Rubén Vera - 241456
Sara Soriano - 240007

REPORT LAB2

Exercise 1.

- **PHASE 1: Using the communicator MPI COMM WORLD print the rank of each process, the name of the communicator and the total amount of processes.**

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

int main( int argc, char *argv[] ) {
  int rank, size;
  MPI_Init( &argc, &argv );
  char processor_name[MPI_MAX_OBJECT_NAME];
  int name_len;
  MPI_Comm SPLIT_COMM_;
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_get_name(MPI_COMM_WORLD, processor_name, &name_len);
  if(rank == 0){
    printf("PHASE 1\n\n");
  }
  for(int i=0; i<size; ++i){
    if(rank==i) printf( "Hi, I'm rank %d. My communicator is %s and has a size of %d processes.\n", rank, processor_name ,size );
    fflush(stdout);
    MPI_Barrier(MPI_COMM_WORLD);
  }
  fflush(stdout);
  MPI_Barrier(MPI_COMM_WORLD);
```

Primerament, amb el MPI_Comm_rank() i el MPI_Comm_size() mirem quin rang té cada procés i el guardem a la variable rank i el número de processos el qual el guardem a la variable size. A més amb el MPI_Comm_get_name() es guarda el nom del comunicador de cada procés a la variable processor name i a name_len guardarem la mida del nom del comunicador. Després fem un fflush() per tal que es buidi el buffer del print i fem una barrera per a que tots els processos s'esperin abans de començar la següent iteració o abans de sortir del for.

- **PHASE 2: Split the communicator MPI COMM WORLD into 4 sub-communicators with 4 processes each. Name each sub-communicator differently. For each process, print its rank in the MPI COMM WORLD communicator and in the new one.**

```c
  if(rank == 0){
    printf("\nPHASE 2\n\n");
  }
  int color = rank/4;
  char processor_name_2[MPI_MAX_OBJECT_NAME];
  char comm_name[MPI_MAX_OBJECT_NAME];
  int x = 0;
  int rank_comm_2,size_comm_2;
  MPI_Comm_split(MPI_COMM_WORLD, color, x,&SPLIT_COMM_);
  snprintf(processor_name_2, sizeof(processor_name_2), "SPLIT_COMM_%d", color);
  MPI_Comm_set_name(SPLIT_COMM_, processor_name_2);
  MPI_Comm_get_name(SPLIT_COMM_, comm_name, &name_len);
  MPI_Comm_rank(SPLIT_COMM_, &rank_comm_2);
  MPI_Comm_size(SPLIT_COMM_, &size_comm_2);
  for(int i = 0; i < size; i++){
    if(rank == i) printf("Hi, I was rank %d. My communicator is %s and has a size of %d processes. Now I'm rank %d in communicator %s which has %d processes.
\n",rank, processor_name, size, rank_comm_2, comm_name, size_comm_2);
    // fflush(stdout);
    MPI_Barrier(MPI_COMM_WORLD);
  }
  fflush(stdout);
  MPI_Barrier(MPI_COMM_WORLD);
```

Tal i com es demana en l'enunciat i es veu en l'anterior captura, el que es fa en aquesta fase és dividir el comunicador MPI_COMM_WORLD en quatre comunicadors diferents. Per fer-ho hem utilitzat el MPI_Comm_split(), per tal d'obtenir 4 comunicadors amb quatre processos cadascun. Color = rank/4 és

el número de comunicador al que és assignat cada procés. Després, simplement hem anat assignant el nombre corresponent a cada comunicador. Per finalitzar, hem seguit els mateixos passos que en la fase anterior.

- **PHASE 3: Create a group of processes containing the even ranks. Create a communicator for the group called EVEN COMM. For each process, print its rank in the PHASE 2 communicator and in the new communicator.**

```
  if(rank == 0){
    printf("\nPHASE 3\n\n");
  }
  char comm_name_3[MPI_MAX_OBJECT_NAME];
  char comm_set_name_3[MPI_MAX_OBJECT_NAME];
  snprintf(comm_set_name_3,sizeof(comm_set_name_3), "EVEN_COMM");
  MPI_Group group_world;
  MPI_Group even_group;
  int group[size/2];
  for(int i = 0; i < size; i++){
    if(i%2 == 0) group[i/2] = i;
  }
  MPI_Comm EVEN_COMM;
  MPI_Comm_group(MPI_COMM_WORLD, &group_world);
  MPI_Group_incl(group_world, size/2, group, &even_group);
  int size_even_comm, rank_even_comm;
  MPI_Comm_create_group(MPI_COMM_WORLD, even_group,0, &EVEN_COMM);
  if(EVEN_COMM == MPI_COMM_NULL);
  else{
    MPI_Comm_set_name(EVEN_COMM, comm_set_name_3);
    MPI_Comm_get_name(EVEN_COMM, comm_name_3, &name_len);
    MPI_Comm_rank(EVEN_COMM, &rank_even_comm);
    MPI_Comm_size(EVEN_COMM, &size_even_comm);
  }
  for(int i = 0; i < size; i++){
    if(rank == i && i%2==0) printf("Hi, I was rank %d in communicator %s which had %d processes. Now I'm rank %d in communicator %s which has %d processes.\n
", rank_comm_2, comm_name, size_comm_2, rank_even_comm, comm_name_3, size_even_comm);
    MPI_Barrier(MPI_COMM_WORLD);
  }
  fflush(stdout);
  MPI_Barrier(MPI_COMM_WORLD);
```

Aquesta fase és una mica més complexa que les anteriors, ja que hem d'agafar només uns processos en concret. Especificament, només volem que el nou comunicador contingui els processos amb rang parell.
Per fer-ho creem un grup que contingui els processos del MPI_COMM_WORLD. Posteriorment, es crea el grup on, al MPI_Group_incl(), col·locarem només els parells. Després, es crea el comunicador amb MPI_Comm_create_group() a partir dels processos de even_group, on li assignarem el seu nom com hem fet prèviament. I, finalment, farem un print amb el que se'ns demana.

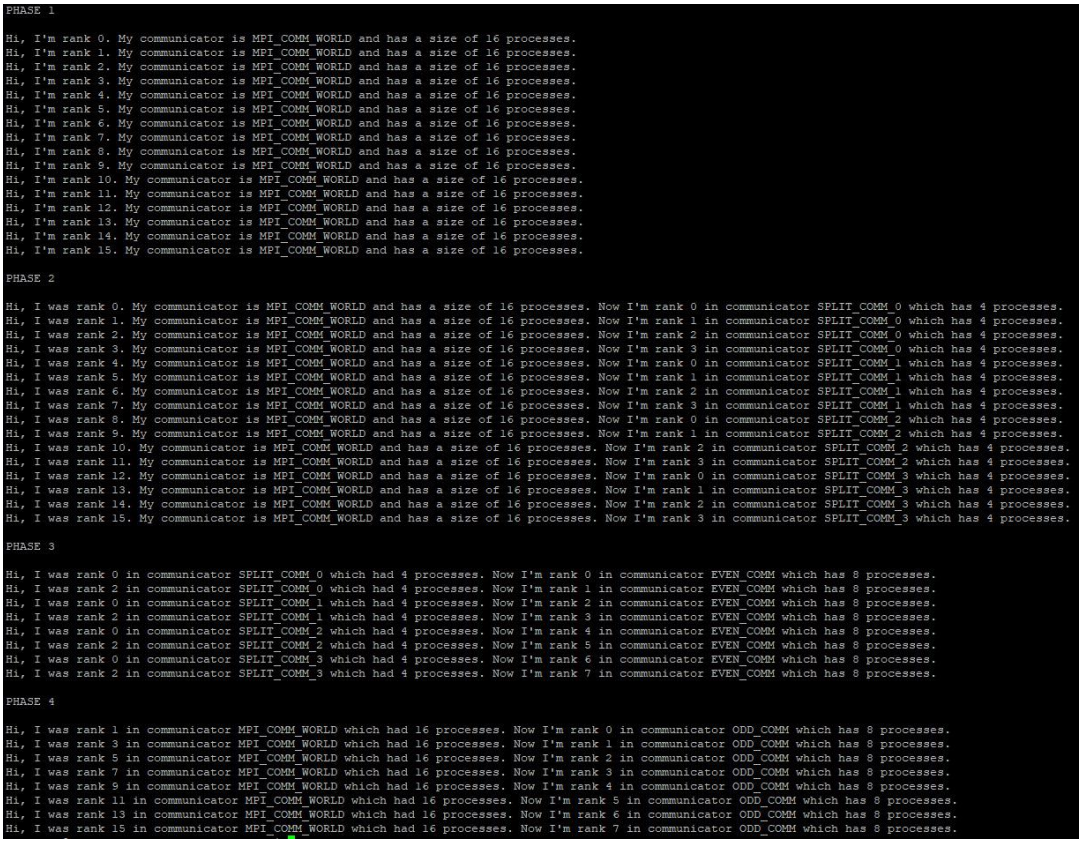
- **PHASE 4: Create the group of processes in MPI COMM WORLD that are not in EVEN COMM using MPI Group functionalities. Create a communicator for this new group called ODD COMM. Print the rank of each process in the MPI COMM WORLD and in ODD COMM.**

```
  if(rank == 0){
    printf("\nPHASE 4\n\n");
  }
  char comm_name_4[MPI_MAX_OBJECT_NAME];
  char comm_set_name_4[MPI_MAX_OBJECT_NAME];
  snprintf(comm_set_name_4,sizeof(comm_set_name_3), "ODD_COMM");
  MPI_Group odd_group;
  for(int i = 0; i < size; i++){
    if(i%2 != 0) group[i/2] = i;
  }
  MPI_Comm ODD_COMM;
  MPI_Group_incl(group_world, size/2, group, &odd_group);
  int size_odd_comm, rank_odd_comm;
  MPI_Comm_create_group(MPI_COMM_WORLD, odd_group,0, &ODD_COMM);
  if(ODD_COMM == MPI_COMM_NULL);
  else{
    MPI_Comm_set_name(ODD_COMM, comm_set_name_4);
    MPI_Comm_get_name(ODD_COMM, comm_name_4, &name_len);
    MPI_Comm_rank(ODD_COMM, &rank_odd_comm);
    MPI_Comm_size(ODD_COMM, &size_odd_comm);
  }
  for(int i = 0; i < size; i++){
    if(rank == i && i%2!=0) printf("Hi, I was rank %d in communicator %s which had %d processes. Now I'm rank %d in communicator %s which has %d processes.\n
", rank, processor_name, size, rank_odd_comm, comm_name_4, size_odd_comm);
    MPI_Barrier(MPI_COMM_WORLD);
  }
  fflush(stdout);
  MPI_Barrier(ODD_COMM);
  MPI_Finalize();
  return 0;
}
```

Rubén Vera - 241456
Sara Soriano - 240007

Per dur a terme la darrera fase s'ha seguit els mateixos passos en que en la fase anterior, amb la diferència de que ara no s'agafen els processos amb rang parell sino els senars. I l'altre diferència és a l'hora d'imprimir que es mostra el rang inicial de MPI_COMM_WORLD, a diferència del rang de SPLIT_COMM_.

El resultat final de totes aquestes quatre fases és el que es mostra en la següent imatge:



Rubén Vera - 241456
Sara Soriano - 240007

Exercise 2.

- **Write a function in C that reads a binary file in parallel. The function must have the structure specified.**

```c
double *par_read(char *in_file, int *p_size, int rank , int nprocs){
  MPI_File fh;
  MPI_Offset filesize;
  MPI_Status status;
  double *par_read;

  MPI_File_open(MPI_COMM_WORLD, in_file, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
  MPI_File_get_size(fh, &filesize);

  filesize = filesize / sizeof(double);
  (*p_size) = filesize/nprocs;
  par_read = (double *) malloc(*p_size*sizeof(double));

  MPI_File_set_view(fh, rank*(*p_size)*sizeof(double), MPI_DOUBLE, MPI_DOUBLE, "native", MPI_INFO_NULL);
  MPI_File_read(fh, par_read, *p_size, MPI_DOUBLE, &status);

  MPI_File_close(&fh);
  return par_read;
}
```

Primerament, obrim el arxiu amb nom in_file en el mode “Read Only”, i guardarem el “handle” a fh. Seguidament, guardarem la mida del fitxer a la variable filesize. La funció MPI_File_get_size() retorna la mida en bytes, per tant, hem de dividir el resultat entre la “size of doubles”.
Seguidament, es calcula el offset que serà el p_size, és a dir, el tamany de l’arxiu entre el número de processos per tal que cada procés tingui la mateixa càrrega de treball. Aleshores, guardem memoria suficient per a que tots puguin emmagatzemar tot el que llegiran a continuació. Es fa que cadascun col·loqui el seu punter amb MPI_File_set_view() a on li toca, és a dir, a rank*p_size*sizeof(double). Un cop tenen el punter col·locat, començaran a llegir un total de p_size elements.

- **Use the function to read the files in the paths ”/shared/Labs/Lab 2/array p.bin” and ”/shared/Labs/Lab 2/array q.bin”. The binary files store doubles. Parallelize the dot product using MPI. Combine it with the shared memory version (lab1) to obtain a hybrid version.**

```c
void main(int argc, char* argv[]){
  int size, rank;
  int p_size;
  double *p, *q;
  double total, final_v;
  total = 0;
  final_v = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  p = par_read("/shared/Labs/Lab_2/array_p.bin", &p_size, rank, size);
  q = par_read("/shared/Labs/Lab_2/array_q.bin", &p_size, rank, size);
  double start = omp_get_wtime();
  #pragma omp parallel for simd reduction(+:total)
  for(int i = 0; i < p_size; i++){
    total+=p[i]*q[i];
  }
  MPI_Allreduce(&total, &final_v, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  double end = omp_get_wtime();
  double time = end - start;
  printf("%.4e\t%f\n", time, final_v);

  MPI_Finalize();
}
```

Per crear els nous vector p i q hem utilitzat la funció que hem creat en l'apartat anterior, cridant amb els fitxers corresponents que se'ns indica en l'enunciat de la pràctica. Per continuar, hem agafat la versió amb memòria compartida del Lab1. Per tal d'obtenir una versió híbrida hem hagut d'inicialitzar el MPI (amb MPI_Init()), on a partir de les funcions MPI_Comm_size() i MPI_Comm_rank() hem obtingut la informació necessària per cridar a la nostra funció. A més a més, per obtenir el resultat final hem hagut de realitzar un MPI_Allreduce() on la variable total de cada procés s'anirà sumant amb la resta i es guardarà el resultat en final_v. Per acabar, simplement hem hagut de tancar el MPI, amb la funció corresponent.

- **Submit a job executing the code and obtain results for 1 process with 1, 2, 4, 8 and 16 threads (you can use the sample slurm script dotp job.cmd as a base for your job files). Submit a job executing the code and obtain results for 1, 2, 4, 8 and 16 processes with 1 thread each. Which case scales better? Plot the strong speedup for both cases.**

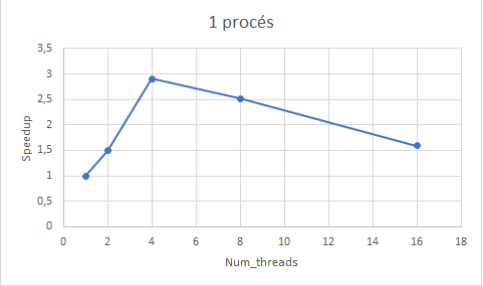
1 procés:

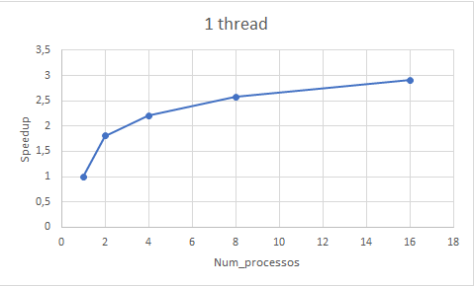
1 thread	2 threads	4 threads	8 threads	16 threads
6.5938e-02	4.4154e-02	2.2633e-02	2.6205e-02	4.1407e-02

1 thread:

1 procés	2 processos	4 processos	8 processos	16 processos
6.5938e-02	3.6527e-02	2.2985e-02	2.5584e-02	2.2264e-02

Dels anterior resultats veiem que en general triga menys tenir més threads que no tenir més processos, però que hi ha un moment que com més threads hi hagi més trigarà ja que en aquest punt ja no es òptim utilitzar més threads.
Per tant, podem dir que en el cas dels processos afegir-hi més només fa que s'obtingui un speedup lineals arribats a aquest punt en que ja no és tan útil utilitzar més recursos. Cosa que no es aixi en el cas d'afegir-hi més threads com s'ha explicat.
Els següents gràfics corresponen als "strong speedup":





- **Submit jobs executing the code and obtain results for various combination of processes and threads: 2-12, 4-6, 6-4, 12-2. Which is the best case?**

  Els resultats obtinguts de les diverses combinacions es mostren en la següent taula:

2-12	4-6	6-4	12-2
1.6297e-02	1.6528e-02	2.4475e-02	2.0601e-02

  Per tant, el millor cas ha estat amb 2 processos i 12 threads, tot i que molt a prop del resultat amb 4 processos i 6 threads. Per tant, dels resultats obtinguts, podem veure que són millors les combinacions on tenim més threads que processos.

Rubén Vera - 241456
Sara Soriano - 240007

Exercise 3.

- **Implement an MPI version of the matrix vector product.**

```
void main(int argc, char* argv[]){
  int size, rank;
  int p_size;
  double *A, *b;
  double *final_v;
  double total;
  total = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  A = par_read("/shared/Labs/Lab_2/matrix.bin", &p_size, rank, size);
  b = par_read("/shared/Labs/Lab_2/matrix_vector.bin", &p_size, rank, size);
  double *vector;
  vector = (double*) calloc(p_size*size,sizeof(double));
  final_v = (double *) calloc(p_size*size,sizeof(double));
  MPI_Allgather(&b[0], p_size, MPI_DOUBLE, vector, p_size, MPI_DOUBLE, MPI_COMM_WORLD);
  double start = omp_get_wtime();
  #pragma omp parallel for simd reduction(+:total)
  for(int i = 0; i < p_size; i++){
    for(int j = 0; j < p_size*size; j++){
      total+=A[ind(i,j,p_size*size)]*vector[j];
    }
    final_v[i] = total;
    total = 0;
  }
  double end = omp_get_wtime();
  double time = end - start;
  if(size == 1){
    printf("%.4e\t%f\t%f\t%f\t%f\n", time, final_v[0], final_v[2048], final_v[4096], final_v[6144]);
  }
  else{
   printf("%.4e\tc[%d]: %f\n", time, rank, final_v[0]);
  }
  MPI_Finalize();
}
```

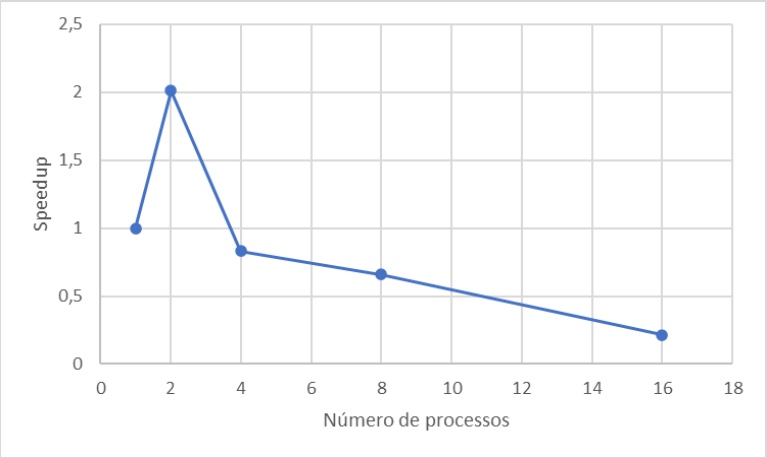
A partir de la funció creada en l'exercici anterior, hem definit la matriu A i el vector b que són les dues variables que s'utilitzaran per fer el producte matriu vector. Com quan hi hagi més d'un procés corrent cadascú tindrà una part de b, el que hem fet ha sigut un MPI_Allgather() amb els paràmetres corresponents, per tal de que tots tinguin el vector complet de b. A continuació, hem realitzat dos bucles per fer possible aquesta multiplicació.
Per veure com queda el resultat de final_v, que correspon a la c del pdf, hem fet dos tipus de prints depèn de si només hi ha un procés o més d'un. El resultat de l'execució tal i com se'ns donava el *job.cmd* és:

```
6.4063e-02      -3.271111       -3.692022       -11.577431      8.032121
4.9061e-02      c[3]: 8.032121
5.7117e-02      c[1]: -3.692022
5.7266e-02      c[2]: -11.577431
7.7072e-02      c[0]: -3.271111
```

- **Submit a job executing the code and obtain results for 1, 2, 4, 8 and 16 processes (you can use the sample slurm script dotp job.cmd as a base for your job files). Plot the strong speedup.**

1	2	4	8	16
6.4063e-02	3.1846e-02	7.702e-02	9.7405e-02	2.9663e-01

Rubén Vera - 241456
Sara Soriano - 240007



Com es pot veure al gràfic, a partir de 2 processos ja comença a ser poc útil afegir-hi més, inclús és contraproduent ja que el speedup està inclús per sota d'1. Per la qual cosa el temps és major en fer-ho amb 16 processos que en fer-ho amb 1 tal i com es veu a la taula creada.

Rubén Vera - 241456
Sara Soriano - 240007

Exercise 4.

- **Write a program matrix.c that performs the following steps. Each process fills a NxN matrix (where N is the total number of processes) with 0s, except for the diagonal elements that are initialized with the process' rank identifier.**

```
int *new_matrix(int size, int rank){
  int *A;
  A  = (int*)malloc(size*size*sizeof(int));
  for(int i = 0; i < size; i++){
    for(int j = 0; j < size; j++){
      if(i == j){
        A[ind(i,j,size)] = rank;
      }
      else{
        A[ind(i,j,size)] = 0;
      }
    }
  }
  return A;
}
```

Com diu l'enunciat creem una matriu de mida size*size per la qual cosa guardem l'espai de memoria necessari per tal que això es pugui realitzar correctament. Seguidament, fem dos bucles i diem que si i = j, és a dir, si ens trobem en un valor de la diagonal, volem que es posi el rang del procés en execució. En el cas que no ens trobem a la diagonal el valor serà 0.

- **Next, each process sends to process 0 an array with all elements of its diagonal using a collective routine.**

```
void print_matrix(int *matrix, int size){
  for(int i = 0; i< size; i++){
    for(int j = 0; j < size; j++){
      printf("%d ", matrix[ind(i,j,size)]);
    }
    printf("\n");
  }
}
```

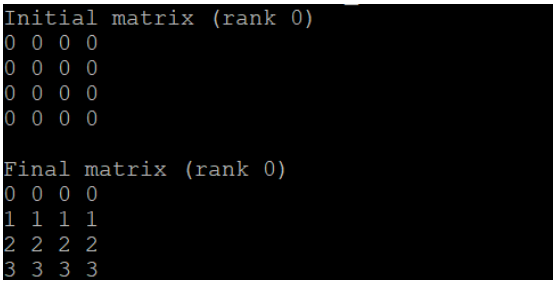
```
void main(int argc, char* argv[]){
  int size, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int *matrix, *final_matrix;
  matrix = (int*)malloc(size*size*sizeof(int));
  matrix = new_matrix(size, rank);

  if(rank == 0){
    final_matrix = (int*)malloc(size*size*sizeof(int));
    printf("Initial matrix (rank 0)\n");
    print_matrix(matrix, size);
  }
  else{
    final_matrix = NULL;
  }

  MPI_Datatype diagonaltype;
  MPI_Type_vector(size, 1, size + 1, MPI_INT, &diagonaltype);
  MPI_Type_commit(&diagonaltype);

  MPI_Gather(&matrix[ind(0, 0, size)], 1, diagonaltype, final_matrix, size, MPI_INT, 0, MPI_COMM_WORLD);

  if(rank == 0){
    printf("\nFinal matrix (rank 0)\n");
    print_matrix(final_matrix, size);
  }
  MPI_Type_free(&diagonaltype);
  MPI_Finalize();
}
```

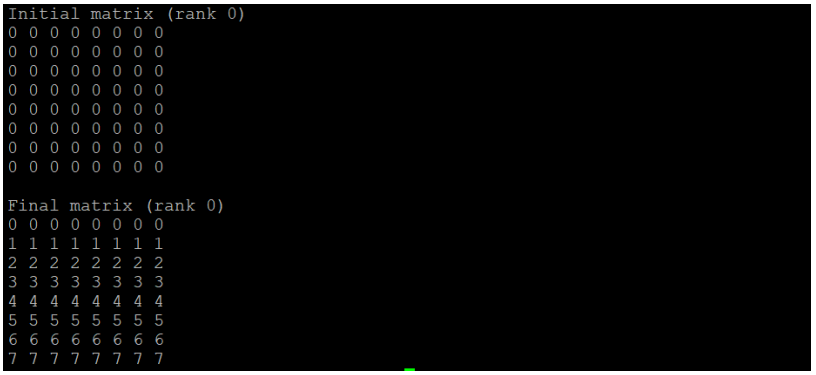
En la primera captura d'aquest apartat es mostra com hem implementat la funció print_matrix(), on només hem hagut de realitzar un doble bucle i

imprimir el resultat del valor corresponent de la matriu en aquella posició. Acabada la fila, s'imprimeix un '\n' per veure visualment on es canvi de fila.

En la segona captura es mostra el main(). En ell hem inicialitzat una regió MPI i hem obtinguts les variables rank i size, com ja hem explicat en exercicis anteriors. A continuació, hem inicialitzat dues matrius on una correspondrà a la matriu de cada procés original i l'altre (final_matrix) serà la matriu resultat que s'imprimirà. Per tant, aquesta matriu només la tindrà el procés 0, perquè és qui volem que imprimeixi.
Un cop fet això, creem un nou MPI_Datatype que agafa tots els valors de la diagonal d'una matriu. Per fer-ho definim diagonaltype com un vector amb una count = size ja que en una matriu hi ha "size" valors diagonals, en aquest cas 4. Blocklen = 1 ja que només agafarem un valor a partir d'on li diguem. El stride serà (size + 1) ja que des de que agafem un valor fins que ens trobem el següent, és a dir, des d'una diagonal fins la següent haurà passat (size + 1) posicions. Després posem que els tipus de valors seran MPI_INT i finalment, diem que això serà el diagonaltype.
Un cop creat el "data type", haurem de fer el seu "commit" i fem un Gather() per a que el procés 0 tingui tots els vectors amb els valors de les diagonals i els col·loqui segons li vinguin, és a dir, a cada fila col·locarà el vector que li vingui del procés corresponent.
Finalment, es mostra per pantalla la matriu resultant a partir de cridar a la funció print_matrix(). I s'allibera l'espai de memòria utilitzat per dur a terme el diagonaltype.

- **Try it with 4 and 8 processes.**





A la primera imatge podem veure com la matriu inicial del rang 0 es l'esperada és a dir tot ple de 0's ja que el rang és 0 i la matriu final es la matriu on les files estan plenes del rang de cada procés. Tot això s'ha realitzat mitjançant l'esmentat en l'apartat anterior.
En la segona imatge, podem veure com torna a succeir el mateix, la matriu inicial es l'esperada i la final també, on es pot veure en cada fila el número del rang de cada procés. En aquest cas, com hem utilitzat 8 processos anem des del 0 fins el 7.

Rubén Vera - 241456
Sara Soriano - 240007

Exercise 5.

- **Modify your binary file reading function so it reads integers and add it to game of life.c. Variable local entries has to store the matrix entries in each process.**

  Per dur a terme aquesta part, només hem hagut de modificar en la funció on posava doubles per ints, ja que ara el que es voldrà llegir no són dobles sino enters. De manera que el codi ens ha quedat tal i com es mostra en la següent imatge:

```
int *par_read(char *in_file, int *p_size, int rank , int nprocs){
  MPI_File fh;
  MPI_Offset filesize;
  MPI_Status status;
  int *par_read;

  MPI_File_open(MPI_COMM_WORLD, in_file, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
  MPI_File_get_size(fh, &filesize);

  filesize = filesize / sizeof(int);
  (*p_size) = filesize/nprocs;
  par_read = (int *) malloc(*p_size*sizeof(int));

  MPI_File_set_view(fh, rank*(*p_size)*sizeof(int), MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
  MPI_File_read(fh, par_read, *p_size, MPI_INT, &status);

  MPI_File_close(&fh);
  return par_read;
}
```

```
/* Ex 5.1 Read the binary file */
int *local_matrix = par_read(FILENAME, &local_entries, rank, nprocs);
```

  A més a més, hem vist que se'ns donava una funció que es diu parallel_read que fa la mateixa funció que la nostra. Tot i així, hem decidit utilitzar la nostra perquè se'ns especificava en aquest apartat.

- **Define the neighbours of each process. Each process has 2 neighbours, next and prev.**

  Per tal de definir els "neighbours" hem hagut de fer un estudi per casos. On el next i el prev serà els processos anteriors i posteriors, respectivament, sempre i quan no estiguem ni en el primer procés ni en el últim. Si ens trobem en el primer procés aleshores el prev serà l'últim i el next el segon. De manera semblant, l'últim procés tindrà com a prev al penúltim procés i com a next al primer.

```
else {
/* Ex 5.2: Define neighbours */
  if(rank != nprocs-1 && rank != 0){
    next = rank+1;
    prev = rank-1;
  }
  else if(rank == 0){
    next = rank+1;
    prev = nprocs-1;
  }
  else if(rank == nprocs-1){
    next = 0;
    prev = rank-1;
  }

}
```

- **Make rank 0 print a bitmap with the initial configuration. Rank 0 has to gather other processes local matrix.**

```
/* Ex 5.3: Make rank 0 print the problem */

int *full_matrix = calloc(total_entries, sizeof(int));
MPI_Gather(local_matrix, local_entries, MPI_INT, full_matrix, local_entries, MPI_INT, 0, MPI_COMM_WORLD);
if(rank == 0){
  bitmap(full_matrix, total_entries, row_size, nprocs, 0);
}
```

Per tal de fer que s'imprimeixi el bitmap, primerament hem creat un espai de memòria per la full_matrix de tamany total_entries. És a dir, per les entrades locals de cada procés que després de fer el Gather() quedarà de tamany total_entries. Seguidament, fem el MPI_Gather per tal que tots els processos tinguin la full_matrix. Finalment, fem que només el procés 0 utilitzi la funció bitmap per tal d'aconseguir el bitmap amb la configuració inicial.

- **Communicate rows to neighbours. upper send row has to be sent to prev and lower send to next. Complementary cases have to be received.**

```
else {
  /* Ex 5.4: Communicate upper and lower rows to the neighbouring processes */
    MPI_Send(lower_send, row_size, MPI_INT, next, 0, MPI_COMM_WORLD);
    MPI_Send(upper_send, row_size, MPI_INT, prev, 0, MPI_COMM_WORLD);
    MPI_Recv(lower_recv, row_size, MPI_INT, next, 0, MPI_COMM_WORLD, &stat);
    MPI_Recv(upper_recv, row_size, MPI_INT, prev, 0, MPI_COMM_WORLD, &stat);
}
```

Per tal de comunicar les files amb els veïns hem utilitzat les funcions MPI_Send() i MPI_Recv() com es pot veure en la imatge per tal d'establir aquesta comunicació. S'ha realitzat de tal manera que al "lower" s'hi envia el next i se'n rep el prev, mentre que al "upper" s'hi envia el prev i se'n rep el next.

- **Make rank 0 print a bitmap of the final configuration.**

```
/* Ex 5.5: Output the final state */
MPI_Gather(local_matrix, local_entries, MPI_INT, full_matrix, local_entries, MPI_INT, 0, MPI_COMM_WORLD);
if(rank == 0){
  bitmap(full_matrix, total_entries, row_size, nprocs, iter);
}
```

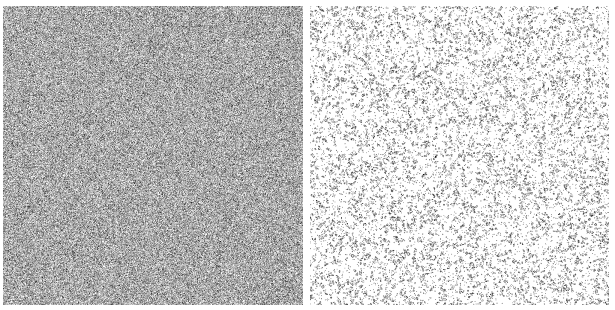
Per tal de realitzar aquest apartat hem seguit el mateix procediment que en el tercer apartat d'aquest exercici. És a dir, primer hem realitzat un Gather() per tal de que el procés 0 tingui tota la informació necessària. Per finalitzar si es el cas del procés 0, hem cridat a la bitmap().

- **Provide the initial and final bitmaps for life1.bin and life2.bin with 1 and 8 processes.**

  En les següents imatges podem veure primerament el bitmap inicial per life1.bin amb 1 procés i al costat, el bitmap final per life1.bin amb 1 procés:

  En les següents imatges podem veure primerament el bitmap inicial per life1.bin amb 8 processos i al costat, el bitmap final per life1.bin amb 8 processos:

  En les següents imatges podem veure primerament el bitmap inicial per life2.bin amb 1 procés i al costat, el bitmap final per life2.bin amb 1 procés:



En les següents imatges podem veure primerament el bitmap inicial per life2.bin amb 8 processos i al costat, el bitmap final per life2.bin amb 8 processos: