ETH zürich

**High-Performance Computing Lab for CSE** **2020**

Student: Konrad Handrick Discussed with: -

## Solution for Project 3

Due date: 23.03.2020, 12:00pm

**HPC Lab for CSE 2020 — Submission Instructions**
**(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)**

- Assignments must be submitted to Moodle (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. Implementing linear algebra functions and the stencil operators [40 points]

I implemented the BLAS-L1 routines at first trying to go for the **fast** version using intrinsics (cf. the commented code in "linalg.cpp"). After a little comparison it turned out that -O3 does a good job optimizing instructions, also thanks to the well designed classes and the implemented operators of the "Field" object.

I tried to preserve cache locality and to enforce this using several different pragmas but I did not find the correct ones GCC is willing to accept as guides for compilation (these have been left in and will yield a warning when compiling). Further, no blocking strategy I tried gave better performance overall.

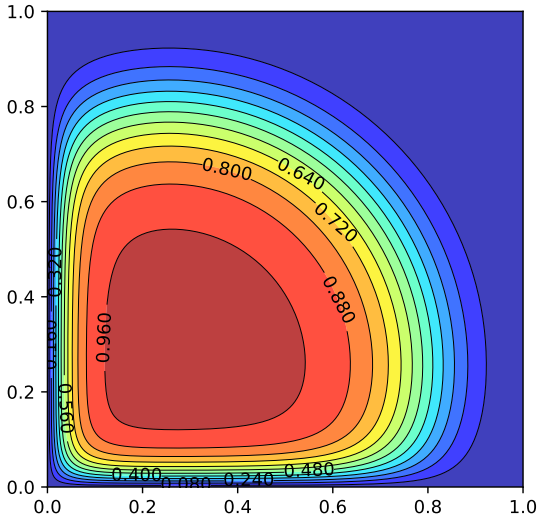


Figure 1: Result of the example parameters on a 1024x1024 grid

## 2. Adding OpenMP to the nonlinear PDE mini-app [60 points]

To parallelize I tried several strategies: pure OMP threads, OMP tasks and finally OMP sections. The last one yielded the best results and will be the goto strategy I'm comparing herein.

The first thing I learned was that for a nested loop without data dependencies **for schedule(static,1)** works almost always faster than **for collapse(2)**. Additionally I discovered that OpenMP has builtin SIMD support (which is not much more than a single keyword but makes for a good optimization hint for the compiler).

The overall parallelization strategy in "operators.cpp" I pursued was built around the idea that at first the inner grid was solved in parallel and then the rest of the workload is distributed among the available threads while also parallelizing the loops the threads have to traverse. Since we have a rather data-oblivious problem at hand I tried building in **nowait**s as often as possible to avoid synchronization overhead.

Parallelizing "linalg.cpp" already yielded more better performaing code by all measures.

### 2.1. Strong scaling

We can observe very visible performance gains in the following plots. It is just puzzling that we need **lots** of cores to have sustainable speedup. For example we can see nice speedup for $N = 1024$ and multiple threads at first - performance does not really consistently keep up adding more cores into the calculation. This could be a hint for a non-optimal parallelization strategy or the banal case: Amdahl's Law prohibiting ideal speedup.

For small problem sizes, the overhead of context switches introduces bottlenecks that make the multithreaded versions a lot slower compared to three or less threads.

As asserted in the Project 3 PDF we have somewhat varying amounts of CGD iterations because of floating-point errors accumulating. Since we move around in a floating-point dominated space we cannot expect bitwise identical results for varying numbers of threads for the same problems. We can only assure a certain tolerance that is in line with machine precision metrics. This is because of the non-deterministic nature of instructions being executed in parallel and the non-associativity of operations on floating-point numbers.

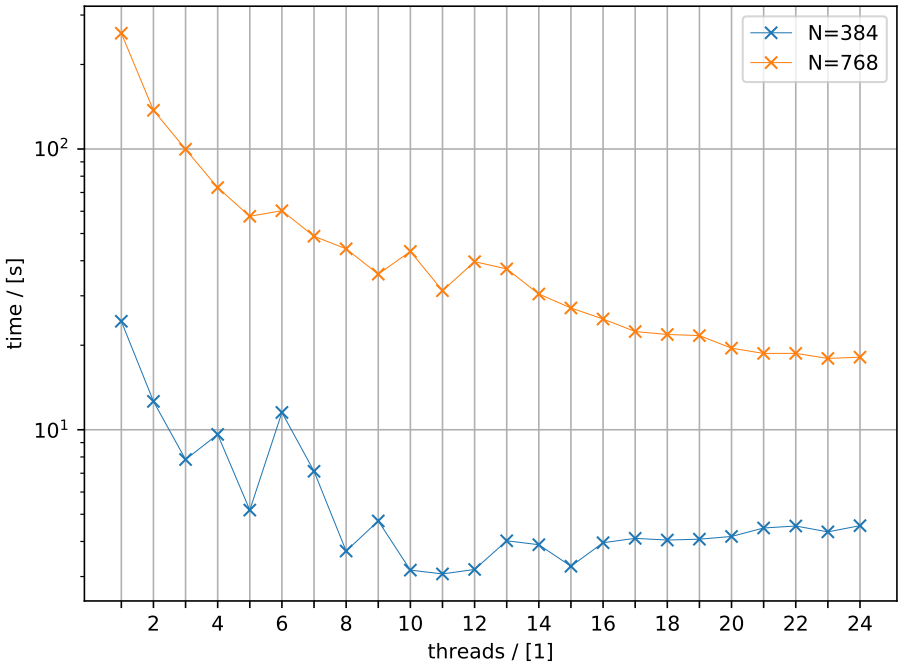


Figure 2: N=384, 768; time in log scale

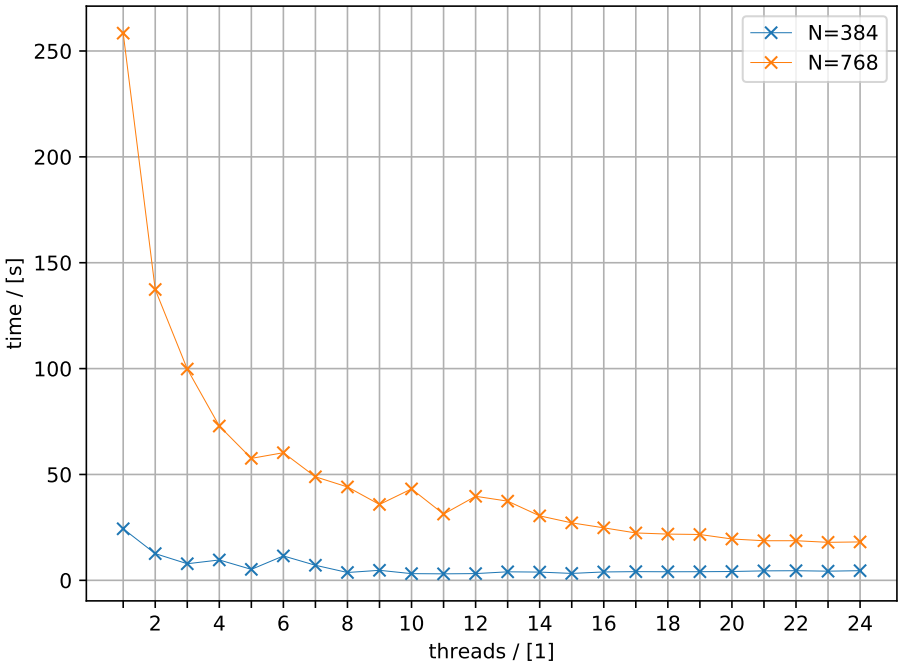


Figure 3: N=384, 768; time in linear scale

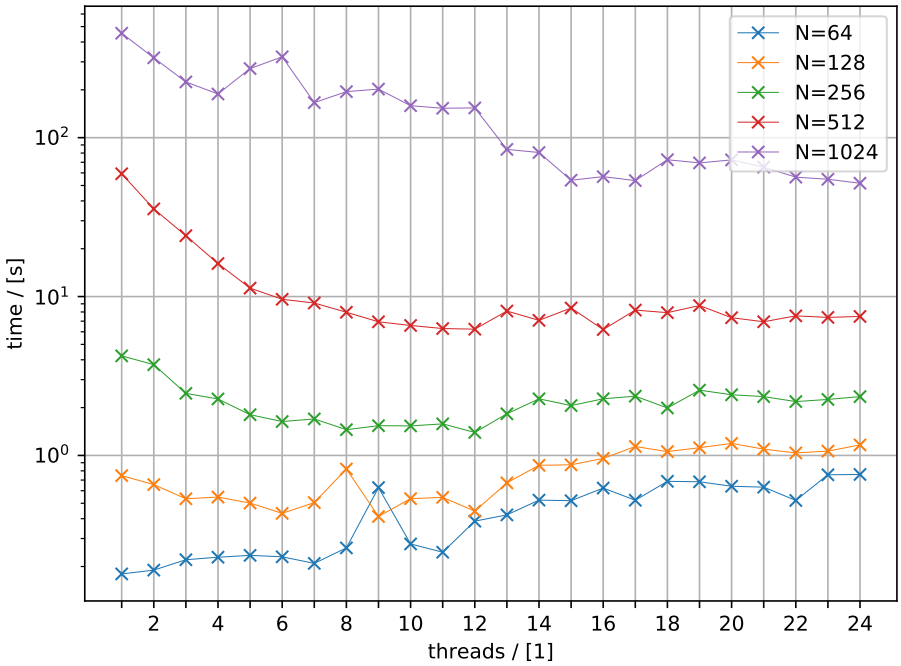


Figure 4: Strong scaling plot for N=64,...,1024 for t=1,...,24 threads; time in log scale

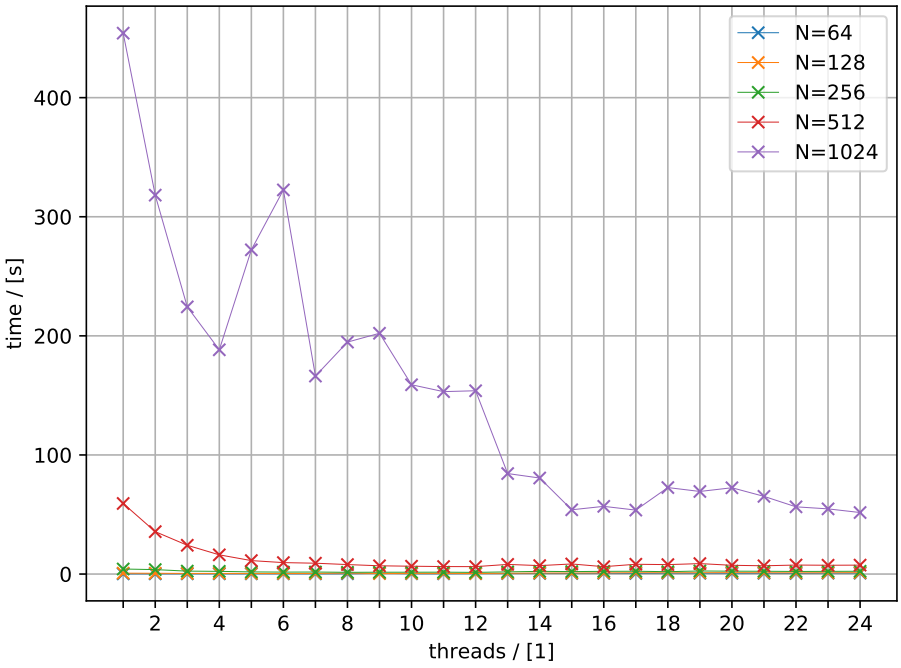


Figure 5: Strong scaling plot for N=64,...,1024 for t=1,...,24 threads; time in linear scale

## 2.2. Weak scaling

For the somewhat confident strong scaling plot to hold up the timing should agree for a **weak scaling** plot. Unfortunately, time per thread with fixed workload per thread ($N = 2^6$ per thread) goes up exponentially. To see if this is just overshooting in the beginning or a hint for bad parallelization I would need to compare with a lot more threads on a much bigger problem sizes which I do not have access to. On the other hand, this could be a natural conclusion of the given problem since we're working on a quadratic grid.

To investigate this issue I ran *perf stat -d* on the weak scaling route locally (on the Euler cluster I do not have access to perf). It turns out that

- around 25% to 30% of all L1d cache hits are misses, always
- for $N = 512$ and 4 threads the last level cache misses (LLC is L2 cache per usual in perf) shoot up to nearly 60% (from less than 1% for 1, 2 threads)
- "abnormally" many context switches do not make for performance degradation in a weak scaling study for this code
- instructions generated by parallelized version of the code aren't bloated compared to the serial version (cycles consumed per core remain in $\mathcal{O}(N^3)$)
- instructions per cycle deteriorate using more cores / larger problem sizes

This hints at non-ideal configurations for the parallelization I attempted. Further measures that should thus be taken could be looking for even better cache locality, better translation of large problem sizes in the TLB (better blocking techniques) or even totally different parallelization strategies. To further investigate I would look at more metrics and analyses from the BPF toolchain but for now perf gave strong insight without there being too steep of a learning curve.

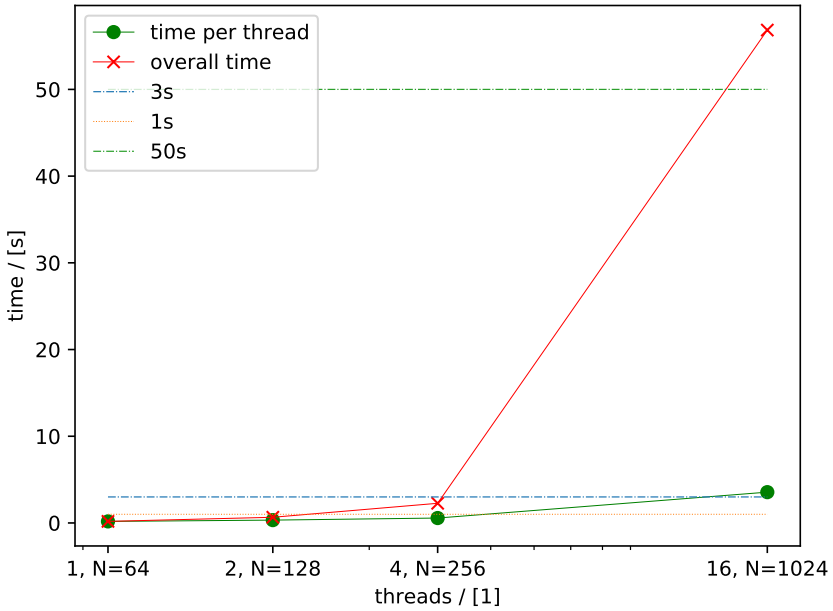


Figure 6: Weak scaling, time to solution; time in linear scale

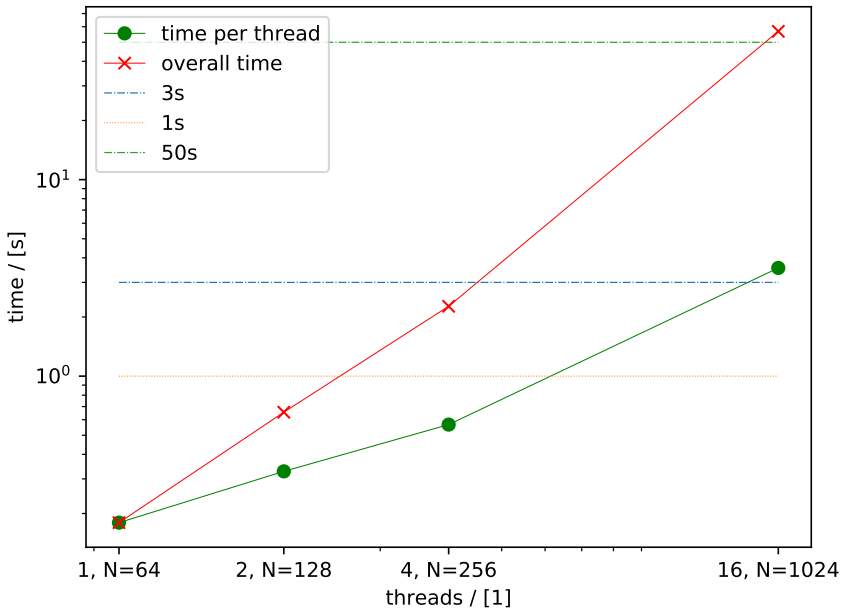


Figure 7: Weak scaling, time to solution; time in log scale