Práctica 2 - Algoritmos de nivel intermedio.

Moreno González Gabriela.

3CV9 - Analysis and Design of Parallel Algorithms.

October 9, 2017

Algoritmos paralelos de nivel intermedio.

Producto escalar, merge sort, multiplicación de matrices y comunicación en malla.

El paso hacia una programación de algoritmos de manera eficiente es empleando funciones más avanzadas y resolviendo problemas que justamente impliquen el uso de ellas. Así, los 4 programas que se desarrollaron conforme a la página son:

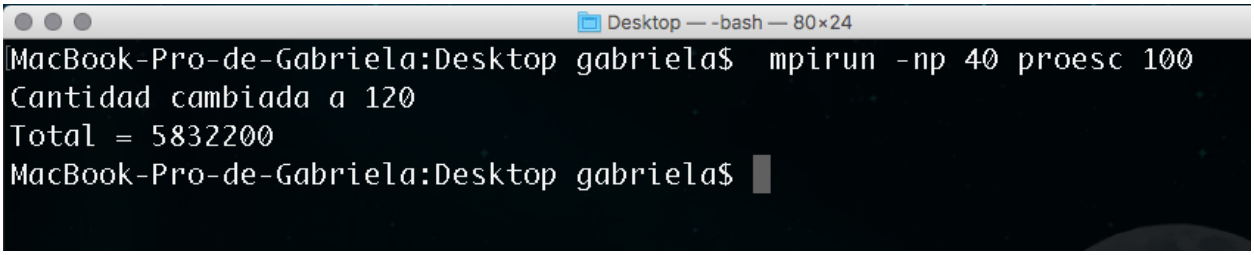
1. Producto escalar:

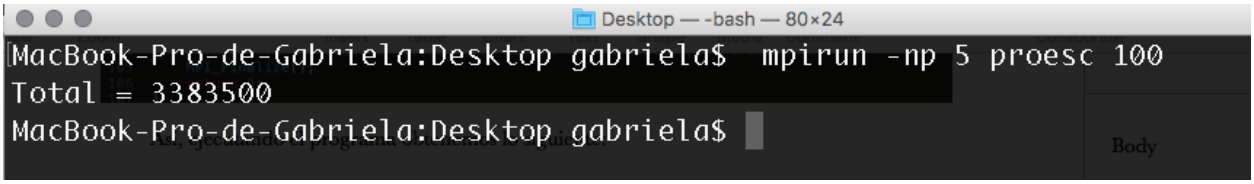
El código del programa es el siguiente:

```
#include "mpi.h"
#include <vector>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int tama, rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (argc < 2)
    {
        if (rank == 0)
        {
            printf(" No se ha especificado numero de elementos, multiplo de la cantidad de entrada, por defect
            cout << "\nUso: <ejecutable> <cantidad>" << endl;
        }
        tama = size * 100;
    }
    else
    {
        tama = atoi(argv[1]);
        if (tama < size)
            tama = size;
        else
        {
            int i = 1, num = size;

            while (tama > num)
            {
```

```
            ++i;
            num = size*i;
        }

        if (tama != num)
        {
            if (rank == 0)
                cout << "Cantidad cambiada a " << num << endl;
            tama = num;
        }
    }
}

// Creacion y relleno de los vectores
vector<long> VectorA, VectorB, VectorALocal, VectorBLocal;
VectorA.resize(tama, 0);
VectorB.resize(tama, 0);
VectorALocal.resize(tama/size, 0);
VectorBLocal.resize(tama/size, 0);
if (rank == 0) {
    for (long i = 0; i < tama; ++i) {
        VectorA[i] = i + 1; // Vector A recibe valores 1, 2, 3, ..., tama
        VectorB[i] = (i + 1)*10; // Vector B recibe valores 10, 20, 30, ..., tama*10
    }
}

// Repartimos los valores del vector A
MPI_Scatter(&VectorA[0], // Valores a compartir
        tama / size, // Cantidad que se envia a cada proceso
        MPI_LONG, // Tipo del dato que se enviara
        &VectorALocal[0], // Variable donde recibir los datos
        tama / size, // Cantidad que recibe cada proceso
        MPI_LONG, // Tipo del dato que se recibira
        0,  // proceso principal que reparte los datos
        MPI_COMM_WORLD); // Comunicador (En este caso, el global)
// Repartimos los valores del vector B
MPI_Scatter(&VectorB[0],
        tama / size,
        MPI_LONG,
        &VectorBLocal[0],
        tama / size,
        MPI_LONG,
        0,
        MPI_COMM_WORLD);

// Calculo de la multiplicacion escalar entre vectores
long producto = 0;
for (long i = 0; i < tama / size; ++i) {
    producto += VectorALocal[i] * VectorBLocal[i];
}
long total;

// Reunimos los datos en un solo proceso, aplicando una operacion
// aritmetica, en este caso, la suma.
MPI_Reduce(&producto, // Elemento a enviar
        &total, // Variable donde se almacena la reunion de los datos
        1, // Cantidad de datos a reunir
        MPI_LONG, // Tipo del dato que se reunira
        MPI_SUM, // Operacion aritmetica a aplicar
        0, // Proceso que recibira los datos
        MPI_COMM_WORLD); // Comunicador

if (rank == 0)
    cout << "Total = " << total << 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;
}
```

Así, ejecutando el programa obtenemos lo siguiente:





2. Producto Matriz Vector.

El código de este programa es el siguiente:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <mpi.h>

using namespace std;

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

    int numeroProcesadores,
            idProceso;
    long **A, // Matriz a multiplicar
            *x, // Vector que vamos a multiplicar
            *y, // Vector donde almacenamos el resultado
            *miFila, // La fila que almacena localmente un proceso
            *comprueba; // Guarda el resultado final (calculado secuencialmente), su valor
                        // debe ser igual al de 'y'

    double tInicio, // Tiempo en el que comienza la ejecucion
            tFin; // Tiempo en el que acaba la ejecucion

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numeroProcesadores);
    MPI_Comm_rank(MPI_COMM_WORLD, &idProceso);

    A = new long *[numeroProcesadores]; // Reservamos tantas filas como procesos haya
    x = new long [numeroProcesadores]; // El vector sera del mismo tamanio que el numero
    // de procesadores

    // Solo el proceso 0 ejecuta el siguiente bloque
    if (idProceso == 0) {
        A[0] = new long [numeroProcesadores * numeroProcesadores];
        for (unsigned int i = 1; i < numeroProcesadores; i++) {
            A[i] = A[i - 1] + numeroProcesadores;
```

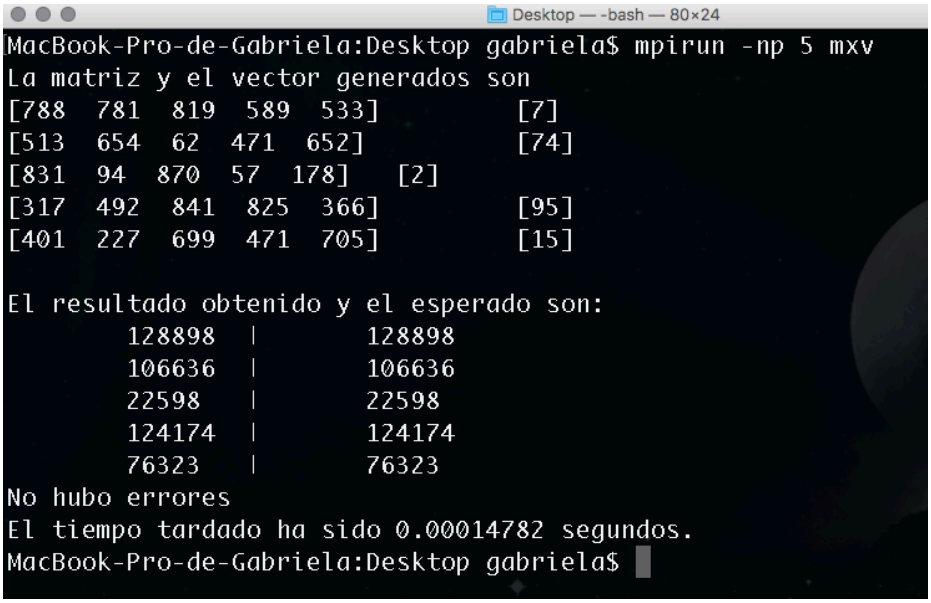
```
36          // Reservamos especio para el resultado
37          y = new long [numeroProcesadores];
38
39          // Rellenamos 'A' y 'x' con valores aleatorios
40          srand(time(0));
41          cout << "La matriz y el vector generados son " << endl;
42          for (unsigned int i = 0; i < numeroProcesadores; i++) {
43              for (unsigned int j = 0; j < numeroProcesadores; j++) {
44                  if (j == 0) cout << "[";
45                  A[i][j] = rand() % 1000;
46                  cout << A[i][j];
47                  if (j == numeroProcesadores - 1) cout << "]";
48                  else cout << "  ";
49              }
50              x[i] = rand() % 100;
51              cout << "\t  [" << x[i] << "]" << endl;
52          }
53          cout << "\n";
54
55          // Reservamos espacio para la comprobacion
56          comprueba = new long [numeroProcesadores];
57          // Lo calculamos de forma secuencial
58          for (unsigned int i = 0; i < numeroProcesadores; i++) {
59              comprueba[i] = 0;
60              for (unsigned int j = 0; j < numeroProcesadores; j++) {
61                  comprueba[i] += A[i][j] * x[j];
62              }
63          }
64      } // Termina el trozo de codigo que ejecuta solo 0
65
66      // Reservamos espacio para la fila local de cada proceso
67      miFila = new long [numeroProcesadores];
68
```

```
69      // Repartimos una fila por cada proceso, es posible hacer la reparticion de esta
70      // manera ya que la matriz esta creada como un unico vector.
71      MPI_Scatter(A[0], // Matriz que vamos a compartir
72              numeroProcesadores, // Numero de columnas a compartir
73              MPI_LONG, // Tipo de dato a enviar
74              miFila, // Vector en el que almacenar los datos
75              numeroProcesadores, // Numero de columnas a compartir
76              MPI_LONG, // Tipo de dato a recibir
77              0, // Proceso raiz que envia los datos
78              MPI_COMM_WORLD); // Comunicador utilizado (En este caso, el global)
79
80      // Compartimos el vector entre todas los procesos
81      MPI_Bcast(x, // Dato a compartir
82              numeroProcesadores, // Numero de elementos que se van a enviar y recibir
83              MPI_LONG, // Tipo de dato que se compartira
84              0, // Proceso raiz que envia los datos
85              MPI_COMM_WORLD); // Comunicador utilizado (En este caso, el global)
86
87
88      // Hacemos una barrera para asegurar que todas los procesos comiencen la ejecucion
89      // a la vez, para tener mejor control del tiempo empleado
90      MPI_Barrier(MPI_COMM_WORLD);
91      // Inicio de medicion de tiempo
92      tInicio = MPI_Wtime();
93
94      long subFinal = 0;
95      for (unsigned int i = 0; i < numeroProcesadores; i++) {
96          subFinal += miFila[i] * x[i];
97      }
98
99      // Otra barrera para asegurar que todas ejecuten el siguiente trozo de codigo lo
100     // mas proximamente posible
101     MPI_Barrier(MPI_COMM_WORLD);
```

```
103     tFin = MPI_Wtime();
104
105     // Recogemos los datos de la multiplicacion, por cada proceso sera un escalar
106     // y se recoge en un vector, Gather se asegura de que la recoleccion se haga
107     // en el mismo orden en el que se hace el Scatter, con lo que cada escalar
108     // acaba en su posicion correspondiente del vector.
109     MPI_Gather(&subFinal, // Dato que envia cada proceso
110             1, // Numero de elementos que se envian
111             MPI_LONG, // Tipo del dato que se envia
112             y, // Vector en el que se recolectan los datos
113             1, // Numero de datos que se esperan recibir por cada proceso
114             MPI_LONG, // Tipo del dato que se recibira
115             0, // proceso que va a recibir los datos
116             MPI_COMM_WORLD); // Canal de comunicacion (Comunicador Global)
117
118     // Terminamos la ejecucion de los procesos, despues de esto solo existira
119     // el proceso 0
120     // Ojo! Esto no significa que los demas procesos no ejecuten el resto
121     // de codigo despues de "Finalize", es conveniente asegurarnos con una
122     // condicion si vamos a ejecutar mas codigo (Por ejemplo, con "if(rank==0)".
123     MPI_Finalize();
124
125     if (idProceso == 0) {
126
127         unsigned int errores = 0;
128
129         cout << "El resultado obtenido y el esperado son:" << endl;
130         for (unsigned int i = 0; i < numeroProcesadores; i++) {
131             cout << "\t" << y[i] << "\t|\t" << comprueba[i] << endl;
132             if (comprueba[i] != y[i])
133                 errores++;
134         }
135
```

```
136         delete [] y;
137         delete [] comprueba;
138         delete [] A[0];
139
140         if (errores) {
141             cout << "Hubo " << errores << " errores." << endl;
142         } else {
143             cout << "No hubo errores" << endl;
144             cout << "El tiempo tardado ha sido " << tFin - tInicio << " segundos." << endl;
145         }
146
147     }
148
149     delete [] x;
150     delete [] A;
151     delete [] miFila;
152
153 }
```

Así, corriendo el programa podemos observar lo siguiente:



3. Merge sort

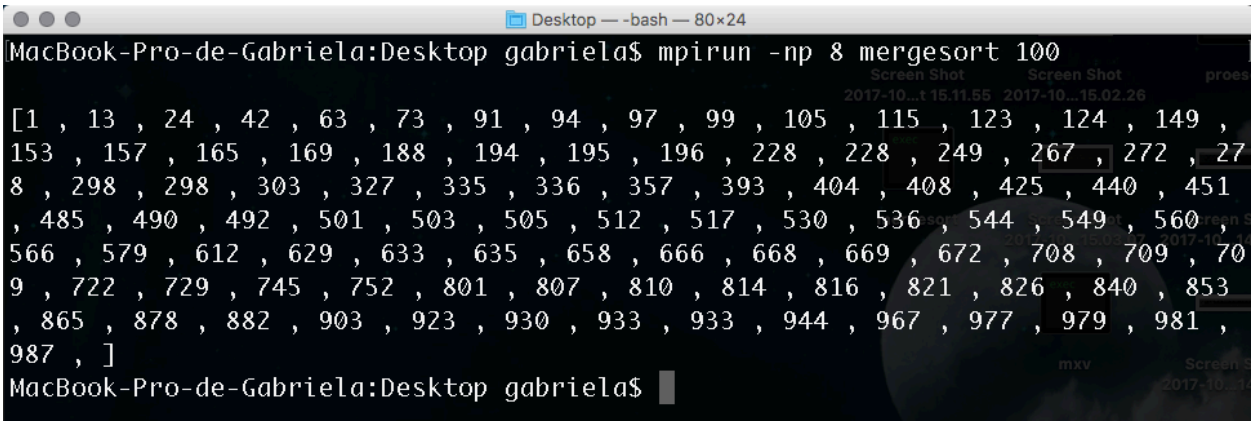
Es un algoritmo que sirve para ordenar números, es muy fácil de paralelizar. El código es el siguiente:

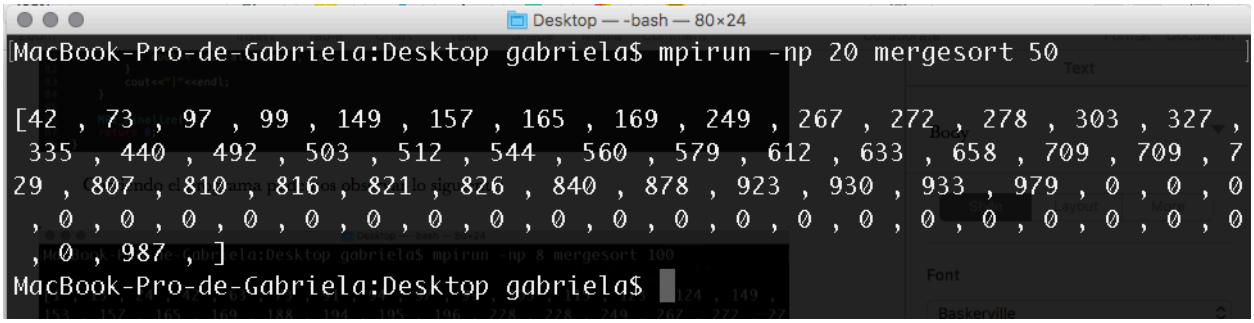
```cpp
#include <algorithm>
#include <vector>
#include "mpi.h"
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int rank, size, tama;
    vector<int> Global;//Vector a ordenar
    vector<int> *Local;//parte del vector

    MPI_Init(&argc, &argv);//iniciamos el entorno MPI
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);//obtenemos el identificador del proceso
    MPI_Comm_size(MPI_COMM_WORLD,&size);//obtenemos el numero de procesos

    if( size % 2 != 0 ){//El numero de procesos deberia debe ser par para aplicar este algoritmo
        cout << "El numero de procesos debe ser par" << endl;
        MPI_Abort(MPI_COMM_WORLD,1);//abandonamos la ejecucion.
    }

    if(argc < 2){
        if(rank == 0)
            cout<< "No recibio parametro con el tamaño de vector, por defecto sera 1000"<<endl;
        tama = 1000;
    }else{
        tama = atoi(argv[1]);
    }

    if(rank == 0){//el proceso 0 genera un vector desordenado.
        for(int i = 0; i < tama;++i){
            Global.push_back(rand()%1000);
        }
    }
```

```cpp
    Local = new vector<int>(tama/size);// reservamos espacio para el vector local a cada proceso.

    //Repartimos el vector entre todos los procesos.

MPI_Scatter(&Global[0],tama/size,MPI_INT,&((*Local)[0]),tama/size,MPI_INT,0,MPI_COMM_WORLD);

    //Cada proceso ordena su parte.
    sort(Local->begin(),Local->end());

    vector<int> *ordenado;
    MPI_Status status;
    int paso = 1;

    //Ahora comienza el proceso de mezcla.
    while(paso<size)
    {
    // Cada pareja de procesos
        if(rank%(2*paso)==0) // El izquierdo recibe el vector y mezcla
        {
            if(rank+paso<size)// los procesos sin pareja esperan.
            {
                vector<int> localVecino(Local->size());
                ordenado = new vector<int>(Local->size()*2);


MPI_Recv(&localVecino[0],localVecino.size(),MPI_INT,rank+paso,0,MPI_COMM_WORLD,&status);
                merge(
Local->begin(),Local->end(),localVecino.begin(),localVecino.end(),ordenado->begin() );

                delete Local;
                Local = ordenado;
                ordenado = NULL;
            }
        }
```

```
69          else // El derecho envia su vector ordenado y termina
70          {
71              int vecino = rank-paso;
72              MPI_Send(&((*Local)[0]),Local->size(),MPI_INT,vecino,0,MPI_COMM_WORLD);
73              break;//Sale del bucle
74          }
75          paso = paso*2;// el paso se duplica ya que el numero de procesos se reduce a la mitad.
76      }
77
78      if(rank == 0){
79          cout<<endl<<"[";
80          for(unsigned int i = 0; i<Local->size();++i){
81              cout<< (*Local)[i]<<" , ";
82          }
83          cout<<"]"<<endl;
84      }
85
86      MPI_Finalize();
87      return 0;
88  }
```

Corriendo el programa podemos observar lo siguiente:





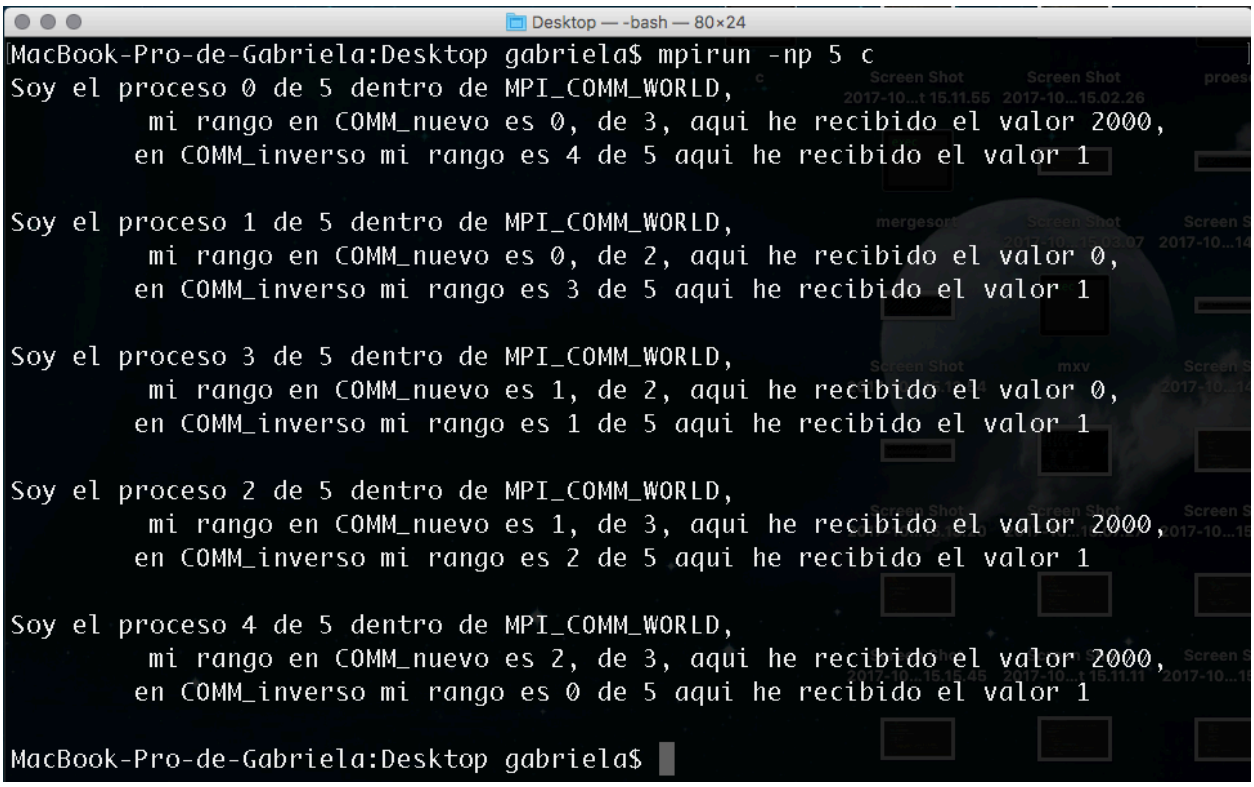
4. Comunicadores.

El código para efectuar la comunicación entre procesos es el siguiente:

```cpp
#include "mpi.h"
#include <vector>
#include <cstdlib>
#include <iostream>
using namespace std;

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

    MPI_Init(&argc, &argv); //iniciamos el entorno MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //obtenemos el identificador del proceso
    MPI_Comm_size(MPI_COMM_WORLD, &size); //obtenemos el numero de procesos

    MPI_Comm comm // nuevo comunicador para pares o impares
            , comm_inverso; // nuevo para todos los procesos pero con rank inverso.
    int rank_inverso, size_inverso;
    int rank_nuevo, size_nuevo;
    int a;
    int b;

    if (rank == 0) {
        a = 2000;
        b = 1;
    } else {
        a = 0;
        b = 0;
    }

    int color = rank % 2;
    // creamos un nuevo cominicador
    MPI_Comm_split(MPI_COMM_WORLD // a partir del comunicador global.
            , color // los del mismo color entraran en el mismo comunicador
            // lo pares tiene color 0 y los impares 1.
            , rank, // indica el orden de asignacion de rango dentro de los nuevos comunicadores
```

```cpp
            &comm); // Referencia al nuevo comunicador creado.
    // creamos un nuevo comunicador inverso.
    MPI_Comm_split(MPI_COMM_WORLD, // a partir del comunicador global.
            0 // el color es el mismo para todos.
            , -rank // el orden de asignacion para el nuevo rango es el inverso al actual.
            , &comm_inverso); // Referencia al nuevo comunicador creado.

    MPI_Comm_rank(comm, &rank_nuevo); // obtenemos el nuevo rango asignado dentro de comm
    MPI_Comm_size(comm, &size_nuevo); // obtenemos numero de procesos dentro del comunicador

    MPI_Comm_rank(comm_inverso, &rank_inverso); // obtenemos el nuevo rango asignado en comm_inverso
    MPI_Comm_size(comm_inverso, &size_inverso); // obtenemos numero de procesos dentro del comunicador

    //Probamos a enviar datos por distintos comunicadores
    MPI_Bcast(&b, 1, MPI_INT,
            size - 1, // el proceso con rango 0 dentro de MPI_COMM_WORLD sera root
            comm_inverso);
    MPI_Bcast(&a, 1, MPI_INT,
            0, // el proceso con rango 0 dentro de comm sera root
            comm);

    cout << "Soy el proceso " << rank << " de " << size << " dentro de MPI_COMM_WORLD,"
            "\n\t mi rango en COMM_nuevo es " << rank_nuevo << ", de " << size_nuevo <<
            ", aqui he recibido el valor " << a <<
            ",\n\ten COMM_inverso mi rango es " << rank_inverso << " de " << size_inverso
<<
            " aqui he recibido el valor " << b <<"\n"<< endl;


    MPI_Finalize();
    return 0;
}
```

Corriendo el programa obtenemos lo siguiente:


Desktop — -bash — 80×24
MacBook-Pro-de-Gabriela:Desktop gabriela$ mpirun -np 5 c
Soy el proceso 0 de 5 dentro de MPI_COMM_WORLD,
mi rango en COMM_nuevo es 0, de 3, aqui he recibido el valor 2000,
en COMM_inverso mi rango es 4 de 5 aqui he recibido el valor 1
Soy el proceso 1 de 5 dentro de MPI_COMM_WORLD,
mi rango en COMM_nuevo es 0, de 2, aqui he recibido el valor 0,
en COMM_inverso mi rango es 3 de 5 aqui he recibido el valor 1
Soy el proceso 3 de 5 dentro de MPI_COMM_WORLD,
mi rango en COMM_nuevo es 1, de 2, aqui he recibido el valor 0,
en COMM_inverso mi rango es 1 de 5 aqui he recibido el valor 1
Soy el proceso 2 de 5 dentro de MPI_COMM_WORLD,
mi rango en COMM_nuevo es 1, de 3, aqui he recibido el valor 2000,
en COMM_inverso mi rango es 2 de 5 aqui he recibido el valor 1
Soy el proceso 4 de 5 dentro de MPI_COMM_WORLD,
mi rango en COMM_nuevo es 2, de 3, aqui he recibido el valor 2000,
en COMM_inverso mi rango es 0 de 5 aqui he recibido el valor 1
MacBook-Pro-de-Gabriela:Desktop gabriela$