LLPP – Assignment 1 Report

Aljaz Kovac, Jakob Nordgren, Simon Jaklowsky

Introduction

Three versions of the tick() function were implemented. Via command line arguments, the user can choose which implementation to run as well as the number of threads for the parallel versions. An automatic set of scripts was also created to run experiments on and plot the speedup of different implementations and different number of threads. All team members collaborated on this assignment equally.

Usage

The demo program has been extended with the following command line arguments:

--implementation IMPL
Run with specified implementation **IMPL: SEQ | CTHREAD | OMP**
--threads N
Run with **N** number of threads when in **CTHREAD** or **OMP** mode.

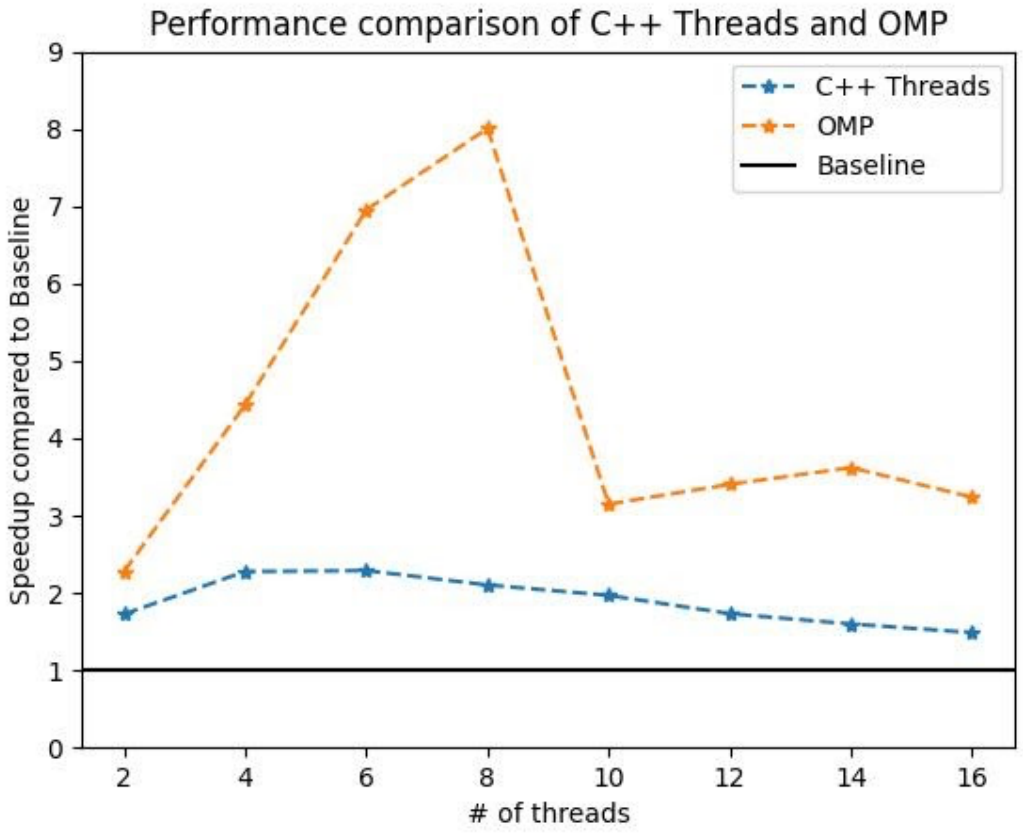
Example:
$ demo/demo --implementation OMP --threads 8 scenario.xml
Run the demo with the OpenMP implementation, using 8 threads

Implementations and benchmarking

The **serial** version uses a for-each loop to update each agent.
In the **C++ Threads** version, each thread is given a segment of the agent vector to operate upon.
Finally, the **OpenMP** version is close to identical to the serial version, but here a compiler directive is used to task OpenMP with parallelizing the loop.

Timing tests were performed on the UPPMAX system, in interactive mode:
interactive -A uppmax2021-2-28 -M snowy -p core -n 1 -c 8

Below is a plot showing the performance of different implementations:



Questions

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

Task-level parallelism (OpenMP) and Thread-level parallelism (C++ threads and OpenMP).

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

OpenMP: pragma omp parallel and pragma omp for.

C++: Creating short-lived worker threads on each computation (chosen implementation).

Alternatively, we considered creating a pool of worker threads during simulation setup, and then reusing these threads on every tick().

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

a. **How is the workload distributed across the threads?**

OpenMP: The default scheduler in OpenMP is static, so each thread receives the same sized chunks of the agent vector. The update operation is the same for every agent so workload is evenly distributed.

C++: The vector of agents is divided evenly among the threads (exception: if/when the number of threads does not divide the amount of agents evenly). If we assume that the workload for each agent in each timestep is the same then the total workload should also be evenly distributed across the threads.

b. **Which number of threads gives you the best results? Why?**

OpenMP: 8 threads gives the best result. Since the simulation was run with 8 cores on one node allowed the 8 threads to be distributed evenly on the 8 cores.

C++: 6 threads gives the best result. The overhead of creating/destroying threads when every time tick() is called might be the reason why the speedup starts decreasing when the number of threads is greater than 6.

c. **What are the possible drawbacks of this version?**

OpenMP: In this non-advanced task it is difficult to find any drawbacks with using OpenMP. Since the workload is balanced among the threads it is not necessary to do anything more advanced than static scheduling. In more complex applications there might be some drawbacks with using OpenMP due to the loss of fine grained control over how the workload is distributed among threads.

C++: We are creating and destroying threads at each call of tick().

d. **Why did you choose this version over the other alternatives?**

OpenMP: It was the simplest solution. Also, the resulting code is very readable, and looks very much like a serial implementation.

C++: It was reasonably simple to implement, and allowed for choosing the number of threads at runtime.

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

OpenMP gives a better speedup. This is due to OpenMP using thread pools, i.e, threads are created when needed, but not destroyed when they finish so that they can be reused if needed again during runtime. In this application a large amount of overhead is avoided compared to the C++ thread implementation where the threads are created and destroyed at each invocation of the tick() function.

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

Find the optimal number of threads to run with. Create the threads only at the start of the simulation, i.e outside of the tick() method/function, and destroy them at the end of the simulation.

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

C++: If we choose to run this with 2 threads, then we would have a chunk size of 3, and the 7th agent would "spill" into the last thread. So one thread would handle 3 agents and one would handle 4 agents.

OpenMP: If the number of threads were specified as above we would assume the distribution is the same.

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

OMP environment variables:

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_PLACES