# TMA4280 Introduction to supercomputing
# Problem set 4

Raymond Toft

26. februar 2016

This exercise have been used to get a introduction into how to utilize parallel programming using MPI and OpenMP in a simple C program.

## 1 Introduction

This introduction to utilization of parallel programming is performed by considering simple computations with a vector. Here we are invited to look at a vector $v \in \mathbb{R}$ n with the elements defined as

$$v(i) = \frac{1}{i^2}, \quad i = 1, ..., n \tag{1}$$

The operation we want to perform on this vector is the sum of all the elements, $S_n$ numerically,

$$S_n = \sum_{i=1}^{n} v(i). \tag{2}$$

As a reference when we calculate the difference between the numerical approximation and a exact solution, we use the given value $S = \lim_{n\to\infty} S_n = \pi^2/6$.

## 2 Writing the program

Since the goal is to end up with a parallel program, as much of the code as possible is written in separate functions. This is in general good programming approach, but it is easy to skip the division of the code in such small programs that is considered here. The two most central functions is generating the vector in $initVector()$ that allocates and initializes the vector for a given vector struct and computing the sum in $vectorSum()$. The main function switches between MPI, openMP and/or serial mode and declares the different vector structs, calls the necessary MPI functions, computes the difference etc.

### 2.1 OpenMP

To start utilizing the shared memory parallelization through the OpenMP standard, the *omp.h* library is included. In this part of the exercise, there is three for-loops that may be parallelized. The for-loop in the main file is not considered benefitial due to the usage of MPI within the loop. The function $initVector()$ is easily parallelized by adding the command $pragmaompparallelforschedule(static)$. This initializes the distribution of work in equal blocks of elements into each thread. The workload is quite similar for each iteration in this loop, and thus a static distribution is most beneficial. Similarly procedure is done with the $sumVector()$ function.

## 2.2 MPI

By using a distributed memory model as MPI, we rewrite the original program. This is done quite simply by firs changing the code so that an bool desides whether to run openMP or serial. In this program, and also most common it seems, the root processor, or processor 0, is set to generates the vector elements. The parallelization here is considered with the summing of the elements in the vector. In order to use MPI, we have to initialize it in the main function by calling MPI_INIT, MPI_COMM_size, and MPI_COMM_rank to initialize, get the number of processors and the current processor id or rank, respectively. The omp.h library is included as well. For the summing of the elements in the vector, we use two functons from the MPI library. We first call MPI_Scatterv to distribute a certain amount of the vector to each processor, that do a local sum of the block and store the result in a given local variable.In this application, we could simply have used MPI_Scatter since the requested number of processors, 2 and 8, are guaranteed to divide the vector size since it is a power of two. To make the code more robust and versatile the former version of the MPI function where chosen. The next call is MPI_Reduce that collects all the local stored results and by specifying it to use the operation MPI_SUM, it sums up all the collected local sums and store the result in a given variable known to the receiver process.

## 2.3 Combination of OpenMP and MPI

Instead of only using one method of parallelization at the time, both can be combined by using bool statementsin the program. As mentioned before the for loop in the main function that runs the whole program repeatedly. The reason for this is that trying to utilize MPI inside a OpenMP thread might cause some trouble that gives random results in the computations. Because of this, the OpenMP parallelization of the for loop in the main function is simply walked over and is not a part of the final program. On the other hand, it is fine to cast the local summation on several threads inside the MPI call. By running the code, the results show that the implementation of the two parallelization techniques in combination is working.

## 2.4 Convenient MPI calls

The convenient MPI calls that is made in this program is, as mentioned previously, MPI_Scatterv and MPI_Reduce. The first one is quite usefull in many circumstances due to the need of distributing blocks of the vector to the different processors such that they can start working on the data. The MPI_Reduce has many functionalities regarding collecting data from the used processors and in the same time perform simple operations on the collected data. The operation used here is MPI_SUM, but it also offers MPI_PROD, MPI_MIN, MPI_MAX etc. The result from the MPI_Reduce is known only to the root processor, but another function such as MPI_Allreduce makes the result known to all of the processors This functionality is especially practical if the results is to be used in further computations.

## 3 Consderations

### 3.1 Round off error

When the results in of the difference $SS_n$ is compared with use of P=2 and P=8 processors, it shows that the answer is not the same. The error is quite small, and it is not before in the sixth or seventh digit that a difference can be spotted for the larger values of n, but is might cause some trouble. The reason for this is that the processors receives different blocks with different values for the individual elements. The problem starts when the magnitude of the elements and thus the local sum varies so much that the processors round off in different scales. When the result is combined into a global answer the roundoff form one processor might be bigger than the local sum from another. For scientific experiments this can cause a lot of trouble when the results easily becomes very sensitive for round off errors.

### 3.2 Memory requirement

As expected, the memory requirements per processor for the multi-processor program drops compared to the single-processor program when $n \gg 1$.

### 3.3 Floating point operations performed

In this section, the number of floating point operations, flops, that is performed is considered. To generate the vector v, there is performed one division and two multiplications for each element i. This gives a total amount $nFlop = n(1+2) = n3$ of flop. With the first parallelization of the summation, there is performed $nFlop = n1$ flop to compute $S_n$. In this application it was fearly easy to predict the cost of each operation and thus distribute the work evenly between the processors, which makes the program load balanced. On the other hand does the root processor handle and initalte much of the communication, but this is difficult to balance out since the dataset is small and the two collective operations already is optimized for communication balance.

### 3.4 Attractiveness of utilizing parallel programming

For this simple problem, the gain of using parallel programming is not that beneficial considering the size of the program and data set compared with the extra work with utilizing the parallelization. By taking the walltime of the program, it was easy to see that the overhead in initializing MPI and communication had a huge impact on runtime for small problem sizes. On the other hand, if this had been a part of a larger system, it might be getting more relevant to process this in parallel.

## 4 Conclusion

In this exercise a small program have been written get known with two ways of parallelization. The problem considered was generating a vector and computing the sum of it's elements. OpenMP was easily utilized by using a simple line of code to parallelize some of the for loops. MPI where then used to distribute blocks of vector elements to multiple processors where a local sum was computed and retrieved to the root process in order to compute the global sum. A roundoff error issue regarding different sizes of elements on the processes and thereby different "round off scaling". The parallelization was strictly not necessary for this specific problem set, but will be much more attractive for bigger systems.