
# Computación de alto rendimiento

Año: 2021

## TP2

## “USO BÁSICO DE MPI – Comunicaciones Colectivas”

Salim Taleb, Nasim A.

Docente: Garelli, Luciano

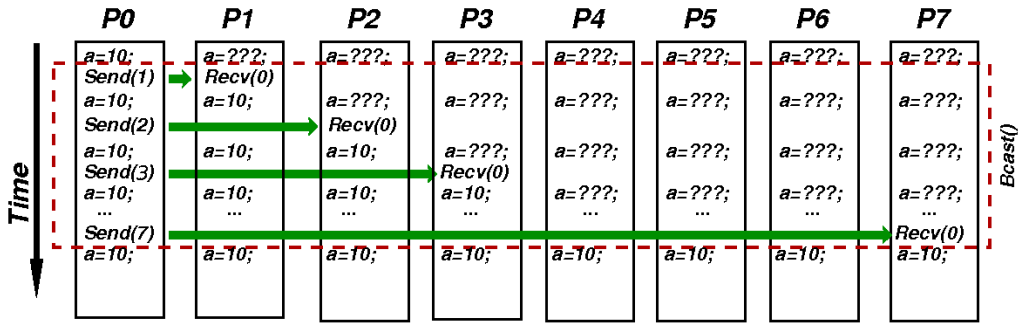
Carrera: Lic. en Bioinformática

# SEMINARIO DE CALCULO PARALELO

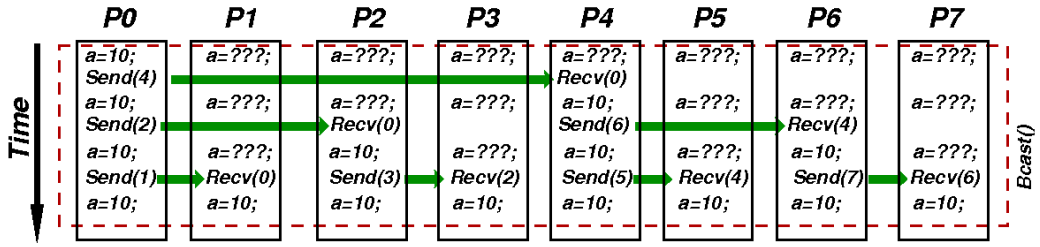
## GUIA DE TRABAJOS PRACTICOS No 2

## USO BASICO DE MPI – Comunicaciones Colectivas

1) 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.

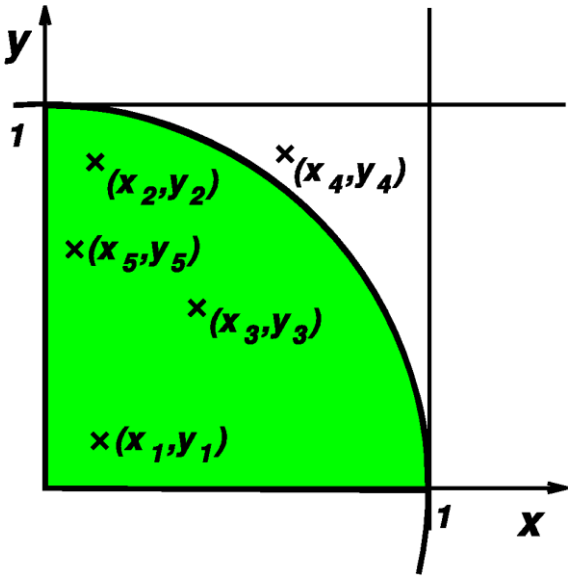



Forma secuencial.




Forma árbol.

2) Calcular el valor de π mediante el uso del método estadístico numérico de Monte Carlo.
Para tal fin, se generaran una secuencia de numero aleatorios (*xi* , *yi*), en donde la probabilidad de que el punto se encuentre dentro del círculo unitario es π/4.
De esta manera se puede estimar el valor de π mediante:

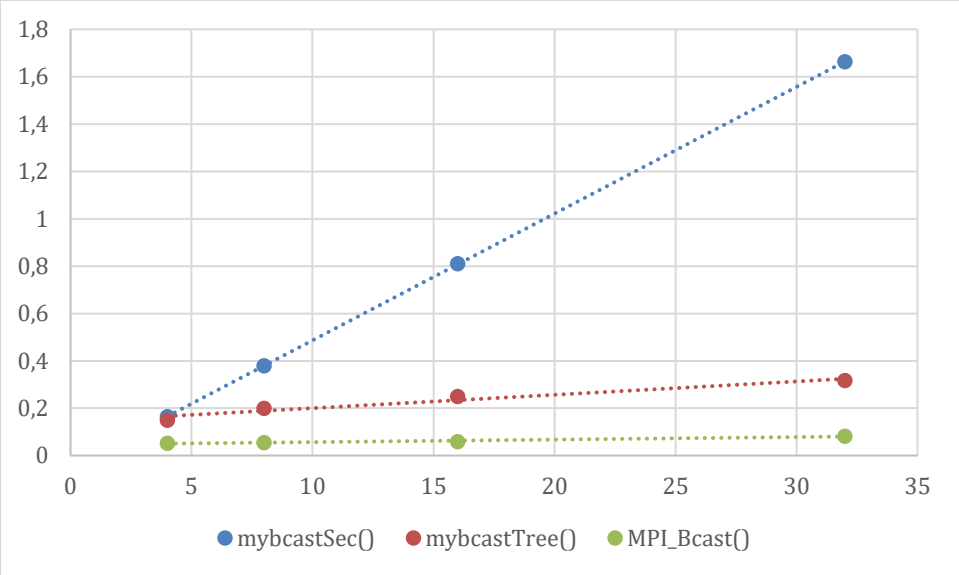



$$\pi = 4 * (\frac{ptos.adentro}{ptos.afuera})$$

Realizar un programa para su ejecución en paralelo, que utilice la función “comp_pi()” provista. Esta función tomará como argumentos el número de puntos a generar. Cada nodo deberá inicializar el generador de números random con diferente semilla, ya que, si se usa la misma en todos los nodos, la secuencia generada en los nodos será la misma.
Tomar tiempos para diferentes cantidades de números aleatorios generados y ver la convergencia del método.

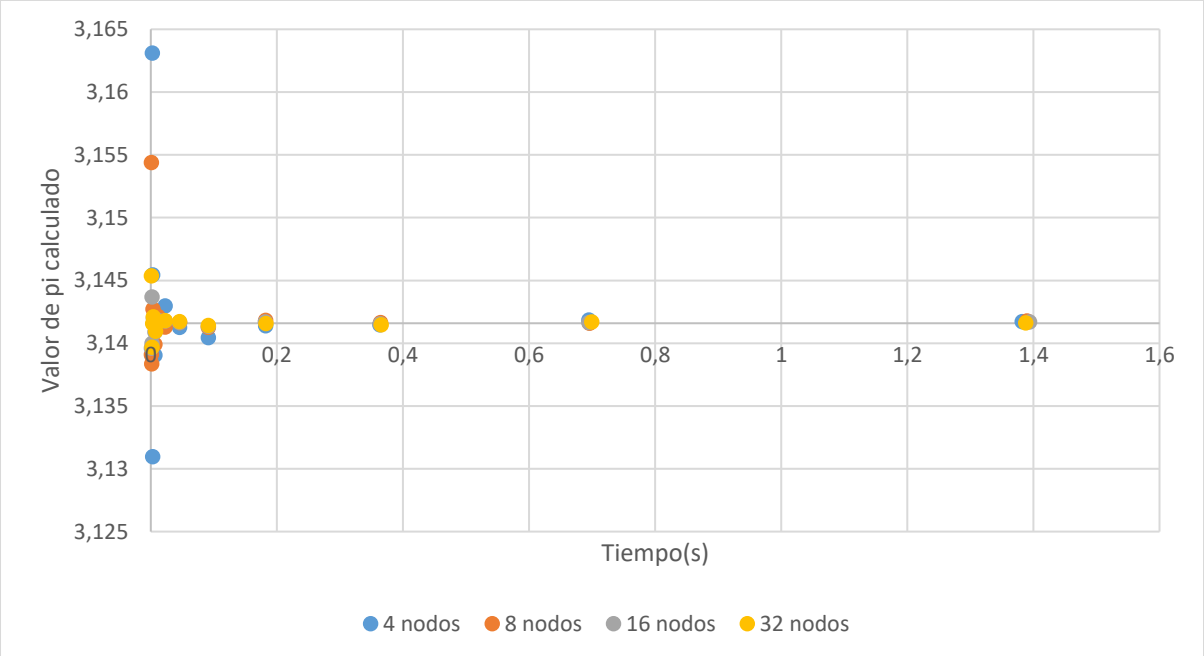
# Desarrollo

1) Después de ejecutar el código en un clúster se obtienen los siguientes resultados:

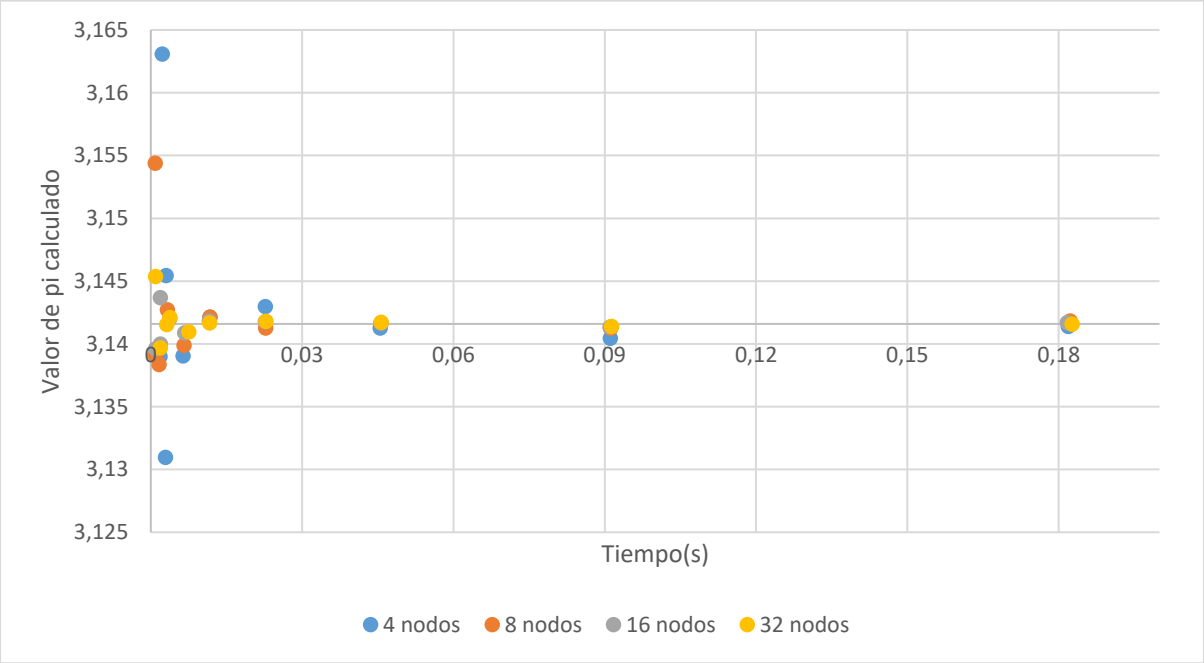



Siendo coherentes con la complejidad de los algoritmos, mybcastSec() al ser un envío de manera secuencia tiene une complejidad de O(n), en cambio, mybcastTree() y MPI_Bcast() tienen una complejidad de $O(\log_2 n)$ por estar estructuras en forma de árbol, el pequeño retardo en mybcastTree() con respecto a MPI_Bcast() puede deberse a que en el primero se envían los datos al nodo 0 para simplificar la función u otras optimizaciones que pueda tener la función.

2) Se corrió el código en un clúster externo con 1,2,4,8,16,32 nodos, se descartarán resultados de las corridas con 1 y 2 nodos porque resultaron iguales y solo cambió el tiempo, por lo que pudo haber algún error. El resultado del análisis es el siguiente:



Haciendo un pequeño acercamiento a la zona cercana al 0:




Se puede ver como los valores obtenidos en mayor cantidad de nodos son muchos mas exactos, dado que se utilizan más puntos, y no presentan una cantidad de tiempo muy superior en comparación al resto

# Códigos

Ejercicio1:

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

template<class T>
void mybcastSec(T *buff, long tam,MPI_Datatype tipo, int fuente,
MPI_Comm com)
{
    MPI_Status status;
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    //Se realiza un envio desde la fuente a cada nodo que no sea la
fuente
    for (int i=0;i<size;i++)
        if(i!=fuente)
            {
                if(rank==fuente)MPI_Send(buff,tam,tipo,i,i,com);

if(rank==i)MPI_Recv(buff,tam,tipo,fuente,i,com,&status);
            }
}

template<class T>
void mybcastTree(T *buff, long tam,MPI_Datatype tipo, int fuente,
MPI_Comm com)
{
    MPI_Status status;
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    int n1=0, n2=size;
    //Por simplicidad, si el nodo 0 no es la fuente se envia primero a
él los datos
    if (fuente!=0)
    {
        if(rank==0) MPI_Recv(buff,tam,tipo,fuente,0,com,&status);
        if(rank==fuente) MPI_Send(buff,tam,tipo,0,0,com);
    }
    //Se realiza un envio en forma de arbol a los nodos desde el 0
    while ((n2-n1)!=1)
    {
        int middle = (n1+n2)/2;
        if (rank==n1) MPI_Send(buff,tam,tipo,middle,middle,com);
        else if (rank==middle)
MPI_Recv(buff,tam,tipo,n1,middle,com,&status);
        if (rank<middle) n2 = middle;
        else n1=middle;
    }
}
```

```
int main(int argc, char **argv) {

    int rank, size;
    double starttime, endtime;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    double *buff= new double[20000000];

    MPI_Barrier(MPI_COMM_WORLD);
    starttime=MPI_Wtime();
    mybcastSec(buff, 20000000, MPI_DOUBLE, 2, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    endtime=MPI_Wtime();
    if(rank==0)printf("El tiempo para mybcastSec() fue de:
%f\n",endtime-starttime);


    MPI_Barrier(MPI_COMM_WORLD);
    starttime=MPI_Wtime();
    mybcastTree(buff, 20000000, MPI_DOUBLE, 2, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    endtime=MPI_Wtime();
    if(rank==0)printf("El tiempo para mybcastTree() fue de:
%f\n",endtime-starttime);


    MPI_Barrier(MPI_COMM_WORLD);
    starttime=MPI_Wtime();
    MPI_Bcast(buff, 20000000, MPI_DOUBLE, 2, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    endtime=MPI_Wtime();
    if(rank==0)printf("El tiempo para MPI_Bcast() fue de:
%f\n",endtime-starttime);

    MPI_Finalize();
}
```

Ejercicio2:

```
#include <stdio.h>
#include <mpi.h>
#include <random>
#define sqr(x)   ((x)*(x))

double comp_pi(int points)
{
    double x_coord,
    y_coord,
    pi,
    r;
    int inside,n;
    inside = 0;
    for (n = 1; n <= points; n++)
    {
        x_coord  = (double)random()/(double)RAND_MAX;
        y_coord = (double)random()/(double)RAND_MAX;
        if ((sqr(x_coord) + sqr(y_coord)) <= 1.0)inside++;
    }
    pi = 4.0 *(double)inside/(double)points;
    return(pi);
}

int main(int argc, char **argv) {

    int rank, size;
    double res, prom, starttime, endtime;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    srand(rank);
    for (int i=10000;i<50000002;i=i*2)
    //Se aumenta en cada iteracion la cantidad de puntos a calcular
por nodo
    {
        MPI_Barrier(MPI_COMM_WORLD);
        starttime=MPI_Wtime();
        double res=comp_pi(i);
        MPI_Reduce(&res,&prom,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
        endtime=MPI_Wtime();
        if (rank==0)
        {
            prom/=size;
            printf("%d,%f,%f;\n",i,endtime-starttime,prom);
        }
    }
    MPI_Finalize();
}
```