Heat Diffusion: openMP and MPI

Steven Bronsveld
Wouter Damen
Kirsten Hagenaars
Bram Pulles

June 5, 2020

Contents

1 Hardware and software **2**

2 Testing **2**

3 Sequential **3**

4 OpenMP **5**

5 MPI **9**

6 Performance **13**

7 Task division **14**

1 Hardware and software

Table 1 shows the hardware specifications of the CPU from the computer that is used for getting test results. The computer further has 16 GB of ram and is running Arch Linux with kernel version 5.6.15-arch1-1. All of the programs are compiled with gcc version 10.1.0 and the following compiler flags: `-Wall -Wextra -Werror -pedantic -O3`.

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Address sizes:	39 bits physical, 48 bits virtual
CPU(s):	8
On-line CPU(s) list:	0-7
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	94
Model name:	Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
Stepping:	3
CPU MHz:	1200.022
CPU max MHz:	3500.0000
CPU min MHz:	800.0000
BogoMIPS:	5202.65
Virtualization:	VT-x
L1d cache:	128 KiB
L1i cache:	128 KiB
L2 cache:	1 MiB
L3 cache:	6 MiB
NUMA node0 CPU(s):	0-7

Table 1: Hardware specifications.

2 Testing

In order to gather performance information we use a tester made by one of the team members. The tester is written in Bash and supports a wide variety of options, including: minimum and maximum N and rank over which will be iterated, the epsilon value, the initial heat value, a timeout, the number of iterations to run, and the suite which we want to test (MPI, openMP, sequential or any combination of these). It further has a verbosity which can be set so we can produce human readable output or output in the CSV format so it is easy to process the test data and make graphs from them.

Lastly, we used a seperate Bash script to call the tester on a wide variety of configurations which automatically runs all of them, post processes the CSV

output further and puts the results in seperate files, this way we could run tests for hours without having to do anything manually.

3 Sequential

Unless stated otherwise, runtimes shown in figures have been acquired using $N = 100000$*,* `eps` $= 0.01$ *for the comparisons on* N *and* `eps` $= 0.001$ *for comparisons on heat, and* `HEAT` $= 100$*. Furthermore, every version was executed* 20 *times of which the average is shown in the figure with a deviation shown on the resulting line draw through all the average points.*

We have added a check for malloc failure to the program. After running valgrind on the program, which alerted us to the memory leak that was left by the heap-allocated vectors, we also added the freeing of the allocated memory at the end of the program.

We have also implemented the option of passing the values for N, EPS and HEAT as command-line arguments to the program, which allows us to test more easily and automatically. Parsing of the arguments is not included in the runtimes.

Since we came up with multiple possible improvements to the program, we decided to make multiple versions and compare these. Here is a list of these versions and how they differ from the original program (aside from what is mentioned above). Note that the openMP and the MPI versions use the same naming conventions.

- In the *split* version, the relaxation step and the stability check are done in separate loops. The stability check terminates once some `i` for which `(fabs(out[i] - in[i]) > eps)` holds is found.
- In the *split_zero* version, the relaxation step and the stability check are done in separate loops and the stability check is terminated early in the same way as this is done in *split*. Since the array resulting of the relaxation ends with some amount of zero's (possibly none) and we know where those zero's will start, we terminate the loop just before arriving at the zero's.
- In the *join* version, the relaxation step and stability check are combined into one loop. A variable `stable` is kept which is returned at the end to signify if the new array is stable. The `stable` variable is updated during every iteration of the relaxation for loop if it is not already declared unstable, `if (stable) stable = fabs(out[i] - in[i]) > eps`.
- In the *join_zero* version, the relaxation step and stability check are combined into one loop just like the *join* version. Similarly to *split_zero*, the loop terminates early since we know from which point on the remainder of the array is all zero's.

The test results for these two different versions are shown in figure 2 and 1. We also show the test results of the original version in figure 18 and 3.

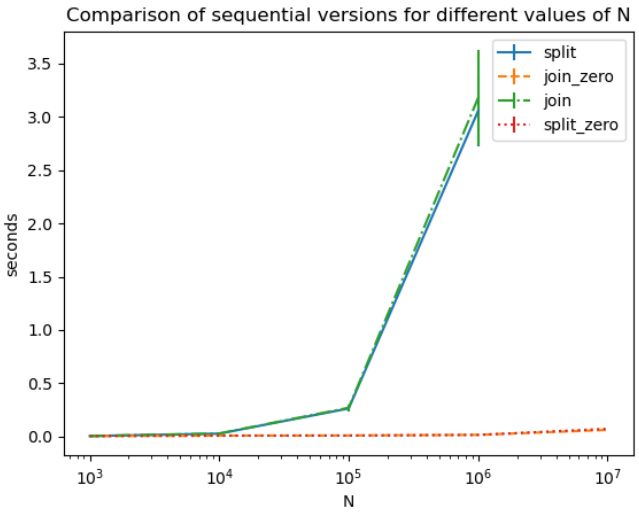

Figure 1: Optimized sequential programs.

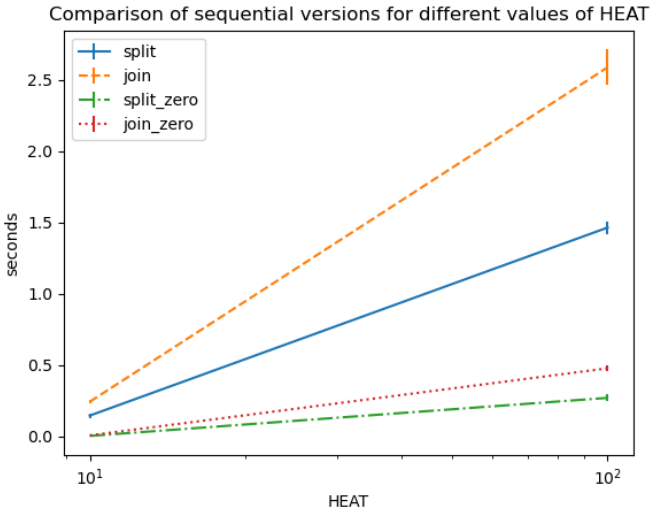

Figure 2: Optimized sequential programs.

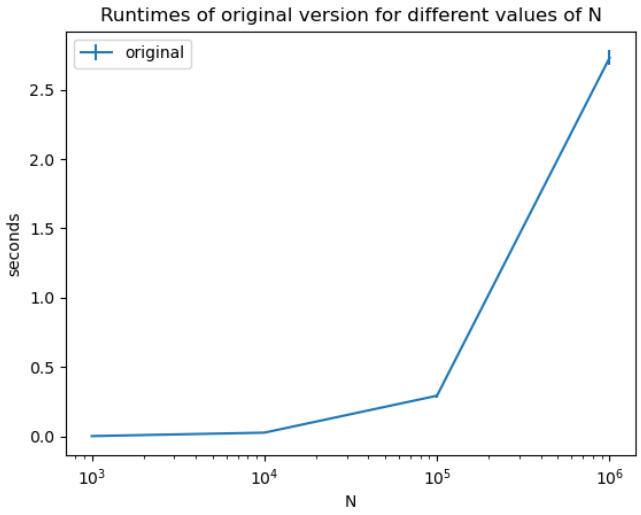

Figure 3: Original relax program.

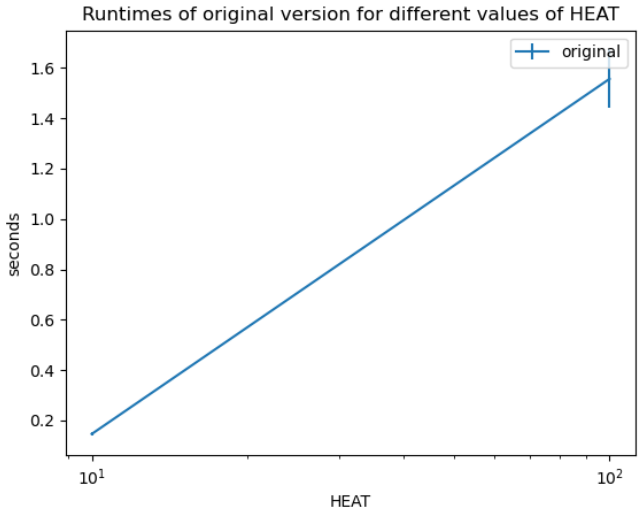

Figure 4: Original relax program.

4 OpenMP

For the openMP versions of the program, we have used `#pragma omp parallel for schedule(...)` for the loops in the functions `init` and `relaxAndStable`. To find out which scheduling strategy performes the best, we have made a comparison between the scheduling strategies, see figure 5 and 6. The results are shown in the following figure, note that the dynamic scheduling strategy is

not included since it often took more than 10 seconds, making it way slower than the others.

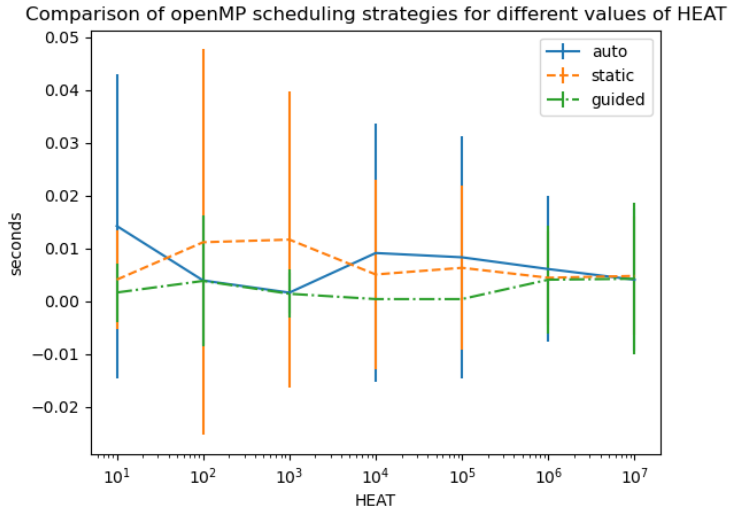

Figure 5: openMP scheduling strategy programs.

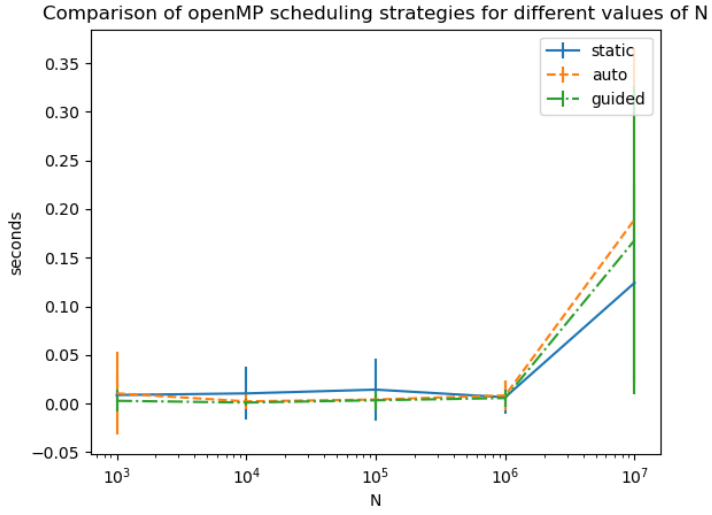

Figure 6: openMP scheduling strategy programs.

From figure 18 it looks like guided gives the best runtimes results, however figure 6 also suggests static will be faster with a big vector size. We have used the static stategy for the all of the openMP versions, since at the start of the project it was not clear yet which one was the fastest and static looked like the best candidate.

We have also tested if using openMP's reduction decreases runtimes. We found that this is not a good approach, because we want to stop searching once we find one instance of `fabs(out[i] - in[i]) <= eps` being false, which does not happen when using `reduction(&&: stable)`. Using openMP's support for for-loops in the `init` and `relaxAndStable` functions does achieve this. The difference between using and not using reduction is shown in figure 7 and 8. It is clear that using no reduction is way more efficient.

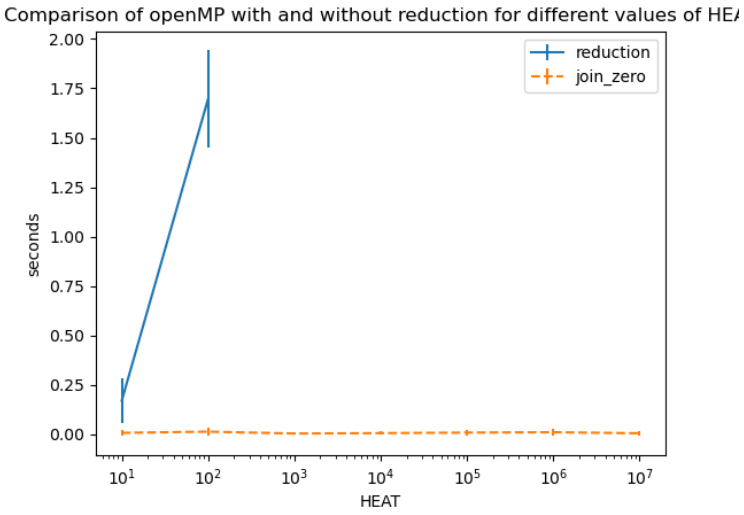

Figure 7: openMP program with/without reduction.

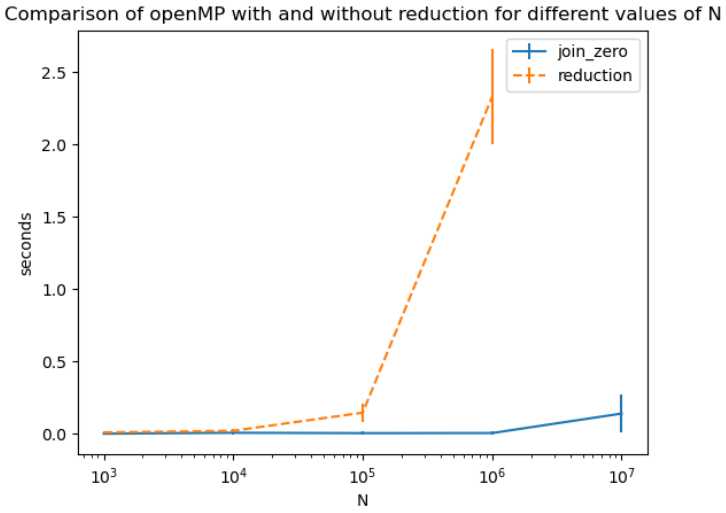

Figure 8: openMP program with/without reduction.

As stated before, we have made multiple optimized sequential versions. We have applied the openMP's support for loops on all of these versions to see which performes best when openMP is used. The following figure shows the results.

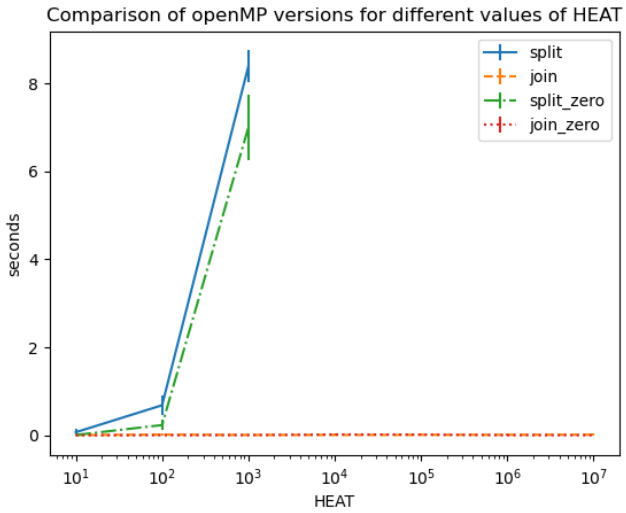

Figure 9: openMP versions vs HEAT

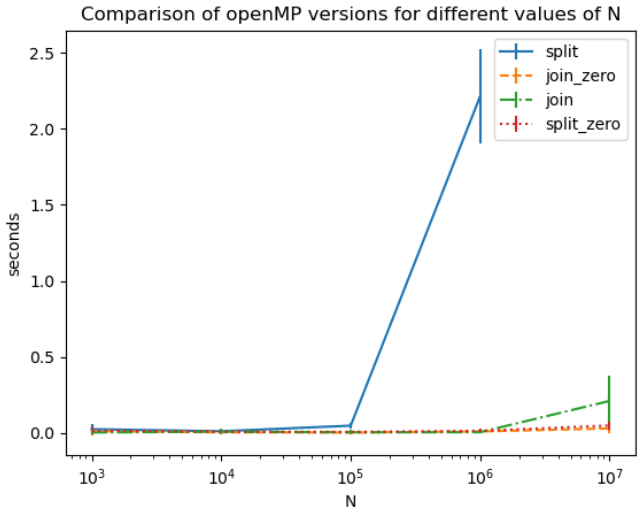

Figure 10: openMP versions vs N

Surprisingly, even though the *split* version performed faster in the sequential setting, the *joined* version performs faster in openMP.

5 MPI

For MPI we compared the performance of versions in which we use `MPI_Allgather` and `MPI_Allreduce` and of versions in which we manually communicate the results using `MPI_Send` and `MPI_Recv`. It was apparent that using `MPI_Allgather` for the heat array and `MPI_Allreduce` for the stability boolean is much faster than manual messages. The reason for this, as discussed in the lectures, is that the straightforward way to do manual messages causes one process to do all of the reduction work. Whereas the MPI reduction methods will automatically optimize the message structure to divide the reduction work across the processes as much as possible. The following eight figures compare the versions for different values of HEAT, N and the number of ranks

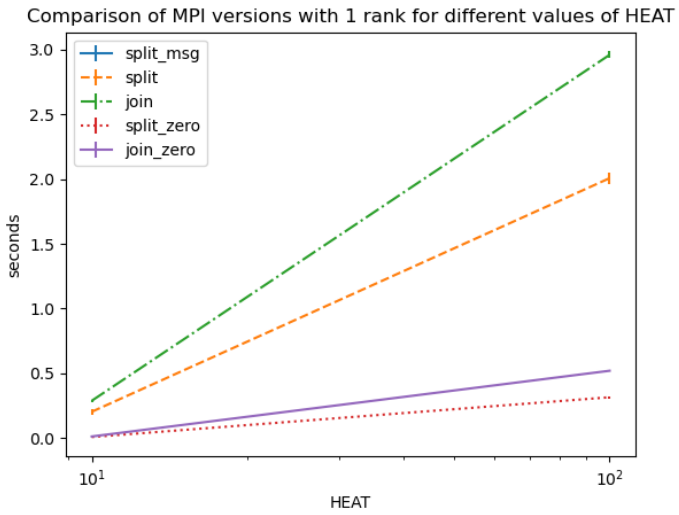

Figure 11: MPI versions vs HEAT on 1 rank

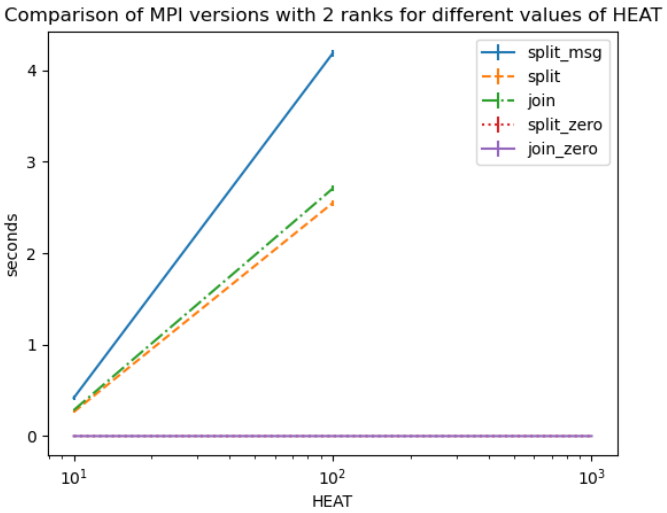

Figure 12: MPI versions vs HEAT on 2 ranks

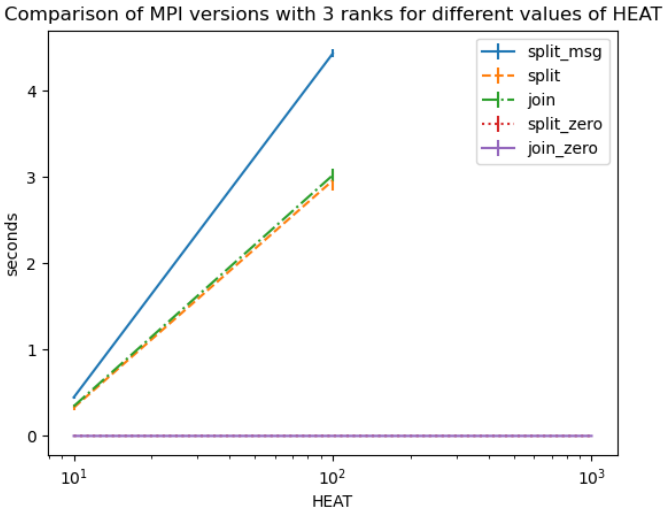

Figure 13: MPI versions vs HEAT on 3 ranks

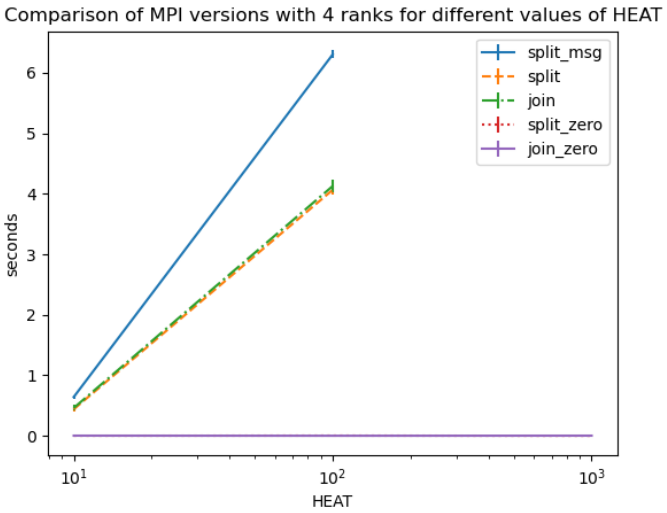

Figure 14: MPI versions vs HEAT on 4 ranks

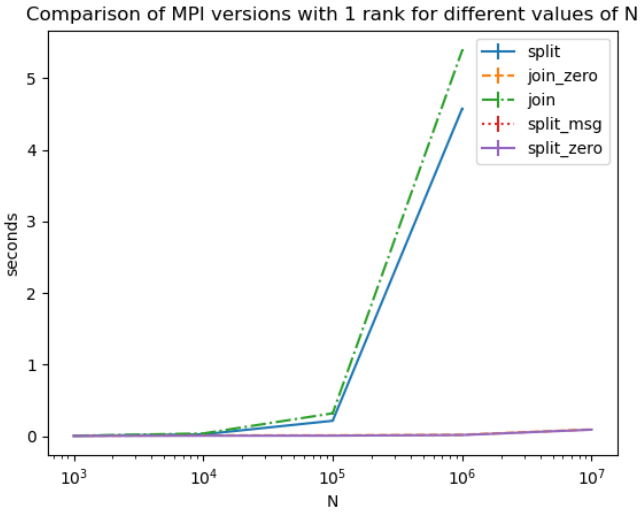

Figure 15: MPI versions vs HEAT on 1 rank

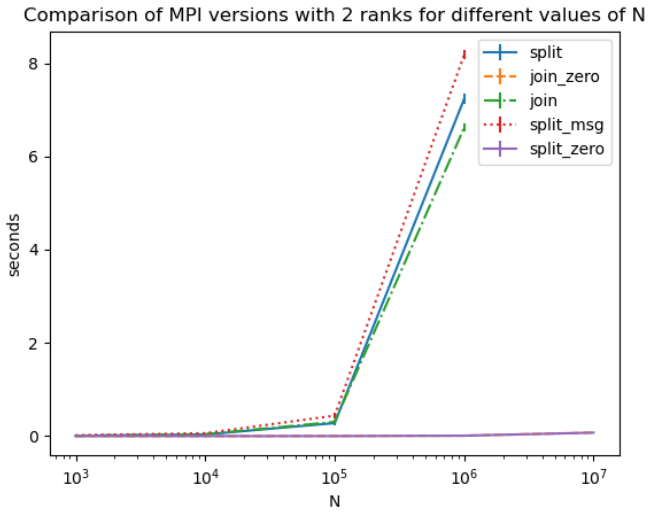

Figure 16: MPI versions vs HEAT on 2 ranks

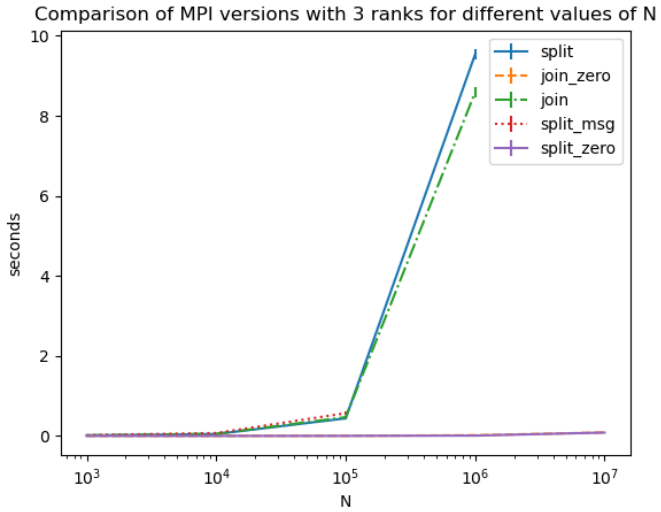

Figure 17: MPI versions vs HEAT on 3 ranks

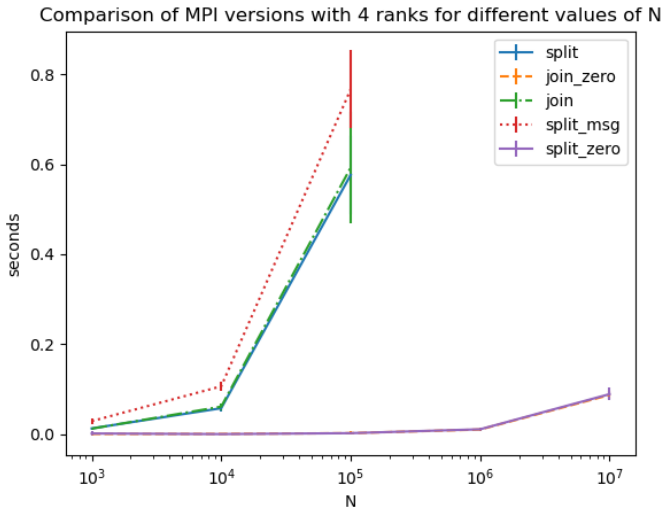

Figure 18: MPI versions vs HEAT on 4 ranks

6 Performance

We made a general observation regarding the size of the vector. Whenever we increase the vector size all of the non-zero versions become considerably slower, while the zero versions keep the same runtime, since they ignore a huge part of the vector, namely everything after the first zero. This optimization has a smaller impact when the heat value is increased or the epsilon value is decreased. Because in this case the zero versions have a big advantage at the start, when the heat is not diffused so far yet. However, the further the heat diffuses the smaller the advantage of the zero versions becomes. Thus, if the vector is relatively small the advantage of the zero versions is negligable.

The sequential algorithms are faster when split is used than when join is used. Because with split we can stop checking if the array is stable and return when we reached a point which is not below epsilon yet. With join we can stop computing the difference between the two arrays, however we still need to check if we already found a value which proves that the array is unstable. This means that join needs to do more work than the split versions, so the split versions are faster than the join versions.

The openMP algorithms are faster when join is used than when split is used. Because join can easily be run in parallel, since all the results are gathered into one `stable` variable which is returned at the end. But when using split checking if the array is stable cannot be run in parallel, since the for loop stops/returns whenever an unstable cell in the array is found. This means that the split version slows the openMP algorithms down a lot due to the sequential nature of the `isStable` function. So for openMP the join versions are way faster than the split versions.

For MPI, we conclude that on a small number of 4 cores, there is little to no performance gain. Likely because the overhead of sending messages and sharing results between all ranks nullifies the performance gain from spreading the work

across the cores. We do however have strong scalability for these algorithms, since the array of 10^n indexes can easily be divided across however many ranks there are. So we would expect that MPI runs fastest on such large systems.

Comparing all versions, we conclude that openMP, the *join_zero* version, performs best on the one-system setups we tested it on. This is to be expected, since openMP was really designed for this kind of environment, whereas MPI was designed for multi-system setups.

7 Task division

Here we give an overview of the task division between the members of our group.

- **Steven Bronsveld and Wouter Damen** worked on the MPI code. They first planned out what optimizations where possible, and what different MPI-techniques where. Then they devided the work such that Wouter made the programm using the different forms of gather, and Steven used MPI-messages.
- **Kirsten Hagenaars and Bram Pulles** worked on the openMP code. Bram further made the testing setup, the Bash scripts, and the Makefiles to compile all of the code. Kirsten additionally made all of the plots in this report using Python.
- **All**: We frequently met each other online to discuss our findings and the progress we made on the project. Each one of us wrote the section in the report corresponding to the code on which he/she has worked. We all worked on improving the sequential version before we splitted into two groups and dived into openMP and MPI seperately. The decision to split into pairs was made to reduce the overhead of communication when working on the individual sections. We also shared ideas for optimizations and gave each other mental support.