Gabriela Moreno González.
3CV9 - Analysis and Design of Parallel Algorithms.
September 9, 2017
Práctica 1.

Envia y recibe en paralelo.
MPI_Send & MPI_Recv

El envío y recepción de mensajes es básico para cualquier lenguaje de programación. La parte de programación en paralelo es la que nos permite recibir y enviar datos de un conjunto de procesos que están resolviendo un pedazo del problema, lo que nos permitir reducir el tiempo de ejecución de n a n/x, donde x será nuestro número de procesos empleados por cada pedazo del problema.

En esta práctica se realizó un programa que permitiera a los procesos comunicarse entre sí, mandando mensajes desde el proceso 0 hasta el proceso que se indicara, mostrando en pantalla cuando un proceso recibía de otro.

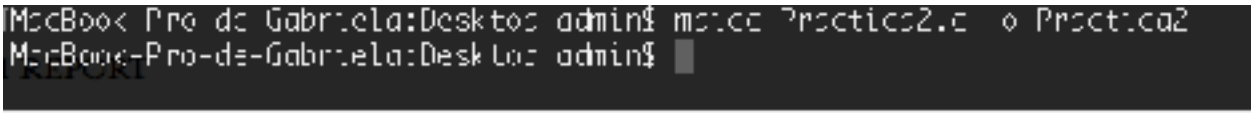
El código fuente empleado es el que se muestra en la siguiente imagen y nos muestra las funciones básicas para enviar y recibir mensajes mediante las funciones MPI_Send y MPI_Recv respectivamente. Como podemos observar en el código, el uso de estas funciones es muy similar a las funciones para manejar archivos en C de lectura y escritura, sin embargo aquí estamos manejando procesos.

También se ve que para mandar todos los procesos no fue necesario utilizar algún for que fuera enviando y recibiendo los datos, sino que simplemente comprobando que la pila se iba vaciando y cuando llegara a un tope mandara el último proceso para después finalizar con la ejecución fue suficiente.

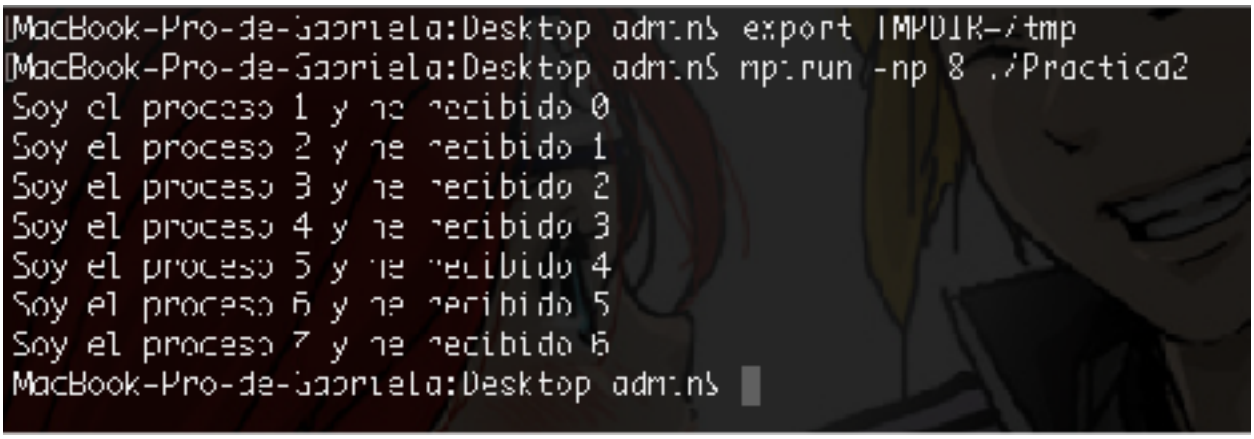
De igual forma podemos observar que ya se manejan los tipos de datos como MPI_INT, así como existen MPI_FLOAT, MPI_DOUBLE, etc.

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

int main(int argc, char *argv[])
{
    int rank, size, contador;
    MPI_Status estado;

    MPI_Init(&argc, &argv); // Inicializamos la comunicacion de los procesos
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Obtenemos el numero total de hebras
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Obtenemos el valor de nuestro identificador

    if(rank == 0)
    {
        MPI_Send(&rank //referencia al vector de elementos a enviar
                ,1 // tamaño del vector a enviar
                ,MPI_INT // Tipo de dato que envias
                ,rank+1 // pid del proceso destino
                ,0 //etiqueta
                ,MPI_COMM_WORLD); //Comunicador por el que se manda

    }
    else
    {
        MPI_Recv(&contador // Referencia al vector donde se almacenara lo recibido
                ,1 // tamaño del vector a recibir
                ,MPI_INT // Tipo de dato que recibe
                ,rank-1 // pid del proceso origen de la que se recibe
                ,0 // etiqueta
                ,MPI_COMM_WORLD // Comunicador por el que se recibe
                ,&estado); // estructura informativa del estado

        printf("Soy el proceso %d y he recibido %d\n",rank,contador);
        contador++;

        if(rank != size-1)
        {
            MPI_Send(&contador, 1 ,MPI_INT ,rank+1 , 0 ,MPI_COMM_WORLD);
        }
    }

    // Terminamos la ejecucion de las hebras, despues de esto solo existira
    // la hebra 0
    // ¡Ojo! Esto no significa que las demas hebras 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;
}
```

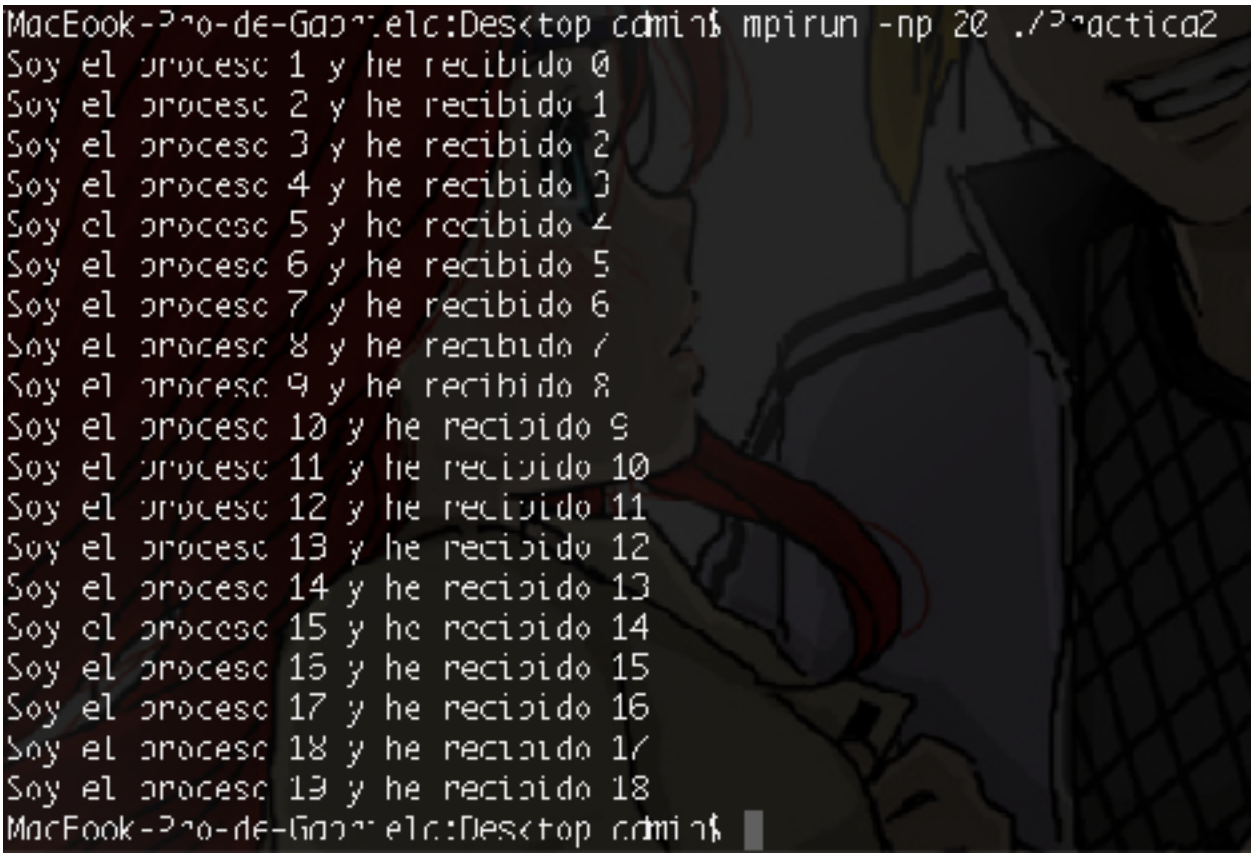
sAhora compilaremos el programa de la siguiente manera:



Y ejecutemos con 8 procesos:



¿Qué sucede si corremos con más procesos?



División de cálculos.

MPI_Bcast & MPI_Reduce

Los algoritmos paralelos nos permiten resolver problemas de forma eficaz debido a que dividen el cálculo de nuestro problema en pequeños sub problemas y todos son calculados a la par (se hace el cálculo de los n sub problemas en paralelo). En este programa se aprendió a realizar el cálculo de Pi mediante dividir dicho cálculo en varios sub procesos. El código empleado es:

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"   // Biblioteca de MPI

int main(int argc, char *argv[])
{
    int n, // Numero de iteraciones
        rank, // Identificador de proceso
        size; // Numero de procesos
    double PI25DT = 3.14159265358979323846264338327950288419716939937510;
    double mypi, // Valor local de PI
        pi,   // Valor globarl de PI
        h,   // Aproximacion del area para el calculo de PI
        sum;  // Acumulador para la suma del area de PI

    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)
    {
        printf("introduce la precision del calculo (n > 0): \n");
        scanf("%d",&n);
    }

    // El proceso 0 reparte al resto de procesos el numero de iteraciones
    // que calcularemos para la aproximacion de PI
    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);
    if (n <= 0){
        MPI_Finalize();
        exit(0);
    }else {
        // Calculo de PI
        h = 1.0 / (double) n;
        sum = 0.0;
        for (int i = rank + 1; i <= n; i += size) {
            double x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
        mypi = h * sum;

        // Todos los procesos ahora comparten su valor local de PI.
        // lo hacer recuciendo su valor local a un proceso
        // seleccionada a traves de una operacion aritmetica.
        MPI_Reduce(&mypi, // Valor local de PI
                   &pi,  // 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_SUM,  // 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)
            printf("El valor aproximado de PI es %lf, con un error de %lf\n", pi,fabs(pi - PI25DT));
    }

    // 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;
}
```

La ejecución del programa es la siguiente: