High-Performance Computing
Prof. Dr. Ivan Kisel
Robin Lakos, Akhil Mithran, Oddharak Tyagi

High-Performance Computer Architectures
Practical Course
SoSe 2025

Issued: 04.06.2025 / Due: 11.06.2025

Institute for Computer Science

Exercise Sheet 6

Multi-Threading (OpenMP)

We have already seen the performance boost from using SIMD instructions, which allow performing the same operation on sets of data simultaneously. Ideally, the overhead of distributing data is almost entirely eliminated by interpreting memory cells as a vector of data, where instructions are applied in parallel.

Another approach to parallelization is multi-threading, where a task is divided into subtasks distributed across threads and processed in parallel. Compared to SIMD, one could consider each memory cell in the SIMD vector as a subtask that can be solved in parallel. Multi-threading, however, is not restricted to the same instruction being applied and can process any independent task, albeit with added overhead.

In this exercise, you will work with the OpenMP library, which makes it easy to parallelize specific regions of your code automatically, without needing to define threads for each task manually.

Submission: Submit a PDF containing your answers, explanations, and analysis, along with all source code files, including the full CMake project for your Game of Life implementation.

To compile and run your code, use the OpenMP compiler flag `-fopenmp` to enable multi-threading. A complete compilation command is provided at the top of each source code file.

Exercise 6.1: Multi-Threading Questions

Answer the following questions in your PDF.

i) Assume a for-loop in C++ with two quick, independent iterations. Would it make sense to use multi-threading for this loop? Explain why or why not.

ii) Explain the difference between the concepts of *concurrency* and *parallelism*. With this background, consider a real-world example where you use the maximum number of cores available on your system to parallelize tasks. Even if task distribution overhead is negligible, would your task be perfectly parallelized? Explain why or why not.

iii) What happens in C++ if subtasks that are dependent on each other are processed in parallel by multiple threads? Consider differences between read and write accesses and explain *race conditions* in this context.

iv) Consider the task of calculating the sum of n elements in an array using multi-threading, with n CPU cores available. Describe how to solve the problem in $\mathcal{O}(\log_2 n)$ time without using OpenMP's `reduction` statement or similar. Assume there are no side-effects from the operating system, and all overhead takes $\mathcal{O}(1)$ time. A formal proof is not required.

v) Examine your system. Identify your CPU model, its number of supported threads, and any relevant hardware information. Write these details down and adjust the number of threads for the following tasks accordingly. For example, if your system supports 8 threads, determine a good number of threads to use.

Exercise 6.2: Hello Again, and Again, and Again...

Open `HelloWorld.cpp` and modify the program as described below.

i) Multi-Threading

Apply multi-threading using OpenMP to create 10 threads, each printing "Hello, world!" once. Modify the program to include each thread's ID in the output. Explain the results in your PDF.

Note: The expected output for this part is that threads will stream their output to the console in parallel. This often results in interleaved and jumbled outputs, such as: `"HeHello, llo, world world 97"`. This behavior may vary depending on system activity or other factors. Run the program multiple times to observe variations without recompiling. If it does not work on your machine, just proceed with the exercise.

ii) OMP Critical

Use the OpenMP `critical` directive to fix the issues caused by simultaneous thread output to the console. Compare the results and include your findings in the PDF.

Exercise 6.3: Bug-Fixing

The directory `bugs` contains four source code files, each with OpenMP-specific bugs. Debug these files, fix the issues, and explain the problems and solutions in your PDF.

Exercise 6.4: Calculating π

The source code in `Pi.cpp` includes three functions for calculating π using Riemann sums[1] to approximate the integral:

$$\int_0^1 \frac{4}{1+x^2}\,dx$$

Two of the functions are incomplete and should be implemented by you.

i) OpenMP without Reduction-Statement

Use OpenMP to multi-thread the calculation of π without explicitly using a reduction statement. Store partial results for each thread, and sum them up afterward. Measure the time taken for all relevant operations and compare it to the single-threaded version. Include the expected and actual runtimes in your PDF and explain any differences.

Hint: A container for storing partial results per thread may be helpful.

ii) OpenMP with Reduction-Statement

Rewrite the code to calculate π using OpenMP's `reduction` statement to directly sum the results. Measure the time taken and compare it to the previous results. Document the expected and actual runtimes in your PDF, and explain any discrepancies.

Exercise 6.5: Matrix Computations

In this exercise, extend the `Matrix.cpp` file, which currently contains scalar and SIMD implementations

[1] https://en.wikipedia.org/wiki/Riemann_integral

for matrix computations. Implement a new SIMD+OpenMP version to further optimize the computations and achieve faster performance.

Test your SIMD+OpenMP implementation, measuring its performance. Compare the expected and actual speedups on your machine. Discuss any discrepancies in your PDF, considering factors like hardware limitations or computational overhead.

To reinstall `Vc`, use the script:

```
./install_vc.sh
```

Alternatively, copy the `Vc` directory from the previous exercise and run the `export_paths.sh` script.

Exercise 6.6: Game of Life

Parallelize your Game of Life implementation from the introductory C++ exercise using OpenMP. Identify regions without dependency issues and apply OpenMP to those sections. Compare the runtime of your parallelized version to the original, and document your findings in your PDF.

Hint: Add the OpenMP compiler flag to your `CMakeLists.txt` file.

Heeeeeeelp!

If your bash scripts do not run out of the box (e.g., ”permission denied”), please modify the file permissions to allow execution:

```
chmod +x ./SCRIPTNAME.sh
```

If this does not help, check if you own the file. You can change the ownership to your user by running:

```
chown <your-username>:<your-username> ./SCRIPTNAME.sh
```

If nothing helps, please ask your tutor.