HOCH SCHULE OFFEN BURG

Instruction sheet for lab class 5

Note that you need to download the matrix class (in file matrix.h) and the simplistic test framework (in file test.h) and put them in your source directory.

Exercise 1

Download the file raceCondition.cpp from Moodle.

1. Read the code carefully and understand what the code does (or is supposed to do).
2. Compile the file and run it. Does the result meet your expectations?

Before reading on, think about these questions and discuss them in your group.

The first problem results in money getting lost or being created in the transferMoney function, i.e. the sum of the balances in the two accounts may change. Fix this problem without using mutexes or locks! Describe your solution.

>>> Made balance at Account struct atomic.
added .load in if condition

The second problem may lead to negative balances. Fix this problem as well and describe your solution.

>>> wrapped access of "from" and "to" balances at transferMoney by mutex lock and unlock

Exercise 2

The following program is supposed to output "foobar" 100 times on the command line, with thread 1 always writing "foo" and thread 2 always writing "bar". But that doesn't work, the output doesn't look as desired.

Modify the program so that the output is always "foobar". You can't change the distribution of the output between the threads, so you need to make sure that the threads are synchronized appropriately.

```
#include <iostream>
#include <thread>

const int n = 100;

void foo() {
  for (int i = 0; i < n; ++i) {
    std::cout << "foo";
  }
}

void bar() {
  for (int i = 0; i < n; ++i) {
    std::cout << "bar\n";
  }
}

int main() {
  std::thread t1(foo);
  std::thread t2(bar);

  t1.join();
  t2.join();
}
```

Upload your program to Moodle.

Exercise 3

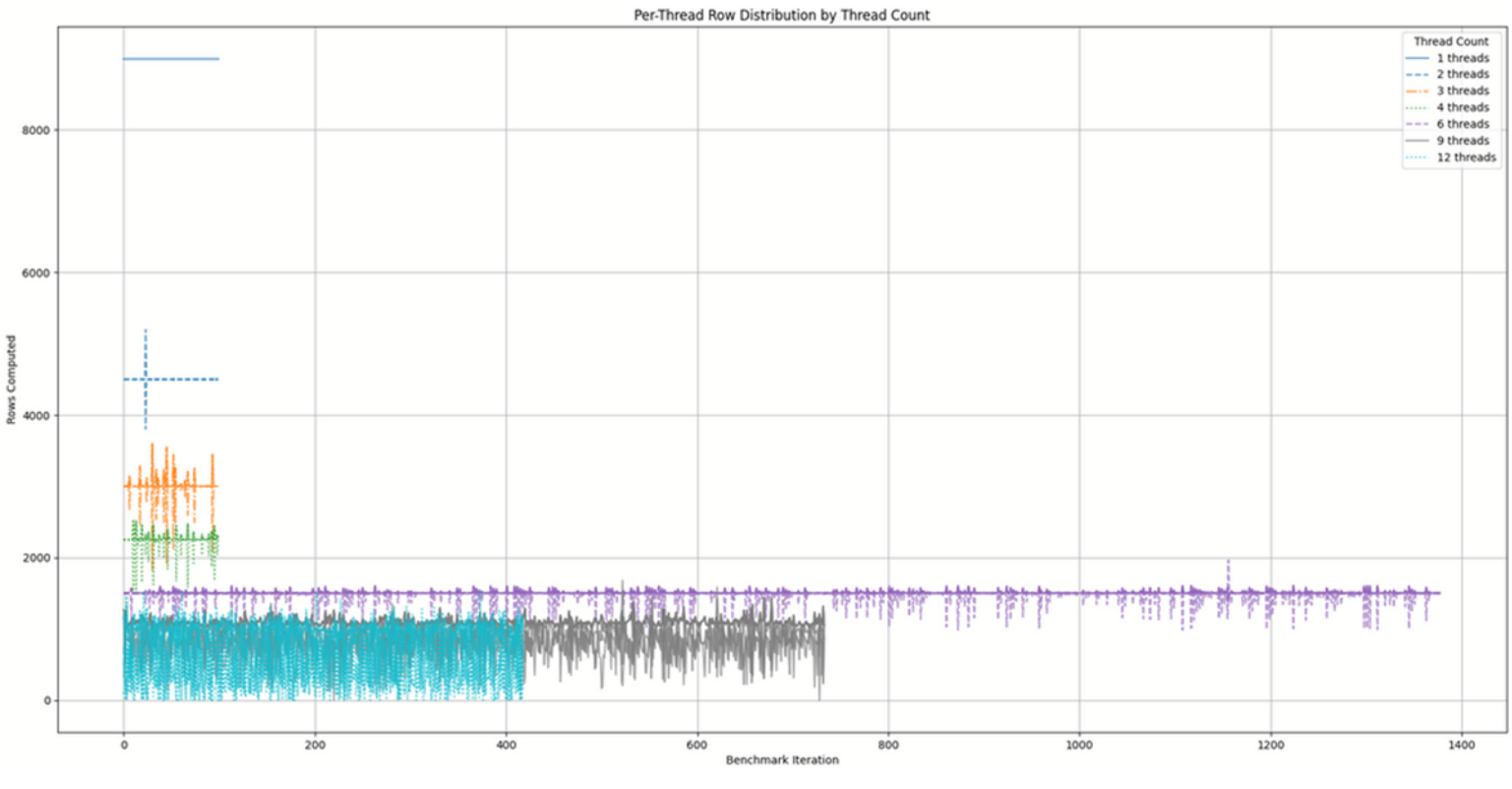
Download the file matrixVectorProduct.cpp from Moodle.
You shall implement the product of a matrix with a vector, using multithreading.

A simple and obvious solution is to distribute the indices to be computed by each thread a priori (similar to the computation of pi in the lecture).
Implement this parallelization and perform a benchmark for n=1,2,3,4,6,9,12 threads on the cluster.

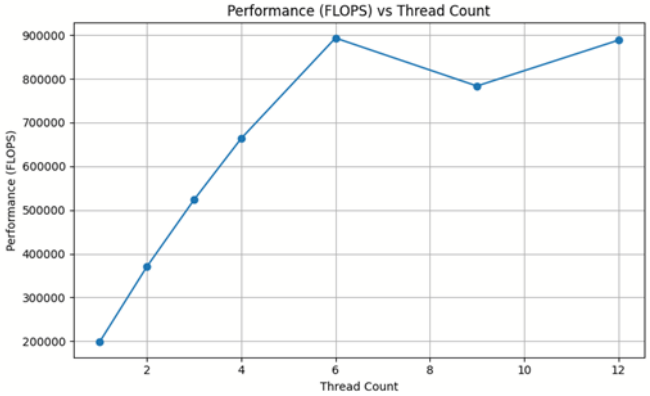
$i \cdot n + j$

0 1 2 −1 −2 −3 2 3 4

i	j		
0 1 2		1	3
−1 −2 −3		0,5	−3
2 3 4		$0,3\overline{3}$	3



Create a plot of thread number vs. performance and insert it here.



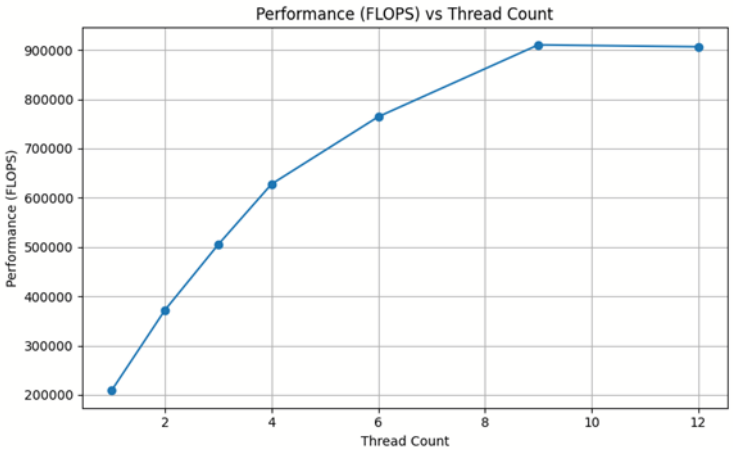
The parallelization strategy might be inefficient if some products are computed faster than others. Another strategy would be to create a separate thread for each row of the matrix. However, this is also inefficient, as creating thousands of threads creates a very large overhead.

Implement a dynamic strategy, where a given number of threads solves the problem in such a way that after computing one line, a thread calculates the next row that has not yet been processed yet. counter which is atomic, thread increments and instant starts calculation of line

Record which thread calculated how many rows. Does this number vary or is it roughly the same for each thread? (Look above for graph)

It depends on the thread count, the higher the thread count, the more likely it is for threads to get no work at all.

Again, benchmark your program and create a plot of thread count vs. performance.



Which version is faster, the first one or this one? Does it depend on the number of threads?

The first version is faster than the second for 6 threads but second performs better at 9 and 12 threads. The second also is slightly better for 1 thread but else worse.

Exercise 4: Thread safe functions

Look up, what a “thread safe function” is. Give a brief explanation in your own words.

A thread safe function is a function which can be used by multiple threads, where no unexpected behavior occurs. Shared state is either avoided or protected by mechanisms like atomic, mutex, semaphor etc. Other accessing threads might get blocked.

Now look up “reentrant safe function”. What is the difference to thread safety?

Reentrant functions are functions which only use local variables and don't access any global or static stuff, they don't call non-reentrant functions and have no side effects. They can be safely interrupted and reentered, even when used by other threads in the meantime.

All reentrant functions are thread safe but not all thread safe functions are reentrant.