Ghent University

Parallel and Distributed Software Systems

# Homework Assignment 4

*Tibo Vanheule*


Academic Year 2023-2024

# 1 Correctness of multi-threaded code

### (Q1.1): pinpoint the first error + explain why the code is incorrect

On line 45, adding a number to a certain variable in memory is not thread-safe as it involves multiple steps. Each of these steps can also use pipelining and hence, the entire operation will require multiple clock instructions. If other threads write to this memory location while another thread is computing an updated value, this will result in data corruption (incorrect results). The ultimate outcome will depend on the precise order in which threads are writing to this memory location. This is called a data race.

Solved using the second method, a mutex together with a local variable.

Additional changes were made to solve the memory leaks reported in Valgrind.

### (Q1.2): pinpoint the second error + explain why the code is incorrect

threadID is passed as reference, this causes incorrect behaviour as the variable could be updated when the thread evaluates it. Solved by passing the lvalue instead of an rvalue.

# 2 Influence of memory access patterns on software performance

### (Q2.1) Report the runtimes and speedup.

| **# Threads** | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| **Runtime** | 0.434 | 0.830 | 0.940 | 0.900 | 0.444 |
| **Speedup** | 1 | 0.523 | 0.461 | 0.481 | 0.975 |

Table 1: Results without optimisation on an HPC node. Speedup is relative to thread one.

### (Q2.2) Report the runtimes and speedup.

| **# Threads** | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| **Runtime** | 0.101 | 0.124 | 0.110 | 0.103 | 0.056 |
| **Speedup** | 1 | 0.814 | 0.918 | 1,01 | 1.804 |

Table 2: Results with optimisation on a hpc node. Speedup is relative to thread one.

## 2.1 (Q2.3) Explain different behaviour in obtained speedup.

Speedups in both tables show that multiple threads are not better for a small number of threads. In fact, the running times get worse and only when using 16 threads they get better. This comes due to the for loop of a thread where the numbers are summed, there we read and write back the value of localSum. When the thread loads the value in its cache, it gets invalidated by another thread. This forces memory access and slow running times.

solutions:

1. Use a local variable and write to LocalSums after the summation.
2. Make LocalSums larger and let each thread write to index ThreadID*64. This results in each thread writing to indexes which are bigger than the size of a cache line.

The second solution is the fastest but consumes more memory.

Additional changes were made to solve the memory leaks reported in Valgrind.