Assignment 2: Parallelization with MPI and Performance Analysis

Computer Architecture and High Performance Computing

Christian Wiskott, 01505394.

June 19, 2020

Outline and MPI parallelization-strategy

In the second part of this course an optimized parallel MPI-program had to be written as an improvement over a non-parallel code, conducting a series of row and column operations on a generated matrix of size N=8192 using STEPS=200 iterations. The goal was to achieve a substantially faster execution time while maintaining the functionality of the code.

This was achieved by using a combination of Scatter, Gather and Send/Receive functions that distributed the work equally among the processes. The tasks of the MASTER process were to generate the matrix, send out the rows and columns to the slave processes and to receive and manage the results. Figure 1 outlines the scheme that was used to achieve the parallel execution and was developed in collaboration with Justus Rass.

The computation was separated into two phases, due to the altering computational wave fronts between the phases, which lead to a dependency between the second- and the first phase and thus a fixed order has to be followed to ensure a correct checksum. After the initialization of the matrix by the MASTER, the first phase begins during which the MASTER sends an equal amount of rows of the matrix to the slaves via *MPI_Scatter*, which individually calculate their corresponding results. Here the parallel code comes into play in a divide-and-conquer fashion. After the slaves are finished the MASTER gathers the updated rows via *MPI_Gather()* and overwrites the initial matrix with the new rows.

With that, the first phase is completed and reigns in the second phase, which deals with the columns of the matrix. Due to the columns of the matrix not being contiguous, the distributed columns had to be resized by using *MPI_Vector* and

MPI_create_rezised. This solution has also been developed in collaboration with Justus Rass. Now the resized columns can be distributed via another *Scatter*-Function, which the slaves receive and conduct the corresponding column operations. In order to complete phase 2 the updated columns are sent back the the MASTER via a *Send/Receive*-Block and are used to update the resulting matrix. With both phases complete, this concludes one iteration of the calculation. After all iterations have been completed, the resulting matrix is run through the last section of the code, which calculates the checksum i.e. adds up all matrix elements to check for the correct result.

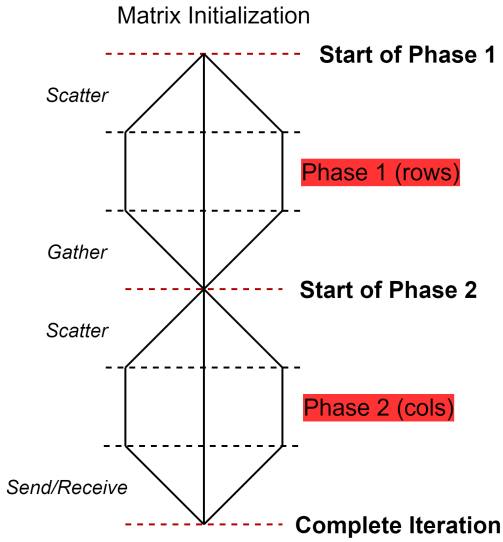

Figure 1: Visual representation of the parallelization strategy.

The reason, the *Send/Receive*-Block in the second phase was used, was due to problems with the *Gather* function at high matrix dimensions using the resized columns of the matrix, which interestingly was not the case with the rows in the first phase. But since the *Send/Receive* serves the same purpose, it was adopted as an alternative. Tests were conducted to also compare *Scatter* and *Send/Receive* in both phases. They showed, that the *Scatter* function completed the given task faster and thus was included in the final implementation.

The final checksum of the MPI-implementation resulted in the value 479265832472.00, which coincides with the serial version.[1] Table 2 showcases the comparison of run

[1]To ensure matching results, a typo in the serial program was fixed. In line 66 there is a stray semicolon which ends the column operation prematurely.

Amount of Nodes	Execution Time [s]
1	25.71
2	159.37
3	187.19
4	248.37

Table 1: Increasing run time for increasing number of nodes utilized for the computation. For testing purposes the problem was reduced to a smaller size with N=8192, STEPS=10 and using 16 processes.

times of serial and MPI for the desired problem size. It clearly shows, that the MPI-version vastly outperforms the serial program, with size=16 being the optimal amount of processes for this problem and beating *gcc* by a factor of ~ 5.5 and *icc* by a factor of 2.75. For a visual representation of the correlation between amount of processes and run time, please consult fig. 2.

Table 1 shows, that the amount of nodes utilized for the calculation is of utmost importance. For a smaller problem size, the run times deteriorate very rapidly for more than one node. This could stem from the high communication cost between the nodes, which would not be present in the case of one single node. Therefore one node was used for the computation of the full problem size.

In order to monitor the system during the computations, the command *qstat -f* was used to examine the state of load of the currently used node. It was useful to watch the average load in the beginning of the job, since values higher than roughly 10 resulted in far longer run-times than during a low average load, and rendered the results not usable for the comparison. To further monitor the system, the average run times for an average loop using different numbers of processes in the computation was calculated using *MPI_Barrier* and *MPI_Wtime* and were included in the table 2.

In conclusion the full problem size should be tackled with one node and 16 processes to ensure the fastest possible runtime.

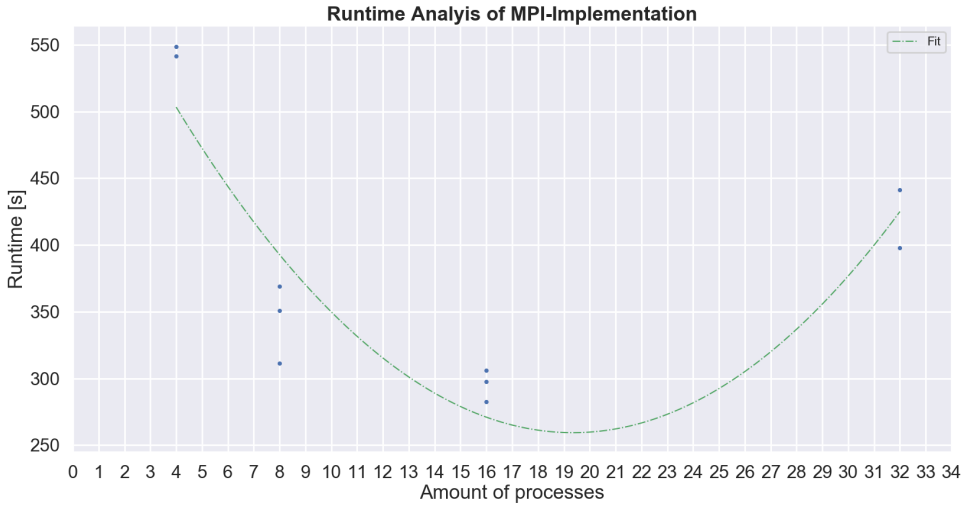

Figure 2: Run time analysis of the MPI-Code for different amounts of processes. The optimal amount seems to be 16 processes.

Size	Execution Time [s]	Average Loop Time [s]	Avg. Load
4	548.724725	2.75	3.2
8	311.264275	1.57	1.2
16	282.653266	1.41	0.3
32	398.007919	1.97	2.4

(a) MPI Run Time Analysis

Compiler	Execution Time [s]	Average Loop Time [s]
gcc	1545.53	7.73
icc	777.10	3.89

(b) Serial Run Time Analysis

Table 2: Comparison of the best run-times of the serial and the MPI-Code. In all cases the end case was evaluated using N=8192, STEPS=200 and utilizing one node.