Simulation and Rendering of Colliding Spheres

Isabel Rosa Alexandros-Stavros Iliopoulos Tao B. Schardl Charles E. Leiserson

Department of Electrical Engineering and Computer Science & Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

Abstract—**This assignment provides students with hands-on experience on engineering software for performance. Students are tasked with optimizing a graphical n-body simulation to run fast on a modern shared-memory multicore using serial and parallel program optimizations. This open-ended assignment invites students to develop and test optimizations to make the program run as fast as possible while still producing the same results. Students learn how to diagnose performance bottlenecks, develop serial and parallel program optimizations, and evaluate the correctness and performance of their changes. We found that students in 6.172, MIT's undergraduate course on performance engineering of software systems, are excited by this project, especially seeing their optimizations improve the visual smoothness of the simulation and enable it to handle larger problem sizes and tighter time constraints. The materials for the assignment are publicly available at** `https://github.com/ailiop/EduHPC-22-Peachy-Sphere-Simulation`.

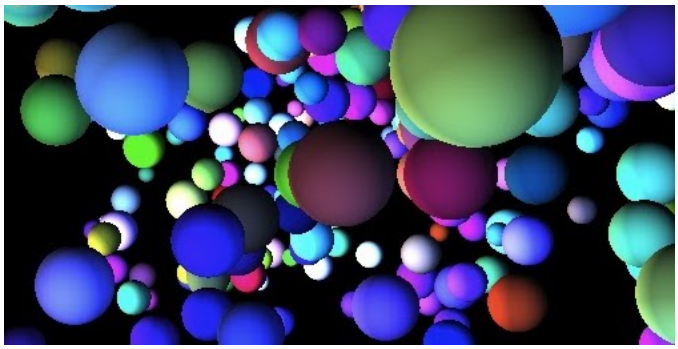

Fig. 1: Screenshot from a simulation with 250 spheres with different volumes, masses, and colors.

I. Assignment overview

This assignment teaches students about parallel computing and ***software performance engineering*** — writing code that runs fast and uses computing resources efficiently — through a project to optimize a graphical n-body simulation. Students are given a fully functional 400-line serial program in C that performs two tasks: (1) a physical simulation of massive spheres interacting via gravity and elastic collisions, and (2) a graphical rendering of the simulation using ray tracing without reflections. Figure 1 shows a screenshot of such a simulation's rendering. Students are tasked with speeding up this reference implementation for a modern shared-memory multicore computer that is widely available via the cloud. The assignment is open-ended, inviting students to think creatively and explore their own ideas to make the program run as fast as possible. This assignment gives students hands-on experience with many aspects of software performance engineering, from identifying and evaluating performance bottlenecks using modern tools, to experimenting with optimization techniques, including algorithmic improvements, serial program optimizations, and task parallelism using OpenCilk [1], [2].

Student implementations are evaluated with a collection of 80 input sets called ***tiers***. Each tier increases the size of the problem, i.e., the number of spheres and the rendering resolution. An implementation passes a tier if it reaches a target frame rate and its output matches that of the reference implementation exactly. Students are not expected to clear all available tiers. The assignment materials include utilities for testing the correctness and performance of an implementation on a range of tiers and for visualizing errors in rendered scenes.

Although the reference code base is simple and presents a few easy optimization opportunities, there are several challenges to achieving high performance. The simulation and rendering components of the reference implementation take comparable amounts of time. Students must optimize both components, as improvements in one can cause the other to become the bottleneck. The assignment handout discusses several algorithmic optimizations to reduce the complexity of the initial serial computation, such as geometrically pruning redundant ray-tracing computations and exploiting the symmetry of gravitational forces. Some of these optimizations make parallelization more challenging. For example, naively parallelizing a triangular loop iteration for symmetric pairwise force calculation introduces races.

The assignment uses the Cilk language extensions to C and the OpenCilk task-parallel platform for parallelization [1]–[3]. Students are also encouraged to use two productivity tools in OpenCilk to check for determinacy races [4] and measure the work and span of parallel computations [5], [6]. This choice of parallel programming platform simplifies the students' work to schedule and load-balance the parallel computation and enables them

to explore parallel algorithmic improvements with relative ease. In addition, the assignment requires students to submit deterministic parallel code, which the students can verify using the OpenCilk determinacy-race detector.

The assignment is designed for students to tackle in teams of two over a period of three weeks. There are deadlines for two submissions of implementations and write-ups: a beta and a final submission. The beta phase lasts two weeks and students are instructed to focus on algorithmic and serial optimizations before parallelizing their implementations. After the beta deadline, the code and corresponding tiers for all submissions is made public to all students, who are free to peruse but not directly copy code. The final phase lasts one week. Students are given explicit tier-to-grade thresholds for the beta but not the final submission, whose performance targets are left open-ended. Performance targets for both submissions are determined with optimized implementations by the course staff. Students do not compete with each other.

II. Key concepts

The assignment covers various concepts of shared-memory parallel programming and software performance engineering. Parallel programming concepts include task-parallel algorithms, races, reductions, work/span analysis, and coarsening to reduce scheduling overhead. Performance-engineering concepts include profiling to identify performance bottlenecks, the importance of algorithmic improvements and serial optimizations before adding parallelism, memory locality, and basics of floating-point operations and non-associativity.

III. Context and requirements

This assignment was used in the Fall of 2021 as part of the MIT course 6.172 *Performance Engineering of Software Systems* (renumbered in 2022 to 6.106). Between 130 and 160 students take the course each year, the majority of whom are junior or senior undergraduates. Course prerequisites include undergraduate algorithms, undergraduate computer architecture, and software engineering, but not parallel programming. During the course, students learn and apply performance-engineering skills including serial and parallel performance analysis, bit tricks and vectorization, assembly language and compiler optimizations, task-parallel programming, races and synchronization, and cache-oblivious algorithms.

The course, which has been taught and developed over 14 years, is structured around four multiweek team projects. This assignment is the second project in the course, but the first where students experiment with parallel programming.

The median achieved speedup for this assignment among students in the Fall of 2021 was roughly 80×. The top-performing teams achieved over 130×.

IV. Strengths and variants

A key strength of the assignment is that it is realistic and engaging. Students exercise a variety of techniques for performance engineering, including parallelization, to optimize a program with several interacting components. The easy-to-master initial code base enables students to start implementing optimizations quickly. Performance improvements have direct impact visually as smoother simulations and quantitatively as cleared tiers. Together with the open-ended design of the project, these features encourage students to exercise their creativity and look for ways to further improve performance. Moreover, students find the graphical element of the assignment exciting. For example, one student posted the following on MIT Confessions, a Facebook page where MIT students submit anonymous notes [7]:

> This 6.172 project is so cool, I can't believe we're really out here optimizing a ray tracing engine. It's projects like this that make me really think about how lucky I am to be going to this school to face interesting challenges.
>
> Plus the renderings are the prettiest thing ever, so thankful to all the work that went into making this project for this year!!

Several elements of the assignment's design facilitate its adoption. Multicore machines are practically ubiquitous and widely available via the cloud. C is a commonly used programming language, especially for applications where performance is a concern. The tiers system for performance evaluation makes it easy to grade submissions and can be tweaked to set different performance expectations. Although we contend that using OpenCilk for parallelization has multiple benefits, especially for students with no experience in parallel programming, it is possible to use other parallel platforms like OpenMP [8] or Intel oneAPI Threading Building Blocks [9].

It is possible to narrow or widen the scope of the assignment with relatively small modifications. Shorter projects, for example, might focus on optimizing only the simulation or rendering component. Different choices for correctness testing can also affect the assignment scope. We require the program to be deterministic and to produce identical results to the reference implementation, which precludes some optimization strategies that students might explore, such as approximation techniques. We chose this approach primarily to tailor the assignment to our target audience of undergraduates who are new to performance engineering and parallel programming. Relaxing these requirements opens up new possibilities for performance improvements.

References

[1] OpenCilk. [Online]. Available: https://www.opencilk.org/

[2] T. B. Schardl, I.-T. A. Lee, and C. E. Leiserson, "Brief Announcement: Open Cilk," in *SPAA*, pp. 351–353.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *JPDC*, vol. 37, no. 1, pp. 55–69.

[4] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in Cilk programs," *TCS*, vol. 32, no. 3, pp. 301–326.

[5] Y. He, C. E. Leiserson, and W. M. Leiserson, "The Cilkview scalability analyzer," in *SPAA*, pp. 145–156.

[6] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson, "The Cilkprof scalability profiler," in *SPAA*, pp. 89–100.

[7] MIT Confessions - Post #49891. [Online]. Available: https://bit.ly/3e13dz6

[8] OpenMP. [Online]. Available: https://www.openmp.org/

[9] Intel oneAPI Threading Building Blocks. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html