Low-Level Parallel Programming

Assignment 1
Deadline: January 24, 23:59

Uppsala University, Spring 2024

1 Deadlines, Demonstrations, and Submissions

Each lab report and source codes zipped together has to be submitted by its corresponding deadline. During the lab session on the same day, demonstrate your working solution to a lab assistant. Be prepared to answer questions.

2 Lab Instructions

The objective of this lab is to become familiar with the project code and constructs from OpenMP and C++ Threads.

3 System description

Crowd (or Pedestrian) simulation is the study of the movement of people through various environments in order to understand the collective behaviour of crowds in different situations. Accurate simulations could be useful when designing airports or other public spaces, for understanding crowd behaviour departing sporting events or festivals, and for dealing with emergencies related to such situations. Understanding crowd behaviour in normal and abnormal situations helps improve the design of the public spaces for both efficiency of movement and safety, answering questions such as where the (emergency) exits should be put.

This system simulates an area of people[1] walking around. Each person, or *agent*, is moving towards a circular sequence of *waypoints*[2], in the 2D area. The starting locations and waypoints are specified in a *scenario configuration file*[3], which is loaded at the beginning of the program run. At the beginning of the project the agents are ghost-like - they may walk straight through each other without taking care of any collision. With each update of the world two things happen:

1. The *model* is updated, allowing agents to act.
2. The graphical visualization is updated.

The time that passes for each such update thus decides the frame rate.

3.1 Code base

The project is divided into two modules. The library *libpedsim* handles the pedestrian model, where all the simulation logic is performed. It contains classes to represent agents and necessary data structures for the algorithms used. *Demo* is an application that uses libpedsim and displays a visual representation. It also handles parsing of scenario files and sets up the world. Most of the code changes you make will be in libpedsim.

[1] Even though they look like little balls, they are really people, we assure you.
[2] A physical point in the 2D space
[3] such as the given "scenario.xml" in the project root directory

The update of the world/model is done in the *tick()* function in `ped_model.cpp`. Right now this function does not do anything; it will be your job in Assignment 1 to write the code so that the function works as intended.

4 Finding Bottlenecks and Parallelizing with C++ Threads and OpenMP

In this assignment you need to implement and parallelize the tick function in `ped_model.cpp`. The *tick* function is called for each time step of the simulation. The function is supposed to 1) retrieve each agent, 2) calculate its next desired position and finally 3) set its position to the calculated desired one. In Assignment 3 we will introduce collision detection but for now we just assume that an agent may go wherever it wants to even if several agents end up at the same position (they are ghost pedestrians!). Therefore, **do not use** the *move* function for now, as it is not relevant for Assignment 1, but for Assignment 3.

The desired position of an agent depends on its current destination. The destination is decided by a list of **waypoints**. When an agent reaches a destination, it picks the next destination from the list. This sequence is endless, as reached destinations are appended to the end of of the list. If you wish, you can make your own cool scenarios. Look at the source code to find out how.

For this assignment you need to make yourself familiar with mainly two classes: the Tagent class (ped_agent.cpp/h) and the Model (ped_model.cpp/h). Here you need to implement two versions, one that uses **OpenMP** and one that uses **C++ Threads**.

The instructions for Assignment 1:

A. Implement the serial version of the tick function in ped_model.cpp, such that in each time step each agent moves (without using the `move` function for Assignment 3).

B. Identify sources of parallelism.

C. Reflect upon different ways to apply OpenMP and C++ Threads. Consider, amongst others, data distribution, load balancing, and thread creation overhead.

D Choose the most suitable implementation for OpenMP and for C++ Threads. Implement the chosen parallelization and create a parallel OpenMP, and a parallel C++ Thread version. The three versions (serial, OpenMP and C++ Thread) must be interchangeable i.e., they should produce identical results. Make it possible to easily choose between implementations, either by recompilation or command-line argument.

E. Vary the number of threads employed and generate a plot showing the **speedup** against the number of threads, using the serial version as baseline. *Hint:* Use the `-timing-mode` option in order to run the simulation without GUI and to retrieve the performance statistics.

Running the same code with different implementations (OpenMP, threads, vector,...) for comparison can be cumbersome as you have to change some configurations and re-compile for each different implementation. A neat way to run the same code with either OpenMP, threads, vector implementation without re-compiling is to have a command line argument which selects which implementation to run. You may however choose yourself how to solve this part. Or there is also the change-the-variable-and-recompile option if you prefer more manual work.

4.1 Questions

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

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

C. Once for OpenMP and once for C++ Threads, explain your chosen implementation. Include, amongst others, answers to the following questions:

- How is the workload distributed across the threads?

- Which number of thread gives you the best results? Why?
- What are the possible drawbacks of this version?
- Why did you choose this version over the other alternatives?

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

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

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

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

5 Submission Instructions

Submit the code and a **short** report including all answers to above questions in **PDF** format to studium.

1. Your code **has** to compile.
2. **Clean** your project (run **make submission** which will copy code files from demo and libpedsim and make a tarball called submission.tar.gz.
3. Document how to choose different versions (serial, C++ Threads, and OpenMP implementation).
4. Include a specification of the machine you were running your experiments on.
5. State the question and task number.
6. State on the assignment whether all group members have contributed equally towards the solution - it will be important for your final grade.
7. Include all generated plots.
8. Put everything into a zip file and name it team-n-lastname$_1$-lastname$_2$-lastname$_3$.zip, where n is your team number.
9. Upload the zip file on studium by the corresponding deadline.

6 Acknowledgment

This project includes software from the PEDSIM[4] simulator, a microscopic pedestrian crowd simulation system, that was originally adapted in 2015 for the Low-Level Parallel Programming course at Uppsala University.

[4] http://pedsim.silmaril.org/