Low-Level Parallel Programming
Report for Assignment 1 by Team 13

Gideon LANDEMAN, Noa LERCH and Leon HEJDENBERG PHILIP

25th January 2022

All experiments are run on the UPPMAX server with the configuration given by the following command line argument, `interactive -A uppmax2021-2-28 -M -snowy -p core -n 1 -c 4 -t 1:00:01 --gres=gpu:1`. Note that we use 4 cores.

Part 1

Results

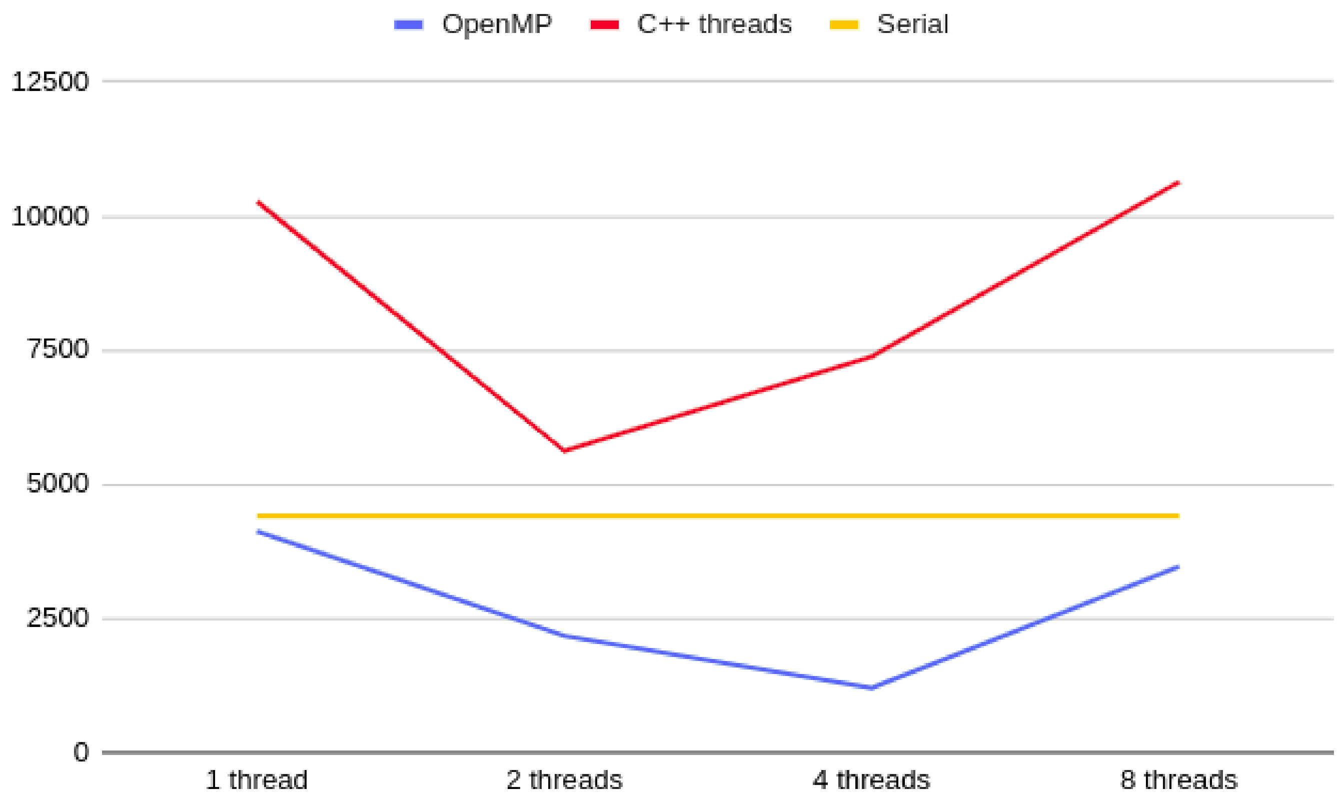

Figure 1: The figure represents the reference times obtained when runnning the `demo/demo scenario.xml` from the 1DL550_Assignment directory with the `--timing-mode` flag.

Part 2
Questions

A What kind of parallelism is exposed in the identified method?

The parallelism identified in the serial version of `tick()` is found in the loop which iterates over each agent in order to determine its next position.

B List at least two alternatives for the OpenMP, and two alternatives for the C++ Threads implementation that you considered.

B.1 OpenMP

One alternative for the OpenMP implentaion would be to use pragma omp parallel in conjunction with `#pragma omp for` in order to provide parallelism. However, another option would be to use the pragma omp parallel for compiler directive.

B.2 C++ Threads

Our initial implementation was to hard-code two threads that would take an approximately equal workload. This method was easy to implement but very lacking in flexibility; there was no easy way to change the number of threads in use. This would have to be done by copying previous code, which is not a particularly sustainable solution.

Another method is to create a pool of threads to which we distribute work throughout execution. This method may provide performance benefits when used in a context where we want to reuse threads across execution times, since the creating a thread incurs a performance cost. A downside of this method is that it is more memory intensive as it keeps threads in memory even when they are not used. We opted not to use the pool method because our use case is very simple and implementing this would take longer and is bit excessive.

C Once for OpenMP and once for C++ Threads, explain your chosen implementation.

C.1 OpenMP

C.1.1 How is the workload distributed across the threads?

Each of the threads are given as equal a number of agents as possible from `std::vector<Ped::Tagent*> agents` (which represents the pedestrians in the simulation).

C.1.2 Which number of threads give you the best results? Why?

Four threads give the best performance. Execution with eight threads takes almost three times as long as with four threads and more than 50% longer than with two threads.

In our implementation and with our data, it appears four threads is faster than two because the greater parallelism allows for more operations to execute in the same amount of time. The reason why running four threads is faster than eight threads in our situation is most likely because it takes time to create threads. At some point the cost of creating threads becomes greater than the performance increase, which appears to have occurred here.

C.1.3 What are the possible drawbacks of this version?

Drawbacks of using OpenMP in comparison to for instance C++ Threads could be that the potential for fine-tuning might be lower since it is not possible to assign load to individual threads in the same manner.

C.1.4 Why did you choose this version over the other alternatives?

All we wanted to parallelize was the for-loop, hence `#pragma omp parallell for` sufficed for what we wanted to achieve. We could have used a `#pragma omp parallell` block with a parallelized for-loop within, but since no operations are performed outside of the for-loop, this was not necessary.

C.2 C++ Threads

C.2.1 How is the workload distributed across the threads?

Each of the threads are given as equal a number of agents as possible from `std::vector<Ped::Tagent*> agents` (which represents the pedestrians in the simulation).

C.2.2 Which number of threads give you the best results? Why?

For the chosen implementation utilising only two threads gave the best performance. This was not expected as 4 cores were available. However, this is likely due to the amount of work to be performed being small in relation to the time and resources required to create additional threads.

C.2.3 What are the possible drawbacks of this version?

Possible drawbacks of this version include it being less simplistic compared to the OpenMP version. Also, it is currently necessary to recompile the program in order to change the number of threads in use, whereas the OpenMP version supports environmental variable declarations, which can be declared on the command line.

C.2.4 Why did you choose this version over the other alternatives?

We followed examples both from the lectures, as well as the C++ documentation. We concluded that dividing the agents into contiguous blocks for each thread, and creating each thread in a for-loop would simplify modifying the number of threads.

D Which version (OpenMP, C++ Threads) gives you better results? Why?

OpenMP gives us the highest performance. Interestingly, our serial implementation seems to run faster than C++ threads.

E What can you improve such that the worse performing version could catch up with the faster version?

Something which could be done to improve the C++ Threads version would be to explore the implementation that uses a pool of threads from which capacity is drawn as required. Another option could be to try distributing work to the threads in another type of chunk. For instance giving thread1 ”agent0, agent5, agent10, agent5” rather than ”agent0, agent1, agent2, agent3”. However, this is unlikely to have a large impact since the agents are assumed to be independent from one another.

F Consider a scenario with 7 agents. Using a CPU with 4 cores, how would your two versions distribute the work across threads?

7 is a prime number and cannot be evenly divided across any integer number of cores more than 1. Our C++ threads implementation uses integer division to determine the amound of ”work” to be done by each thread.

G For your OpenMP solution, what tools do you have to control the workload distribution?

For the OpenMP solution, a couple of tools are available to adjust the workload distribution. One of them is the option to assign environment variables in order to choose the number of threads used, `OMP_NUM_THREADS=`. It is also possible to adjust the loop scheduling which is used to break down the for-loop iterations using `OMP_SCHEDULE=[static|dynamic|guided|auto]`. Lastly, there are also options for stating the stack size of threads and for mapping threads to certain cores.

Part 3
How to choose between serial, C++ Threads and OpenMP

In order to choose between serial, C++ Threads and OpenMP it’s important to take into consideration what application is being built and what hardware it is going to run on. If only one core is available, it does not make sense to use C++ Threads or OpenMP since the threads would just be queued. On the other hand, if multiple cores are available, C++ Threads and OpenMP differ in the granularity available. C++ Threads could have the potential to be more performant than OpenMP but the implementation complexity is a higher.

Part 4
Team contributions

All team members contributed equally.