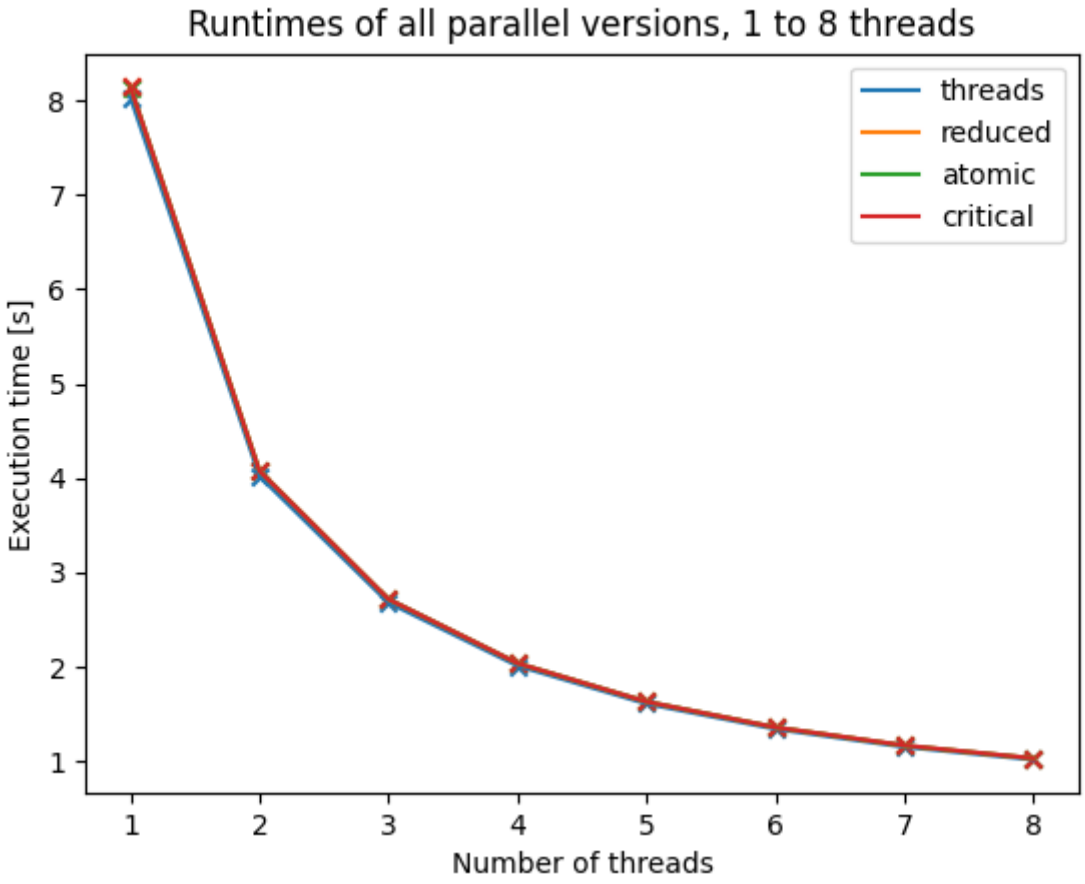
Assignment 3

by Tobias Hosp, Marcel Alexander Huber and Thomas Klotz

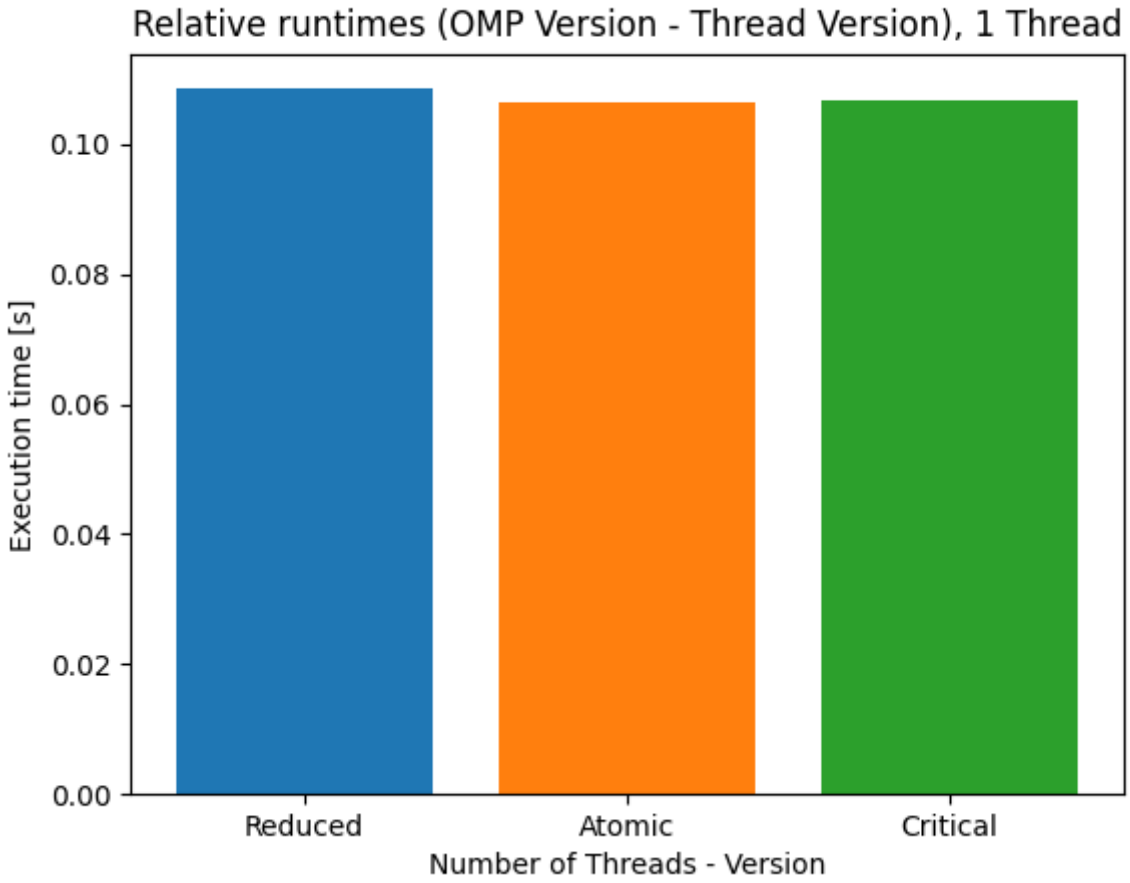
Task 1

Measurements on LCC2

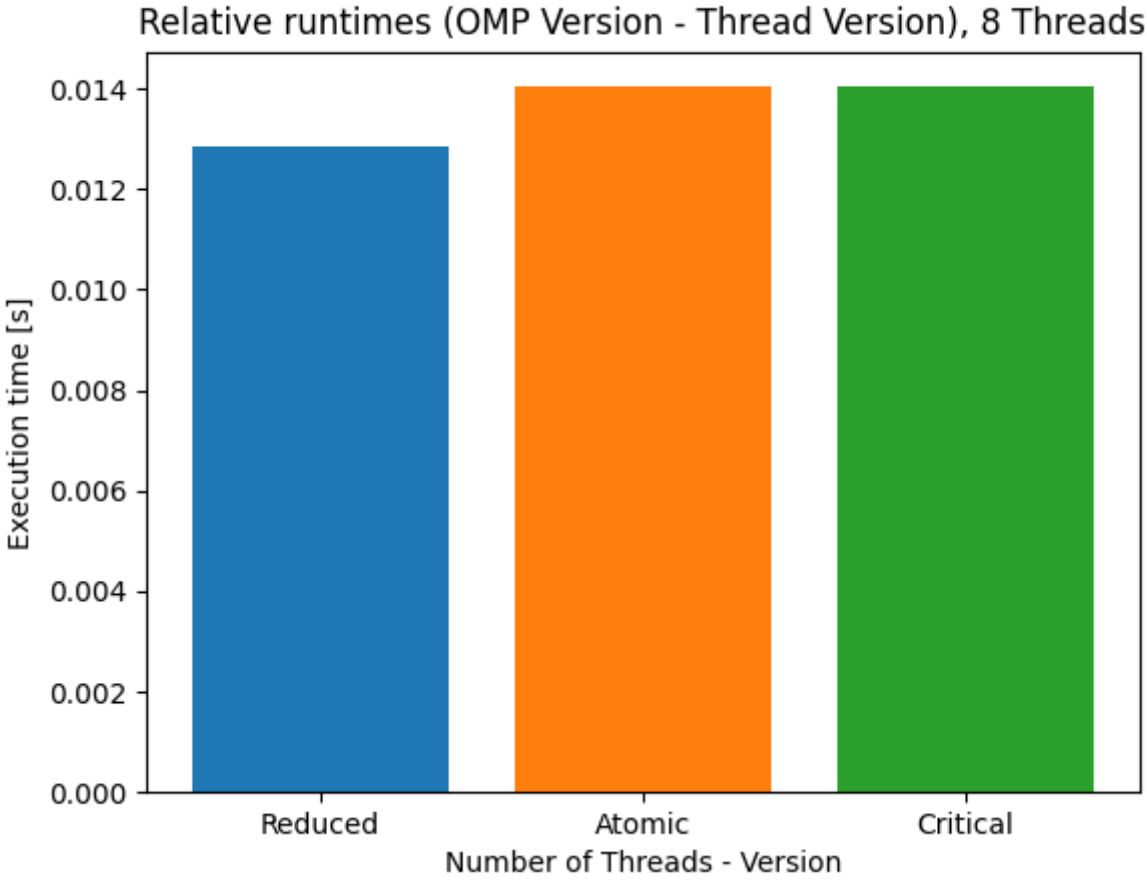
Total execution times for OMP-Versions with `Critical`, `Atomic` and `Reduction` constructs. Execution times from last weeks parallel version using pthreads is also provided. Everything got compiled with -Ofast flag.



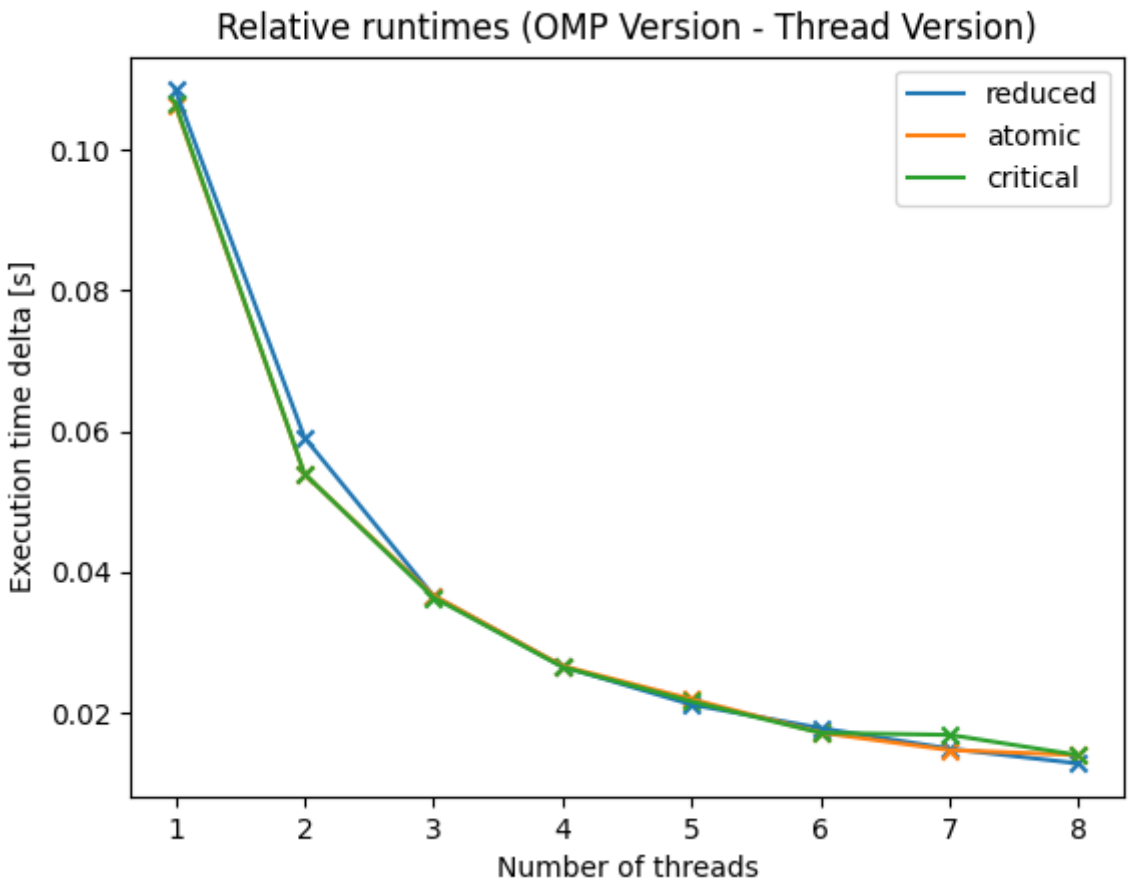
One can observe that the total runtimes seem to be very close together. To get a better picture of the situation, one can watch the following diagrams. All of those print the execution times of the omp versions subtracted by the execution time of the pthread version.



All versions seem to be sligthly slower (0.1s) for only one thread. This is probably due to overhead produced by the runtime system.



The omp versions are still slower then the pthread version when getting executed with 8 threads. However, the delta decreased from 0.1 seconds to ~0.04 to ~0.02 seconds. The differences in execution time of the different omp versions are within margin of error and cannot be reproduced upon repeated execution.



The deltas in omp runtime vs pthread runtime suggest that OMP gets more efficient the more threads are used.

Following data was used to create the plots:

number_of_threads	thread version	reduced version	atomic version	critical version
1	8.035584	8.144062	8.141868	8.142125
2	4.021773	4.080918	4.075714	4.075752
3	2.681191	2.717831	2.717898	2.717617
4	2.011988	2.038604	2.038685	2.038528
5	1.609647	1.630848	1.631626	1.631307
6	1.342244	1.360112	1.359467	1.359431
7	1.151712	1.166655	1.166459	1.168626
8	1.017521	1.030363	1.031557	1.031552

Using /usr/bin/time

- The 'real world time' elapsed from program start to finsish is called **wall clock time**.
- The time the process spends in user mode (program execution, library calls) is called **user time**. If there is more then one thread in use, **user time** shows the sum of the times each thread spent executing its task.
- **System time** is the time spent in Kernel mode (syscalls).

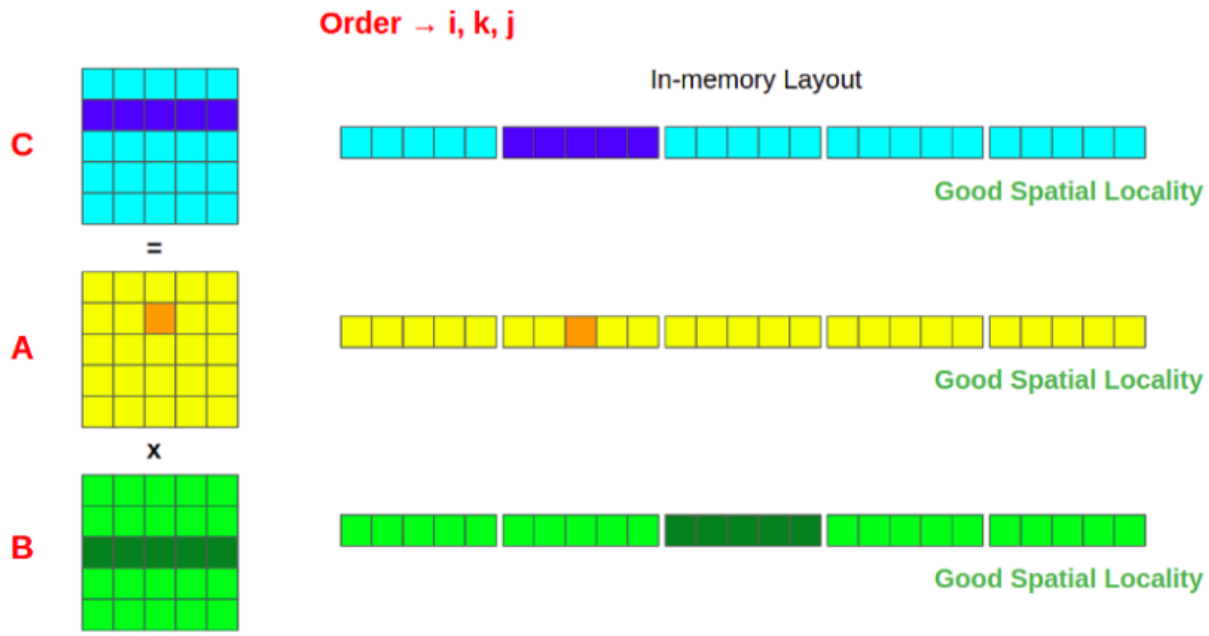
The wall clock time given by `/usr/bin/time` does match the time given by the measurement function of OpenMp almost exactly.
For one thread, wall clock time is equal to user time. For more then one thread, the user time matches the time that the program would take to execute on one thread.

version	**OMP time**	**wall clock time**	**user time**
atomic, 1 thread	8.1428	8.14	8.14
atomic, 8 thread	1.0226	1.02	8.17
critical, 1 thread	8.1428	8.14	8.14
critical, 8 thread	1.0212	1.02	8.16
reduction, 1 thread	8.1427	8.14	8.14
reduction, 8 thread	1.0318	1.03	8.18
pthread, 1 thread	8.0365	8.03	8.03
pthread, 8 thread	1.0077	1.01	8.05

Task2

Changes:

- Parallelize matrix filling with OpenMP and using threadsafe rand_r() function (doesn't help the measured time, cause Measuring begins afterwards)
- Changed the sequence of the nested for-loops from i->j->k to i->k->j so there are less cache-misses:



- Used reduction to simplify the sum of the matrices, because the last for-loop, where "res" is evaluated, is a reduction problem case and be evaluated above where "res_local" is calculated
- Deleting outer-most #pragma, cause there are only for-loops executed and these are handled seperatly

Measurements (Optimization-flag: Ofast)

number_of_threads	unefficient version	efficient version
1	20.71	5.24
2	21.4	2.78
4	22.05	1.84
8	24.23	0.98

Task 3

In this task an iterative variant of merge sort had to be implemented and parallelized with OpenMP.

To parallelize the algorithm, only the line `#pragma omp parallel for schedule(static)` was added before the inner for loop, since it was not applicable for the outer loop, because of the (for this variant of parallelization) invalid iteration increment of the loop.

By this line and the static scheduling type, the loop gets divided into equal-sized chunks as far as possible, so these can be executed parallel by several threads.

```
void mergesort(int32_t array[], int32_t temp[]) {
    for(int subarray_size = 1; subarray_size <= N - 1; subarray_size *= 2) {
#pragma omp parallel for schedule(static)
        for(int subarray_start = 0; subarray_start < N - 1; subarray_start += 2 * subarray_size)
{
            int start = subarray_start;
            int middle = subarray_start + subarray_size - 1;
            int end = min(subarray_start + 2 * subarray_size - 1, N - 1);
            merge(array, temp, start, middle, end);
        }
    }
}
```

Measurements (Optimization-flag: O2)

sequential	parallel (1 thread)	parallel (8 threads)
19.659 s	19.738 s	6.763 s

The measurements are as expected. The algorithm finishes his task significantly faster when parallelized.