Report on MPI – Alexandre Temperville

Introduction

I will present the files I use like this, the first line of the array corresponding to the name of the concerning file and the second one to its contents, in occurrence the program.

Name of the file
Program

I will explain how my programs work, maybe I won't detail so much some things for not being too pompous with simple things. Moreover, I let some comments in my programs, which are complements of my explanations on this report, and I think they make my programs more readable, even if I could erase them for this report. I particularly put attention with what we have to put in the primitives we have discovered in these works.

I take the freedom to let some displays at the screen which are not fundamentally useful (except to control my results and if I don't have any errors) and the calculation of the execution time (not always asked, but interesting).

For some exercises, I will propose many programs, not necessarily asked, the last one being more readable and optimized, because I think this is interesting to perform these programs and they can be used a day or an other.

When I compile and execute on my computer, I have to name the executables of the form *./exe* otherwise there is an error in the execution, although in the room 106 I can just name them *exe* and it works without problem.

As one goes along, I avoid to say the same things we could have seen before, in particular concerning the displays at the screen.

I use some colors in my programs (the same like in 'gedit') for more visibility, sometimes I do not use the same colors but I use the colors like this :

blue : comments
pink : strings
green : types of variables
maroon : loops, conditions and return
purple : everything after a #

Remark about the display of the comments with *printf* :
As the processes execute the program together, sometimes the processes displays their messages at the same time and we can see that the messages are melted on the screen. As the display is not very necessary, we could remove the lines beginning by *printf*, I let them for more visibility as I said before.

A word about MPI

MPI (Message Passing Interface) is a *SPMD* (Single Program Multiple Data).

MPI_ANY_SOURCE : any process.

MPI_COMM_WORLD : virtual machine containing all the processes we want to use.

MPI_Init(NULL, NULL) : looks in a file MPI.hosts to see what machines we use and how.

MPI_Wtime() : gives the time at the moment.

MPI_Comm_size(MPI_COMM_WORLD, &P) : gives the number of processes in the virtual machine *MPI_COMM_WORLD* in the variable *P.*

MPI_Comm_rank(MPI_COMM_WORLD, &rank) : gives the rank of the process in course in the virtual machine *MPI_COMM_WORLD* in the variable *rank.*

An important remark :

In the room 106, we can see that the programs I will use with *MPI_Scatter*, *MPI_Gather* should be faster than in my PC, because they are very good dealing with processes on a lot of computers, contrary to my PC which works alone.

1st sheet : Parallel application programs MPI

1) MPICH2 environment configuration

In order to compile in *MPI*, we have first to configure our environment. Thus, we have just to follow the steps described in the 1st sheet. I can show what I put in the file *mpd.hosts* according the PC I am working.

On the computers in room 106 in Lille 1 :

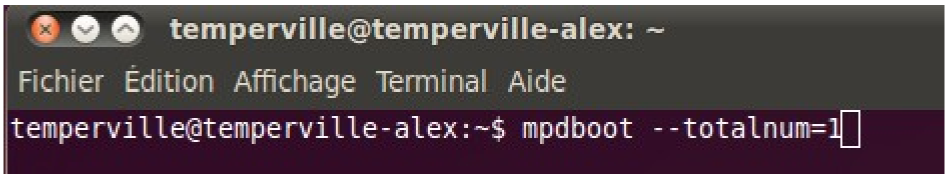
mpd.hosts
mpd.hosts mathcalc11 mathcalc10 mathcalc01 mathcalc03 mathcalc04 mathcalc02

On my own computer :

mpd.hosts
mpd.hosts temperville-alex

In order to use the library *MPI*, we need previously to launch *mpd* on each machine we use.

In my own computer, I have to enter in the terminal the following command :


temperville@temperville-alex: ~
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~$ mpdboot --totalnum=1
```

In room 106, I can enter *mpdboot --totalnum=6* at the maximum as I put 6 machines in the file *mpd.hosts*.

2) Programming configuration

Here is my file "*hosts*" :

hosts
hosts mathcalc11 mathcalc10:3 mathcalc01:2

3) Compilation and execution

Question 1 :

inconnu.c

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

int main()
{
  int source, cible, nb, recu;
  double debut, fin;
  MPI_Status status;
  MPI_Request request;

  MPI_Init(NULL,NULL);
  debut = MPI_Wtime();
  MPI_Comm_rank(MPI_COMM_WORLD, &source);
  MPI_Comm_size(MPI_COMM_WORLD, &nb);
  cible = (source + 1) % nb;
  MPI_Isend(&source, 1, MPI_INT, cible, 1, MPI_COMM_WORLD, &request);
  MPI_Recv(&recu, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status);
  fin = MPI_Wtime();
  printf("Le numéro %d a recu la valeur %d. Temps d'execution = %f \n", source, recu, fin-debut);
  MPI_Finalize();
  return 0;
}
```

Question 3 :

The program was executed 10 or 100 times more quickly for some processes, generally with those on the same computer, indeed there are more time communications between 2 computers than between 2 processes.

Question 4 :

The program "*inconnu*" does the following :
Each process sends its rank to the process of rank *'rank+1'* (except the process of rank *'nb-1'* which sends to the process *0*) and then displays what it received and with its execution time.

4) Communications one-to-all and all-to-one

1. One-to-all

So as to the process of rank *0* sends the today date to all the others processes, I use a header file, which contains some constant arrays defining the days and the months. I could put it in the file *one_to_all.c* but as I wanted to give the exact today date, I wanted to use a structure existing in the library <time.h> so I present it in this header.

Explanations for getting the today date :

To give the date, we have to use some primitives, in particular we need a '*timestamp*' which allow us to represent time data. Precisely, the *timestamp* is the number of seconds passed since January, the 1st of 1970.

The function *time* allows us to retrieve this *timestamp*. Indeed, this function returns a variable of type *time_t* (this type being defined as in the library <time.h>).

Knowing the timestamp, we can use conversions to give any type of time data, these conversions are given thanks to the function *localtime* and the structure *tm*, implemented in the library <time.h>.

Thanks to this and with the procedure *sprintf*, the process of rank 0 has created the date as a string. We will send this date to each process thanks to the primitives *MPI_Isend* and *MPI_Recv*. Let explain the parameters of these primitives :

Sending :

As we are in the process of rank *0* (because of the *if (rank == 0)*) :

MPI_Isend(date, 100, MPI_CHAR, target, 1, MPI_COMM_WORLD, &request);

means that the process which can do this instruction (so only the process of rank *0* here) sends the string *date* containing *100* characters (type *MPI_CHAR*) to the process of rank *target* and this sending has a tag (here *1*, we explain this in the receipt), *MPI_COM_WORLD* is a data which has informations about all the processes running for our program and relies them with a virtual machine, lastly, the *request* parameter is just a parameter of error, in case there is a problem in the sending.

Reception :

MPI_Recv(date_received, 100, MPI_CHAR, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status);

means that the processes which can do this instruction (so every process here) store the string (type *MPI_CHAR*) of *100* characters coming from any process (*MPI_ANY_SOURCE*) having a tag of *1* in a string called *date_received*. The receipt is identified by the corresponding sending of same tag. The parameter *status* is a parameter of errors.

Display :

For each process, I display at the screen what every process has received and its corresponding execution time.

one_to_all.c

```
# include <mpi.h>
# include "date.h"

int main()
{
  int rank, target, nb;
  double begin, end;
  char date[100], date_received[100];
  time_t timestamp;
  struct tm *t;
  MPI_Status status;
  MPI_Request request;

  MPI_Init(NULL,NULL);
```

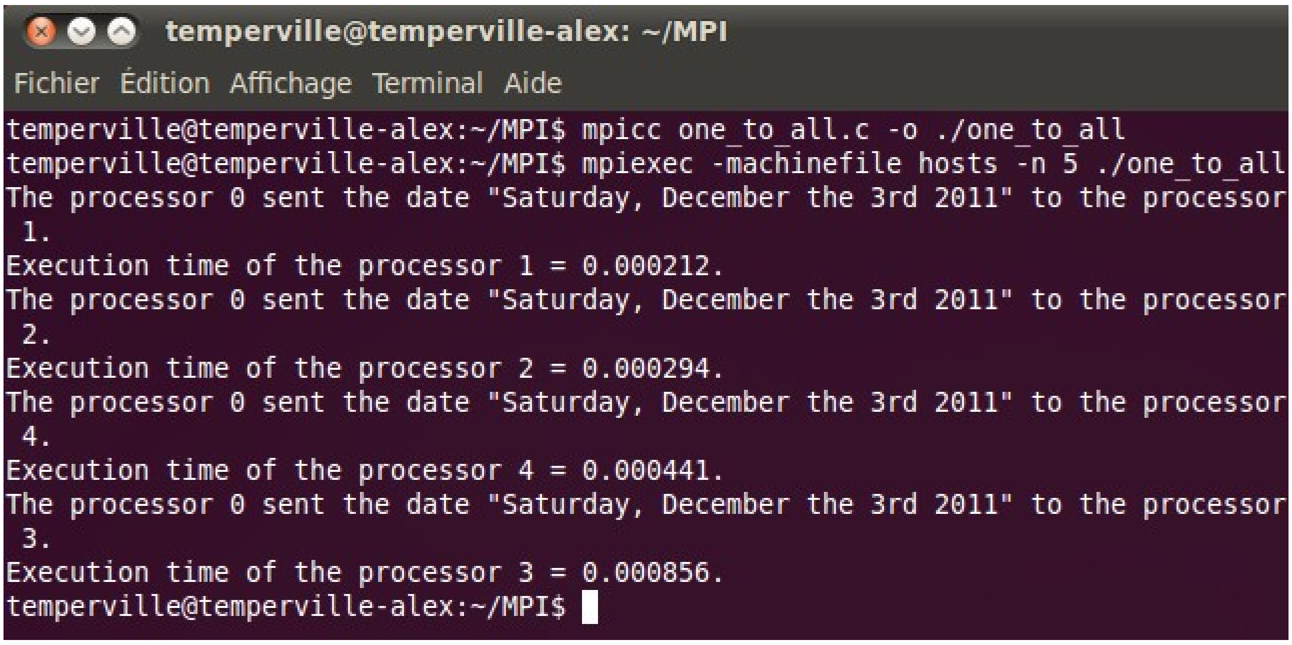
```
 begin = MPI_Wtime();

 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &nb);

/* ======================== Processus of rank 0 ============================ */

 if (rank == 0)
   {
     timestamp = time(NULL);
     t = localtime(&timestamp);

/* According to the date, we define the date differently (because of the suffix for the number of
day).
   The function "sprintf" add the string in second argument into the first one, like a "printf" but for
printing in the variable 'date', and not at the screen. */
 if (t->tm_mday % 20 == 1)
    sprintf(date, "%s, %s the %d%s %d", day[t->tm_wday], month[t->tm_mon], t->tm_mday, "st",
1900 + t->tm_year);
 if (t->tm_mday == 31)
    sprintf(date, "%s, %s the %d%s %d", day[t->tm_wday], month[t->tm_mon], t->tm_mday, "st",
1900 + t->tm_year);
 if (t->tm_mday % 20 == 2)
    sprintf(date, "%s, %s the %d%s %d", day[t->tm_wday], month[t->tm_mon], t->tm_mday, "nd",
1900 + t->tm_year);
 if (t->tm_mday % 20 == 3)
    sprintf(date, "%s, %s the %d%s %d", day[t->tm_wday], month[t->tm_mon], t->tm_mday, "rd",
1900 + t->tm_year);
 else
    sprintf(date, "%s, %s the %d%s %d", day[t->tm_wday], month[t->tm_mon], t->tm_mday, "th",
1900 + t->tm_year);

// Sends the date to all the processes, except the one of rank 0 :
     for (target=1; target<nb; target++)
          MPI_Isend(date, 100, MPI_CHAR, target, 1, MPI_COMM_WORLD, &request);
   }

/* ================== All the processes of rank different from 0 ================== */

 else
   {
// Reception of the date :
     MPI_Recv(date_received, 100, MPI_CHAR, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, &status);

// Display of the date and execution time for each process different from 0 :
     printf("The processor 0 sent the date \"%s\" to the processor %d.\n", date_received, rank);
     end = MPI_Wtime();
     printf("Execution time of the processor %d = %f.\n", rank, end-begin);
   }

 MPI_Finalize();
```

```
  return 0;
}
```

date.h

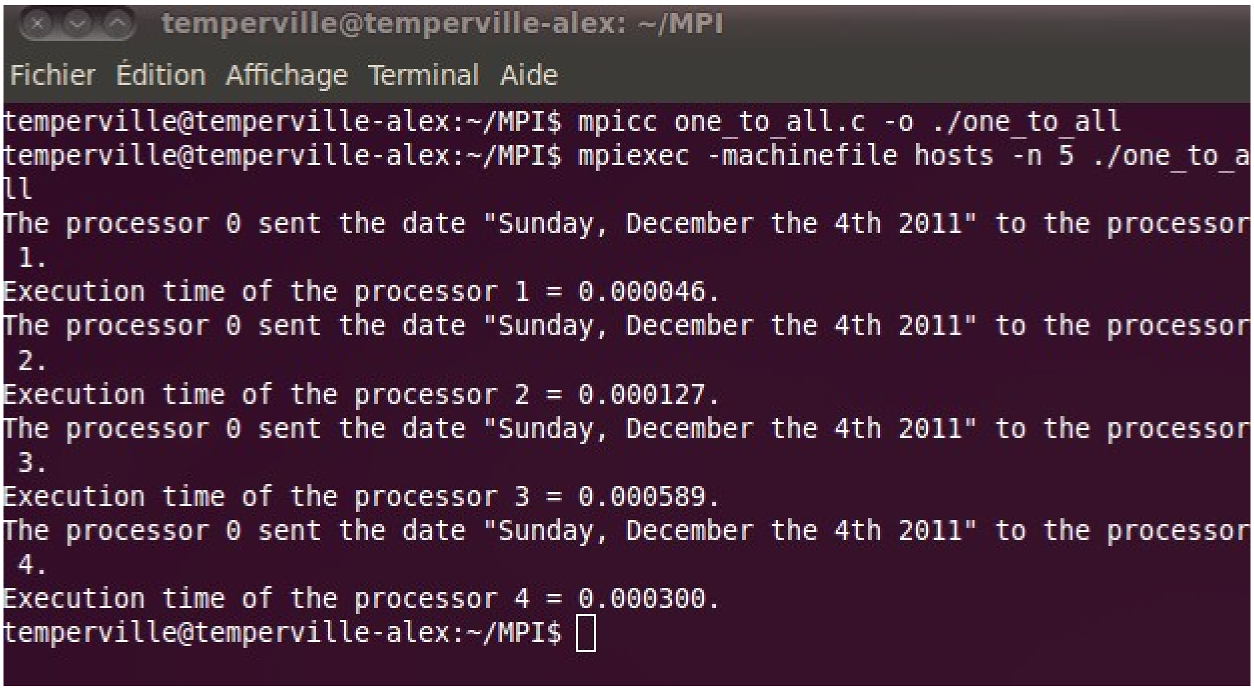
```
# ifndef DATE_H
# define DATE_H
# include <stdio.h>
# include <time.h>
# include <stdlib.h>
# include <string.h>

// I present the structure of tm, which gives time data, used in the program 'one_to_all.c'.

/* ======================= Structure of tm ======================= ========

{  int tm_sec;        seconds (0,59)
   int tm_min;        minutes (0,59)
   int tm_hour;       hours since midnight (0,23)
   int tm_mday;        day of the month (0,31)
   int tm_mon;        month since January (0,11)
   int tm_year;       years happened since 1900
   int tm_wday;       day since Sunday (0,6)
   int tm_tm_yday;    day since the 1st January (0,365)
   int tm_isdst;      jet lag };  */

/* These strings will be used in order to give the date in letters. */
const char * day[] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"};

const char * month[] = {"January", "February", "Marth", "April", "May", "June", "July", "August",
"September", "October", "November", "December"};

# endif /* DATE_H */
```

To execute this program, we can see what I enter in my computer with the following picture, where I use 5 processes defined in my file hosts :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc one_to_all.c -o ./one_to_all
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 5 ./one_to_all
The processor 0 sent the date "Saturday, December the 3rd 2011" to the processor
 1.
Execution time of the processor 1 = 0.000212.
The processor 0 sent the date "Saturday, December the 3rd 2011" to the processor
 2.
Execution time of the processor 2 = 0.000294.
The processor 0 sent the date "Saturday, December the 3rd 2011" to the processor
 4.
Execution time of the processor 4 = 0.000441.
The processor 0 sent the date "Saturday, December the 3rd 2011" to the processor
 3.
Execution time of the processor 3 = 0.000856.
temperville@temperville-alex:~/MPI$ 
```

The next day, when we do the same, we have this :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc one_to_all.c -o ./one_to_all
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 5 ./one_to_a
ll
The processor 0 sent the date "Sunday, December the 4th 2011" to the processor
 1.
Execution time of the processor 1 = 0.000046.
The processor 0 sent the date "Sunday, December the 4th 2011" to the processor
 2.
Execution time of the processor 2 = 0.000127.
The processor 0 sent the date "Sunday, December the 4th 2011" to the processor
 3.
Execution time of the processor 3 = 0.000589.
The processor 0 sent the date "Sunday, December the 4th 2011" to the processor
 4.
Execution time of the processor 4 = 0.000300.
temperville@temperville-alex:~/MPI$ 
```

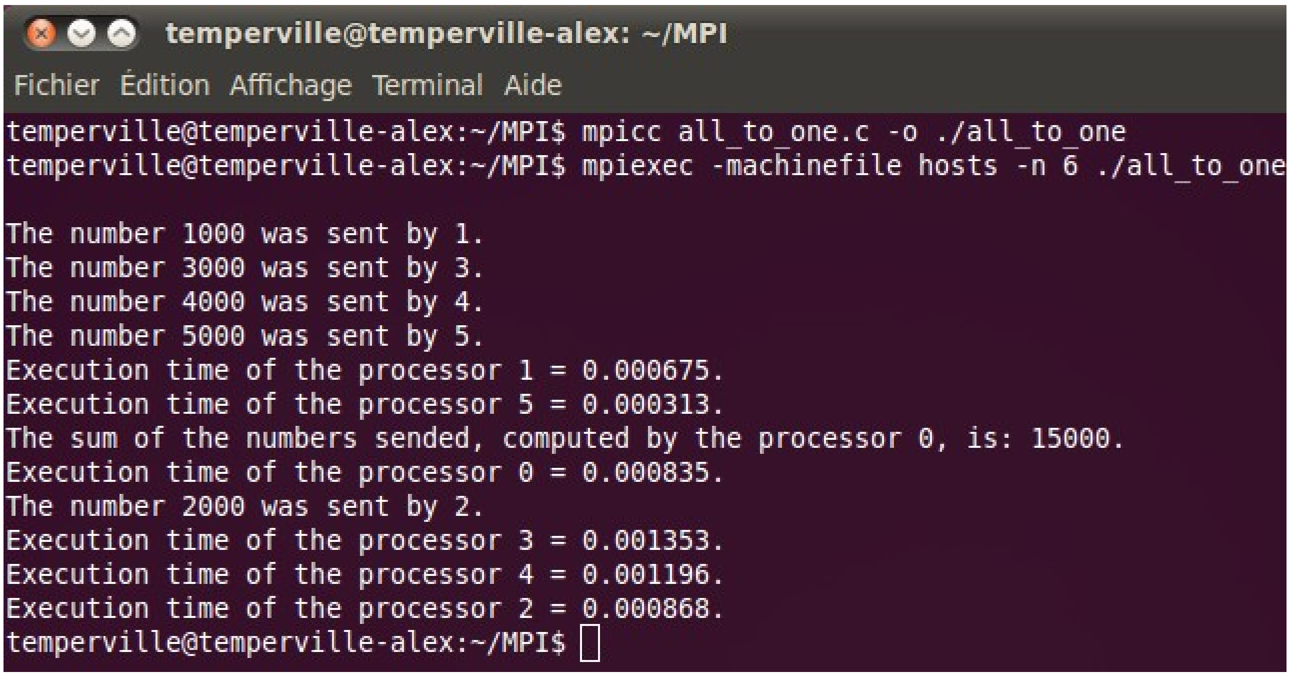
2. All-to-one

In this program, each process of rank different from *0* computes a number *M*, sends this number to the process of rank *0*, which computes the sum of these numbers. Then, we display it.

all_to_one.c

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

int main()
{
 int i, rank, M, nb, M_received, sum;
 double begin, end;

 MPI_Status status;
 MPI_Request request;

 MPI_Init(NULL,NULL);
 begin = MPI_Wtime();

 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &nb);


/* ================= All the processes of rank different from 0 =================== */

 if (rank != 0)
   {
// Sends the number M to the process 0 :
    M = 1000*rank;
    MPI_Isend(&M, 1, MPI_INT, 0, 12, MPI_COMM_WORLD, &request);
    printf("The number %d was sent by %d.\n", M, rank);
   }

/* ======================== Process of rank 0 ============================= */

 else
   {
// Receives the number M from each process of rank different of 0, and computes their sum :
    sum = 0;
    for (i=1; i<nb; i++)
    {
     MPI_Recv(&M_received, 1, MPI_INT, i, 12, MPI_COMM_WORLD, &status);
     sum = sum + M_received;
    }

// Displays the sum of the numbers sent :
    printf("The sum of the numbers sended, computed by the processor 0, is: %d.\n", sum);
   }

/* =========================== Every process ============================ */

// Displays the execution time of each process :
 end = MPI_Wtime();
 printf("Execution time of the processor %d = %f.\n", rank, end-begin);

 MPI_Finalize();
 return 0;
}
```

In the terminal, we have :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc all_to_one.c -o ./all_to_one
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 6 ./all_to_one

The number 1000 was sent by 1.
The number 3000 was sent by 3.
The number 4000 was sent by 4.
The number 5000 was sent by 5.
Execution time of the processor 1 = 0.000675.
Execution time of the processor 5 = 0.000313.
The sum of the numbers sended, computed by the processor 0, is: 15000.
Execution time of the processor 0 = 0.000835.
The number 2000 was sent by 2.
Execution time of the processor 3 = 0.001353.
Execution time of the processor 4 = 0.001196.
Execution time of the processor 2 = 0.000868.
temperville@temperville-alex:~/MPI$
```

5) Data packing

Now we want that each process sends its rank in number and in letters to the process of rank *0*. So we have to send two types of data :

- data of the type *MPI_INT*
- data of the type *MPI_CHAR*

We could send them one after one as we did with the primitives *MPI_Isend* twice or send these data just with one *MPI_Isend* thanks to the primitive *MPI_Pack*, used before the sending.

Creation of the pack to send :

To create a pack, we need to use the following primitive as much as is needful :

MPI_Pack(&rank, 1, MPI_INT, t, 1000, &position, MPI_COMM_WORLD);
MPI_Pack(&letters, 100, MPI_CHAR, t, 1000, &position, MPI_COMM_WORLD);

We put in a variable *t* which can contains *1000* elements the data of different types. We need a parameter of *position*, which precises where we put each element. First of all, before using *MPI_Pack*, we initialize this parameter to *0*. The primitive *MPI_Pack* increments this position each time it fills one element.

Here we put the rank at the position *0* in *t*, then the *letters* since the position *1*.

Sending of the pack :

Then we can send *t*, which is of the size *position* (the last position used when entering the letters), it is a data of type *MPI_PACKED*, and we send this pack to the process ***0*** with an arbitrarily tag (I choose *12*) :

*MPI_Isend(t, position, MPI_PACKED, **0**, 12, MPI_COMM_WORLD, &request);*

Reception of the pack :

Finally, we receive from each process the pack with the primitive *MPI_Recv*. But after, we have a pack to open. In order to open this pack, we need to use the primitive *MPI_UNPACK* , from the position *0* :

MPI_Unpack(t, 1000, &position, &nb_received, 1, MPI_INT, MPI_COMM_WORLD);
MPI_Unpack(t, 1000, &position, &letters_received, 100, MPI_CHAR, MPI_COMM_WORLD);

We unpack the variable *t* containing *1000* elements from the current *position.* Firstly the rank stored in *nb_received* of type *MPI_INT* containing just *1* element, then the letters stored in *letters_received* of type *MPI_CHAR* containing *100* elements.

Display of the sending data :

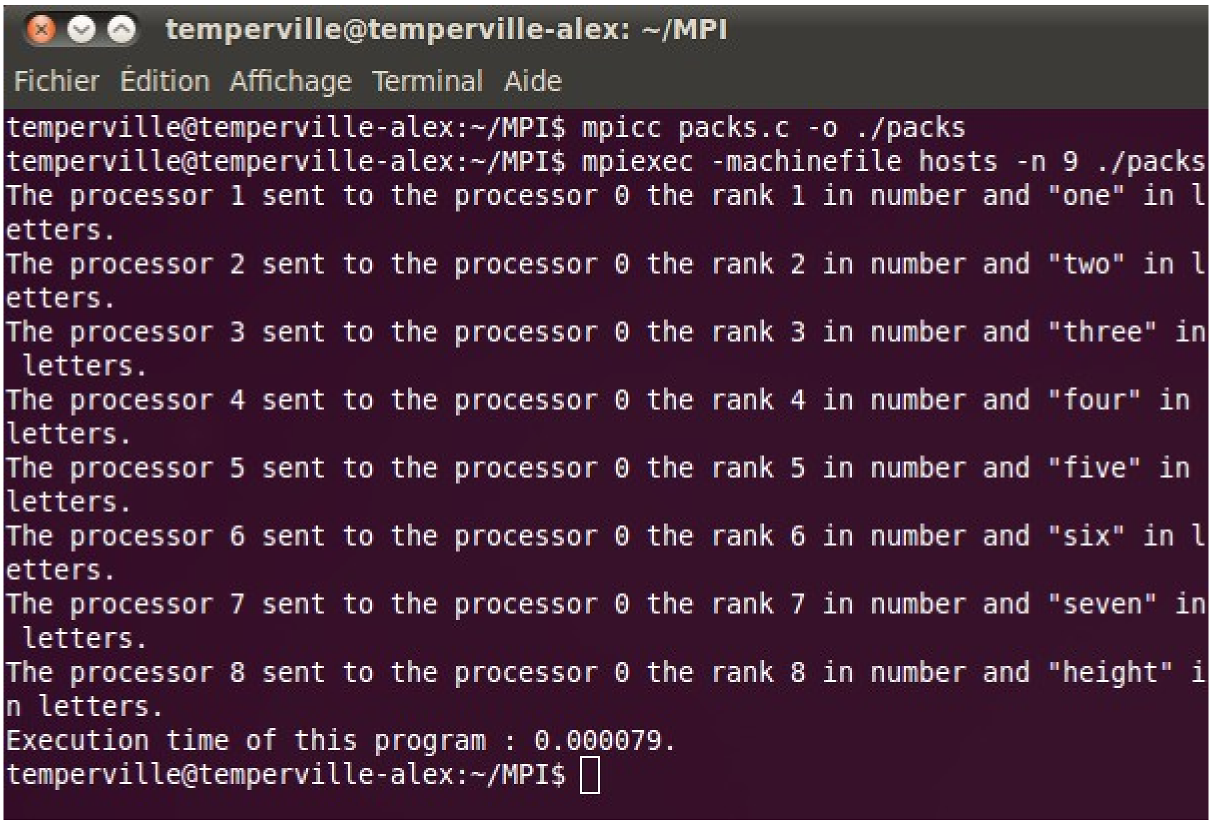
Now we have all our data and we can display by each process.

packs.c

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

int main()
{
 int i, rank, nb, nb_received, position;
  double begin, end;
  MPI_Status status;
  MPI_Request request;
  char t[1000];
  char letters[100];
  char letters_received[100];

  MPI_Init(NULL,NULL);
  begin = MPI_Wtime();

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nb);

/* ================= All the processes of rank different from 0 =================== */

  if (rank != 0)
    {
// According to the rank of the process, the rank in letters should be different :
      if (rank == 1)
        strcpy(letters, "\"one\"");
      if (rank == 2)
        strcpy(letters, "\"two\"");
      if (rank == 3)
        strcpy(letters, "\"three\"");
      if (rank == 4)
        strcpy(letters, "\"four\"");
      if (rank == 5)
```

```
        strcpy(letters, "\"five\"");
      if (rank == 6)
        strcpy(letters, "\"six\"");
      if (rank == 7)
        strcpy(letters, "\"seven\"");
      if (rank == 8)
        strcpy(letters, "\"height\"");
      if (rank == 9)
        strcpy(letters, "\"nine\"");
      if (rank == 10)
        strcpy(letters, "\"ten\"");
      if (rank == 11)
        strcpy(letters, "\"eleven\"");
      if (rank == 12)
        strcpy(letters, "\"twelve\"");
      if (rank == 13)
        strcpy(letters, "\"thirteen\"");
      if (rank == 14)
        strcpy(letters, "\"fourteen\"");
      if (rank == 15)
        strcpy(letters, "\"fifteen\"");
      if (rank > 15)
        strcpy(letters, "\"Sorry, I just know how to count until 15 !\"\n.");

// Creates a package to send containing different types of variable :
      position = 0;
      MPI_Pack(&rank, 1, MPI_INT, t, 1000, &position, MPI_COMM_WORLD);
      MPI_Pack(&letters, 100, MPI_CHAR, t, 1000, &position, MPI_COMM_WORLD);

// Sends data in the pack previously made to the process 0 :
      MPI_Isend(t, position, MPI_PACKED, 0, 12, MPI_COMM_WORLD, &request);
      end = MPI_Wtime();
    }

/* ========================= Process of rank 0 ============================ */

  else
    {
// Receives the packs, opens them and displays what they contain :
  for(i=1; i<nb; i++)
      {
        position = 0;
        MPI_Recv(t, 1000, MPI_PACKED, MPI_ANY_SOURCE, 12, MPI_COMM_WORLD, 
&status);
        MPI_Unpack(t, 1000, &position, &nb_received, 1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(t, 1000, &position, &letters_received, 100, MPI_CHAR, 
MPI_COMM_WORLD);
        printf("The processor %d sent to the processor 0 the rank %d in number and %s in letters.\n", 
nb_received, nb_received, letters_received);
      }

// Displays the execution time of the program :
    end = MPI_Wtime();
    printf("Execution time of this program : %f.\n", end-begin);
  }

 MPI_Finalize();

 return 0;
}
```

We can see now what this program does :



temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc packs.c -o ./packs
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 9 ./packs
The processor 1 sent to the processor 0 the rank 1 in number and "one" in l
etters.
The processor 2 sent to the processor 0 the rank 2 in number and "two" in l
etters.
The processor 3 sent to the processor 0 the rank 3 in number and "three" in
 letters.
The processor 4 sent to the processor 0 the rank 4 in number and "four" in
letters.
The processor 5 sent to the processor 0 the rank 5 in number and "five" in
letters.
The processor 6 sent to the processor 0 the rank 6 in number and "six" in l
etters.
The processor 7 sent to the processor 0 the rank 7 in number and "seven" in
 letters.
The processor 8 sent to the processor 0 the rank 8 in number and "height" i
n letters.
Execution time of this program : 0.000079.
temperville@temperville-alex:~/MPI$
```

6) <u>Ring communications</u>

In this exercise, we want to transmit a token between the processes successively.
To begin, the process of rank *0* send the token to the process *1*, and then the process is blocked because of the blocking primitive *MPI_Recv*.

Then the process of rank *1* which was blocked because of the primitive *MPI_Recv* can now receive the token from the process of rank *0*, send it to the process of rank *2*, display this token, and finish the program.

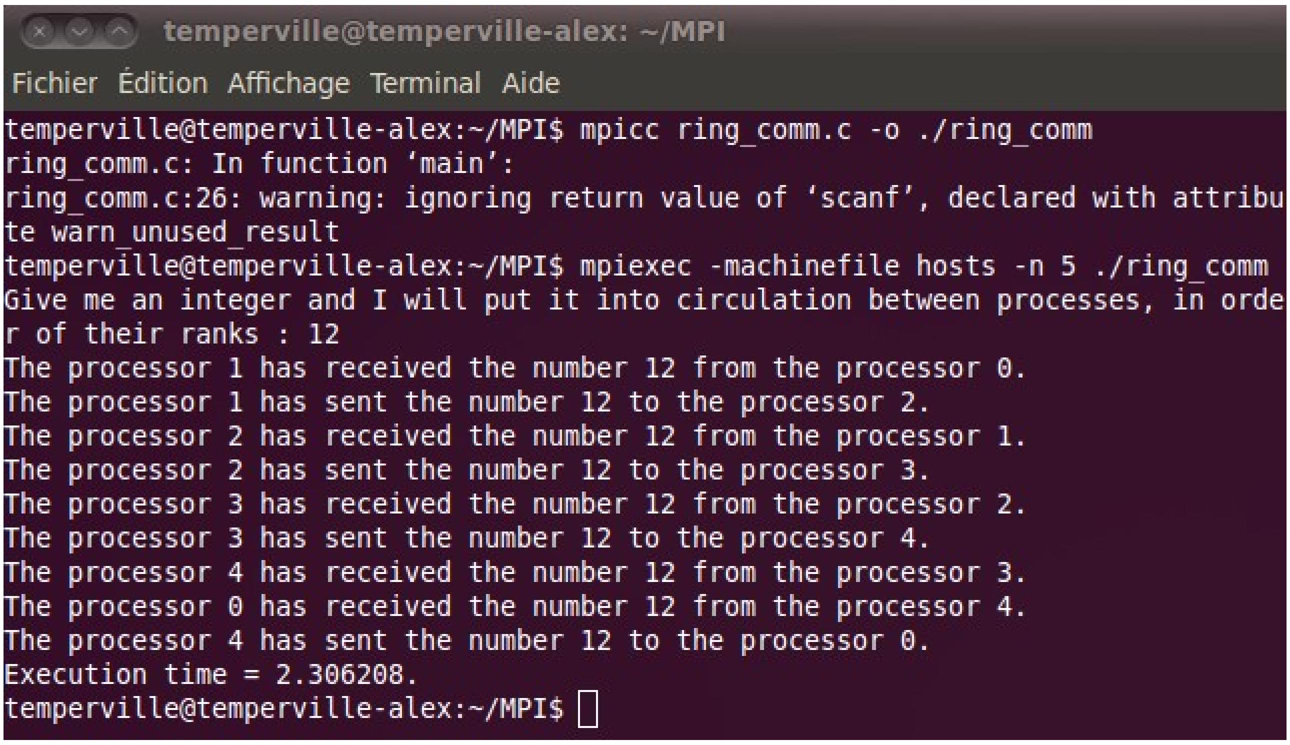
I try to change the tags in order to specify precisely with no doubts which process must receive the sending, the program also works with the same tag everywhere. It could be very useful if we have to send several data from one process, although we saw we could use *MPI_Pack*.

So, process after process, each one receives the token, send it to the process of following rank, display it and finish the program. The process of rank *0* is the last one to finish the program, because it is waiting for the reception from the process of rank *'nb-1'*.

ring_comm.c

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

int main()
{
  int rank, nb, k;
  double begin, end;
  MPI_Status status;
  MPI_Request request;

  MPI_Init(NULL,NULL);
  begin = MPI_Wtime();

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nb);

/* ========================= Process of rank 0 ============================ */

  if (rank == 0)
    {
// Defines the integer we want to put into circulation between processes :
      printf("Give me an integer and I will put it into circulation between processes, in order of their
ranks : ");
      scanf("%d", &k);

// Sends this number to the process '1' :
      MPI_Isend(&k, 1, MPI_INT, rank+1, rank, MPI_COMM_WORLD, &request);

// Receives this number from the process 'nb-1' :
      MPI_Recv(&k, 1, MPI_INT, nb-1, nb-1, MPI_COMM_WORLD, &status);

// Displays the number received and the execution time of the process '0' :
      printf("The processor %d has received the number %d from the processor %d.\n", rank, k, nb-
1);
      end = MPI_Wtime();
      printf("Execution time = %f.\n", end-begin);
    }

/* ============== All the processes of rank different from 0 and nb-1 =============== */

  if ((rank != 0) && (rank != nb-1))
    {
// Receives the number sent by the process 'rank-1' :
      MPI_Recv(&k, 1, MPI_INT, rank-1, rank-1, MPI_COMM_WORLD, &status);
      printf("The processor %d has received the number %d from the processor %d.\n", rank, k,
rank-1);

// Sends the number received to the process 'rank+1' :
      MPI_Isend(&k, 1, MPI_INT, rank+1, rank, MPI_COMM_WORLD, &request);
      printf("The processor %d has sent the number %d to the processor %d.\n", rank, k, rank+1);
    }

/* ======================= Process of rank nb-1 ========================== */

  if (rank == nb-1)
    {
// Receives the number sent by the process 'nb-2' :
      MPI_Recv(&k, 1, MPI_INT, rank-1, rank-1, MPI_COMM_WORLD, &status);
      printf("The processor %d has received the number %d from the processor %d.\n", rank, k,
rank-1);

// Sends the number received to the process '0' :
      MPI_Isend(&k, 1, MPI_INT, 0, rank, MPI_COMM_WORLD, &request);
      printf("The processor %d has sent the number %d to the processor 0.\n", rank, k);
    }

  MPI_Finalize();
  return 0;
}
```

In the terminal, we get this :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc ring_comm.c -o ./ring_comm
ring_comm.c: In function ‘main’:
ring_comm.c:26: warning: ignoring return value of ‘scanf’, declared with attribu
te warn_unused_result
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 5 ./ring_comm
Give me an integer and I will put it into circulation between processes, in orde
r of their ranks : 12
The processor 1 has received the number 12 from the processor 0.
The processor 1 has sent the number 12 to the processor 2.
The processor 2 has received the number 12 from the processor 1.
The processor 2 has sent the number 12 to the processor 3.
The processor 3 has received the number 12 from the processor 2.
The processor 3 has sent the number 12 to the processor 4.
The processor 4 has received the number 12 from the processor 3.
The processor 0 has received the number 12 from the processor 4.
The processor 4 has sent the number 12 to the processor 0.
Execution time = 2.306208.
temperville@temperville-alex:~/MPI$ 
```

We can notice than when we compile, we have an error with *scanf*. I don't have this error in the room 106 but just on my computer, and this error doesn't prevent me to run the program correctly, so this is not very important.

2nd sheet : Collective communications with MPI

1) MPI_Bcast primitive

Here the process of rank ***0*** has to send the same message to every process. Instead of using a loop with the primitive *MPI_Send* and *MPI_Recv*, we have a primitive which do this but just with one instruction, this is *MPI_Bcast* :

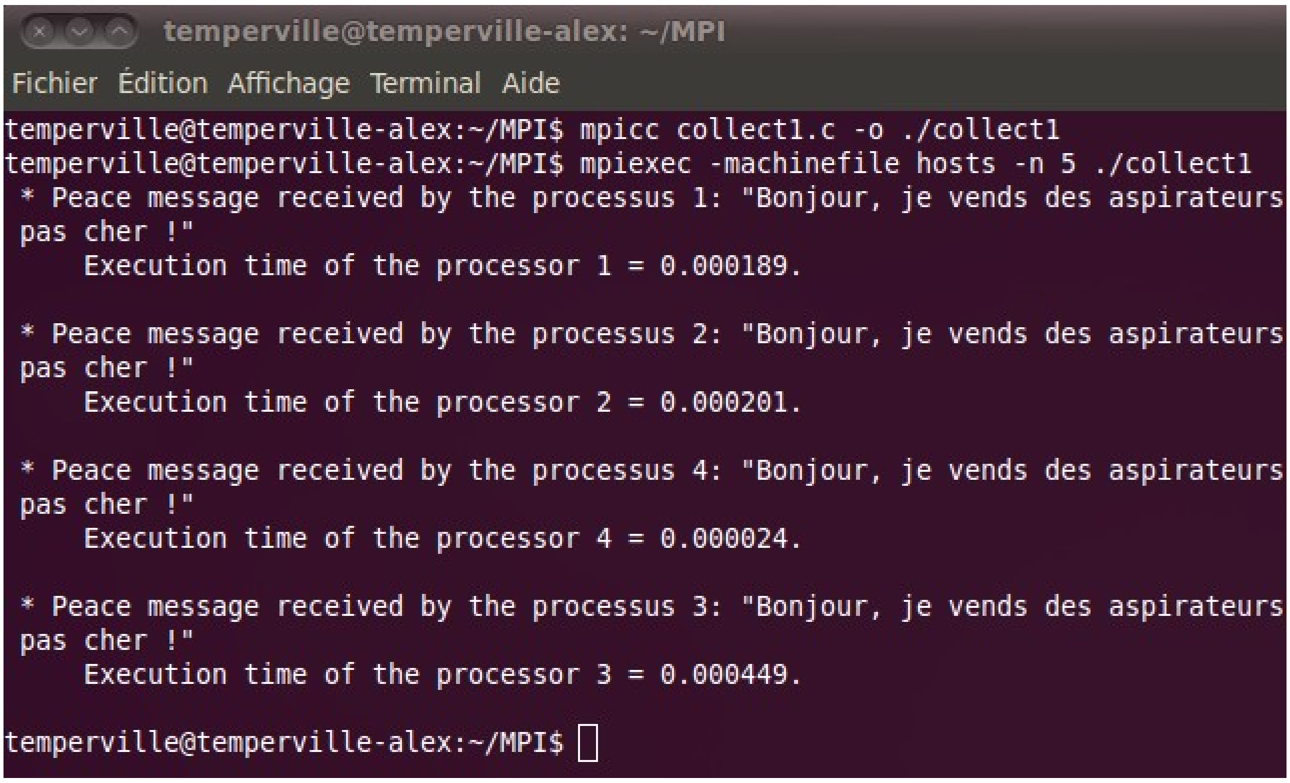
MPI_Bcast(message, 1000, MPI_CHAR, ***0****, MPI_COMM_WORLD);*

So here, the process of rank ***0*** sends the variable *message* of type *MPI_CHAR* and of size *1000* to every process included in the virtual machine *MPI_COMM_WORLD*.

collect1.c

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

int main()
{
  int rank;
  double begin, end;
  char message[1000];

  MPI_Init(NULL,NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  begin = MPI_Wtime();

// Creation of the message in the process 0 :
  if (rank == 0)
    strcpy(message, "Bonjour, je vends des aspirateurs pas cher !");

// Sends the message to all the processes, then they receive the message :
  MPI_Bcast(message, 1000, MPI_CHAR, 0, MPI_COMM_WORLD);

// Displays the execution time and the message received by all the processes :
  if (rank != 0)
   {
     end = MPI_Wtime();
     printf(" * Peace message received by the processus %d: \"%s\"\n     Execution time of the
processor %d = %f.\n\n", rank, message, rank, end-begin);
   }

  MPI_Finalize();
  return 0;
}
```

In the terminal, we have :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc collect1.c -o ./collect1
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 5 ./collect1
 * Peace message received by the processus 1: "Bonjour, je vends des aspirateurs
 pas cher !"
     Execution time of the processor 1 = 0.000189.

 * Peace message received by the processus 2: "Bonjour, je vends des aspirateurs
 pas cher !"
     Execution time of the processor 2 = 0.000201.

 * Peace message received by the processus 4: "Bonjour, je vends des aspirateurs
 pas cher !"
     Execution time of the processor 4 = 0.000024.

 * Peace message received by the processus 3: "Bonjour, je vends des aspirateurs
 pas cher !"
     Execution time of the processor 3 = 0.000449.

temperville@temperville-alex:~/MPI$ 
```

This primitive is faster than using *MPI_Isend* and *MPI_Recv* with loops, so using *MPI_Bcast* is a way to optimize programs.

Conclusion : ***MPI_Bcast*** **does a one_to_all sending.** (Bcast for Broadcast)

2) MPI_Scatter primitive

1. Version 1

Length of the sub-arrays sent :

If *N/P* is not an integer (with *P* number of processes), we define an integer *M* such that *M/P* is an integer, and we will create in the process of rank *0* an array of size *M* where there are *0* after the N^{th} element. A such array is created in the process of rank *0* with the function *rand()*.

The variable *l* = *N/P* is the length of the sub-arrays we will send.

Sending of a part of the array *tab* :

We now send a part of the array *tab*, beginning at a particular element : the element *tab[target*l]*. We take *l* elements from this one, there are of type *MPI_INT*, and we send them to process of rank ***target***, with an arbitrarily tag (here *2*). We do this in a loop with :

*MPI_Isend(&tab[target*l], l, MPI_INT,* ***target****, 2, MPI_COMM_WORLD, &request);*

Reception of the sub-array :

Then, each process receives the sub-array stored in the variable *sub_tab* of size *l*, coming from the process of rank ***0***, with the following instruction :

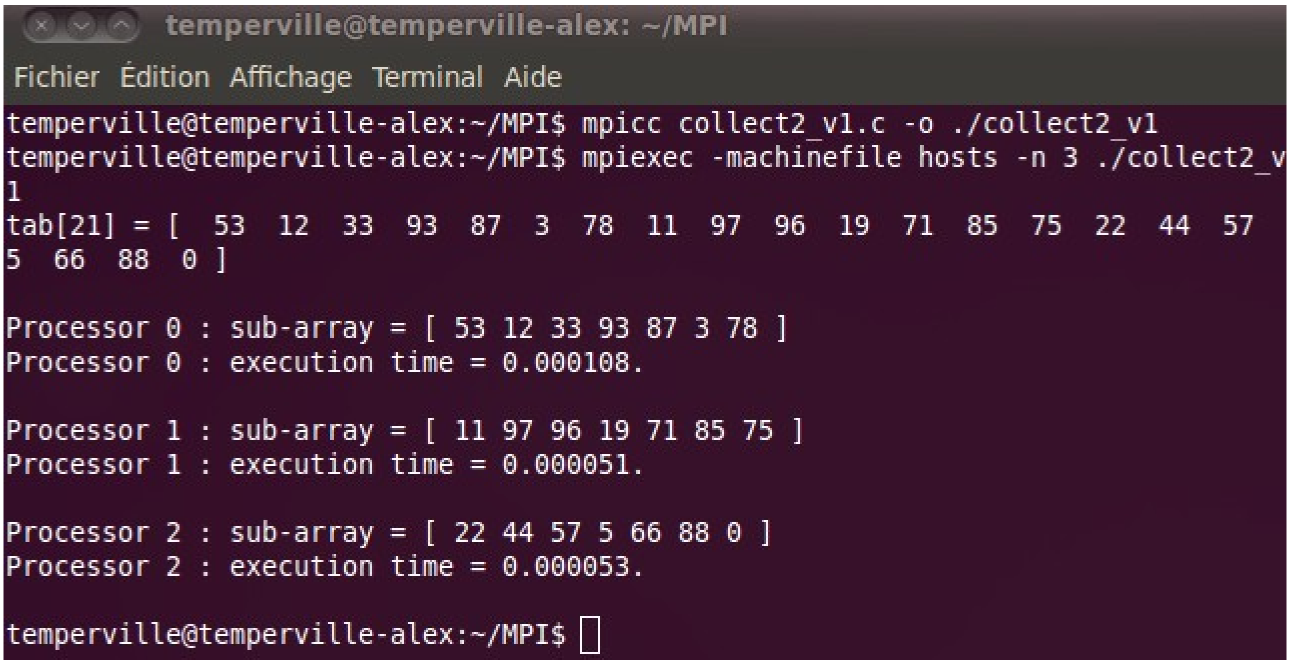
MPI_Recv(sub_tab, l, MPI_INT, ***0****, 2, MPI_COMM_WORLD, &status);*

collect2_v1.c

```c
# include <stdio.h>
# include <mpi.h>
# include <time.h>
# define N 20

int main()
{
  int M, i, l, target, nb, rank;
  double begin, end;
  int tab[30], sub_tab[30];
  MPI_Request request;
  MPI_Status status;

  MPI_Init(NULL,NULL);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nb);

  begin = MPI_Wtime();

// M = length of the considered array to send
// l = length of the sub-arrays
    M = N;
    while ( M % nb != 0)
      M++;
    l = M/nb;

/* ======================== processus of rank 0 ============================ */

if (rank == 0)
  {
    srand(time(NULL));

// Creation of the array tab :
    for (i=0; i<M; i++)
      {
        if (i<N)
          tab[i] = rand()%100;
        else
          tab[i] = 0;
      }

// Display of tab :
  printf("tab[%d] = [ ", M);
  for (i=0; i<M ; i++)
    printf(" %d ", tab[i]);
  printf("]\n\n");

// Sends a sub-array of tab to each process :
  for (target=0; target<nb; target++)
```

```
    MPI_Isend(&tab[target*l], l, MPI_INT, target, 2, MPI_COMM_WORLD, &request);
  }

/* ========================== All the processes =========================== */

// Reception of a sub-array for each process :
  MPI_Recv(sub_tab, l, MPI_INT, 0, 2, MPI_COMM_WORLD, &status);

// Display of the elements received by each process :
  printf("Processor %d : sub-array = [ ", rank);
  for (i=0; i<l ; i++)
    printf("%d ", sub_tab[i]);
  printf("]\n");

// Display of the execution time of each process :
  end = MPI_Wtime();
  printf("Processor %d : execution time = %f.\n\n", rank, end-begin);

  MPI_Finalize();

  return 0;
}
```

In the terminal, we get this :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc collect2_v1.c -o ./collect2_v1
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 3 ./collect2_v
1
tab[21] = [  53  12  33  93  87  3  78  11  97  96  19  71  85  75  22  44  57
5  66  88  0 ]

Processor 0 : sub-array = [ 53 12 33 93 87 3 78 ]
Processor 0 : execution time = 0.000108.

Processor 1 : sub-array = [ 11 97 96 19 71 85 75 ]
Processor 1 : execution time = 0.000051.

Processor 2 : sub-array = [ 22 44 57 5 66 88 0 ]
Processor 2 : execution time = 0.000053.

temperville@temperville-alex:~/MPI$ 
```

2. Version 2

Now, we will use a new primitive allowing to prevent using so much *MPI_Isend* and *MPI_Recv*, this is the primitive *MPI_Scatter* :

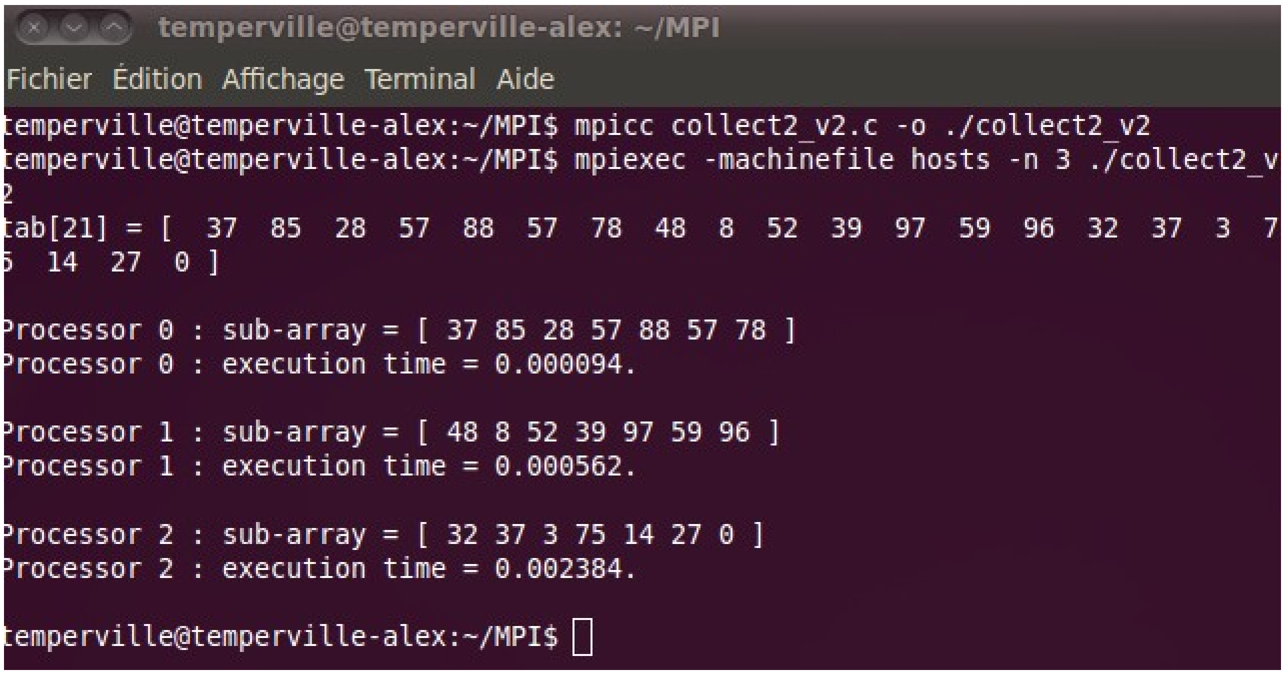
MPI_Scatter(&tab, l, MPI_INT, &sub_tab, l, MPI_INT, ***0****, MPI_COMM_WORLD);*

The process of rank ***0*** sends the array *tab* and distributes its elements between the receiver processes. And each process will receive *l* elements of *tab*. In the virtual machine *MPI_COMM_WORLD*, each process stores in an array *sub_tab* its piece of the array *tab* it receives.

collect2_v2.c

```
# include <stdio.h>
# include <mpi.h>
# include <time.h>
# define N 20

int main()
{
  int M, i, l, nb, rank;
  double begin, end;
  int tab[30], sub_tab[30];

  MPI_Init(NULL,NULL);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nb);

  begin = MPI_Wtime();

// M = length of the considered array to send
// l = length of the sub-arrays
    M = N;
    while ( M % nb != 0)
      M++;
    l = M/nb;

// Process 0 creates the array tab and displays it :
if (rank == 0)
  {
    srand(time(NULL));
    for (i=0; i<M; i++)
      {
        if (i<N)
          tab[i] = rand()%100;
        else
          tab[i] = 0;
      }

  printf("tab[%d] = [ ", M);
  for (i=0; i<M ; i++)
    printf(" %d ", tab[i]);
  printf("]\n\n");
  }

// Process 0 sends tab to each process :
  MPI_Scatter(&tab, l, MPI_INT, &sub_tab, l, MPI_INT, 0, MPI_COMM_WORLD);
// Display of the elements received by each process :
  printf("Processor %d : sub-array = [ ", rank);
  for (i=0; i<l ; i++)
    printf("%d ", sub_tab[i]);
  printf("]\n");

// Display of the execution time of each process :
  end = MPI_Wtime();
  printf("Processor %d : execution time = %f.\n\n", rank, end-begin);

  MPI_Finalize();
  return 0;
}
```

In the terminal, we have :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc collect2_v2.c -o ./collect2_v2
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 3 ./collect2_v
2
tab[21] = [  37  85  28  57  88  57  78  48  8  52  39  97  59  96  32  37  3  7
5  14  27  0 ]

Processor 0 : sub-array = [ 37 85 28 57 88 57 78 ]
Processor 0 : execution time = 0.000094.

Processor 1 : sub-array = [ 48 8 52 39 97 59 96 ]
Processor 1 : execution time = 0.000562.

Processor 2 : sub-array = [ 32 37 3 75 14 27 0 ]
Processor 2 : execution time = 0.002384.

temperville@temperville-alex:~/MPI$ 
```

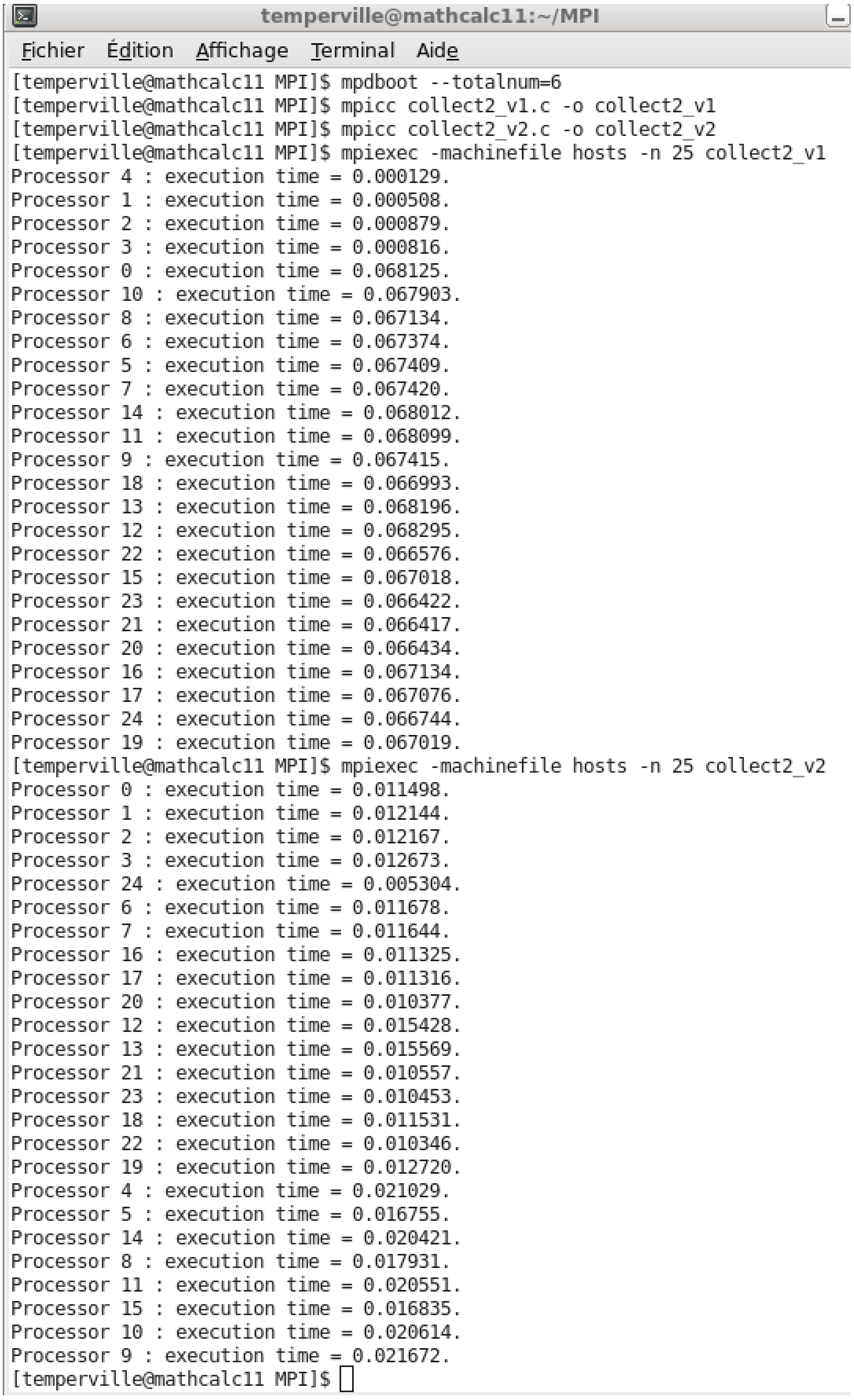
3. Conclusion about execution time of the 2 versions

I run the two previous programs for 25 processes in the room 106 by erasing the display of the arrays for more visibility, we just look at the execution times for the two programs. We saw one display among a lot I have done to make the following conclusion. To do that, I modify my file hosts like this :

hosts

```
# hosts
matcalc10
matcalc11
matcalc01
matcalc02
matcalc03
matcalc04
```

In the terminal, we get :


temperville@mathcalc11:~/MPI
Fichier Édition Affichage Terminal Aide

```
[temperville@mathcalc11 MPI]$ mpdboot --totalnum=6
[temperville@mathcalc11 MPI]$ mpicc collect2_v1.c -o collect2_v1
[temperville@mathcalc11 MPI]$ mpicc collect2_v2.c -o collect2_v2
[temperville@mathcalc11 MPI]$ mpiexec -machinefile hosts -n 25 collect2_v1
Processor 4 : execution time = 0.000129.
Processor 1 : execution time = 0.000508.
Processor 2 : execution time = 0.000879.
Processor 3 : execution time = 0.000816.
Processor 0 : execution time = 0.068125.
Processor 10 : execution time = 0.067903.
Processor 8 : execution time = 0.067134.
Processor 6 : execution time = 0.067374.
Processor 5 : execution time = 0.067409.
Processor 7 : execution time = 0.067420.
Processor 14 : execution time = 0.068012.
Processor 11 : execution time = 0.068099.
Processor 9 : execution time = 0.067415.
Processor 18 : execution time = 0.066993.
Processor 13 : execution time = 0.068196.
Processor 12 : execution time = 0.068295.
Processor 22 : execution time = 0.066576.
Processor 15 : execution time = 0.067018.
Processor 23 : execution time = 0.066422.
Processor 21 : execution time = 0.066417.
Processor 20 : execution time = 0.066434.
Processor 16 : execution time = 0.067134.
Processor 17 : execution time = 0.067076.
Processor 24 : execution time = 0.066744.
Processor 19 : execution time = 0.067019.
[temperville@mathcalc11 MPI]$ mpiexec -machinefile hosts -n 25 collect2_v2
Processor 0 : execution time = 0.011498.
Processor 1 : execution time = 0.012144.
Processor 2 : execution time = 0.012167.
Processor 3 : execution time = 0.012673.
Processor 24 : execution time = 0.005304.
Processor 6 : execution time = 0.011678.
Processor 7 : execution time = 0.011644.
Processor 16 : execution time = 0.011325.
Processor 17 : execution time = 0.011316.
Processor 20 : execution time = 0.010377.
Processor 12 : execution time = 0.015428.
Processor 13 : execution time = 0.015569.
Processor 21 : execution time = 0.010557.
Processor 23 : execution time = 0.010453.
Processor 18 : execution time = 0.011531.
Processor 22 : execution time = 0.010346.
Processor 19 : execution time = 0.012720.
Processor 4 : execution time = 0.021029.
Processor 5 : execution time = 0.016755.
Processor 14 : execution time = 0.020421.
Processor 8 : execution time = 0.017931.
Processor 11 : execution time = 0.020551.
Processor 15 : execution time = 0.016835.
Processor 10 : execution time = 0.020614.
Processor 9 : execution time = 0.021672.
[temperville@mathcalc11 MPI]$ 
```

For the first processes, the *version 1* of this program is faster, but as soon as there are exchanges between computers, the execution time increases a lot. The *version* 2 is more regular and with a lot of machines, it is the best version.

Conclusion : ***MPI_Scatter*** **is a one-to-all which distributes data between the processes and optimizes the program when we use several computers in parallel.**

3) MPI_Reduce primitive

We want to sum the elements of the array in the process of rank *0*. If the size of its array is very huge, it could be very interesting to send parts of this array into others processes which will compute a partial sum of this array, resend their result to the process of rank *0* and then, the process of rank *0* would have just to sum the partial sums to get the total sum of the elements of its array.
If the array has a small size, this idea will be longer than computing directly this sum, because of the time of communication between the processes.
Nevertheless, I apply this program with a small array (of size 20), just to see what it gives. We keep the idea that we have more interest to use it with a huge size (but it won't be readable on the terminal).

Here, using the previous program called *collect2_v2.c*, we want to sum the elements of the array of each process, this is done by the variable *partial_sum*. Now, we want to send this partial sums and add them to obtain the total sum of *tab*. Instead of using *MPI_Isend*, *MPI_Recv*, loops and computing directly the total sum, the *MPI_Reduce* primitive does this with this instruction :

MPI_Reduce(&partial_sum, &total_sum, 1, MPI_INT, MPI_SUM, ***0****, MPI_COMM_WORLD);*

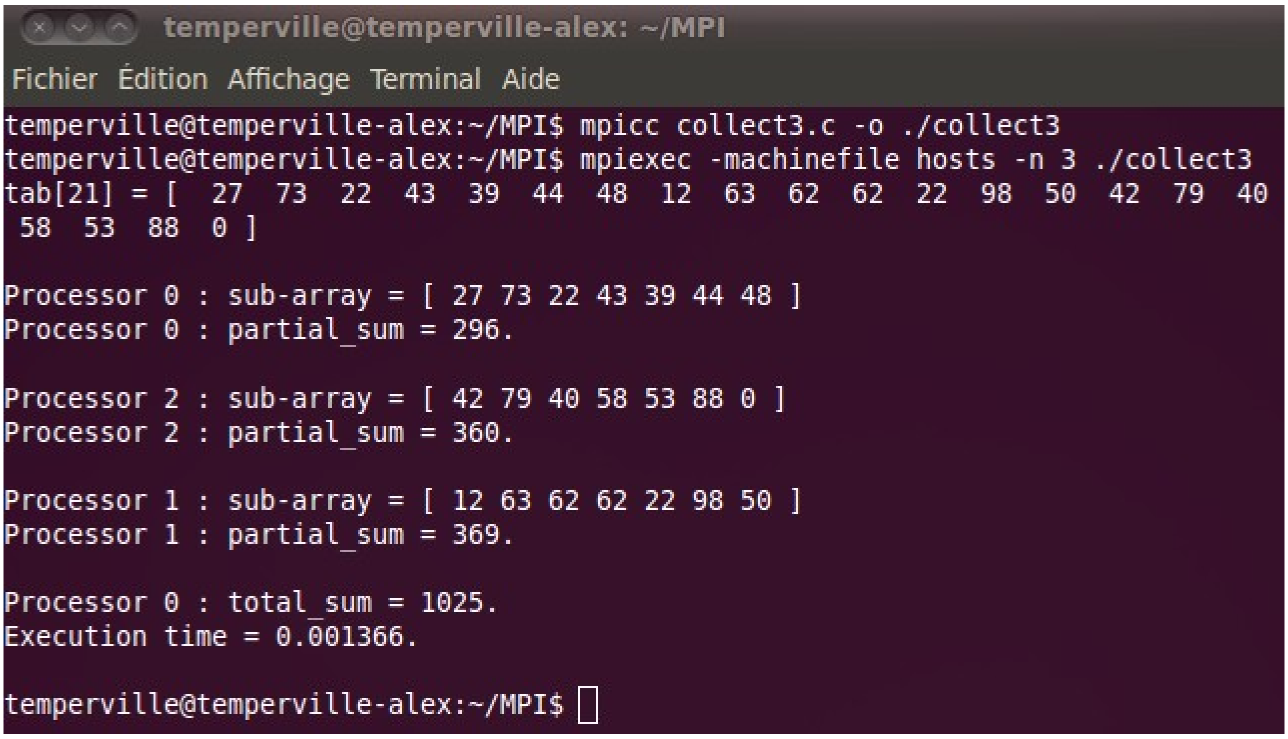
The process of rank ***0*** receives from each process the variable *partial_sum* of size *1* and does a sum (thanks to *MPI_SUM*) which will be stored in the variable *total_sum* (the variables are of type *MPI_INT*).

collect3.c

```
# include <stdio.h>
# include <mpi.h>
# include <stdlib.h>
# define N 20

int main()
{
  int M, i, l, nb, rank, partial_sum, total_sum;
  double begin, end;
  int tab[30], sub_tab[30];

  MPI_Init(NULL,NULL);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nb);

  begin = MPI_Wtime();

// M = length of the considered array to send
// l = length of the sub-arrays
    M = N;
    while ( M % nb != 0)
      M++;
    l = M/nb;

// Process 0 creates the array tab and displays it :
if (rank == 0)
  {
    srand(time(NULL));
    for (i=0; i<M; i++)
      {
        if (i<N)
          tab[i] = rand()%100;
        else
          tab[i] = 0;
      }

  printf("tab[%d] = [ ", M);
  for (i=0; i<M ; i++)
    printf(" %d ", tab[i]);
  printf("]\n\n");
  }

// Process 0 sends tab to each process :
  MPI_Scatter(tab, l, MPI_INT, &sub_tab, l, MPI_INT, 0, MPI_COMM_WORLD);

// Each process computes the sum of the elements of sub_tab it receives :
  partial_sum = 0;
  for (i=0; i<l; i++)
    partial_sum += sub_tab[i];

// Display of the elements received by each process and the partial sums :
  printf("Processor %d : sub-array = [ ", rank);
  for (i=0; i<l ; i++)
    printf("%d ", sub_tab[i]);
  printf("]\n");
  printf("Processor %d : partial_sum = %d.\n\n", rank, partial_sum);

// The process 0 receives all the partial sums and sums them :
  MPI_Reduce(&partial_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

// The process 0 displays the total sum of the elements of tab and displays the execution time of
this program :
  if (rank == 0)
    {
      end = MPI_Wtime();
      printf("Processor 0 : total_sum = %d.\n", total_sum);
      printf("Execution time = %f.\n\n", end-begin);
    }

  MPI_Finalize();
```

```
 return 0;
}
```

In the terminal, we get :



temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc collect3.c -o ./collect3
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 3 ./collect3
tab[21] = [  27  73  22  43  39  44  48  12  63  62  62  22  98  50  42  79  40
 58  53  88  0 ]

Processor 0 : sub-array = [ 27 73 22 43 39 44 48 ]
Processor 0 : partial_sum = 296.

Processor 2 : sub-array = [ 42 79 40 58 53 88 0 ]
Processor 2 : partial_sum = 360.

Processor 1 : sub-array = [ 12 63 62 62 22 98 50 ]
Processor 1 : partial_sum = 369.

Processor 0 : total_sum = 1025.
Execution time = 0.001366.

temperville@temperville-alex:~/MPI$ 
```

<u>Conclusion :</u> *MPI_Reduce* is a all-to-one which receives data from other processes and does operations on them.

Some operations possible with *MPI_Reduce* :

MPI_SUM	*Sum elements*
MPI_PROD	*Multiply elements*
MPI_MAX	*Looks for the maximum*
MPI_MIN	*Looks for the minimum*

4) <u>MPI_Gather primitive</u>

Here, each process build its own array of size *N* and we want to build in the process of rank *0* a big array, by grouping all the arrays. Instead of using *MPI_Isend* and *MPI_recv*, we will use the primitive *MPI_Gather* :

(here I put green and orange to distinguish data about tab and big_tab)

*MPI_Gather(tab, N, MPI_INT, big_tab, N, MPI_INT, **0**, MPI_COMM_WORLD);*

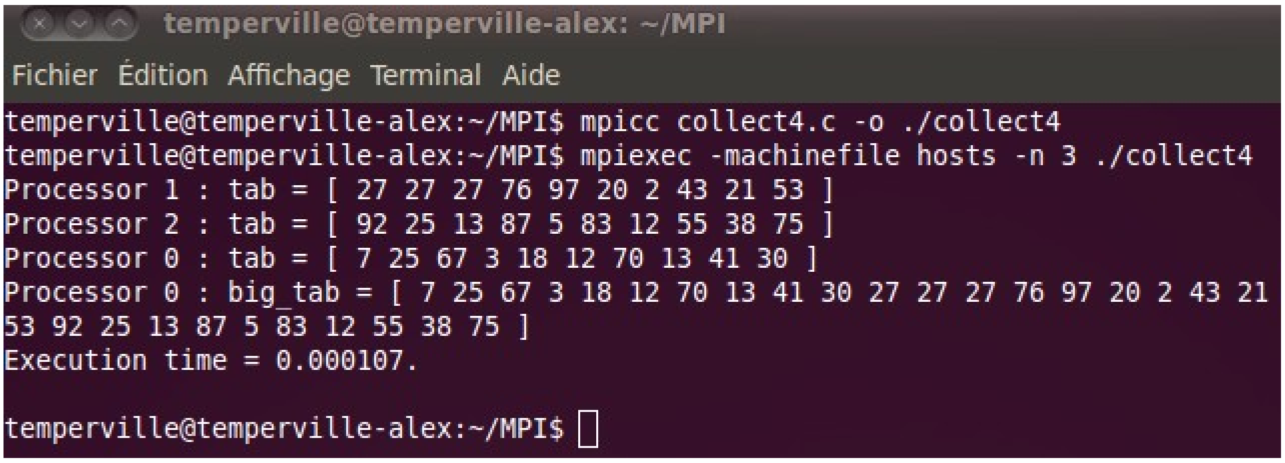
This primitive gathers in *big_tab* (in the process of rank ***0***) *N* elements of the arrays *tab* of size *N* successively.

collect4.c

```c
# include <stdio.h>
# include <mpi.h>
# include <string.h>
# include <stdlib.h>
# define N 10

int main()
{
  int i, nb, rank;
  double begin, end;
  int tab[N];
  int* big_tab;

  MPI_Init(NULL,NULL);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nb);
  big_tab = (int*) malloc(nb*N*sizeof(int));

  begin = MPI_Wtime();

// Allows to initialize the rand() function differently for each process :
  srand(time(NULL)+rank);

// Definition of a random array of N elements in each process :
  for (i=0; i<N; i++)
    tab[i] = rand()%100;

// Display of the elements of the array of each process :
  printf("Processor %d : tab = [ ", rank);
  for (i=0; i<N ; i++)
    printf("%d ", tab[i]);
  printf("]\n");

// Sends the array of each process to the process 0 and puts them into the array big_tab
successively :
  MPI_Gather(tab, N, MPI_INT, big_tab, N, MPI_INT, 0, MPI_COMM_WORLD);

// Display of the array big_tab and the execution time of this program :
  if (rank == 0)
    {
      printf("Processor 0 : big_tab = [ ");
      for (i=0; i<N*nb ; i++)
        printf("%d ", big_tab[i]);
      printf("]\n");
      end = MPI_Wtime();
      printf("Execution time = %f.\n\n", end-begin);
    }

  free(big_tab);
```

```
 MPI_Finalize();
 return 0;
}
```

In the terminal, we get :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc collect4.c -o ./collect4
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 3 ./collect4
Processor 1 : tab = [ 27 27 27 76 97 20 2 43 21 53 ]
Processor 2 : tab = [ 92 25 13 87 5 83 12 55 38 75 ]
Processor 0 : tab = [ 7 25 67 3 18 12 70 13 41 30 ]
Processor 0 : big_tab = [ 7 25 67 3 18 12 70 13 41 30 27 27 27 76 97 20 2 43 21
53 92 25 13 87 5 83 12 55 38 75 ]
Execution time = 0.000107.

temperville@temperville-alex:~/MPI$ 
```

Conclusion : ***MPI_Gather* is a all-to-one sending.**

3rd sheet : Distributed Matrix-Vector computation

1) Generators

In this program, each process builds a part of the triangular upper matrix, *N* sized, where the value of the nonzero elements is *1* in the odd columns and *2* in the even columns.

The program asked ends before the comments "Display of the whole matrix", but in order to display the sub-matrix we have build correctly, I wanted to use the principle of the token we use in the program *ring_comm* of the *1st sheet*. After this comment we can see the transmission of a token, and then we display the sub-matrix process after process.

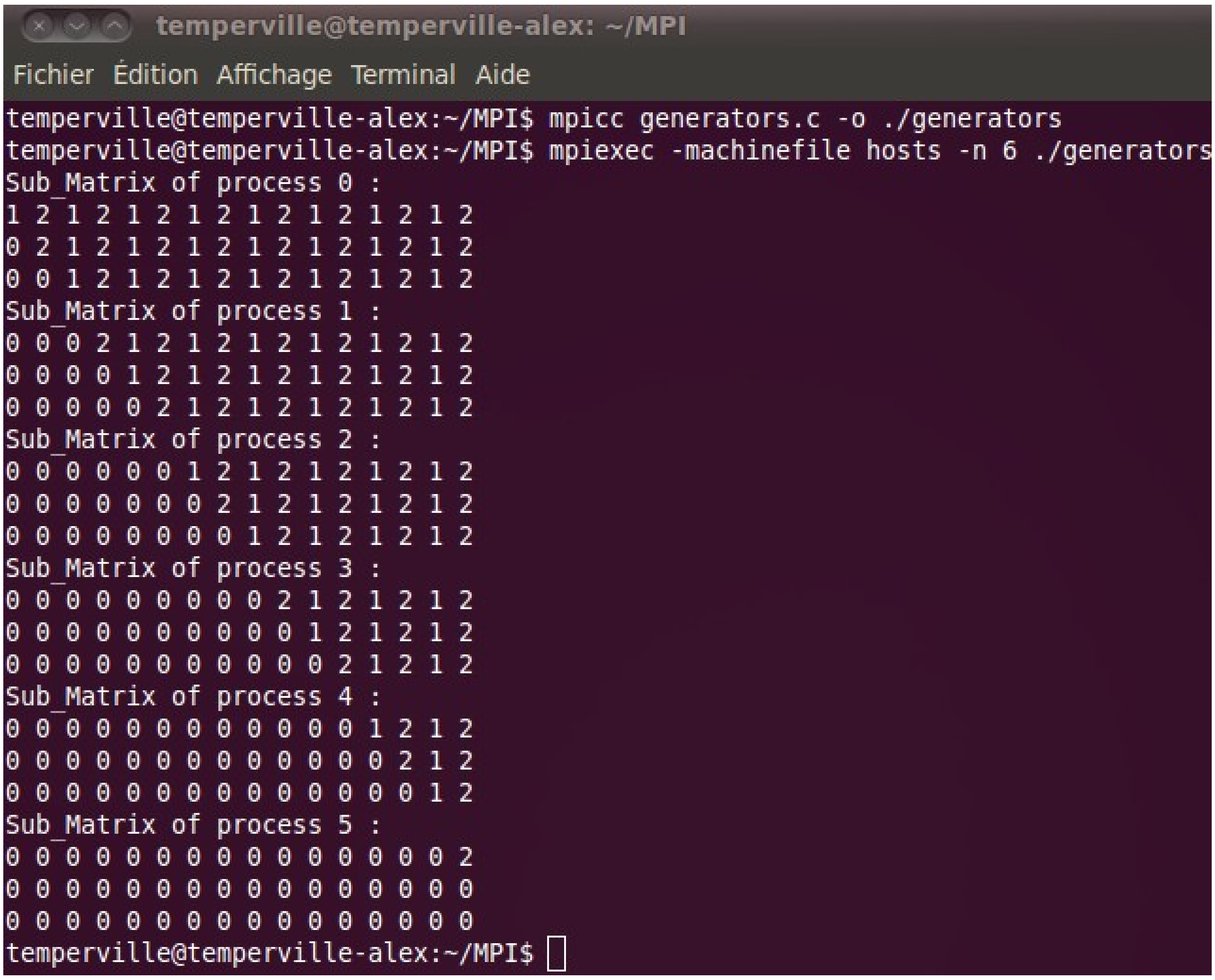
generators.c

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

# define N 16  // matrix size

void Build_My_Matrix1(int** Sub_Mat, int rank, int l)
{
  int i, j;

  for (j=0; j<rank*l; j++)
    Sub_Mat[0][j] = 0;

  for (j=rank*l; j<N; j++) // Creation of the first line.
  {
    if (j%2 == 0)
      Sub_Mat[0][j] = 1;
    else
      Sub_Mat[0][j] = 2;
  }

  for (i=1; i<l; i++) // Creation of the other lines by a recursive process.
  {
    for (j=0; j<N ; j++)
      Sub_Mat[i][j] = Sub_Mat[i-1][j];

    Sub_Mat[i][rank*l-1+i] = 0; // We flip the first non-zero value of the line into a zero.
  }
}

int main()
{
  int M, i, j, k, l, P, rank;
  int **Sub_Mat;
  MPI_Status status;
  MPI_Request request;

  MPI_Init(NULL,NULL);
```

```
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &P);

// M = number of lines of the matrix such that M%P == 0.
// l = number of lines in each sub-matrix
   M = N;
   while ( M % P != 0)
     M++;
   l = M/P;

// Allocation of memory for Sub_Mat :
 Sub_Mat = (int**) malloc(l*sizeof(int*));
 for (i=0; i<l; i++)
   Sub_Mat[i] = (int*) malloc(M*sizeof(int));

// Definition of the matrix Sub_Mat :
 Build_My_Matrix1(Sub_Mat, rank, l);

/* =================== Display of the whole matrix ========================= */

 if (rank == 0)
   {
     k = 1;
     printf("Sub_Matrix of process %d :\n", rank);
     for (i=0; i<l; i++)
       {
         for (j=0; j<N; j++)
              printf("%d ", Sub_Mat[i][j]);

         printf("\n");
       }
     MPI_Isend(&k, 1, MPI_INT, rank+1, rank, MPI_COMM_WORLD, &request);
   }

 if ((rank != 0) && (rank != P-1))
   {
     MPI_Recv(&k, 1, MPI_INT, rank-1, rank-1, MPI_COMM_WORLD, &status);
     printf("Sub_Matrix of process %d :\n", rank);

     for (i=0; i<l; i++)
       {
         for (j=0; j<N; j++)
              printf("%d ", Sub_Mat[i][j]);

         printf("\n");
       }

     MPI_Send(&k, 1, MPI_INT, rank+1, rank, MPI_COMM_WORLD);
   }

 if (rank == P-1)
   {
```

```
// Receives the number sent by the process 'P-2' :
      MPI_Recv(&k, 1, MPI_INT, rank-1, rank-1, MPI_COMM_WORLD, &status);
      printf("Sub_Matrix of process %d :\n", rank);
      for (i=0; i<l; i++)
        {
         for (j=0; j<N; j++)
              printf("%d ", Sub_Mat[i][j]);
         printf("\n");
        }
    }

  for (i=0; i<l; i++)
    free(Sub_Mat[i]);
  free(Sub_Mat);

  MPI_Finalize();
  return 0;
}
```

In the terminal, we have :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc generators.c -o ./generators
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 6 ./generators
Sub_Matrix of process 0 :
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
0 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
0 0 1 2 1 2 1 2 1 2 1 2 1 2 1 2
Sub_Matrix of process 1 :
0 0 0 2 1 2 1 2 1 2 1 2 1 2 1 2
0 0 0 0 1 2 1 2 1 2 1 2 1 2 1 2
0 0 0 0 0 2 1 2 1 2 1 2 1 2 1 2
Sub_Matrix of process 2 :
0 0 0 0 0 0 1 2 1 2 1 2 1 2 1 2
0 0 0 0 0 0 0 2 1 2 1 2 1 2 1 2
0 0 0 0 0 0 0 0 1 2 1 2 1 2 1 2
Sub_Matrix of process 3 :
0 0 0 0 0 0 0 0 0 2 1 2 1 2 1 2
0 0 0 0 0 0 0 0 0 0 1 2 1 2 1 2
0 0 0 0 0 0 0 0 0 0 0 2 1 2 1 2
Sub_Matrix of process 4 :
0 0 0 0 0 0 0 0 0 0 0 0 1 2 1 2
0 0 0 0 0 0 0 0 0 0 0 0 0 2 1 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2
Sub_Matrix of process 5 :
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
temperville@temperville-alex:~/MPI$ 
```

2) Exercise

1. Using MPI_Send and MPI_Recv

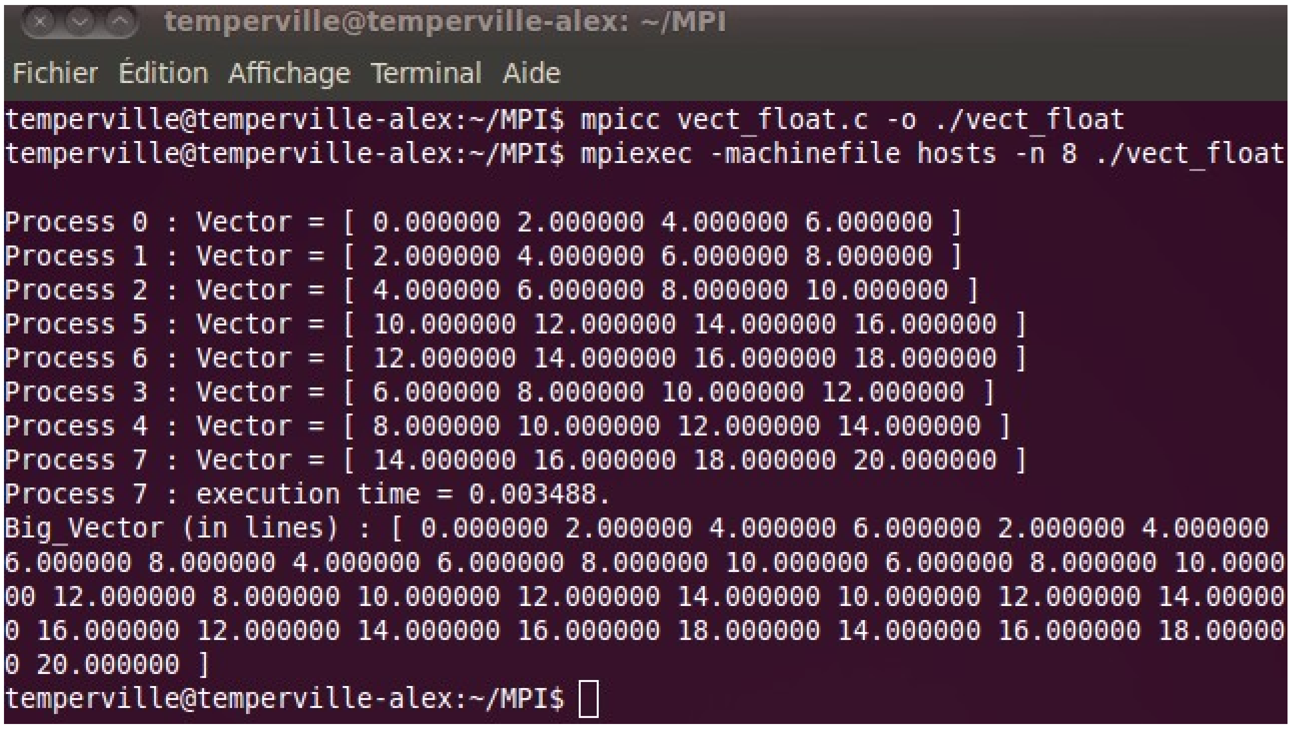
To do an all-to-all exchange of sub-vectors built with the procedure *Build_Vector_1* and with the primitives *MPI_Send* and *MPI_Recv*, we can do the following :

vect_float.c

```
# include <mpi.h>
# include <stdio.h>
# include <stdlib.h>
# define M 4

void Build_Vector_1(int K, int rank, float *Vector)
{
  int line;
  for (line=0; line<K; line++)
    Vector[line] = (rank + line)*2.;
}

int main()
{
  int i, j, k, rank, P;
  int e=0;
  double begin, end;
  float Vector[M];
  float *Big_Vector;
  MPI_Status status;

  MPI_Init(NULL,NULL);
  begin = MPI_Wtime();

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &P);

  Big_Vector = (float *) calloc(P*M,sizeof(float));

// Each process builds its own vector :
  Build_Vector_1(M, rank, Vector);

// We stock this vector in the Big_Vector (instead of sending to itself, that is useless
// in this case where we have to use only MPI_Send and MPI_Recv primitives) :
  for (i=0; i<M; i++)
    Big_Vector[rank*M+i] = Vector[i];

/* We begin to consider sending process one after one.
   In each consideration, there is one MPI_Send and one MPI_Recv in P iterations.
   When one process has sent its vector to everyone and that everyone has received it,
   we can consider the following process to do the same thing. */
  for (i=0; i<P; i++)
```

```
    {
      if (rank == i)
            for (k=0; k<P; k++)
              {
                if (k != rank)
                  MPI_Send(Vector, M, MPI_FLOAT, k, rank, MPI_COMM_WORLD);
              if (e == 0)
                {
                  e++;
                      printf("Process %d : Vector = [ ", rank);
                          for (j=0; j<M; j++)
                            printf("%f ", Vector[j]);
                          printf("]\n");
                }
              }
      else // (rank != i)
            MPI_Recv(&Big_Vector[i*M], M, MPI_FLOAT, i, i, MPI_COMM_WORLD, &status);
    }

// Display of Big_Vector by an arbitrary process
  if (rank == P-1)
    {
      end = MPI_Wtime();
      printf("Process %d : execution time = %f.\n", rank, end-begin);
      printf("Big_Vector (in lines) : [ ");
      for (i=0; i<M*P; i++)
            printf("%f ", Big_Vector[i]);
      printf("]\n");
    }

  free(Big_Vector);
  MPI_Finalize();
  return 0;
}
```

In the terminal, we get :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc vect_float.c -o ./vect_float
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 8 ./vect_float

Process 0 : Vector = [ 0.000000 2.000000 4.000000 6.000000 ]
Process 1 : Vector = [ 2.000000 4.000000 6.000000 8.000000 ]
Process 2 : Vector = [ 4.000000 6.000000 8.000000 10.000000 ]
Process 5 : Vector = [ 10.000000 12.000000 14.000000 16.000000 ]
Process 6 : Vector = [ 12.000000 14.000000 16.000000 18.000000 ]
Process 3 : Vector = [ 6.000000 8.000000 10.000000 12.000000 ]
Process 4 : Vector = [ 8.000000 10.000000 12.000000 14.000000 ]
Process 7 : Vector = [ 14.000000 16.000000 18.000000 20.000000 ]
Process 7 : execution time = 0.003488.
Big_Vector (in lines) : [ 0.000000 2.000000 4.000000 6.000000 2.000000 4.000000
6.000000 8.000000 4.000000 6.000000 8.000000 10.000000 6.000000 8.000000 10.0000
00 12.000000 8.000000 10.000000 12.000000 14.000000 10.000000 12.000000 14.00000
0 16.000000 12.000000 14.000000 16.000000 18.000000 14.000000 16.000000 18.00000
0 20.000000 ]
temperville@temperville-alex:~/MPI$ 
```

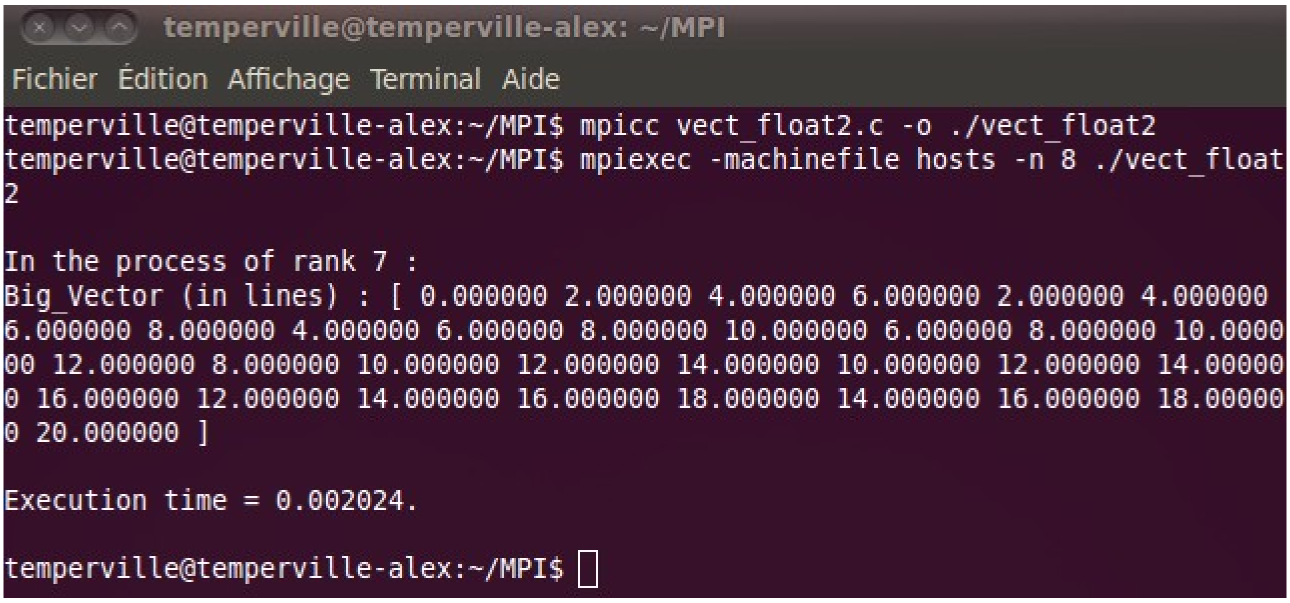
2. Using MPI_Gather (optimized program)

In this part, I wanted to perform the previous program using the primitive *MPI_Gather*. I obtain this :

vect_float2.c

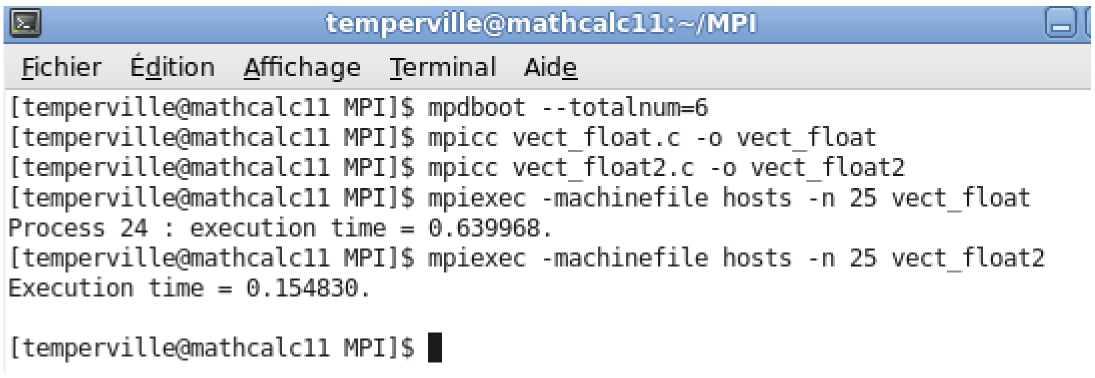
```c
# include <mpi.h>
# include <stdio.h>
# include <stdlib.h>
# define M 4

void Build_Vector_1(int K, int rank, float *Vector)
{
  int line;
  for (line=0; line<K; line++)
    Vector[line] = (rank + line)*2.;
}

int main()
{
  int i, k, rank, P;
  double begin, end;
  float Vector[M];
  float *Big_Vector;

  MPI_Init(NULL,NULL);
  begin = MPI_Wtime();
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &P);
  Big_Vector = (float *) calloc(P*M, sizeof(float));

// Each process builds its own vector :
  Build_Vector_1(M, rank, Vector);

// Sends the array of each process to the process k and puts them into the array Big_Vector
successively :
  for(k=0; k<P; k++)
    MPI_Gather(Vector, M, MPI_FLOAT, Big_Vector, M, MPI_FLOAT, k,
MPI_COMM_WORLD);

// Display of Big_Vector by an arbitrary process :
  if (rank == P-1)
    {
      printf("\nIn the process of rank %d :\nBig_Vector (in lines) : [ ", rank);
      for (i=0; i<M*P; i++)
          printf("%f ", Big_Vector[i]);
      printf("]\n\n");
      end = MPI_Wtime();
      printf("Execution time = %f.\n\n", end-begin);
    }

  free(Big_Vector);
  MPI_Finalize();
  return 0;
}
```

In the terminal, we get :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc vect_float2.c -o ./vect_float2
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 8 ./vect_float
2

In the process of rank 7 :
Big_Vector (in lines) : [ 0.000000 2.000000 4.000000 6.000000 2.000000 4.000000
6.000000 8.000000 4.000000 6.000000 8.000000 10.000000 6.000000 8.000000 10.0000
00 12.000000 8.000000 10.000000 12.000000 14.000000 10.000000 12.000000 14.00000
0 16.000000 12.000000 14.000000 16.000000 18.000000 14.000000 16.000000 18.00000
0 20.000000 ]

Execution time = 0.002024.

temperville@temperville-alex:~/MPI$ 
```

With my own computer, this result does not seem optimized but as soon as I use it in the room 106 with 25 processes in 5 machines, we get this and we can observe that indeed, the second version of this program is the best one :


temperville@mathcalc11:~/MPI
Fichier Édition Affichage Terminal Aide

```
[temperville@mathcalc11 MPI]$ mpdboot --totalnum=6
[temperville@mathcalc11 MPI]$ mpicc vect_float.c -o vect_float
[temperville@mathcalc11 MPI]$ mpicc vect_float2.c -o vect_float2
[temperville@mathcalc11 MPI]$ mpiexec -machinefile hosts -n 25 vect_float
Process 24 : execution time = 0.639968.
[temperville@mathcalc11 MPI]$ mpiexec -machinefile hosts -n 25 vect_float2
Execution time = 0.154830.

[temperville@mathcalc11 MPI]$ 
```

3) Problem

1. A first program

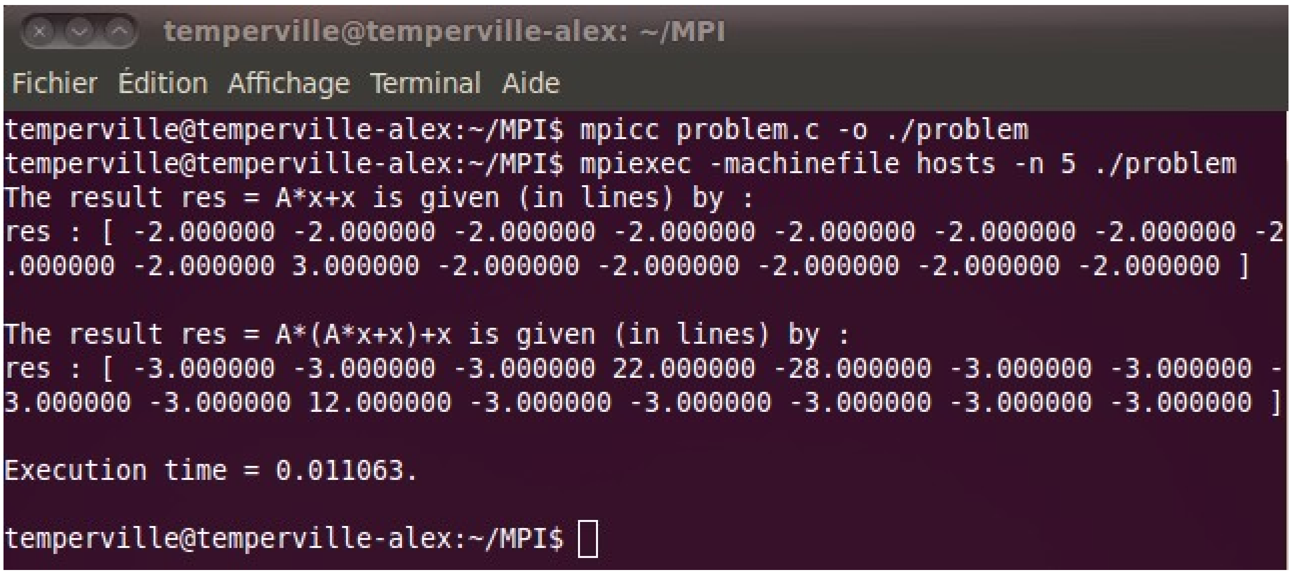
This program is very long, that is why I propose in the second part of this question to give a shorter program. I fix *N = 15* in this problem but we could ask the user with a *scanf* to give the size *N* he wants for the matrix *A*. I think that with all the explanations I have done before and the comments I put in this program, it is understandable.

problem.c

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

// Procedure which builds a sub-matrix of A :
void Build_Matrix(int N, int P, int rank, double **Matrix)
{
  int h = N/P;
  int line, col, local_line;
  for (line=h*rank; line<h*(rank+1); line++)
    {
      local_line = line - h*rank;
      for (col=0; col<N; col++)
            {
              if (col == line)
                Matrix[local_line][col] = 1.0;
              else if (col == line+5)
                Matrix[local_line][col] = -5.0;
              else if (col == line+6)
                Matrix[local_line][col] = 5.0;
              else
                Matrix[local_line][col] = 0.0;
            }
    }
}

// Procedure which builds the vector x :
void Build_Vector_2(int N, double *Vector)
{
  int line;
  for (line=0; line<N; line++)
    Vector[line] = -1.0;
}

int main()
{
  int i, j, k, M, h, rank, P, iter;
  double begin, end;
  double *x, *y, *res;
  double **Sub_Mat;
  int N = 15;
  MPI_Status status;

  MPI_Init(NULL,NULL);
  begin = MPI_Wtime();

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &P);

  // M = number of lines of the matrix A such that M%P == 0.
  // h = number of lines in each sub-matrix
  M = N;
  while ( M % P != 0)
    M++;
  h = M/P;

  // Allocation of memory for vectors and matrix we use :
  x = (double*) malloc(M*sizeof(double));
  y = (double*) malloc(h*sizeof(double));
  res = (double*) malloc(M*sizeof(double));

  Sub_Mat = (double**) malloc(h*sizeof(double*));
  for (i=0; i<h; i++)
    Sub_Mat[i] = (double*) malloc(M*sizeof(double));

  // Each process creates its own vector x and sub-matrix of A :
  Build_Vector_2(M, x);
  Build_Vector_2(M, res); // so x == res.
  Build_Matrix(M, P, rank, Sub_Mat);

// We iterate twice the computation of A*res+x (the first computation makes A*x+x (as res=x), and
the next one makes A*res+x = A*(A*x+x)+x) :

  for (iter=0; iter<2; iter++)
    {
      // Partial construction of y = A*res + x :
      for (i=0; i<h; i++)
            {
              y[i] = 0;
              for (j=0; j<M; j++)
                y[i] = y[i] + Sub_Mat[i][j]*res[j];
```

```
           y[i] = y[i] + x[i];
         }

     // Each process puts its partial solution y into the global solution res :
     for (i=0; i<h; i++)
         res[rank*h+i] = y[i];

     // Each process sends to every process of different rank its peace of res :
     for (i=0; i<P; i++)
         {
           if (rank == i)
             {
              for (k=0; k<P; k++)
                   {
                    if (k != rank)
                      MPI_Send(y, h, MPI_DOUBLE, k, rank, MPI_COMM_WORLD);
                   }
             }

           else // (rank != i)
             MPI_Recv(&res[i*h], h, MPI_DOUBLE, i, i, MPI_COMM_WORLD, &status);
         }

     // Display of res by the last process
     if (rank == P-1)
         {

        if (iter == 0)
         {
              printf("The result res = A*x+x is given (in lines) by :\nres : [ ");
              for (i=0; i<M; i++)
                   printf("%f ", res[i]);
              printf("]\n\n");
         }

           else // (iter == 1)
                    {
                     printf("The result res = A*(A*x+x)+x is given (in lines) by :\nres : [ ");
                     for (i=0; i<M; i++)
                       printf("%f ", res[i]);
                     printf("]\n\n");

                     end = MPI_Wtime();
                     printf("Execution time = %f.\n\n", end-begin);
                    }
         }
    }

// Deallocation of the arrays :
  free(x);
  free(y);
  free(res);
  for (i=0; i<h; i++)
    free(Sub_Mat[i]);
  free(Sub_Mat);

  MPI_Finalize();
  return 0;
}
```

In the terminal, we get :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc problem.c -o ./problem
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 5 ./problem
The result res = A*x+x is given (in lines) by :
res : [ -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2
.000000 -2.000000 3.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 ]

The result res = A*(A*x+x)+x is given (in lines) by :
res : [ -3.000000 -3.000000 -3.000000 22.000000 -28.000000 -3.000000 -3.000000 -
3.000000 -3.000000 12.000000 -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 ]

Execution time = 0.011063.

temperville@temperville-alex:~/MPI$ 
```

2. A second program

Here, I perform the previous program by using *MPI_Gather.* I erase also the display of the results in the program but I keep it in the execution to see what we obtain.

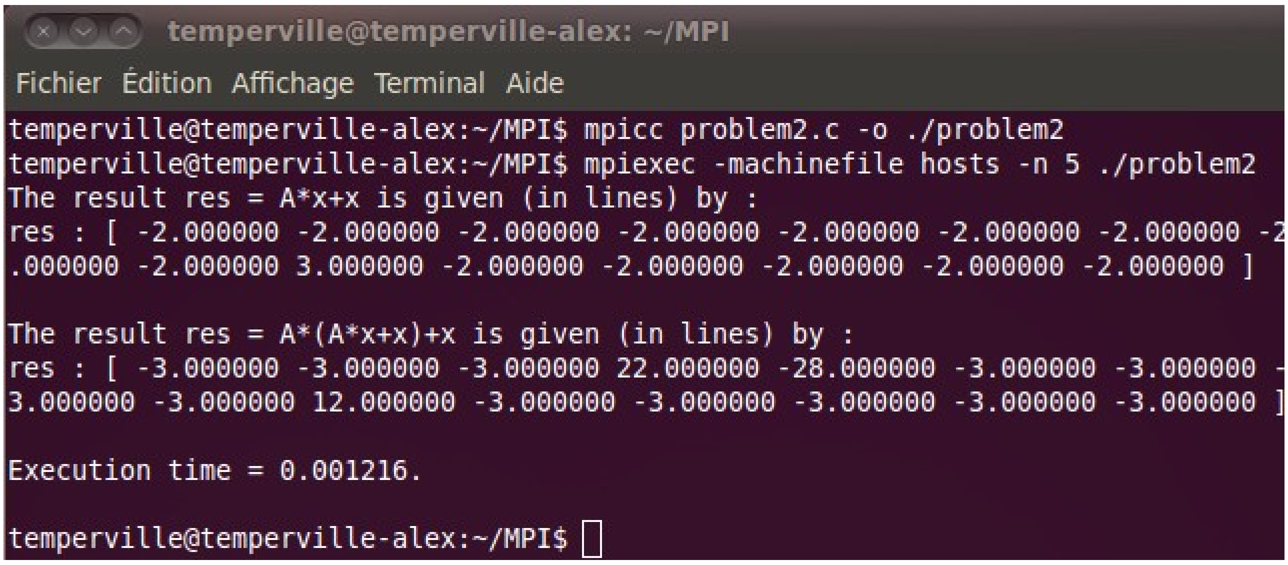
problem2.c

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

// Procedure which builds a sub-matrix of A :
void Build_Matrix(int N, int P, int rank, double **Matrix)
{
  int h = N/P;
  int line, col, local_line;
  for (line=h*rank; line<h*(rank+1); line++)
    {
      local_line = line - h*rank;
      for (col=0; col<N; col++)
         {
           if (col == line)
             Matrix[local_line][col] = 1.0;
           else if (col == line+5)
```

```
              Matrix[local_line][col] = -5.0;
            else if (col == line+6)
              Matrix[local_line][col] = 5.0;
            else
              Matrix[local_line][col] = 0.0;
          }
    }
}

// Procedure which builds the vector x :
void Build_Vector_2(int N, double *Vector)
{
  int line;
  for (line=0; line<N; line++)
    Vector[line] = -1.0;
}

int main()
{
  int i, j, k, M, h, rank, P, iter;
  double begin, end;
  double *x, *y, *res;
  double **Sub_Mat;
  int N = 15;

  MPI_Init(NULL,NULL);
  begin = MPI_Wtime();

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &P);

  // M = number of lines of the matrix A such that M%P == 0.
  // h = number of lines in each sub-matrix
  M = N;
  while ( M % P != 0)
    M++;
  h = M/P;

  // Allocation of memory for vectors and matrix we use :
  x = (double*) malloc(M*sizeof(double));
  y = (double*) malloc(h*sizeof(double));
  res = (double*) malloc(M*sizeof(double));

  Sub_Mat = (double**) malloc(h*sizeof(double*));
  for (i=0; i<h; i++)
    Sub_Mat[i] = (double*) malloc(M*sizeof(double));

  // Each process creates its own vector x and sub-matrix of A :
  Build_Vector_2(M, x);
  Build_Vector_2(M, res); // so x == res.
  Build_Matrix(M, P, rank, Sub_Mat);
```

```
// We iterate twice the computation of A*res+x (the first computation makes A*x+x (as res=x), and
the next one makes A*res+x = A*(A*x+x)+x)
  for (iter=0; iter<2; iter++)
    {
      // Partial construction of y = A*res + x :
      for (i=0; i<h; i++)
        {
          y[i] = x[i];
          for (j=0; j<M; j++)
            y[i] = y[i] + Sub_Mat[i][j]*res[j];
        }

// Sends the vector y of each process to the processes k and puts them into the vector res
successively :
  for(k=0; k<P; k++)
    MPI_Gather(y, h, MPI_DOUBLE, res, h, MPI_DOUBLE, k, MPI_COMM_WORLD);

  free(x);
  free(y);
  free(res);
  for (i=0; i<h; i++)
    free(Sub_Mat[i]);
  free(Sub_Mat);
  MPI_Finalize();
  return 0;
}
```

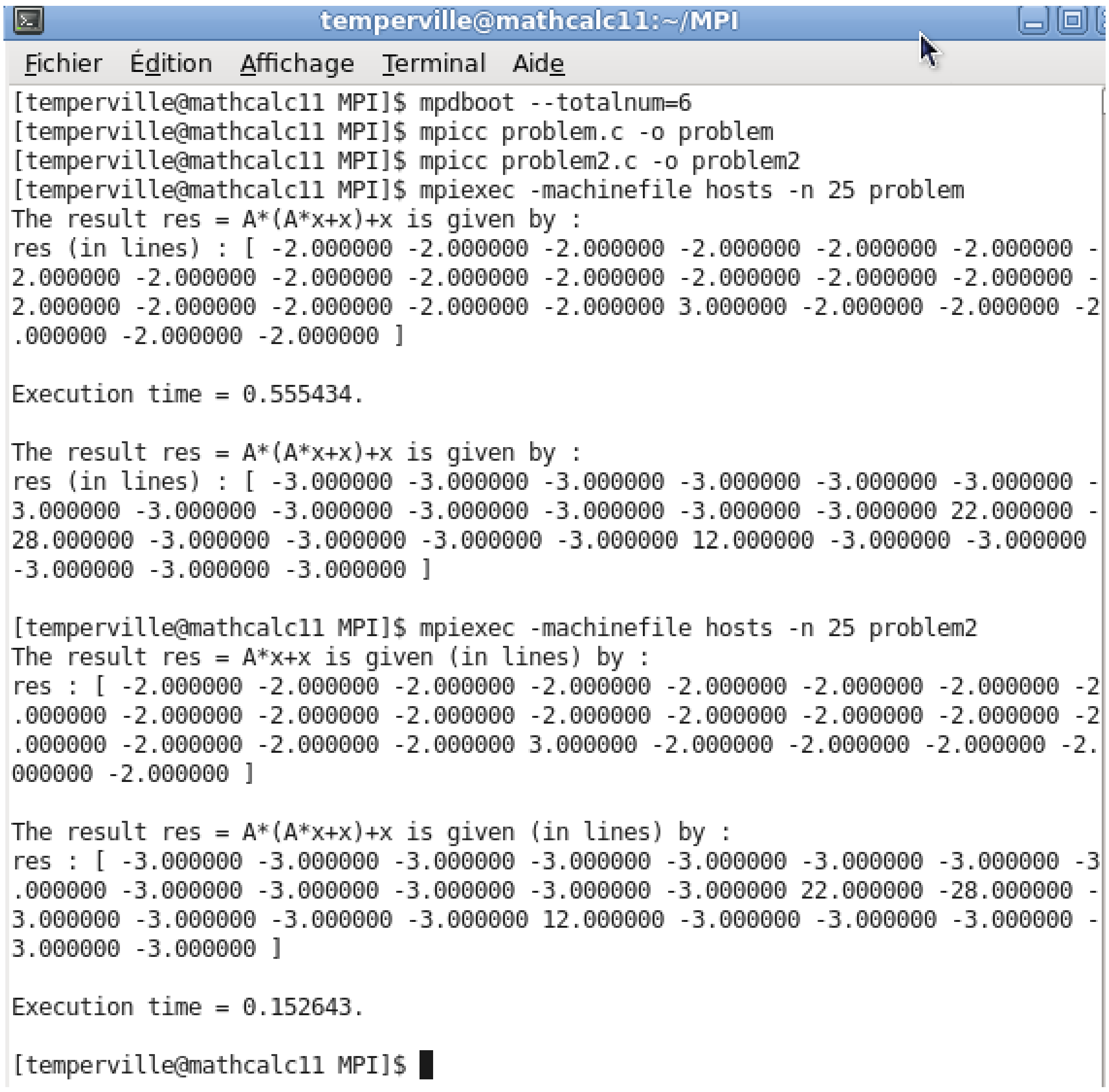
In the terminal, we get :


temperville@temperville-alex: ~/MPI
Fichier Édition Affichage Terminal Aide

```
temperville@temperville-alex:~/MPI$ mpicc problem2.c -o ./problem2
temperville@temperville-alex:~/MPI$ mpiexec -machinefile hosts -n 5 ./problem2
The result res = A*x+x is given (in lines) by :
res : [ -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2
.000000 -2.000000 3.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 ]

The result res = A*(A*x+x)+x is given (in lines) by :
res : [ -3.000000 -3.000000 -3.000000 22.000000 -28.000000 -3.000000 -3.000000 -
3.000000 -3.000000 12.000000 -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 ]

Execution time = 0.001216.

temperville@temperville-alex:~/MPI$ 
```

3. Display with more computers

In the room 106, with 25 processes, we obtain this in the terminal, and we can see that the second version is faster than the first one :


temperville@mathcalc11:~/MPI
Fichier Édition Affichage Terminal Aide

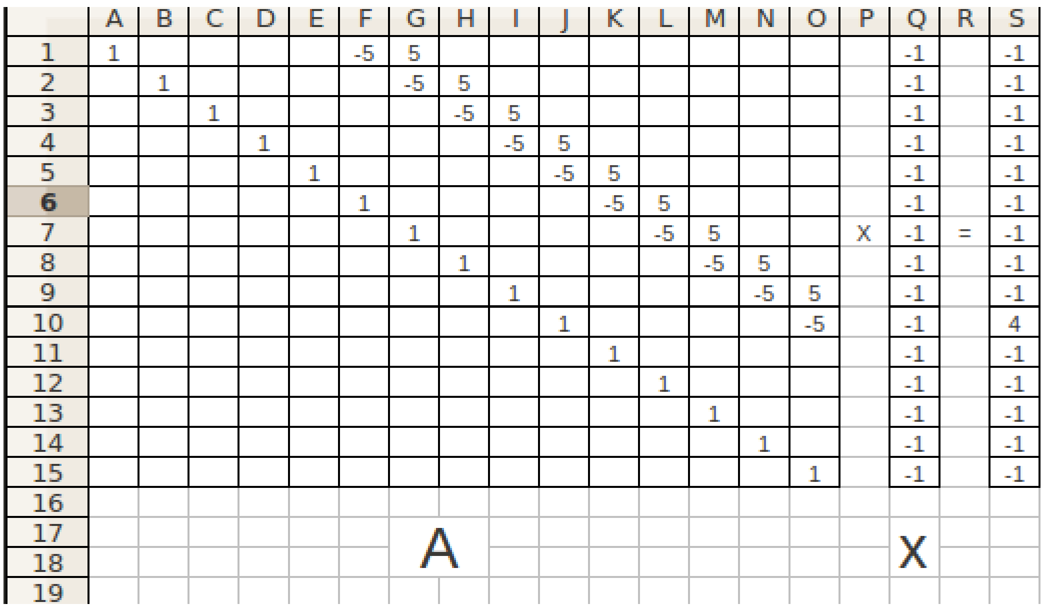
```
[temperville@mathcalc11 MPI]$ mpdboot --totalnum=6
[temperville@mathcalc11 MPI]$ mpicc problem.c -o problem
[temperville@mathcalc11 MPI]$ mpicc problem2.c -o problem2
[temperville@mathcalc11 MPI]$ mpiexec -machinefile hosts -n 25 problem
The result res = A*(A*x+x)+x is given by :
res (in lines) : [ -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -
2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -
2.000000 -2.000000 -2.000000 -2.000000 -2.000000 3.000000 -2.000000 -2.000000 -2
.000000 -2.000000 -2.000000 ]

Execution time = 0.555434.

The result res = A*(A*x+x)+x is given by :
res (in lines) : [ -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 -
3.000000 -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 22.000000 -
28.000000 -3.000000 -3.000000 -3.000000 -3.000000 12.000000 -3.000000 -3.000000
-3.000000 -3.000000 -3.000000 ]

[temperville@mathcalc11 MPI]$ mpiexec -machinefile hosts -n 25 problem2
The result res = A*x+x is given (in lines) by :
res : [ -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2
.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2.000000 -2
.000000 -2.000000 -2.000000 -2.000000 3.000000 -2.000000 -2.000000 -2.000000 -2.
000000 -2.000000 ]

The result res = A*(A*x+x)+x is given (in lines) by :
res : [ -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 -3
.000000 -3.000000 -3.000000 -3.000000 -3.000000 -3.000000 22.000000 -28.000000 -
3.000000 -3.000000 -3.000000 -3.000000 12.000000 -3.000000 -3.000000 -3.000000 -
3.000000 -3.000000 ]

Execution time = 0.152643.

[temperville@mathcalc11 MPI]$ █
```

4. <u>Verification</u>

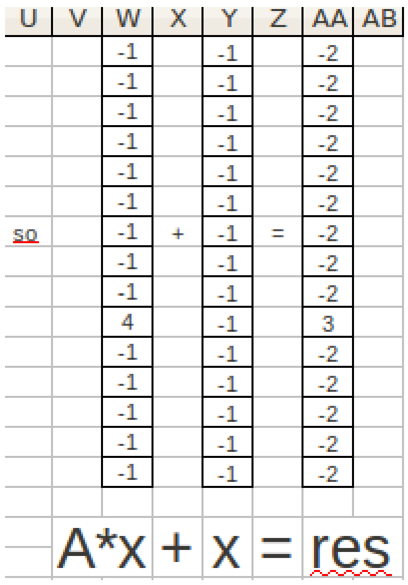
Thanks to a spreadsheet, for *N=15*, I verify my results for by doing the operations res = A*x+x, then res2 = A*res+x. This is what our two previous programs do.

We notice we get the same answers.

For the first calculation, we have thanks to the spreadsheet :

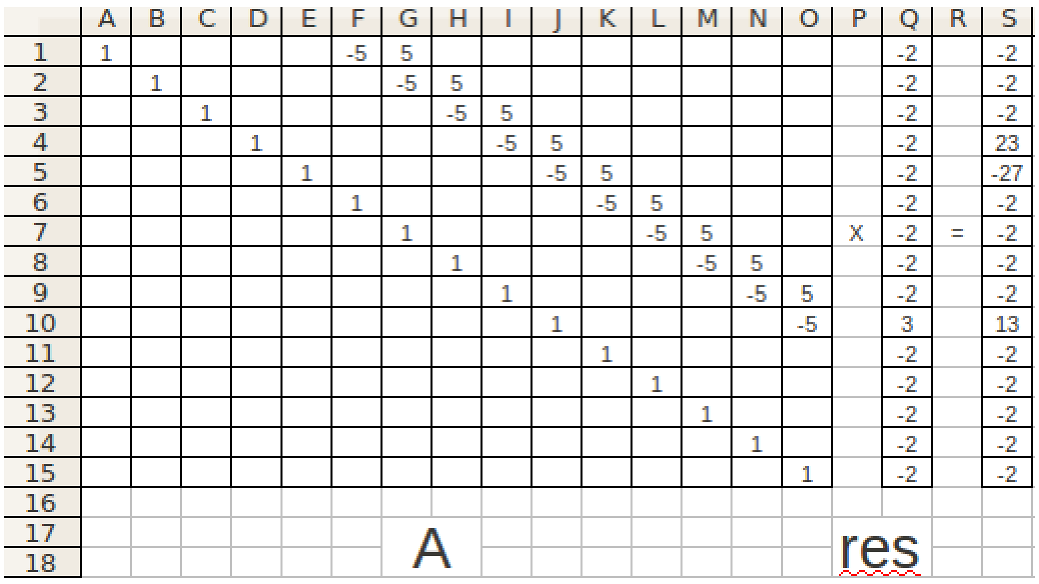


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	1					-5	5										-1		-1
2		1					-5	5									-1		-1
3			1					-5	5								-1		-1
4				1					-5	5							-1		-1
5					1					-5	5						-1		-1
6						1					-5	5					-1		-1
7							1					-5	5			X	-1	=	-1
8								1					-5	5			-1		-1
9									1					-5	5		-1		-1
10										1					-5		-1		4
11											1						-1		-1
12												1					-1		-1
13													1				-1		-1
14														1			-1		-1
15															1		-1		-1
16																			
17							A										X		
18																			
19																			

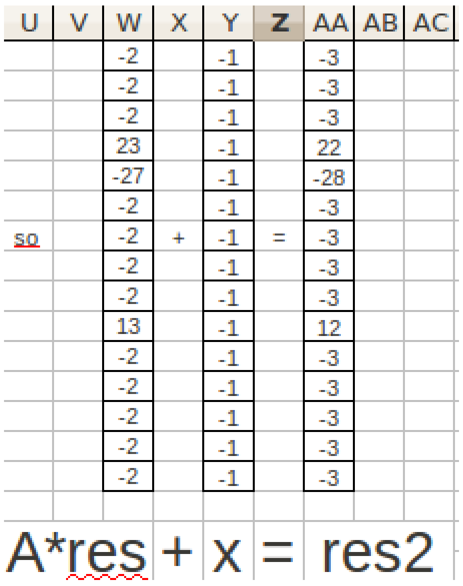


U	V	W	X	Y	Z	AA	AB
		-1		-1		-2	
		-1		-1		-2	
		-1		-1		-2	
		-1		-1		-2	
		-1		-1		-2	
		-1		-1		-2	
so		-1	+	-1	=	-2	
		-1		-1		-2	
		-1		-1		-2	
		4		-1		3	
		-1		-1		-2	
		-1		-1		-2	
		-1		-1		-2	
		-1		-1		-2	
		-1		-1		-2	
	A*x		+	x	=	res	

For the second calculation, we have thanks to the spreadsheet :



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	1					-5	5										-2		-2
2		1					-5	5									-2		-2
3			1					-5	5								-2		-2
4				1					-5	5							-2		23
5					1					-5	5						-2		-27
6						1					-5	5					-2		-2
7							1					-5	5			X	-2	=	-2
8								1					-5	5			-2		-2
9									1					-5	5		-2		-2
10										1					-5		3		13
11											1						-2		-2
12												1					-2		-2
13													1				-2		-2
14														1			-2		-2
15															1		-2		-2
16																			
17																			
18							A										res		



U	V	W	X	Y	Z	AA	AB	AC
		-2		-1		-3		
		-2		-1		-3		
		-2		-1		-3		
		23		-1		22		
		-27		-1		-28		
		-2		-1		-3		
so		-2	+	-1	=	-3		
		-2		-1		-3		
		-2		-1		-3		
		13		-1		12		
		-2		-1		-3		
		-2		-1		-3		
		-2		-1		-3		
		-2		-1		-3		
		-2		-1		-3		
A*res			+	x	=	res2		