Assignment 2: Report

Name: **Lukas Hinterleitner**
ID: **51912219**

Based on your solution and understanding of the assignment and lecture slides, fill out the questions below:

1. Identify opportunities for parallelism in the sequential version of the code by stating which loops can be parallelized using OpenMP. Consider all loops of the sequential code and explain why these loops can be parallelized, and how did you proceed to parallelize them with OpenMP.

   The main opportunities for parallelism in the sequential code are the loops that iterate over the matrix *U* und *W*. These loops are independent, meaning that each iteration of the loop does not depend on the results of the other iterations. This is especially true for the initialization loops. For the calculation loops, this is also true since data is only written to the W matrix in the first for-loop. Hence, the *U* matrix where data will be read from different locations is not touched. The second for-loop that can be parallelized only transfers data from *W* to *U* and is independent from the previous loop and hence, does not interfere with it. Summarized, I parallelized all outer-for-loops and OpenMP. For the loop where the *diffnorm* is calculated, I used the term *reduction(+:diffnorm)* to synchronize the variable *diffnorm* with the threads since it is the stopping criterion for the do-while-loop.

2. Was there any loop in your OpenMP code that you could not parallelize? Explain why you could not parallelize it via data dependency analysis.

   Yes, the do-while-loop. This loop was not parallelizeable because then the U and W operations would not be independent of each other anymore which leads to data races etc.

3. In your OpenMP code, you were supposed to implicitly define the data-scope of all variables that are used within a parallel region. Justify your data-scoping choices for each variable within the extent of a parallel region.

   Basically, the variables, *int I,j;* are defined in the outmost scope of the main method. When we leave the code like that, we have to tell OpenMP to use this variables in private for each thread, e.g. append to the for-loop pragmas the clause *private(i, j).* But since we need to implicitly define the data scope of the variables, I moved the variable declaration of *int I,j;* into their corresponding for-loops. The *diffnorm* variable has to be defined outside of their corresponding loops since it is our stopping criterion that needs to be synchronized.

4. Was there any variable in your parallel code that you had to protect with OpenMP synchronization constructs? If yes, identify the variable, and explain how you protected the accesses to this variable.

   Yes, the *diffnorm* variable. I appended the OpenMP *reduction(+:diffnorm)* to the corresponding for-loop pragma.

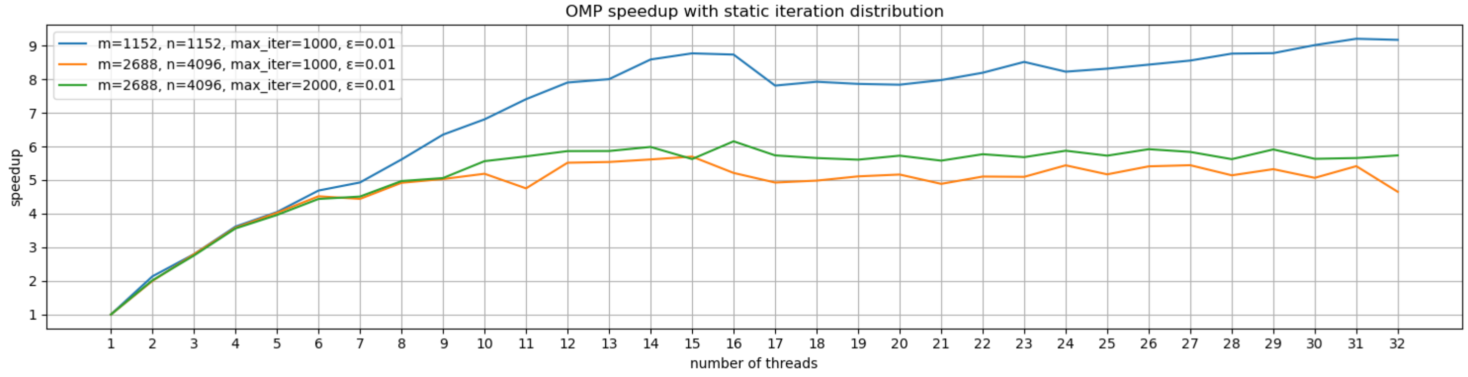
5. In your code you most likely used OpenMP `for` construct to parallelize the different loops of a2.cpp. In the context of this problem, which scheduling strategy would you use to distribute the iterations of the different loops? Justify your scheduling choices. You should resort to the plots of 7. to verify your hypothesis.

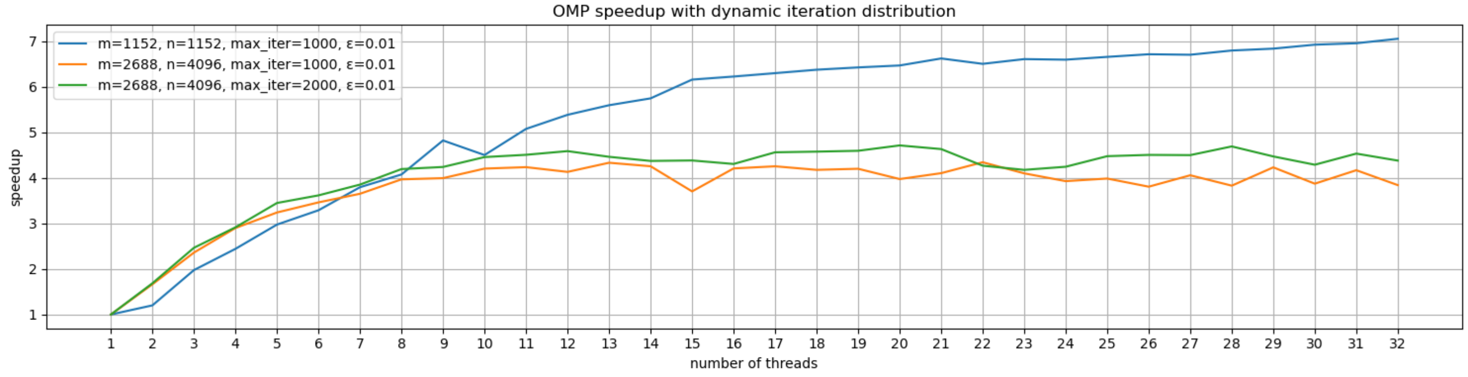
   In the code, the loops have a rather uniform workload which makes static scheduling the way to go. In the plots of 7., we can also see that the static schedule strategy leads to sightly better speedup results.

6. Consider the two loops inside the `while-do` loop of `a2.cpp`. Assuming you parallelized both of these `for`-loops with OpenMP, discuss whether the `nowait` clause could be applied to the first `for`-loop.

   No, it cannot be applied. Because afterwards the code transfers the values from *W* to *U* and when the code does not wait till the previous computation loop is finished we can get data races.

7. Include a speedup plot of your OpenMP solution, with static and dynamic (default chunksize) iteration distribution on ALMA (for 1, 2, 4, 8, 16 and 32 threads). Remember: Use the horizontal axis to represent the number of cores, while the vertical is used to represent the speedup (not execution time).





If the images are too small, they are also included in the submission as png-files.

8. Different matrix sizes result in different performance in both OpenMP and MPI versions. Try to explain this behavior. Consider memory hierarchy and communication/synchronization within and across (MPI) the nodes.

   When the matrix size is small, the entire matrix or a significant part of it may fit into the cache, which can significantly speed up the computation. However, as the matrix size increases, more cache operations may occur, leading to slower memory accesses and reduced performance.

   In the MPI program, processes need to exchange data to update the boundary values of their local matrices. This communication involves data transfer over the network, which is much slower than memory access. As the matrix size increases, the amount of computation increases, but the amount of communication (the number of boundary values) remains roughly the same. Therefore, the communication overhead becomes less significant, and the program can achieve better performance.

9. Explain differences between OpenMP and MPI code. Which code is suitable for which type of architecture? How about the code complexity? What are the common problems that you need to deal with in these approaches?

   Basically, the main difference is that OpenMP is a shared-memory parallel programming model and its code is structured using compiler directives (pragmas) which mark certain regions that should behave in a certain way. With MPI, one has to actively implement the communication between processes and how this communication should look like which makes it a bit more complex than OpenMP. Common problem with OpenMP are multiple threads that try to access or modify the same memory location which can lead to race conditions. Since one has to implement the communication model in MPI common problems are deadlocks but also race conditions can happen over multiple processes.

10. Measure the overall execution time of your MPI program, and measure the time required to collect data on rank 0 for verification (discuss differences for different input arguments). Explain how you measured execution time in the MPI code, and which values you used to print the elapsed time on rank 0. Why these values correctly represent the execution time of your program? Is the time required to collect data different when the program is executed on a single node?

    The execution time is measured using *MPI_Wtime()*. This function is called twice: once before the main computation starts and once after it finishes. The difference between these two times gives the total execution time of the computation. Afterwards, I printed the elapsed time only for rank 0. After the computation each sub-part of the matrix is then gathered on rank 0. Hence, all other parts of the computation are also finished and it is sufficient to only print the elapsed time for rank 0 to get our overall computation time since it also includes communication times between processes, nodes, etc.

    The time required to collect data is probably different when executed on a single node since the program does not need to send the data over the network. However, in the case of the ALMA cluster, all the nodes can probably communicate over an internal network which is quite fast and there won't be much of a difference. Nevertheless, the execution time of course increases with communication latency.

11. How is the data distributed among the MPI processes? How big is each matrix on each process with respect to the M and N. How did you handle the case where (`M%numprocs != 0`)?

    Each process gets a sub-matrix with *int local_M = M/numprocs;* rows. When *M%numprocs != 0* the last process gets a bigger or smaller sub-matrix. I calculated the rows for the last process as followed:

    *local_M = M – (local_M * (numprocs – 1));*

    Then for all processes, *local_M* is again updated: *local_M += 2;*

    The additional two rows serve as padding rows for communication between the processes. The number of columns stays the same over all processes since I chose to do a horizontal splitting of the matrix.

12. What data needs to be communicated between MPI processes and at which points in your code? Which MPI routines have you used to accomplish this?

    First and foremost the rows inside the boundary need to be communicated between processes as stated in the task description on page 9. For that, I used a deadlock save variant of a non-blocking receive *MPI_Irecv()* and a blocking send *MPI_Send()*. Additionally, the *diffnorm* has to be computed globally since it is calculated over the whole matrix and it is the stopping criterion for each process. Therefore, I used the *MPI_Allreduce()* routine to calculate the global *diffnorm* for each process. Last but not least, to merge all the sub-parts of the matrix to a big one at the end, I used *MPI_Gatherv()* to correctly merge them together even tough the last process may have a differently sized sub-matrix.

13. Are there any points in the code where you need to be careful not to introduce a deadlock? If so, where?

    As already written above. The communication between the processes regarding the rows needed for the computation is prone to deadlocks when using only blocking routines.

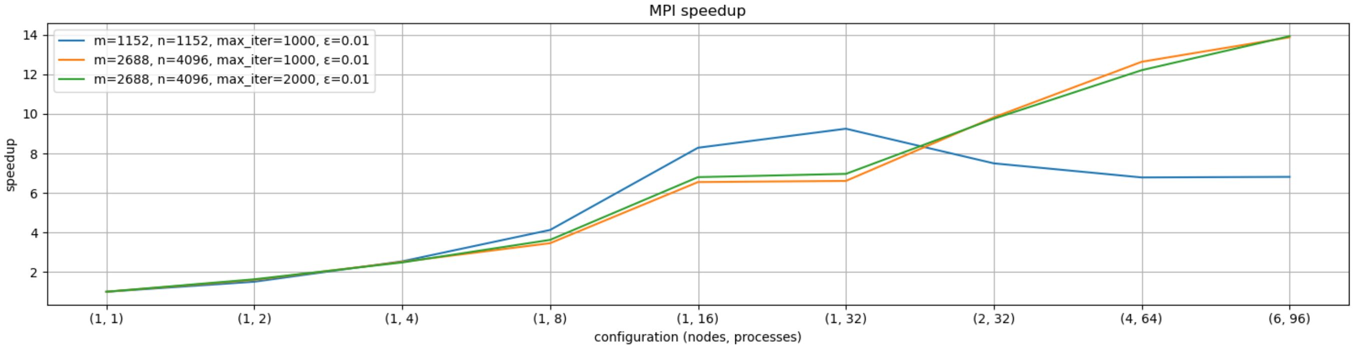
14. Briefly explain how you collected data to rank 0 at the end of your code for verification on rank 0. Which routines have you used and how? What data was relevant, and on which ranks?

    As already explained in 12., I used the *MPI_Gatherv()* routine to gather a dynamic amount of data. Basically, each process sends its sub-matrix to rank 0 without the top and bottom padding rows (it can be of different size for the last process when *`M%numprocs != 0`*).

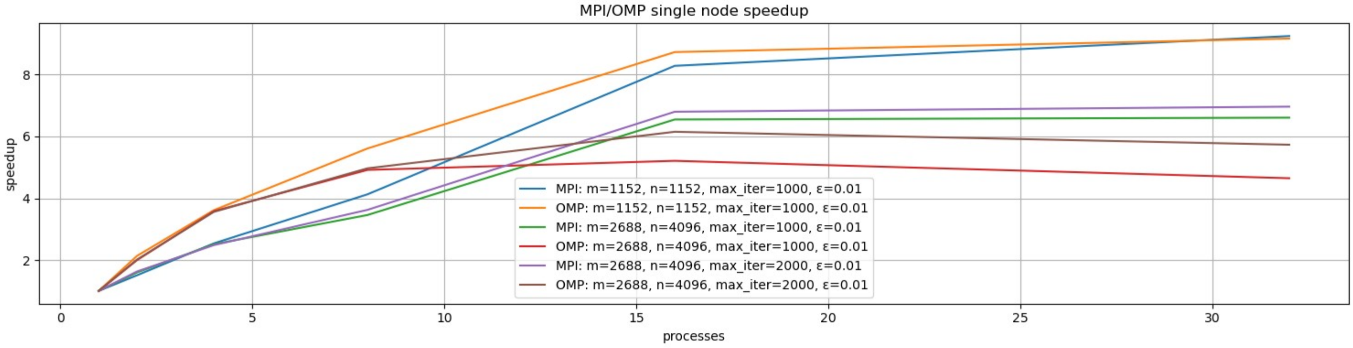
15. Include a table with execution times and speedup of the OpenMP and MPI codes for different configurations given in the slides. Speedup needs to be measured with respect to the sequential version (around 50, 112, and 5 seconds for different versions on ALMA nodes – see slides). If you have these in an Excel sheet, you can just attach it with your submission. Plot OpenMP and MPI single node configurations on a separate speedup graph for comparison and discuss the performance differences.

    The execution times and the speedup is saved in separate csv-files. They are named execution-time-xxx-*.

    The following image represents the speedup using MPI.



The following image represents the speedup comparison for OpenMP and MPI on a single node.



We can see that the speedup rises faster in the beginning when using OpenMP. However, as the number of processes increases even further, the MPI speedup excels the speedup of OpenMP.