Problem set 4 - TMA4280

Morten Vassvik and Jon Vegard Venås

February 2016

The serial code is implemented in `compute_S.c`. The usage of `uint64_t` (from the `stdint.h` library, a typedef corresponding to `unsigned long long int`) is included to be able to study $n = 2^k$ up to[1] $k = 31$. The corresponding capacity for `unsigned int` and `unsigned short` (32-bit and 16-bit) are $k = 15$ and $k = 9$. Moreover, we include the `time.h` library to calculate computational time.

Note that we compute the sum

$$S_n = \sum_{i=1}^{n} \frac{1}{i^2}$$

by adding the terms in reversed order to obtain the optimal accuracy[2].

The OpenMP parallelization is obtained from the serial code by adding `#pragma` commands above each `for`-loop. We substitute the `time.h` library with `omp.h` library to compute the computational wall time with `omp_get_wtime()`. Additional arguments to the `#pragma` are used to specify intent and hints, such as shared and private variables, if parallellization are to be performed across several loops, or if a reduction to be performed.

Some results for the MPI implementation is reported in Table 1, while the full set of results is plotted in Figure 1. All though process zero should be responsable for generating the vector elements, there is no need to allocate the full vector as each sub vector can be created, sent and overrided for the next sub vector. This allows for an evenly distribution of memory across the processes, such that the memory requirement for each process is reduced by a factor P (number of processes) compared to the single-process program.

As before, we can use MPI and OpenMP in combination by extending the MPI implementation with `#pragma` commands above each `for`-loop. Note that there is no need to declare the index `i` as private, as it is declared inside the `for`-loop.

We had to use `MPI_Send()` from rank 0 to send the vectors to all other processes, who used `MPI_Recv()` to recieve the same data. `MPI_Reduce` was used to perform the sum of the partial sums, but are not nescessary, as the same results could be done with Send/Recv.

[1] Note that the dominant restriction on the data type for the variable n is when we compute $\frac{1}{n^2}$ in the final term in the sum. With 64 bits available the product `n*n` makes sence for $n = 2^{31}$.

[2] Adding numbers of similar exponents increase accuracy.

Table 1: Comparison of the error $|S - S_n|$ for different values of k and number of processors P.

k	Serial	OpenMP	MPI
3	$1.175\,12{\cdot}10^{-1}$	$1.175\,12{\cdot}10^{-1}$	$1.175\,12{\cdot}10^{-1}$
14	$6.103\,329{\cdot}10^{-5}$	$6.103\,329{\cdot}10^{-5}$	$6.103\,329{\cdot}10^{-5}$
31	$4.656\,613{\cdot}10^{-10}$	$4.656\,613{\cdot}10^{-10}$	$4.656\,613{\cdot}10^{-10}$

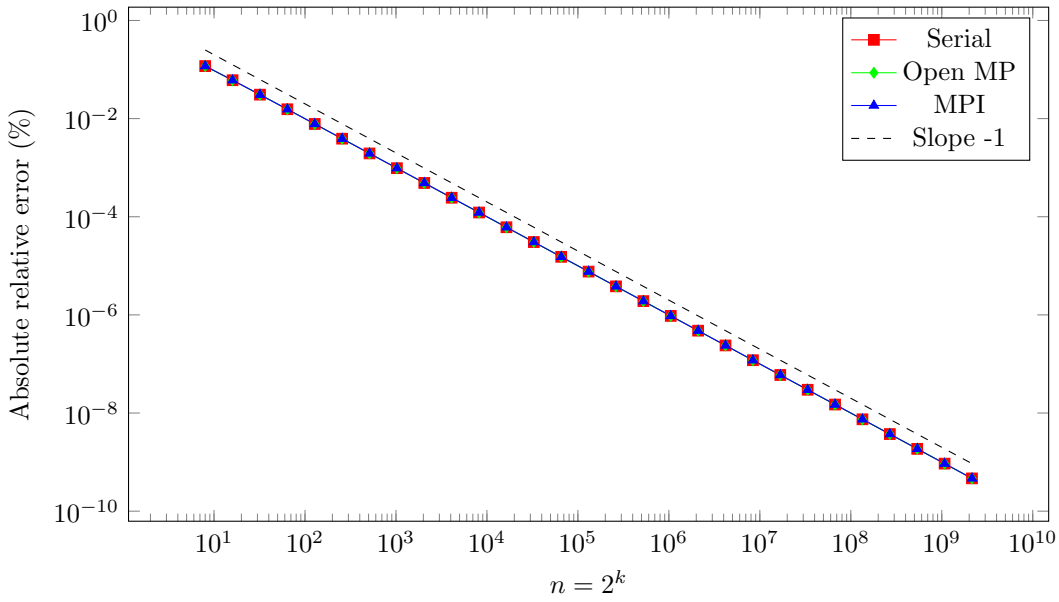

Figure 1: Plot of the error $|S - S_n|$ for $k = 3, \ldots, 31$ for the three different implementations.

Table 2: Comparison of the error $|S - S_n|$ for different values of k and number of processors P.

k	$P = 1$	$P = 2$	$P = 8$
3	$1.175\,120\,146\,940\,314 \cdot 10^{-1}$	$1.175\,120\,146\,940\,312 \cdot 10^{-1}$	$1.175\,120\,146\,940\,314 \cdot 10^{-1}$
14	$6.103\,329\,364\,326\,449 \cdot 10^{-5}$	$6.103\,329\,364\,259\,835 \cdot 10^{-5}$	$6.103\,329\,364\,304\,244 \cdot 10^{-5}$
31	$4.656\,612\,873\,077\,393 \cdot 10^{-10}$	$4.656\,612\,873\,077\,393 \cdot 10^{-10}$	$4.656\,615\,093\,523\,442 \cdot 10^{-10}$

`MPI_Wtime()` is convenient to get the wall-time. We also need to use the standard initialization and finalization routines, `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, and `MPI_Finalize`.

We could have included non blocking functions `MPI_Isend()` and `MPI_Irecv()` to let the rank 0 process continue without waiting for `MPI_Recv()` to return, but as this would require a barrier elsewhere and not give significant improvements, we chose not to.

The answers should be approximitely the same, but not exactly, due to the accumulation of floating point errors. In addition, the order in which a sum is performed will change its value. This is illustrated in Table 2.

For each of the n elements in the vector $\boldsymbol{v}$, we perform an integer multiplication, a type cast (from int to double) and a floating point division. Moreover, the iteration index will be incremented and checked against n.

The vector generation is done on only one processor (for MPI), which does not result in a load balanced program as the vectorgeneration consumes about the same time as the summation. It is possible to solve this problem if each process generated their own part of the vector $\boldsymbol{v}$.

Summing a series of numbers is ideal for parallel computing, but it depends on the problem itself. In this case, the series is so slowly converging that there is a huge accumulation of floating point errors. This results in the result unpredictable and varying precision in the final results, as the order in which the sum is performed is important. Moreover, for the MPI implementation, it should be more load balanced.