Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP

Bjoern Knafla, Claudia Leopold

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 219-226, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

© 2007 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume38

Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP

Bjoern Knafla and Claudia Leopold

University of Kassel, Research Group Programming Languages / Methodologies
Wilhelmshoeher Allee 73, 34121 Kassel, Germany
E-mail: {*bknafla, leopold*}*@uni-kassel.de*

Future computer games need parallel programming to meet their ever growing hunger for performance. We report on our experiences in parallelizing the game-like C++ application OpenSteerDemo with OpenMP. To enable deterministic data-parallel processing of real-time agent steering behaviour, we had to change the high-level design, and refactor interfaces for explicit shared resource access. Our experience is summarized in a set of guidelines to help parallelizing legacy game code.

1 Introduction

Computer games have typically been written as sequential programs. To exploit multi-core game consoles and PCs, games need to be parallelized.

While parallelization has been actively researched and deployed for scientific applications for decades, computer games are a new area for this programming paradigm. Instead of writing parallel code from scratch, many game developers will lever their existing sequential code infrastructure by refactoring and rewriting the parts that profit most from parallelization.

We chose *OpenSteerDemo*[1] and the *OpenMP*[2] programming system to gain first insights into the parallelization of legacy game-like applications, with focus on artificial intelligence. OpenSteerDemo is a testbed for the C++ open-source library *OpenSteer* written by Reynolds[1]. It simulates and graphically displays the steering of autonomous computer-controlled characters, called agents, in real-time. Simple local environment-aware movement of individual agents emerges to a complex group behaviour.

Internally, OpenSteerDemo runs a main loop as it is typical for games. Each iteration simulates one time step, which cycles through the agents, and thereby simulates the movement and updates the state of each agent (position, velocity, direction), as well as graphically displays the agents. Agent positions are stored in a spatial data structure that supports queries for a given agent's neighbors (which influence the steering of the agent). Thus, in the sequential version, an agent sees the updated states of the agents processed before it, but the old states of the agents processed after it.

Our OpenMP-parallelized version of OpenSteerDemo treats agents as independent entities and processes them in a data-parallel fashion. Each cycle through the agents (update stage) is divided into a simulation sub-stage, followed by a modification sub-stage. During the simulation sub-stage, agents only read their local environment information and store their individual state change wish (called steering vector), independent of each other. In the modification sub-stage, all agents independently write their state based on the steering vector.

This high-level design, with its clear separation of read- and write-accesses to shared resources, results in a deterministic simulation with no need for locks. Details of the high-level design and further techniques for parallelization are outlined below. Measured speedups are 1.92 for two OpenMP threads, and 3.54 for four threads, for the update stage.

The paper starts with an introduction of OpenSteer and OpenSteerDemo, in Sect. 2. Then, Sect. 3 outlines problems with parallelization, and describes how we refactored and parallelized the code. The lessons we learnt are summarized in a set of guidelines in Sect. 4. Next, Sect. 5 contains time measurements, while Sect. 6 overviews related work. Section 7 finishes with conclusions.

2 OpenSteer and OpenSteerDemo

OpenSteer is an open-source C++ library that implements artificial intelligence (AI) steering behaviours for computer-controlled characters (agents) in games. Agents in OpenSteerDemo react to their surrounding only by means of these steering behaviours. Each steering behaviour, such as "walk along a given path" or "avoid an obstacle", corresponds to a steering vector that is computed from the current agent's state (position, orientation, velocity) and its local environment (nearby obstacles and neighboring agents, etc.). A steering vector describes an agent's "wish" to change its state through movement to react to the current situation. Simple behaviours may be combined into complex ones, by combining the corresponding vectors. Moreover, the movement of individual agents results in complex group behaviour.

OpenSteerDemo is a sandbox for rapid prototyping of combined steering behaviours. It runs the typical main loop of real-time games[3]. In each iteration, it passes through several stages, among them the update stage and the graphics stage:

update stage In this stage, each agent calls a method that computes and combines the steering vectors. Then, the agent writes its state and updates its entry in the spatial data structure used for fast neighbour search. Thus, when an agent is updated, it sees the new states of the agents processed before it, and the old states of the agents processed after it.

graphics stage After all agents have been updated, the graphics stage uses *OpenGL* to render the new world state (which comprises the states of all agents) into a picture called *frame*. The number of frames that can be drawn per second is called *frame rate*.

Typically, a game would pass the world state produced by the update stage into a separate physics stage to detect and resolve collisions and to establish a physically correct game world state. This is not done in OpenSteerDemo and hence not treated in this paper.

3 Refactoring and Parallelization

Parallelizing OpenSteerDemo required to refactor it. This section starts with a review of our first parallelization attempt without refactoring, and identifies problems in the code that made this approach fail. The rest of the section describes the new approach. Subsection 3.2 contains some general remarks. Then, subsections 3.3 and 3.4 are devoted to specific techniques. One of them, the separation of the update stage into a simulation and a modification sub-stage, is described in detail. Other techniques are only briefly sketched.

3.1 Problems for Parallelization

Our first approach to parallelize OpenSteerDemo was to incrementally insert OpenMP directives into the sequential code. Unfortunately, the program became messy this way, making reliability and correctness questionable. Moreover, a lot of critical sections were needed, which effectively serialized the program. We searched for reasons why the sequential code was hard to parallelize, and identified the following:

- Excessive use of global variables, deep inheritance hierarchies and strongly interdependent classes make the control and data flow hard to follow.
- As already explained, there are data dependencies between the updates of different agents within an iteration. Concurrent processing of the agents leads to race conditions.
- For some steering behaviours, a random number generator is called. The order of calls is deterministic in the sequential program, but may vary for different executions of the parallel program. Therefore, the same seed does not guarantee identical results, which complicates typical requirements of games, such as recording a simulation run for replay or the synchronization of game sessions over a network.
- OpenGL is not thread-safe. Therefore, concurrent OpenGL calls from different threads may lead to undefined behaviour.

3.2 Overall Approach

We based refactoring and parallelization of the program on the following considerations:

1. The steering vectors of different agents are computed independently. This naturally gives rise to data parallelism, which is the major source of parallelism in the program.
2. The order of simulating agents must not influence the outcome, otherwise race conditions would occur in a parallel environment. Data dependencies between agents were eliminated by modifying the program so that the whole update stage refers to the old world state, i.e., the state at the beginning of the time step. While this modification changed the outcome as compared to the sequential version, all agents are treated the same now, unrelated to their order of simulation.
3. For higher speedups, synchronization between agents should be reduced to a minimum.
4. For every frame, all agents need to be updated, otherwise agent movement looks choppy. Thus, all parallel processing must be finished by the end of the main loop iteration. Because of thread-safety problems with OpenGL, we restricted parallelization to the update stage, decoupled the update and graphics stage, and left the graphics stage for future work.
5. For simplicity, we do not use a spatial hash for the spatial data structure (as the sequential version does), but just a spatial hash interface around a C++ STL vector. Thus, query and access operations require time $O(n^2)$. We expect the parallel version of OpenSteerDemo to have further performance potential, if a more efficient, real spatial data structure is used.

3.3 Separation of Simulation and Update

We divided the update stage into a simulation sub-stage and a modification sub-stage. The simulation sub-stage reads in data and computes the combined steering vectors for all agents, whereas the modification sub-stage accomplishes the actual updates. Thus, there is no need for critical sections; a barrier after the simulation sub-stage and one after the modification sub-stage are sufficient. This modification was not as easy as it may appear, because accesses to global variables are difficult to recognize in the legacy code, especially where pointers are involved. To make shared resource usage explicit, we eliminated all global variables and now pass data into C++ functions and methods as arguments. C++'s *const* keyword helps in enforcing that data can only be read, and that methods can not modify the state of their object. The update stage uses a parallel region, inside which:

1. A parallel *for*-loop runs through all agents and computes their steering vectors (simulation sub-stage). Steering vectors are stored in a C++ *STL vector*, where each element corresponds to one agent and is only accessed by this agent's thread. We use the *dynamic* scheduling parameter of OpenMP because the computational expense of different agents may vary depending on their local environment, for example when they have to react to nearby obstacles.

2. Another parallel *for*-loop accomplishes the updates (modification sub-stage). Each agent changes its own state only, and therefore no race conditions may occur. *Static* scheduling is sufficient here.

3.4 Other Refactoring Techniques

Refactoring of OpenSteerDemo involved further techniques, which for brevity are only sketched here:

render feeder All agents get a reference to a so-called render feeder. The render feeder defers calls to the graphics subsystem, by first collecting graphics primitives to be rendered in a *thread storage* (see below), and later feeding them to the graphics renderer in a purely sequential stage. Within the update stage, threads can only access the graphics subsystem via the render feeder. Thus, the developer of steering behaviours is shielded from unintentional indirect non-thread-safe OpenGL calls.

thread storage Each of our OpenMP threads owns a data slot within a thread storage object. We use it to store graphics primitives that are created during the simulation and modification sub-stages. When accessing the storage, the thread is identified by its thread number, and only gets access to its associated slot. Therefore, no race conditions can occur. In sequential regions, all contained slots can be read. As an aside, this feature is difficult to use with nested parallelism, because OpenMP enumerates threads at different levels with the same numbers. Direct OpenMP support for truly unique thread numbers would be desirable.

random number generators Every agent uses its own random number generator, to guarantee deterministic behaviour. All functions and methods that need random numbers have been changed to get a reference to a generator as a parameter, instead of invoking a random number function with non-deterministic outcome somewhere in the code.

4 Lessons Learned

Based on our experiences with OpenSteerDemo, we developed the following guidelines, which we expect to be useful for parallelizing legacy game code in general:

- Refactor code for simplicity. Parallelization is error-prone and hard[4]. The more complex the code (e.g. deep inheritance hierarchies with interwoven method calls), the harder it is to find and prevent race conditions and non-thread-safe function calls.
- Keep the parallel structure simple, especially in the first version. If complex code is incrementally cluttered with OpenMP pragmas, it becomes hard to understand, and race conditions may easily slip in.
- Identify shared resources and make access to them (read, write, read/write) explicit in function and method interfaces. Use the `const` keyword of C++ where appropriate, to clearly indicate whether a function may give rise to race conditions.
- Change the high-level design (architecture) of the code to reduce the need for low-level coordination and synchronization.

Additionally, common and valuable software development knowledge suggests to:

- Profile first. Identify the section(s) of code that benefit most from optimization.
- Use tools that help analyzing, debugging, and profiling parallel programs. See reference[5] for more information.
- Measure performance to see if the parallel code scales. Deploy data parallelism where applicable, because compared to task parallelism it typically has higher scalability potential[9].

5 Performance

Performance was measured on a dual-processor dual-core 2 GHz AMD Opteron machine with 2 GB RAM and two Nvidia 7800 GTX graphics cards in SLI mode. The machine is running Linux.

Figure 1 depicts the average frame rates in frames per second (fps) for a simulation of pedestrians on a pathway, and Fig. 2 for a flock of birds. On the x-axis, different versions of OpenSteerDemo are arranged:

- the original version using a spatial hash
- the original version using a simple vector instead
- the parallel version with OpenMP support disabled in the compiler
- the parallel version run with 1, 2, and 4 threads (on 1, 2, and 4 processors, respectively).

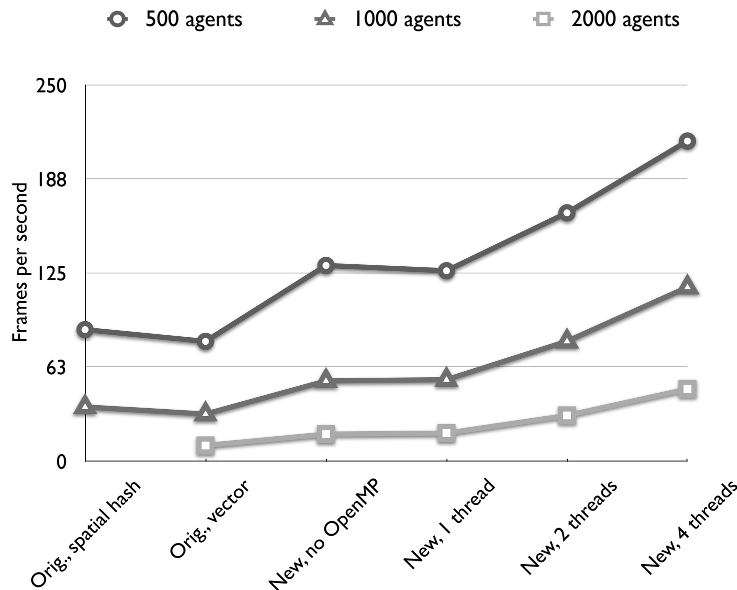

Figure 1. Simulation of pedestrians

The y-axis represents the frame rate. Curves marked with different symbols correspond to distinct numbers of simulated agents (see legend).

The figure refers to measurements for a whole iteration of the main loop, i.e., including both the parallel update stage and other, non-parallel stages such as graphics. For the pedestrian example, one can compute a speedup of about 2.65 for four threads (based on running time, not frame rate as in the figure). Not shown in the figure, we also measured the speedup of the update stage in separation, which is about 3.48 for four threads. For the birds example, the speedups are 2.84 for the whole iteration, and 3.54 for the update stage, respectively. All numbers have been computed taking the OpenMP-disabled version as the sequential base.

It can be observed that the speedup increases with the number of simulated agents. This may be partly due to a lower synchronization overhead per agent on the two OpenMP barriers, and chiefly to a higher computation-to-communication ratio in the case of many agents. More agents lead to a higher agent density since in our experiments they are located in a fixed-sized area or volume. Consequently, an agent needs to inspect more neighbors to evaluate its new position, and thus carries out more arithmetic operations that profit from parallelization.

Agents of the pedestrian example try to stay on a pathway, therefore crowding along the path, while the bird agents spread over the whole simulation world. Most pedestrian agents are surrounded by many other agents that they need to inspect to determine the ones influencing them. As the analysis of neighbors dominates the running time, the spatial hash does not show any advantages over the simple vector. In contrast, the birds spread freely in their simulation world, and therefore the spatial hash quickly excludes many agents from

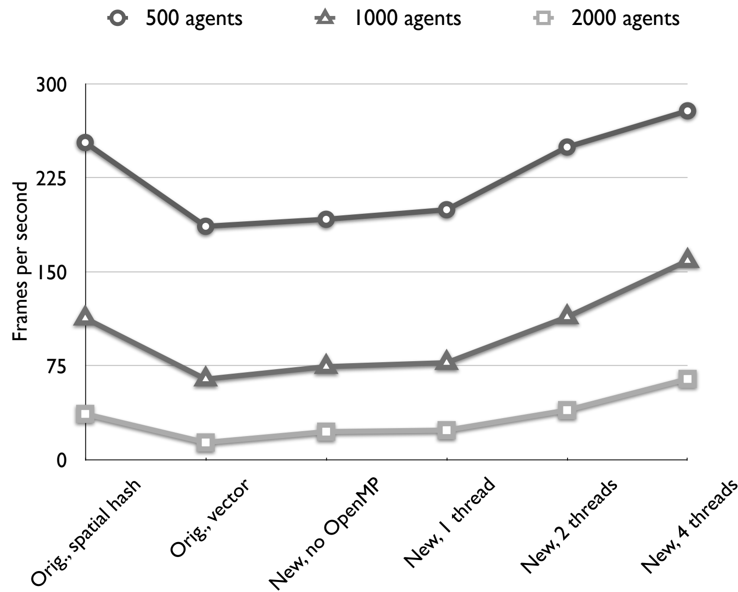

Figure 2. Simulation of a flock of birds

further inspection. With increasing bird density this effect decreases.

6 Related Work

Closest to our work is the *PSCrowd* system of Reynolds[6], which simulates crowds or flocks of agents on the Playstation 3. That platform is based on the *Cell* processor, which contains multiple heterogeneous cores and is programmed with specific libraries and tools. Quinn[7] simulates the behaviour of pedestrians in evacuation scenarios using *MPI*.

Unlike these works, we target homogeneous shared-memory computers, and deploy OpenMP. OpenMP is easier to use than other parallel programming systems, and may thus be a more frequent choice for the parallelization of legacy code.

As another difference, both Reynolds and Quinn subdivide space and map partitions (or more precisely the simulation of the agents inside these partitions) onto processors or processor cores. Quinn's system may experience load imbalances; PSCrowd balances load by using fine-grained partitions. In our approach, agents are mapped to processors with the help of OpenMP's static and dynamic scheduling parameters instead of using a spatial partitioning scheme. Therefore no explicit load balancing is needed.

Valve, an entertainment software and technology company, presented a parallelization approach for games in a press event in 2006[8]. Their approach does not use OpenMP, but is based on low-level synchronization primitives among threads, such as spin-locks and non-locking queues. Other work on parallelizing games has been presented at the last two years' Game Developers Conferences[10].

7 Conclusions

Parallelizing OpenSteerDemo required a lot of code refactorization. Among that was a high-level design change to split the update stage into a simulation sub-stage that reads data, and a modification sub-stage that writes them. Moreover, we changed interfaces by e.g. providing references to a render feeder and a random number generator. With these changes, we derived a simple data-parallel program with no need for critical sections. We are optimistic to gain even better performance with improved spatial data structures, and by introducing parallelism into parts of the graphics subsystem in the future.

An outcome of our work is the observation that it is worthwhile to first refactor complicated code before starting with parallelization. This and other observations have been formulated as a set of guidelines that may be useful for parallelizing game legacy code in general.

References

1. C. W. Reynolds, *OpenSteer Website*, `http://opensteer.sourceforge.net`, (2004)
2. OpenMP Architecture Review Board, *OpenMP Application Program Interface, Version 2.5 May 2005*, `http://www.openmp.org/ drupal/ mp-documents/ spec25.pdf` (2005)
3. D. Sánchez-Crespo, *Core Techniques and Algorithms in Game Programming*, New Riders, (2003).
4. E. A. Lee, *The Problem with Threads*, Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, (2006). The published version of this paper is in IEEE Computer, **39**, 33–42, (2006).
5. M. Süß, C. Leopold, *Common Mistakes in OpenMP and How To Avoid Them*, in: Proc. International Workshop on OpenMP - IWOMP'06, (2006).
6. C. W. Reynolds, *Big Fast Crowds on PS3*, in: Proc. 2006 ACM SIGGRAPH symposium on Videogames, (2006).
7. M. J. Quinn, R. A. Metoyer and K. Hunter-Zaworski, *Parallel Implementation of the Social Forces Model*, in: Pedestrian and Evacuation Dynamics 2003, E. R. Galea, (ed.), (2003).
8. J. Reimer, *Valve goes multicore*, `http://arstechnica.com/ articles/ paedia/ cpu/ valve-multicore.ars` (2006).
9. M. Voss, *De-mystify Scalable Parallelism with Intel Threading Building Block's Generic Parallel Algorithms*, `http://www.devx.com/ cplus/ Article/ 32935` (2006).
10. Game Developers Conference, `http://www.gdconf.com/` (2007).