

# TMA4280 – Project 1

Maximilian Kieler

March 7, 2017

## 1 Introduction

Aim of the Project is to implement an algorithm to sum a series in a serial way and in parallel by using MPI and OpenMP. The exact tasks and further specifications can be found in [].

## 2 Serial Implementation

It is task to implement a serial method to find the solution of two sums. Therefore we use the language c++. The implementation is done in an object oriented style and uses the template mechanism, which make it possible to use the same basic class for both sums. We provide a method to solve the sum and print the result. Furthermore there exists a unit test, which prints the error of the result to the original value for a sum over three terms. Finally there exists a verification test, which gives the error in dependence to the number of summed terms. As we see in Fig. 1 the convergence of series 2 is better than of series 1. Less terms are needed to get the most exact result which is possible on a computer with double precision.

## 3 MPI Implementation

The implementation should be done in the way, that we allocate a vector, fill it with the values of the terms and distribute the to the processes. This is not really a suitable solution to the problem in the way that we allocate an possibly big array, but we need the values only once. The summation can be done immediately after evaluating the terms, but however, the task is written to practice the MPI mechanism. To avoid the

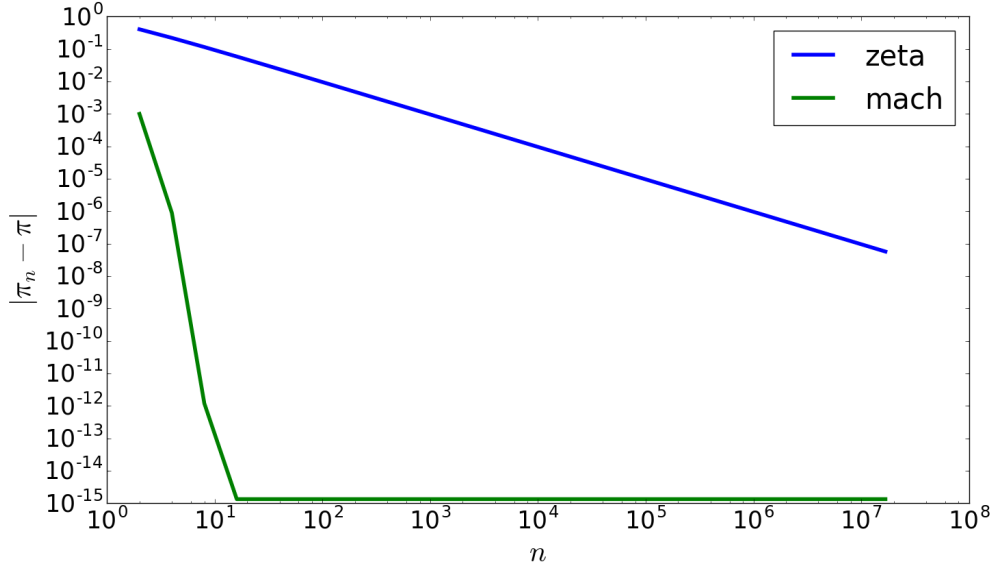


Figure 1: The convergence of the zeta series is worse than the mach series. This series converge after four terms to machine precision. The use of a big number of terms is here useless and parallelisation not necessary. However the worse convergence of the zeta series make it necessary to use many terms to get a good result.

allocation of a big array in the root process, we decided to allocate only a array of the size of the distributed array and fill this in the loop before sending. This leads to a slight deparallelisation and is only suggestive if the filling process takes not to much time. After sending the distributed array every process receive its part and sum over it. Afterwards every partial result will be returned to the root process and summed together.

The program measure the wall time by using the MPI wall time function. As we are interested in the execution time of the program we start the measurement in the root process and stop it there after all processes have returned their results and added together. It turns out that in reason of the restricted memory size the expected speed up do not occur. The loop over the vector elements is faster than the communication between the processes. My jobs at the supercomputer lille were not accepted and remains all in the queue, so I was constrained to use my own computer. Therefore the number of processes is restricted to 4. Nevertheless Tab. ?? shows the obtained speed up for the

P	$N = 10^3 \times 10^6$	$N = 10^4 \times 10^6$
1	$1 \rightarrow 1.21s$	$1 \rightarrow 10.29s$
2	1.62	1.56
4	1.72	1.61

Table 1: Bla

P	$N = 10^3$	$N = 10^4$	$N = 10^5$	$N = 10^6$
1	$1 \rightarrow 8.82 \cdot 10^{-6}s$	$1 \rightarrow 185.9 \cdot 10^{-6}s$	$1 \rightarrow 9.84 \cdot 10^{-6}s$	$1 \rightarrow 1.10 \cdot 10^{-3}s$
2	0.25	1.37	1.13	0.97
4	0.15	0.89	0.87	0.78

Table 2: Bla

zeta series. To test whether our program works in parallel or not, we added a loop to the critical part, the sum over the vector, in each process. This means we do this critical part many times, instead of once. This leads to a longer calculation time. This version shows an expected speed up behaviour, see Tab. 1.

We can not expect, that the result of the MPI implementation is completely the same as for the serial implementation. There we add a small number to a number of a much higher magnitude. Due to floating number calculation the small number will be seen as zero. In the MPI implementation we calculate local all small numbers of the same magnitude, so they reach possibly a higher magnitude and add this partial sums afterwards. Therefore this result can be more precise, but in case of the mach series, which converge fast it is dangerous, because we add a lot of terms which are under the machine precision and not reliable. However we found for the zeta series small deviation of the error in dependence on the number of processes, see Tab. 3.

P	Error $\cdot 10^{-5}$
2	5.96818054963144
4	5.96818054887649
8	5.96818054954262
16	5.96818054963144

Table 3: Bla

P	$N = 10^3$	$N = 10^4$	$N = 10^5$	$N = 10^6$
1	$1 \rightarrow 4.77 \cdot 10^{-5}s$	$1 \rightarrow 101 \cdot 10^{-6}s$	$1 \rightarrow 1.96 \cdot 10^{-3}s$	$1 \rightarrow 12.5 \cdot 10^{-3}s$
2	0.76	0.84	2.61	2.16
4	0.30	0.96	0.74	1.64

Table 4: Bla

## 4 OMP Implementation

We use OpenMP in the way the we parallize the loop which fills the vector. This operation is completely load balanced, so no further options are necessary. The second addition is done by parallize the loop which sums the vector by using the OpenMP reduction. These two simple additions can also be added in the MPI implementation. In Tab. 4 we observe for big numbers of  $N$  and good speed up, this is compared to MPI possible, because we have less communication of the memory. But the complete computing time is higher than the MPI implementation, further is the fluctuation of the time high on my computer due to other processes running in the background. We observe similar fluctuations of the error in dependence of the number of processes as in the MPI implementation.

## 5 Conclusion

As previous said, the way of implementing the problem by first filling a vector is not efficient. The number of FLOPs for filling a vector of size  $n$  is given by  $\text{FLOP}(\text{fill}) = n \cdot \text{FLOP}(\text{eval})$  (if we assume that assignment is also a FLOP), where  $\text{FLOP}(\text{eval})$  is the number of FLOPS needed to evaluate one term of the series. For the mach sum there are more FLOPs needed than for the zeta sum. The summation over the vector consists in assign the value, add it, and assign the result, which gives us  $3n$  FLOPs. More efficient is to collapse the two loops and do the summation immediately. Nevertheless the way of the implementation is perfectly load balanced if the number of terms is a multiple of the number of processes. This is in reason of simplicity the only allowed case in the implementation. Otherwise the will be one process which only have to sum over  $n - n\%(P - 1)$  remaining terms. However the use of parallelism is for this problem not necessary. We have seen that the zeta series converges slowly, so many terms are needed, here it could make sense to parallize the loop, but it is much easier to use the mach series which converges after 4 terms to machine precision. For a loop over 4 terms the overhead to organize the parallel environment is much more time expensive as to

compute it in serial. To solve this problem in parallel is only useful as an exercise for learning purpose.