You Robin
Houlier Nicolas

TMA 4280: Report Project 1

Link of the repository associated to the project: https://github.com/MrChapelle/TMA4280LABS

Important fact:

In each folder of the Git repository, there is a ReadMe file which details the executions to do to launch the programs linked to the different questions.

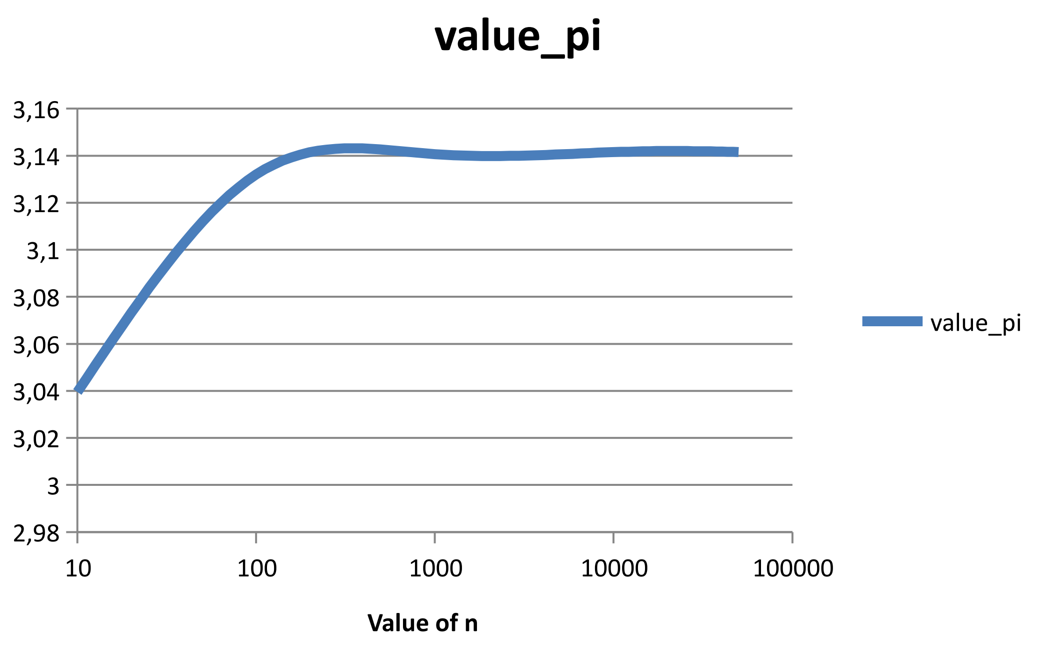
The aim of this project was an introduction to Supercomputing and parallel programming using C programming and MPI/openMP technologies to sum the elements of an array and then approximate the value of Pi.
The execution time and error curbs were plot using Python and Excel.
Each member of the group was responsible of one part of the project (Zeta/Mach). However the final work is a collective work.
Please do not pay to much attention to the graphs which can not be clear because of the extremely small results. The error and time execution files have been created to report these results.

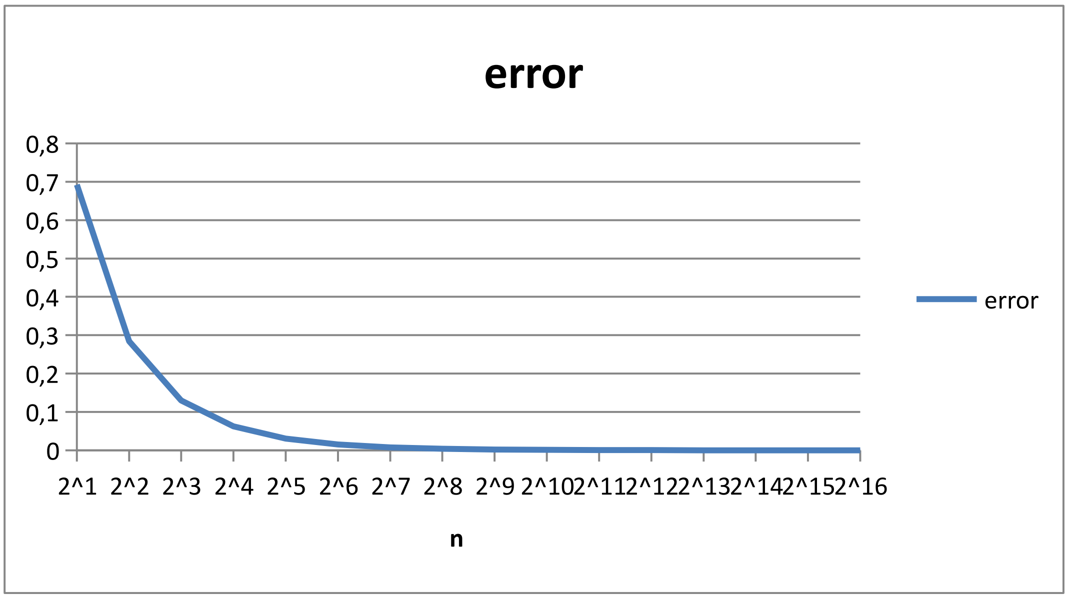
Answers to questions:

Question 1:

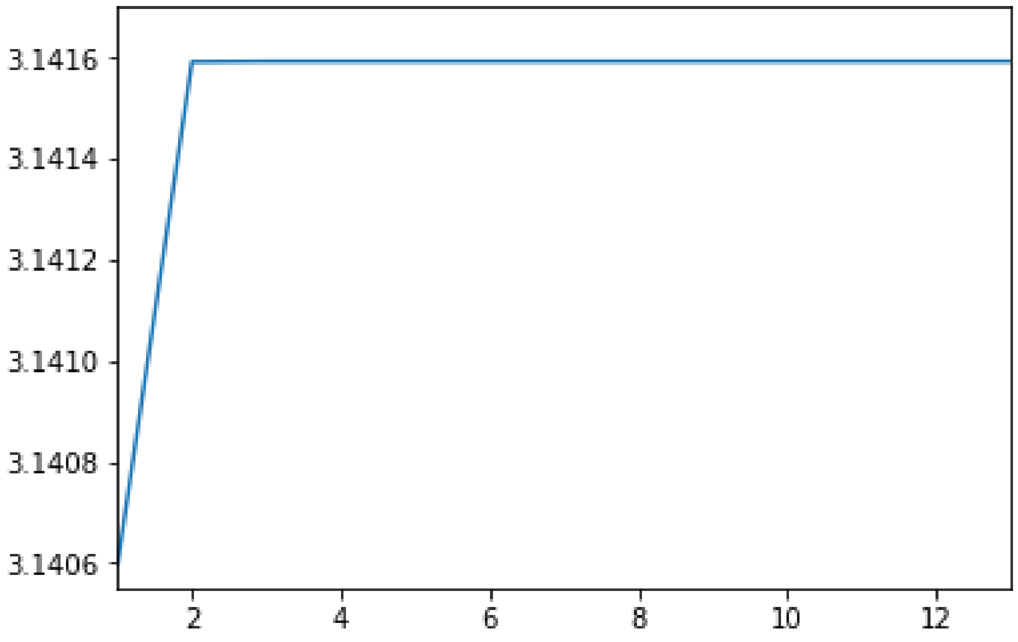
Please refer to the Git repository (zeta0/mach0).
We can plot the different values of Pi in function of n:

Zeta method:





<u>Mach method:</u>



X-axis is graduated with a log2 ladder: (2**2,2**4…)

We can see that mach converge faster than zeta to the value of pi, we can't distinguish the difference between n = 2**4 and the next values graphically.

Question 2:

A unit test is used to ensure that the specific function is working as expected. A parallel code is longer than a conventional code and it uses more modules, so it's even more important to use tests to assure the good work of a program especially after a modification. The programs linked to this question are contained in the zeta0/mach0 folders. Please read the associated ReadMe files.

Question 3:

When n is too large, our computers can't calculate the result (n>=20).
We can see that the approximation of Pi with mach is faster, however the two solutions give an approximate solution with a precision of 10E(-20).

Question 4:

The problem of this data distribution is that the root process only distribute portions of the array to child processes and compute the final sum but he doesn't calculate a partial sum which is a lack of efficiency. We can improve our code by giving a portion of the array to the root process (master process).

Question 5:

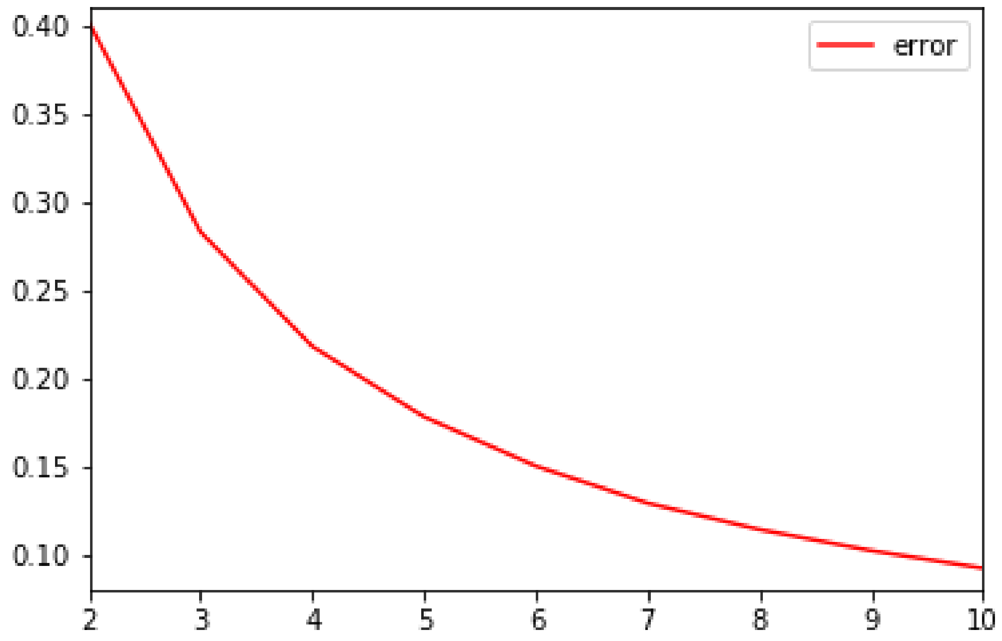
To make the program work using openMPI, we only used basic MPI function:

MPI_Init → to initialize the openMPI mechanism
MPI_Status → get the current status
MPI_Send → send data to processes
MPI_Recv → advertise process that he will receive some data
MPI_Finalize → close openMPI

To calculate the wall time, we used the Time library included by default, we calculate the difference of wall time between the beginning and the end of the program, when the root process has calculated the value of pi.
The program detect if the number of processes is a power of 2 or not.

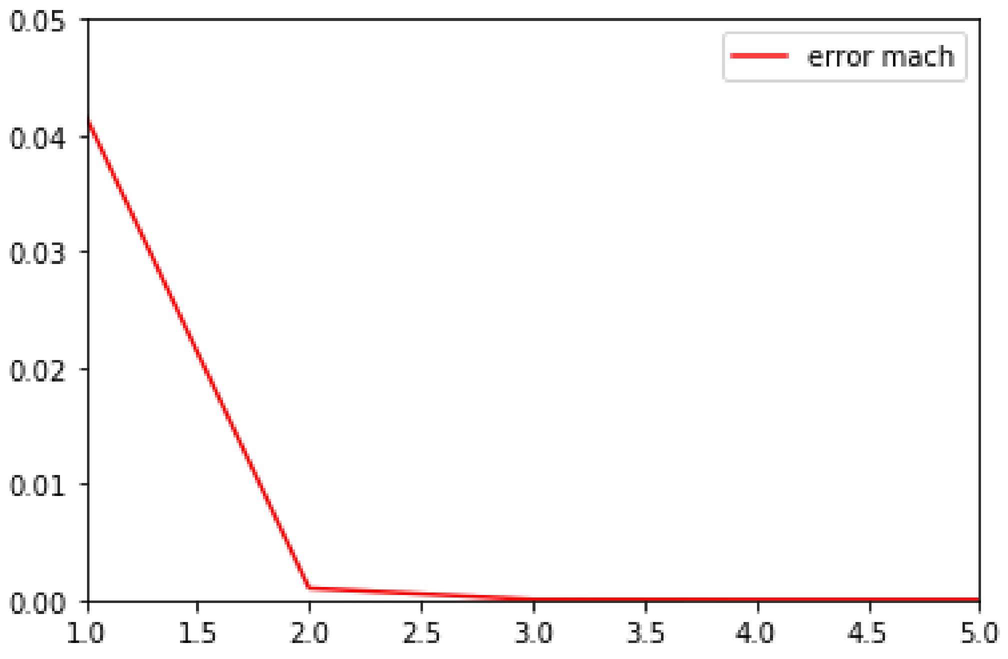
We can plot the value of the error and the timings for each methods:

<u>Zeta method:</u>
This is the error |pi-pin| for n in [2,10]. After that the error diminish more and more slowly, down to 0 at 10E-6



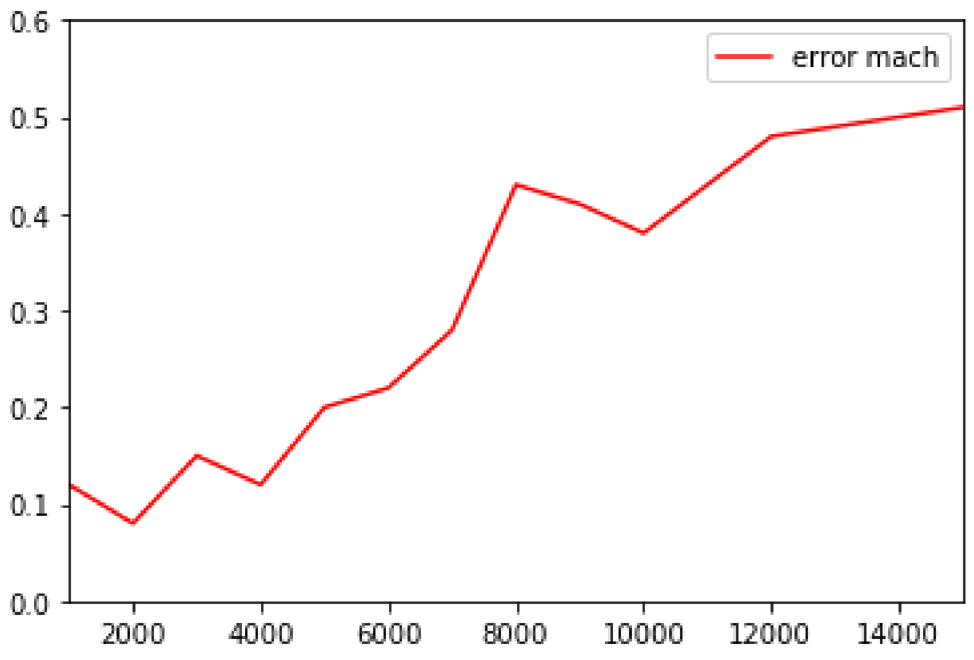
Mach method:

We can plot the error and timings for P = 2: (in function of n)



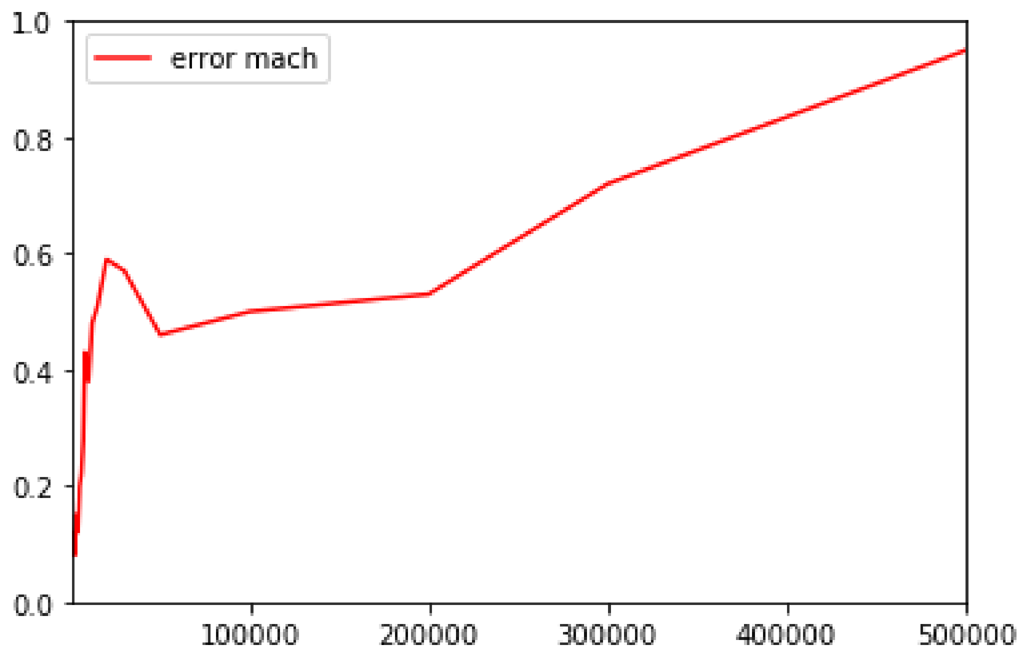
Errors are registered in the files errorp*.txt, for P = 1,2,4,8.

We can also plot timings for P = 2:



Y-axis represent time execution in seconds.

We can also calculate the time execution when n is very big which is better because the time needed by the processes to communicate is less and less important compare to the calculation time.



Question 6:

We can plot on the same graph the errors for P = 2 and P = 8. But the result will be same than in the question 5. A better way to answer the question is to compare the values in the files errorp2.txt and errorp8.txt in the folder mach1.

Error	P = 2	P = 8
N=1	4,17E-002	4,17E-002
N=2	9,95E-004	9,95E-004
N=3	2,83E-005	2,83E-005
N=4	8,81E-007	8,81E-007
N=5	2,88E-008	2,88E-008
N=6	9,74E-010	9,74E-010
N=7	3,37E-011	3,37E-011
N=8	1,19E-012	1,19E-012
N=9	4,30E-014	4,30E-014

For N >10, the error is fixe at about 10e-16.

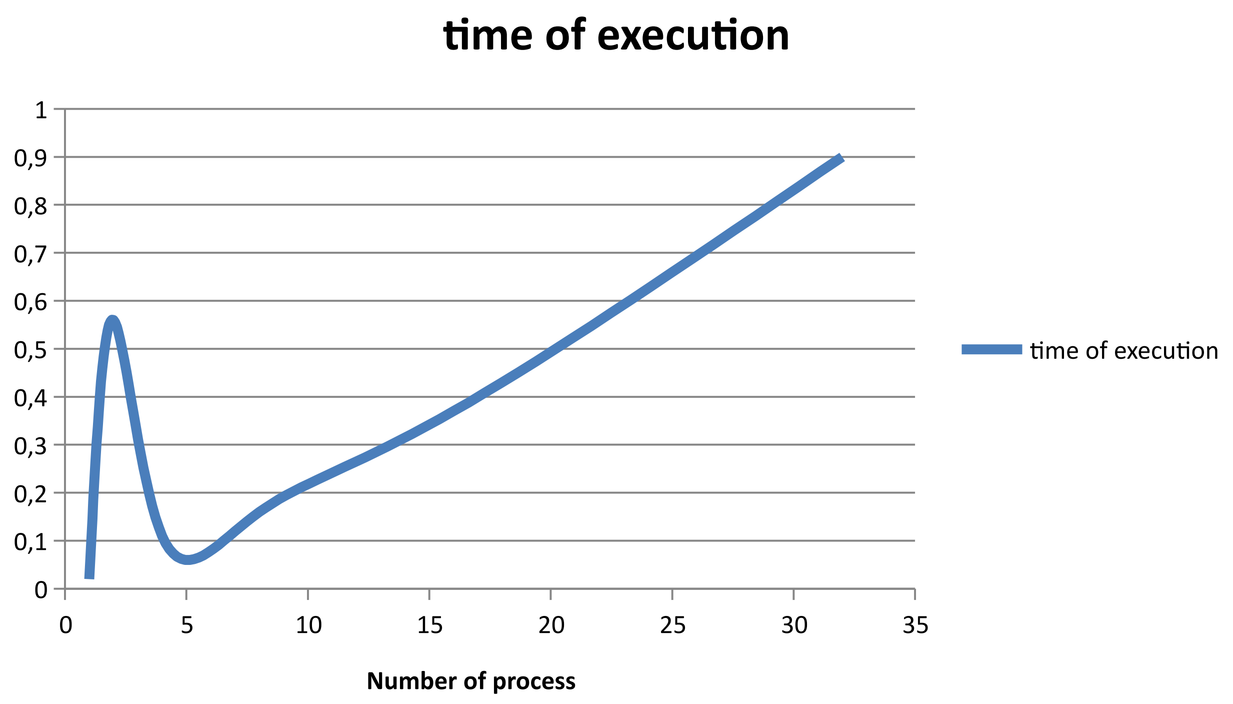
The values are exactly the same. This is the case because of our work repartition to each process. Any case of the array is forgotten and each partial sum can be calculated without floating point representation errors (ex: 1 + 0,00000000000000001 = 1).
With this sharing, the result should be exactly the same for each number of process.

We also have exactly the same errors for the Zeta method.

Question 7:

Please refer to the ReadMe file in the reduc folder.

After a small scaling study, we obtain this graph:



The best time of execution is obtained for 4 process, after this is longer because of the number of inter-process communications due to the increasing number of process.

Question 8:

Please Refer to the ReadMe files of mach2 and zeta2.

Similarly we can compare the errors for 2 threads and 8 threads for the mach method for example. We also obtain the same results than with openMPI method :

Error	NT = 2	NT = 8
N=1	4,17E-002	4,17E-002
N=2	9,95E-004	9,95E-004
N=3	2,83E-005	2,83E-005
N=4	8,81E-007	8,81E-007
N=5	2,88E-008	2,88E-008
N=6	9,74E-010	9,74E-010
N=7	3,37E-011	3,37E-011
N=8	1,19E-012	1,19E-012
N=9	4,30E-014	4,30E-014

NT = number of threads

This is normal because the calculus of Pi is the same, the repartition is the same, the only thing which changes is that partial sums are calculated by threads.

Question 9:

Please refer to the ReadMe file of the hybrid folder to run the program .

This program respect the format of the openMPI method, however in each process, we apply the openMP method to compute the partial sum.
We could have done the contrary, respect the openMP format, and in each thread, use openMPI method. We have chosen the first one.

The hybrid version has been implemented for mach and zeta methods.

Question 10:

The more the number of processes is high, the less the needed memory per process will be important for n>>1.
However, the more the number of processes will be high, the more time for communication between processors will be needed.

It depends of the method, for example for zeta method :

Let's consider an array of size N.
For each case pf the array we have to compute 1/(1+i)**2 which is done with: 1 addition, 1 multiplication and 1 division. So three floating point operations:

Z(N) = 3*N = O(N) with Z(N) the number of floating point operations for the zeta method with an array of size N.

Similarly we can compute the M(N) number (for mach method), we would obtain:

M(N) = lambda*N = O(N)

It's something linear in the number of cases in both methods.

To know the number of operation to compute Sn, we have to fix the number of process P (thread T) and N.

Each partial vector has a size N/P or (N/P) +1. (respectively N/T or (N/T) +1)
Our program is not perfect, we could calculate the partial sums and the final sum with dichotomy but we don't.
So each partial sums need N/P -1 or N/P operations (respectively N/T -1 or N/T)

We only have one process (thread) which has a vector of size N/P +1. So the final number of addition is:

Z(Sn) = (P-1) + (P-1)(N/P -1) + N/P = 2N*(P-1)/P = 2N – 2N/P (resp 2N – 2N/T).

The multi process program is load balanced because each process has the same number of floating-point operations (additions) to do (to 1 near).

Question 11:

This problem is to simple to see the real efficiency of parallelism. We don't really see the difference in terms of errors of time execution.
However, the more the problem is complex, the more [with an adapted number of processes] the problem will be divided into processes and the more our program will be effective.