# **Resolución de la práctica calificada**

**TEORIA**

**1. ¿Qué es un proceso de una computadora?**

Nos referimos al conjunto de instrucciones que ejecutará el [microprocesador](https://www.ecured.cu/Microprocesador) mientras lee un programa determinado. Esto también implica a la memoria reservada y a sus contenidos, el estado de ejecución en determinado momento, y la información que permite al sistema operativo planificar.

**2. Comunicación punto a punto entre dos procesos**

El paso de mensajes entre dos procesos se basa en dos operaciones ENVIAR Y RECIBIR. En caso del uso del lenguaje de programación C/C++, cuando un proceso requiere enviar un mensaje a otro proceso, se usa la función MPI\_Send(…), y cuando se desea recibir se usa MPI\_Receive(…)

#include<iostream>

#include”mpi.h”

using namespace std;

int main(int arg, char \*args[]){

int rank,size;

MPI\_Init(&arg,&args);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

int number;

if (rank == 0) {

number = -1;

MPI\_Send(&number, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

} else if (rank == 1) {

MPI\_Receive(&number, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Proceso 1 received number %d from process 0\n",

number);

}

MPI\_Finalize();

}

**3. Memoria RAM, Memoria Cache y Memoria Compartida**

Memoria Ram: Random Access Memory o memoria de acceso aleatorio, es utilizada para procesar toda la información del sistema, en donde se ejecuta la mayor parte del software: el Sistema Operativo, el software de aplicación y otros programas similares.

Memoria Cache: (Memoria Auxiliar) Memoria de Acceso rápido a los recursos de CPU, permite el almacenamiento momentáneo de los datos recién procesados en el bufer, es más rápida que la RAM pero más pequeña que la misma. Beneficia al microprocesador.

Memoria Virtual: Simulación de la Memoria RAM que posee una máquina pero en mayor tamaño, es la gestión de la memoria que se encarga que el sistema operativo realiza para acceder a una mayor cantidad de memoria de la físicamente disponible

**4. Memoria distribuida y compartida**

Memoria Distribuida: Cada procesador tiene su memoria Local, permitiendo su acceso de datos más rápido, pero pueden transferir o intercambiar datos con un intercomunicador. Representan la creación hibrida de dos tipos de computación paralelos. Es una abstracción utilizada para compartir datos entre computadores que no comparten memoria física

Memoria Compartida: Cuando acceden a un mismo espacio de direcciones. Todos los procesadores están conectados a un intercomunicador para acceder a la memoria permitiendo así el intercambio de datos entre dos programas ejecutándose al mismo tiempo.

**5. Comandos de parámetro MPI**

MPI\_Send(

void\* data,//dirección del arreglo a enviar

int count, //tamaño del arreglo a enviar

MPI\_Datatype datatype, //tipo de dato a enviar

int destination, //pid del proceso que recibira

int tag, //etiqueta del mensaje

MPI\_Comm communicator) //comunicador

MPI\_Recv(

void\* data,//dirección del arreglo a recibir

int count, //tamaño del arreglo a recibir

MPI\_Datatype datatype, //tipo de dato

int source, //pid del proceso que envia

int tag, //etiqueta del mensaje

MPI\_Comm communicator, //comunicador

MPI\_Status\* status) // estructura estado del envio

MPI\_Reduce(

void\* data, //dirección del buffer de envio

void resultado, //dirección del buffer de recepcion

int count, //numero de elementos que se enviaran

MPI\_Datatype datatype, //tipo de dato

MPI\_OP operador, //operación de reduccion

int root, //proceso raíz o receptor

MPI\_Comm communicator) //comunicador

MPI\_Allreduce(

void\* send\_data,//dirección del buffer de envio

void\* recv\_data, //dirección del buffer de recepcion

int count, //numero de elementos que se enviaran

MPI\_Datatype datatype, //tipo de dato

MPI\_Op op, //operación de reduccion

MPI\_Comm communicator) //comunicador

**PRÁCTICA**

**6.**

#include <math.h>

#include <iostream>

using namespace std;

int main(int argc, char \*argv[])

{

int n,rank,size, count=0, elemento\_buscado, resultado;

int B[];

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

if (rank == 0) {

cout<<"Introduce la cantidad de elementos (n): ";

cin>>n;

cout<<"Elemento buscado es: ";

cin>>elemento\_buscado;

int A[n] = llenarVector(n);

MPI\_Bcast(&A,

n,

MPI\_INT,

0,

MPI\_COMM\_WORLD);

}else{

MPI\_Receive(&B,

n,

MPI\_INT,

0,

MPI\_COMM\_WORLD);

for(int i=rank\*n/size;i<(rank+1)\*n/size;i++){

if(B[i]==elemento\_buscado){

count++;

}

}

MPI\_Reduce(

&count,

&resultado,

1,

MPI\_INT,

MPI\_SUM,

0,

MPI\_COMM\_WORLD);

}

if(rank==0){

cout<<"El numero de veces que sale "<<elemento\_buscado<<" es: "<<resultado<<endl;

}

MPI\_Finalize();

return 0;

}

**7. Ejercicio**

#include <math.h>

#include "mpi.h" // Biblioteca de MPI

#include <cstdlib> // Incluido para el uso de atoi

#include <iostream>

using namespace std;

int main(int argc, char \*argv[])

{

int n, m // Numero de iteraciones

rank, // Identificador de proceso

size,

myprodFinal; // Numero de procesos

int mypi,

prod;

MPI\_Init(&argc, &argv); // Inicializamos los procesos

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); // Obtenemos el numero total de procesos

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // Obtenemos el valor de nuestro identificador

// Solo el proceso 0 va a conocer el numero de iteraciones que vamos a

// ejecutar para la aproximacion de PI

if (rank == 0) {

cout<<"introduce n: ";

cin>>n;

int array[n];

for (int i = rank + 1; i <= n; i += size) {

cout<<"introduce el valor"<<i<<" del array: ";

cin>>m;

array[i] =m;

cout<<endl<<x;

MPI\_Bcast(&array, // Puntero al dato que vamos a enviar

n, // Numero de datos a los que apunta el puntero

MPI\_INT, // Tipo del dato a enviar

0, // Identificacion del proceso que envia el dato

MPI\_COMM\_WORLD);

}

MPI\_Bcast(&n, // Puntero al dato que vamos a enviar

1, // Numero de datos a los que apunta el puntero

MPI\_INT, // Tipo del dato a enviar

0, // Identificacion del proceso que envia el dato

MPI\_COMM\_WORLD);

}

// El proceso 0 reparte al resto de procesos el numero de iteraciones

// que calcularemos

if (n <= 0){

MPI\_Finalize();

exit(0);

}else {

prod= 1;

for (int i = rank\*n/(rank+1) ; i <= n/rank-1; i += 1) {

prod\*=[i]\*[1+1];

cout<<endl<<x;

}

myprod = prod;

// Todos los procesos ahora comparten su valor local de PI,

// lo hacen reduciendo su valor local a un proceso

// seleccionada a traves de una operacion aritmetica.

MPI\_Reduce(&myprod, // Valor local de PI

&myprodFinal, // Dato sobre el que vamos a reducir el resto

1, // Numero de datos que vamos a reducir

MPI\_DOUBLE, // Tipo de dato que vamos a reducir

MPI\_PROD, // Operacion que aplicaremos

0, // proceso que va a recibir el dato reducido

MPI\_COMM\_WORLD);

// Solo el proceso 0 imprime el mensaje, ya que es la unica que

// conoce el valor de PI aproximado.

if (rank == 0)

cout<< "El valor aproximado del producto es: " << myprodFinal

<< endl;

}

// Terminamos la ejecucion de los procesos, despues de esto solo existira

// el proceso 0

// ¡Ojo! Esto no significa que los demas procesos no ejecuten el resto

// de codigo despues de "Finalize", es conveniente asegurarnos con una

// condicion si vamos a ejecutar mas codigo (Por ejemplo, con "if(rank==0)".

MPI\_Finalize();

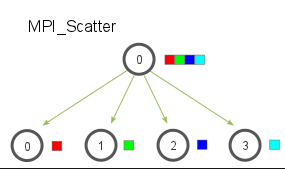
return 0;

}

**8. Ejercicio**

Para 8 procesos y 16 elementos.

**a.** Se puede usar la función MPI\_Scatter para enviar a cada de los 8 procesos. 2 elementos, y hacer operaciones



**b.** Se puede usar la función MPI\_Gather para recibir 2 elementos de los 8 procesos, y hacer operaciones

