Parallel Computing
Assignment 1

Onno de Gouw
Laura Kolijn
Stefan Popa
Denise Verbakel

July 12, 2020

Table of Contents

Hardware and Software Specification

Before we start of with discussing the different tasks of this assignment, we will specify the hardware and software that we used in order to test and run our programs. Here, VM is used as an abbreviation for Virtual Machine.

Hardware Onno de Gouw:

- CPU: i7-6700
- Number of cores (on VM): 4
- Clock Frequency: 3.40GHz
- Cache Memory: 8MB Cache
- RAM (on VM): 4GB RAM DDR4
- RAM frequency: 2133MHz
- Other: Kali Linux 2020.2, VirtualBox
- Compiler Version: GCC 9.3.0
- MPICH Compiler Version: 3.3.2

Hardware Stefan Popa:

- CPU: i5-4570
- Number of cores (on VM): 4
- Clock Frequency: 3.20GHz
- Cache Memory: 6MB Cache
- RAM (on VM): 8GB RAM DDR3
- RAM frequency: 1333MHz
- Other: Ubuntu 20.04, VirtualBox
- Compiler Version: GCC 9.3.0
- MPICH Compiler Version: 3.3.2

Hardware Laura Kolijn:

- CPU: i5-6200U
- Number of cores: 4
- Clock Frequency: 2.30GHz
- Cache Memory: 3MB Cache
- RAM: 16GB RAM DDR4
- RAM frequency: 2400MHz
- Other: Ubuntu 18.04.4 LTS, Dual Boot
- Compiler Version: GCC 7.5.0
- MPICH Compiler Version: 3.3a2

Hardware Denise Verbakel:

- CPU: i7-8750H
- Number of cores (on VM): 4
- Clock Frequency: 2.21GHz
- Cache Memory: 9MB Cache
- RAM (on VM): 2GB RAM DDR4
- RAM frequency: 2666MHz
- Other: Kali Linux 2019.3, VirtualBox
- GCC Compiler Version: GCC 9.3.0
- MPICH Compiler Version: 3.3.2

We will highlight some of the observations that we can make from these specs.
As can be seen above, half of the machines has an i7 processor and the other two have an i5 processor. Next to this, we are all using a machine with 4 cores. This is going to be important in Task 3: MPI. Another thing that can be seen is that three out of four testing environments are within a Virtual Machine. Finally, three out of four machines are using the same compiler versions for both GCC and MPICH: GCC version 9.3.0 and MPICH version 3.3.2. Besides these specifications, we also used compiler flags in order to run our program. The commands we used (so including the compiler flags) are:

- For the sequential version: `gcc -O3 -o relax relax.c`
- For the OpenMP version: `gcc -O3 -fopenmp -o openmp openmp.c`
- For the MPI version: `mpicc -Wall -g -O3 -o mpi mpi.c`

Details about these commands, such as why using `-O3`, will be explained in the appropriate sections.

Task 1: Sequential Evaluation

Choices Performance Evaluation

First, we will evaluate the performance of the sequential code that was provided with this project. We made sure that we were using the highest level of compiler optimisation on our machines (`-O3`). We ran the sequential program on different computers having different types of hardware and software and we will present runtime results in the form of a table followed by a graph.

We chose to place the wallclock timer around the do-while loop (the workloop) and thus we did not include the allocation and the freeing of memory. We made this choice for two (main) reasons: (1) these two steps are not really relevant for the runtime of the overall algorithm and (2) the comparison between the OpenMP and MPI versions gets a lot easier this way. We used the wallclock timer that was provided in slideset 3 (MPI Basics).

For all tasks, we decided to use two `EPS-HEAT` value pairs and for each of those pairs we used multiple values for `N`. The two `EPS-HEAT` value pairs that we used, are `EPS = 0.01` in combination with `HEAT = 150.0` and `EPS = 0.0005` in combination with `HEAT = 400.0`. The reason we chose those two value pairs, is that for the first pair the OpenMP version runs (partially) faster than the sequential version and for the second pair the MPI version runs (partially) faster.
The `N` values that we decided on for the first pair, range from 100,000 up to and including 20,000,000.
The `N` values that we decided on for the second pair, range from 1,000 up to and including 500,000.
We chose these values to have an interval of runtimes that ranges approximately between 0.1 seconds and 4 minutes (and thus have a reasonable range of runtimes regarding the total time needed to test everything).

Next to this, we also made the choice to use valgrind instead of gprof and/or gperftools. Why we did this, is because we all had a little bit of experience with using valgrind already as opposed to the other two tools that were mentioned. The results that we obtained when running valgrind for the sequential version can be found below:

```
==2467== Memcheck, a memory error detector
==2467== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2467== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2467== Command: ./relax
==2467==
size   : 0 M (0 MB)
heat   : 150.000000
epsilon: 0.010000
Number of iterations: 3630
Total time: 0.199997 seconds
==2467==
==2467== HEAP SUMMARY:
==2467==     in use at exit: 0 bytes in 0 blocks
==2467==   total heap usage: 3 allocs, 3 frees, 17,024 bytes allocated
==2467==
==2467== All heap blocks were freed -- no leaks are possible
==2467==
==2467== For lists of detected and suppressed errors, rerun with: -s
==2467== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As can be seen, valgrind showed that all heap blocks were freed - so no leaks are possible - and no other errors were found. We can thus conclude that we have a memory-leak free sequential version.

Implementation Process

For task 1, we went through a process that we will describe now.

When we started improving the given sequential version (`relax1.c`), the first thing that we noticed was that it was possible to combine the functions `relax()` and `isStable()`. This version we described in the file `relax2.c` and the changed code looks like the following:

```
bool optimized(double *in, double *out, int n, double eps)
{
   int i;
   bool res = true;

   for(i=1; i<n-1; i++) {
      out[i] = 0.25*in[i-1] + 0.5*in[i] + 0.25*in[i+1];
      res = res && (fabs(in[i] - out[i]) <= eps);
   }
   return res;
}
```

However, when we implemented this function, we realized that the function `relax()` was called one time too often, slowing the program down. Next to this, we realized that the `isStable()` function was running longer than needed because it always goes through the complete for-loop. Therefore, we realized that we should split the functions again and make `isStable()` stop earlier. In `relax3.c`, we kept the same code as provided for `relax()` and changed `isStable()` to the following code:

```
bool isStable(double *old, double *new, int n, double eps)
{
   int i;

   for(i=1; i<n-1; i++) {
      if (fabs(old[i] - new[i]) > eps)
         return false;
   }
   return true;
}
```

As can be seen, we decided to immediately stop the for-loop when one value is not stable (since then the whole array is automatically not stable). After this, we dived deeper into the code: we changed `i++` to `++i` since `++i` can sometimes result in faster computation. Next to this, we also changed `malloc` into `calloc` and added a check if we are actually given the requested memory. Note that the last improvement is not connected to our wallclock runtime, but just makes the program less error-prone as can be seen from the valgrind run. This change can be found in our final version called `relax.c` and can also be found in `relax3.c`. The changed code of `malloc` into `calloc` looks like this:

```
double *allocVector(int n)
{
   double *v;
   v = (double *)calloc(n, sizeof(double));
   if (v == NULL)
      exit(-1);
   v[0] = HEAT;
   return v;
}
```

After this, we had another idea, which can be seen in `relax4.c`: what the code does now is relaxing a lot of redundant indices, since it will just relax 0 over and over again (which will always result in 0). We made a new variable called `bound`, which indicated until which point the vector needed to be relaxed.

The do-while loop now looks like the following:

```
do {
    if (bound < n)
        bound ++;
    tmp = a;
    a = b;
    b = tmp;
    relax(a, b, bound);
    iterations ++;
} while(!isStable(a, b, bound, EPS));
```

This new implementation had a huge speedup, but it is based on the assumption that the `HEAT` value is always placed in the beginning. In the tutorial it has been said that it should not have this assumption, but we decided to keep this version inside our implementation process folder. Our final version that we will use for testing, will be the version without this assumption.

In our `Task1` directory you will find the `Implementation_Process_Sequential`, which holds our old versions of the code. The final and tested version will be found in the file `relax.c`.

The runtime results that we obtained, broken down per person, can be found in the sections below.

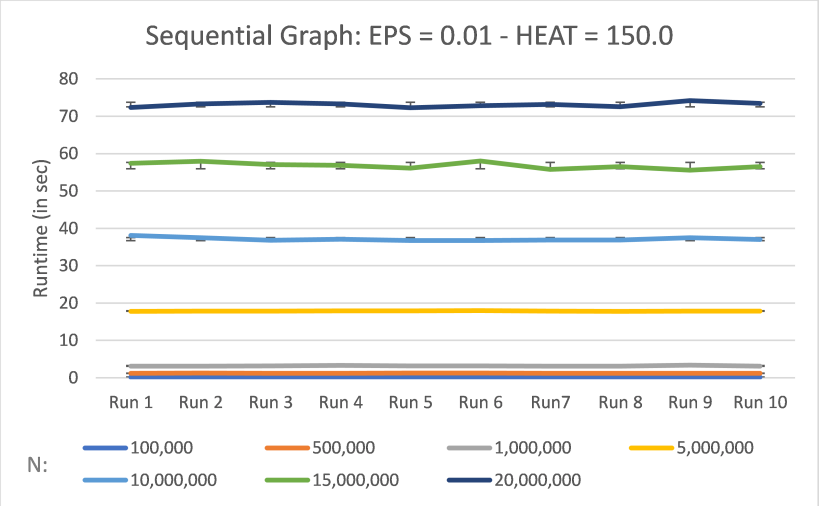
Results of Onno de Gouw

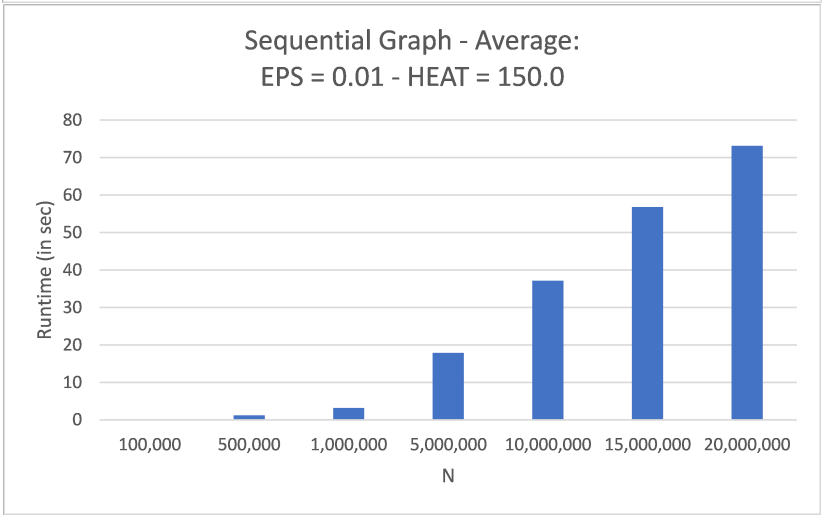
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.200745	0.211086	0.207481	0.199023	0.197791	0.193057	0.193246
500,000	1.178012	1.232141	1.201301	1.206321	1.228479	1.231481	1.200763
1,000,000	3.096323	3.099636	3.148132	3.316196	3.157435	3.147619	3.119028
5,000,000	17.746934	17.867936	17.849433	17.880813	17.895231	17.968412	17.820488
10,000,000	38.069043	37.461691	36.819963	37.091198	36.743661	36.706622	36.867783
15,000,000	57.436956	57.943592	57.100732	56.862846	56.127376	58.004707	55.802264
20,000,000	72.351175	73.286611	73.723068	73.338429	72.310591	72.854286	73.167569

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.197957	0.210638	0.197618	0.2008642	0.211086	0.193057
500,000	1.217563	1.190263	1.207724	1.2094048	1.232141	1.178012
1,000,000	3.081459	3.344521	3.110206	3.1620555	3.344521	3.081459
5,000,000	17.776891	17.842041	17.834863	17.8483042	17.968412	17.746934
10,000,000	36.842534	37.461367	37.021431	37.1085293	38.069043	36.706622
15,000,000	56.557108	55.589412	56.537224	56.7962217	58.004707	55.589412
20,000,000	72.523541	74.162982	73.440785	73.1159037	74.162982	72.310591

The graphs below will display the results just obtained:



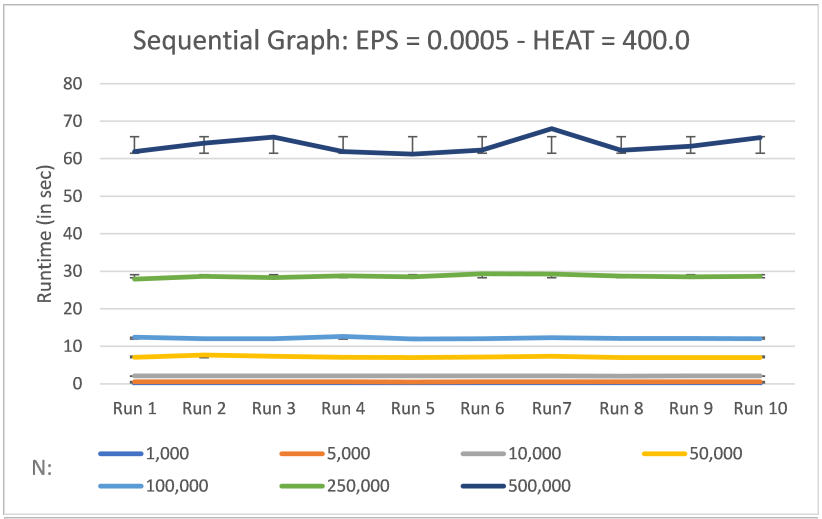


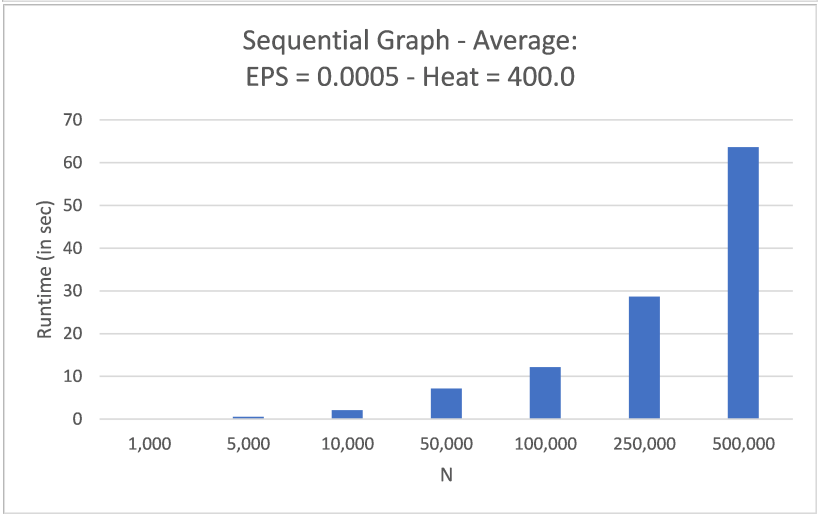
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.091641	0.091631	0.093199	0.091871	0.092647	0.092409	0.093698
5,000	0.568791	0.573669	0.574899	0.577408	0.567624	0.577417	0.569811
10,000	2.091776	2.143672	2.086691	2.082453	2.078141	2.090979	2.098098
50,000	7.066034	7.676889	7.318871	7.053262	7.020621	7.150919	7.334353
100,000	12.458484	12.032244	12.050647	12.636608	11.977141	12.039609	12.277469
250,000	27.932434	28.646545	28.291084	28.767423	28.492058	29.305571	29.247455
500,000	61.879528	64.136043	65.783624	61.920341	61.229354	62.329377	68.001748

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.092962	0.094446	0.092961	0.0927465	0.094446	0.091631
5,000	0.578739	0.571663	0.569979	0.5730000	0.578739	0.567624
10,000	2.073651	2.087311	2.090233	2.0923005	2.143672	2.073651
50,000	6.989314	6.969801	7.008538	7.1588602	7.676889	6.969801
100,000	12.097203	12.062151	12.047446	12.1679002	12.636608	11.977141
250,000	28.743507	28.482333	28.678174	28.6586584	29.305571	27.932434
500,000	62.224736	63.350905	65.652538	63.6508194	68.001748	61.229354

The graphs below will display the results just obtained:





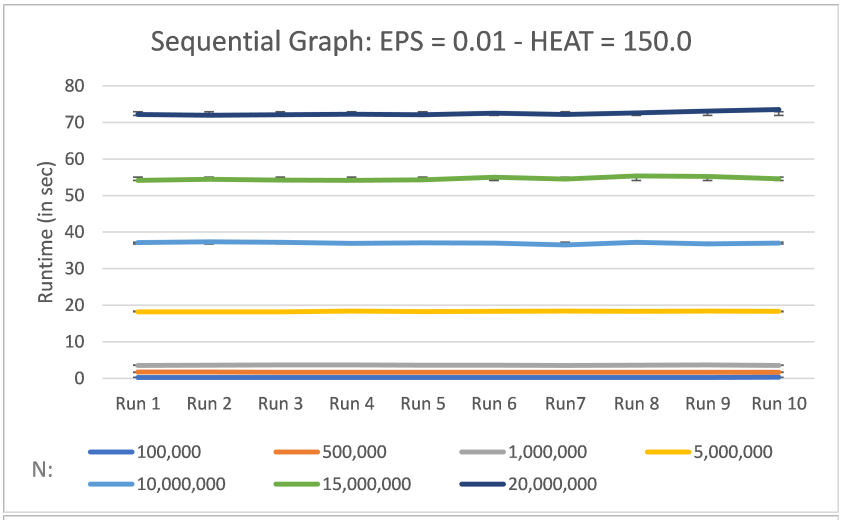
Results of Laura Kolijn

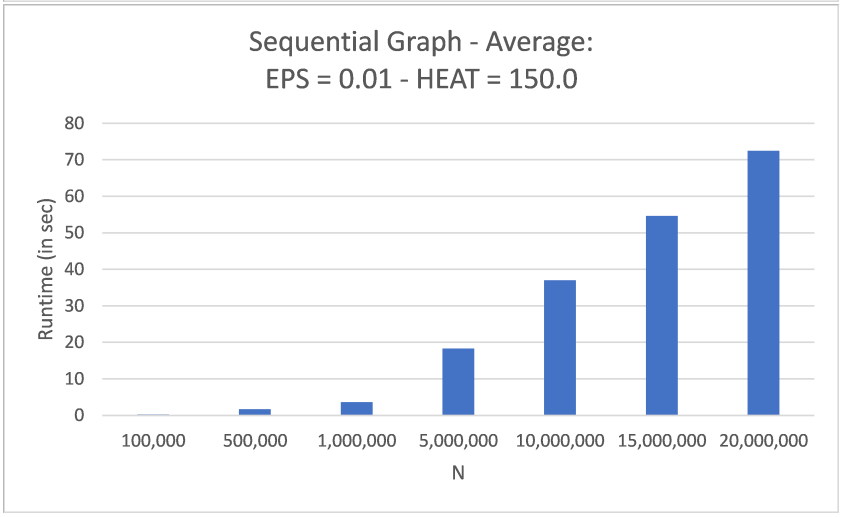
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.272616	0.270106	0.255337	0.256069	0.254615	0.281845	0.257573
500,000	1.735822	1.719991	1.697318	1.701836	1.688987	1.680483	1.683910
1,000,000	3.561564	3.578044	3.645522	3.713363	3.576726	3.590299	3.526105
5,000,000	18.225007	18.225856	18.161912	18.372262	18.278441	18.317011	18.391398
10,000,000	37.171691	37.362801	37.188496	36.932814	37.049182	37.009263	36.479260
15,000,000	54.142264	54.459213	54.216043	54.160875	54.267320	55.017862	54.538413
20,000,000	72.143560	71.958241	72.095684	72.208537	72.082880	72.545068	72.141681

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.260116	0.256131	0.288988	0.2653396	0.288988	0.254615
500,000	1.689968	1.698391	1.693216	1.6989922	1.735822	1.680483
1,000,000	3.617740	3.714553	3.570903	3.6094819	3.714553	3.526105
5,000,000	18.358187	18.410231	18.335458	18.3075760	18.410231	18.161912
10,000,000	37.198810	36.756817	36.992167	37.0141301	37.362801	36.479260
15,000,000	55.371953	55.246278	54.596396	54.6016617	55.371953	54.142264
20,000,000	72.592867	73.073262	73.515617	72.4357397	73.515617	71.958241

The graphs below will display the results just obtained:



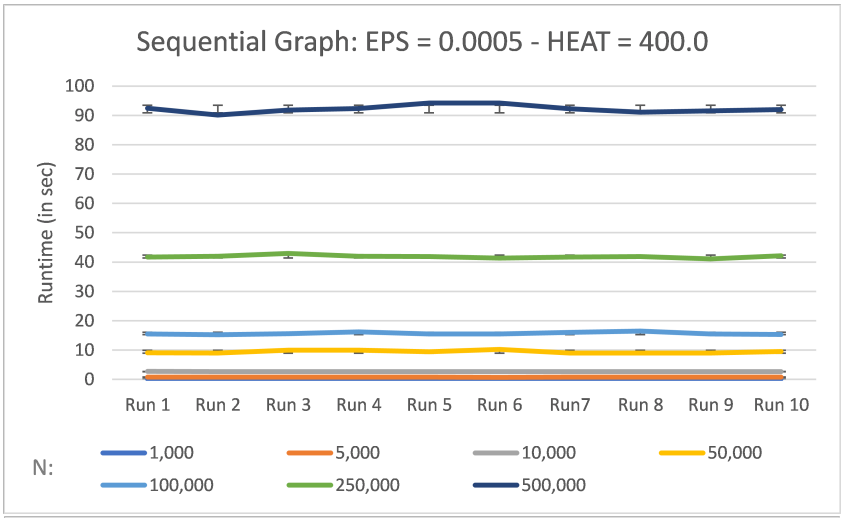


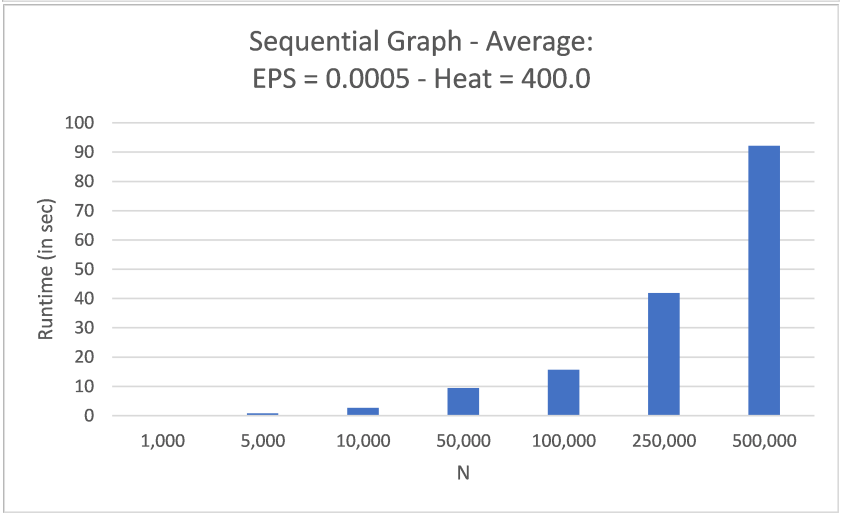
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.135648	0.135761	0.136282	0.166584	0.137409	0.135598	0.138996
5,000	0.770171	0.795764	0.793271	0.774273	0.772894	0.764824	0.794993
10,000	2.748546	2.680173	2.672331	2.681959	2.680818	2.678476	2.686541
50,000	9.065931	8.946123	9.979761	9.937966	9.462611	10.253714	8.992772
100,000	15.524515	15.185789	15.611032	16.230189	15.517269	15.467715	16.035695
250,000	41.670923	41.989114	42.970174	41.943437	41.895842	41.346948	41.722743
500,000	92.418570	90.067414	91.763600	92.349239	94.173571	94.214789	92.238765

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.136149	0.163849	0.139342	0.1425618	0.166584	0.135598
5,000	0.767366	0.790554	0.785283	0.7809393	0.795764	0.764824
10,000	2.675422	2.687867	2.689111	2.6881244	2.748546	2.672331
50,000	9.027680	8.975854	9.474479	9.4116891	10.253714	8.946123
100,000	16.435934	15.482963	15.349142	15.6840243	16.230189	15.185789
250,000	41.874638	41.097219	42.142438	41.8653476	42.970174	41.346948
500,000	91.056991	91.522261	91.965267	92.1770467	94.214789	90.067414

The graphs below will display the results just obtained:





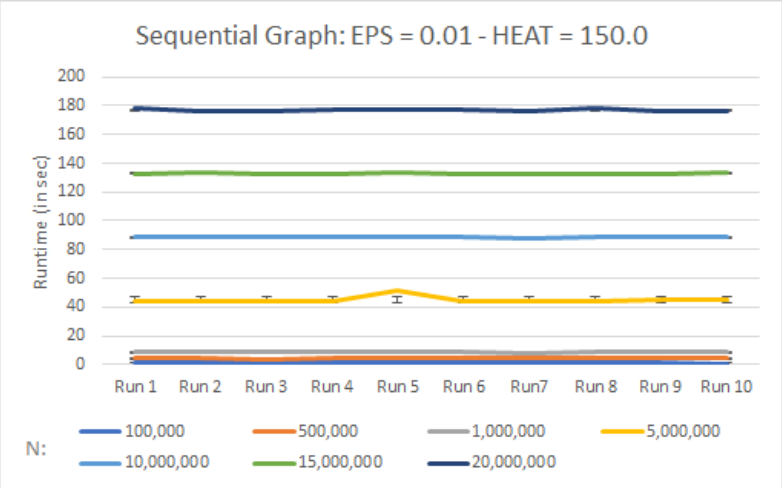
Results of Stefan Popa

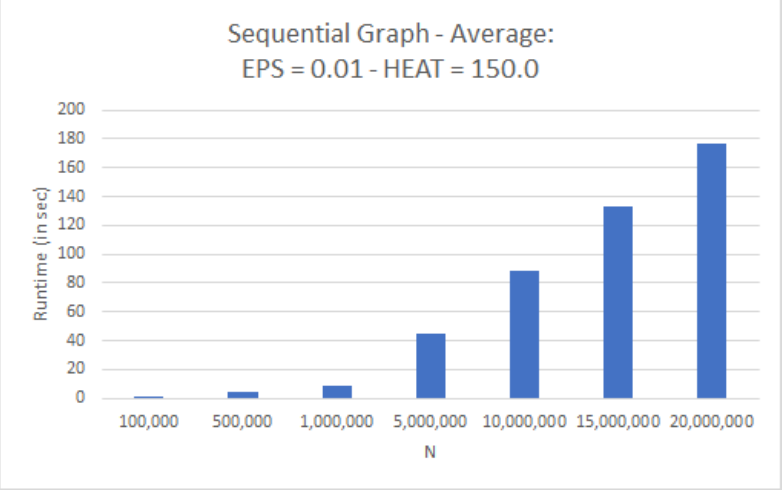
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.899194	0.897822	0.894323	0.903135	0.911935	0.915605	0.895053
500,000	4.412372	4.442161	4.378756	4.394377	4.398971	4.421614	4.414630
1,000,000	8.766392	8.742477	8.716901	8.704465	8.707313	8.696686	8.617588
5,000,000	43.690728	44.234013	44.099087	44.531489	51.851468	43.963039	44.300966
10,000,000	88.626643	88.594249	88.492735	88.573866	88.641489	88.689301	87.802646
15,000,000	132.871866	133.529251	132.841636	132.529273	133.102541	132.872553	132.876351
20,000,000	178.012247	176.414586	176.659486	177.312137	177.223136	176.891127	176.645331

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.903726	0.886656	0.884781	0.8992230	0.915605	0.884781
500,000	4.412280	4.436573	4.457921	4.4169655	4.457921	4.378756
1,000,000	8.638150	8.719157	9.220955	8.7530084	9.220955	8.617588
5,000,000	44.244252	45.086447	44.654304	45.0655793	51.851468	43.690728
10,000,000	88.653724	88.653724	88.532755	88.5261132	88.689301	87.802646
15,000,000	132.876452	132.855291	133.271866	132.9627080	133.529251	132.529273
20,000,000	178.113134	176.641269	176.721224	177.0633677	178.113134	176.414586

The graphs below will display the results just obtained:



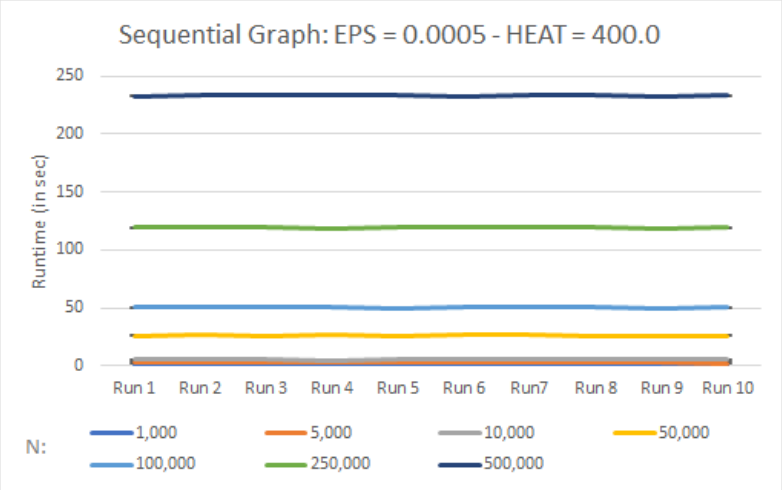


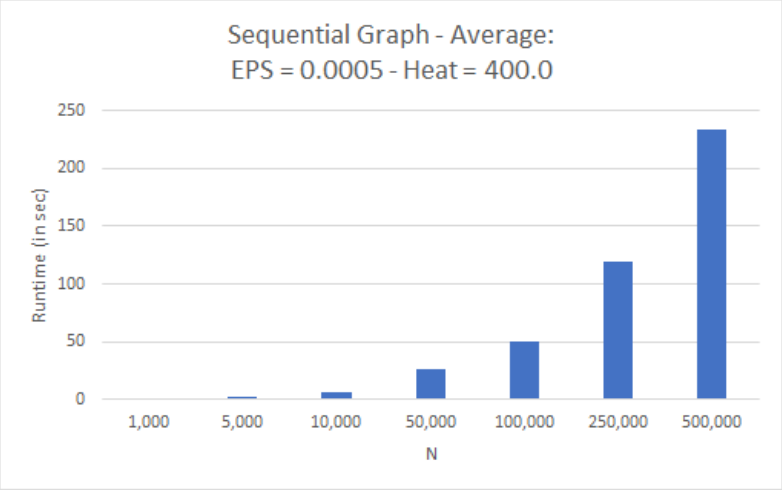
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.568161	0.563993	0.569159	0.569033	0.575001	0.573918	0.572911
5,000	2.638266	2.647428	2.650159	2.713366	2.651392	2.653093	2.643995
10,000	6.186123	6.171961	6.182943	6.103138	6.182335	6.141250	6.184718
50,000	25.922578	26.351510	26.207797	26.387605	26.176104	26.695875	26.264188
100,000	50.409110	50.229694	50.205840	50.501121	49.855136	50.227491	51.359346
250,000	119.208973	119.15175	119.518165	118.962975	120.304969	120.114916	119.324819
500,000	232.902305	233.163301	233.384158	234.152459	233.706927	232.919692	233.054309

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.572832	0.580036	0.581211	0.5726255	0.581211	0.563993
5,000	2.647629	2.656502	2.627977	2.6529807	2.713366	2.627977
10,000	6.115955	6.132909	6.222775	6.1624107	6.222775	6.103138
50,000	25.965318	26.171991	26.208127	26.2351093	26.695875	25.922578
100,000	50.148624	49.808882	50.278288	50.3023532	51.359346	49.808882
250,000	119.958568	118.575053	119.553793	119.4673981	120.304969	118.575053
500,000	233.459062	232.728107	234.001664	233.3471984	234.152459	232.728107

The graphs below will display the results just obtained:





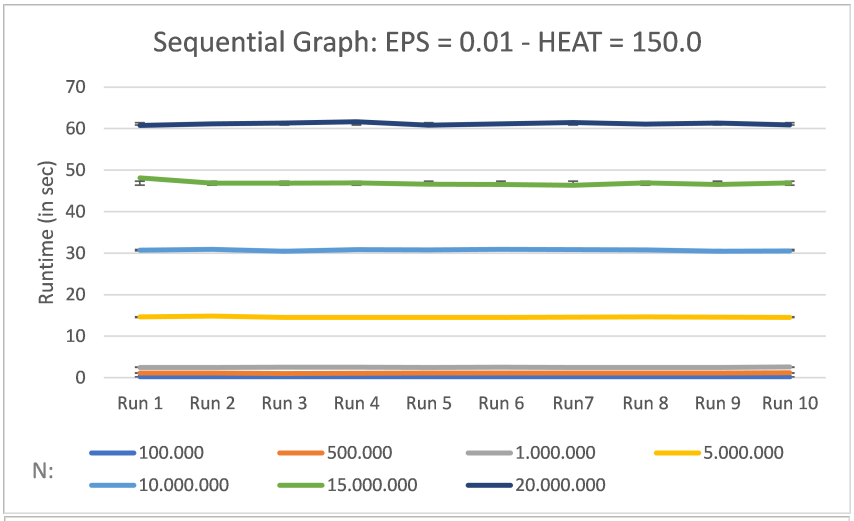
Results of Denise Verbakel

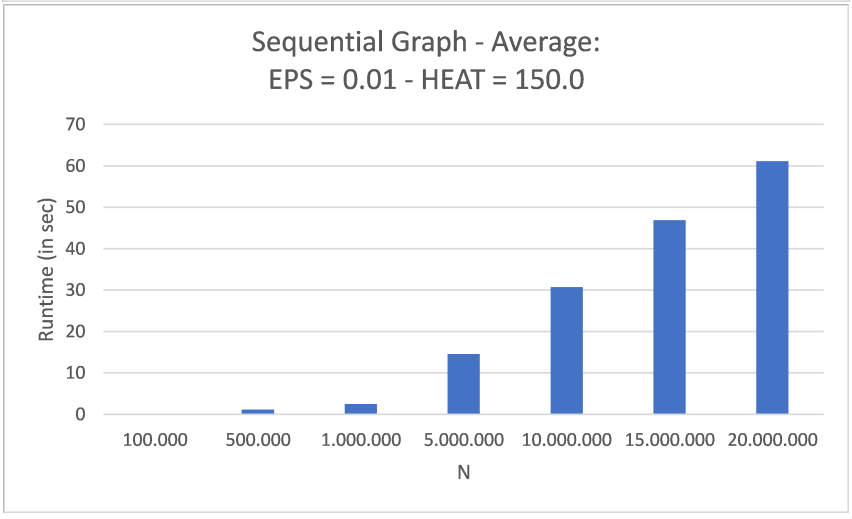
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.200745	0.211086	0.207481	0.199023	0.197791	0.193057	0.193246
500,000	1.122797	1.111724	1.024878	1.044058	1.121961	1.132165	1.113628
1,000,000	2.472943	2.484673	2.532124	2.528871	2.493324	2.512386	2.482265
5,000,000	14.618164	14.835464	14.530003	14.503219	14.513245	14.502395	14.585213
10,000,000	30.735053	30.929305	30.474932	30.816665	30.788866	30.879939	30.823787
15,000,000	48.115924	46.864984	46.842850	46.911528	46.617424	46.547076	46.358532
20,000,000	60.733339	61.132243	61.305400	61.675308	60.795968	61.156519	61.452330

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.197957	0.210638	0.197618	0.2008642	0.211086	0.193057
500,000	1.151379	1.132569	1.179969	1.1135128	1.179969	1.024878
1,000,000	2.472206	2.491686	2.582486	2.5052964	2.582486	2.472206
5,000,000	14.626270	14.557759	14.491640	14.5763372	14.835464	14.491640
10,000,000	30.778082	30.431444	30.499920	30.7157993	30.929305	30.431444
15,000,000	46.890156	46.541423	46.946496	46.8636393	48.115924	46.358532
20,000,000	61.082831	61.298811	60.890420	61.1523169	61.675308	60.733339

The graphs below will display the results just obtained:



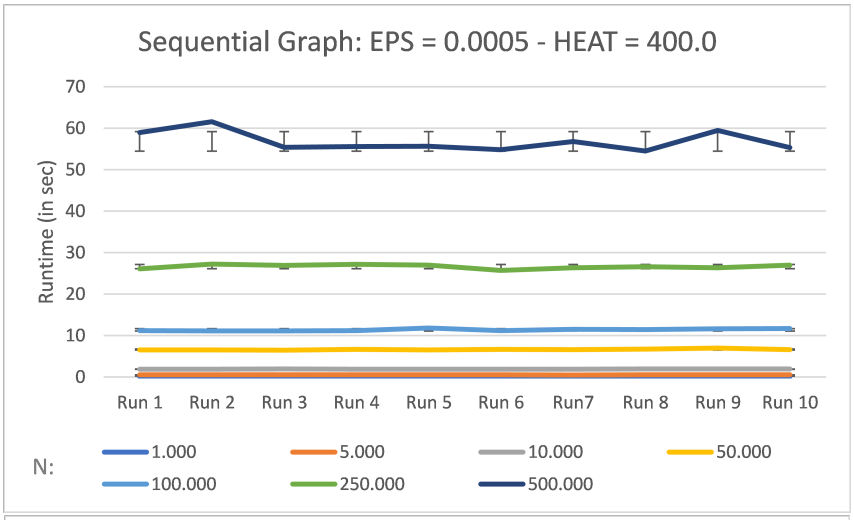


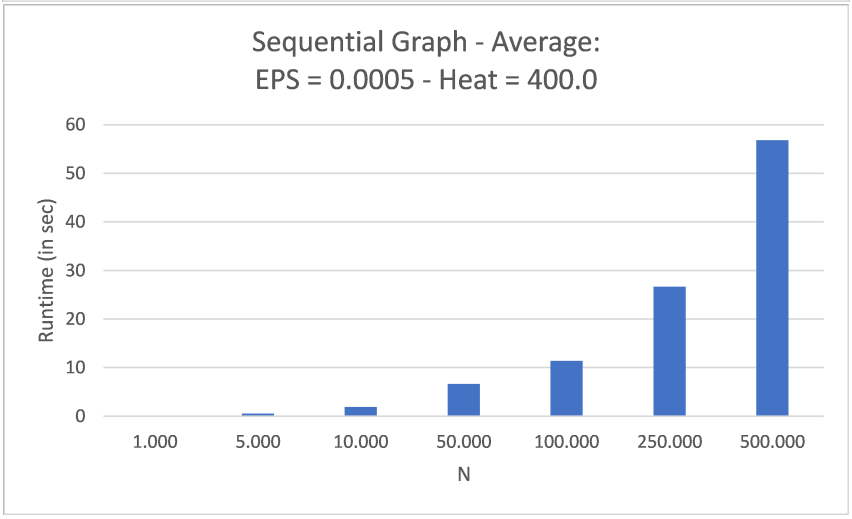
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.086497	0.086135	0.086655	0.083933	0.086028	0.091014	0.087857
5,000	0.536705	0.543556	0.531963	0.541627	0.547345	0.535978	0.529439
10,000	1.913687	1.916437	1.940009	1.893565	1.908564	1.900233	1.892854
50,000	6.566140	6.544213	6.483787	6.687127	6.517935	6.684091	6.590419
100,000	11.182851	11.112432	11.130479	11.166519	11.853149	11.158372	11.476866
250,000	26.095571	27.210850	26.943160	27.202807	26.964066	25.725792	26.357813
500,000	58.979415	61.555500	55.382536	55.618009	55.664956	54.840370	56.779658

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.087361	0.086158	0.086803	0.8684410	0.091014	0.083933
5,000	0.533817	0.539447	0.534089	0.5373966	0.547345	0.529439
10,000	1.934198	1.959802	1.925347	1.9184696	1.959802	1.892854
50,000	6.716493	6.982491	6.629571	6.6402267	6.982491	6.483787
100,000	11.454065	11.615051	11.689461	11.3839245	11.853149	11.112432
250,000	26.606640	26.341501	26.987449	26.6435649	27.210850	25.725792
500,000	54.503440	59.495937	55.341060	56.8160881	61.555500	54.503440

The graphs below will display the results just obtained:





Task 2: OpenMP

Choices Performance Evaluation

After having evaluated the performance of the sequential code, we continued with implementing and evaluating the performance of an OpenMP version of this program. Again, we made sure that we were using the highest level of compiler optimisation on our machines as described in Task 1: Sequential Evaluation. The runtime results will be presented in the form of a table, followed by a graph.

For the same reason we described in Task 1: Sequential Evaluation, we chose to place the wallclock timer around the do-while loop (the workloop), excluding the time that it takes to initialize the threads.

Regarding the choices we made for the values of `EPS`, `HEAT` and N, we refer back to Task 1: Sequential Evaluation.

The results that we obtained when running valgrind for the OpenMP version can be found below:

```
==2478== Memcheck, a memory error detector
==2478== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2478== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2478== Command: ./openmp
==2478==
size   : 0 M (0 MB)
heat   : 150.000000
epsilon: 0.010000
Number of iterations: 3630
Total time: 37.166603 seconds
==2478==
==2478== HEAP SUMMARY:
==2478==     in use at exit: 3,520 bytes in 8 blocks
==2478==   total heap usage: 12 allocs, 4 frees, 53,360 bytes allocated
==2478==
==2478== LEAK SUMMARY:
==2478==    definitely lost: 0 bytes in 0 blocks
==2478==    indirectly lost: 0 bytes in 0 blocks
==2478==      possibly lost: 864 bytes in 3 blocks
==2478==    still reachable: 2,656 bytes in 5 blocks
==2478==         suppressed: 0 bytes in 0 blocks
==2478== Rerun with --leak-check=full to see details of leaked memory
==2478==
==2478== For lists of detected and suppressed errors, rerun with: -s
==2478== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As can be seen, valgrind showed that there are a few hundred bytes that are possibly lost. However, we found that "compiling with the option `-fopenmp` will introduce a (pseudo-) memory leak"[1]. Therefore, we decided that for this task we did not have to worry about those possibly lost bytes.

[1] https://medium.com/@auraham/pseudo-memory-leaks-when-using-openmp-11a383cc4cf9

Implementation Process

Now that we created an improved version of the given sequential program and we also analysed its runtime performance, we started to make the program run in parallel using the application programming interface called OpenMP (Open Multi-Processing). We will now describe the process that we went through in order to make this program parallel.

For both `openmp1.c` and `openmp2.c` we tried to parallelize the function `isStable()` by using different values for the scheduling type that we used, namely `guided`. We also tried the other scheduling types, e.g. `static`, `dynamic` and `runtime`, but these three types did have a worse runtime than `guided`. These versions of the program had worse runtimes than the sequential version, since the `-O3` flag causes the compiler to stop the loop for `isStable` as soon as it encounters a difference that is bigger than `EPS`. In the case of our parallel implementation, this is not possible since we have different threads running this function for different parts of the array. The function now looks as follows:

```
bool isStable(double *old, double *new, int n, double eps)
{
   int i;
   bool res = true;

   #pragma omp parallel for schedule(guided,10) reduction (&& : res)
   for(i=1; i<n-1; i++) {
      res = res && (fabs(old[i] - new[i]) <= eps);
   }
   return res;
}
```

For the next version, `openmp.c`, we parallelized the loop in the function `relax()`, using `guided` scheduling with an initial size of 11. This resulted in a program that was faster than the sequential version and after trying the things that are described below, we decided that this would be our final version. The function now looks as follows:

```
void relax(double *in, double *out, int n)
{
   int i;
   #pragma omp parallel for schedule(guided,11)
   for(i=1; i<n-1; ++i) {
      out[i] = 0.25*in[i-1] + 0.5*in[i] + 0.25*in[i+1];
   }
}
```

Next, we tried to parallelize the workloop as can be found in `openmp3.c`. In short, we separated the work evenly between the different threads. After each of the threads relaxed its own part of the array, it checks whether that part of the array is stable and then all threads combine their results and write it to the shared variable `res`. This version was again slower than the sequential version, this time because of the `OMP single` and `OMP barrier` blocks that we used. We rewrote the workloop in the following way:

```
#pragma omp parallel shared(iterations, res)
{
   int nr_threads = omp_get_num_threads();
   bool local_res;
   double local_iterations;
   int my_id, loop_start, loop_end, i;

   my_id = omp_get_thread_num();

   loop_start = my_id * (n/nr_threads);
   if (my_id == 0)
      loop_start = 1;
```

```
    loop_end = (my_id + 1) * (n/nr_threads);
    if (my_id == nr_threads-1)
       loop_end = n-1;

    for (local_iterations = 0; !res; ++local_iterations) {
       /* compute new values */
       relax(a, b, loop_start, loop_end);

       /* check for convergence */
       local_res = true;
       #pragma omp barrier
       #pragma omp single
       {
          res = true;
       }

       local_res = isStable(a, b, loop_start, loop_end, EPS);

       #pragma omp critical
       {
          res = res && local_res;
       }

       /* "copy" a to b by swapping pointers */
       #pragma omp barrier
       #pragma omp single
       {
          tmp = a;
          a = b;
          b = tmp;
       }
    }

    /* save number of iterations in shared variable */
    #pragma omp single
    {
       iterations = local_iterations;
    }
}
```

In `openmp4.c` we have rewritten the for-loop into a do-while loop and we got rid of one `OMP single` block. Instead of letting each thread have a local variable and in the end let only one thread write the value of this local variable to the global variable, we only have one of the threads increment the global variable `iterations`. The workloop was rewritten in the following way:

```
#pragma omp parallel shared(iterations, res)
{
    int nr_threads = omp_get_num_threads();
    bool local_res;
    double local_iterations;
    int my_id, start, stop, i;

    my_id = omp_get_thread_num();

    start = my_id * (n/nr_threads);
```

```
    if (my_id == 0)
        start = 1;

    stop = (my_id + 1) * (n/nr_threads);
    if (my_id == nr_threads-1)
        stop = n-1;

    do {
        /* compute new values */
        relax(a, b, start, stop);

        /* check for convergence */
        local_res = true;
        #pragma omp barrier
        #pragma omp single
        {
            res = true;
        }

        local_res = isStable(a, b, start, stop, EPS);

        #pragma omp critical
        {
            res = res && local_res;
        }

        /* "copy" a to b by swapping pointers */
        #pragma omp barrier
        #pragma omp single
        {
            tmp = a;
            a = b;
            b = tmp;
            iterations++;
        }
    } while(!res);
}
```

In `openmp5.c` we combined the two remaining `OMP single` blocks into one `OMP single` block, to try and make this version faster. However, in the end the runtimes for this version were still worse than the runtimes for the sequential version so we decided to not use this version for the measurements.
The workloop in `openmp5.c` looks as follows:

```
#pragma omp parallel shared(iterations, res)
{
    int nr_threads = omp_get_num_threads();
    bool local_res;
    int local_iterations;
    int my_id, start, stop, i;

    my_id = omp_get_thread_num();

    start = my_id * (n/nr_threads);
    if (my_id == 0)
        start = 1;

    stop = (my_id + 1) * (n/nr_threads);
```

```
    if (my_id == nr_threads-1)
        stop = n-1;

    do {
        /* compute new values */
        relax(a, b, start, stop);

        /* check for convergence */
        local_res = true;
        #pragma omp barrier
        #pragma omp single
        {
            /* "copy" a to b by swapping pointers */
            res = true;
            tmp = a;
            a = b;
            b = tmp;
            iterations++;
        }

        local_res = isStable(b, a, start, stop, EPS);

        #pragma omp critical
        {
            res = res && local_res;
        }
        #pragma omp barrier
    } while(!res);
}
```

In our `Task2` directory you will find the `Implementation_Process_OpenMP`, which holds our old versions of the code. The final and tested version will be found in the file `openmp.c`.

The runtime results that we obtained, broken down per person, can be found in the sections below.

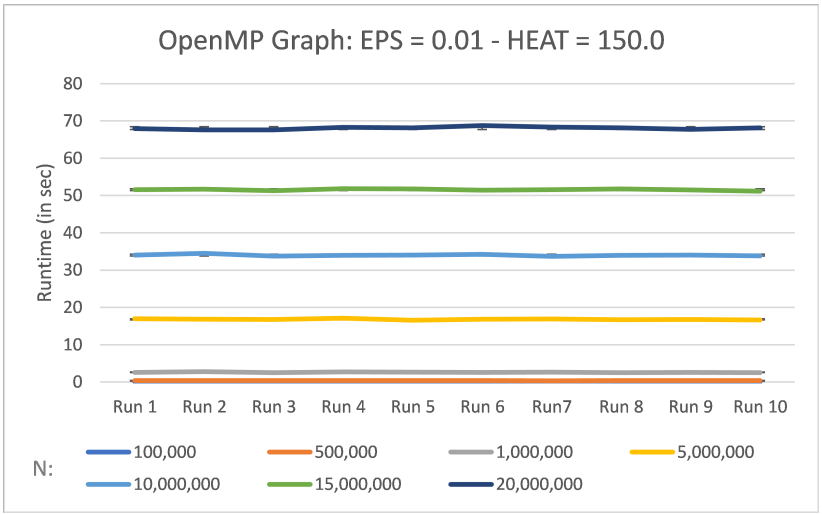
Results of Onno de Gouw

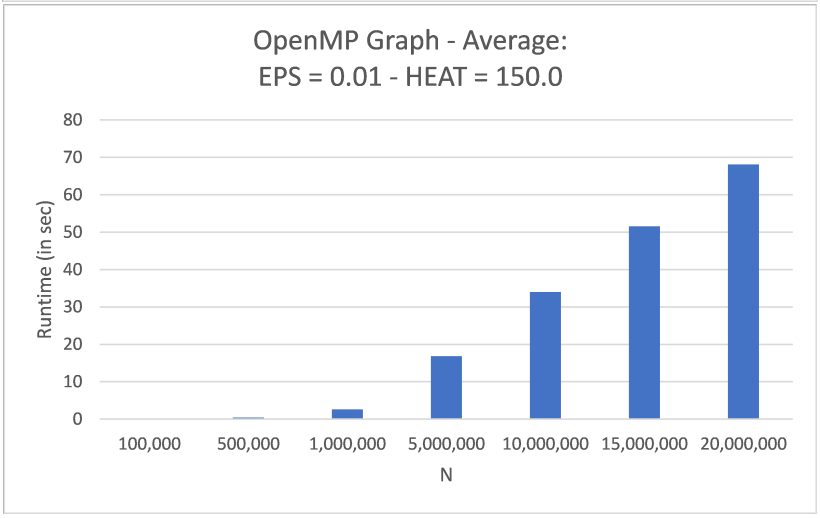
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.080677	0.073001	0.081004	0.073428	0.082995	0.076859	0.083435
500,000	0.409393	0.400704	0.400417	0.407714	0.398513	0.418038	0.375071
1,000,000	2.581175	2.791696	2.534086	2.745755	2.621896	2.562391	2.634368
5,000,000	16.973613	16.848304	16.758884	17.083617	16.551902	16.846964	16.883882
10,000,000	34.058994	34.531764	33.785833	33.955543	34.067175	34.271297	33.686472
15,000,000	51.582582	51.697808	51.312339	51.822339	51.802384	51.433012	51.535211
20,000,000	67.935528	67.619446	67.578641	68.315097	68.123167	68.794557	68.335801

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.085165	0.081619	0.083782	0.0801965	0.085165	0.073001
500,000	0.387507	0.418557	0.386051	0.4001965	0.418557	0.375071
1,000,000	2.532814	2.558941	2.528089	2.6091211	2.791696	2.528089
5,000,000	16.698453	16.746194	16.637704	16.8029517	17.083617	16.551902
10,000,000	33.970074	34.020513	33.843316	34.0190981	34.531764	33.686472
15,000,000	51.792631	51.532649	51.171326	51.5682281	51.822339	51.171326
20,000,000	68.173901	67.759964	68.151034	68.0787136	68.794557	67.578641

The graphs below will display the results just obtained:



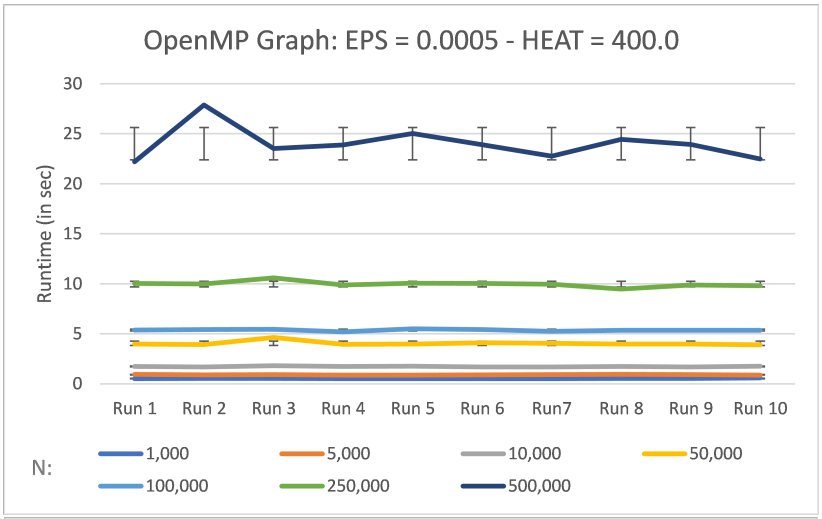


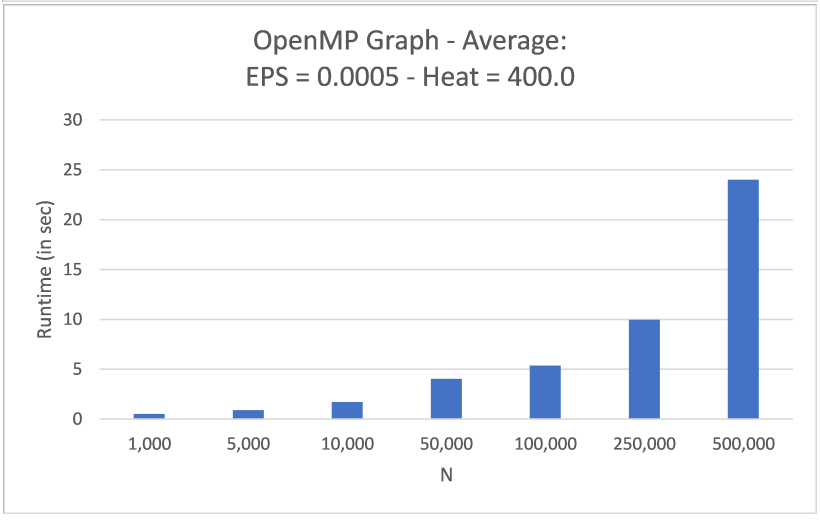
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.513494	0.550222	0.525583	0.503719	0.506913	0.518316	0.505313
5,000	0.950502	0.905537	0.914877	0.880453	0.873548	0.904268	0.926688
10,000	1.740583	1.690043	1.800921	1.738801	1.771059	1.686121	1.686465
50,000	3.960601	3.923813	4.645591	3.952784	3.979169	4.092077	4.040087
100,000	5.381218	5.425063	5.449818	5.190714	5.496208	5.416261	5.250051
250,000	10.025881	9.980381	10.587583	9.880182	10.052291	10.037617	9.962051
500,000	22.196831	27.869187	23.510077	23.885961	25.009011	23.910222	22.751927

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.525832	0.539037	0.578255	0.5266684	0.578255	0.503719
5,000	0.932913	0.921686	0.868055	0.9078527	0.950502	0.868055
10,000	1.743644	1.690604	1.766951	1.7315192	1.800921	1.686121
50,000	3.969004	3.985131	3.892812	4.0441069	4.645591	3.892812
100,000	5.356772	5.338549	5.335648	5.3640302	5.496208	5.190714
250,000	9.477175	9.870812	9.802197	9.9676170	10.587583	9.477175
500,000	24.445241	23.936245	22.467984	23.9982686	27.869187	22.196831

The graphs below will display the results just obtained:





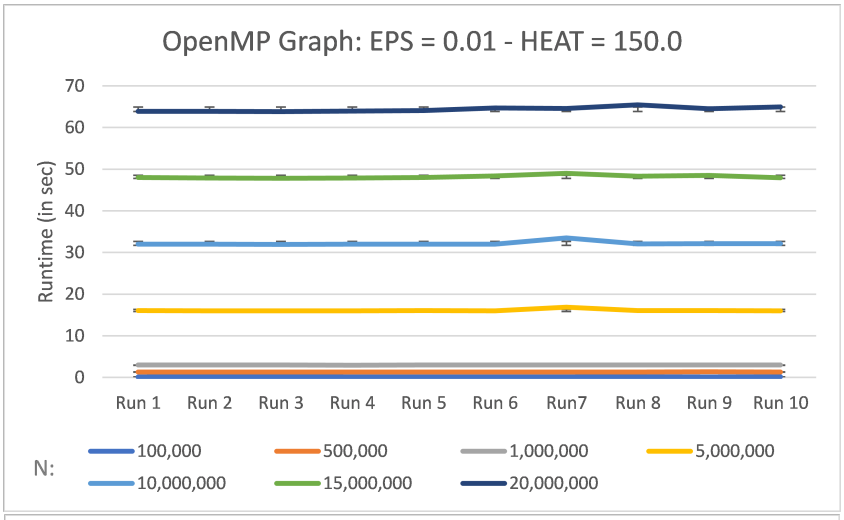
Results of Laura Kolijn

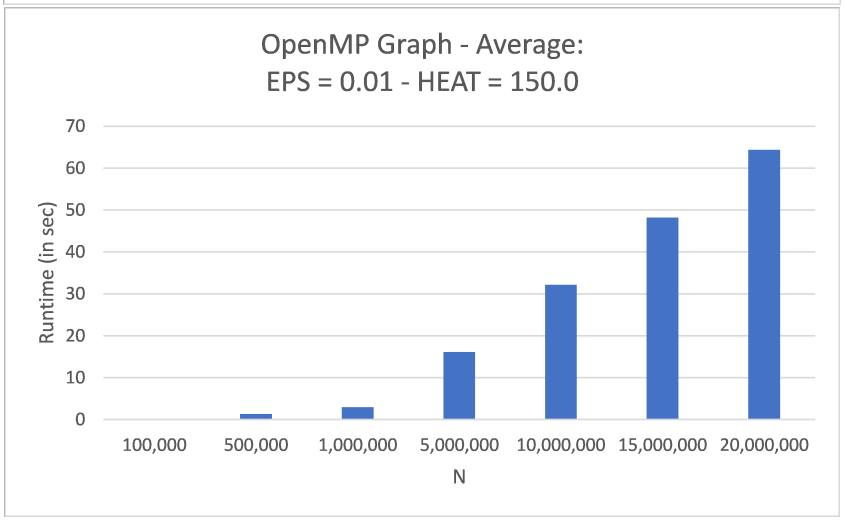
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.136794	0.163964	0.145022	0.142690	0.138967	0.138835	0.139979
500,000	1.294145	1.283963	1.293262	1.300325	1.300842	1.301044	1.293873
1,000,000	2.968485	2.974365	2.971300	2.963104	2.975534	2.995460	2.984025
5,000,000	16.031720	15.960726	15.976645	15.988248	16.031411	16.015387	16.852775
10,000,000	31.984434	31.972275	31.928935	32.000989	32.022136	31.967404	33.470046
15,000,000	48.020694	47.869656	47.795514	47.890381	47.963847	48.367528	49.021470
20,000,000	63.850081	63.866751	63.790803	63.945629	64.057503	64.715069	64.558576

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.144685	0.143806	0.145124	0.1439866	0.163964	0.136794
500,000	1.285420	1.345891	1.302259	1.3001024	1.345891	1.283963
1,000,000	2.979384	2.988243	2.981917	2.9781817	2.995460	2.963104
5,000,000	16.021739	16.059273	16.012724	16.0950648	16.852775	15.960726
10,000,000	32.037841	32.099865	32.096732	32.1580657	33.470046	31.928935
15,000,000	48.317813	48.478001	47.948570	48.1673474	49.021470	47.795514
20,000,000	65.440572	64.505079	64.963085	64.3693148	65.440572	63.790803

The graphs below will display the results just obtained:



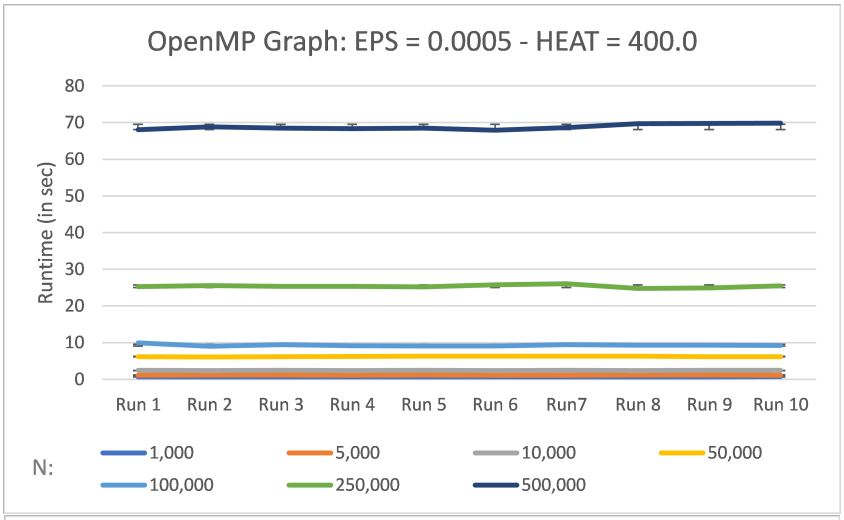


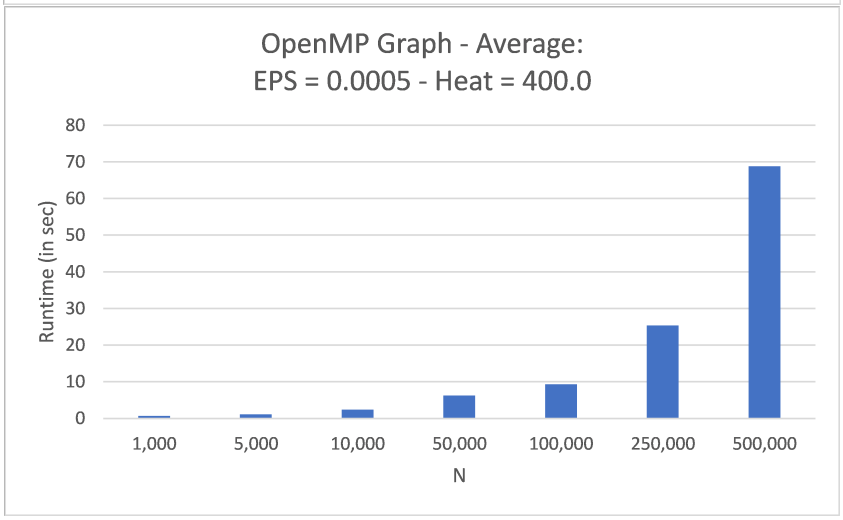
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.666973	0.659137	0.664582	0.690138	0.666843	0.651107	0.673561
5,000	1.075953	1.149913	1.180001	1.081534	1.164087	1.083521	1.096747
10,000	2.454024	2.406200	2.416406	2.402691	2.417425	2.404132	2.419698
50,000	6.141555	6.119509	6.148283	6.262788	6.297063	6.278545	6.320907
100,000	9.928285	9.008116	9.428990	9.145027	9.061543	9.089436	9.413748
250,000	25.306352	25.578045	25.360501	25.345615	25.225107	25.767718	26.041997
500,000	68.064775	68.816531	68.514604	68.351794	68.468546	67.948078	68.651713

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.660239	0.668583	0.769574	0.6770737	0.769574	0.651107
5,000	1.138397	1.186117	1.102744	1.1259014	1.186117	1.075953
10,000	2.410608	2.429988	2.447844	2.4209016	2.454024	2.402691
50,000	6.287460	6.149423	6.145709	6.2151242	6.320907	6.119509
100,000	9.268161	9.261432	9.223173	9.2827911	9.928285	9.008116
250,000	24.765940	24.941249	25.480841	25.3813365	26.041997	24.765940
500,000	69.735928	69.786771	69.876569	68.8215309	69.786771	67.948078

The graphs below will display the results just obtained:





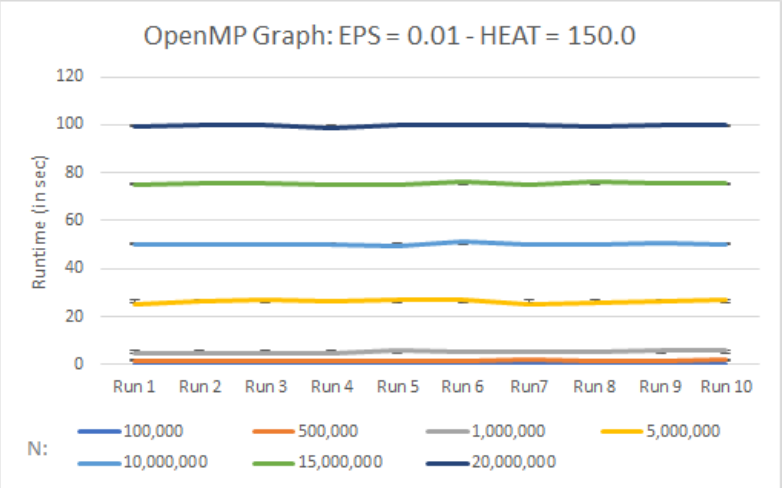
Results of Stefan Popa

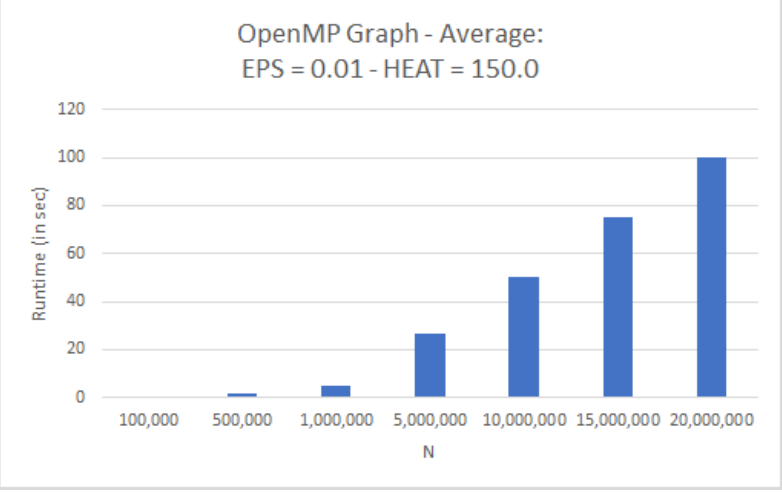
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.317285	0.340708	0.313450	0.368023	0.348073	0.326982	0.315436
500,000	1.615848	1.632016	1.669401	1.656151	1.636639	1.660077	1.916060
1,000,000	4.943805	4.926263	4.805304	4.815827	5.675643	5.261058	5.277606
5,000,000	25.340524	26.446108	26.883782	26.316272	27.318791	26.867472	25.407879
10,000,000	49.889622	50.361572	50.380513	50.122451	49.565840	51.106443	50.359999
15,000,000	75.045481	75.413574	75.759199	75.008738	74.737440	76.348442	74.883674
20,000,000	99.437203	99.664730	100.069926	98.957374	100.259927	99.919940	100.104748

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.324433	0.336149	0.320772	0.3311311	0.368023	0.313450
500,000	1.616846	1.602467	1.913995	1.6919500	1.916060	1.602467
1,000,000	5.225715	5.639394	5.939226	5.2509841	5.939226	4.805304
5,000,000	25.733955	26.596701	26.740817	26.3652301	27.318791	25.340524
10,000,000	49.896021	50.664181	50.286218	50.2632860	51.106443	49.565840
15,000,000	76.004877	75.348442	75.634230	75.4184097	76.348442	74.737440
20,000,000	99.385715	99.781242	99.944253	99.7525058	100.259927	98.957374

The graphs below will display the results just obtained:



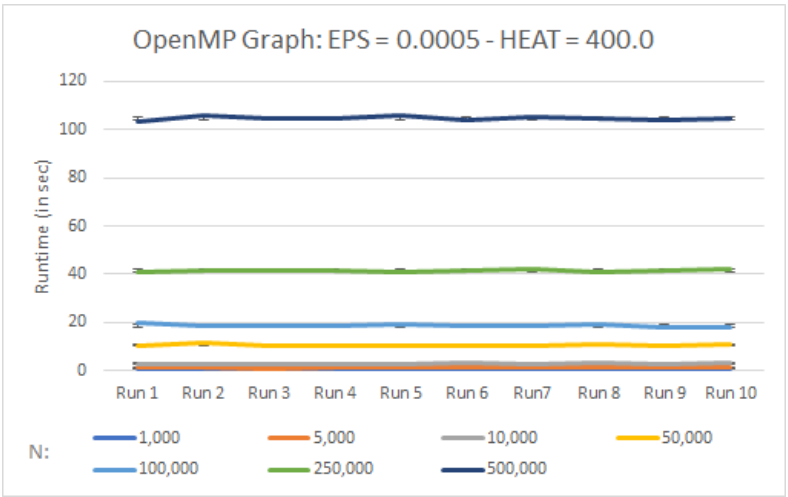


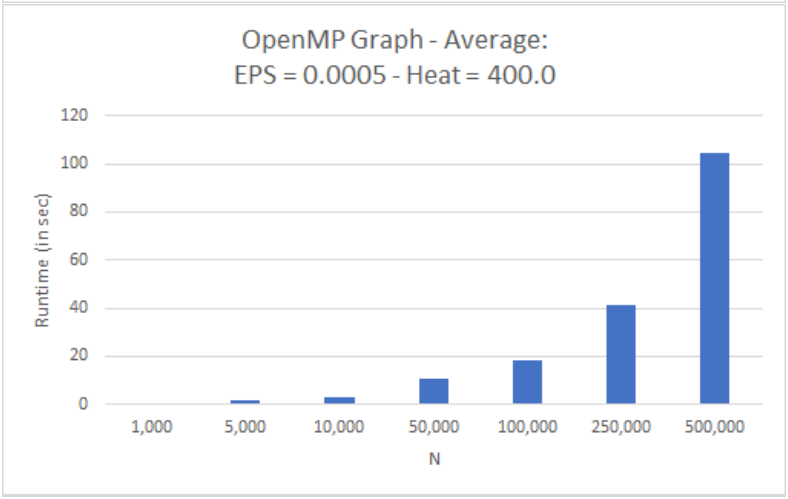
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.759208	0.709102	0.723261	0.778935	0.759329	0.74722	0.743747
5,000	1.449319	1.509016	1.424678	1.533326	1.509670	1.498724	1.506283
10,000	2.960727	2.967102	2.919928	2.965896	2.956458	3.061347	2.940954
50,000	10.638589	11.624309	10.381951	10.277017	10.407740	10.522455	10.317698
100,000	19.767344	18.425653	18.618834	18.558288	19.126618	18.492624	18.726820
250,000	40.975475	41.640059	41.451853	41.350861	40.871007	41.636759	42.315953
500,000	103.400967	106.01799	104.564991	104.347404	105.741218	103.820172	105.374418

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.735831	0.739269	0.759138	0.745504	0.778935	0.709102
5,000	1.487439	1.481476	1.503234	1.4903165	1.533326	1.424678
10,000	3.120711	2.957024	3.322334	3.0172481	3.322334	2.919928
50,000	10.805469	10.258880	10.725235	10.5959343	11.624309	10.258880
100,000	19.126618	17.871044	18.205610	18.6919453	19.767344	17.871044
250,000	40.943635	41.591238	42.042051	41.4818891	42.315953	40.871007
500,000	104.240042	103.888420	104.512967	104.5908589	106.017990	103.400967

The graphs below will display the results just obtained:





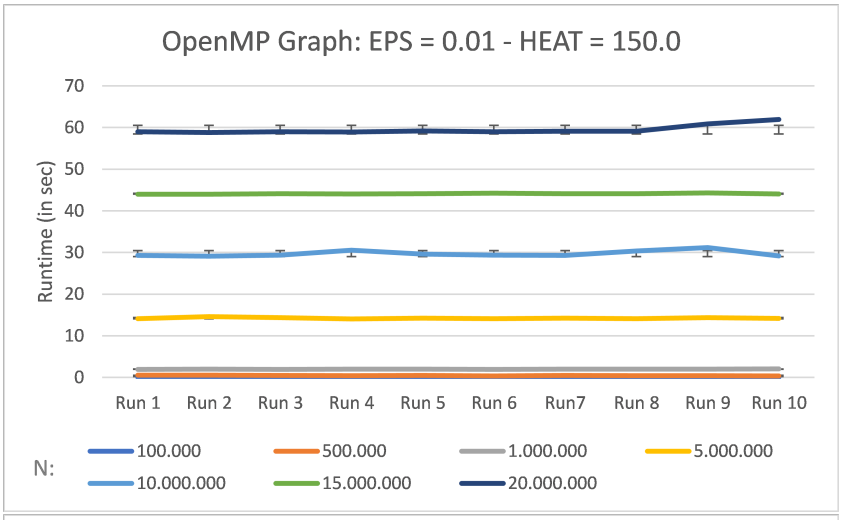
Results of Denise Verbakel

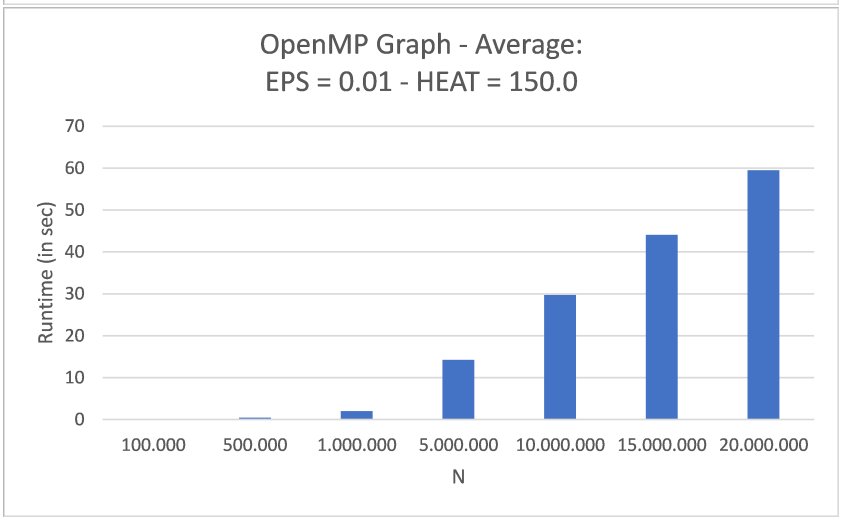
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.071601	0.073761	0.079011	0.073047	0.077453	0.072971	0.084701
500,000	0.570289	0.539128	0.506779	0.412993	0.483313	0.358706	0.483784
1,000,000	1.927271	1.976801	1.962761	1.968173	2.000223	1.963984	1.979396
5,000,000	14.116109	14.596571	14.367592	14.064953	14.241270	14.111934	14.244915
10,000,000	29.290292	29.131274	29.384546	30.544089	29.635637	29.379942	29.275782
15,000,000	44.006304	43.978693	44.083934	44.070316	44.110103	44.221242	44.108488
20,000,000	58.986632	58.770091	58.955151	58.924956	59.163161	58.955206	59.087022

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.071750	0.085880	0.078689	0.0768864	0.085880	0.071601
500,000	0.431331	0.417450	0.366130	0.4569903	0.570289	0.358706
1,000,000	1.969136	2.019964	2.083510	1.9851219	2.083510	1.927271
5,000,000	14.085502	14.343509	14.187570	14.2359925	14.596571	14.064953
10,000,000	30.390443	31.151652	29.187614	29.7371271	31.151652	29.131274
15,000,000	44.092728	44.306682	44.056785	44.1035275	44.306682	43.978693
20,000,000	59.123718	60.845962	61.935575	59.4747474	61.935575	58.770091

The graphs below will display the results just obtained:



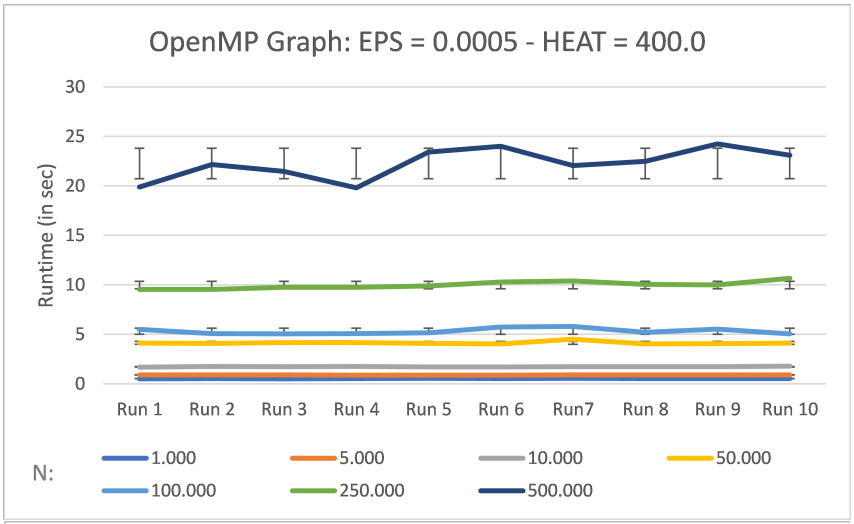


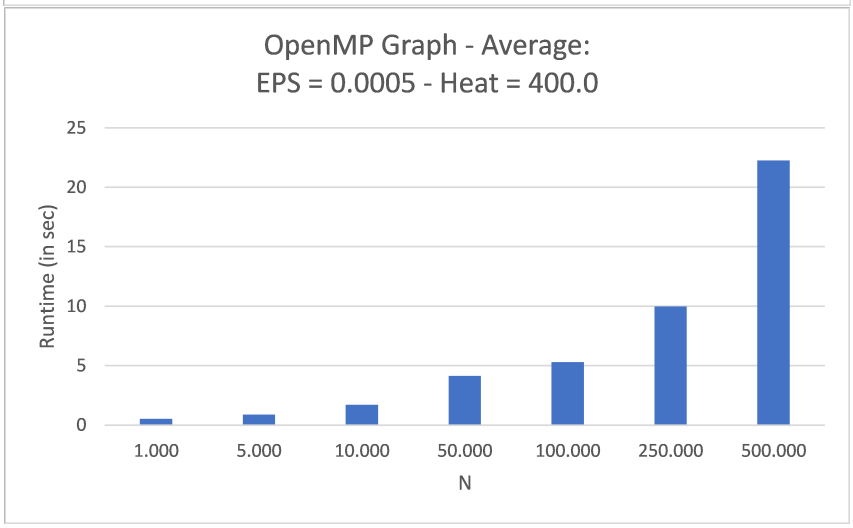
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.490524	0.504089	0.492285	0.509758	0.549420	0.512700	0.534196
5,000	0.904851	0.889823	0.901355	0.862213	0.864163	0.863530	0.900671
10,000	1.671828	1.735884	1.709902	1.739052	1.695536	1.682238	1.710589
50,000	4.097483	4.085064	4.159870	4.165293	4.060909	4.009447	4.503518
100,000	5.479457	5.055128	5.041541	5.061866	5.148823	5.729079	5.789816
250,000	9.537100	9.522384	9.733974	9.748688	9.871497	10.268254	10.377598
500,000	19.881921	22.161699	21.453302	19.792544	23.407942	24.005931	22.049239

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.517068	0,517953	0.519180	0.5147173	0.549420	0.490524
5,000	0.883779	0.880087	0.900504	0.8850976	0.904851	0.862213
10,000	1.728597	1.721384	1.762408	1.7157418	1.762408	1.671828
50,000	4.017047	4.054476	4.110724	4.1263831	4.503518	4.009447
100,000	5.187924	5.516653	5.038043	5.3048330	5.789816	5.038043
250,000	10.046695	9.977775	10.645306	9.9729271	10.645306	9.522384
500,000	22.477853	24.241004	23.087903	22.2559338	24.241004	19.792544

The graphs below will display the results just obtained:





Task 3: MPI

Choices Performance Evaluation

After having evaluated the performance of the sequential code and the OpenMP version, we continued with implementing and evaluating the performance of an MPI version of this program. Again, we made sure that we were using the highest level of compiler optimisation on our machines as described in Task 1: Sequential Evaluation. The runtime results will be presented in the form of a table, followed by a graph.

For the same reason we described in Task 1: Sequential Evaluation, we chose to place the wallclock timer around the do-while loop (the workloop), excluding the time that it takes to initialize the ranks and chunks belonging to these ranks. If we would not have done this, the functions that are needed to initialize everything before we can use MPI, would cause too many differences in runtime between this version and the sequential and OpenMP version.

Regarding the choices we made for the values of `EPS`, `HEAT` and N, we refer back to Task 1: Sequential Evaluation.

The results that we obtained when running valgrind for the MPI version can be found below:

```
==2544== Memcheck, a memory error detector
==2544== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2544== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2544== Command: mpirun -N 4 ./mpi
==2544==
size   : 0 M (0 MB)
heat   : 150.000000
epsilon: 0.010000
Number of iterations: 3630
Total time: 0.014293 seconds
==2544==
==2544== HEAP SUMMARY:
==2544==     in use at exit: 0 bytes in 0 blocks
==2544==   total heap usage: 1,036 allocs, 1,036 frees, 2,358,731 bytes allocated
==2544==
==2544== All heap blocks were freed -- no leaks are possible
==2544==
==2544== For lists of detected and suppressed errors, rerun with: -s
==2544== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As can be seen, valgrind showed that all heap blocks were freed - so no leaks are possible - and no other errors were found. We can thus conclude that we have a memory-leak free MPI version. Next to this, as mentioned in the section called Hardware and Software Specification, it is important that we all run the `mpirun` command with the option `-N 4`. This option gave us the best runtimes. This is because we all run on machines that use 4 cores and thus get the most out of our machines.

Implementation Process

Now that we created an improved version of the given sequential program and we also analysed its runtime performance, we started to make the program run in parallel using the Message Passing Interface, abbreviated as MPI. Before we start describing the process that we went through in order to make this program run in parallel, we want to denote something: if '...' is used in the code, this means that a piece of code is left away in order to make explanation more clear.

In `mpi1.c` we tried to parallelize the function `relax()` by using MPI. The first thing we noticed when parallelizing using MPI was that we needed to use the following statements:

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank_id);
MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);
MPI_Finalize();
```

Here, the first statement is needed to be able to use MPI functions, the second and third statement are needed to give each rank a number and to know how many ranks are running this program and the last statement is needed to clean everything up after running the MPI program. Next, in order to parallelize the `relax()` function, we came up with an idea that consisted of splitting up the whole array into smaller chunks and letting every rank relax a smaller part of the array. In order to do this, we decided to make use of pointer arithmetic and set the starting pointer of every rank at another position in the entire array. The code below shows this idea: it determines how large a chunk for every rank is going to be and places the pointers `a` and `b` at their respective beginning of the array.

```
chunk = n/num_ranks;
remainder = n % num_ranks;

//If we have a remainder for chunk: divide over processes
if (remainder != 0) {
   if (rank_id < remainder)
   chunk++;
}

...

a = a + local_start;
b = b + local_start;
```

Next, since every rank will relax its own (little) piece of the array, we noticed that they need to communicate with the other ranks running on the sub-arrays containing the index after the end of the array and the index in front of the array. The reason for this is that it is needed for proper relaxing. We decided to let all ranks having an even rank number send data first to their neighbors and after that receive it from their neighbors, whereas the odd rank numbers would first receive and then send their data. The reason that we chose to do it in this way, is because we are then sure that only half of the ranks would be sending and half of the ranks would be receiving, avoiding a possible deadlock. In the process of sending data to neighbors we also needed to make sure that rank 0 would not send data to a previous rank - since there are none - and this same reasoning holds for the last rank as well, since there is no 'next' rank to send data to. This obviously does not only hold for sending, but also holds for receiving data from non-existing ranks. The code that we created in first instance for exchanging data with neighbors looked like the following:

```
if (rank_id % 2 == 0) {
   if (rank_id != last_rank) {
      // Send last element to next process
      MPI_Send(&b[chunk-1], 1, MPI_DOUBLE, rank_id + 1, 0, MPI_COMM_WORLD);
      MPI_Recv(&recv_above, 1, MPI_DOUBLE, rank_id + 1, 0, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
   }
```

```
    if (rank_id != 0) {
        // Send first element to previous process
        MPI_Send(&b[0], 1, MPI_DOUBLE, rank_id - 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&recv_below, 1, MPI_DOUBLE, rank_id - 1, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
}
else {
    // Receive element from previous process
    MPI_Recv(&recv_below, 1, MPI_DOUBLE, rank_id - 1, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    MPI_Send(&b[0], 1, MPI_DOUBLE, rank_id - 1, 0, MPI_COMM_WORLD);

    if (rank_id != last_rank) {
        // Receive element from next process
        MPI_Recv(&recv_above, 1, MPI_DOUBLE, rank_id + 1, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Send(&b[chunk-1], 1, MPI_DOUBLE, rank_id + 1, 0, MPI_COMM_WORLD);
    }
}
```

Now that we received all the information we needed from other sub-arrays in order to relax, it was time to rewrite the `relax()` function itself. The main change we made is that we added if-statements in order to be able to distinguish between how to calculate the new sub-array (rank 0 for instance needs to perform different calculations than rank 1, since rank 0 only has one neighbor while rank 1 has two). To be able to use the obtained values from the neighbors, which we stored in `recv_above` and `recv_below`, we decided to pass them as an argument to the `relax()` function. This resulted in the following adapted `relax()` function:

```
void relax(double *in, double *out, int start, int stop, int before, int after)
{
    int i;
    for(i=start; i<stop; i++) {
        if (i == 0)
            out[i] = 0.25*before + 0.5*in[i] + 0.25*in[i+1];
        else if (i == stop-1)
            out[i] = 0.25*in[i-1] + 0.5*in[i] + 0.25*after;
        else
            out[i] = 0.25*in[i-1] + 0.5*in[i] + 0.25*in[i+1];
    }
}
```

After each rank had relaxed its own part of the entire array (sub-array), we wanted to check whether we have to break out of the do-while loop. We came up with the idea that only one rank (rank 0) would check whether all other ranks should stop or not by determining whether the full array is stable. However, before we could do this, we needed to send all sub-arrays to rank 0 and collect the data here. At the moment where rank 0 had received all data from the other ranks, it called the function `isStable()`. Once rank 0 had decided that the array is stable, it would set the variable `stopping_condition` to 1 and send back this answer to all other ranks. However, if the array was not stable, it would not set this variable, but keep the value in the way it was initialized, namely 0. After this, it of course also sent this value back to all other ranks in order to communicate that the array was not stable yet and thus all other ranks should continue working. The code for this checking looks like the following:

```
if (rank_id == 0) {
    // Put every value calculated by other processes at the right spot in b:
    // gather everything in 0
```

```
    for (int i = 1; i < num_ranks; i++) {
        MPI_Recv(b + i * chunk, chunk, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
    if (isStable(a, b, n, EPS)) {
        stopping_condition = 1;
        for (int i = 1; i < num_ranks; i++) {
            MPI_Send(&stopping_condition, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    }
    else {
        for (int i = 1; i < num_ranks; i++) {
            MPI_Send(&stopping_condition, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    }
}
else {
    MPI_Send(b, chunk, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
```

Because after this process every rank knows whether it should stop or not, we are able to check for this `stopping_condition` variable in the beginning of the do-while loop (whenever we are not in the first iteration). If this value is a 1 we would break out of the loop; otherwise we would continue with it. This can be seen below:

```
if (iterations != 0) {
    if (rank_id != 0)
        MPI_Recv(&stopping_condition, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

    if (stopping_condition)
        break;

    ...
}
```

We also realized that we can only exchange information with other ranks (neighbors) if there are actually other ranks running. It does not make sense to run a parallel program without multiple ranks. Therefore, we decided to force our program to be run with at least two ranks. We did this by throwing an error and terminating the program if this does not happen, as can be seen here:

```
if (num_ranks > 1) {
    ...
}
else {
    printf("Please use more than one process\n");
    MPI_Finalize();
    exit(0);
}
```

Finally, since we do not want to print the general information of our parallel program multiple times, we decided to assign rank 0 as being the rank that prints output for our program (and thus also handle the wallclock timer).

However, what we did not take into account with our complete implementation and idea of this parallel program, is that in MPI different ranks do not make use of shared memory. Therefore, our whole idea was highly inefficient. We could also notice this when running and testing our program. In order to solve our problems, we decided to apply the following changes. First off, we noticed that we allocated way too much memory and that we were sending every iteration of the do-while loop a lot of data - the entire sub-arrays - to process 0. We decided to change this in `mpi2.c`.

We changed the allocation of memory such that every task allocated not a size of `N`, but a size of `chunk + 2`. Here we made use of calloc as well, instead of using malloc. Now, our usage of the different neighbor ranks changed because of this: we did not have to save the values communicated in separate variables but we were able to store the data at index 0 and `chunk + 1` of the allocated array. Also, we could change the implementation of `relax()` back to the original version since we now do not use `recv_above` and `recv_below` anymore. In order to optimize our new version even further, we decided to use the optimized version of `isStable()` as is presented in Task 1: Sequential Evaluation.

Because we changed the allocation of the arrays and because we do not send everything to process 0 to check whether the complete array is stable anymore, we decided to let each rank check this for themselves. We also decided to now use the MPI function `MPI_Allreduce` in order to determine whether the tasks should stop and break out of the do-while loop or not. If this function causes the value `check` to be 0, all tasks will stop with the work-loop; otherwise they will keep going. The usage of the `MPI_Allreduce` function, decreases the usage of `MPI_Recv` at two places in the code and the usage of `MPI_Send` at three places in the code. Now the communication in order to check the loop condition looks as follows:

```
if (isStable(a, b, chunk, EPS)) {
    running = 0;
}

MPI_Allreduce(&running, &check, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

What we realized next, is that there are quite many if-statements in our do-while loop. We decided to change this and these changes can now be found in our final version as well, namely in the file that is now called `mpi.c`. We found out that the first if-statement that breaks the program out of the loop was redundant. The if-statement that checks whether more than 1 task is used also does not need to be in the while loop and therefore, we placed it before this while loop. Next to this, we tried changing the idea for checking on even and odd rank numbers (as we introduced before in `mpi1.c`). In the end it turned out that we were able to replace this idea by first letting tasks send their data to all neighbors and after that let all ranks receive data from all neighbors. The version we created now ran even faster than checking on odd or even, which could probably be caused by the overhead that the modulo operation provided. We now ended up having communication with neighbor processes in the following way:

```
if (rank_id != last_rank) {
    // Send last element to next process
    MPI_Send(&b[chunk], 1, MPI_DOUBLE, rank_id + 1, 0, MPI_COMM_WORLD);
}
if (rank_id != 0) {
    // Send first element to previous process
    MPI_Send(&b[1], 1, MPI_DOUBLE, rank_id - 1, 0, MPI_COMM_WORLD);
}
if (rank_id != last_rank) {
    // Receive first element from next process
    MPI_Recv(&b[chunk+1], 1, MPI_DOUBLE, rank_id + 1, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
if (rank_id != 0) {
    // Receive last element from previous process
    MPI_Recv(&b[0], 1, MPI_DOUBLE, rank_id - 1, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
```

After this, we saw that we were have a double if-statement two times. However, if we merged this into one if-statement check - so by telling the tasks to first send and then receive data - the implementation got slower and therefore we did not change this. The reason this happens is because if we merge them, it could be the case that every rank is waiting until the last rank sends some data, causing a mayor delay.

After this version, we tried a lot to speed it up, but unfortunately without success. A noticeable try is included as `mpi3.c`. In this version we tried to use `MPI_Bcast` function instead of using the `MPI_Allreduce` function:

```
MPI_Bcast(&running, 1, MPI_INT, rank_id, MPI_COMM_WORLD);

check = 0;
int result;


for (int i = 0; i < num_ranks; i++) {
    if (rank_id != i) {
        MPI_Bcast(&result, 1, MPI_INT, i, MPI_COMM_WORLD);
        check += result;
    }
}

check += running;
```

Unfortunately, our idea to change this did not speed up our implementation.

In our `Task3` directory you will find the `Implementation_Process_MPI`, which contains our old versions of the code. Together with the explanation of our implementation process that we have just described, these files help in understanding even better what we went trough. Our final and also tested on run-time performance version can be found as the file `mpi.c`.

The runtime results that we obtained from these tests can be found in the sections below. We broke down our results per person.

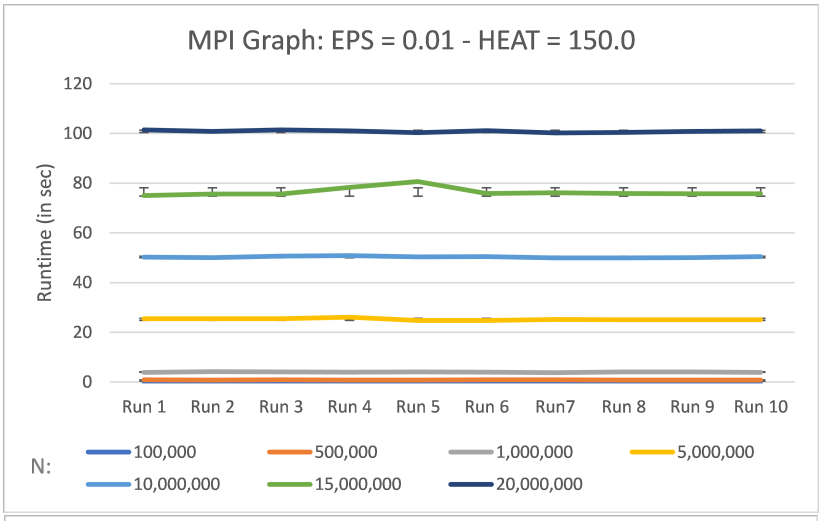
Results of Onno de Gouw

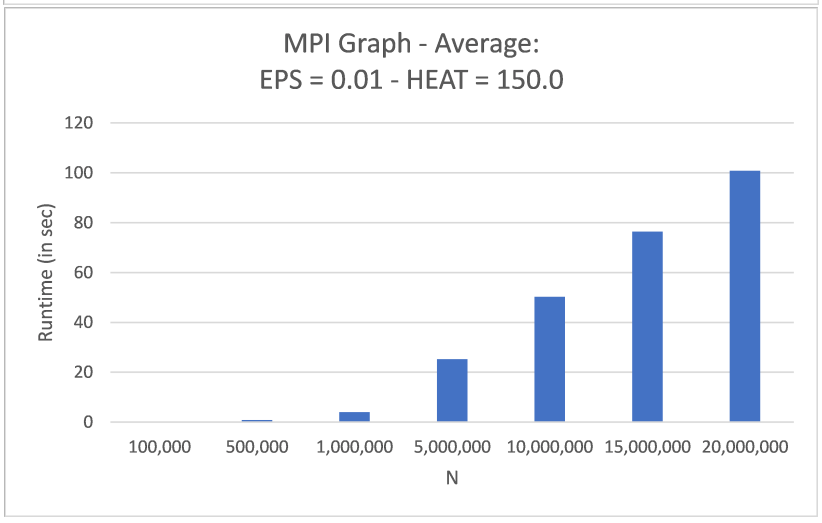
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.140956	0.140202	0.137964	0.161061	0.127549	0.139753	0.179487
500,000	0.875106	0.819938	0.866603	0.799281	0.834112	0.898741	0.910488
1,000,000	3.920805	4.149516	4.066864	4.008061	4.111227	4.018311	3.815291
5,000,000	25.480256	25.506526	25.428805	26.031728	24.796928	24.770093	25.126697
10,000,000	50.247091	50.007926	50.621419	50.826811	50.374701	50.426752	49.930757
15,000,000	74.997825	75.636993	75.619609	78.315722	80.634308	75.782591	76.111243
20,000,000	101.359354	100.761555	101.449242	100.997144	100.266578	101.103071	100.151962

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.152969	0.134951	0.130311	0.1445203	0.179487	0.127549
500,000	0.841648	0.832652	0.832349	0.8510918	0.910488	0.799281
1,000,000	4.102786	4.035977	3.904099	4.0132937	4.149516	3.815291
5,000,000	25.025732	25.072096	25.093616	25.2332477	26.031728	24.770093
10,000,000	49.949971	50.062653	50.465448	50.2913529	50.826811	49.930757
15,000,000	75.826317	75.764793	75.678295	76.4367696	80.634308	74.997825
20,000,000	100.368479	100.764970	101.017528	100.8239883	101.449242	100.151962

The graphs below will display the results just obtained:



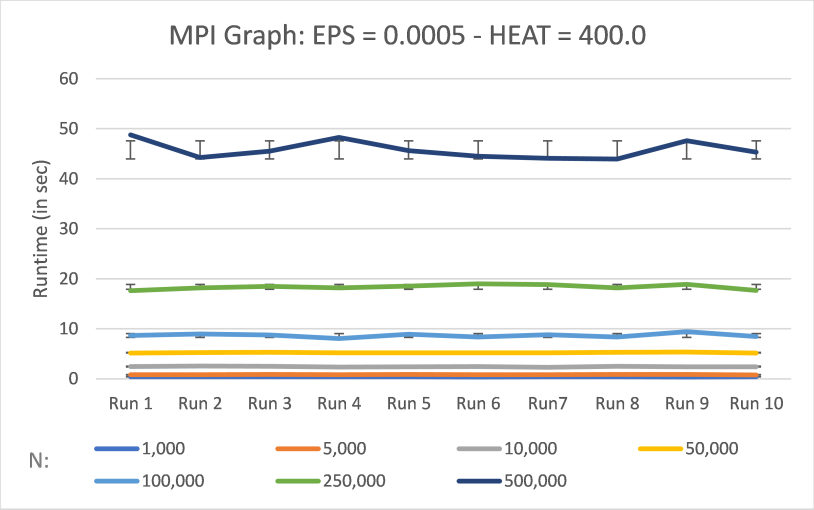


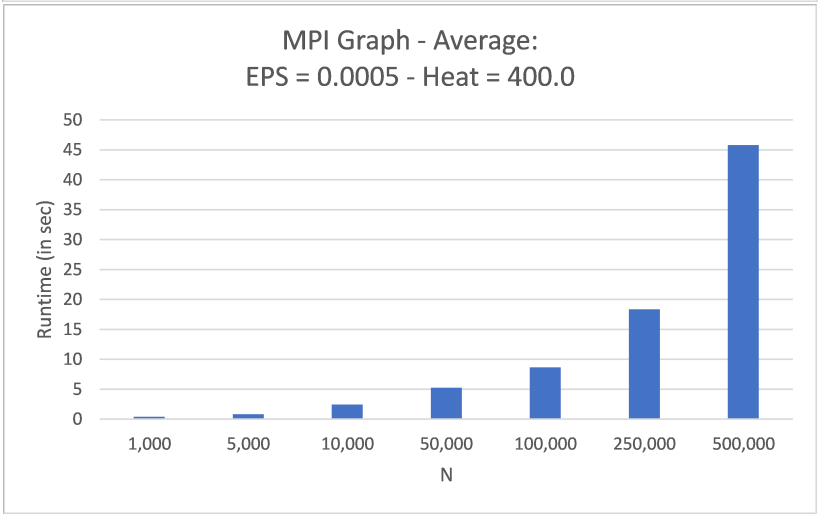
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.422891	0.414638	0.423421	0.411313	0.395503	0.387788	0.405075
5,000	0.831431	0.813731	0.849541	0.841171	0.861455	0.839254	0.827993
10,000	2.457638	2.564938	2.523486	2.340096	2.410648	2.437446	2.315527
50,000	5.170964	5.248057	5.294552	5.186279	5.223025	5.205526	5.218061
100,000	8.680315	8.957155	8.744286	8.024966	8.892528	8.348551	8.794993
250,000	17.641078	18.176436	18.489034	18.159147	18.523713	18.984513	18.861352
500,000	48.768266	44.242326	45.522388	48.267377	45.619695	44.471719	44.059264

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.418645	0.386611	0.390894	0.4056779	0.423421	0.386611
5,000	0.851303	0.856071	0.783479	0.8355429	0.861455	0.783479
10,000	2.477326	2.415585	2.374663	2.4317353	2.564938	2.315527
50,000	5.291717	5.327093	5.163936	5.2329210	5.327093	5.163936
100,000	8.365811	9.413723	8.457951	8.6680279	9.413723	8.024966
250,000	18.190271	18.898967	17.674041	18.3598552	18.984513	17.641078
500,000	43.926941	47.610037	45.291863	45.7779876	48.768266	43.926941

The graphs below will display the results just obtained:





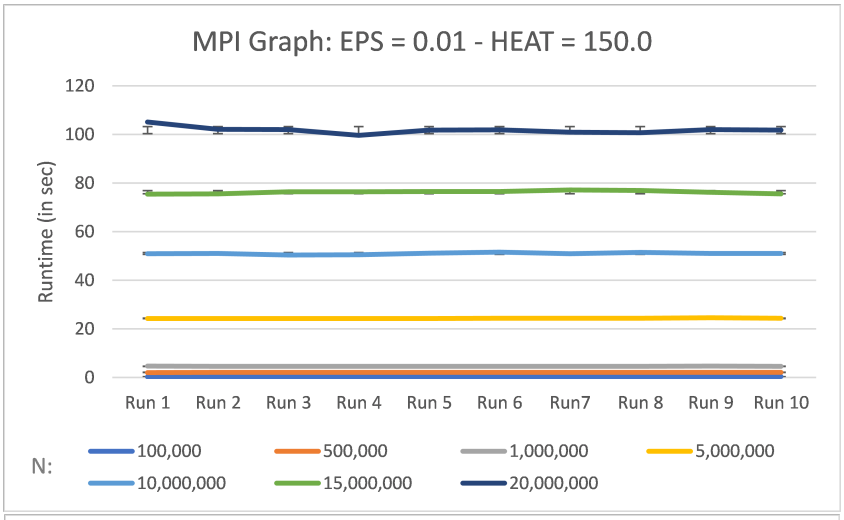
Results of Laura Kolijn

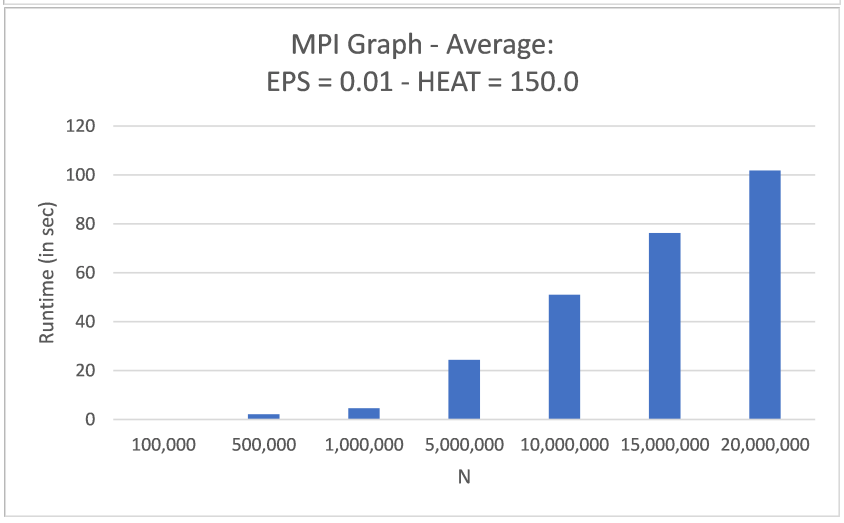
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.278320	0.311726	0.353222	0.318210	0.327719	0.326269	0.344079
500,000	2.037493	2.052866	2.081286	2.102213	2.084892	2.076304	2.095276
1,000,000	4.683550	4.576172	4.567215	4.581755	4.560134	4.579906	4.602851
5,000,000	24.265960	24.248440	24.228733	24.259560	24.274861	24.394338	24.313012
10,000,000	50.925643	51.029690	50.364554	50.487886	51.136834	51.562739	50.960491
15,000,000	75.417589	75.519037	76.422434	76.459926	76.489672	76.514021	77.151282
20,000,000	105.156561	102.095900	101.972350	99.618000	101.803021	101.923657	100.900433

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.347844	0.322623	0.333421	0.3263433	0.353222	0.27832
500,000	2.078404	2.102272	2.096280	2.0807286	2.102272	2.037493
1,000,000	4.611355	4.673762	4.567231	4.6003931	4.683550	4.560134
5,000,000	24.345918	24.519083	24.363279	24.3213184	24.519083	24.228733
10,000,000	51.426499	51.047275	51.026061	50.9967672	51.562739	50.364554
15,000,000	77.008008	76.198569	75.510078	76.2690616	77.151282	75.417589
20,000,000	100.759829	102.006366	101.834337	101.8070454	105.156561	99.618000

The graphs below will display the results just obtained:



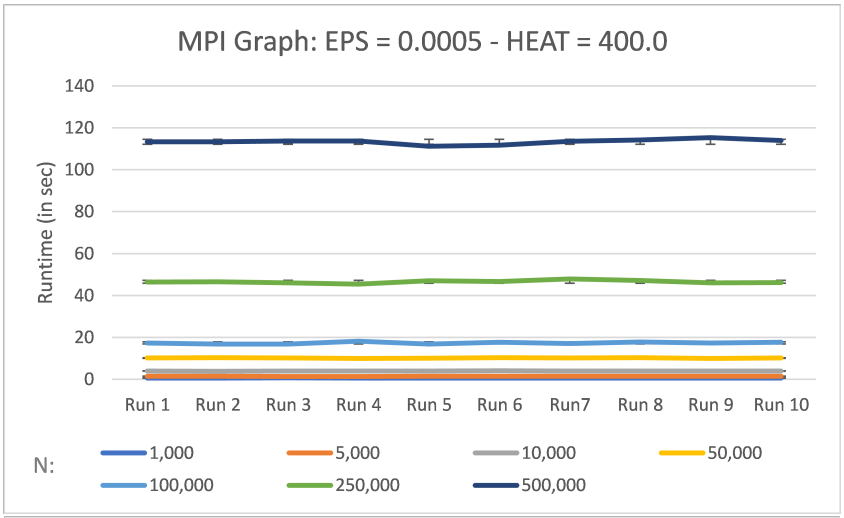


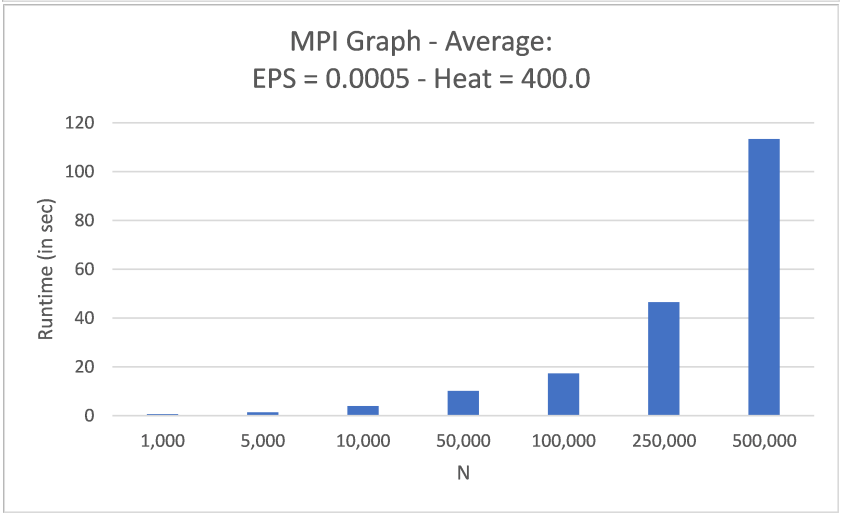
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.559778	0.619250	0.690104	0.665094	0.670590	0.639868	0.603669
5,000	1.424660	1.513925	1.382188	1.377795	1.427904	1.471121	1.438957
10,000	3.926214	3.888195	3.942164	3.958851	3.933194	4.048291	3.942236
50,000	10.205544	10.382558	10.244327	9.982976	10.059985	10.317277	10.214539
100,000	17.308554	16.825445	16.882827	18.189728	16.858995	17.656757	17.142839
250,000	46.381039	46.598713	46.051995	45.387681	47.017609	46.658241	47.879822
500,000	113.303409	113.273276	113.662600	113.728590	111.149107	111.638364	113.570970

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.662814	0.6168930	0.611806	0.6339866	0.690104	0.559778
5,000	1.490479	1.4530410	1.478900	1.4458970	1.513925	1.377795
10,000	3.971754	3.9592800	4.013040	3.9583219	4.048291	3.888195
50,000	10.329002	9.9893220	10.249210	10.1974740	10.382558	9.982976
100,000	17.816640	17.2928890	17.760814	17.3735488	18.189728	16.825445
250,000	47.142397	46.0805880	46.173167	46.5371252	47.879822	45.387681
500,000	114.140889	115.2777460	113.886925	113.3631876	115.277746	111.149107

The graphs below will display the results just obtained:





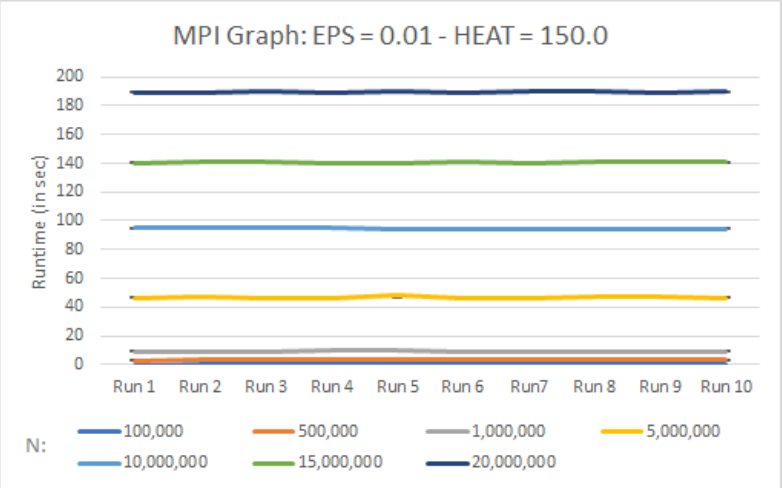
Results of Stefan Popa

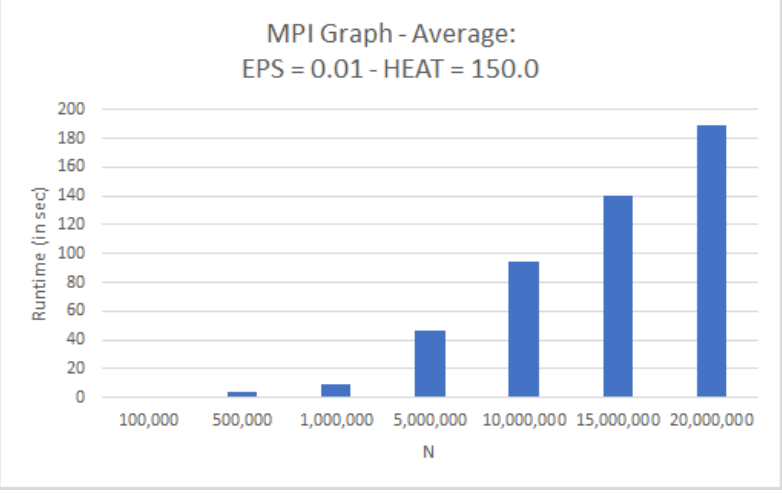
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.684209	0.724668	0.721466	0.734311	0.695823	0.754488	0.734890
500,000	3.634701	3.874375	3.715942	3.798280	3.798217	3.738975	3.713534
1,000,000	9.161755	9.214154	9.313576	10.273787	9.417046	9.231305	9.280888
5,000,000	46.685359	47.578925	46.644444	46.284604	48.081906	45.994092	46.121111
10,000,000	94.880342	95.059223	95.212884	95.102255	94.406302	93.867052	93.880256
15,000,000	140.082673	141.04708	140.718898	139.98883	139.707816	141.095545	140.417983
20,000,000	188.859711	189.144779	190.197614	189.435024	189.641531	188.559884	190.216832

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.719893	0.723056	0.745234	0.7238038	0.754488	0.684209
500,000	3.730822	3.678645	4.007544	3.7691035	4.007544	3.634701
1,000,000	9.144791	9.181113	8.956144	9.3174559	10.273787	8.956144
5,000,000	47.104061	47.283139	46.521688	46.8299329	48.081906	45.994092
10,000,000	94.579289	94.105263	94.411281	94.5504147	95.212884	93.867052
15,000,000	140.683077	140.897604	141.047121	140.5686627	141.095545	139.707816
20,000,000	190.036589	189.474043	190.121837	189.5687844	190.216832	188.559884

The graphs below will display the results just obtained:



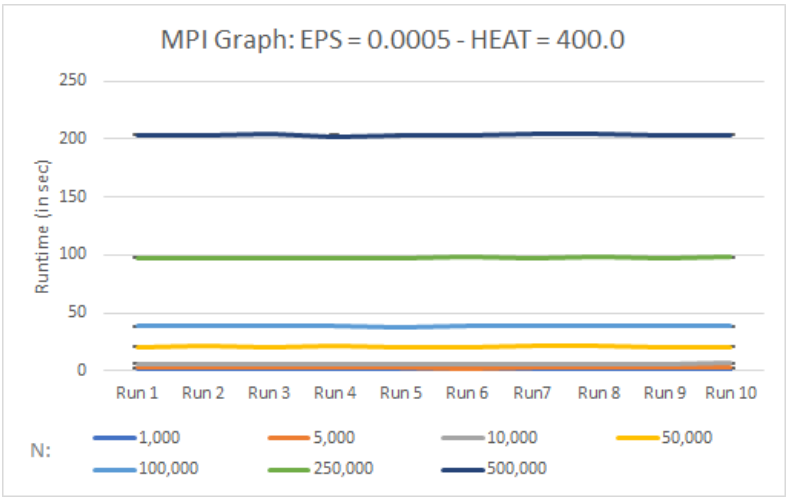


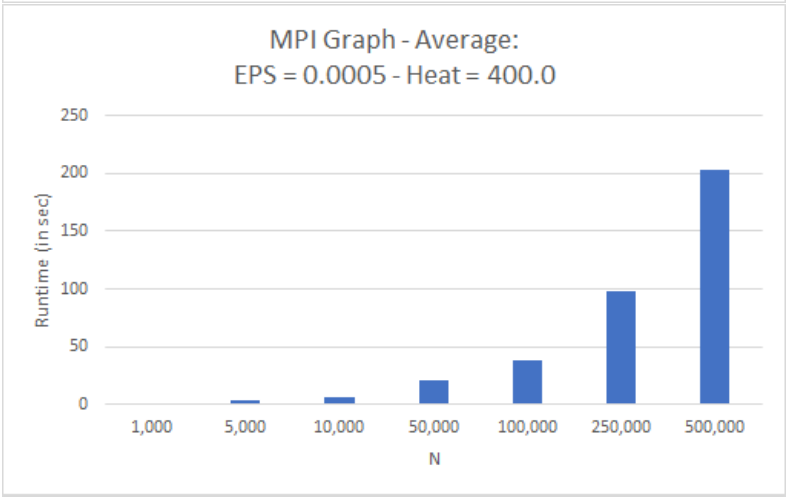
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	1.341257	1.316972	1.266185	1.359394	1.305751	1.332479	1.310100
5,000	3.000493	3.015208	3.088248	3.150140	3.018746	2.975862	3.121744
10,000	6.080313	6.087163	6.130065	6.054605	6.202797	6.277563	6.083579
50,000	20.705570	21.297632	20.912147	22.239051	20.727570	20.838115	21.659528
100,000	38.733861	38.859019	38.754936	38.615133	38.601742	39.094164	39.116043
250,000	97.113043	97.383636	97.470244	96.926969	97.766741	99.131204	97.546837
500,000	203.245800	203.535948	204.080257	202.415794	202.724643	203.437192	203.899543

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	1.273715	1.291907	1.265194	1.3062954	1.359394	1.265194
5,000	3.010609	3.025857	3.331799	3.0738706	3.331799	2.975862
10,000	6.048111	6.267923	7.069209	6.2301328	7.069209	6.048111
50,000	21.281168	20.920746	20.210704	21.0792231	22.239051	20.210704
100,000	38.949488	38.615133	39.093752	38.8433271	39.116043	38.601742
250,000	98.265743	96.926969	98.064390	97.6595776	99.131204	96.926969
500,000	204.175800	203.500930	203.262526	203.4278433	204.175800	202.415794

The graphs below will display the results just obtained:





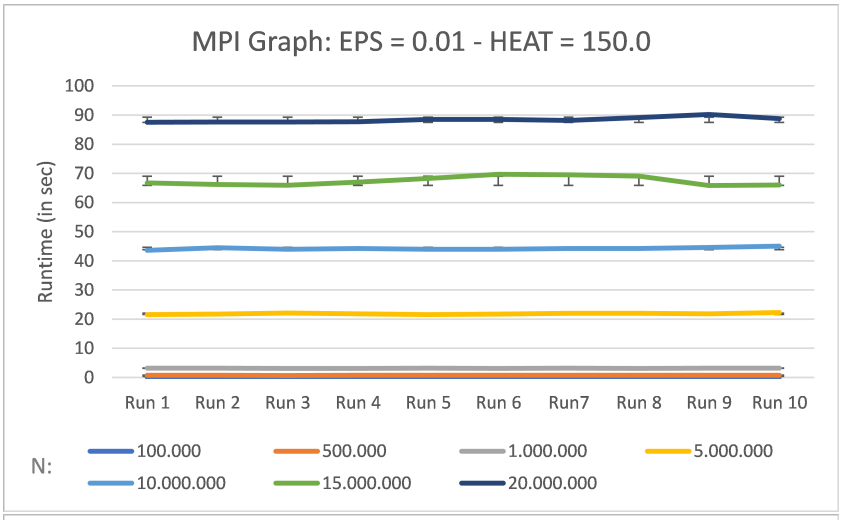
Results of Denise Verbakel

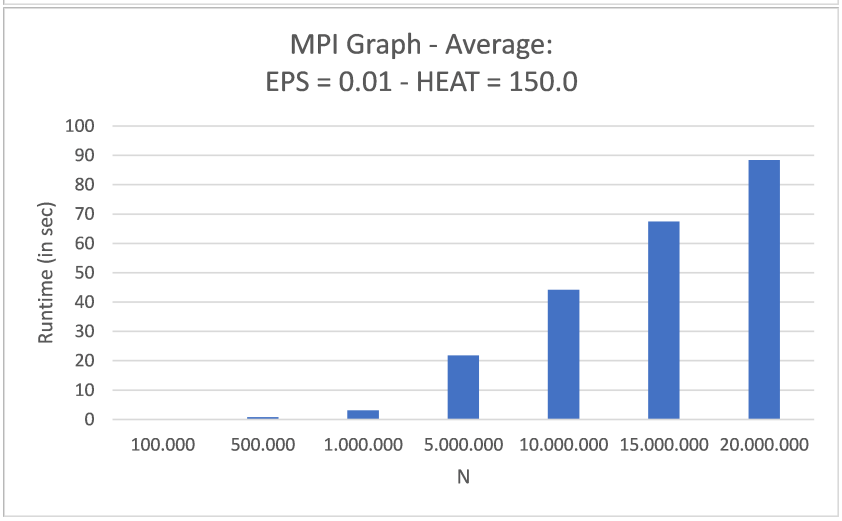
The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0`:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
100,000	0.116156	0.128137	0.125836	0.117138	0.125722	0.121385	0.129094
500,000	0.734729	0.774459	0.714145	0.771836	0.752860	0.789293	0.738912
1,000,000	3.163765	3.180493	3.102440	3.120542	3.165442	3.096422	3.171316
5,000,000	21.550144	21.765802	22.062203	21.850588	21.548727	21.739872	21.962013
10,000,000	43.573832	44.469123	43.938773	44.202815	43.994775	43.976066	44.214623
15,000,000	66.774807	66.221693	65.899295	67.019467	68.250448	69.686135	69.544810
20,000,000	87.494122	87.588305	87.597566	87.752093	88.475759	88.511422	88.192923

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
100,000	0.130014	0.134279	0.132111	0.1259872	0.134279	0.116156
500,000	0.742907	0.741913	0.746682	0.7507736	0.789293	0.714145
1,000,000	3.063459	3.191483	3.148695	3.1404057	3.191483	3.063459
5,000,000	21.993840	21.803740	22.249060	21.8525989	22.249060	21.548727
10,000,000	44.201356	44.569346	45.067063	44.2207772	45.067063	43.573832
15,000,000	69.069942	65.878225	66.053002	67.4397824	69.686135	65.878225
20,000,000	89.151110	90.208496	88.814702	88.3786498	90.208496	87.494122

The graphs below will display the results just obtained:



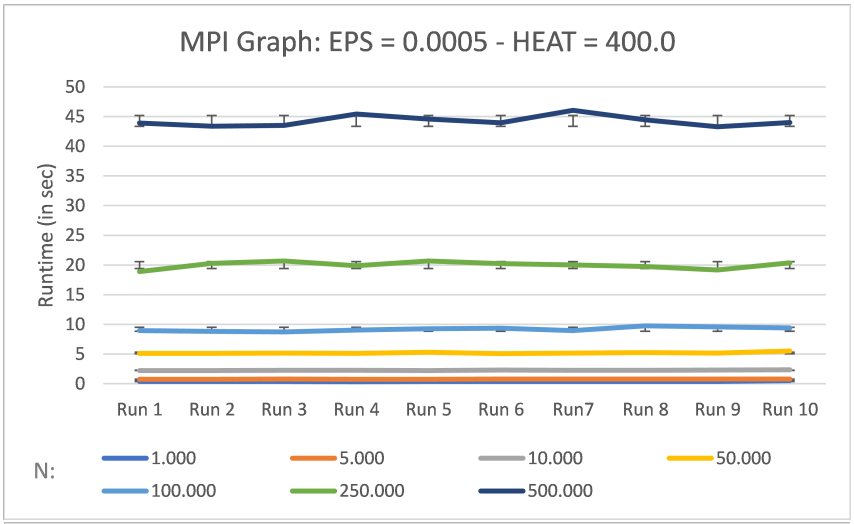


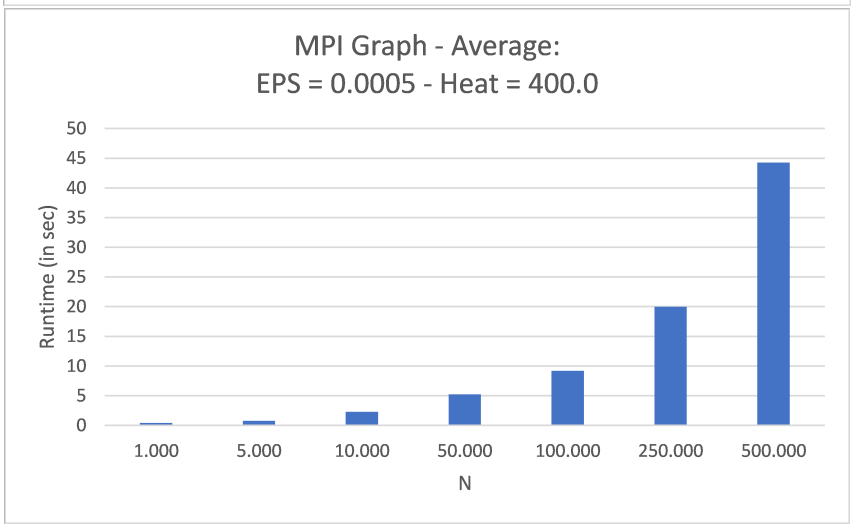
Next to these values, when the parameters `EPS = 0.0005` and `HEAT = 400.0` are used, the results (again expressed in seconds) are:

N	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
1,000	0.375073	0.408064	0.387286	0.369690	0.383956	0.385111	0.391739
5,000	0.761893	0.773472	0.787900	0.755848	0.750110	0.780225	0.781412
10,000	2.202791	2.235692	2.255551	2.269342	2.224050	2.306099	2.279582
50,000	5.105562	5.134397	5.179246	5.140582	5.297360	5.074258	5.164429
100,000	8.956018	8.808440	8.745498	9.023069	9.273222	9.341074	8.946059
250,000	18.894275	20.260282	20.689754	19.858912	20.679163	20.219700	20.013092
500,000	43.918683	43.355759	43.500686	45.420668	44.576451	43.962210	46.053115

N	Run 8	Run 9	Run 10	Average	Slowest Run	Fastest Run
1,000	0.385490	0,383151	0.528827	0.3998387	0.528827	0.369690
5,000	0.783917	0,781221	0.779341	0.7735339	0.787900	0.750110
10,000	2.270368	2.296651	2.345512	2.2685638	2.345512	2.202791
50,000	5.273418	5.178719	5.519762	5.2067733	5.519762	5.074258
100,000	9.766207	9.600961	9.397590	9.1858138	9.766207	8.745498
250,000	19.731063	19.158288	20.370102	19.9874631	20.689754	18.894275
500,000	44.423482	43.265842	43.998594	44.2475490	46.053115	43.265842

The graphs below will display the results just obtained:

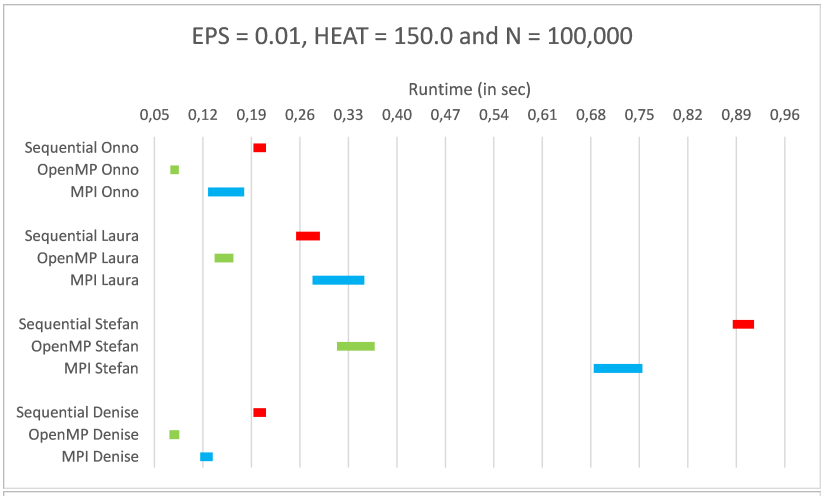


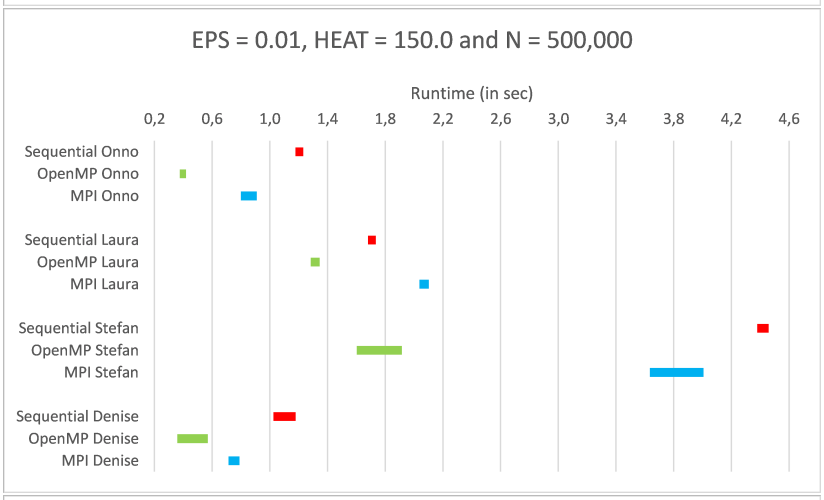


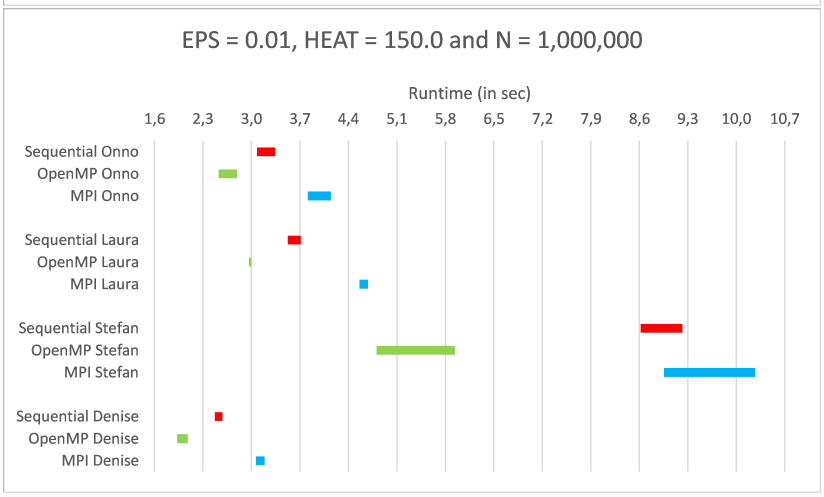
Task 4: Performance

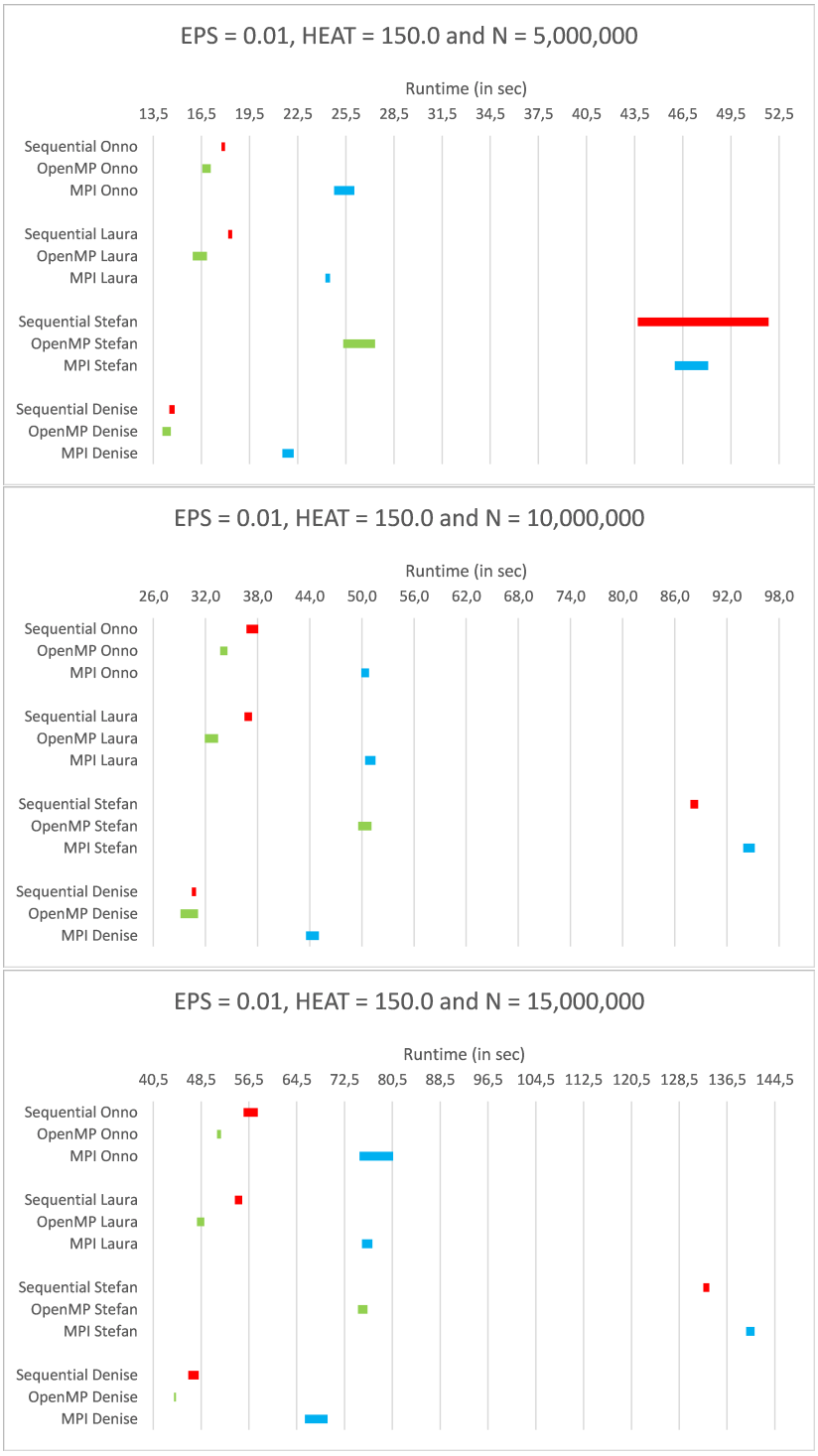
Speedup and Efficiency

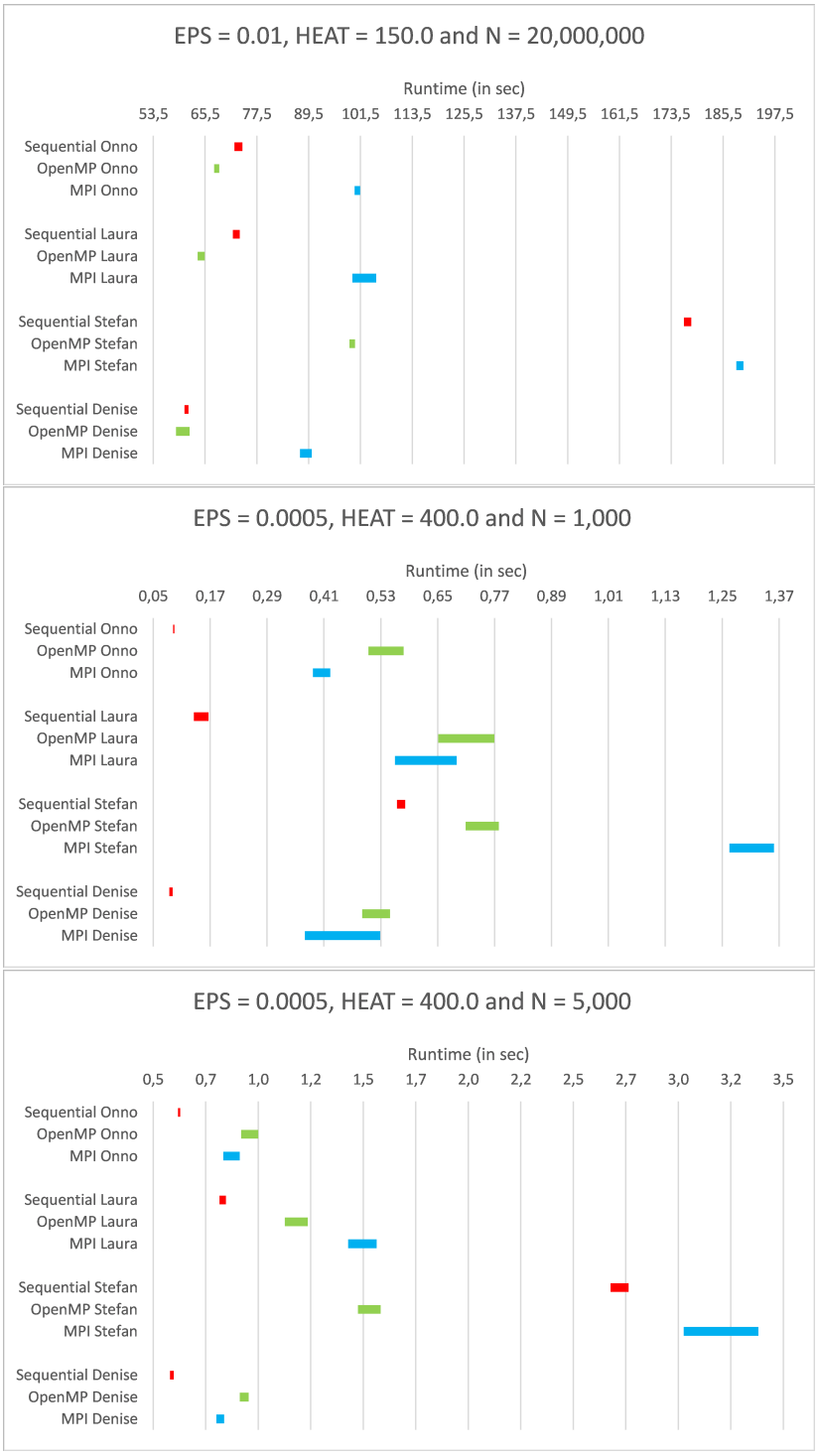
We decided to look at speedups per `N` in combination with different values of `EPS` and `HEAT`. We did this because before testing we all noticed that for different `N`'s, other programs run faster. A quick overview of the results per `N` can be seen below. The graphs represent the ranges of runtimes per `N`, so in these graphs we have an overview of which version (Sequential, OpenMP or MPI) was (on average) the fastest.

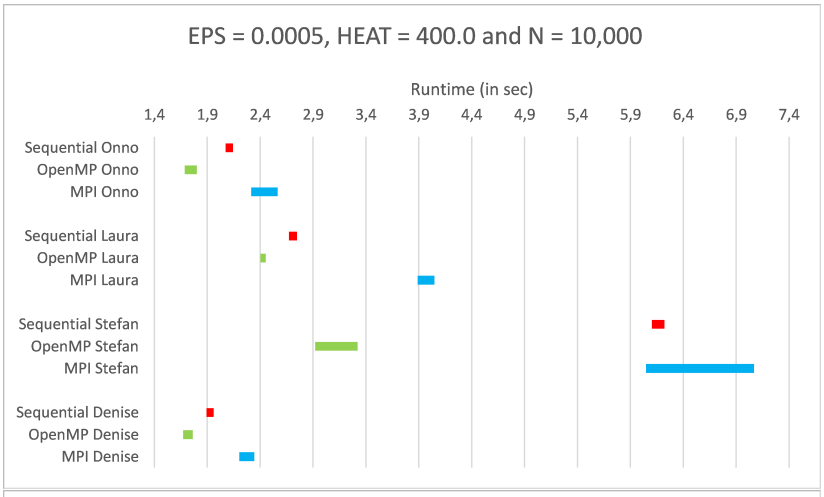


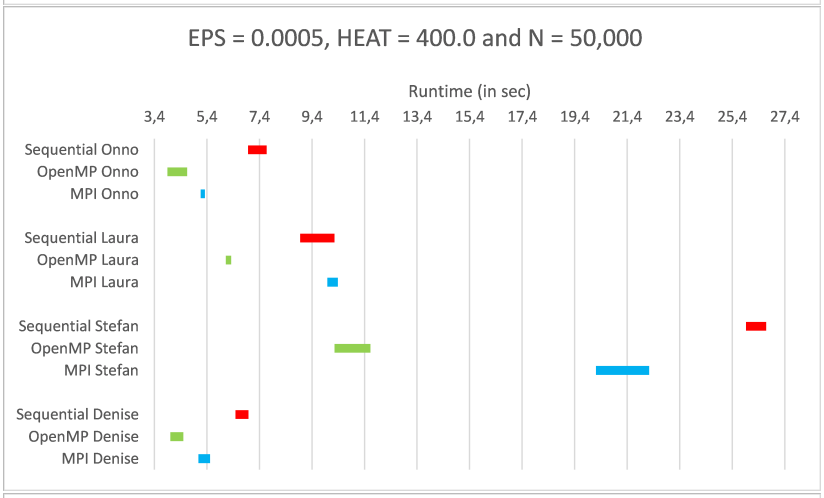


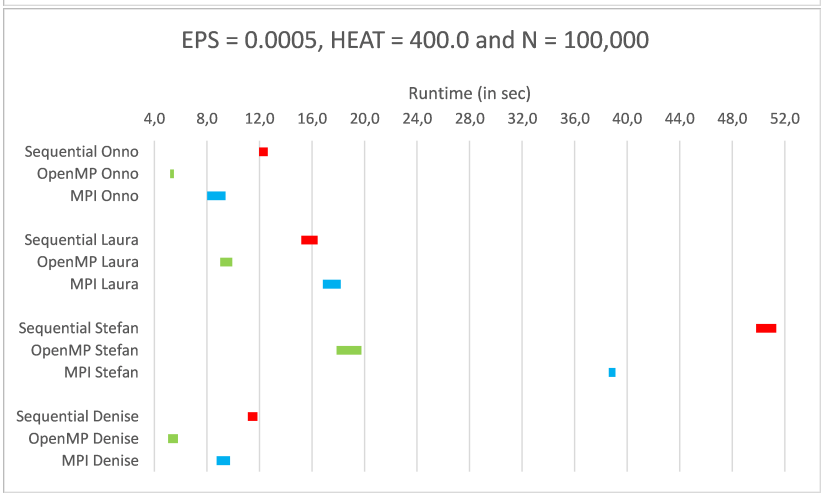


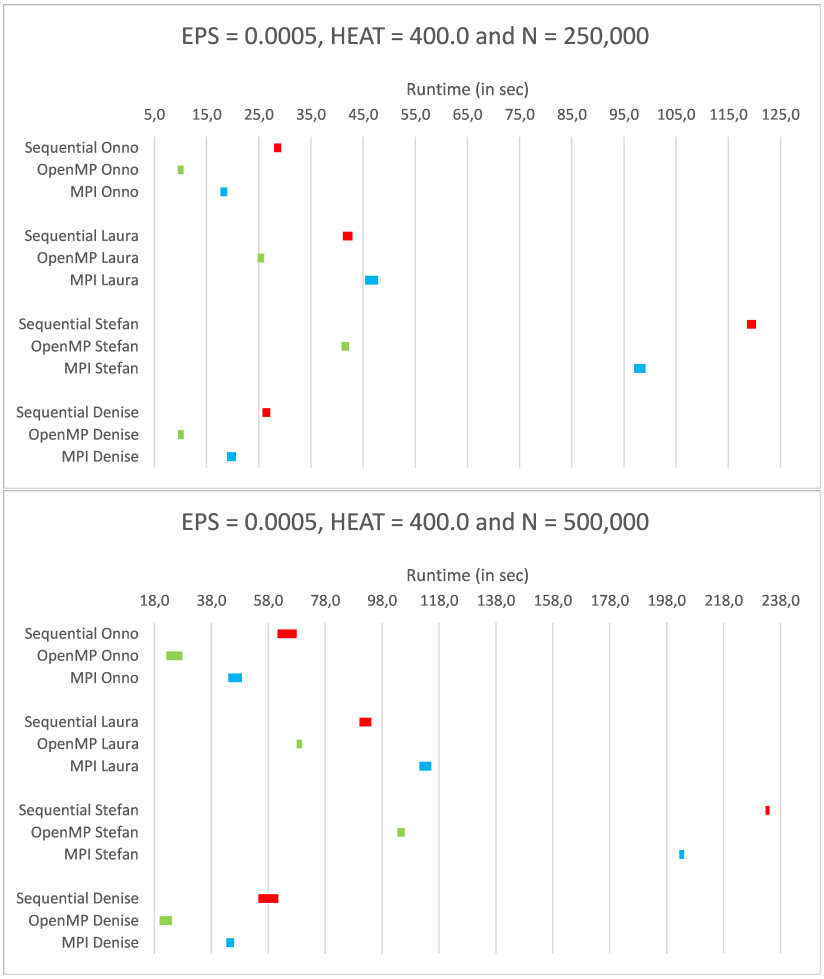

EPS = 0.01, HEAT = 150.0 and N = 5,000,000
Runtime (in sec)
13,5 16,5 19,5 22,5 25,5 28,5 31,5 34,5 37,5 40,5 43,5 46,5 49,5 52,5
Sequential Onno
OpenMP Onno
MPI Onno
Sequential Laura
OpenMP Laura
MPI Laura
Sequential Stefan
OpenMP Stefan
MPI Stefan
Sequential Denise
OpenMP Denise
MPI Denise
EPS = 0.01, HEAT = 150.0 and N = 10,000,000
Runtime (in sec)
26,0 32,0 38,0 44,0 50,0 56,0 62,0 68,0 74,0 80,0 86,0 92,0 98,0
Sequential Onno
OpenMP Onno
MPI Onno
Sequential Laura
OpenMP Laura
MPI Laura
Sequential Stefan
OpenMP Stefan
MPI Stefan
Sequential Denise
OpenMP Denise
MPI Denise
EPS = 0.01, HEAT = 150.0 and N = 15,000,000
Runtime (in sec)
40,5 48,5 56,5 64,5 72,5 80,5 88,5 96,5 104,5 112,5 120,5 128,5 136,5 144,5
Sequential Onno
OpenMP Onno
MPI Onno
Sequential Laura
OpenMP Laura
MPI Laura
Sequential Stefan
OpenMP Stefan
MPI Stefan
Sequential Denise
OpenMP Denise
MPI Denise


EPS = 0.01, HEAT = 150.0 and N = 20,000,000
Runtime (in sec)
53,5 65,5 77,5 89,5 101,5 113,5 125,5 137,5 149,5 161,5 173,5 185,5 197,5
Sequential Onno
OpenMP Onno
MPI Onno
Sequential Laura
OpenMP Laura
MPI Laura
Sequential Stefan
OpenMP Stefan
MPI Stefan
Sequential Denise
OpenMP Denise
MPI Denise
EPS = 0.0005, HEAT = 400.0 and N = 1,000
Runtime (in sec)
0,05 0,17 0,29 0,41 0,53 0,65 0,77 0,89 1,01 1,13 1,25 1,37
Sequential Onno
OpenMP Onno
MPI Onno
Sequential Laura
OpenMP Laura
MPI Laura
Sequential Stefan
OpenMP Stefan
MPI Stefan
Sequential Denise
OpenMP Denise
MPI Denise
EPS = 0.0005, HEAT = 400.0 and N = 5,000
Runtime (in sec)
0,5 0,7 1,0 1,2 1,5 1,7 2,0 2,2 2,5 2,7 3,0 3,2 3,5
Sequential Onno
OpenMP Onno
MPI Onno
Sequential Laura
OpenMP Laura
MPI Laura
Sequential Stefan
OpenMP Stefan
MPI Stefan
Sequential Denise
OpenMP Denise
MPI Denise


EPS = 0.0005, HEAT = 400.0 and N = 10,000
Runtime (in sec)
1,4 1,9 2,4 2,9 3,4 3,9 4,4 4,9 5,4 5,9 6,4 6,9 7,4
Sequential Onno
OpenMP Onno
MPI Onno
Sequential Laura
OpenMP Laura
MPI Laura
Sequential Stefan
OpenMP Stefan
MPI Stefan
Sequential Denise
OpenMP Denise
MPI Denise


EPS = 0.0005, HEAT = 400.0 and N = 50,000
Runtime (in sec)
3,4 5,4 7,4 9,4 11,4 13,4 15,4 17,4 19,4 21,4 23,4 25,4 27,4
Sequential Onno
OpenMP Onno
MPI Onno
Sequential Laura
OpenMP Laura
MPI Laura
Sequential Stefan
OpenMP Stefan
MPI Stefan
Sequential Denise
OpenMP Denise
MPI Denise


EPS = 0.0005, HEAT = 400.0 and N = 100,000
Runtime (in sec)
4,0 8,0 12,0 16,0 20,0 24,0 28,0 32,0 36,0 40,0 44,0 48,0 52,0
Sequential Onno
OpenMP Onno
MPI Onno
Sequential Laura
OpenMP Laura
MPI Laura
Sequential Stefan
OpenMP Stefan
MPI Stefan
Sequential Denise
OpenMP Denise
MPI Denise



Now that we created those graphs and we have a nice summary of our results, we can start drawing conclusions from this: the speedup and the efficiency. The speedup can be calculated by filling in the formula $\text{SpeedUp} = \frac{\text{Time}_{\text{Sequential}}}{\text{Time}_{\text{Parallel}}}$. Next to this, the efficiency can be computed with $\frac{\text{SpeedUp}}{\text{Nr of cores}}$.

For `EPS = 0.01`, `HEAT = 150.0` and `N = 100,000`:

- Onno
  OpenMP: $\text{SpeedUp} \approx \frac{0.2008642}{0.0801965} \approx 2.5$ and $\text{Efficiency} \approx \frac{2.5}{4} \approx 0.6$
  MPI: $\text{SpeedUp} \approx \frac{0.2008642}{0.1445203} \approx 1.4$ and $\text{Efficiency} \approx \frac{1.4}{4} \approx 0.3$

- Laura
  OpenMP: $\text{SpeedUp} \approx \frac{0.2653396}{0.1439866} \approx 1.8$ and $\text{Efficiency} \approx \frac{1.8}{4} \approx 0.5$
  MPI: $\text{SpeedUp} \approx \frac{0.2653396}{0.3263433} \approx 0.8$ and $\text{Efficiency} \approx \frac{0.8}{4} \approx 0.2$

- Stefan
  OpenMP: $\text{SpeedUp} \approx \frac{0.899223}{0.3311311} \approx 2.7$ and $\text{Efficiency} \approx \frac{2.7}{4} \approx 0.7$
  MPI: $\text{SpeedUp} \approx \frac{0.899223}{0.7238038} \approx 1.2$ and $\text{Efficiency} \approx \frac{1.2}{4} \approx 0.3$

- Denise
  OpenMP: $\text{SpeedUp} \approx \frac{0.2008642}{0.0768864} \approx 2.6$ and $\text{Efficiency} \approx \frac{2.6}{4} \approx 0.7$

MPI: SpeedUp $\approx \frac{0.2008642}{0.1259872} \approx 1.6$ and Efficiency $\approx \frac{1.6}{4} \approx 0.4$

For `EPS = 0.01`, `HEAT = 150.0` and `N = 500,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{1.2094048}{0.4001965} \approx 3.0$ and Efficiency $\approx \frac{3.0}{4} \approx 0.8$
  MPI: SpeedUp $\approx \frac{1.2094048}{0.8510918} \approx 1.4$ and Efficiency $\approx \frac{1.4}{4} \approx 0.4$

- Laura
  OpenMP: SpeedUp $\approx \frac{1.6989922}{1.3001024} \approx 1.3$ and Efficiency $\approx \frac{1.3}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{1.6989922}{2.0807286} \approx 0.8$ and Efficiency $\approx \frac{0.8}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{4.4169655}{1.69195} \approx 2.6$ and Efficiency $\approx \frac{2.6}{4} \approx 0.7$
  MPI: SpeedUp $\approx \frac{4.4169655}{3.7691035} \approx 1.2$ and Efficiency $\approx \frac{1.2}{4} \approx 0.3$

- Denise
  OpenMP: SpeedUp $\approx \frac{1.1135128}{0.4569903} \approx 2.4$ and Efficiency $\approx \frac{2.4}{4} \approx 0.6$
  MPI: SpeedUp $\approx \frac{1.1135128}{0.7507736} \approx 1.5$ and Efficiency $\approx \frac{1.5}{4} \approx 0.4$

For `EPS = 0.01`, `HEAT = 150.0` and `N = 1,000,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{3.1620555}{2.6091211} \approx 1.2$ and Efficiency $\approx \frac{1.2}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{3.1620555}{4.0132937} \approx 0.8$ and Efficiency $\approx \frac{0.8}{4} \approx 0.2$

- Laura
  OpenMP: SpeedUp $\approx \frac{3.6094819}{2.9781817} \approx 1.2$ and Efficiency $\approx \frac{1.2}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{3.6094819}{4.6003931} \approx 0.8$ and Efficiency $\approx \frac{0.8}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{8.7530084}{5.2509841} \approx 1.7$ and Efficiency $\approx \frac{1.7}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{8.7530084}{9.3174559} \approx 0.9$ and Efficiency $\approx \frac{0.9}{4} \approx 0.2$

- Denise
  OpenMP: SpeedUp $\approx \frac{2.5052964}{1.9851219} \approx 1.3$ and Efficiency $\approx \frac{1.3}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{2.5052964}{3.1404057} \approx 0.8$ and Efficiency $\approx \frac{0.8}{4} \approx 0.2$

For `EPS = 0.01`, `HEAT = 150.0` and `N = 5,000,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{17.8483042}{16.8029517} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{17.8483042}{25.2332477} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

- Laura
  OpenMP: SpeedUp $\approx \frac{18.307576}{16.0950648} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{18.307576}{24.3213184} \approx 0.8$ and Efficiency $\approx \frac{0.8}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{45.0655793}{26.3652301} \approx 1.7$ and Efficiency $\approx \frac{1.7}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{45.0655793}{46.8299329} \approx 1.0$ and Efficiency $\approx \frac{1.0}{4} \approx 0.2$

- Denise
  OpenMP: SpeedUp $\approx \frac{14.5763372}{14.2359925} \approx 1.0$ and Efficiency $\approx \frac{1.0}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{14.5763372}{21.8525989} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

For `EPS = 0.01`, `HEAT = 150.0` and `N = 10,000,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{37.1085293}{34.0190981} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{37.1085293}{50.2913529} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

- Laura
  OpenMP: SpeedUp $\approx \frac{37.0141301}{32.1580657} \approx 1.2$ and Efficiency $\approx \frac{1.2}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{37.0141301}{50.9967672} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{88.5261132}{50.263286} \approx 1.8$ and Efficiency $\approx \frac{1.8}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{88.5261132}{94.5504147} \approx 0.9$ and Efficiency $\approx \frac{0.9}{4} \approx 0.2$

- Denise
  OpenMP: SpeedUp $\approx \frac{30.7157993}{29.7371271} \approx 1.0$ and Efficiency $\approx \frac{1.0}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{30.7157993}{44.2207772} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

For `EPS = 0.01`, `HEAT = 150.0` and `N = 15,000,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{56.7962217}{51.5682281} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{56.7962217}{76.4367696} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

- Laura
  OpenMP: SpeedUp $\approx \frac{54.6016617}{48.1673474} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{54.6016617}{76.2690616} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{132.962708}{75.4184097} \approx 1.8$ and Efficiency $\approx \frac{1.8}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{132.962708}{140.5686627} \approx 0.9$ and Efficiency $\approx \frac{0.9}{4} \approx 0.2$

- Denise
  OpenMP: SpeedUp $\approx \frac{46.8636393}{44.1035275} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{46.8636393}{67.4397824} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

For `EPS = 0.01`, `HEAT = 150.0` and `N = 20,000,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{73.1159037}{68.0787136} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{73.1159037}{100.8239883} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

- Laura
  OpenMP: SpeedUp $\approx \frac{72.4357397}{64.3693148} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{72.4357397}{101.8070454} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{177.0633677}{99.7525058} \approx 1.8$ and Efficiency $\approx \frac{1.8}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{177.0633677}{189.5687844} \approx 0.9$ and Efficiency $\approx \frac{0.9}{4} \approx 0.2$

- Denise
  OpenMP: SpeedUp $\approx \frac{61.1523169}{59.4747474} \approx 1.0$ and Efficiency $\approx \frac{1.0}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{61.1523169}{88.3786498} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

For `EPS = 0.0005`, `HEAT = 400.0` and `N = 1,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{0.0927465}{0.5266684} \approx 0.2$ and Efficiency $\approx \frac{0.2}{4} = 0.05$
  MPI: SpeedUp $\approx \frac{0.0927465}{0.4056779} \approx 0.2$ and Efficiency $\approx \frac{0.2}{4} = 0.05$

- Laura
  OpenMP: SpeedUp $\approx \frac{0.1425618}{0.6770737} \approx 0.2$ and Efficiency $\approx \frac{0.2}{4} \approx 0.05$
  MPI: SpeedUp $\approx \frac{0.1425618}{0.6339866} \approx 0.2$ and Efficiency $\approx \frac{0.2}{4} \approx 0.06$

- Stefan
  OpenMP: SpeedUp $\approx \frac{0.5726255}{0.745504} \approx 0.8$ and Efficiency $\approx \frac{0.8}{4} \approx 0.2$
  MPI: SpeedUp $\approx \frac{0.5726255}{1.3062954} \approx 0.4$ and Efficiency $\approx \frac{0.4}{4} \approx 0.1$

- Denise
  OpenMP: SpeedUp $\approx \frac{0.0868441}{0.5147173} \approx 0.2$ and Efficiency $\approx \frac{0.2}{4} \approx 0.04$
  MPI: SpeedUp $\approx \frac{0.0868441}{0.3998387} \approx 0.2$ and Efficiency $\approx \frac{0.2}{4} \approx 0.05$

For `EPS = 0.0005`, `HEAT = 400.0` and `N = 5,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{0.573}{0.9078527} \approx 0.6$ and Efficiency $\approx \frac{0.6}{4} = 0.15$
  MPI: SpeedUp $\approx \frac{0.573}{0.8355429} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

- Laura
  OpenMP: SpeedUp $\approx \frac{0.7809393}{1.1259014} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$
  MPI: SpeedUp $\approx \frac{0.7809393}{1.445897} \approx 0.5$ and Efficiency $\approx \frac{0.5}{4} \approx 0.1$

- Stefan
  OpenMP: SpeedUp $\approx \frac{2.6529807}{1.4903165} \approx 1.8$ and Efficiency $\approx \frac{1.8}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{2.6529807}{3.0738706} \approx 0.9$ and Efficiency $\approx \frac{0.9}{4} \approx 0.2$

- Denise
  OpenMP: SpeedUp $\approx \frac{0.5373966}{0.8850976} \approx 0.6$ and Efficiency $\approx \frac{0.6}{4} \approx 0.2$
  MPI: SpeedUp $\approx \frac{0.5373966}{0.7735339} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

For `EPS = 0.0005`, `HEAT = 400.0` and `N = 10,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{2.0923005}{1.7315192} \approx 1.2$ and Efficiency $\approx \frac{1.2}{4} = 0.3$
  MPI: SpeedUp $\approx \frac{2.0923005}{2.4317353} \approx 0.9$ and Efficiency $\approx \frac{0.9}{4} \approx 0.2$

- Laura
  OpenMP: SpeedUp $\approx \frac{2.6881244}{2.4209016} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{2.6881244}{3.9583219} \approx 0.7$ and Efficiency $\approx \frac{0.7}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{6.1624107}{3.0172481} \approx 2.0$ and Efficiency $\approx \frac{2.0}{4} \approx 0.5$
  MPI: SpeedUp $\approx \frac{6.1624107}{6.2301328} \approx 1.0$ and Efficiency $\approx \frac{1.0}{4} \approx 0.2$

- Denise
  OpenMP: SpeedUp $\approx \frac{1.9184696}{1.7157418} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{1.9184696}{2.2685638} \approx 0.8$ and Efficiency $\approx \frac{0.8}{4} \approx 0.2$

For `EPS = 0.0005`, `HEAT = 400.0` and `N = 50,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{7.1588602}{4.0441069} \approx 1.8$ and Efficiency $\approx \frac{1.8}{4} \approx 0.5$
  MPI: SpeedUp $\approx \frac{7.1588602}{5.232921} \approx 1.4$ and Efficiency $\approx \frac{1.4}{4} \approx 0.4$

- Laura
  OpenMP: SpeedUp $\approx \frac{9.4116891}{6.2151242} \approx 1.5$ and Efficiency $\approx \frac{1.5}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{9.4116891}{10.197474} \approx 0.9$ and Efficiency $\approx \frac{0.9}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{26.2351093}{10.5959343} \approx 2.5$ and Efficiency $\approx \frac{2.5}{4} \approx 0.6$
  MPI: SpeedUp $\approx \frac{26.2351093}{21.0792231} \approx 1.2$ and Efficiency $\approx \frac{1.2}{4} \approx 0.3$

- Denise
  OpenMP: SpeedUp $\approx \frac{6.6402267}{4.1263831} \approx 1.6$ and Efficiency $\approx \frac{1.6}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{6.6402267}{5.2067733} \approx 1.3$ and Efficiency $\approx \frac{1.3}{4} \approx 0.3$

For `EPS = 0.0005`, `HEAT = 400.0` and `N = 100,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{12.1679002}{5.3640302} \approx 2.3$ and Efficiency $\approx \frac{2.3}{4} \approx 0.6$
  MPI: SpeedUp $\approx \frac{12.1679002}{8.6680279} \approx 1.4$ and Efficiency $\approx \frac{1.4}{4} \approx 0.4$

- Laura
  OpenMP: SpeedUp $\approx \frac{15.6840243}{9.2827911} \approx 1.7$ and Efficiency $\approx \frac{1.7}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{15.6840243}{17.3735488} \approx 0.9$ and Efficiency $\approx \frac{0.9}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{50.3023532}{18.6919453} \approx 2.7$ and Efficiency $\approx \frac{2.7}{4} \approx 0.7$
  MPI: SpeedUp $\approx \frac{50.3023532}{38.8433271} \approx 1.3$ and Efficiency $\approx \frac{1.3}{4} \approx 0.3$

- Denise
  OpenMP: SpeedUp $\approx \frac{11.3839245}{5.304833} \approx 2.1$ and Efficiency $\approx \frac{2.1}{4} \approx 0.5$
  MPI: SpeedUp $\approx \frac{11.3839245}{9.1858138} \approx 1.2$ and Efficiency $\approx \frac{1.2}{4} \approx 0.3$

For `EPS = 0.0005`, `HEAT = 400.0` and `N = 250,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{28.6586584}{9.967617} \approx 2.9$ and Efficiency $\approx \frac{2.9}{4} \approx 0.7$
  MPI: SpeedUp $\approx \frac{28.6586584}{18.3598552} \approx 1.6$ and Efficiency $\approx \frac{1.6}{4} \approx 0.4$

- Laura
  OpenMP: SpeedUp $\approx \frac{41.8653476}{25.381337} \approx 1.6$ and Efficiency $\approx \frac{1.6}{4} \approx 0.4$
  MPI: SpeedUp $\approx \frac{41.8653476}{46.5371252} \approx 0.9$ and Efficiency $\approx \frac{0.9}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{119.4673981}{41.4818891} \approx 2.9$ and Efficiency $\approx \frac{2.9}{4} \approx 0.7$
  MPI: SpeedUp $\approx \frac{119.4673981}{97.6595776} \approx 1.2$ and Efficiency $\approx \frac{1.2}{4} \approx 0.3$

- Denise
  OpenMP: SpeedUp $\approx \frac{26.6435649}{9.9729271} \approx 2.7$ and Efficiency $\approx \frac{2.7}{4} \approx 0.7$
  MPI: SpeedUp $\approx \frac{26.6435649}{19.9874631} \approx 1.3$ and Efficiency $\approx \frac{1.3}{4} \approx 0.3$

For `EPS = 0.0005`, `HEAT = 400.0` and `N = 500,000`:

- Onno
  OpenMP: SpeedUp $\approx \frac{63.6508194}{23.9982686} \approx 2.7$ and Efficiency $\approx \frac{2.7}{4} \approx 0.7$
  MPI: SpeedUp $\approx \frac{63.6508194}{45.7779876} \approx 1.4$ and Efficiency $\approx \frac{1.4}{4} \approx 0.4$

- Laura
  OpenMP: SpeedUp $\approx \frac{92.1770467}{68.8215309} \approx 1.3$ and Efficiency $\approx \frac{1.3}{4} \approx 0.3$
  MPI: SpeedUp $\approx \frac{92.1770467}{113.3631876} \approx 0.8$ and Efficiency $\approx \frac{0.8}{4} \approx 0.2$

- Stefan
  OpenMP: SpeedUp $\approx \frac{233.3471984}{104.5908589} \approx 2.2$ and Efficiency $\approx \frac{2.2}{4} \approx 0.6$
  MPI: SpeedUp $\approx \frac{233.3471984}{203.4278433} \approx 1.1$ and Efficiency $\approx \frac{1.1}{4} \approx 0.3$

- Denise
  OpenMP: SpeedUp $\approx \frac{56.8160881}{22.2559338} \approx 2.6$ and Efficiency $\approx \frac{2.6}{4} \approx 0.6$
  MPI: SpeedUp $\approx \frac{56.8160881}{44.247549} \approx 1.3$ and Efficiency $\approx \frac{1.3}{4} \approx 0.3$

As we can see for many cases the parallel versions speed up the program. However, there are a few values for which the program is slower when implemented in parallel. We see this happening at:

- For the values `EPS = 0.01`, `HEAT = 150.0` and `N = 100,000`, for Laura the MPI version is sometimes slower than the sequential version.
- For the values `EPS = 0.01`, `HEAT = 150.0` and `N = 500,000`, for Laura the MPI version is slower than the sequential version.
- For the values `EPS = 0.01`, `HEAT = 150.0` and `N = 1,000,000`, for everybody the MPI version is (sometimes) slower than the sequential version.
- For the values `EPS = 0.01`, `HEAT = 150.0` and `N = 5,000,000`, for everybody the MPI version is (sometimes) slower than the sequential version.
- For the values `EPS = 0.01`, `HEAT = 150.0` and `N = 10,000,000`, for everybody the MPI version is slower than the sequential version.
- For the values `EPS = 0.01`, `HEAT = 150.0` and `N = 15,000,000`, for everybody the MPI version is slower than the sequential version.
- For the values `EPS = 0.01`, `HEAT = 150.0` and `N = 20,000,000`, for everybody the MPI version is slower than the sequential version.
- For the values `EPS = 0.0005`, `HEAT = 400.0` and `N = 1,000`, for everybody the MPI version and OpenMP version are both slower than the sequential version.
- For the values `EPS = 0.0005`, `HEAT = 400.0` and `N = 5,000`, for everybody except Stefan the MPI version and OpenMP version are both slower than the sequential version. For Stefan only the MPI version was slower.
- For the values `EPS = 0.0005`, `HEAT = 400.0` and `N = 10,000`, for everybody the MPI version is (sometimes) slower than the sequential version.
- For the values `EPS = 0.0005`, `HEAT = 400.0` and `N = 50,000`, for Laura the MPI version is most of the time slower than the sequential version.
- For the values `EPS = 0.0005`, `HEAT = 400.0` and `N = 100,000`, for Laura the MPI version is slower than the sequential version.
- For the values `EPS = 0.0005`, `HEAT = 400.0` and `N = 250,000`, for Laura the MPI version is slower than the sequential version.
- For the values `EPS = 0.0005`, `HEAT = 400.0` and `N = 500,000`, for Laura the MPI version is slower than the sequential version.

The biggest speedup of OpenMP achieved in our testing is 3.0 with the efficiency 0.8 (for Onno with the values `EPS = 0.01`, `HEAT = 150.0` and `N = 500,000`). The biggest speedup of MPI achieved is 1.6 with the efficiency 0.4 (for Onno with the values `EPS = 0.0005`, `HEAT = 400.0` and `N = 250,000` and for Denise with the values `EPS = 0.01`, `HEAT = 150.0` and `N = 100,000`). More about when which version is fastest, will be elaborated in the subsection Findings.

Strong Scaling

Strong scaling is defined as "how the solution time varies with the number of processors for a fixed total problem size"[2]. We decided to test the strong scaling for 3, 4 and 5 processors. We combined these with the following values:

- For `EPS = 0.01`, `HEAT = 150.0` and `N = 1,000,000`
- For `EPS = 0.0005`, `HEAT = 400.0` and `N = 50,000`

[2]https://books.google.nl/books?id=nePEDwAAQBAJ

The results are processed in tables below. The following results (in seconds) are obtained with the parameters `EPS = 0.01`, `HEAT = 150.0` and `N = 1,000,000` using OpenMP:

Nr. processes	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
3 - Onno	2.432714	2.589978	2.618998	2.694331	2.521308	2.498611	2.567746
4 - Onno	2.581175	2.791696	2.534086	2.745755	2.621896	2.562391	2.634368
5 - Onno	2.658962	2.798058	2.636064	2.584311	2.647842	2.616623	2.736943
3 - Laura	2.889848	2.891931	2.906112	2.917683	2.969267	2.924237	2.884001
4 - Laura	2.968485	2.974365	2.971300	2.963104	2.975534	2.995460	2.984025
5 - Laura	3.028491	2.992291	2.999363	3.037748	3.001987	2.993736	2.988780
3 - Stefan	5.686984	5.454421	5.338725	5.184444	5.277975	5.712368	4.897466
4 - Stefan	4.943805	4.926263	4.805304	4.815827	5.675643	5.261058	5.277606
5 - Stefan	5.264441	5.295044	5.313456	4.915704	4.974398	5.066655	4.953509
3 - Denise	2.120602	1.939247	1.919711	1.949584	2.073669	1.973484	1.941789
4 - Denise	1.927271	1.976801	1.962761	1.968173	2.000223	1.963984	1.979396
5 - Denise	2.256109	2.146841	2.111906	2.197986	2.116704	2.158871	2.087642

Nr. processes	Run 8	Run 9	Run 10
3 - Onno	2.617887	2.762295	2.546799
4 - Onno	2.532814	2.558941	2.528089
5 - Onno	2.783392	2.696517	2.695788
3 - Laura	2.893768	2.889736	2.892460
4 - Laura	2.979384	2.988243	2.981917
5 - Laura	3.000555	3.000118	3.002289
3 - Stefan	4.891232	4.894522	4.953157
4 - Stefan	5.225715	5.639394	5.939226
5 - Stefan	5.008952	5.036899	4.937693
3 - Denise	2.002842	2.049170	2.088693
4 - Denise	1.969136	2.019964	2.083510
5 - Denise	2.099175	2.158310	2.372373

The following results (in seconds) are obtained with the parameters `EPS = 0.0005`, `HEAT = 400.0` and `N = 50,000` using OpenMP:

Nr. processes	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
3 - Onno	4.336985	4.274474	4.302532	4.172401	4.151751	4.325488	4.212163
4 - Onno	3.960601	3.923813	4.645591	3.952784	3.979169	4.092077	4.040087
5 - Onno	10.476311	9.783537	10.436304	9.842171	9.588588	10.022582	9.836966
3 - Laura	6.197651	6.222109	6.258650	6.271258	6.240976	6.242261	6.201431
4 - Laura	6.141555	6.119509	6.148283	6.262788	6.297063	6.278545	6.320907
5 - Laura	9.247740	9.311236	9.219657	9.314528	9.305072	9.441908	9.239172
3 - Stefan	12.169568	11.94499	11.942131	11.937884	11.913028	12.055279	11.996252
4 - Stefan	10.638589	11.624309	10.381951	10.277017	10.40774	10.522455	10.317698
5 - Stefan	18.397122	18.167353	18.213818	18.039124	17.823221	18.433875	19.077051
3 - Denise	4.133032	4.142095	4.220471	4.338099	4.303402	4.029054	4.058405
4 - Denise	4.097483	4.085064	4.159870	4.165293	4.060909	4.009447	4.503518
5 - Denise	54.672169	118.508079	88.149029	132.459984	92.300676	60.656481	112.700194

Nr. processes	Run 8	Run 9	Run 10
3 - Onno	4.168105	4.189223	4.157358
4 - Onno	3.969004	3.985131	3.892812
5 - Onno	9.830529	9.625191	10.292251
3 - Laura	6.144387	6.262251	6.241235
4 - Laura	6.287460	6.149423	6.145709
5 - Laura	9.253972	9.282294	9.274030
3 - Stefan	12.181741	12.140613	12.256477
4 - Stefan	10.805469	10.25888	10.725235
5 - Stefan	18.575969	17.98381	18.067032
3 - Denise	4.027446	4.111672	4.268191
4 - Denise	4,017047	4.054476	4.110724
5 - Denise	72.404081	67.259612	138.448037

The following results (in seconds) are obtained with the parameters `EPS = 0.01`, `HEAT = 150.0` and `N = 1,000,000` using MPI:

Nr. processes	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
3 - Onno	4.388994	3.987837	3.904253	3.752231	4.061236	3.858926	3.858822
4 - Onno	3.920805	4.149516	4.066864	4.008061	4.111227	4.018311	3.815291
5 - Onno	69.426751	70.523023	70.967168	72.729064	71.198403	71.223105	69.939663
3 - Laura	4.924764	4.940371	5.013338	4.953227	5.061254	5.002156	4.979190
4 - Laura	4.683550	4.576172	4.567215	4.581755	4.560134	4.579906	4.602851
5 - Laura	68.037361	69.235896	65.904830	70.389035	69.792207	68.399531	72.413060
3 - Stefan	9.863247	9.87641	10.139821	10.0629	10.015731	9.984232	10.061265
4 - Stefan	9.161755	9.214154	9.313576	10.273787	9.417046	9.231305	9.280888
5 - Stefan	69.714693	69.768445	69.492713	70.054548	69.582643	69.690334	70.147016
3 - Denise	3.219295	3.254591	3.231524	3.227607	3.302230	3.246356	3.273810
4 - Denise	3.163765	3.180493	3.102440	3.120542	3.165442	3.096422	3.171316
5 - Denise	72.532118	72.922560	71.194823	71.290766	73.083499	73.229252	73.275009

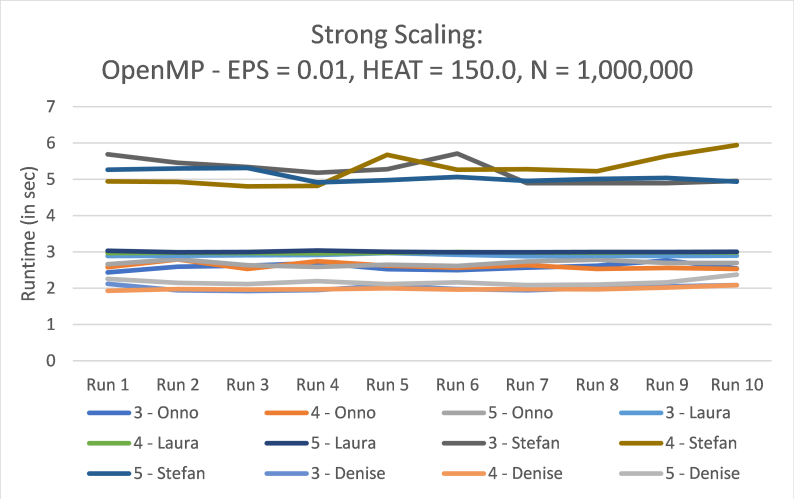
Nr. processes	Run 8	Run 9	Run 10
3 - Onno	3.771712	3.979401	3.889877
4 - Onno	4.102786	4.035977	3.904099
5 - Onno	68.503522	70.595191	71.679231
3 - Laura	4.979450	5.055052	4.965860
4 - Laura	4.611355	4.673762	4.567231
5 - Laura	67.924041	71.049012	72.181785
3 - Stefan	10.061485	9.981881	10.469196
4 - Stefan	9.144791	9.181113	8.956144
5 - Stefan	70.16624	70.883551	69.622309
3 - Denise	3.198061	3.252672	3.203435
4 - Denise	3.063459	3.191483	3.148695
5 - Denise	71.810793	70.775391	72.219237

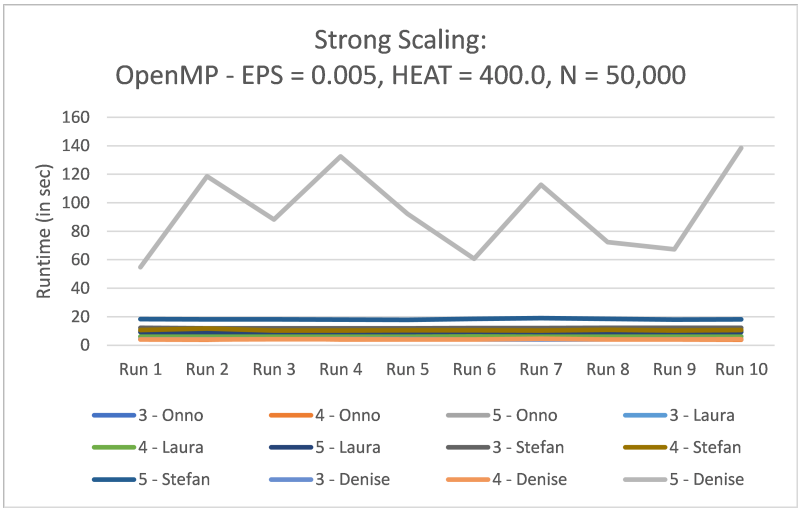
The following results (in seconds) are obtained with the parameters `EPS = 0.0005`, `HEAT = 400.0` and `N = 50,000` using MPI:

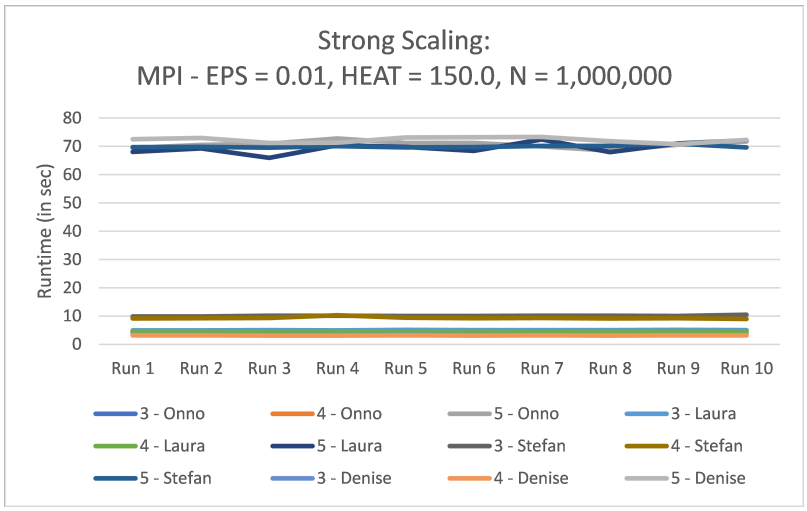
Nr. processes	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
3 - Onno	5.997571	5.824464	5.712093	5.837153	5.840951	5.731272	5.941682
4 - Onno	5.170964	5.248057	5.294552	5.186279	5.223025	5.205526	5.218061
5 - Onno	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible
3 - Laura	12.236665	12.274910	12.201041	10.292209	11.037033	10.437740	10.211264
4 - Laura	10.205544	10.382558	10.244327	9.982976	10.059985	10.317277	10.214539
5 - Laura	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible
3 - Stefan	24.364412	24.583503	24.345693	24.606448	25.164793	25.312852	24.521169
4 - Stefan	20.70557	21.297632	20.912147	22.239051	20.72757	20.838115	21.659528
5 - Stefan	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible
3 - Denise	6.447856	6.232818	6.415312	6.486369	6.476315	6.707439	6.172743
4 - Denise	5.105562	5.134397	5.179246	5.140582	5.297360	5.074258	5.164429
5 - Denise	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible

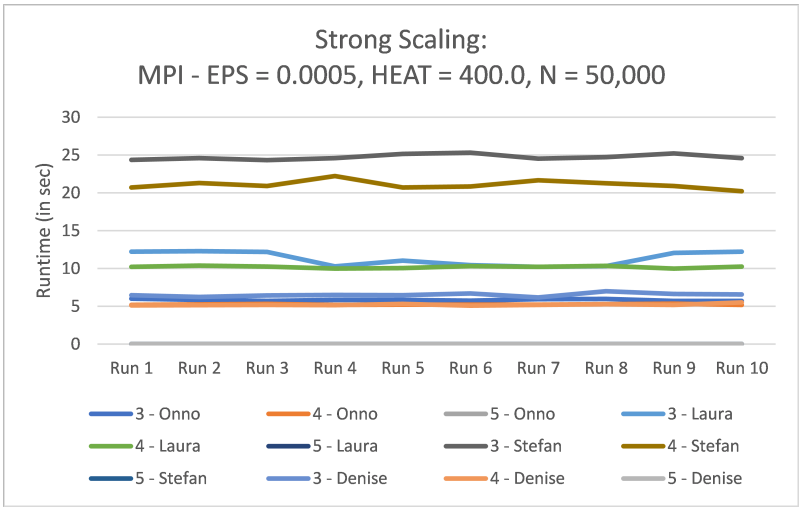
Nr. processes	Run 8	Run 9	Run 10
3 - Onno	5.953687	5.712448	5.717926
4 - Onno	5.291717	5.327093	5.163936
5 - Onno	Infeasible	Infeasible	Infeasible
3 - Laura	10.303923	12.067062	12.233467
4 - Laura	10.329002	9.989322	10.249210
5 - Laura	Infeasible	Infeasible	Infeasible
3 - Stefan	24.720362	25.227473	24.59824
4 - Stefan	21.281168	20.920746	20.210704
5 - Stefan	Infeasible	Infeasible	Infeasible
3 - Denise	6.992051	6.642201	6.569536
4 - Denise	5.273418	5.178719	5.519762
5 - Denise	Infeasible	Infeasible	Infeasible

The results can also be seen in the following graphs:




Strong Scaling:
OpenMP - EPS = 0.005, HEAT = 400.0, N = 50,000
Runtime (in sec)
160
140
120
100
80
60
40
20
0
Run 1
Run 2
Run 3
Run 4
Run 5
Run 6
Run 7
Run 8
Run 9
Run 10
3 - Onno
4 - Onno
5 - Onno
3 - Laura
4 - Laura
5 - Laura
3 - Stefan
4 - Stefan
5 - Stefan
3 - Denise
4 - Denise
5 - Denise


Strong Scaling:
MPI - EPS = 0.01, HEAT = 150.0, N = 1,000,000
Runtime (in sec)
80
70
60
50
40
30
20
10
0
Run 1
Run 2
Run 3
Run 4
Run 5
Run 6
Run 7
Run 8
Run 9
Run 10
3 - Onno
4 - Onno
5 - Onno
3 - Laura
4 - Laura
5 - Laura
3 - Stefan
4 - Stefan
5 - Stefan
3 - Denise
4 - Denise
5 - Denise


Strong Scaling:
MPI - EPS = 0.0005, HEAT = 400.0, N = 50,000
Runtime (in sec)
30
25
20
15
10
5
0
Run 1
Run 2
Run 3
Run 4
Run 5
Run 6
Run 7
Run 8
Run 9
Run 10
3 - Onno
4 - Onno
5 - Onno
3 - Laura
4 - Laura
5 - Laura
3 - Stefan
4 - Stefan
5 - Stefan
3 - Denise
4 - Denise
5 - Denise

From these results we can conclude that generally running with 4 cores is faster than running with 3 or 5 cores. The results with small runtimes sometimes showed that 3 cores were slightly faster. However, for bigger `N` values this difference disappears and 4 is really the best option for running tests on a 4 core machine. What stands out is that for MPI you should definitely not run with 5 cores. Either it gives a major slow down or makes the run infeasible (more than 5 minutes runtime and still not finished).

Weak Scaling

Weak scaling is defined as "how the solution time varies with the number of processors for a fixed problem size per processor"[3]. We decided to test the weak scaling for 3, 4 and 5 processors. We combined these with the following values:

- For `EPS = 0.01`, `HEAT = 150.0` and `N = 750,000` (and thus 3 processors)
  For `EPS = 0.01`, `HEAT = 150.0` and `N = 1,000,000` (and thus 4 processors)
  For `EPS = 0.01`, `HEAT = 150.0` and `N = 1,250,000` (and thus 5 processors)
- For `EPS = 0.0005`, `HEAT = 400.0` and `N = 37,500` (and thus 3 processors)
  For `EPS = 0.0005`, `HEAT = 400.0` and `N = 50,000` (and thus 4 processors)
  For `EPS = 0.0005`, `HEAT = 400.0` and `N = 62,500` (and thus 5 processors)

Here, we increase the size of `N` every time with a chunk of 12,500 (for the values `EPS = 0.0005` and `HEAT = 400.0`) and with a chunk of 250,000 (for the values `EPS = 0.01` and `HEAT = 150.0`), which is the fixed problem size per processor in those cases. The results are processed in tables below. The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0` using OpenMP:

Nr. processes	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
3 - Onno	1.738226	1.717496	1.791157	1.803054	1.797481	1.744424	1.764409
4 - Onno	2.581175	2.791696	2.534086	2.745755	2.621896	2.562391	2.634368
5 - Onno	3.649631	3.775121	3.699657	3.679087	3.649994	3.693451	3.666325
3 - Laura	2.075854	2.072059	2.068295	2.153423	2.106218	2.127145	2.078935
4 - Laura	2.968485	2.974365	2.971300	2.963104	2.975534	2.995460	2.984025
5 - Laura	3.805279	3.809765	3.800407	3.797888	3.807357	3.800944	3.799918
3 - Stefan	3.787538	3.32242	3.758448	3.279455	3.294943	3.454183	3.305639
4 - Stefan	4.943805	4.926263	4.805304	4.815827	5.675643	5.261058	5.277606
5 - Stefan	6.521916	6.299707	6.525152	6.597411	6.475008	6.438677	6.767908
3 - Denise	1.179828	1.198545	1.255148	1.192248	1.199098	1.155333	1.192217
4 - Denise	1.927271	1.976801	1.962761	1.968173	2.000223	1.963984	1.979396
5 - Denise	3.003654	2.939152	2.956051	2.940679	2.719196	2.938381	2.931621

Nr. processes	Run 8	Run 9	Run 10
3 - Onno	1.812676	1.746953	1.775323
4 - Onno	2.532814	2.558941	2.528089
5 - Onno	3.595903	3.699857	3.554998
3 - Laura	2.075819	2.132133	2.094898
4 - Laura	2.979384	2.988243	2.981917
5 - Laura	3.829117	3.820155	3.862485
3 - Stefan	3.291877	3.299356	3.527568
4 - Stefan	5.225715	5.639394	5.939226
5 - Stefan	6.675565	6.497338	7.088211
3 - Denise	1.231213	1.204148	1.271545
4 - Denise	1.969136	2.019964	2.083510
5 - Denise	2.943845	2.966415	2.991624

[3] https://books.google.nl/books?id=nePEDwAAQBAJ

The following results (in seconds) are obtained with the parameters `EPS = 0.0005` and `HEAT = 400.0` using OpenMP:

Nr. processes	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
3 - Onno	3.879832	3.918065	3.811765	3.909348	3.897165	3.953281	3.849204
4 - Onno	3.960601	3.923813	4.645591	3.952784	3.979169	4.092077	4.040087
5 - Onno	10.349838	10.029047	10.443264	10.241771	10.325298	10.220344	10.257551
3 - Laura	5.492245	5.619302	5.524871	5.923323	5.594699	5.520054	5.504576
4 - Laura	6.141555	6.119509	6.148283	6.262788	6.297063	6.278545	6.320907
5 - Laura	9.995419	10.068884	10.047408	10.011195	9.986322	10.013058	10.019657
3 - Stefan	9.724733	9.671235	9.749862	9.544789	9.621811	9.776789	9.597869
4 - Stefan	10.638589	11.624309	10.381951	10.277017	10.40774	10.522455	10.317698
5 - Stefan	19.301692	19.12824	19.199148	18.841724	19.086938	19.017847	19.569093
3 - Denise	3.832631	3.700626	3.864003	3.888503	3.825800	3.819008	3.701267
4 - Denise	4.097483	4.085064	4.159870	4.165293	4.060909	4.009447	4.503518
5 - Denise	84.605910	130.315170	167.967337	154.423751	115.536207	148.668417	41.399758

Nr. processes	Run 8	Run 9	Run 10
3 - Onno	3.760364	4.103758	3.774786
4 - Onno	3.969004	3.985131	3.892812
5 - Onno	10.858494	10.380621	10.734384
3 - Laura	5.876238	5.555981	5.496763
4 - Laura	6.287460	6.149423	6.145709
5 - Laura	10.018856	10.132229	10.039181
3 - Stefan	9.664472	9.626403	10.121037
4 - Stefan	10.805469	10.25888	10.725235
5 - Stefan	18.864105	19.481531	19.248885
3 - Denise	3.757686	3.738660	3.741207
4 - Denise	4.017047	4.054476	4.110724
5 - Denise	92.392088	71.004113	144.439001

The following results (in seconds) are obtained with the parameters `EPS = 0.01` and `HEAT = 150.0` using MPI:

Nr. processes	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
3 - Onno	2.651827	2.489451	2.580253	2.551139	2.530102	2.465267	2.576344
4 - Onno	3.920805	4.149516	4.066864	4.008061	4.111227	4.018311	3.815291
5 - Onno	70.783657	69.256036	68.575486	71.147263	69.607697	69.039192	70.053894
3 - Laura	3.598422	3.611997	3.664652	3.601497	3.619709	3.637032	3.704575
4 - Laura	4.683550	4.576172	4.567215	4.581755	4.560134	4.579906	4.602851
5 - Laura	69.173408	65.246783	69.374623	66.359374	68.255549	75.140670	68.929408
3 - Stefan	7.425639	7.335083	6.996623	6.9801	7.396101	7.061216	7.25704
4 - Stefan	9.161755	9.214154	9.313576	10.273787	9.417046	9.231305	9.280888
5 - Stefan	68.569073	68.836161	69.171023	70.053758	70.272934	68.836161	68.519121
3 - Denise	1.932914	1.920018	2.005637	1.933878	1.934241	1.963960	1.925092
4 - Denise	3.163765	3.180493	3.102440	3.120542	3.165442	3.096422	3.171316
5 - Denise	71.699293	71.877129	72.099063	71.662629	69.546613	69.587406	71.588608

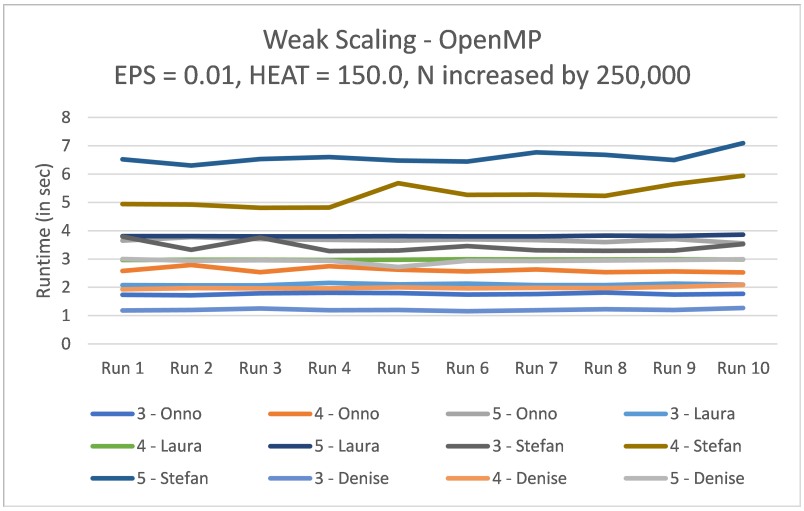
Nr. processes	Run 8	Run 9	Run 10
3 - Onno	2.810984	2.668134	2.630891
4 - Onno	4.102786	4.035977	3.904099
5 - Onno	71.307102	69.627024	70.659404
3 - Laura	3.600649	3.672019	3.600734
4 - Laura	4.611355	4.673762	4.567231
5 - Laura	67.619920	68.620805	69.091984
3 - Stefan	6.978896	7.115232	6.99934
4 - Stefan	9.144791	9.181113	8.956144
5 - Stefan	69.091307	68.49494	68.538108
3 - Denise	1.947403	2.008422	1.988311
4 - Denise	3.063459	3.191483	3.148695
5 - Denise	72.255626	71.734768	72.046103

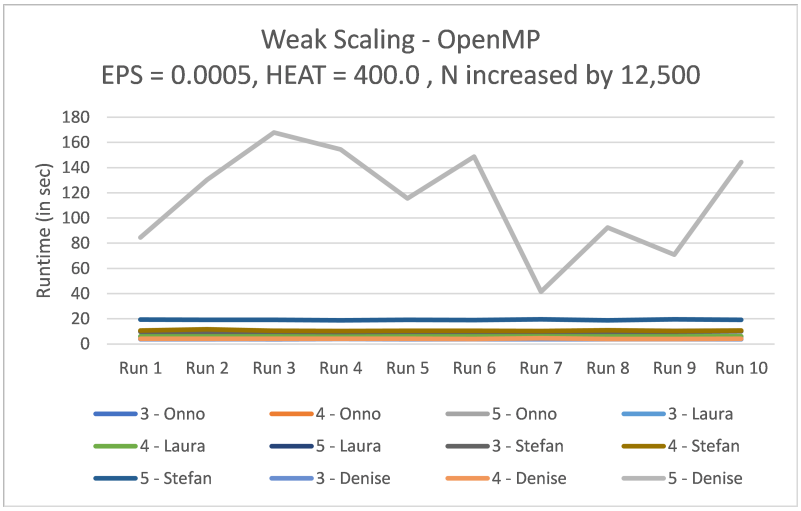
The following results (in seconds) are obtained with the parameters `EPS = 0.0005` and `HEAT = 400.0` using MPI:

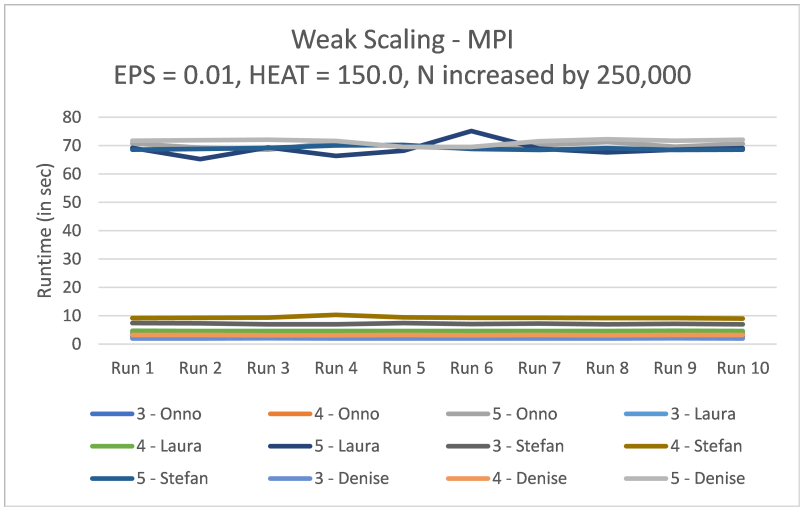
Nr. processes	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7
3 - Onno	5.002867	4.904877	4.959427	5.028234	4.978054	4.892158	4.979517
4 - Onno	5.170964	5.248057	5.294552	5.186279	5.223025	5.205526	5.218061
5 - Onno	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible
3 - Laura	8.254322	8.083335	8.163470	8.285281	8.114074	8.255352	8.248493
4 - Laura	10.205544	10.382558	10.244327	9.982976	10.059985	10.317277	10.214539
5 - Laura	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible
3 - Stefan	19.016018	19.364124	19.445082	20.122531	19.422085	19.930428	20.539136
4 - Stefan	20.70557	21.297632	20.912147	22.239051	20.72757	20.838115	21.659528
5 - Stefan	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible
3 - Denise	4.901247	5.197135	5.067871	5.108378	5.036353	5.070525	5.215389
4 - Denise	5.105562	5.134397	5.179246	5.140582	5.297360	5.074258	5.164429
5 - Denise	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible	Infeasible

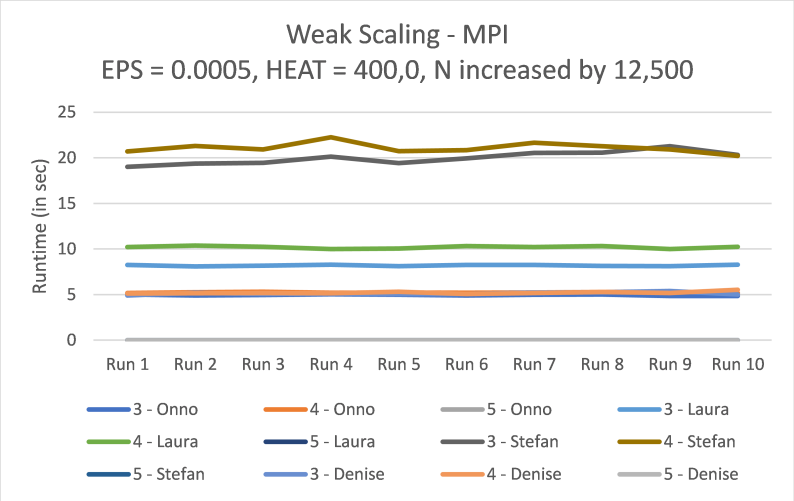
Nr. processes	Run 8	Run 9	Run 10
3 - Onno	5.014181	4.838149	4.830802
4 - Onno	5.291717	5.327093	5.163936
5 - Onno	Infeasible	Infeasible	Infeasible
3 - Laura	8.138762	8.102919	8.275287
4 - Laura	10.329002	9.989322	10.249210
5 - Laura	Infeasible	Infeasible	Infeasible
3 - Stefan	20.575572	21.270356	20.31302
4 - Stefan	21.281168	20.920746	20.210704
5 - Stefan	Infeasible	Infeasible	Infeasible
3 - Denise	5.251378	5.392899	5.009766
4 - Denise	5.273418	5.178719	5.519762
5 - Denise	Infeasible	Infeasible	Infeasible

The results can also be seen in the following graphs:









From these results we can conclude that in all cases the runtime does increase with the size of the input and the number of cores given. We can also again notice that the MPI version is definitely slower than the OpenMP version, and for the case where `EPS = 0.0005` and `HEAT = 400.0`, the MPI version is infeasible to run on all four machines when 5 cores are given.

Effort Discussion

Now, we will discuss and compare the effort that was required to achieve these figures (programming effort and debugging effort). We put a lot of effort into programming and debugging, but even more into making this report (testing, making graphs, writing down results, etc.) in order to end up with an extensive documentation about how our implementations perform. As can be seen from the amount of information in this report, we really wanted to test our program such that we found out how different values for `EPS`, `HEAT` and `N` influenced the different versions of the program.

Findings

In this section, we try to explain our findings. Generally, we see for `EPS = 0.01` and `HEAT = 150.0`, for smaller values of `N`, that the sequential version is the slowest. If we increase these values, MPI will become the slowest version. We also see that OpenMP is in these cases the fastest.

We probably see these results, because MPI will have for these values of `EPS` and `HEAT` a too big overhead caused by communication between ranks. OpenMP does not have this overhead and can therefore outrun the sequential version.

Generally, for `EPS = 0.0005` and `HEAT = 400.0` and for smaller values of `N`, that the parallel versions are slower than the sequential version. If we increase these `N` values a bit, MPI will become the slowest version and OpenMP will become the fastest version. If we increase the size of `N` even more, the sequential version will be the slowest. Thus, with these values, for sufficiently large `N` within reasonable runtime, generally (except for Laura), both our parallel versions will run faster than the sequential version.

We probably see these results, because both MPI and OpenMP will split up the array well enough to be able to outrun the sequential version (for bigger `N` values). The overhead of communicating between ranks and threads is in this case small enough. Because of the proper distribution of data, both parallel versions will be faster in the end than the sequential version.

As we can see from the graphs on pages 56 and 60, for Denise the OpenMP version gave really volatile results when running with 5 cores. For some reason, the HP CoolSense did not start like it usually does while testing our programs. This could have affected the runtimes, but we cannot come to a concrete conclusion since we do not have a lot of information to go on for this. It could be caused by using different kind of hardware specifications or even minor software differences.

Next to this, we noticed that Stefan has some really slow runs where others had faster runtimes. One cause of this might be the fact that he has the oldest processor among all the machines used. Another difference is that he is using Ubuntu as operating system for his VM, while Denise and Onno use Kali and Laura is running Ubuntu directly on her machine (dual boot). He also has hyperthreading deactivated on his machine, which could also cause slower runtimes.

Finally, for Laura we can see that the MPI version for the values `EPS = 0.0005` and `HEAT = 400.0` is always slower than the sequential version (and most of the times also slower than the OpenMP version). This comes in contrast with what happens for the rest of the machines. A reason for this might be that Laura also does not have the newest processor. Additionally, another major difference that could have caused this is the fact that she is running the Ubuntu operating system directly on her machine. However, this last reason does not seem like a logical explanation for the slower MPI runtimes.

After all, it seems like Onno has the most consistent results, which happen to fit with the general conclusions that we have made before.

Further directions of investigation

First of all, a possible further direction of investigation is trying to get the parallel versions even more efficient by maybe using more advanced OpenMP and MPI commands.

Next, what could have been done is investigate further on strong and weak scaling: run these tests with multiple values of `N` and see if there is some kind of connection between these results.

Finally, we could also investigate the case in which we could actually make the assumption that the heat is always placed in the beginning (as shortly described in Task 1: Sequential Evaluation).

Task 5: Team

In this task, we will describe on how we divided up the work.

Our main idea was to work together (so all four of us) as much as we could since we decided that nobody should get a much higher or lower grade: we are in this together and decided to make this assignment as a team. Therefore, we started off with this assignment by making Task 1 together using Discord as communication platform. After this, we started discussing and writing down ideas for Task 2 and Task 3. Since programming already written down ideas is quite difficult with four people at the same time, we decided to split up our time into two groups of two people:

- OpenMP group: Laura Kolijn and Stefan Popa
- MPI group: Onno de Gouw and Denise Verbakel

After running different tests and finding out how slow/fast the implementation was, we grouped again on Discord and discussed about improvements and new ideas. This continued until the moment for testing: we all did our tests individually in order to not let Discord influence our runtime results.

Next to the implementation of the programs, we also needed to write this report and process our findings. We decided to do this together in Overleaf, which made it possible to work on it at the same time. Here, we wrote out the ideas we already discussed and briefly noted in Discord.

In conclusion, we all believe we can say that we really solved this whole assignment as a team. Every member of the team was very involved with the project and has thought about different ideas and possibilities. We agree that we all roughly put in the same amount of effort and that the communication went well between us.