar);
67                 MPI_Send(&paqueteMax, 2, MPI_DOUBLE, rankEnviar, 0,
MPI_COMM_WORLD);
68             }
69
70
71         }
72         MPI_Barrier(MPI_COMM_WORLD);
73     }
74
75     if (rank == 0)
76     {
77         // cout<<"El valor m ximo es "<<valorMax<<" y est  en "<<
rankMaxV<<endl;
78         printf("%f,%f\n", paqueteMax[0], paqueteMax[1]);
```

```
79     }  
80  
81  
82  
83  
84     MPI_Finalize();  
85  
86  
87 }
```

### 3.2.2 Ejecución y análisis

Decidí que su ejecución y validación se haga desde una notebook de jupyter, la cual puede verse renderizada desde el repositorio de github [2]. En ella se generan valores aleatorios que luego serán leídos por los diferentes procesadores y se ejecutará el reduce para buscar el máximo y su rank. Luego de obtener las salidas para cada uno de los conjuntos de datos realiza la verificación de que los datos devueltos estén bien, de esta forma pueden comprobarse diferentes sets de datos.

```
import pandas as pd  
  
df = pd.read_csv("datos/salida.csv")  
  
i = 0  
for _, row in df.iterrows():  
    print(test(row["ValorMax"], row["RankPoseedor"], numeros[i]))  
    i+=1  
  
[Out]  
True  
True  
True  
True  
True  
True  
True  
True  
True
```

Fig. 4: Resultado obtenido

Como se puede apreciar en 4 para valores distintos se obtuvieron los valores correctos en

todos ellos.



## 4 Ejercicio 3

### 4.1 Consigna

Implementar un función utilizando MPI que permita mostrar el contenido de un determinado buffer, que será el resultado de concatenar varios buffers de tamaño variable (por procesador) ordenados según el proceso, como muestra la siguiente figura. Presentar el código implementado conjuntamente con algún ejemplo de utilización de dicha función. Adicionalmente, emplear una función colectiva vectorizada para obtener el mismo resultado.

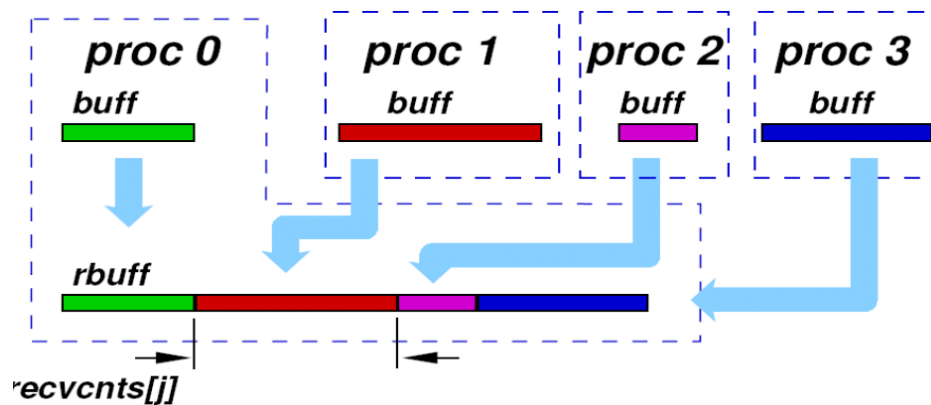


Fig. 5: Representación

### 4.2 Resolución

#### 4.2.1 Implementación con MPI

Para resolver la situación que enuncia el problema, decidí implementar una función que realice se comporte y tenga la misma firma que el gather vectorizado. Esta permite concatenar distintos buffers de tamaño variable y de forma ordenada.

Con la función correctamente desarrollada decidí utilizarla generando buffers de la misma forma que en clase e imprimir por consola el resultado tras utilizar la función implementada.

Además realicé el mismo procedimiento pero simplemente utilizando *MPI\_Gatherv(...)*.

Estos códigos mencionados son los siguientes:

### Funciones

```
1 #include "funciones3.h"
2 #include <iostream>
3 #include <vector>
4
5 using namespace std;
6
7 void MostrarConMPI(const void *sendbuf, int sendcnt, MPI_Datatype
    tipoDeDatoEnv, void *recvbuf, const int *recvcnt, const int *displs,
    MPI_Datatype tipoDeDatoRecv, int raiz, MPI_Comm comm)
8 {
9     int rank;
10    MPI_Comm_rank(comm, &rank);
11    MPI_Gatherv(sendbuf, sendcnt, tipoDeDatoEnv, recvbuf, recvcnt, displs,
    tipoDeDatoRecv, raiz, comm);
12
13 }
14
15 void MostrarSinMPI(double *sendbuf, int sendcnt, MPI_Datatype
    tipoDeDatoEnv, double *recvbuf, const int *recvcnt, const int *displs,
    MPI_Datatype tipoDeDatoRecv, int raiz, MPI_Comm comm)
16 {
17     int rank, mtag = 0;
18     MPI_Comm_rank(comm, &rank);
19     if(rank == raiz)
20     {
21         int tam;
22         MPI_Comm_size(comm, &tam);
23         int cont = 0;
24         vector<double> buffIntermedio;
25         for(int i = 0 ; i < sendcnt ; i++)
26         {
27             recvbuf[i] = sendbuf[i];
28             cont++;
29         }
```

```
30     for(int i = 0 ; i < tam ; i++)
31     {
32         if(i!=rank)
33         {
34             buffIntermedio.resize(recvcnt[i]);
35             MPI_Recv(&buffIntermedio[0], recvcnt[i], tipoDeDatoRecv, i
, mtag, comm, MPI_STATUS_IGNORE);
36             for(int j = 0 ; j < recvcnt[i] ; j++)
37             {
38                 recvbuf[cont] = buffIntermedio[j];
39                 cont++;
40             }
41         }
42     }
43 }
44 else
45 {
46     MPI_Send(sendbuf, sendcnt, tipoDeDatoEnv, raiz, mtag, comm);
47 }
48
49 }
```

## Main

```
1 #include <stdio.h>
2 #include <mpi.h>
3 #include <iostream>
4 #include <vector>
5 #include "modulos/funciones3.h"
6
7 using namespace std;
8
9
10
11 int main(int argc, char **argv) {
12
13
14     int rank, size;
15     MPI_Init(&argc, &argv);
16     MPI_Status status;
```

```
17     MPI_Comm_rank(MPI_COMM_WORLD, &rank); //Paso por referencia rank y la
    modifica
18     MPI_Comm_size(MPI_COMM_WORLD, &size);
19
20     int sendcnt = rank+1;
21     double *sbuff = new double[rank+1];
22     for(int j = 0 ; j < sendcnt ; j++) sbuff[j] = rank*1000+j;
23
24     int rsize = size*(size+1)/2;
25     int *recvcnts = NULL;
26     int *displs = NULL;
27     double *rbuff = NULL;
28
29     if (!rank) {
30         rbuff = new double[rsize];
31         recvcnts = new int[size];
32         displs = new int[size];
33         for (int j=0; j<size; j++) recvcnts[j] = (j+1);
34         displs[0]=0;
35         for (int j=1; j<size; j++)
36             displs[j] = displs[j-1] + recvcnts[j-1];
37     }
38
39     if(!rank) printf("Cada uno va a tener:\n");
40
41
42     for(int i = 0 ; i < size ; i++)
43     {
44         if(rank == i)
45         {
46             printf("Rank: %d\n",i);
47             for(int i = 0 ; i < rank+1 ; i++) cout<<"\t"<<sbuff[i]<<endl;
48         }
49         MPI_Barrier(MPI_COMM_WORLD);
50     }
51
52     if(!rank) printf("\n-----\n");
53     MPI_Barrier(MPI_COMM_WORLD);
54
```

```
55     MostrarSinMPI(sbuff, sendcnt, MPI_DOUBLE, rbuff, recvcnts, displs,  
MPI_DOUBLE, 0, MPI_COMM_WORLD);  
56  
57  
58     if(!rank)  
59     {  
60         printf("Luego del gather vectorizado propio:\n");  
61         for(int i = 0 ; i < rsize ; i++) cout<<"\t"<<rbuff[i]<<endl;  
62     }  
63  
64     double *rbuff2 = NULL;  
65     if(!rank) rbuff2 = new double[rsize];  
66  
67     MPI_Barrier(MPI_COMM_WORLD);  
68  
69     MostrarConMPI(sbuff, sendcnt, MPI_DOUBLE, rbuff2, recvcnts, displs,  
MPI_DOUBLE, 0, MPI_COMM_WORLD);  
70     if(!rank)  
71     {  
72         printf("\n-----\n");  
73         printf("Luego del gather vectorizado de MPI:\n");  
74         for(int i = 0 ; i < rsize ; i++) cout<<"\t"<<rbuff2[i]<<endl;  
75     }  
76  
77     MPI_Finalize();  
78  
79  
80 }
```

### 4.2.2 Ejecución

Para su ejecución lo compile y corré en mi computadora, obteniendo la siguiente salida:

```
justo@pop-os: ~/Documen...
Cada uno va a tener:
Rank: 0
    0
Rank: 1
    1000
    1001
Rank: 2
    2000
    2001
    2002
Rank: 3
    3000
    3001
    3002
    3003

-----
Luego del gather vectorizado propio:
    0
    1000
    1001
    2000
    2001
    2002
    3000
    3001
    3002
    3003

-----
Luego del gather vectorizado de MPI:
    0
    1000
    1001
    2000
    2001
    2002
    3000
    3001
    3002
    3003
justo@pop-os:~/Documentos/Facu/CAR/Practica-CAR/TP 2/EJ 3$
```

Fig. 6: Salida tras correr el código presentado

### 4.2.3 Análisis

El análisis surge de visualizar correctamente la salida presentada previamente 6, en esta se puede ver como se presentan ordenadamente los buffers de cada uno de los procesos por separados y una vez ya unidos en un solo buffer con los dos procedimientos descritos en la sección de la implementación.

## 5 Ejercicio 4

### 5.1 Consigna

Implementar la versión paralela del Teorema de los Números Primos con distribución de carga estática.

Buscar los número primos hasta  $1e7$  empleando las siguientes particiones  $1e3$ ,  $1e4$ ,  $1e5$ ,  $1e6$ ,  $2e6$ , empleando 5 nodos. Realizar un análisis del balance de carga en los procesadores.

Obtener las distribuciones por procesador de tiempo consumido en cálculo y en comunicación/sincronización para cada partición.

Graficar el tiempo consumido en función de la partición. ¿Qué conclusiones puede sacar de la gráfica?.

### 5.2 Resolución

#### 5.2.1 Implementación con MPI

Para la resolución de este problema decidí implementar un ciclo que recorra distintos tamaños de chunks (los especificados por el enunciado). Con ellos, en primer lugar, cada uno de los nodos calcula las porciones que tiene asignadas y empieza a verificar si cada uno de los número que les corresponde es primo o no. Por último imprimen su identificador, el tiempo que les llevo de cálculo, el transcurrido para la sincronización y finalmente el tamaño de chunk.

```
1 #include <stdio.h>
2 #include <mpi.h>
3 #include <iostream>
4 #include <vector>
5 #include <cmath>
6 // #include "modulos/funciones.h"
7
```



```
8 using namespace std;
9
10 int esPrimo(int n)
11 {
12     if(n <= 1) return 0;
13
14     int m = int(sqrt(n));
15
16     for (int j = 2 ; j <= m ; j++)
17         if (!(n%j)) return 0;
18     return 1;
19 }
20
21
22
23
24 int main(int argc, char **argv) {
25     int rank, size;
26     MPI_Init(&argc, &argv);
27     MPI_Status status;
28     MPI_Comm_rank(MPI_COMM_WORLD, &rank); //Paso por referencia rank y la
29     modifica
30     MPI_Comm_size(MPI_COMM_WORLD, &size);
31
32     double chunk;
33     double chunks[5] = {pow(10, 3), pow(10, 4), pow(10, 5), pow(10, 6), 2*
34     pow(10, 6)};
35     double limite = pow(10, 7);
36
37     double t1, tCalculo, tComm, tSync;
38     if(!rank) printf("Procesador,tCalculo,tSync,tComm,Chunk\n");
39     for(int repeticion = 0 ; repeticion <= 6 ; ++repeticion)
40     {
41         for (int ch = 0 ; ch < 5 ; ch++)
42         {
43             t1 = MPI_Wtime();
44             chunk = chunks[ch];
45             int n2=0, primesh=0, primes, n1=rank*chunk;
```

```
45         vector<int> extremos;
46
47         while (n1<limite)
48         {
49             n2 = n1 + chunk;
50             extremos.push_back(n1);
51             extremos.push_back(n2);
52             n1 += size*chunk;
53         }
54
55         for(int i = 0 ; i < extremos.size() ; i=i+2)
56         {
57             n1 = extremos[i];
58             n2 = extremos[i+1];
59             for(int n = n1 ; n < n2 ; n++)
60             {
61                 if(n >= limite) break;
62                 if(esPrimo(n)) primesh++;
63             }
64         }
65         tCalculo = MPI_Wtime()-t1;
66         tSync = MPI_Wtime();
67
68         MPI_Barrier(MPI_COMM_WORLD);
69         t1 = MPI_Wtime();
70         MPI_Reduce(&primesh, &primes, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
71         tComm = MPI_Wtime()-t1;
72         tSync = t1-tSync;
73
74         //if(!rank) printf("Primos hasta %f con chunk = %f: %d\n",
limite, chunk, primes);
75         printf("%d,%f,%f,%f,%f\n", rank, tCalculo, tSync, tComm, chunk
);
76     }
77 }
78 MPI_Finalize();
79 }
```

### **5.2.2 Ejecución en cluster**

Su ejecución se llevó a cabo en el cluster de la facultad de la misma forma que se desarrolló en el informe del trabajo práctico n° 1. Una vez finalizada su ejecución copié la salida a mi computadora personal para trabajar sobre ella.

### **5.2.3 Análisis**

Para analizar los datos obtenidos tras correr mi código en el cluster decidí realizar una gráfica de los tiempos que tardó en realizar las verificaciones de si los números que tiene asignados son primos o no y los tiempos que tardaron en sincronizarse. Esto se puede observar en la siguiente gráfica.

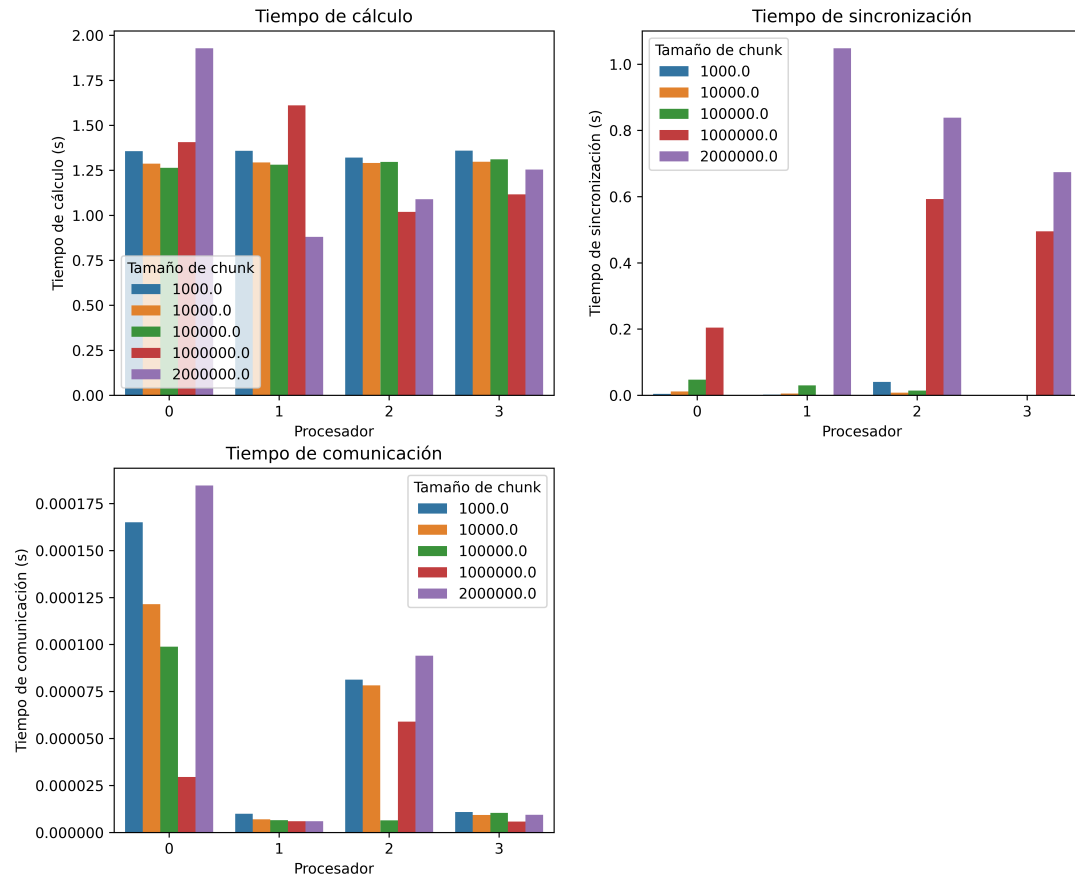


Fig. 7: Tiempos para los distintos procesadores

Como se puede observar en la figura 7 como se puede observar, la mayor diferencia se puede dar en el tiempo de sincronización, creciendo mucho para los tamaños más altos de chunk. Puede llamar la atención a primera vista la ausencia de la franja morada en 0 pero se aprecia que es el tiempo de cálculo más alto, lo cuál tiene sentido.

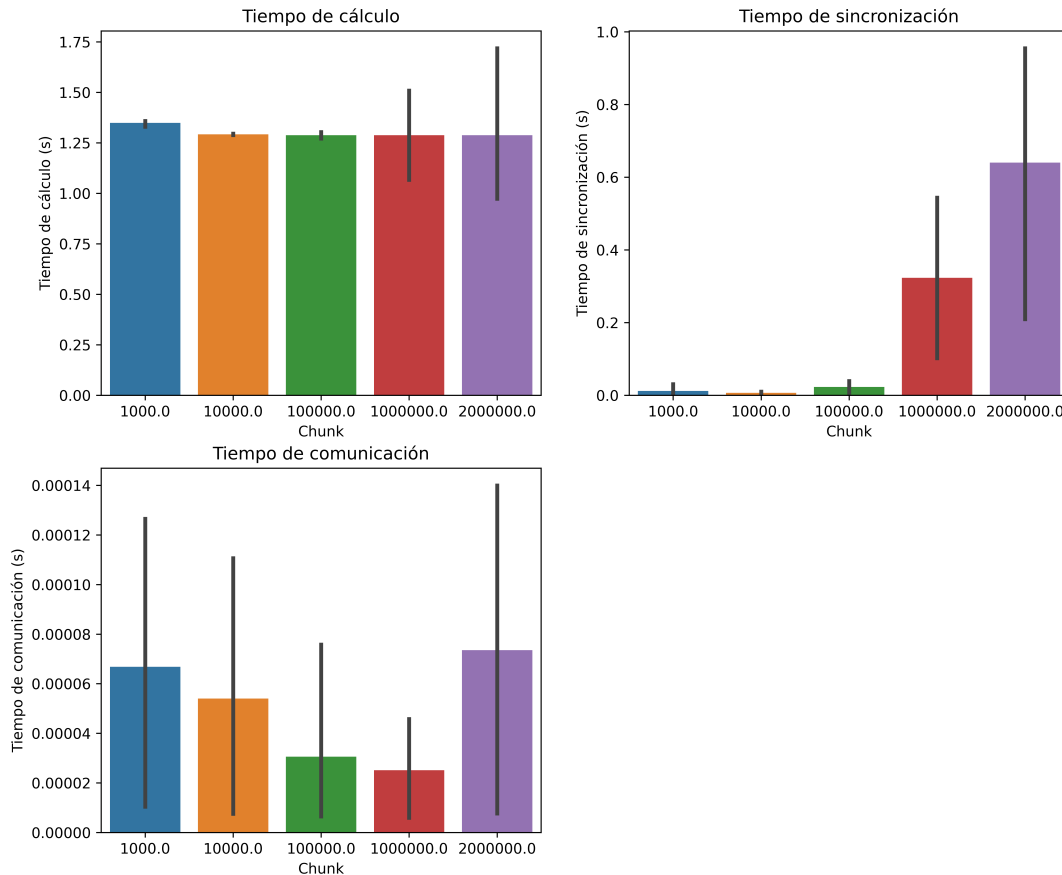


Fig. 8: Tiempos para distintos tamaños de chunk

Si comparamos los distintos tipos de tiempos para el tamaño de chunk, podemos ver que el tiempo de comunicación es casi despreciable, el del cálculo es consistente y en tamaños más grandes hay variabilidad pero tiene sentido por la escalabilidad del cálculo de los números primos. Además esto se verá aflojado en el crecimiento de los tiempos de sincronización como era de esperarse.

## Referencias

- [1] *pandas documentation — pandas 2.1.1 documentation*. URL: <https://pandas.pydata.org/docs/index.html> (visited on 10/12/2023).
- [2] *Notebook de validación*. URL: <https://github.com/justog220/Practica-CAR/blob/main/TP%202/EJ%202/validacion.ipynb> (visited on 10/12/2023).