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

High-Performance Computer Architectures
Practical Course
SoSe 2025

Issued: 11.06.2025 / Due: 18.06.2025

Institute for Computer Science

Exercise Sheet 7

Thread Building Blocks (TBB)

Intel's Thread Building Blocks (TBB) is another API that can be used to parallelize the source code in C++. Unlike OpenMP, which primarily operates through preprocessor directives to establish parallel regions, TBB takes a distinct approach by focusing on the construction and management of objects in parallel. This method offers a different perspective and a set of challenges in parallel programming.

Submission: Submit all of your source code files along with a PDF explaining your implementations, answers to questions and analysis of the results.

Exercise 7.0: Installation of TBB

Additionally to your Vc installation script, please use the installation bash script `install_tbb.sh` in your exercise directory. It will create a subdirectory in your home folder to download and build the TBB library. It will also set the paths correctly to make you able to run your application in the current session.

If you leave your session, please use the `export_paths.sh` to export the correct paths again after logging back in. That way, you should be able to run your code without re-installation again.

To run the installation scripts via terminal, use the following command from its directory:

```
./script_name.sh
```

For each exercise, we prepared a Makefile that also makes use of the dynamically linked library of TBB (and others). You can run each code from its directory by using:

```
make run
```

Exercise 7.1: Matrix Calculations

In this exercise, we will compare the runtime of different parallelization methods again and the new player is Thread Building Blocks. Implement the solutions in `Matrix.cpp` to the following tasks and additionally provide in your PDF a runtime analysis anlong with your outputs by using the measured times.

i) Implementation of TBB

Implement the class `ApplyTBB` and overload the `()`-operator. We will use this class as a functor for the application of TBB to our task. Inside, also make use of SIMD instructions using the Vc library. Then, complete the loop in the file's `main`-method by using TBBs `parallel_for` and the functor to measure the time of TBB in comparison to the already implemented versions.

ii) OpenMP

Additionally to the Thread Building Blocks, we want to compare the time with OpenMP. Modify the Vc implementation in the source code and let it make also use of OpenMP. Save this file as `Matrix2.cpp`. Again, provide a runtime analysis and your outputs.

Exercise 7.2: π Again!

From previous exercises, you already know how to calculate π. Create your own functor to apply the TBB in `Pi.cpp` and use it to calculate π.

When creating the class, think about these hints:

- each thread will calculate its own sum, therefore create necessary class member variables
- for the calculation of π, again overload the `()`-operator and make use of `blocked_range`
- to collect partial results, create a method `void join(...)` that takes a reference to another TBB instance of your class and adds the sum of the other object to its own.
- in the `main`-method, make use of the `parallel_reduce`-method of TBB that works similar to the reduction statement in OpenMP.

Exercise 7.3: Counting Zeros

In this exercise, we want to count the number of occurrences of zeros after throwing random numbers into a mathematical function (see `Count.cpp`).

Here, we will make use of TBB again, but in two different ways. First, we use an atomic counter to count the number of occurrences, in a second approach, we will use a mutex to increment a normal integer. You can use the scalar version as an orientation what exactly is expected. Again, make use of TBB's `blocked_range` to create the loops inside your TBB classes.

For both exercises, include your outputs as well as a runtime anaylsis in your PDF, along with your answers to the questions.

i) Atomic Counter

Create a class `ApplyTBBA` that makes use of an atomic counter `std::atomic<int>` to count of zeros the occurrences of the mathematical function, given the array `a`. Do not create a new array, make use of the existing one.

Please explain why an atomic counter is necessary here and explain its behavior.

ii) Mutex

Create another class `ApplyTBBM` that makes use of a mutex `spin_mutex` to count of zeros the occurrences of the mathematical function, given the array `a`. Again, do not create a new array, make use of the existing one. Within your iterations, use `spin_mutex::scoped_lock` to lock the mutex before incrementing your `int counterParM`.

In which scope do you have to initialize your mutex and why? Additionally, why is the mutex locked but never unlocked explicity?