Assignment 2

Albin Kjellson, Oliver Specht

26. September 2025

1 Task 1

1.1 a)

Compile with *make task_1*, then run
./task1/numerical_integration T N $[-t <seg>]$ $[-h]$ Use $-h$ to print the help message. Use $-t <seg>$ (optionally) to execute the program with each available thread taking $<seg>$ trapezes at once. More about that in subsection 1.3.

1.2 b)

We first implemented a way that each thread gets the same workload as the others. If the number of trapezes is not dividable by the number of threads, the first few threads get one extra trapeze compared to the rest. The result can be seen in Figure 1 or in the attached excel sheet (first table; it is too big for the report here). It should be noted, that running the shell script "*task1/run_task_1_equally.sh*" produces even more data points which have been omitted for visual reasons.

As can be seen in Figure 1, up until a few thousand trapezes, the execution time stays rather constant which is due to the overhead of creating the threads dominating the execution time. This is also the reason, why more threads equal less performance in that range. Afterwards, the program can take advantage of the multi-threading, allowing almost the same execution time for hundreds of millions of trapezes compared to just a few thousand.

With only a few hundred trapezes, the result gets approximately to π with accuracy of a few decimal places.

1.3 c)

For comparison, we implemented a system, where each thread takes a number (like from a ticketing machine) that determines the next segment of trapezes this thread should calculate. For this, we have a counter variable that is guarded by a mutex to prevent data races. Whenever a thread is free, it increments the counter and calculates the trapeze for the previous number. To try out different

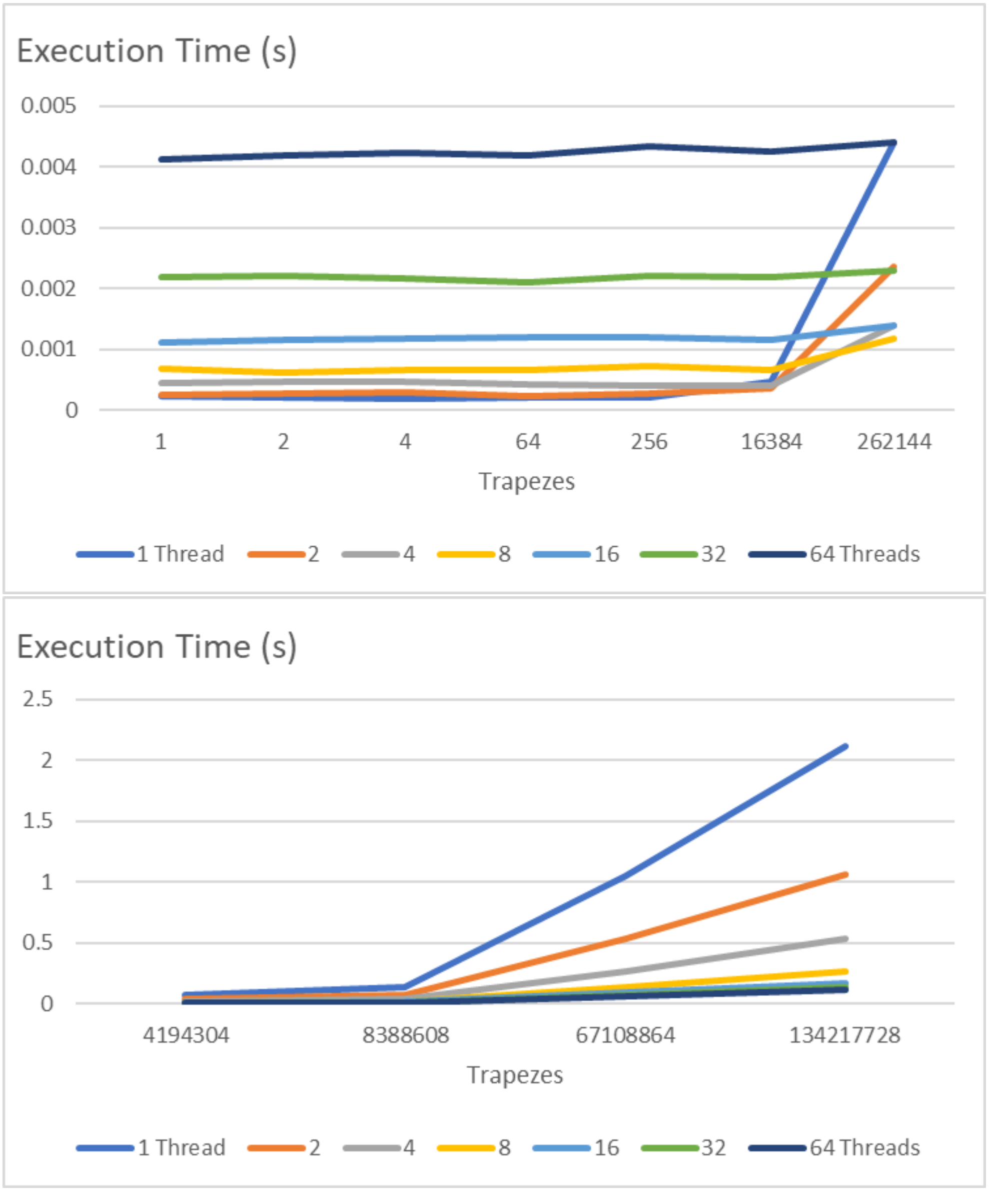


Figure 1: Task 1, execution times for equally distributed work. The diagram is split into two parts to improve visibility.

sizes of segments, it can be passed as an argument when executing with the flag "$-t$ (see subsection 1.1).

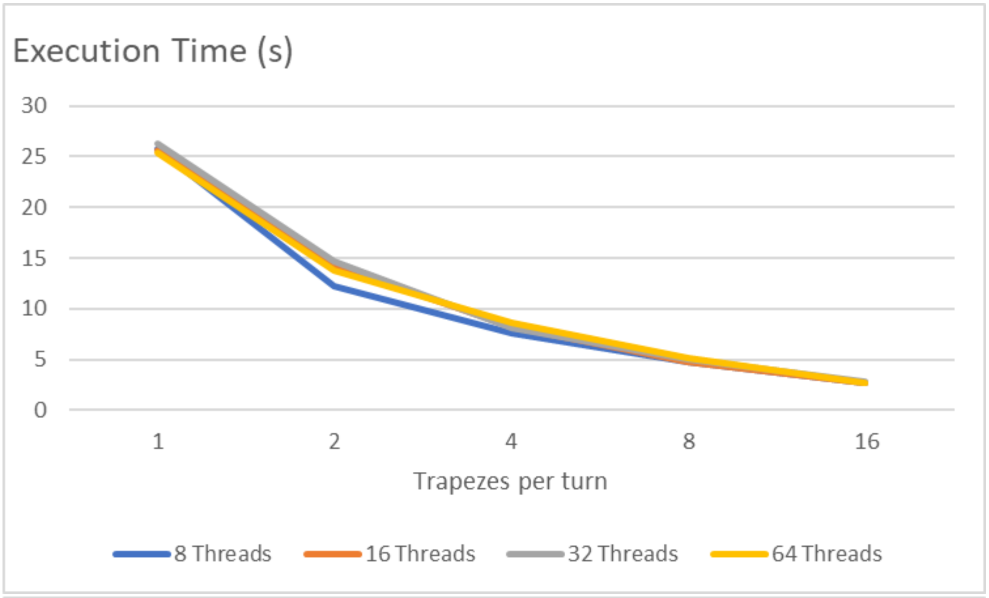
We tried it out for 67108864 (or 2^{26}) trapezes and used 8, 16, 32 and 64 threads. Use "*task1/run_task_1_ticketing.sh*" to run it automatically, then e.g. *./task1/numerical_integration x* 67108864 for comparison, where x is either 8, 16, 32, or 64. The segment sizes (i.e. the trapezes per turn) are all powers of 2 until the total number of trapezes. When using very small segment sizes (like $1 - 32$), the performance is really bad, since the threads spend a huge amount of time at synchronizing with the other threads about the counter variable (i.e. locking and unlocking the mutex and waiting, see Figure 2). The sweet spot is between ~ 1024 and ~ 524288. There, the ticketing version is even a little bit faster than the original version with the same number of threads. If the load is distributed equally among all the threads, the total execution is only as fast as the slowest thread. This drawback is compensated with the ticketing version, since the fastest threads can do more work than the slowest threads. After ~ 524288 trapezes per turn, the execution time skyrockets again because the segments become so large that there are not enough segments for all the threads. At the last data point, only one thread becomes active and has to calculate the whole integral, while all other threads sit idle.

2 Task 2

Compilation
Single-threaded:
```
make task_2_single
```
Multi-threaded:
```
make task_2
```
Run
Single-threaded:
```
./task_2/sieve_single_threaded N
```
Multi-threaded:
```
./task_2/sieve T N
```

No synchronization was used between threads since they act on separate intervals of the set of numbers. This implies that no data races occur and also minimal communication between the threads is required. The threads do, however, need to join in the end for the program to be able to print the full result which means the main thread always needs to wait for the slowest thread. The work is distributed by dividing the set of numbers between $\sqrt{max}$ and max by the number of threads. This could be improved as the number of primes decreases as numbers grow larger. One solution could be that for every new thread created, the chunk size is increased by some factor compared to the previous one.

A large speedup can be observed when increasing the number of cores as can be seen in Figure 3.



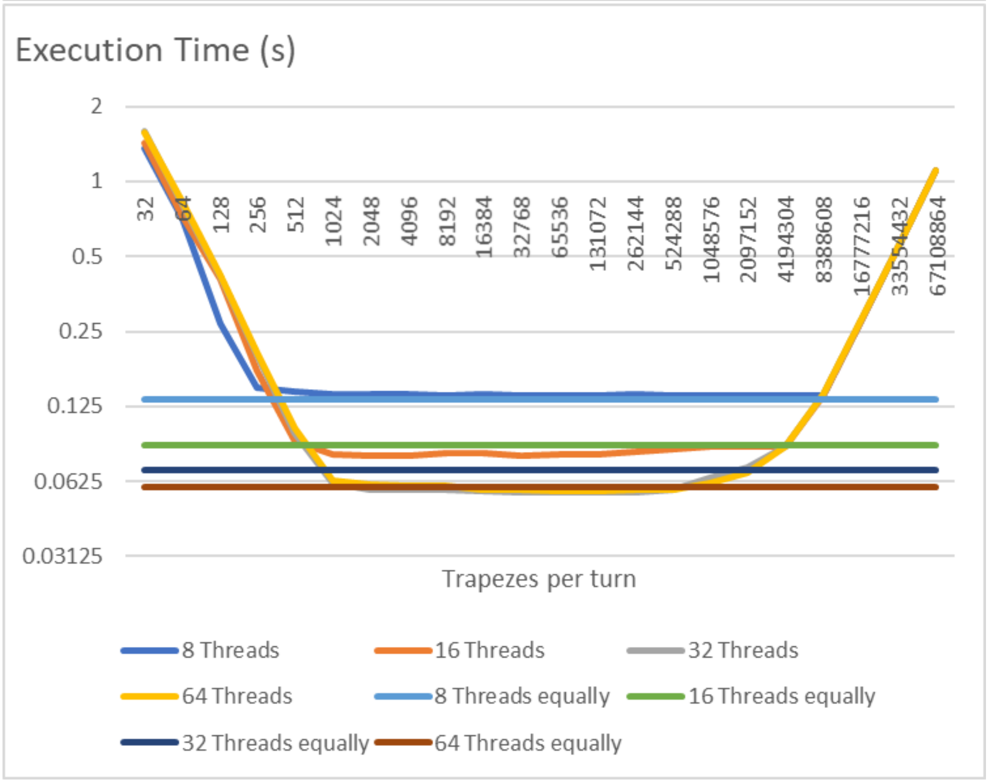


Figure 2: Task 1, execution times for ticketing system vs equally distributed calculating 67, 108, 864 trapezes. The diagram is split into two parts, and the second part uses a logarithmic scale of base 2 to improve visibility.

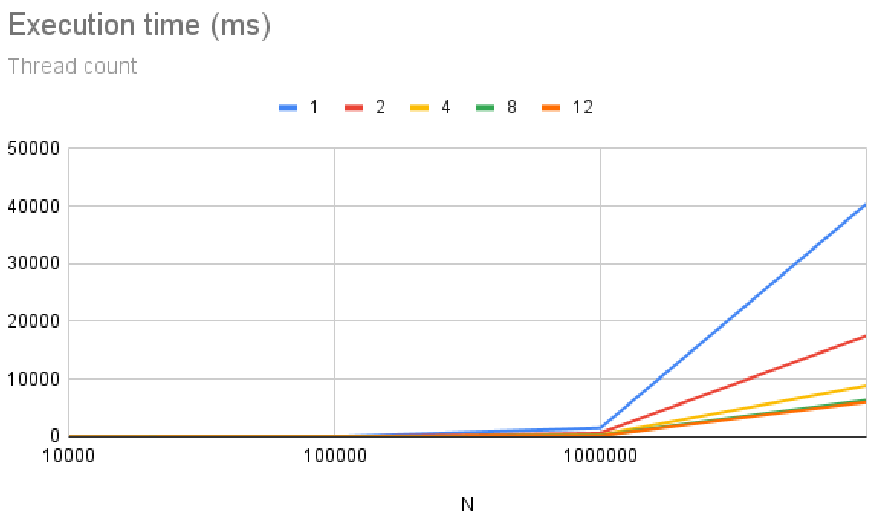


Figure 3: Speedup for different thread counts and N which is the max value to calculate primes.

3 Task 3

3.1 Mutual Exclusion

No, see example implementation. Let two threads increment a shared counter a total of 10000 each. The expected result would be 20000. Before and after the increment, the lock is locked and unlocked. However, most of the time, the total sum is between 10000 and 20000. Sometimes, both threads even end up in a livelock (see subproblem 3). Thus, this protocol does not satisfy mutual exclusion.

To see this behavior, compile with *make task_3* and run the file with input 1 (i.e. *./task3/flaky* 1)

3.2 Starvation-free

No. Suppose we have two threads. One enters the loop, sets the turn to itself, sets the busy flag to true, and exits the function to begin working. Another thread also wants to lock, but stays in the inner loop, because the flag is still true. Now, thread 1 finishes, unlocks (flag is set to false), but locks again immediately, gets by chance the possibility to set turn to itself and the flag to true, and exits again. Meanwhile, thread 2 is still trapped and cannot get the lock $\Rightarrow$ thread 2 starvs.

3.3 Deadlock-free

Yes, it is deadlock-free, however, there is a live-lock. Assume the busy flag is currently set to false. Thread 1 enters the inner loop, sets the turn to itself, exits the inner loop because the flag still false, and sets the flag to true. Now, thread 2 enters the inner loop and sets the turn to itself. Afterwards, thread 1 checks in the outer loop, if busy is still false, which it is not anymore. Thread 1 enters the inner loop again, sets the turn to itself, and is now trapped in the inner loop, because the flag is still set to true. At the same time, thread 2 cannot exit the inner loop, too, because from its perspective the flag has always been set to true. Both threads stay in this loop forever, but are not blocking or waiting $\Rightarrow$ livelock.

To see this behavior, compile with *make task_3* and run the file with input 3 (i.e. $./task3/flaky$ 3)

4 Task 4

Execution instructions:
make task_4 and run *./task4/benchmark_example T* [*list version*], where T is the number of threads and *list version* is an optional parameter to choose between the different list versions. Each version corresponds to the subtask (e.g. version 1 is coarse-grained locking). If not specified or invalid parameter, version 0 (non-thread-safe list) is chosen.

The coarse-grained locking mechanisms were very easy to implement, since you only need to lock the list for the whole operation. So lock the list at the entry point of each function and unlock the list at each exit point of each function. Using a mutex and the MCS lock for the fine-grained locking was a little bit more challenging, since you need to lock and unlock multiple nodes in succession. In the end, in each method, the thread uses a moving frame. It advances to the next node, which is guaranteed to be either locked or free (in any case, it is guaranteed to not be modified in the meantime), and then releases the previous node. Only when it comes to the very first node, there where some problems. It was important to lock the list as a whole, since modifying the head of the list can lead to invalid states for other threads. One example: thread 1 gets node 1, thread 2 also accesses node 2, locks it, and deletes it, now thread 1 tries to lock a node that no longer exists. This is prevented by the head_guard, that is locked while a thread either starts into the list or is currently modifying the first node.

Implementing the MCS lock turned out to be more difficult than expected. We are still running into dead- or livelocks, when using multiple threads, but after hours of debugging we could not figure out why.

As can be seen in Figure 4 there is no speedup over the non-thread-safe single threaded execution. This is probably because the locking overhead is too large for the implemented cases. It would be interesting seeing the results of the other locking methods as this does not seem to give any improvement whatsoever.

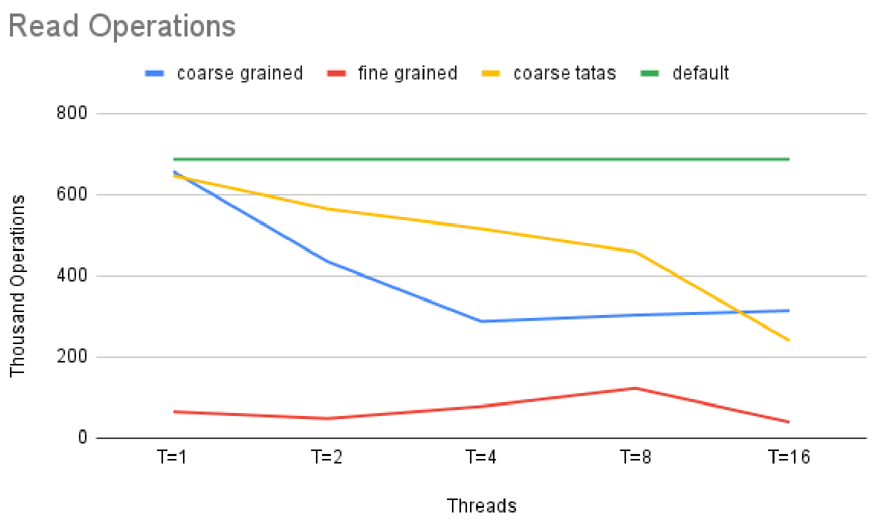


Figure 4: Speedup for read operations.

One aspect that is interesting is that the non-thread-safe version would not work for some scenarios where multiple threads are needed, e.g. for a server that should be able to handle incoming requests concurrently. In that case, the implementations in this part could be useful. Looking at the table again, it can be noted that the coarse tatas version performs better up to about 8 threads for the read operations. For the fine grained locking method, it seems as though its speed increases when increasing the number of threads. This could imply that it scales better than the other methods.

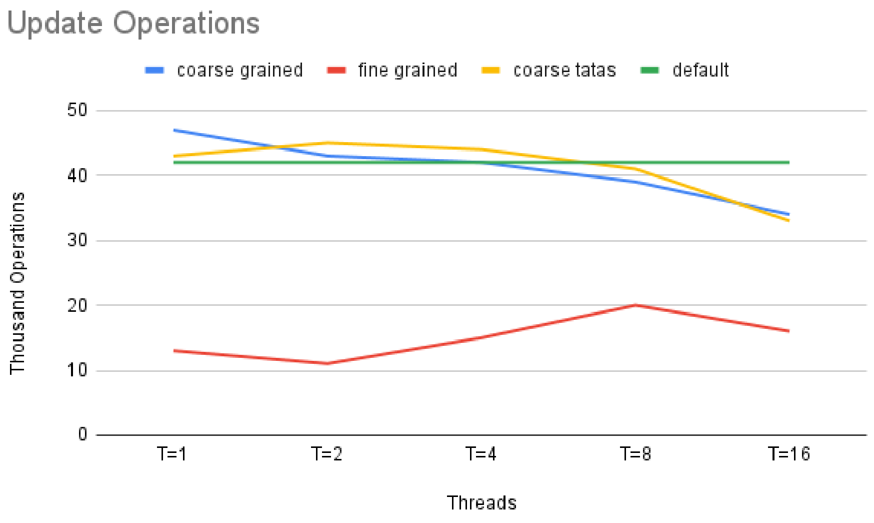


Figure 5: Speedup for update operations.

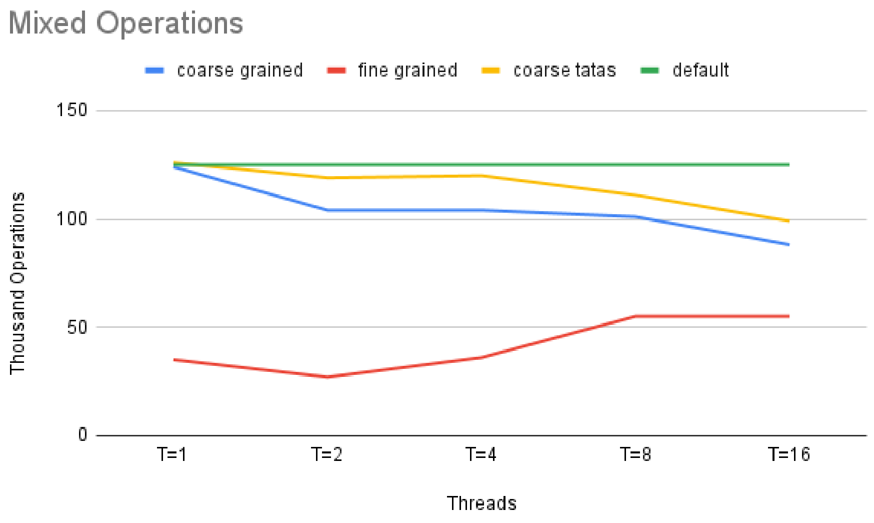


Figure 6: Speedup for mixed operations.