Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing **2022**

Student: Nicolai Hermann Discussed with: Oliver Tryding, Kaveh Boghraty, Valentin Bacher, Florian Kloeppner

---

Solution for Project 2

Due date: 26.10.2022 (midnight)

HPC 2022 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP [10 points]

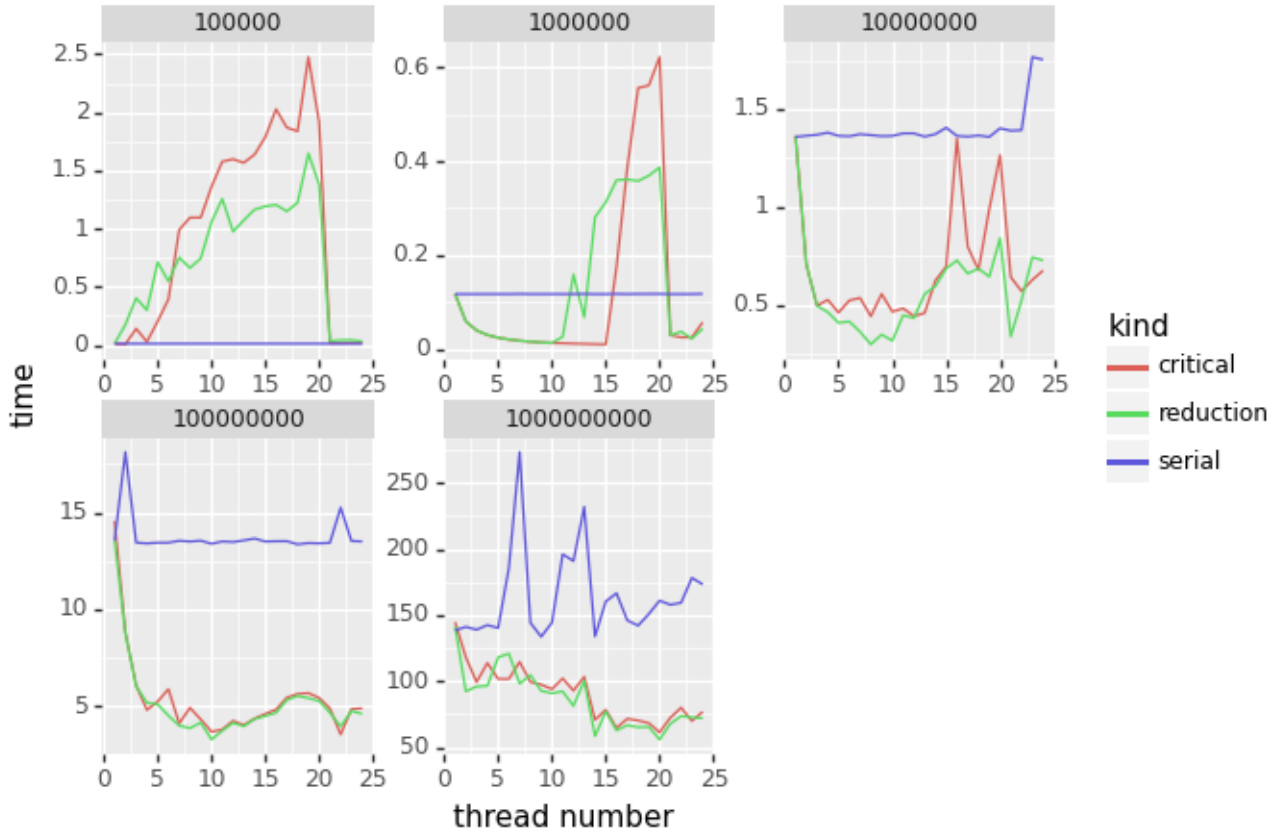

Figure 1: Performance comparison in seconds between different array sizes depicted in the header of each subplot. Each plot shows how all three implementations scale for a increasing number of threads.

Fig. 1 shows the computation time needed using different parallelism methods for different array sizes and different numbers of threads. Note that the critical implementation was further improved as only using `#pragma omp parallel for` and `#pragma omp critical` resulted in enormous computation times. This however was expected since making the only operation in the `for` loop critical removes the parallelism again since each iteration of the for loop can only be executed by one thread at a time. Additionally, the overhead of making sure that only one thread can execute an iteration a, slows the execution down dramatically. For the largest array size, the critical implementation took over 2h for one run, which was intractable to compute for all thread counts. The optimization included using the `for` clause and creating partial sums for every thread and adding them in a critical section to the main sum at the end. This comes close to what the `reduction` clause does which explains the similarity in performance. Using `atomic` instead of `critical` would have increased the performance even further.

Parallelizing the dot products is getting more efficient after $n = 100,000$ with more than 21 threads. After $n = 1,000,000$ it will always be faster when using more than 1 thread. In Fig. 2 and Fig. 3 the parallel efficiency can be seen. In general the efficiency is not great, implying, that adding more threads will only increase the computational speed slightly.

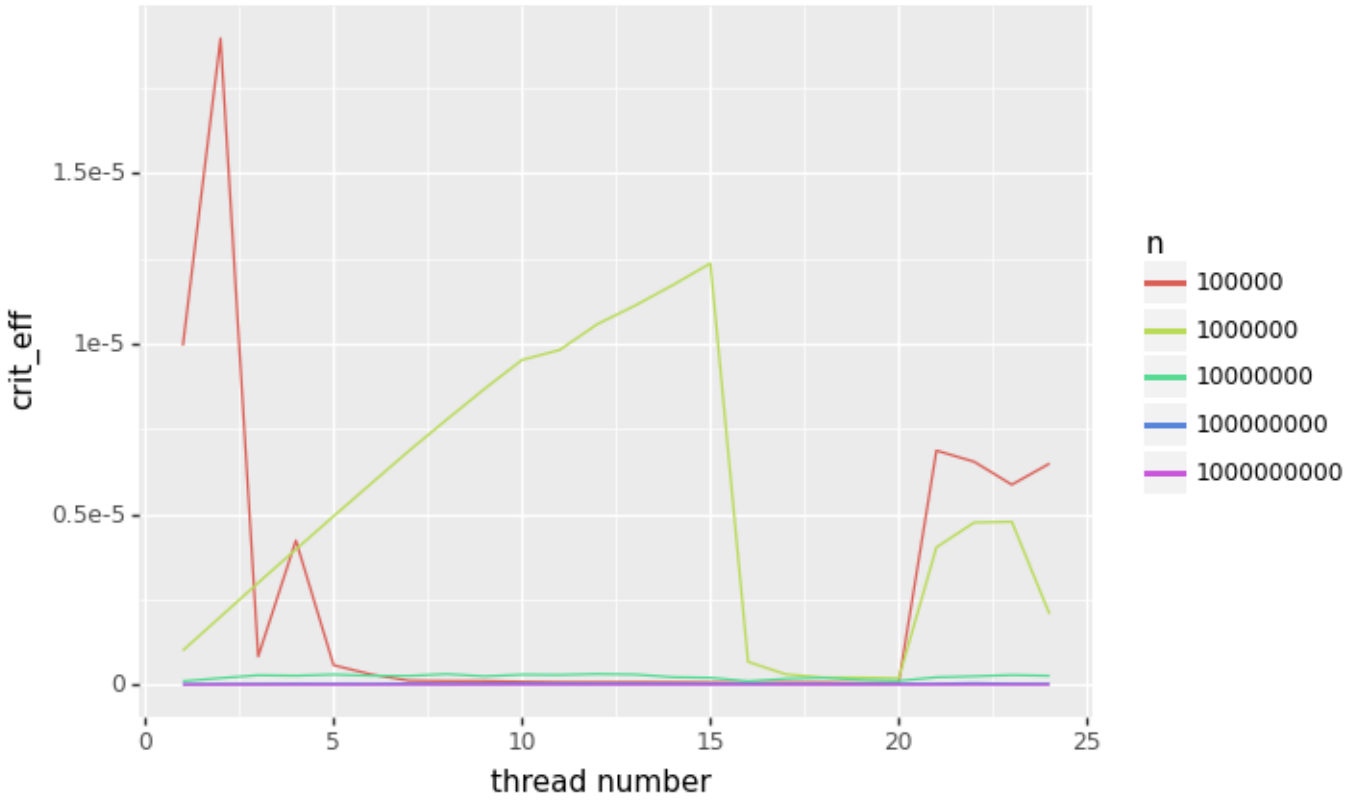

Figure 2: Parallel efficiency for the critical implementation for multiple array sizes n and thread counts.

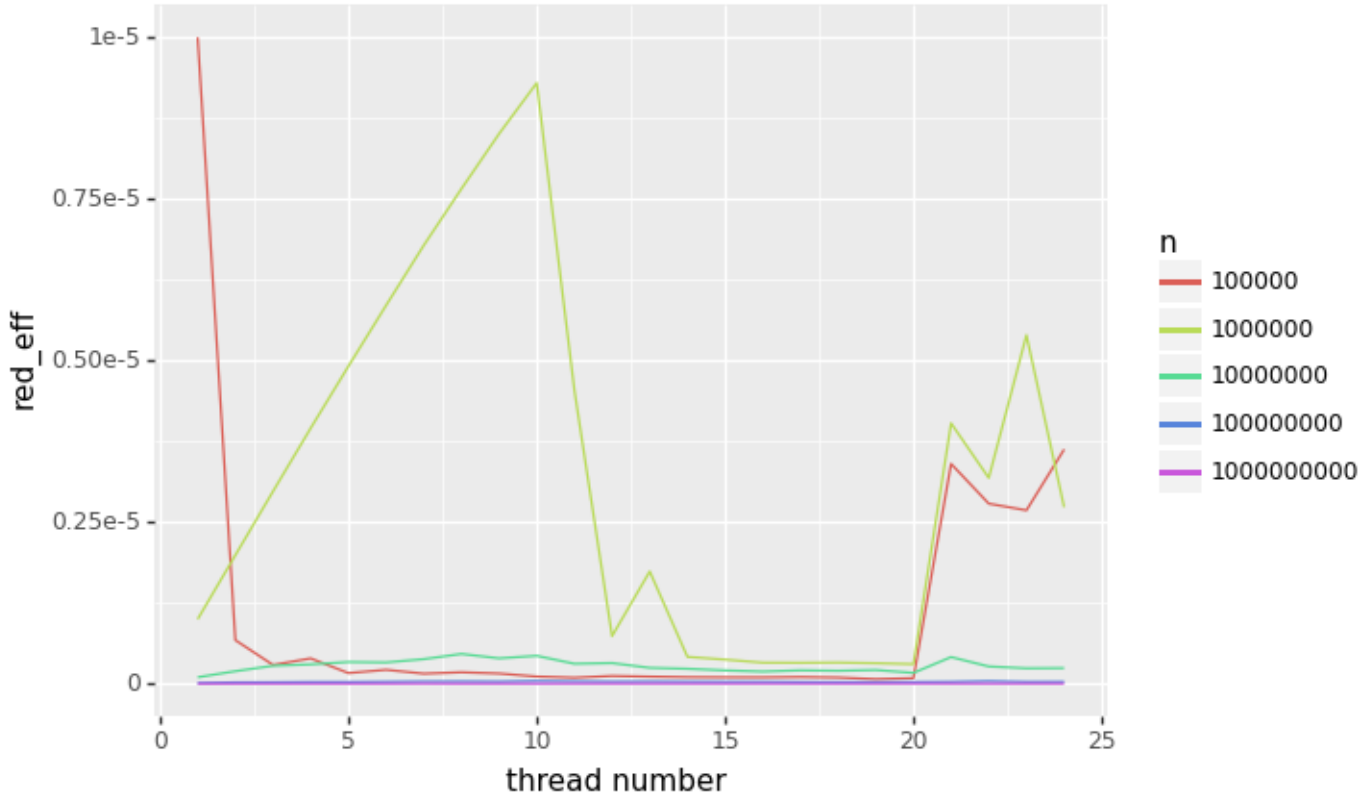

Figure 3: Parallel efficiency for the reduction implementation for multiple array sizes n and thread counts.

2. The Mandelbrot set using OpenMP [30 points]

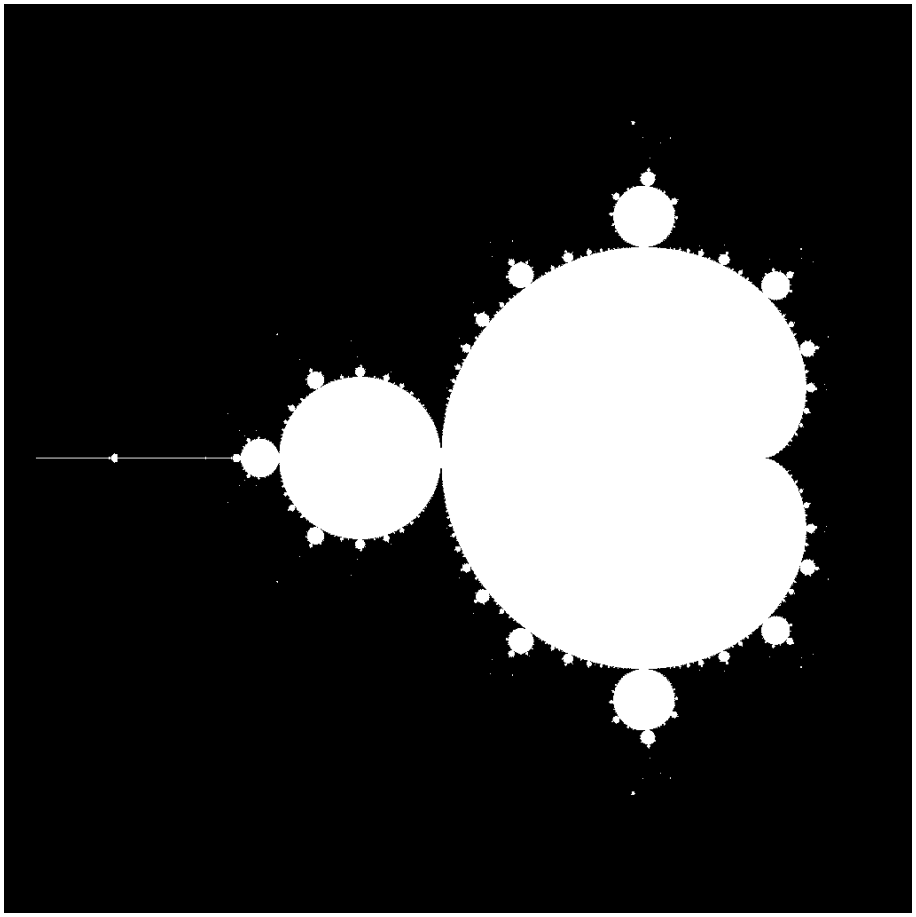

Figure 4: Parallel computed mandel brot.

See the resulting mandel brot in Fig. 4. Since every pixel can be computed independently it is possible to use the `collapse` pragma to parallelize both for loops. In addition since not every pixel computation takes an equal amount of computation, the `schedule(dynamic)` is beneficial as it is capable of distribution work unequally between threads to match the variant computation times of single iterations. This drastically increases the performance of the mandel brot computation. Besides those adaptions some variable declarations were moved into the inner for loop to be private for each thread, `cx` and `cy` are computed in for every pixel and the total iteration counter must be incremented within an `atomic` clause. Find the computational results in Figs. 5, 6, 7 and 8.

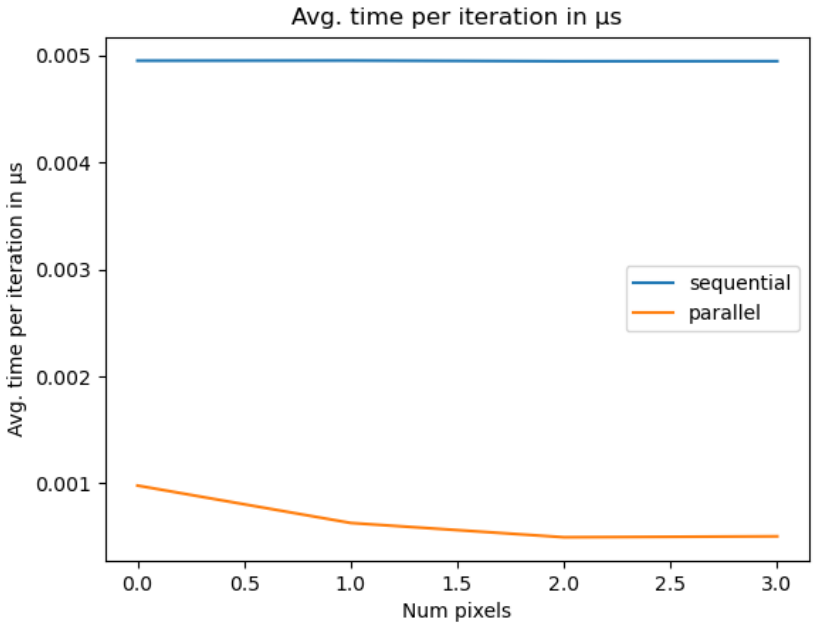

Figure 5: Average time per iteration in μs where one iteration is one iteration of the innermost while loop calculating the con/divergence of the mandel brot series.

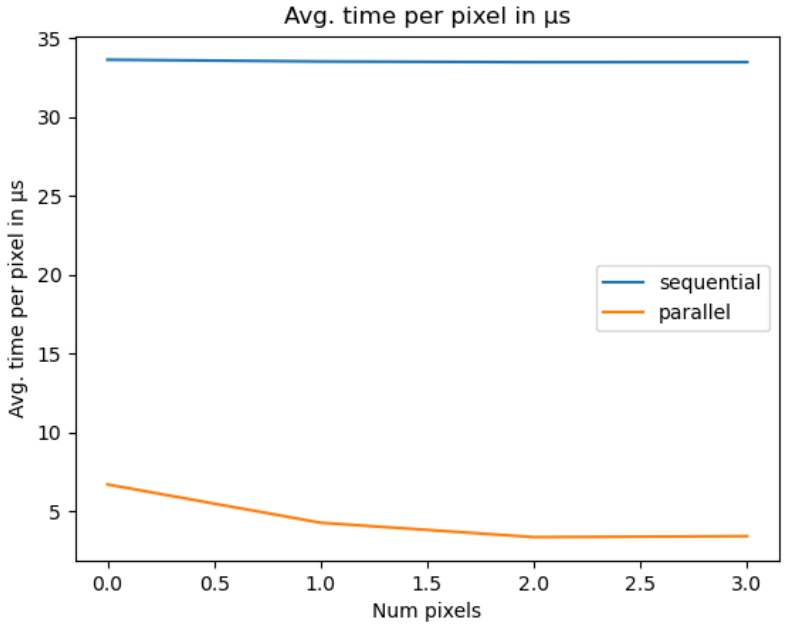

Figure 6: Average computation time per pixel in μs for the sequential and parallel implementation.

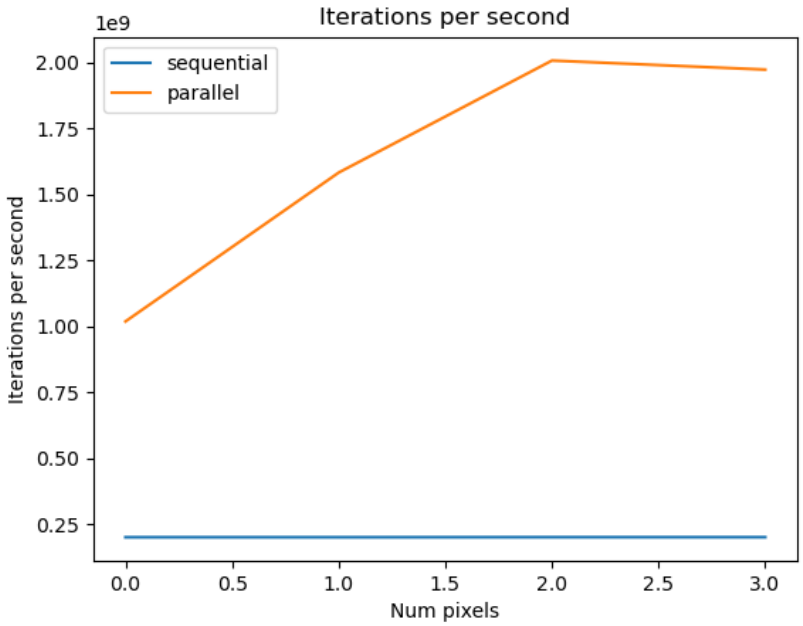

Figure 7: Number of iterations per second for the sequential and parallel implementation.

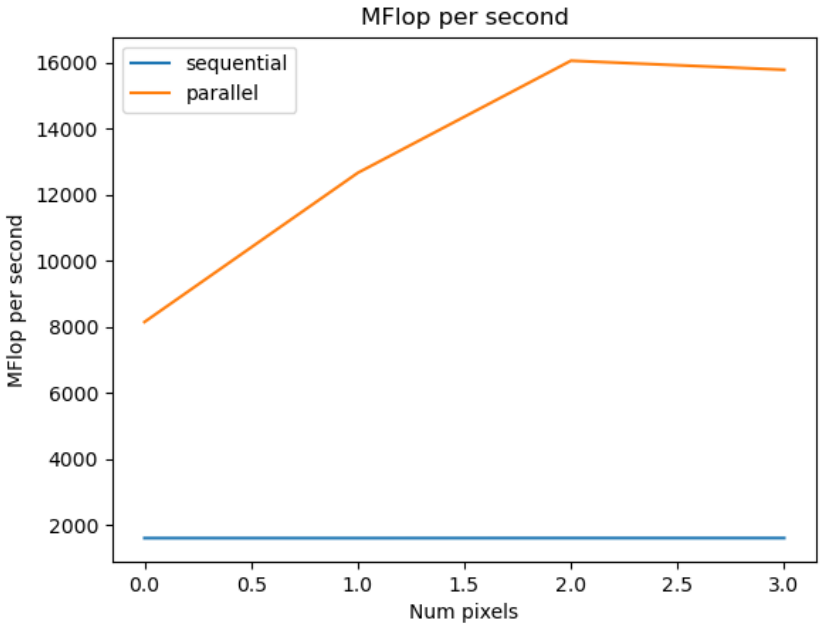

Figure 8: MFlops per second for the sequential and parallel implementation.

3. Bug hunt [15 points]

1. The pragma `for` must be placed directly before the for loop. Since the `schedule` belongs to the `for` pragma it must be moved alongside.

2. If `tid` is not private the last thread to execute `omp_get_thread_num()` will determine the thread id of all threads will leads to odd results at the final print (All threads share the same id). In the for loop the variable `total` will be updated by all threads and at the end each thread prints the result of `total`. The latter leaves the impression that `total` should be

made private as well so each thread has its own copy of `total`. On the contrary, if it was intended to increment the global variable, it would have been sufficient to print it once after all threads are finished. To prevent lost updates and dirty reads a `atomic` pragma was added before incrementing `total`.

3. At a `barrier` pragma all threads will wait until all threads reach the barrier. However since there was a `barrier` in the sections as well, the threads executing the sections got stuck in their `barrier` while the remaining threads waited on the outer `barrier`. This resulted in an infinite waiting scenario. Removing the `barrier` in the sections fixed the issue. The `barrier` in the section was the last logical statement anyway so that thread would have joined up at the common `barrier` with all the other threads.

4. The problem here is that copying the array `a` to all threads exceeds the stacksize of the thread leading to a segmentation fault. With `setenv("OMP_STACKSIZE", "500M", 1)` it is possible to increase the stack size limit for all the threads however it was not possible to increase the memory enough to escape the segmentation fault. Making `a` a global or static variable solved the problem since `a` is no longer stored in the stack. To make them private for every thread `a` was declared as `threadprivate`.

5. The problem here was, that in some occasions, thread 1 acquired lock a and waited for lock b to be released while thread 2 acquired lock b and waited for lock a to be released. None of them released their acquired lock before waiting for the next one which resulted in a deadlock. To solve this problem, thread 1 could release lock a after the first loop is executed, wait for lock b and afterwards a. This order guarantees that all locks get released again, since b will released by thread 2 after it released lock a.

4. Parallel histogram calculation using OpenMP [15 points]

Fig. 9 shows how the parallelism scales over increasing numbers of threads. With a sequential performance of 0.83s, 1-thread performance of 0.85s and best performance of 0.16s for 16 threads.

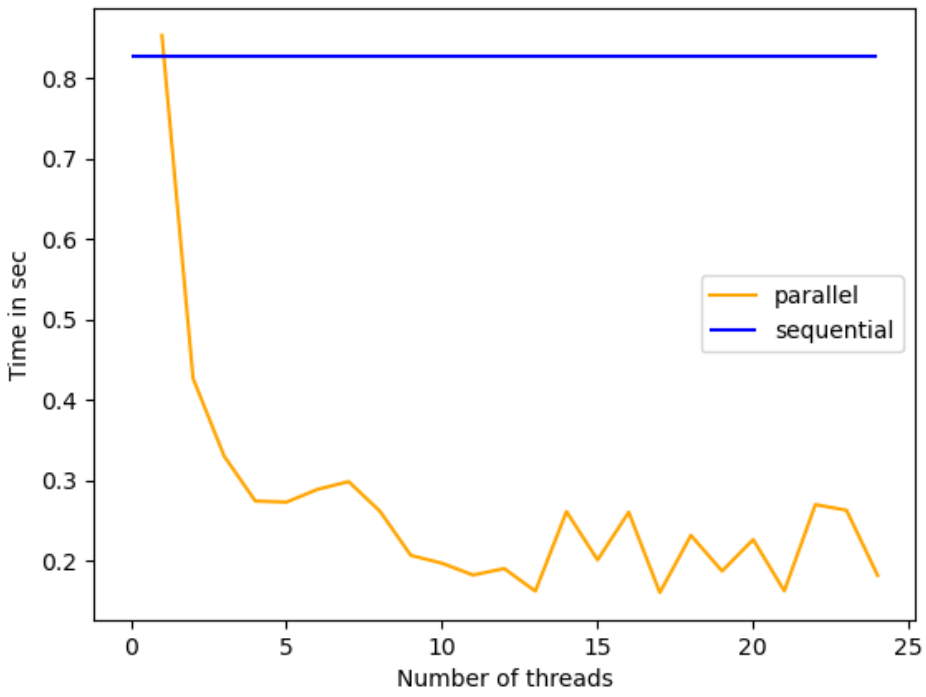

Figure 9: Performance behaviour over increasing numbers of threads for creating a histogram plot.

5. Parallel loop dependencies with OpenMP [15 points]

By dividing N into smaller sized chunks it is possible to compute all powers in those chunks. Since the last chunks will take up most computation time the earlier ones will be a lot faster. Using the schedule dynamic pragma, more threads can be used for the later chunks while fewer ones can compute multiple early blocks. This of cause could also be done without the schedule pragma by assigning threads varying densities of blocks depending on their position. The performance can reach up to 0.45s with 16 threads instead of 6.7 when executed sequentially.

6. Task: Quality of the Report [15 Points]