# sheet 3

Simo Veijola, 1030419

November 11, 2024

# Introduction

In this exercise we transformed an existing, working, pure MPI parallelized program to hybrid MPI-OpenMP solution, and consider the pros and cons of both methods. It has to be noted, that the pros and cons presented here cannot necessarily be generalized to other programs trying to benefit from the hybrid model. This is because of the characteristics of our problem in hand, which is not the best possible candidate for benefiting from hybrid solution. The code has to be synchronized between iterations, and thus the outer-most loop going through the number of iterations cannot be parallelized. This also forces us to initialize the parallel region with OpenMP at each iteration, causing some overhead. I believe, if we were to parallelize an embarrassingly parallel program, requiring no communication and synchronization between iterations, we could possibly benefit more from the hybrid model and it would outperform the pure MPI model. For the problem in hand, I will first present my changes made to the pure MPI model, and the different solution I have tried out. Then I will discuss the results and compare the performance, memory usage, and scaling with the pure MPI solution.

# Program

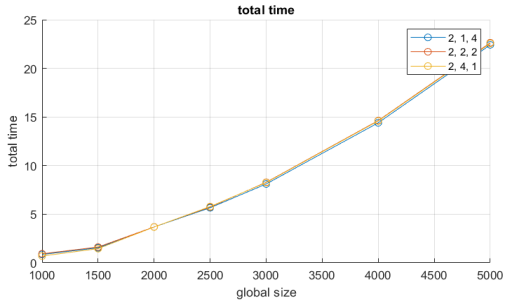
First, the necessary changes made to the program and the scripts. To make the program utilize openmp we of course have to include the 'omp' library, add the '-fopenmp' flag to the Makefile, and actually add the implementations utilizing OpenMP, be it through parallelized for loops or handing tasks to different threads. I chose to use parallelized for loops in my program, as I saw no reason to use tasking. Additionally, the cores used by OpenMP were fixed with $OMP\_PROC\_BIND = true$ and the number of threads were fixed with command $OMP\_NUM\_THREADS = \$SLURM\_CPUS\_PER\_TASK$.
My initial solution was to simply parallelize the heaviest part of the code, that is the interior calculations and leave the communication for one thread only. This solution performed worse than the pure MPI solution. There was also greater variance in the interior calculations utilizing OpenMP. To tackle these problems, I made simple improvements to the interior calculations and later also parallelized communication on all of the 4 sides of the region. In the interior, I pre-calculated the inverse of $dx$ and $dy$ and used multiplication instead of division, and calculated the indices used at the beginning of the first loop and then used simple linear incrementing, instead of recalculating the indices at each step. This reduced the discrepancy between calculation times of the MPI processes and made the hybrid solution to perform as well as the pure MPI solution.
Initially I used single threaded solution for the communication. For some parameters this performed worse than the pure MPI, so I tried to use fine-grained parallelization for the communication, and for handling the incoming data at each node. I chose a simple solution of one thread handling one of the sides for sending and receiving and handling the received data. This choice makes it optimal to use up to four threads with OpenMP. This change also required the usage of $MPI\_Init\_thread$ with level $MPI\_THREAD\_MULTIPLE$. Some benefit was seen from this, because most importantly, it allows us to handle the incoming data with mutliple threads at a time, where each thread waits on some incoming portion of data. We also do not have to utilize the exchange_finalize function, but we can instead wait for only the relevant portion of the requests and handle the ready ones first, reducing some overhead. This performed better than the pure MPI and previous hybrid solution for some input parameters, but also some weird behaviour was present for large data sizes.
For analysis, I used only one write to the file at the end of the iterations, which could affect the results. I did this to make computations generally faster so data can be collected faster.

# Results

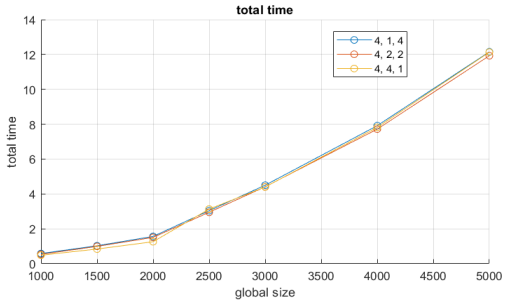
In this section I present the results with respect to the asked traits, that is performance, memory usage, and scaling. We begin with performance testing, comparing different MPI-OpenMP configurations.

## Performance

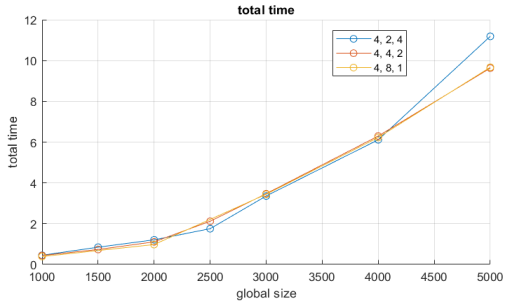
Initially tests were run with only the interior calculations parallelized with OpenMP. Then a few tests were run, where also communication and handling of the incoming data was parallelized utilizing OpenMP for larger data sizes.



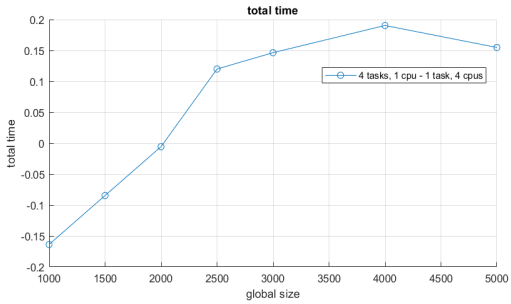

(a) Iteration times (seconds) for 2 nodes and total number of used cores of 4 per node.



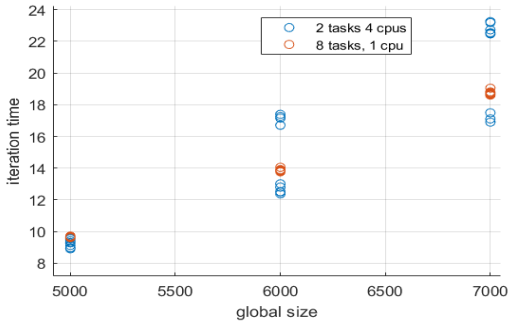

(b) Iteration times (seconds) for 4 nodes and total number of used cores of 4 per node.




(c) Iteration times (seconds) for 4 nodes and total number of used cores of 8 per node.




(d) Using 4 nodes, the difference between average times (seconds) with 4 tasks and 1 CPU per task and with 1 task and 4 CPUs per task




(e) Distribution of runtimes in seconds for pure MPI on red and hybrid solutions on blue.

Figure 1: Total iteration times for different node, node-wise task, and CPU counts utilizing either hybrid or full MPI solution. The curve legends tell the number of nodes, number of tasks per node, and number of CPUs per task in that order.

Figure 1 shows the performance of the modified program with different configurations. Tests were done utilizing 2 or 4 nodes, 2 to 8 tasks per node, and 0 to 4 CPUs per task. For each configuration, 2000 iterations were calculated, and the values presented are mean values of 10 separate runs collected into a .csv file, and visualized using MATLAB. From the figures it can be seen that in general the implementations perform rather similarly with almost identical development across different data sizes. The development follows the space complexity of the algorithm that is $O(n^2)$.

On the figure 1d, difference between run times for 4 nodes with either 4 tasks with 1 CPU or 1 task with 4 CPUs have been visualized. It shows that for larger data, the hybrid model perform slightly better. From the figure 1c, we can see that at data size $5000 \times 5000$ the configuration with 2 tasks per node and 4 CPUs per task performs around 1.5 seconds worse than the other hybrid solution, and the pure MPI solution, which is quite substantial.

Noting the reduction in the relative performance of the hybrid solution, I decided to also try parallelization of the communication and handling of the incoming data. I also utilized *nowait* for the loops handling the

interior calculations, so that the faster threads could possibly proceed first to handle the incoming data. For this method, I used sizes of 5000, 6000, and 7000 for both of the dimensions. As figure 1e shows, this method beat the pure MPI solution with being up to 1 second faster for size 5000. However, for larger sizes there is an interesting phenomenon, where each run using the hybrid model would essentially be either much worse than the pure MPI solution or slightly better than it as seen from the scatter plot. I am not sure why this happens for larger data sizes, and if this has something to do with which of the cores were allocated for OpenMP to use. However, this method has some potential of being faster than the pure MPI solution. This version of the program was returned along with this report.

## Memory usage

For memory usage, only pure MPI and hybrid configurations were tested against each other, pure MPI one with 4 nodes and 4 tasks with 1 CPU and the other one 4 nodes with 1 task with 4 CPUs. Both dimension sizes were set to 5000, adn the code was tested with both writing the files and without, to see what the memory consumption is outside of writing the results with MPI. Typical *seff* outputs for these configurations were as follows, where the first memory consumption is without writing the results to a file:

Pure MPI:

```
    Memory Utilized: 95.88 MB (estimated maximum)
Memory Efficiency: 1.20% of 7.81 GB (500.00 MB/core)
\\
Memory Utilized: 368.94 MB (estimated maximum)
Memory Efficiency: 4.61% of 7.81 GB (500.00 MB/core)
```

Hybrid:

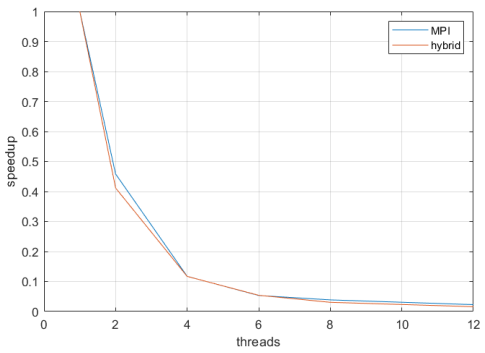
```
    Memory Utilized: 1.18 MB
Memory Efficiency: 0.01% of 7.81 GB
\\
Memory Utilized: 64.45 MB (estimated maximum)
Memory Efficiency: 0.81% of 7.81 GB (500.00 MB/core)
```

These show us, that the pure MPI solution uses 4 times the amount of memory here when we write the results to a file. Without writing to file, the full MPI solution uses even greater amount of memory when compared to the hybrid one. This greater memory usage is somewhat expected since the pure MPI solution has to do more communication than the hybrid solution. This is because the pure MPI has to utilize MPI communication also intranode, whereas the hybrid solution only has to use MPI communication for internode communication. For example, if we have local data of size $1000 \times 1000$, and we split it to $2 \times 2$ tasks of sizes $500 \times 500$, we do intranode communication for 1000 data points at each task. For hybrid solution with 4 threads per task and 1 task only, we get rid of the intranode communications, so in total the data to communicate using MPI is reduced by 4000 data points.

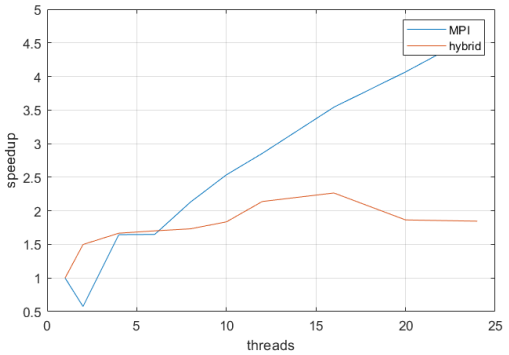
## Strong and weak scaling

For strong scaling we use fixed size for global data of size $6000 \times 6000$ and for weak scaling we use fixed size inside the nodes of $1000 \times 1000$ block per one thread utilized. Thus the global dimension size for weak scaling is 4 times 1000 times the number of tasks times threads per task. Since the nodes in our use have 2x12 cores per node, we plot the strong scaling for pure MPI with 4 nodes and 1 to 24 tasks (only count for which the data can be evenly distributed) with 1 thread, and for hybrid 4 nodes with 1 task and 1 to 24 threads per task. For the weak scaling we use the same configurations but scale the data size and go only up to 12 utilized threads. The baseline is taken with 4 nodes utilizing only one core and the scaling is not tested with various node counts for simplicity.

Figure 2 shows how both of the solution scale with respect to weak and strong scaling. It is quite evident that for small thread counts the hybrid solution performs slightly better than pure MPI solution, which may result from the difference in the amount of data to be communicated by each thread. For more than 4 thread utilized the pure MPI solution clearly scales better, however far from optimal as well. With respect to weak scaling, both methods perform in approximately equal manner. When inspecting the total times, the pure MPI solutions performs better, with the exception for around 4 threads utilized, which theoretically is the optimal for the used hybrid solution due to its implementation.




(a) Weak scaling of both the pure MPI and hybrid solutions.




(b) Strong scaling of both the pure MPI and hybrid solutions.

Figure 2: Weak and strong scaling as a function of total number of threads utilized in one node. To get the total number of cores the thread count by 4.

# Conclusion

The pure MPI methods outperforms the OpenMP with respect to scaling, but loses on the amount of memory used. Also the hybrid method used for the experiments performs well for 4 OpenMP threads, rooting from the internal structure of the solution. It has to be noted however, that initialization of many MPI tasks increases the initialization time, which is not reflected here in the running times. Thus in some scenarios the hybrid model could be seen as the better solution. Also, in the case where large data has to be handled, the OpenMP solution may be the more suitable choice due to consuming less memory. In such cases, we could initialize the maximum possible amount of tasks allowed by the memory constraint, and utilize 4 OpenMP threads inside the tasks.

The results for hybrid solution were somewhat a disappointment. However, there is also a problem with gaining full potential from the OpenMP here. The structure of the problem forces us to initialize a new parallel region at each iteration, which can cause significant overhead in the calculations. It would be interesting to perform similar experiments with a problem where each task can initialize only one parallel OpenMP region, and see how the hybrid version compares to pure MPI in that case. Such problems are most often however embarrassingly parallel.