Report, tma2018

Stein-Erik Bjørnnes & Sindre A. Gjerde

# Intro:

In this report we are going to calculate pi and look at multiprocessor performance. To do this we will use OpenMP and MPI. MPI runs the program in parallel, and we get more processes. OpenMP we get more threads. We use two distinct methods to calculate pi, Reimann Zetta function [Theory eq. 1] and the Machin Formula [Theory eq. 2]

# Theory;

## Pi calculation:

1. Reimann zeta function  with s equal to 2:



1. Machin Formula x  :



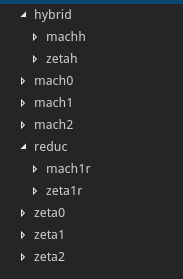
## MPI:

## OpenMP:

# Results:

# Code:

Our directory:



# Questions:

## Q1 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)**

Q2. 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?**

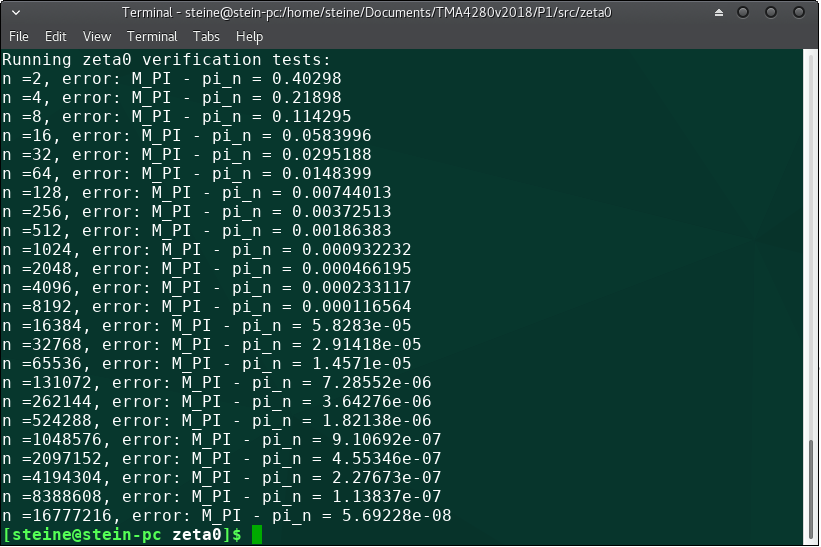
A unit test is useful when parallelizing code because in this way we can make sure that calculations from all processes/threads are taken in account when finishing up. A unit test will in this case uncover if the pi sum from one process/thread was not summed into the final answer because the sum found would be lower than the expected value.

Q3. 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 |π − πn | for n 2 k with k 1, . . . , 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.**



Results shown in the file shows that the error for zeta0 decreases when increasing iterations. We can also see that the error gets cut in half each time we double the iterations. This should be sufficient data to conclude that the algorithm both is stable and converges.

**TODO: MACH0**

Q4. Data distribution.

**As we are interested in computing the sum of all vector elements vi 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.**

The limitations of such an approach is that all of the vector elements must be created by the host process. A better solution would maybe to have process zero to distribute a range for each process to work within, before each process would sum its own values and reduce them in the end.

The reason for this is that there is a heavy load on process zero to generate all the elements and distribute them. The generation of the vector elements is a heavy load and should be distributed since the child processes don’t have any work to do in the meantime anyways.

Q5. 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 |π−πn | 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?**

To complete this task, we used the functions MPI\_Bcast and MPI\_Reduce. MPI\_Bcast was used to distribute the initial vector(s) generated by the root process and the number of elements each process should compute. MPI\_Reduce was used to sum all partial sums from each process to a global sum.

To compute the wall time we used the variable MPI\_Wtime, and stored its value in a local variable in process zero before the run. We later called the MPI\_Wtime again and compared the value we stored from before to the new value and printed the difference.

Q6. 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.**

Q7. 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?**

Q8. 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.**

OpenMP was much easier to implement. We used the sentence

#pragma omp parallel for reduction (+:s)

to reduce(/sum) all values in a for loop to a single variable(in this case s). To compute the wall time we used chrono time, stored the value before the calculation and compared again with chrono time after the run.

Q9. 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?**

The advantage of running a configuration like this is that we can use OpenMP to achieve parallelism within the same memory device since it uses parallelism between threads. To achieve even more parallelism we could use MPI to distribute the load using its message passing interface and by so achieving parallelism on distributed memory devices where each process only works with its own memory, isolated from other.

Q10. Discussion.

**The following topics:**

* **Compare the memory requirement per process for the single-process program and the multi-process program when n >> 1.**
* **How many floating point operations are needed to generate a vector v?**
* **How many are needed to compute Sn?**
* **Is the multi-process program load balanced?**

Q11. Conclusion.

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

# Conclusion