Universitat Oberta de Catalunya

High Performance Computing - CET3

Introduction to MPI

Juan José Rodríguez Aldavero

May 20, 2020

Contents

1 MPI execution environment

In this deliverable we will be introduced to programming using MPI (Message Passing Interface) library. This library is useful to create programs that can work efficiently in parallel and/or distributed architectures, and its advantages are its high degree of portability and flexibility to be used in a heterogeneous set of hardware.

We started exploring this library by programming and running a very simple program with MPI.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  printf("Hello World!\n");
  MPI_Finalize();
}
```

To execute it, we create a script in SGE specifying the number of processes.

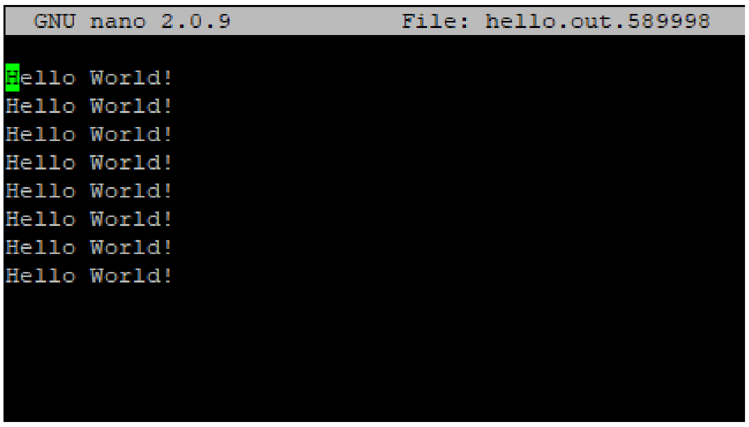
```
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N hello
#$ -o hello.out.$JOB_ID
#$ -e hello.err.$JOB_ID
#$ -pe orte 8

mpirun -np 8 ./hello
```

The compilation will be similar to the other examples in the course but using the mpicc command instead of gcc:

```
mpicc hello.c -o hello
```

The MPI_init command starts the parallel execution section by creating a number of processes ("child" processes) such that the number of processes specified by the mpirun -np command is reached.

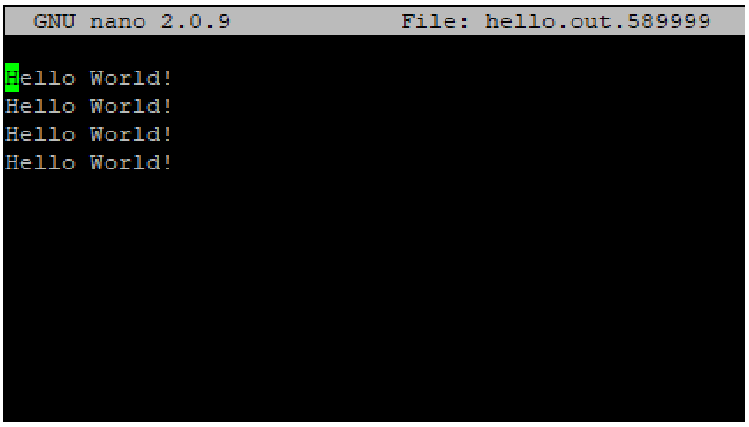
1. **What is the result of the execution of the MPI program discussed above? Why?**
   The result of the execution is the writing in standard output of 8 "Hello World!" messages. This is because 8 different processes have been specified when compiling it, and the code has been executed independently for each of these processes.



```
  GNU nano 2.0.9              File: hello.out.589998

Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Figure 1.1: Results of the execution of hello.c for 8 processes

2. **Explain what happens if you use the mpirun option “-np 4” instead of “-np 8” with the example above.**
   As expected, the number of messages has been reduced to the number of available processes.



```
  GNU nano 2.0.9              File: hello.out.589999

Hello World!
Hello World!
Hello World!
Hello World!
```

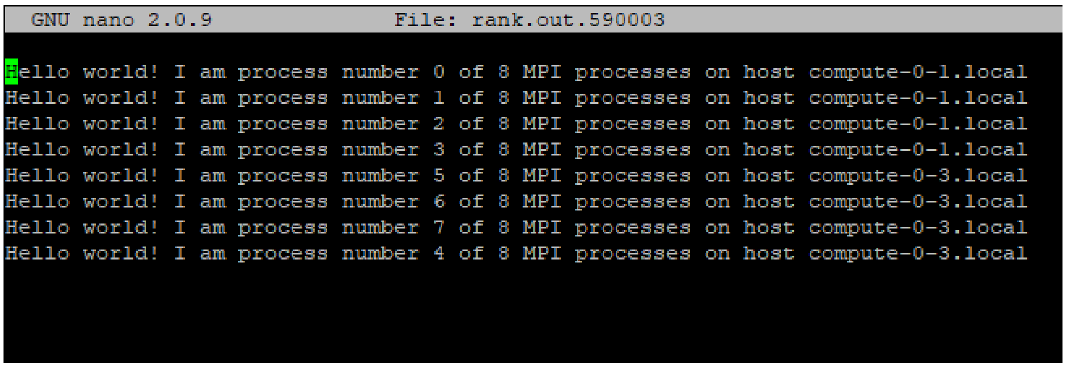
Figure 1.2: Results of the execution of hello.c for 4 processes

2 MPI processes

We continue the work by studying the variable *rank*. MPI allows the creation of logical groups of processes, and within these groups the processes are identified by their rank. That is, given a group of N elements, the rank is an integer $n[0, N - 1]$ that identifies each process. This makes the rank of a process always relative to each process. In MPI we can obtain the range relative to a process by using the command *MPI_Comm_rank*. This procedure is illustrated in the code provided by the course.

```
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv)
{
  int rank, numprocs;
  char hostname[256];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  gethostname(hostname,255);

  printf("Hello world! I am process number %d of %d MPI processes on
      host %s\n", rank,
  numprocs, hostname);
  MPI_Finalize();
  return 0;
}
```

3. **What is the result of the execution of the MPI program above? Why?**
   The result of the code is a set of 8 messages, one for each process, showing the rank of the process within the group (specified by the *MPI_Comm_rank* method and stored in the integer variable *rank*), the number of total processes (specified by the *MPI_Comm_size* method and stored in the integer variable *numprocs*) and the node at which the processes are executed. As each of the nodes of the cluster has four threads, two nodes have been used in the execution of the code as shown on the screen.

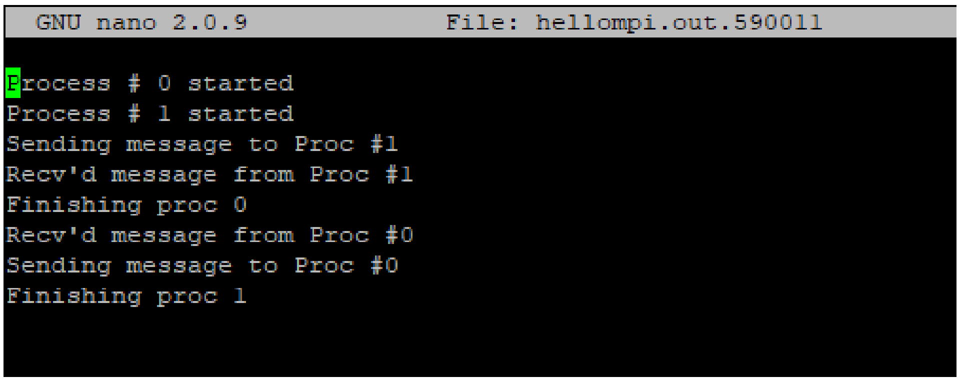


```
  GNU nano 2.0.9              File: rank.out.590003

Hello world! I am process number 0 of 8 MPI processes on host compute-0-1.local
Hello world! I am process number 1 of 8 MPI processes on host compute-0-1.local
Hello world! I am process number 2 of 8 MPI processes on host compute-0-1.local
Hello world! I am process number 3 of 8 MPI processes on host compute-0-1.local
Hello world! I am process number 5 of 8 MPI processes on host compute-0-3.local
Hello world! I am process number 6 of 8 MPI processes on host compute-0-3.local
Hello world! I am process number 7 of 8 MPI processes on host compute-0-3.local
Hello world! I am process number 4 of 8 MPI processes on host compute-0-3.local
```

Figure 2.1: *Results of the execution of rank.c for 8 processes*

3 MPI point to point communications

We continue to study the transmission of messages between processes. A simple code has been provided that opens two MPI processes and sends messages between them called hellompi.c. Executing this program we get the following result:



```
 GNU nano 2.0.9                File: hellompi.out.590011

Process # 0 started
Process # 1 started
Sending message to Proc #1
Recv'd message from Proc #1
Finishing proc 0
Recv'd message from Proc #0
Sending message to Proc #0
Finishing proc 1
```

Figure 3.1: Results of the execution of hellompi.c for 2 processes

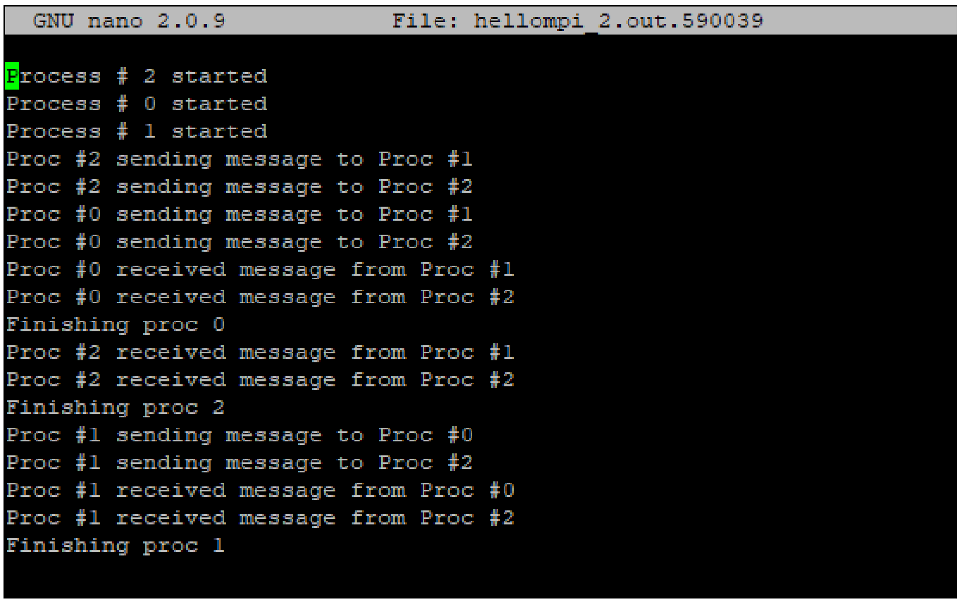
We continue to create an extension of this work using three processes so that each of these sends messages to the rest.

4. **Provide the code of the variant of hellompi.c as described above.**
   The result is analogous to the previous case, only this time the three processes communicate with each other. In essence, the code is identical to that provided by the subject but modifying the following flow control statement:

```
if (MyProc == 0) {
 printf("Proc #0 sending message to Proc #1 \n") ;
 MPI_Send(&msg, 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
 printf("Proc #0 sending message to Proc #2 \n") ;
 MPI_Send(&msg, 1, MPI_CHAR, 2, tag, MPI_COMM_WORLD);

 MPI_Recv(&msg_recpt, 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD, &status);
 printf("Proc #0 received message from Proc #1 \n") ;
 MPI_Recv(&msg_recpt, 1, MPI_CHAR, 2, tag, MPI_COMM_WORLD, &status);
 printf("Proc #0 received message from Proc #2 \n") ;
  }
else if (MyProc == 1) {
 printf("Proc #1 sending message to Proc #0 \n") ;
 MPI_Send(&msg, 1, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
 printf("Proc #1 sending message to Proc #2 \n") ;
 MPI_Send(&msg, 1, MPI_CHAR, 2, tag, MPI_COMM_WORLD);
```

```
  MPI_Recv(&msg_recpt, 1, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf("Proc #1 received message from Proc #0 \n") ;
  MPI_Recv(&msg_recpt, 1, MPI_CHAR, 2, tag, MPI_COMM_WORLD, &status);
  printf("Proc #1 received message from Proc #2 \n") ;
 }
 else {
  printf("Proc #2 sending message to Proc #1 \n") ;
  MPI_Send(&msg, 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
  printf("Proc #2 sending message to Proc #2 \n") ;
  MPI_Send(&msg, 1, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
  MPI_Recv(&msg_recpt, 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD, &status);
  printf("Proc #2 received message from Proc #1 \n") ;
  MPI_Recv(&msg_recpt, 1, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf("Proc #2 received message from Proc #2 \n") ;
 }
```

Executing this modification we obtain the following result:



```
  GNU nano 2.0.9            File: hellompi_2.out.590039

Process # 2 started
Process # 0 started
Process # 1 started
Proc #2 sending message to Proc #1
Proc #2 sending message to Proc #2
Proc #0 sending message to Proc #1
Proc #0 sending message to Proc #2
Proc #0 received message from Proc #1
Proc #0 received message from Proc #2
Finishing proc 0
Proc #2 received message from Proc #1
Proc #2 received message from Proc #2
Finishing proc 2
Proc #1 sending message to Proc #0
Proc #1 sending message to Proc #2
Proc #1 received message from Proc #0
Proc #1 received message from Proc #2
Finishing proc 1
```

Figure 3.2: *Results of the execution of hellompi_2.c for 3 processes*

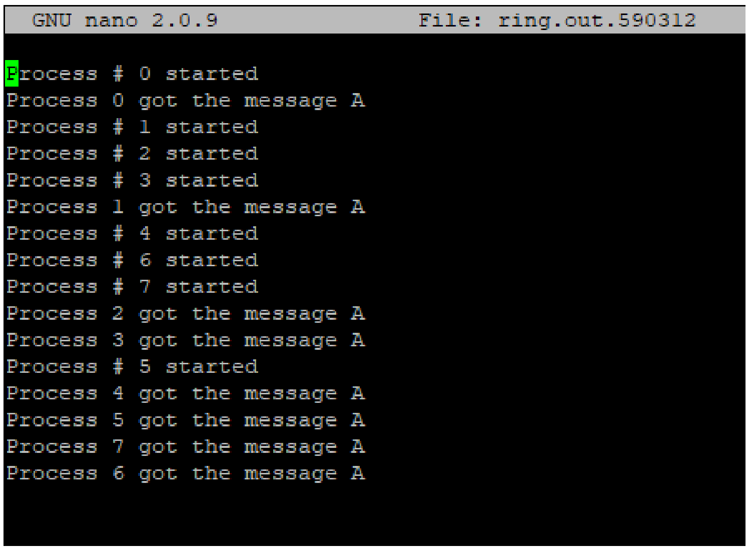
4 Order of the write messages in the standard output

We continue the deliverable by studying the ordering of the messages shown in the standard output. To do this, we create a short code using MPI that displays the messages following a ring, that is, each process sends a message to the next higher ranked process. This will allow us to see how the messages are distributed according to the rank.

5. **Provide the code of your ring program using MPI.**
   The code used is the following:

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

int main( argc, argv )
int argc;
char **argv;
{
  int MyProc, size, tag=1;
  char msg='A';
  MPI_Status status;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &MyProc );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  printf("Process # %d started \n", MyProc);

  if (MyProc == 0) {
   MPI_Send( &msg, 1, MPI_CHAR, MyProc + 1, tag, MPI_COMM_WORLD );
  }
  else {
   MPI_Recv( &msg, 1, MPI_CHAR, MyProc - 1, tag, MPI_COMM_WORLD,
       &status );
   if (MyProc < size - 1)
   MPI_Send( &msg, 1, MPI_CHAR, MyProc + 1, tag, MPI_COMM_WORLD );
  }
  printf( "Process %d got the message %c\n", MyProc, msg );

  MPI_Finalize( );
  return 0;
}
```

6. **What is the order of the messages in the output? Why?**
   Executing the above code we get the following messages:



```
  GNU nano 2.0.9                 File: ring.out.590312

Process # 0 started
Process 0 got the message A
Process # 1 started
Process # 2 started
Process # 3 started
Process 1 got the message A
Process # 4 started
Process # 6 started
Process # 7 started
Process 2 got the message A
Process 3 got the message A
Process # 5 started
Process 4 got the message A
Process 5 got the message A
Process 7 got the message A
Process 6 got the message A
```

Figure 4.1: *Results of the execution of ring.c for 8 processes*

we see how the first process is executed and receives the message immediately, since being the master node this is specified by default. On the other hand, the rest of the processes are started more or less in order (process 5 is delayed a little with respect to the rest) and receive the messages after the start of the processes, but keeping the same order as the initialization of the processes.

5 Basic MPI programming

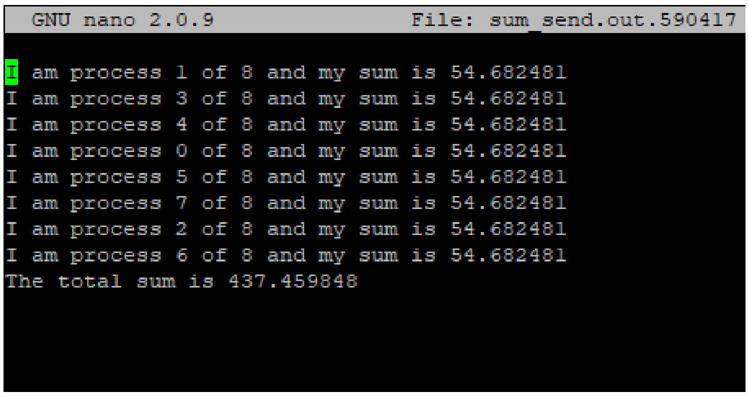
We continue the practice by creating parallel codes from sequential codes using MPI. We will start by writing a parallel code that generates a set of random numbers locally in each process, adds them up locally and then joins them together in a master node to give the total result. There are several ways to do this, but we will focus on the next three:

- Using the command MPI_Send
- Using the command MPI_Gather
- Using the command MPI_Reduce

7. **Provide three simple parallel programs to sum a set of random numbers based on the variants described above.**
   The codes and results for each of the above variants are as follows
   - Using the command MPI_Send

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

int main( argc, argv )
int argc;
char **argv;
{
  int id, ntasks, i, N=100;
  double r, sum, sum_total;
  double array[N];
  char hostname[256];

  MPI_Status status;
  MPI_Init (&argc, &argv);
  MPI_Comm_size ( MPI_COMM_WORLD, &ntasks );
  MPI_Comm_rank ( MPI_COMM_WORLD, &id );

  /*Create an array with random numbers*/
  if ( id == 0 )
  {
    for (i=0; i<N; i++) {
      array[i] = (double)rand()/RAND_MAX;
    }
  }
  /*Broadcast the array to each process and calculate*/
  MPI_Bcast (array, N, MPI_DOUBLE, 0, MPI_COMM_WORLD );
  sum = 0.0;
  for ( i = 0; i < N; i++ )
  {
    sum += array[i];
    }
    printf("I am process %d of %d and my sum is %lf\n", id,
        ntasks, sum);

    /*
    Send the sum back to the master process.
    */
    if ( id != 0 )
    {
      MPI_Send ( &sum, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD );
    }
    else
    {
      sum_total = sum;
      for ( i = 1; i < ntasks; i++ )
      {
        MPI_Recv ( &sum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 1,
            MPI_COMM_WORLD, &status );
        sum_total += sum;
      }
    }

    if ( id == 0 )
    {
      printf("The total sum is %f", sum_total);
    }
    MPI_Finalize();
    exit(0);
}
```

we obtain the following results:



```
  GNU nano 2.0.9                    File: sum_send.out.590417

I am process 1 of 8 and my sum is 54.682481
I am process 3 of 8 and my sum is 54.682481
I am process 4 of 8 and my sum is 54.682481
I am process 0 of 8 and my sum is 54.682481
I am process 5 of 8 and my sum is 54.682481
I am process 7 of 8 and my sum is 54.682481
I am process 2 of 8 and my sum is 54.682481
I am process 6 of 8 and my sum is 54.682481
The total sum is 437.459848
```

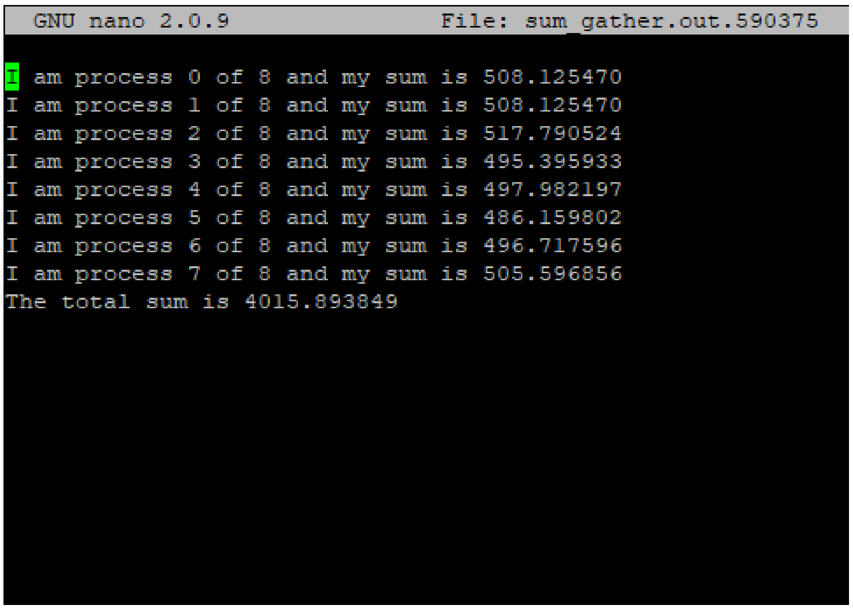
Figure 5.1: Results of the execution of sum_send.c for 8 processes

- Using the command MPI_Gather

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

```
int main(int argc, char *argv[]) {
double r, sum, sum_total=0.0;
int rank, ntasks, i, n;
double *sums = NULL;

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

n = 1000;
srand(rank);

for (i=0; i<n; i++) {
r = (double)rand()/RAND_MAX; /* Random number between 0 and 1 */
sum += r;
}

printf("I am process %d of %d and my sum is %lf\n", rank,
    ntasks, sum);

if (!rank) {
sums = (double *)malloc(sizeof(double) * ntasks);
}

MPI_Gather(&sum, 1, MPI_DOUBLE, sums, 1, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);

if (rank == 0) {
for (i = 0; i < ntasks; i++) {
sum_total += sums[i];
}
printf("The total sum is %f", sum_total);
}
MPI_Finalize();
exit(0);
}
```

with the results



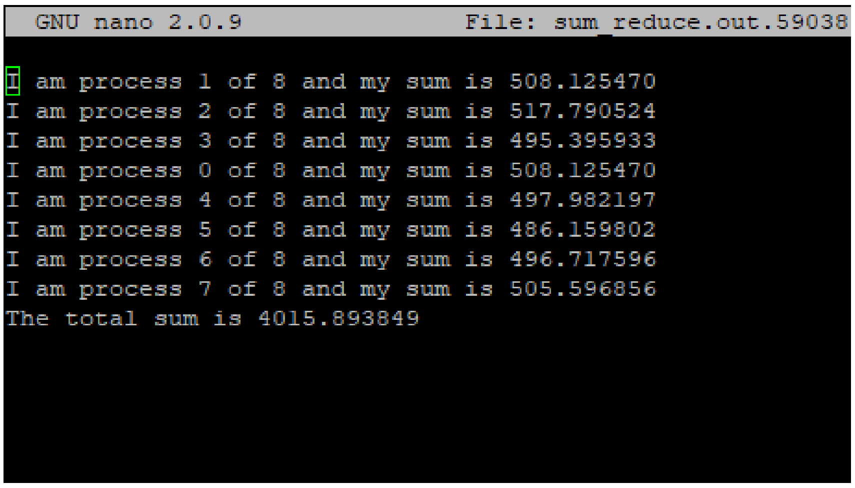
```
 GNU nano 2.0.9                File: sum_gather.out.590375

I am process 0 of 8 and my sum is 508.125470
I am process 1 of 8 and my sum is 508.125470
I am process 2 of 8 and my sum is 517.790524
I am process 3 of 8 and my sum is 495.395933
I am process 4 of 8 and my sum is 497.982197
I am process 5 of 8 and my sum is 486.159802
I am process 6 of 8 and my sum is 496.717596
I am process 7 of 8 and my sum is 505.596856
The total sum is 4015.893849
```

Figure 5.2: *Results of the execution of sum_gather.c for 8 processes*

- Using the command MPI_Reduce

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

int main(int argc, char *argv[]) {
  double r, sum=0.0;
  double sum_total;
  int id, ntasks, i, n;
  char hostname[256];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  gethostname(hostname,255);

  n = 1000;
  srand(id);

  for (i=0; i<n; i++) {
    r = (double)rand()/RAND_MAX; /* Random number between 0 and
        1 */
    sum += r;
  }
  printf("I am process %d of %d and my sum is %lf\n", id,
      ntasks, sum);

  /* Sum the values in all processes with a reduction operation
      */
  MPI_Reduce(&sum, &sum_total, 1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD);

    if (id == 0) {
      printf("The total sum is %f", sum_total);
    }

    MPI_Finalize();
    exit(0);
}
```

The result of this code is the partial sum with value 4015.89:



```
  GNU nano 2.0.9                    File: sum_reduce.out.59038

I am process 1 of 8 and my sum is 508.125470
I am process 2 of 8 and my sum is 517.790524
I am process 3 of 8 and my sum is 495.395933
I am process 0 of 8 and my sum is 508.125470
I am process 4 of 8 and my sum is 497.982197
I am process 5 of 8 and my sum is 486.159802
I am process 6 of 8 and my sum is 496.717596
I am process 7 of 8 and my sum is 505.596856
The total sum is 4015.893849
```

Figure 5.3: *Results of the execution of sum_reduce.c for 8 processes*

8. **Provide a parallel version of p8.c using MPI. Explain your design decisions.**
   We continue the practice by paralleling a code provided by the course, p8.c, which implements a numerical integration method using the rule of the trapezoid. Essentially, the trapezoid rule is implemented by the following function:

```
double f(double x)
{
return(4.0/(1.0+x*x));
}

double fragment(double a, double b, double num_fragments, double h)
{
  double est, x;
  int i;

  est = (f(a) + f(b))/2.0;
  for (i=1; i<=num_fragments-1; i++){
  x = a + i*h;
  est += f(x);
  }
  est = est*h;

  return est;
}
```

while the rest of the code follows a standard sequential structure that can be parallelized in the same way as the codes in the previous section. For simplicity, we rely on the code proposed for the MPI_Reduce section. In this way, it is very easy to parallelize the previous sequential code if we divide the integration region (A, B) in n equal regions, where n is the number of processes assigned to the execution by MPI. In this way, we have the following integration regions:

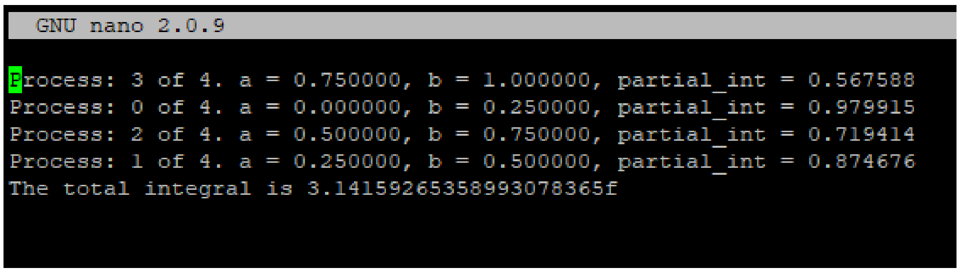
$$a_i = (B - A) \cdot \frac{id_{proc}}{n} \qquad id_{proc} = 0, 1, \ldots, n$$

$$b_i = (B - A) \cdot \frac{id_{proc} + 1}{n}$$

Applying this idea to the code, we get the following:

```
int main(int argc, char **argv) {

  int id, ntasks, i;
  double N=1000000000.0, A = 0.0, B=1.0;
  double n, a, b, h;
  double result;
  double result_total = 0.0;
  char hostname[256];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  gethostname(hostname,255);

  #Define the parameters for each region of integration (process)
  n=N/ntasks, a=(B-A)*((double)id/(double)ntasks),
      b=(B-A)*(((double)id+1)/(double)ntasks);

  h = (b-a)/n;
  result = fragment(a, b, n, h);

  printf("Process: %d of %d. a = %lf, b = %lf, partial_int = %lf\n"
      , id, ntasks, a, b, result);

  MPI_Reduce(&result, &result_total, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);

  if (id == 0) {
    printf("The total integral is %.20ff\n", result_total);
  }

  MPI_Finalize();

  exit(0);
}
```

The result is still correct, and we see that each node supports a part of the program execution:



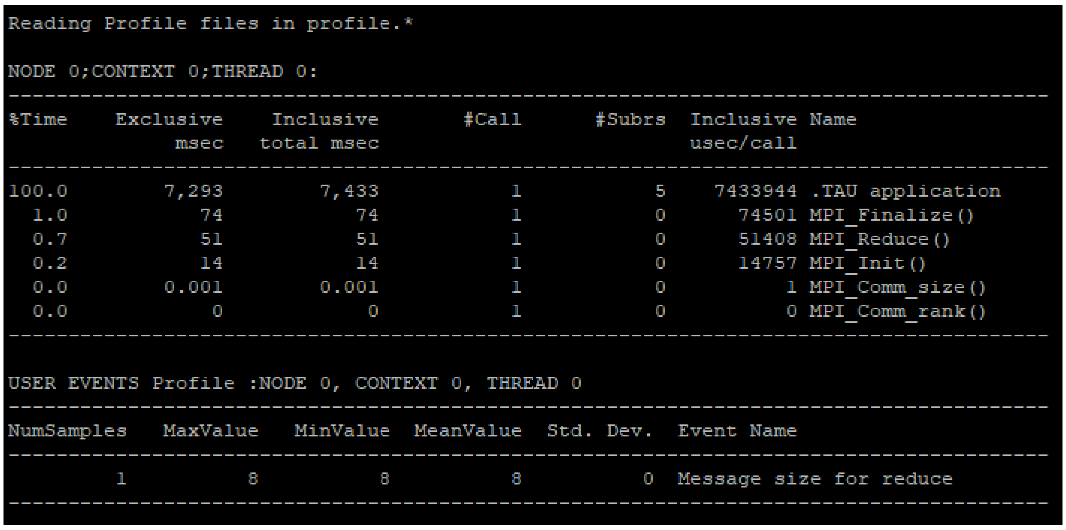
```
  GNU nano 2.0.9

Process: 3 of 4. a = 0.750000, b = 1.000000, partial_int = 0.567588
Process: 0 of 4. a = 0.000000, b = 0.250000, partial_int = 0.979915
Process: 2 of 4. a = 0.500000, b = 0.750000, partial_int = 0.719414
Process: 1 of 4. a = 0.250000, b = 0.500000, partial_int = 0.874676
The total integral is 3.14159265358993078365f
```

Figure 5.4: *Results of the execution of p8.c for 8 processes*

6 | Performance evaluation tools (TAU/EXTRAE/PARAVER)

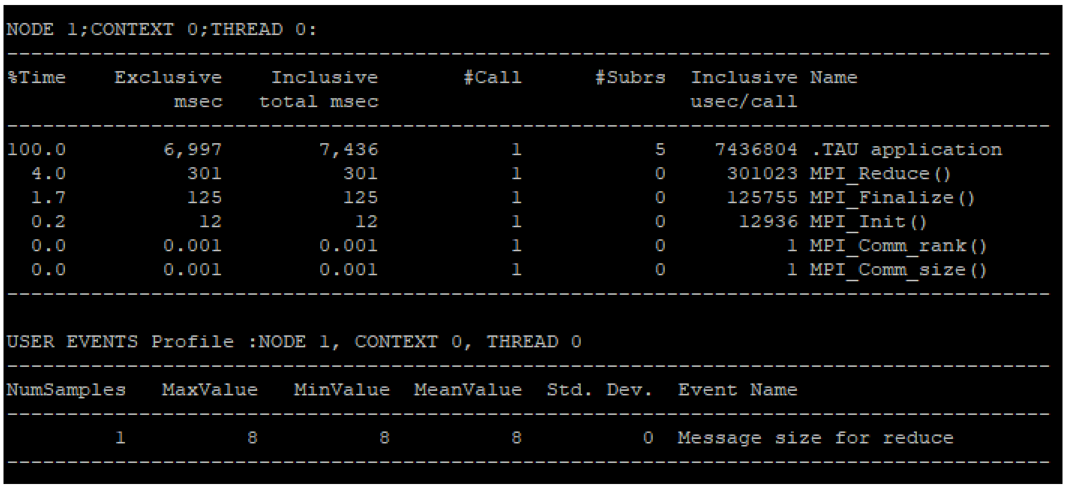
We end this practice by performing a performance analysis and benchmarking of the code previously parallelized p8.c. To do this, we use the performance analysis tools TAU, Extrae and Paraver.

9. **Perform a brief evaluation of your parallel implementation of p8.c using TAU and jumpshot.**
   TAU is a tool that allows observing in a simple way the work done by each node in the parallel execution. Following the instructions of the course we can execute the tool and together with pprof obtain the following reports:



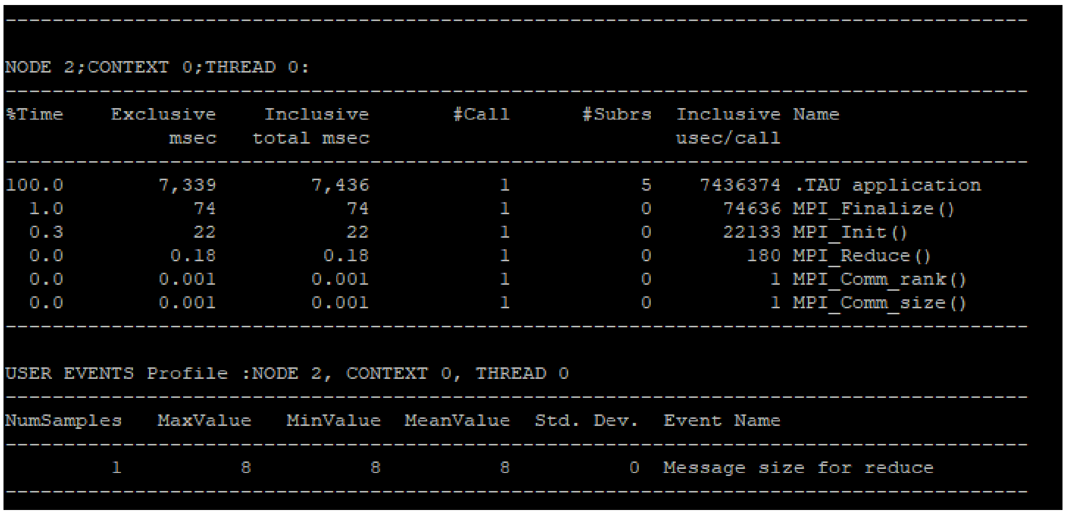
```
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        7,293        7,433           1           5    7433944 .TAU application
  1.0           74           74           1           0      74501 MPI_Finalize()
  0.7           51           51           1           0      51408 MPI_Reduce()
  0.2           14           14           1           0      14757 MPI_Init()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
  0.0            0            0           1           0          0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
        1          8          8          8          0  Message size for reduce
---------------------------------------------------------------------------------------
```

Figure 6.1: *Report for the node 0 using TAU and pprof*



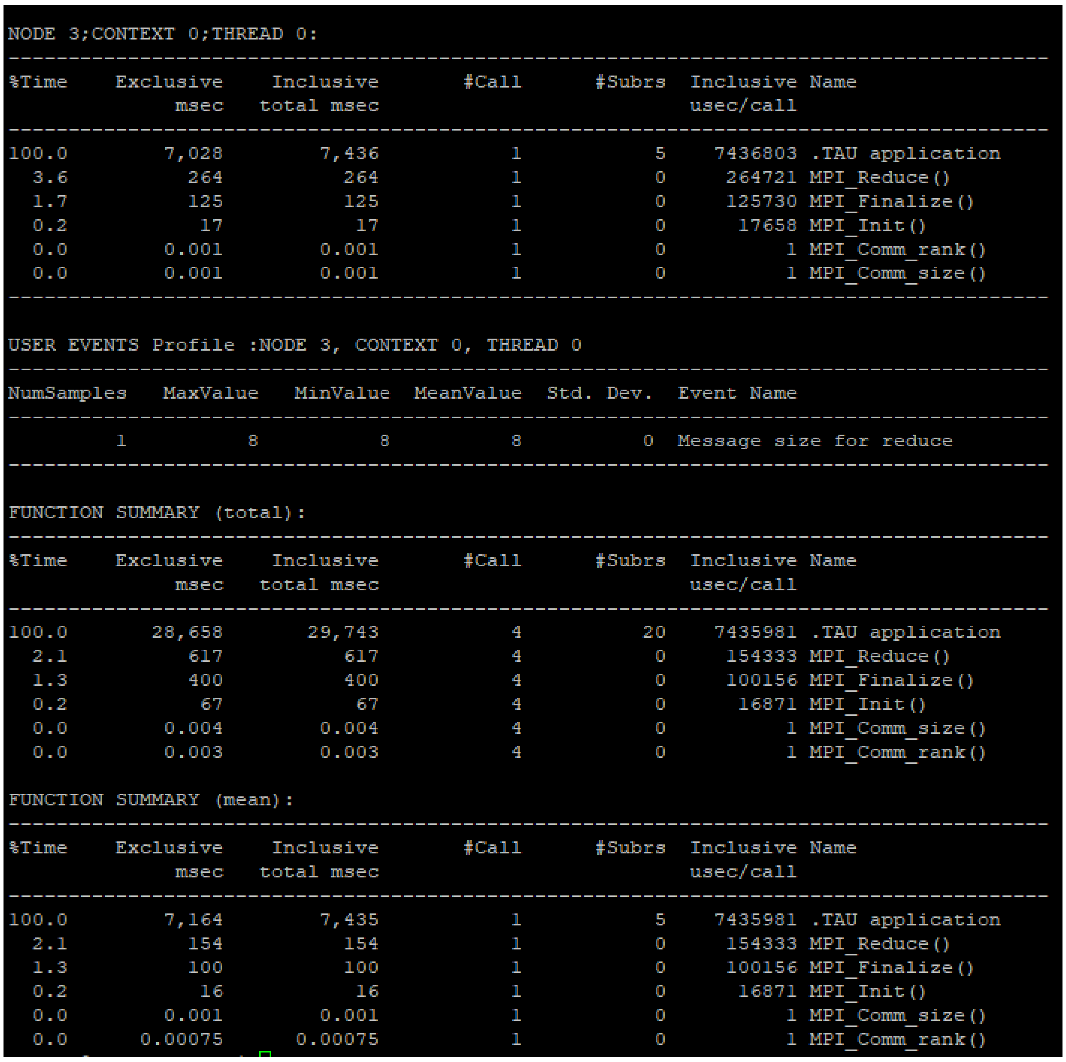
```
NODE 1;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        6,997        7,436           1           5    7436804 .TAU application
  4.0          301          301           1           0     301023 MPI_Reduce()
  1.7          125          125           1           0     125755 MPI_Finalize()
  0.2           12           12           1           0      12936 MPI_Init()
  0.0        0.001        0.001           1           0          1 MPI_Comm_rank()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 1, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
         1          8          8          8          0  Message size for reduce
---------------------------------------------------------------------------------------
```

Figure 6.2: Report for the node 1 using TAU and pprof



```
---------------------------------------------------------------------------------------

NODE 2;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        7,339        7,436           1           5    7436374 .TAU application
  1.0           74           74           1           0      74636 MPI_Finalize()
  0.3           22           22           1           0      22133 MPI_Init()
  0.0         0.18         0.18           1           0        180 MPI_Reduce()
  0.0        0.001        0.001           1           0          1 MPI_Comm_rank()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 2, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
         1          8          8          8          0  Message size for reduce
---------------------------------------------------------------------------------------
```

Figure 6.3: Report for the node 2 using TAU and pprof



```
NODE 3;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        7,028        7,436           1           5    7436803 .TAU application
  3.6          264          264           1           0     264721 MPI_Reduce()
  1.7          125          125           1           0     125730 MPI_Finalize()
  0.2           17           17           1           0      17658 MPI_Init()
  0.0        0.001        0.001           1           0          1 MPI_Comm_rank()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 3, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
         1          8          8          8          0  Message size for reduce
---------------------------------------------------------------------------------------

FUNCTION SUMMARY (total):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0       28,658       29,743           4          20    7435981 .TAU application
  2.1          617          617           4           0     154333 MPI_Reduce()
  1.3          400          400           4           0     100156 MPI_Finalize()
  0.2           67           67           4           0      16871 MPI_Init()
  0.0        0.004        0.004           4           0          1 MPI_Comm_size()
  0.0        0.003        0.003           4           0          1 MPI_Comm_rank()

FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        7,164        7,435           1           5    7435981 .TAU application
  2.1          154          154           1           0     154333 MPI_Reduce()
  1.3          100          100           1           0     100156 MPI_Finalize()
  0.2           16           16           1           0      16871 MPI_Init()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
  0.0      0.00075      0.00075           1           0          1 MPI_Comm_rank()
```

Figure 6.4: *Report for the node 3 using TAU and pprof*

we see that for the four nodes used, the bulk of the execution time is used for the application itself (the calculation of the integral in each node), with execution times around 7.000 milliseconds per node and an implication of between 95 and 99% of the execution time. In turn, the MPI processes (Finalize, Reduce, Init, etc.) take between 100 and 1,000 milliseconds per node and imply between 1 and 5% of the execution time.

On the other hand, the size of each message between nodes always takes a value of 8 bytes, since it is a single element of type *double*.

10. **Perform a brief comparative study of the NPB benchmarks introduced above using Extrae/Paraver.**
On the other hand, we will study the performance with respect to the NPBs (NAS Parallel Benchmarks) using the Extrae tool, which is a tool of the Barcelona Supercomputing Center to create traces.

We create the traces by importing and executing the extract program using the commands:

```
#Import Extrae
export EXTRAE_HOME=/export/apps/extrae
cp /export/apps/extrae/share/example/MPI/extrae.xml .
source /export/apps/extrae/etc/extrae.sh
export EXTRAE_CONFIG_FILE=YOUR_DIRECTORY/extrae.xml
export LD_PRELOAD=/export/apps/extrae/lib/libmpitrace.so

#Execute p8.c
mpirun -np 16 ./p8
```

The result is a set of files of types .pcf, .prv and .row. Once we have them, we import them into the local machine so that we can examine them with Paraver. This way, we can examine the trace file and get the following graphics:

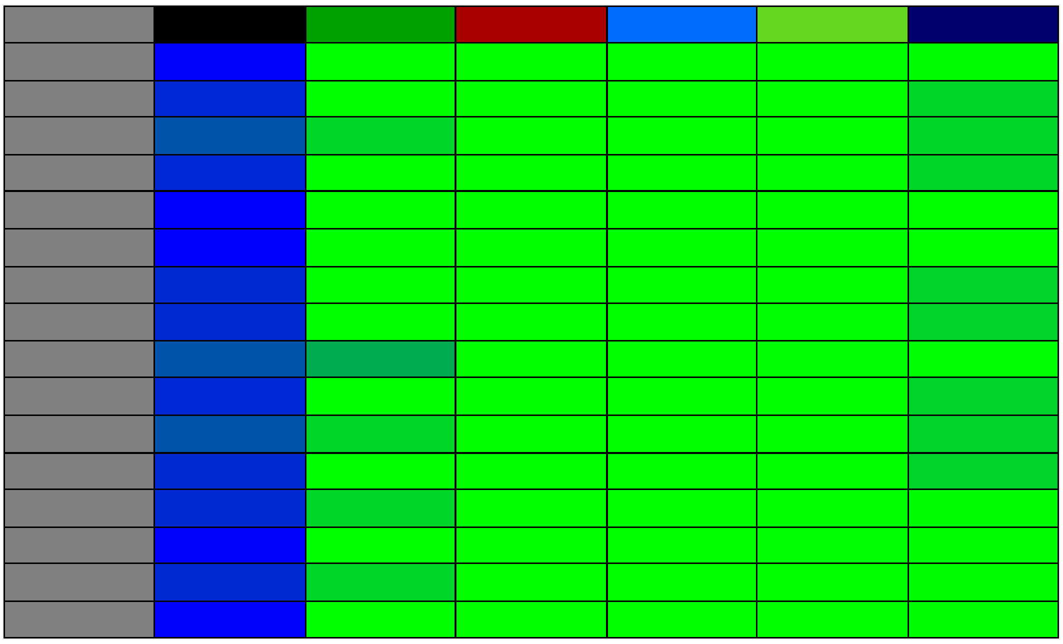


Figure 6.5: *Execution of the mpi_stats.cfg program in Paraver*

This program allows us to display the code execution times for each of the processes in a much more visual way than TAU.

Each of the rows of the graph represents a process while each of the columns represents a functionality of the code (MPI functionalities and the own application). The first column, in black, shows the parts of the code that are not of MPI functionality (the application itself) and the color of each cell represents the execution time. The green color means little execution time while the blue color means more execution time. If we put the mouse over each cell, we can get the percentage of time dedicated to each process. The second row represents the process MPI_Reduce, the third row represents MPI_Comm and so on for the rest. We see that, as we had analyzed previously, the bulk of the execution time is taken by tasks other than MPI (the own calculation of the integral by means of the rule of the trapezium). There are processes, such as 3, 9 or 10 where the integral calculation represents only about 70% of the execution time, and tasks such as MPI_Reduce or MPI_Finalize receive a higher percentage.

We can view this graph in a different way by running the MPI_Call program in the *views* section. The graph is as follows:

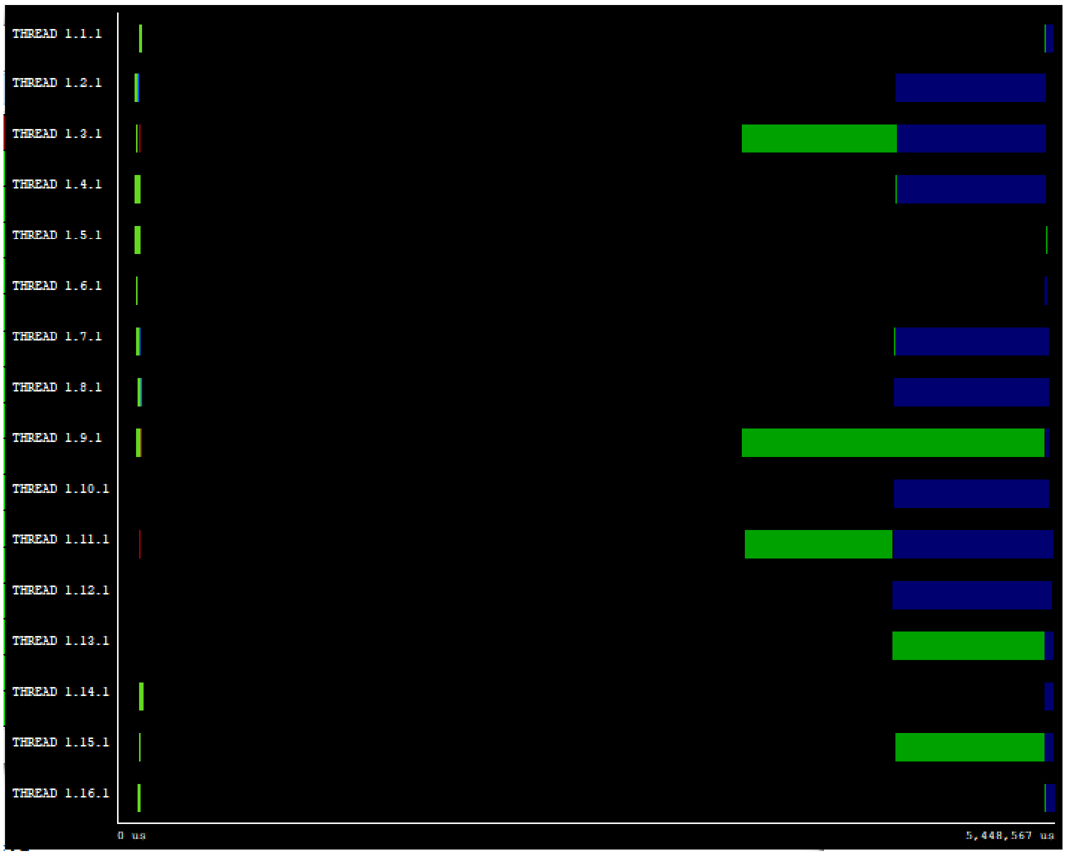


Figure 6.6: *Execution of the MPI_call.cfg program in Paraver*

This program produces the above graphic spread over time. The first coloured lines on the left of the graph represent the MPI initialisation processes (initialisation of the parallel region, obtaining the range, etc.) while the bands at the end represent MPI_Reduce (green) and MPI_Finish (blue) respectively. The black intermediate region represents the execution of the application itself (the calculation of the integral). As we have seen before, the integral calculation represents the majority of the execution time, while the influence of the MPI processes is dominated by the reduction and the finalization. It is curious to observe how there are processes where there is very little execution time of MPI processes, and other processes where it can reach up to 30%. One possible reason for this is that if the region of an integration process is more computationally demanding, the calculation execution time dominates over the MPI execution time, and vice versa if the integration region is easier to calculate.

Finally we can check which processes have suffered a higher computational load in MPI with the program uf_fracion_of_MPI.cfg:

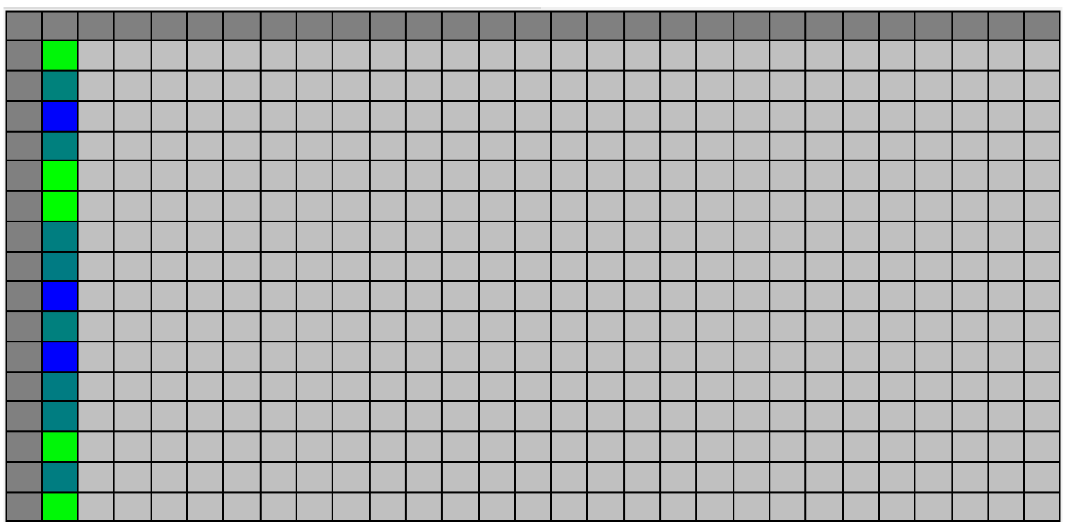


Figure 6.7: *Execution of the uf_fracion_of_MPI.cfg program in Paraver*

We see how processes 3, 9 and 11 have suffered a higher load in MPI with a percentage of 33% of the total computational load of the process.