Homework Assignment 2

General information

Ugent HPC infrastructure

All source files required for this assignment can be found in the documents section on Minerva. All examples should be compiled and run on the 'Delcatty' cluster (part of the UGent HPC). You will need an account for this cluster, which can be requested by going to the following URL and following the instructions:

http://hpc.ugent.be -> "Toegang en beleid" (NL) / "Access policy"

Note that it might take one or two days before your request is approved.

After your request is approved, read the section on "Information for new users" on the wiki page. This explains how to connect to the HPC infrastructure and how to copy files between your system and the UGent HPC. A VPN connection to UGent is required when attempting to connect from outside the UGent network.

Once you connect to the HPC infrastructure, you will be logged onto one of the interface nodes. These nodes can be used to compile software and submit jobs to the different clusters (default = Delcatty), but running software on these interface nodes is generally considered bad practice. Therefore, we ask that you compile and run the benchmark examples below on one of the worker nodes. In order to get access to one of the worker nodes type:

```
qsub -I
```

If at least one core on a worker node is available, this command will open a new shell ('interactive session') on one of the worker nodes (nodeXXXX). In order to get access to a complete machine (16 cores), issue:

```
qsub -I -l nodes=1:ppn=16
```

This will give you exclusive access to one of the nodes which has the advantage that your timings will not be influenced by software from other users. Depending on the occupation of the cluster by other users, it might take some time before the scheduler is able to grant access. You can check the cluster usage this by issuing:

```
pbsmon
```

When you are finished, terminate your session by typing:

```
exit
```

Report

This assignment should be completed **individually**. While it is OK to discuss the solution with fellow students, it is strictly prohibited to exchange and/or copy solutions.

You should hand in your report in **.pdf format**. Your report should contain **(a) your name** and **(b) the answers to the questions using the format listed below**, i.e.

Q1.1: answer to this question

Q2.1: answer to this question

…

Additionally, hand in the modified **matmat.cpp** file. **Write your name in this source file as well (in comment).**

The maximum size for the report is **one A4 page, 10pt**: "In der Beschränkung zeigt sich erst der Meister".

Your report should be named "assignment1_firstname_lastname.pdf", the modified matmat.cpp filename should be renamed to "matmat_firstname_lastname.cpp". Use the Minerva dropbox to submit your solutions. Do not zip or otherwise pack these two files!

The deadline is Wednesday, April 23rd 12:00 (noon), CET. This deadline is firm and non-negotiable.

1 Correctness of multithreaded code

Download the c++ source file `pimontecarlo.cpp` from Minerva. Study the source code. Compile and run the code using the GCC compiler:

```
module load GCC
g++ -O3 pimontecarlo.cpp -o pimontecarlo -lpthread -std=c++11
./pimontecarlo <number of threads>
```

This program can be likely executed without crashing, but is incorrect. Certain race conditions can be detected using the valgrind software, e.g.

```
valgrind --tool=helgrind ./pimontecarlo 2
```

By studying both the output of valgrind and/or the program code itself, you should be able to correct this program. (hint: there are 2 distinct errors in the code, one obvious and one subtle; valgrind detects both of them).

- **(Q1.1): pinpoint the first error + explain why the code is incorrect**
- **(Q2.1): pinpoint the second error + explain why the code is incorrect**
- **Modify the source code and hand in the modified source.**

2 Multithreaded performance

Download the c++ source file `threadsum.cpp` from Minerva. Study the source code. Compile and run the code using the GCC compiler as follows:

```
g++ -O0 threadsum.cpp -o threadsum -lpthread

./threadsum <number of threads>
```

This code sums the elements in an array in a multi-threaded fashion, without the use of mutex. The code is functionally correct.

- Run the code on Delcatty using 1, 2, 4, 8 and 16 threads. (**Q2.1) Report the runtimes and speedup.**
- Now recompile the source using full optimizations:

  ```
  g++ -O3 threadsum.cpp -o threadsum -lpthread
  ```

  Again run the code on Delcatty using 1, 2, 4, 8 and 16 threads. **(Q2.2) Report the runtimes and speedup.**
- **(Q2.3) Explain different behavior in obtained speedup.**
- **Modify the source code such that any speedup problems are resolved, regardless of the optimization chosen. Hand in the modified source.**

3 Influence of memory access patterns on software performance

Download the c++ source file `prodcons.cpp` from Minerva. This code contains skeleton code for the producer-consumer example that we've seen during the lectures. No synchronization is present in the code. During the lectures, we've explained how we can use semaphores in order to signal a producer/consumer thread when the container (an stl vector in this case) is empty/full.

- **Implement the producer/consumer pattern as we've seen during the lectures (see slides), but using (a) condition variable(s) instead of semaphores. Hand in the modified code.**