ETH zürich

High-Performance Computing Lab for CSE **2021**

Student: FULL NAME Discussed with: FULL NAME

Solution for Project 2

Due date: 26.03.2021 (midnight)

HPC Lab for CSE 2021 — 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. Parallel reduction operations using OpenMP [10 points]

The `reduction` clause performs a reduction operation (addition, multiplication, ...) on the variables that appear in its list. A private copy for each list variable is created and initialized for each thread. At the end of the loop, the reduction operation is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

The `critical` directive specifies a region of code that must be executed by only one thread at a time. It is clear that the parallel version using the critical section will perform the worst for large array sizes or high number of thread. At each iteration, there will be a race condition between the threads, each of them trying to load and write on the same address but forced to wait in queue because of the critical clause.

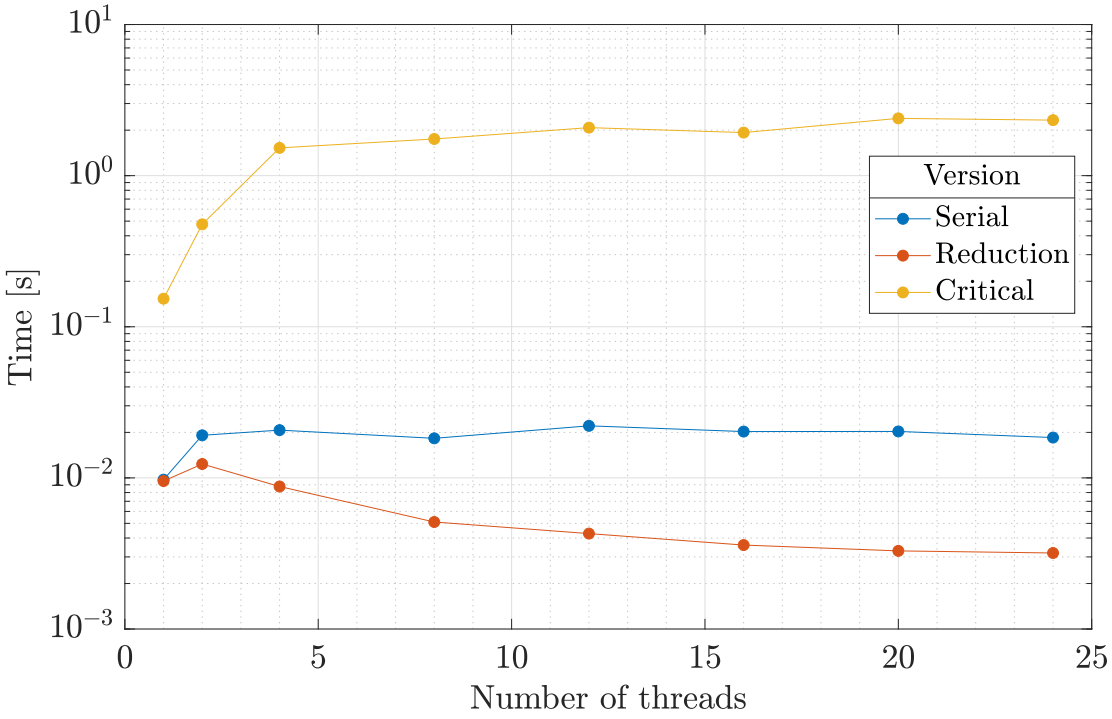

Figure 1: Performance scaling chart for $N = 10^4$.

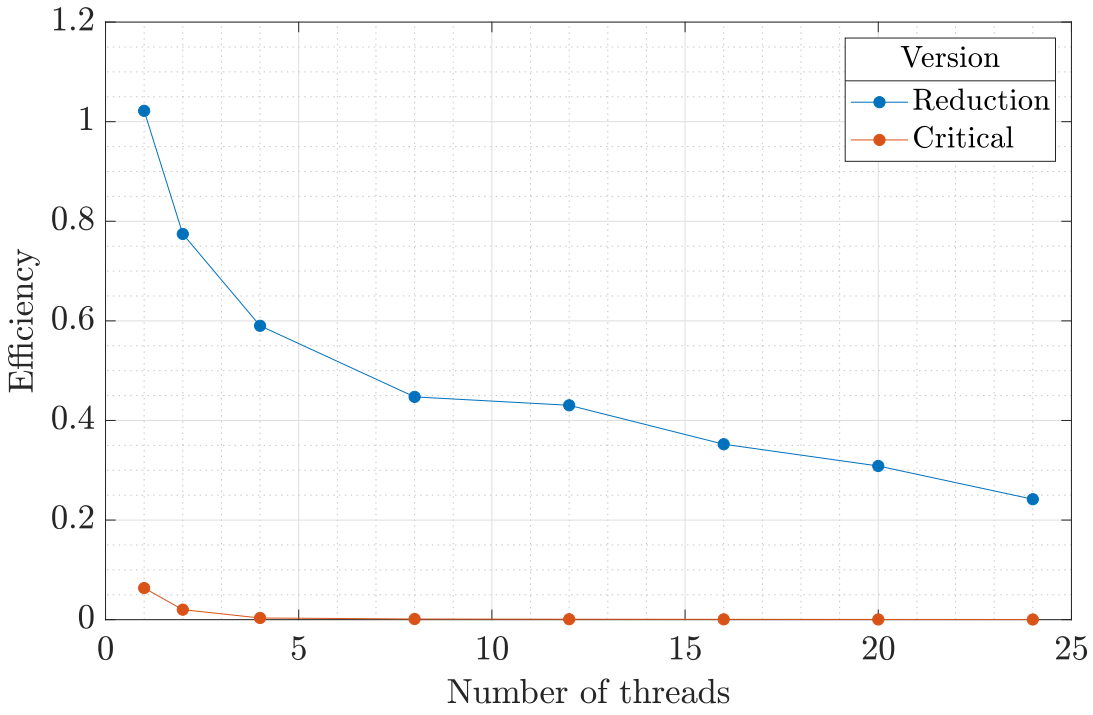

Figure 2: Parallel efficiency for $N = 10^4$.

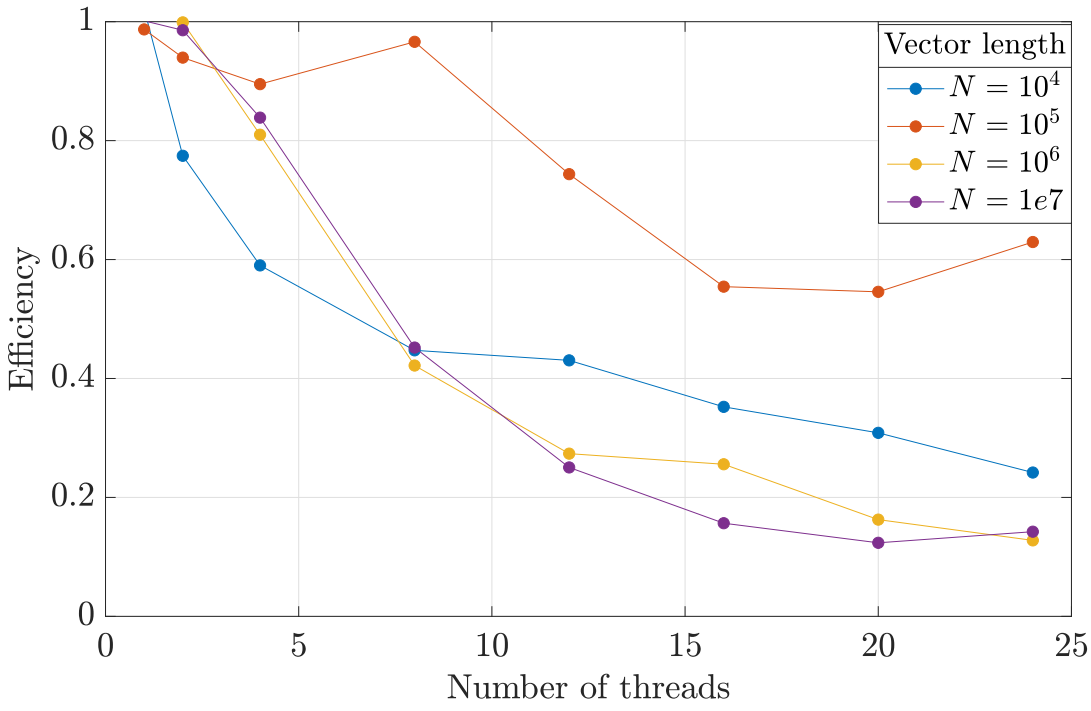

Figure 3: Reduction efficiency

At what size of N would it be beneficial to use a multi-threaded version of the dot product on one compute node of Euler cluster.

2. The Mandelbrot set using OpenMP [30 points]

In this section, the goal is to compute the Mandelbrot set. ++ Intro.

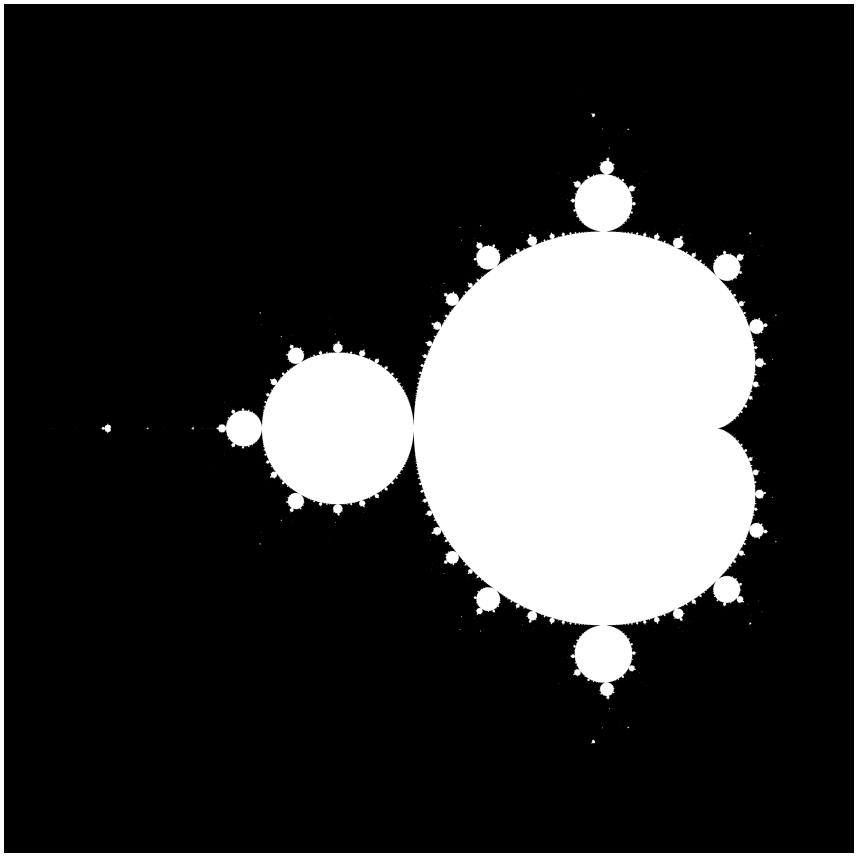

Figure 4: The Mandelbrot Set

	Time [s]	Performance [MFlop/s]
Serial	364.565	2493.34
4-thread	68.920	5292.36
N-thread	...	...

Table 1: Performance table for an image size of 4096×4096 pixels.

	Time [s]	Performance [MFlop/s]
Serial	...	...
4-thread	...	...
N-thread	...	...

Table 2: Performance table for an image size of $x \times x$ pixels.

Run multiple times and add a graphical representation

3. Bug hunt [15 points]

- **Bug 1** : The `#pragma omp parallel for` must be immediately followed by the loop to parallelize, without braces since it is a direct contraction of `#pragma omp parallel` and `#pragma omp for` without braces. Then, the thread number ID private variable `tid` has to be inside the loop, once the iterations have been actually assigned to a specific thread.
- **Bug 2** : Here, the variable `tid` is declared outside the parallel region. Thus, it is shared by default. Consequently, each thread will update the same variable `tid` with its own thread ID

and the prints will be messed up and have no meaningful sense. We could either specify it as private, or declare it inside the parallel region making it private by default.

- **Bug 3** : A `#pragma omp barrier` is placed inside the function `print_results`. Since this function is called inside a section, this creates a deadlock. A section is performed by one single thread and if it encounters a barrier, this thread will wait for the other threads that are not even there and will never be.
- **Bug 4** : Since the array `a` is private, each thread works on its own private copy of the array. In this case, the array is of size $N \times N$ with $N = 1048$ doubles, which exceeds the stacksize. One solution is to increase the stack size with the command `export OMP_STACKSIZE=10000` before executing the code.
- **Bug 5** : This code encounters a deadlock. One thread will begin to execute the first section, locking the array `a` and the other thread will begin to execute the second section, locking the array `b`. After some computation, the first thread will need to access `b`, which is already locked by the other thread. Inversely, the other thread will need to access `a`, which is already locked by the first thread. Both are blocking each other and the program is stopped. One solution would be to place the lock on `a` before the lock on`b` in the second section. In this way, the second thread will have to wait until `a` is unlocked to begin its computation.

4. Parallel histogram calculation using OpenMP [15 points]

The code takes a random sequence with a normal distribution and computes a discrete histogram with 16 bins.

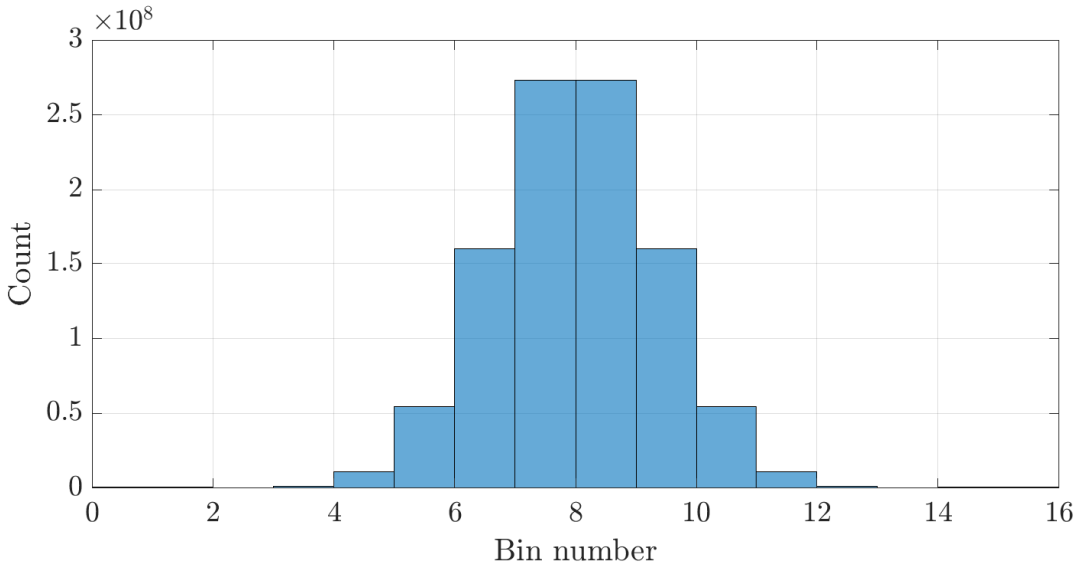

Figure 5: Histogram from a normal distribution with 16 bins.

To avoid race conditions to access the array `dist`, we use a reduction clause `reduction(+:dist)`. Each thread has its own discretized distribution with 16 bins and the final distribution will be the sum of these.

The Tab.3 reports the the runtimes of the serial and parallelized version with different thread numbers. The graphical representation (Fig.6) shows that it actually scales .

	Time [s]
Serial	0.76
1-thread	0.73
2-thread	0.67
4-thread	0.48
8-thread	0.30
12-thread	0.21
16-thread	0.12

Table 3: Performance table for the histogram computation.

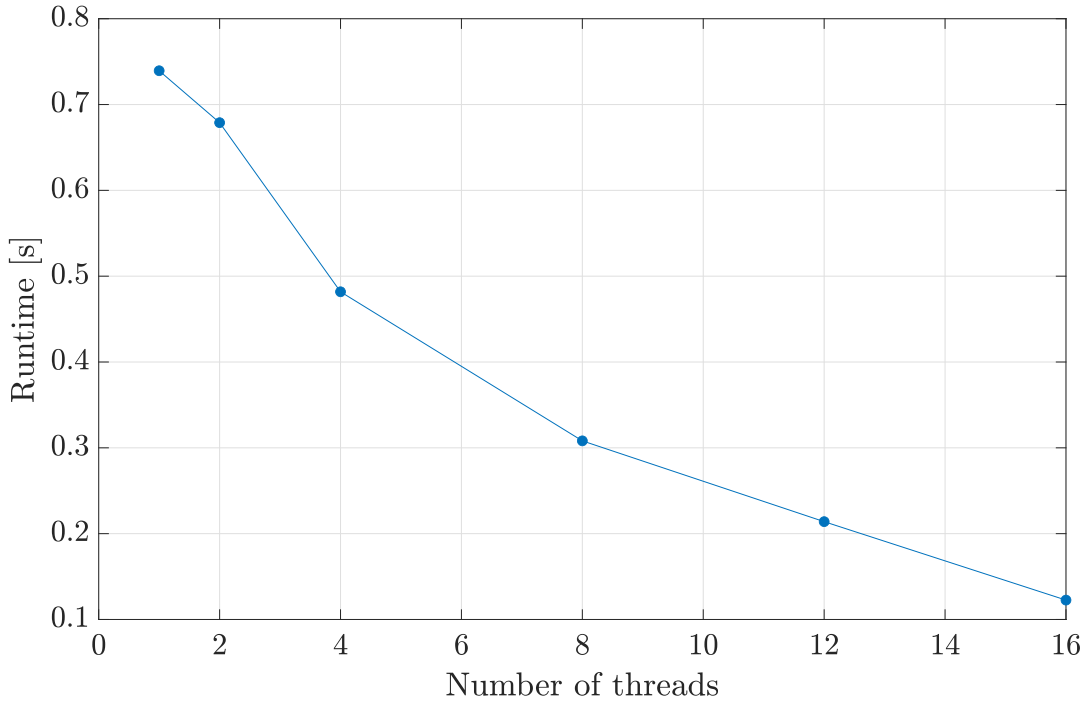

Figure 6: Performance of the parallelized code with different number of threads.

5. Parallel loop dependencies with OpenMP [15 points]

This code shows an example of a loop carry dependency: two or more threads are running parallel, but one needs data from another. The iteration i needs the value of the iteration $i-1$, but iterations of a loop may be executed in parallel.

One solution to this problem would be to replace the iterative calculation `Sn *= up` by a mathematically explicit expression `opt[n] = Sn * pow(up,n)`. However, raising to the power of n is an expensive operation, especially when n becomes very large (up to $N = 2000000000$).

A better solution would be to assign a big chunk of the array to each thread and use the `pow` function only as many times as there are threads, just to initialize the first iteration for each thread. Then the structure of iterative calculation is kept for the whole chunk. In practice, a parallel region is spawned, each with its copy of `Sn` via `firstprivate(Sn)`, which is offset to the right initial value thanks to the `pow` function.

The Tab.4 reports the the runtimes of the serial and parallelized version with different thread numbers. The parallel version of this code only has a slight penalty when running on a single thread compared to the serial version, meaning that the serial performance has been preserved when parallelizing the code. When more threads are used, the parallel version has a good scalability.

	Time [s]
Serial	9.14
1-thread	10.21
2-thread	4.93
4-thread	2.82
8-thread	1.86
12-thread	1.22
16-thread	0.85

Table 4: Performance table runtime

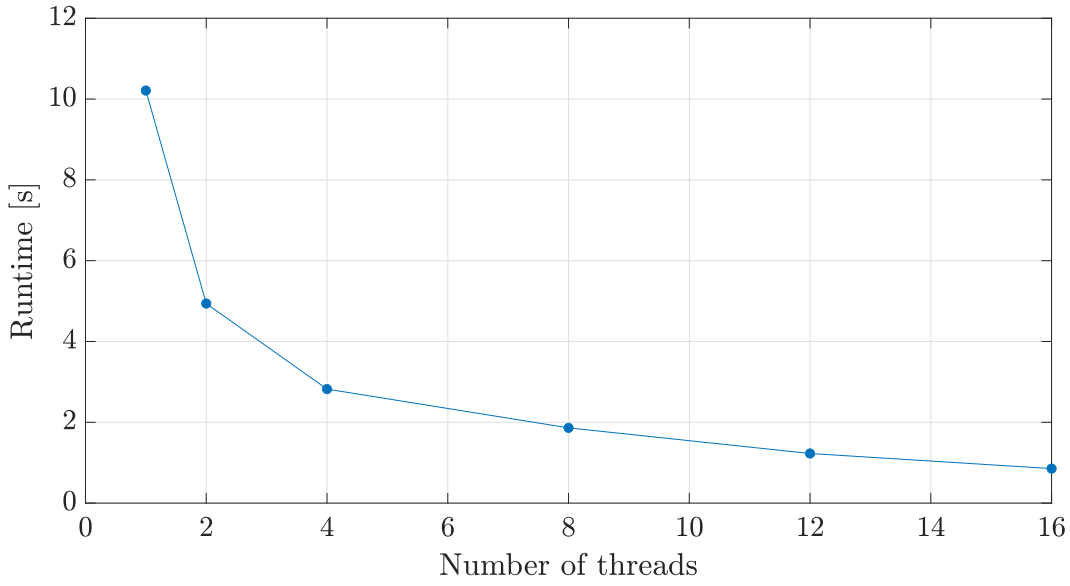

Figure 7: Performance of the parallelized code with different number of threads.

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

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to Moodle.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your OpenMP solutions.
  - your write-up with your name project_number_lastname_firstname.pdf,
- Submit your .tgz through Moodle.