

Parallel Computing

Exercise 2 (May 5, 2026)

Alois Zoitl
alois.zoitl@jku.at

The result is to be submitted by the deadline stated above via the moodle interface. The submitted result is as a .zip or .tgz file which contains:

- a single PDF (.pdf) file with
 - a cover page with the title of the course, your name(s), Matrikelnummer(s), and email-address(es),
 - answers to the questions in the 7 tasks provided below,
 - output generated by your programs,
- the actual source code of your solutions.

If you work in a team the quality of the solution is expected to be higher and grading will take this into account.

Low-Level Shared-Memory Parallel Programming

The purpose of this assignment is to help you understand some of the most important issues in low-level shared-memory parallel programming.

The example simulates partitioning jobs and counting these jobs as operations. The provided version has several flaws which you are to supposed to fix. There is a corresponding chapter in the *PerfBook*¹ and it also relates to the data race examples discussed in the lecture.

Set-up and Preparations

First, please try to get hold of a desktop, laptop or small server with Linux, Unix, or Ubuntu and a modern 'gcc' installed (Ubuntu on the Linux Subsystem for Windows is also possible).

Then unpack the provided zip file and run

```
./configure.sh && make && ./bogus 1 9 # optimized code
```

which should compile the existing 'bogus' version of this exercise, run it, test that it works and print timing information. Then try:

```
./configure.sh -g && make && ./bogus 1 9 # non-optimized code
```

Tasks

The assignment has 7 tasks (percentage achievable points in brackets):

1. Set-Up [10%]

Describe your system including the processor, the number of cores, the operating system version and compiler version.

2. Analysis of 'bogus' [10%]

- What running times do you get for both version?

```
./configure.sh && make && ./bogus 1 9
./configure.sh -g && make && ./bogus 1 9
```

Now try the same with two threads:

```
./configure.sh && make && ./bogus 2 9
./configure.sh -g && make && ./bogus 2 9
```

¹ "Is Parallel Programming Hard, And, If So, What Can You Do About It?", Paul E. McKenney, 2022

- Do the times decrease/increase?
- Do you get correct or incorrect counting?
- What are the three data races in this 'bogus' program?
- Apply one of the data race detection tools and compare its output with your assessment

3 Fix two of the three data-races by using volatile [10%]

Two of the data races can be fixed by telling the compiler not to assume anything about accessing memory and in particular not optimizing away the access (reading and writing a variable or pointer). Use the `volatile` keyword to achieve this or an `ACCESS` macro (as in the *PerfBook*).

```
#define ACCESS(P) (* (volatile typeof (P) *) &(P))
```

The resulting program should be called `incorrect.c` (since it still has a data race). Just place it in the same directory. The makefile should automatically compile it too. Compare running time with those from '2' with:

```
./configure.sh && make && ./incorrect 2 9
./configure.sh && make && ./incorrect 1 9
```

- Do you get speed-up?

4 Fix the main data race through locking [10%]

Use `pthread_mutex_t` as in the demo for the data-race and call the resulting program `locked.c` and run the tests again. This should give the correct result for two threads.

- How much slower are these new versions?

Run it also for 3 and 4 threads to get a better picture about speed-up.

5 Instead of locking use atomic increment [10%]

Call this program `atomic.c`. You probably want to use the now deprecated `__syncXXX` GCC atomic functions instead of C11 atomics (or even C++11 code).

- What are the running times (again for 1,2,3,4 worker threads)?

6 Progress printing [30%]

Start an additional thread which sleeps for 1/100 of a second and then prints the total accumulated counts in percent (use for instance `usleep` for sleeping). Use carriage return '`\r`' instead of '`\n`' to print the percentage always at the start of the same line (or terminal codes). Then add a `fflush` call. Devise a method to stop this thread after all workers have finished their operations. You can of course also do a real progress bar if you fancy that. This program is called `progress.c`. You can base this part on any of the versions before (`bogus.c`, `incorrect.c`, `locked.c`, `atomic.c`).

- What is the overhead of printing?

7 Thread local counting [20%]

Move to a thread local counting scheme `local.c` (start with `incorrect.c`), where each worker thread has its own result, which after a worker has finished is added to the global result. Adapt the progress printing thread to peek into local thread counters and compute a global count result. There are many ways to do this. Again see the corresponding Chapter 5 in the *PerfBook*. You can try all but one is enough (the simplest statistical inexact counting version). A simple solution is to add local counters to the 'worker' structure. But then be careful with 'false sharing', where different threads write to the same cache line. Accordingly experiment with padding (to put those counters into different cache lines).