Introduction to Supercomputing

TMA4280 · Project I

This project is mandatory, counts for 10% of the final grade and can be done in pairs or alone.

Instructions

The deadline is set on the **7th of March 2018**.

The deliverable consists of:

1. a report describing your solutions and interpretation of the results, handed out through Blackboard in PDF format,
2. a GIT repository containing the source code developed to perform the computations.

Practical requirements regarding the code:

- it should be hosted on Github,
- it should be structured in different subdirectories addressing the different questions,
- it should contain results gathered in text files,
- it must be written in C, C++ using C arrays only, or FORTRAN,
- it must use double precision,
- it must compile and run using Makefile targets,
- results presented in the report have to be reproducible.

As soon as you have decided whether you want to work in pairs or alone:

- create a Github repository (one per pair) named `TMA4280v2018`,
- send your name(s) and the link to the repository by email.

All the developments for this project will be contained in a subdirectory named `P1`.

Description

The project deals with the computation of π using different approximations and their parallelization using first MPI, and then with a small modification to use OpenMP.

Considered approximations are based on the development in series,

$$S_n = \sum_{i=1}^{n} v_i.$$

with v_i depending on the chosen method.

1. Riemann zeta function $\zeta(s)$ with $s = 2$: Each term is defined as:

$$v_i = \frac{1}{i^2}, \qquad i = 1, \dots, n$$

and

$$S = \lim_{n\to\infty} S_n = \frac{\pi^2}{6}.$$

2. Machin Formula: Given $x \in [-1, 1]$,

$$v_i(x) = (-1)^{i-1} \frac{x^{2i-1}}{2i-1}$$

and

$$S(x) = \lim_{n\to\infty} S_n(x) = \arctan(x)$$

with

$$\frac{\pi}{4} = 4 \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

In all the questions, the name in bold and enclosed in parenthesis denotes the subdirectory where the implementation should be located. All the computations should run with the command `make test` and in the parallel case specifying the number of processes with the environment variable `NP`. You may factor common code in a library if you want, but it is not a requirement.

Question 1. *Serial implementation.* Write a serial program implementing the computation of π for a given n read from the command line for:

- Method 1. add a program in (**zeta0**)
- Method 2. add a program in (**mach0**)

Question 2. *Unit test.* Every development should come with unit testing to check the logic of the implementation. Such tests should execute quickly and compare a computed value against an expected value. Implement a small test comparing the value of each series with $n = 3$.

- Method 1. add a unit test in (**zeta0**)
- Method 2. add a unit test in (**mach0**)

The test should be implemented in a simple function (no unit test framework required) and executed with `make utest`. Why do you think such test may be useful when parallelizing a computational code?

Question 3. *Verification test.* When the logic of the implementation is tested then the mathematical properties of the algorithm should be assessed: convergence, stability. Such tests should execute fairly quickly and verify the behaviour of the algoritm as compared to the theory if possible. Implement a small test computing the error $|\pi - \pi_n|$ for $n = 2^k$ with $k = 1, \ldots, 24$:

- Method 1. add a verification test in (**zeta0**)
- Method 2. add a verification test in (**mach0**)

The test should be executed with `make vtest` and results should be saved in a file. Comment on the obtained results.

Question 4. *Data distribution.* As we are interested in computing the sum of all vector elements v_i numerically, we will work under the constraint that the values should be put in a vector before being summed. The suggest program deliberatly relies on partitioning and distribution of the data by process zero.

Process zero **only** should be responsible for:

- generating the vector elements,
- partitioning the vector in a way that the problem is load-balanced,
- and distribute the elements to all the processes.

Each process will work on the received data.

- Method 1. add a program in (**zeta1**)
- Method 2. add a program in (**mach1**)

Can you comment on the limitation of such approach for the data distribution and a possible improvement? Provide arguments to support your answer.

Question 5. *MPI implementation.* Modify further the program to compute the approximation of π using both methods such that each process:

- computes a partial sum from its data,
- then all the partial sums should be added together on the root process,
- and then global sum is printed on the standard output by the root process.

Only the root process holds the final value. Report the error $|\pi - \pi_n|$ in double precision for different values of n and the wall time, for different number of MPI processes which are powers of two. The program should contain an assertion and fail if the number of processes is not a power of two, as a design constraint.

- Method 1. modify the program and add a test in (**zeta1**)
- Method 2. modify the program and add a test in (**mach1**)

Plot the error and the timings. Which MPI calls were convenient and/or necessary to use? Can you comment on the methodoloy used for computing the wall time?

Question 6. *Analysis*

Compare the errors from the single-process program and the multi-process program for $P = 2$ and $P = 8$. Should the answer be the same in all cases? Exactly, or approximately? Can you explain why? Provide arguments to support your answer.

Question 7. *Global reduction.* Modify the final step, the reduction with `MPI_SUM`, such that all processes store the global sum: first by using an MPI function and then by implementing the recursive-doubling sum.

- Add a program in (**reduc**)

Do a small scaling study, what do you observe?

Question 8. *OpenMP implementation.* Make the necessary changes needed to use shared memory parallelization with OpenMP.

- Method 1. add a program in (**zeta2**)
- Method 2. add a program in (**mach2**)

Perform the same analysis as for the *MPI implementation*.

Question 9. *Hybrid MPI/OpenMP implementation.* Confirm that your program also works when using OpenMP and MPI in combination. What is the advantage of running such configuration?

Question 10. *Discussion.*

- Compare the memory requirement per process for the single-process program and the multi-process program when $n \gg 1$.
- How many floating point operations are needed to generate a vector $\boldsymbol{v}$?
- How many are needed to compute S_n?
- Is the multi-process program load balanced?

Question 11. *Conclusion.* Do you consider parallel processing attractive for solving this problem? Explain why.