HPC - Exercise 3

Eduardo Gonnelli

03 July 2019

1 Introduction

The aim of this report is to provide the π approximation through the Midpoint method and to introduce a brief notion about Message Passing Interface (MPI). The final result of π approximation was performed employing the primitive *MPI_REDUCE* global communication routine. The procedure is to reduce the final result in the last process called as *size - 1)* and, then, the final result was printed by process 0. The *MPI_TAG* used was 101. The MPI execution time was compared with the OpenMP obtained on the exercise 01. In addition, the MPI was scaled up to 2 nodes of Ulysses. The speedup and efficiency were plotted and compared with the ideal speedup curve.

2 Message Passing Interface (MPI)

The MPI interface is the dominant programming interface for parallel algorithms with distributed memory in the HPC community. In MPI there is some standardization of many global routines of communication and many primitives to perform global calculations. The MPI commands can be used with the most common (sequential) programming languages like C, C++, Java, Fortran, Python, etc. In this work the C++ programming language was chosen.

In MPI, the *MPI_Reduce* allows one to perform a global calculation by aggregating (reducing) the values of a variable using a commutative binary operator. It performs a reduction of data from each process onto the specified root process. Typical such examples are the cumulative sum (*MPI_SUM*) in which sum up entries from all of the processes in a communicator, or the cumulative product (*MPI_PROD*). MPI provides a number of additional operations, that can be used with any of the collective computation routines. For more details about MPI see the references section.

The Figure 1 represents the *MPI_Reduce* code with the cumulative *MPI_SUM*.

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
                   MPI_Op op, int root, MPI_Comm comm);
```

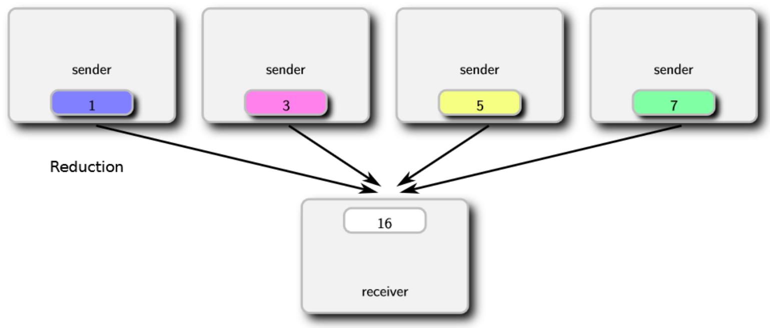
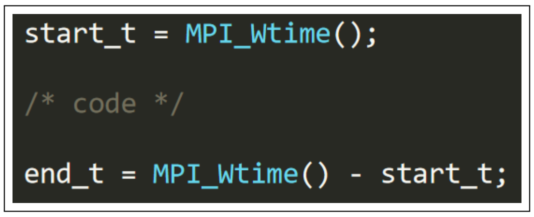


Figure 1: *MPI_Reduce* example - Code and process illustration of the cumulative *MPI_SUM*.

The MPI provides a simple routine that can be used to time programs or sections of programs. The *MPI_Wtime()* returns a double-precision floating-point number that is the time in seconds since some arbitrary point of time in the past. The point is guaranteed not to change during the lifetime of a process. Thus, a time interval can be measured by calling this routine at the beginning and end of a program segment and subtracting the values returned.

The Figure 2 shows the piece o code for measuring the execution time.



```
start_t = MPI_Wtime();

/* code */

end_t = MPI_Wtime() - start_t;
```

Figure 2: Application of *MPI_Wtime()* to measure the execution time.

3 Results

In this section the comparison of execution time between MPI and OpenMP is presented. The speedup was calculated for both cases and compared with the ideal speedup. The efficiency is presented on the Table 1. The approximated value of π was 3.14159 for all processes of MPI. The Figure 3 shows the execution time comparison between both methods. It can be seen that the MPI implementation for the same size problem achieved the lowest time. It was possible to scaled up to 2 nodes of Ulysses and executed the code with 40 processes. In the Figure 4 the speedup comparison is shown. It is possible to notice that the speedup up to 20 processes/threads is very similar between MPI and OpenMP. However, for 24 MPI processes (*mpirun -np 24 ./reduce_ex03.o*) the MPI speedup is very close to the ideal.

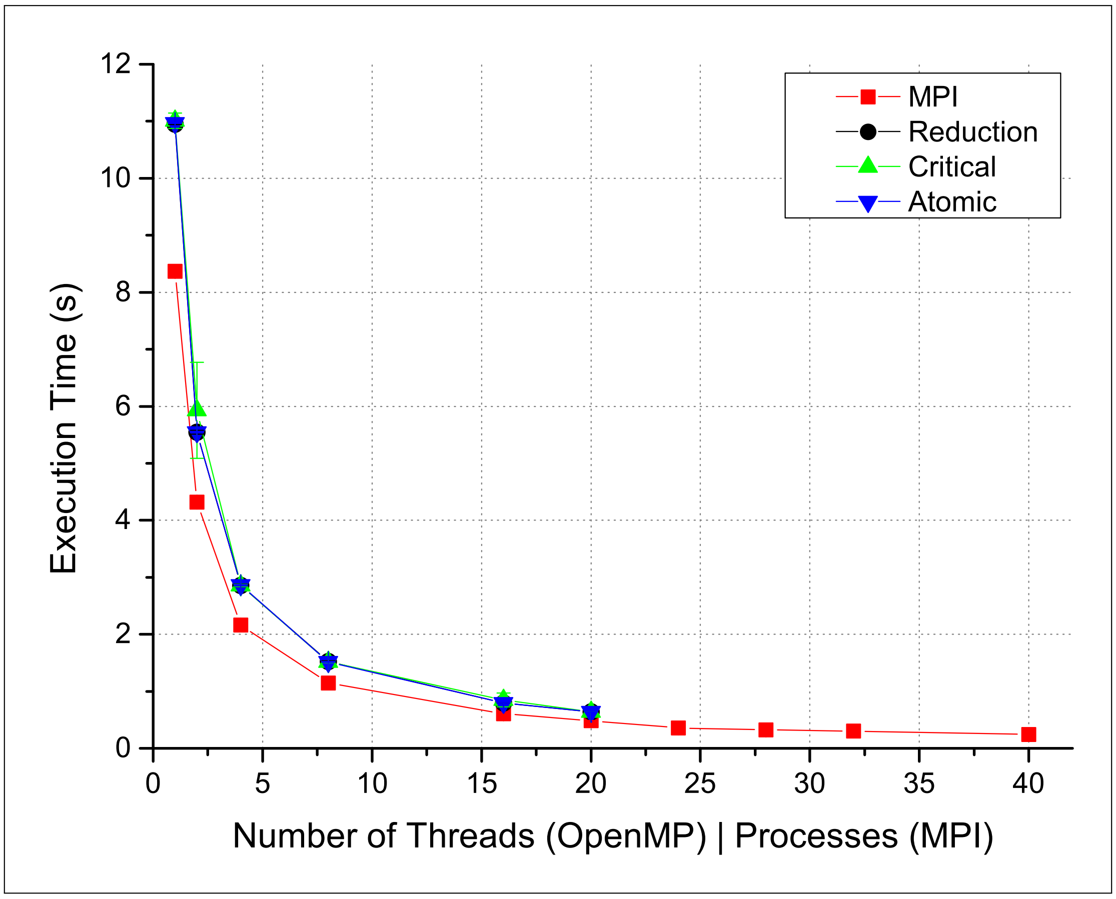


Figure 3: Execution time comparison between the OpenMP and MPI

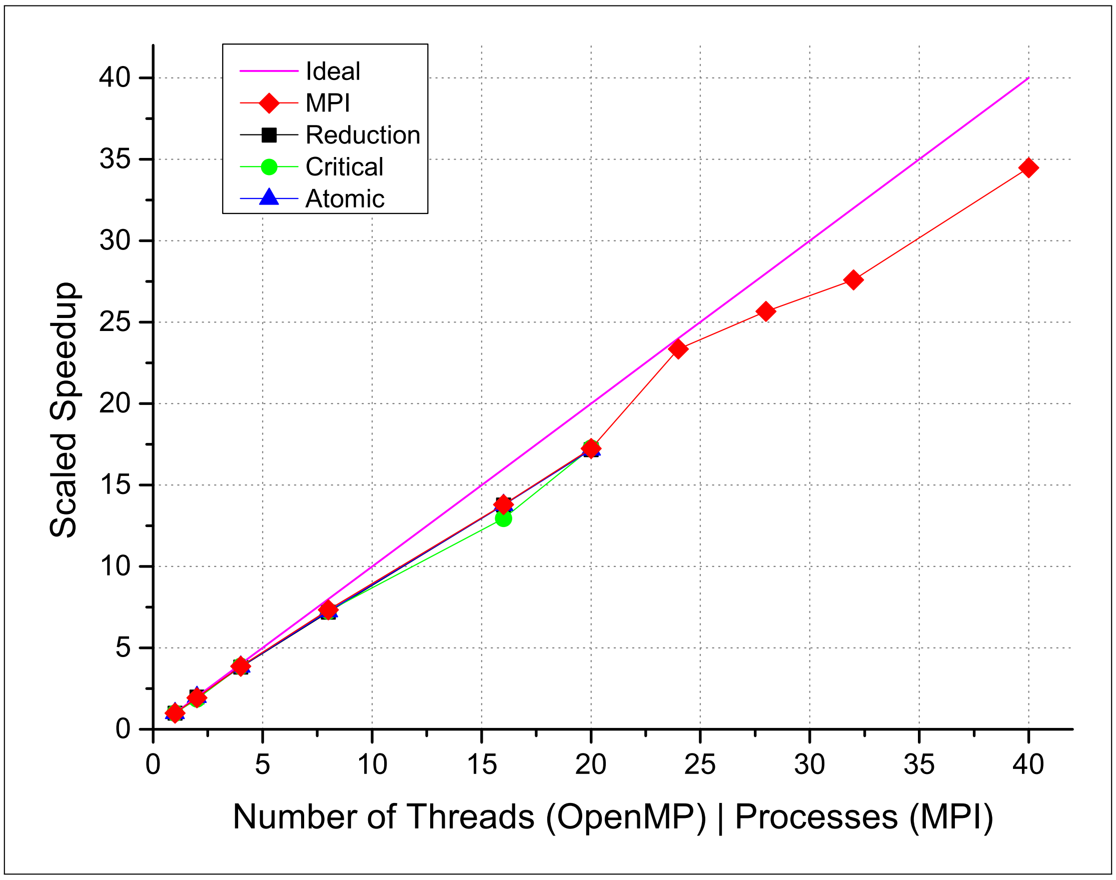


Figure 4: Speedup comparison between the OpenMP and MPI

Table 1: Efficiency Values.

N. of Processes (#)	Efficiency (%) MPI
1	100.00
2	96.94
4	96.82
8	91.60
16	86.19
20	86.19
24	97.27
28	91.66
32	86.19
40	86.17

References

[1] Frank Nielsen. *Introduction to HPC with MPI for Data Science*. Springer, Switzerland, 2016.

[2] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, 2014.

[3] Colfax International: *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors* `https://colfaxresearch.com/`

Appendices

This section contains the codes developed for this work. The compile command for MPI was *mpicc reduce_ex03.cc -o reduce_ex03.o*.

A Codes

Listing 1: Job Script

```bash
#!/bin/bash
#PBS -l nodes=2:ppn=20
#PBS -l walltime=02:30:00
#PBS -q regular

cd /home/egonnell/igirotto/ex03

#load the modules

module load openmpi/1.8.3/intel/14.0

for j in {1..3..1};
do for i in 1 2 4 8 16 20 24 28 32 40;
do mpirun -np $i ./reduce_ex03.o >> time.txt;
sleep 3;
done
done
```

Listing 2: MPI C++ Code

```cpp
#include <iostream>
#include <mpi.h>

double function(const double x);

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

   const int N = 2147483647;
   const double a = 0.0;
   const double b = 1.0;
   const double h = (b-a)/N; // implementation of the midpoint method

   int rank, size;

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

   const unsigned int local_N = N/size;
   double local_pi = 0.0;
   double start_t = 0.0;
   double end_t = 0.0;

//Reference points for computation:
   const double start_computation = rank * local_N;
   const double end_computation = start_computation  + local_N;

start_t = MPI_Wtime();

for (int i = start_computation ; i < end_computation ; ++i)
      {
        double x_i = (i + 0.5)* h;
        local_pi += h*function(x_i);
        //The result will be PI/4, that's why we multiply by 4
      }
double total_pi = 4*local_pi;
  //variable for final result:
   double global_pi = 0.0;
//Computing the final result:
MPI_Reduce(&total_pi, &global_pi, 1, MPI_DOUBLE, MPI_SUM, (size-1), MPI_COMM_WORLD);
end_t = MPI_Wtime() - start_t;

//The last rank will send the message to rank 0:
if ( rank == (size - 1 ))
    {
      MPI_Send(&global_pi, 1, MPI_DOUBLE, 0  , 101, MPI_COMM_WORLD); //101 as TAG


//Rank 0 will recieve the message and print the result
   }
if ( rank == 0 ){
     MPI_Recv(&global_pi, 1, MPI_DOUBLE, (size -1) , 101, MPI_COMM_WORLD, &status);
     std::cout << "This␣is␣the␣Process:␣" << rank << "\n";
     std::cout << "Execution␣time␣=␣" << end_t << "␣s\n";
     std::cout << "Pi␣Approximation␣=␣" << global_pi << "\n";
    }
  MPI_Finalize();
  return 0;
}

double function(const double x)
        {return 1.0/(1.0+(x*x));}
```