Peachy Parallel Assignments (EduHPC 2022)

Rocío Carratalá-Sáez*, Arturo González-Escribano*, Alexandros-Stavros Iliopoulos†, Charles E. Leiserson†, Charlotte Park†, Isabel Rosa†, Tao B. Schardl†, Yuri Torres*, David P. Bunde‡

*_Departamento de Informática_
Universidad de Valladolid
Valladolid, Spain
{rocio,arturo,yuri.torres}@infor.uva.es

†_Department of Electrical Engineering and Computer Science &_
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA, USA
{ailiop,cel,cispark,neboat}@mit.edu, isrosa12300@gmail.com

‡_Department of Computer Science_
Knox College
Galesburg, IL, USA
dbunde@knox.edu

***Abstract*—Peachy Parallel Assignments are model assignments for teaching parallel computing concepts. They are competitively selected for being adoptable by other instructors and "cool and inspirational" for students. Thus, they allow instructors to easily add high-quality assignments that will engage students to their classes.**

This group of Peachy assignments features two new assignments. The first has students speed up a graphical n-body simulation by identifying performance bottlenecks, making algorithmic improvements, and parallelizing the program using OpenCilk. The second has them parallelize a Monte Carlo hill climbing algorithm using one or more of OpenMP, MPI, and CUDA or OpenCL.

***Index Terms*—Peachy Parallel Assignments, parallel computing education, High-Performance Computing education, performance engineering, n-body simulation, ray tracing, Monte Carlo simulation, hill climbing**

I. Introduction

In a perfect world, every course would feature high-quality assignments, ones that force students to apply the course material and master it, ideally while exciting the students so they are drawn into the assignments and provide word-of-mouth advertising for the course and computing more generally. Creating an assignment with all these properties is not easy, however, because it requires a spark of inspiration as well as the standard work of developing starter code, an assignment description, etc. In addition, as with any new assignment, there is a risk of misjudging its difficulty, not providing sufficiently clear instructions, or otherwise failing to deliver on the idea.

Peachy Parallel Assignments aim to address this issue by recognizing high-quality assignments in parallel computing to encourage their reuse and to incentivize the development of new ones. Proposed Peachy Parallel Assignments are submitted to the EduHPC and EduPar workshops. Submissions are evaluated according to the following criteria:

- Tested: All assignments must have been successfully used in a class with real students.
- Adoptable: The assignments should include the materials needed for other instructors to adopt them, such as starter code and handouts. Ideally, they should also be suitable for a range of classes, covering commonly-taught topics with commonly-used technologies and offering a variety of customization options to make them appropriate for a wide variety of institutions and instructor goals.
- Cool and inspirational: The assignments should excite students through interesting applications of the material, appealing graphics, etc. Ideally, students should be excited to work on the assignment and want to tell friends and family about it.

The best assignments are published in the workshop proceedings (e.g. [1], [2]) and then (with all the materials required to adopt them) archived at the Peachy Parallel Assignments website (https://tcpp.cs.gsu.edu/curriculum/?q=peachy).

This paper presents two new Peachy Parallel Assignments which were presented at EduHPC 2022. Section II presents an assignment to improve the performance of an n-body simulation and its ray tracing rendering engine. This assignment gives the students practice with modern performance measurement tools and creates colorful graphical output. It is intended for use on a multicore system using OpenCilk. Section III describes an assignment to parallelize a Monte Carlo hill climbing algorithm. The problem is easy to understand but provides interesting optimization opportunities through its structure and use of (pseudo-)randomness. A nice aspect of this assignment is that versions are provided for OpenMP, MPI, and GPU programming (CUDA). Instructors can adopt one of these as a single assignment or have students parallelize the program using more than one paradigm to illustrate the differences between them.

II. Simulation and Rendering of Colliding spheres

Our first assignment provides students with hands-on experience with engineering software for performance. Students are tasked with optimizing a graphical n-body simulation to run fast on a modern shared-memory multicore system 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 within fixed time constraints. The materials for the assignment are publicly available at https://doi.org/10.5281/zenodo.7114642.

A. 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 [3], [4].

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. Students can run and visualize simulations locally on their computers and are also given remote access to a pool of remote multicore computers on which their submissions are evaluated. 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.

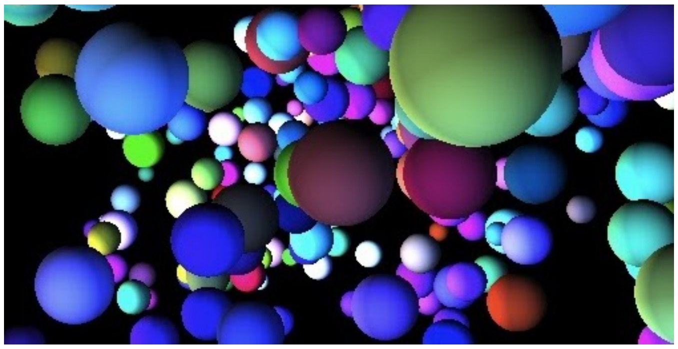


Fig. 1. Screenshot from a simulation with 250 spheres with different volumes, masses, and colors.

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. In addition, efficient implementations must balance parallelization against algorithmic optimizations that reduce work complexity. The assignment handout discusses several optimizations to reduce the work of the initial serial computation, such as exploiting the symmetry of gravitational forces and pruning redundant ray-tracing computations via projections of sphere bounding boxes. There are also several opportunities to use parallelism to optimize both simulation and rendering. For instance, gravitational forces could be computed independently as parallel reductions among all spheres, and rendering rays may be traced in parallel across pixels. But some algorithmic optimizations make parallelization more challenging. For example, naively parallelizing a triangular loop iteration for symmetric pairwise force calculation introduces races. Similarly, one can reduce ray-sphere intersection calculations by processing spheres in order of distance from the rendering viewpoint, but doing so limits the amount of available parallelism. In general, high-performance student implementations should introduce sufficient parallelism without compromising the work-efficiency and locality of serial optimizations.

The assignment uses the Cilk language extensions to C and the OpenCilk task-parallel platform for parallelization [3]–[5]. Students are also encouraged to use two productivity tools in OpenCilk to check for determinacy races [6] and to measure the work, span, and scalability of their parallel computations [7], [8]. 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 OpenCilk determinacy-race detector allows students to verify that their parallel code is deterministic, as the assignment requires.

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 each submission 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.

B. 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 making algorithmic improvements and serial optimizations before adding parallelism, memory locality, and basics of floating-point operations and non-associativity.

C. Context and requirements

This assignment was first used in the Fall of 2021 as part of the MIT course 6.172 *Performance Engineering of Software Systems*.[1] 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. For each project, students are given a working C program and are tasked with accelerating it. 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 $80\times$. The top-performing teams achieved over $130\times$.

D. 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 [9]:

> 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 important. 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 [10] or Intel oneAPI Threading Building Blocks [11].

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 such as using approximate computation methods. We chose this approach to make it easy to verify correctness and 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.

III. Hill Climbing with Monte Carlo

For our second Peachy assignment, we present the fifth example in a series of assignments used in a Parallel Computing course at the Universidad de Valladolid. In each of these assignments, students parallelize a program using different parallel programming models. The assignment targets concepts of shared-memory programming with OpenMP, distributed-memory programming with MPI, and/or GPU programming with CUDA or OpenCL. This assignment is based on a Monte Carlo probabilistic approach for a Hill Climbing algorithm in order to locate the maximum values of a two dimensional function. The program is designed to be simple, easy to understand by students, and to include specific parallelization and optimization opportunities. Although there is a quite

[1]MIT course 6.172 was renumbered to 6.106 in 2022.

direct parallel solution in the three programming models, the program has plenty of opportunities for further improvements. It maintains the same core concepts used in the four previously presented assignments, with a different design approach. It focuses on dealing with non-determinism during execution, the impact of randomization on load-balance, and new relevant optimization challenges with high performance impact. This assignment has been successfully used in parallel programming contests during an optional Parallel Programming course in the third year of Computer Engineering degree.

A. Idea and context

a) Idea: Different programming models use different approaches for the parallelization of application structures. Understanding these differences is key for students to get into more advanced techniques, and to face parallel programming in current heterogeneous platforms. For several years, we have been teaching an optional course of Parallel Programming in the Computer Engineering degree at Universidad de Valladolid. The course introduces the basics of OpenMP, MPI, and CUDA or OpenCL. Four previous peachy parallel assignments have been presented in this series [2], [12]–[14]. The assignment material for all the five assignments can be found at *https://trasgo.infor.uva.es/peachy-assignments/*. All of them are designed to be parallelized by the students during three one-week programming contests, using a single programming model on each one. During the contest they work to obtain the best performance with a mixed competitive and collaborative strategy [15]. Although this kind of assignment can be used to teach a single programming model, using it with different models also shows which concepts and techniques can be reused across the models and which cannot, exposing the differences between models. For example, the students learn the differences between controlling race conditions in shared-memory vs. using distributed data structures with explicit communications, or dealing with tiling and memory hierarchies in GPU coprocessors.

b) Contents: The codes of previous assignments were based on synchronized iterations, with a mix of stencil computations and/or multiagent systems that execute one simulation step in each iteration. Instead, this new assignment is based on the flow of multiple parallel searches with continuous interactions across them without global synchronization points. It can be optimized in many different ways depending on the parallelization and detail level. The assignment also shows the effect of randomization on load-balance, and introduces different possibilities to deal with non-trivial race conditions and cache effects. It maintains a clear focus on simple but effective code parallelizations and optimizations, while introducing more opportunities for advanced students and new and different choices in the three programming models considered.

c) The assignment code: Multi-dimensional functions are used to model a variety of physical systems such as heat transfer, pressure flow, structural vibrations, etc. Finding the values and locations of local and global maximums of such functions is important for the design and safety of many tools and physical structures used in normal day life. Analytical solutions can be really hard to find so numerical approaches are used in many situations.

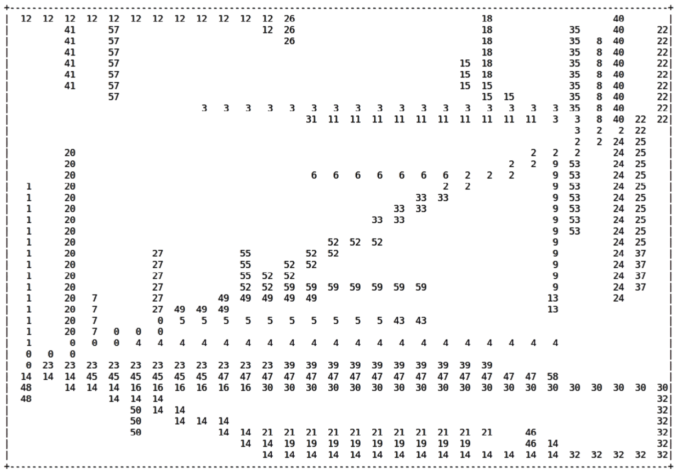
The program provided to the students is based on a classic probabilistic approach to find the maximum values of a two dimensional function on a rectangular area: A Monte Carlo method with a Hill-Climbing algorithm. The function being optimized is predefined at compile time. It is selected to have multiple local maximums and other interesting features in the search area. The search area is discretized in a lattice with a number of 2D control points. The hill-climbing approach is based on selecting a random starting point and iteratively moving the selected point to one of its four neighbors, the one with the highest function value. When the selected point has a value greater than or equal to the values at neighboring locations, a local maximum has been reached and the search stops.

The program uses a probabilistic approach to find the global maximum. It starts a parametric number of hill-climbing searches. Each search can be executed in parallel. To avoid recomputing the costly function, a memoization strategy is used; a matrix is initialized to non-valid values, and a cell is updated with the corresponding function value when a searcher checks that point. Also, when a searcher moves to a cell already visited by a previous searcher, the search can stop. Ancillary structures are used to store which searcher visits each cell for the first time, and which searcher follows the trail of another searcher. After all the searches have finished, the program computes the number of trail steps leading to each local maximum, the total number of travelled points, and the heights of each local maximum found.

Although parallel searches evolve in a non-deterministic way in a shared-memory model, the program results (travelled cells, computed heights, and final statistics) are fully determined by the initialization arguments, which include random seeds to make the results reproducible. The arguments can be chosen to generate specific situations with different numbers of local maximums, main travelling directions across the lattice, numbers of times when searches collide at lattice positions, etc.

B. Using the assignment

a) Contest tools: As in the previous assignments of this series, the provided material includes a sequential code, a test-bed of input arguments, and a handout explaining the assignment. The students can use common compilers and PC platforms to develop and test their codes. The students can use the output of the sequential code to check the results of their parallelized versions. An automatic judge tool with an on-line public ranking is used to provide a fair arena, and to keep the students engaged during the contests with competitive and collaborative rewards [15], [16]. This judge tool has been used since the first assignment in this series, and it has been improved course after course. The judge configuration is done by simply providing tuples of input arguments (representing the scenarios chosen by the teacher), and the



```
+------------------------------------------------------------------------------------------------------------------------+
|  12  12  12  12  12  12  12  12  12  12  12  12  26                                  18                      40        |
|          41      57                          12  26                                  18              35      40      22|
|          41      57                              26                                  18              35   8  40      22|
|          41      57                                                                  18              35   8  40      22|
|          41      57                                                              15  18              35   8  40      22|
|          41      57                                                              15  18              35   8  40      22|
|          41      57                                                              15  15              35   8  40      22|
|                  57                                                                  15  15          35   8  40      22|
|                                   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3  35   8  40      22|
|                                                    31  11  11  11  11  11  11  11  11  11  11   3   3   8  40  22  22|
|                                                                                                       3   2   2  22    |
|                                                                                                       2   2  24  25    |
|          20                                                                                   2   2   2      24  25    |
|          20                                                                               2   2   9  53      24  25    |
|          20                                           6   6   6   6   6   6   6   2   2   2       9  53      24  25    |
|   1      20                                                                   2   2               9  53      24  25    |
|   1      20                                                              33  33                   9  53      24  25    |
|   1      20                                                          33  33                       9  53      24  25    |
|   1      20                                                      33  33                           9  53      24  25    |
|   1      20                                                                                       9  53      24  25    |
|   1      20                                              52  52  52                               9          24  25    |
|   1      20              27              55          52  52                                       9          24  37    |
|   1      20              27              55      52  52                                           9          24  37    |
|   1      20              27              55  52  52                                               9          24  37    |
|   1      20              27              52  52  59  59  59  59  59  59  59                       9          24  37    |
|   1      20   7          27          49  49  49  49  49                                          13          24        |
|   1      20   7          27  49  49  49                                                          13                    |
|   1      20   7           0   5   5   5   5   5   5   5   5   5   5  43  43                                            |
|   1      20   7   0   0   0                                                                                            |
|   1       0   0   0   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4                    |
|   0   0   0                                                                                                            |
|   0  23  23  23  23  23  23  23  23  23  23  23  39  39  39  39  39  39  39  39  39  39                                |
|  14  14  14  45  45  45  45  45  45  47  47  47  47  47  47  47  47  47  47  47  47  47  47  47  58                    |
|  48      14  14  14  16  16  16  16  16  16  30  30  30  30  30  30  30  30  30  30  30  30  30  30  30  30  30  30  30|
|  48              14  14  14                                                                                        32|
|                      50  14  14                                                                                    32|
|                      50      14  14  14                                                                            32|
|                      50              14  14  21  21  21  21  21  21  21  21  21  21  21      46                    32|
|                                          14  14  19  19  19  19  19  19  19  19  19          46  14                32|
|                                              14  14  14  14  14  14  14  14  14  14  14  14  14  14  32  32  32  32  32|
+------------------------------------------------------------------------------------------------------------------------+
```

Fig. 2. Graphical representation of the resulting search trails. Each number corresponds to the searcher id that first travelled a cell.

expected output results. The students do not have access to these secret scenarios. The tool executes the programs on a real parallel system. In order to rank the students the judge tool measures the total number of tests passed before a fail, and the program performance. Depending on which test fails, the students receive a hint on typical problems that can cause failures on that specific test case. The sections of the sequential code that should be parallelized and optimized by the students are clearly marked to avoid parts involved in arguments processing, scenario initialization, OpenMP/MPI/CUDA setup, time measurement, and results output. The original codes can be directly compiled and run by the students, or even submitted to the judge tool before starting to parallelize them.

b) Test cluster: The most appropriate machines from our laboratory cluster are used for each programming contest. For OpenMP, we use a server with 64 cores that can easily show interesting effects related to the use of its 4 NUMA nodes. For MPI, we use two similar interconnected servers with 12 and 32 physical threads respectively. Although these servers have highly different processor performance, we do not discuss this with the students; we consider proportional load distribution in heterogeneous clusters an advanced topic and do not cover it in our course. During the CUDA/OpenCL contest, we use the same servers as in the MPI contest. They are equipped with several NVIDIA GPUs (CUDA 3.5 architecture).

c) Teaching context: The students in our course have already studied concepts of operating systems and concurrency, and they have used the C programming language in a couple of previous courses. There were 44 students enrolled. The degree of participation was high and they made more than 8,100 requests for program execution, including both tests and judgment requests. The assignments are worth 60% of the course grade. For each programming model, along with their final code, the students submit a short video. They also answer live questions from the course staff for 15 minutes. Their position in the leaderboard is also taken into account for the assignment grade.

d) Student's satisfaction: We conducted a survey at the end of the course. For the question "Are you satisfied with the overall experience of the course, activity types, evaluation method, etc.?", the students gave an average rating of 4.12 on a Likert scale from 1 to 5. The students also agreed that the assignment illustrates the main concepts of the course and provides opportunities to deepen their knowledge of the subject. For example, some students optimized their CUDA codes to obtain results 2.3 times faster than the baseline parallel version.

C. Concepts covered

This assignment covers an important class of parallel programs, those based on probabilistic methods and parallel searches with non-trivial interactions. It also presents an interesting real case based on classical algorithm concepts taught in previous courses.

a) Program structure: The program has three main stages: (1) Parallel search, (2) Computing the accumulated trail steps of all searchers reaching each local maximum, and (3) Computing other statistical data using different types of reductions. If desired, the program can also print a text-mode graphical representation of the search with two parts. The first one is a visualization of the trails followed by the searchers (see Fig. 2). This shows the non-deterministic behaviour of each parallel execution in shared-memory models, as the result depends on which searchers arrive at each cell first. The second one is a deterministic result, showing a map with the computed heights. A full example of the output is in the handout provided to the students.

b) Concepts: The basic concepts covered in the OpenMP version of the assignment are loop parallelization, reductions, and elimination of non-trivial race conditions. Some of these can be eliminated by a combination of atomic operations and code restructuring, others by adding or eliminating ancillary structures.

In MPI, the students work with array partitions, reductions, asynchronous operations and communicators. They also discover how to eliminate non-determinism in the searches and how to deal with an unknown number of communication steps and variable size communications.

For GPU programming, the main ideas are embarrassingly parallel kernels, thread-block geometries and sizes, the elimination of non-trivial race conditions, and the usage of simple reductions. The program also shows how randomization may lead to load balance, and the use of fixed-point arithmetic to avoid precision and concurrency problems in many situations, except when floating point mathematical functions are used in GPUs.

Several advanced optimizations can be discovered and applied. For example, changes in the main data structures can result in a large improvement in cache performance for OpenMP and MPI and better coalescencing in CUDA/OpenCL. Students can also use aggressive code reorderings to allow easier parallelization, kernel fusing, and shared memory or non-trivial reductions on GPUs.

D. Variants

Many different scenarios with different behaviours and results can be chosen by the instructor through proper selection of input arguments. For example, different locations and zoom values can be used to select areas with only one maximum or many different local ones, slopes with different gradients in one or more directions, etc. This leads to different combinations of workload, frequency of race conditions in shared memory, or communication combinations and volume in distributed memory. This flexibility is used to choose different test cases for the judge tool to run on each contest. The hard-wired function can also be changed, leading to very different situations.

For simpler assignments, a much shorter and easier version can be devised by simply eliminating the second stage of the program. For a more complex assignment, a hybrid approach using more than one programming model at a time is also possible. Finally, better graphical and online interfaces could be devised to enrich the learning experience. For example, the output text showing the computed heights can be easily transformed in a graphical heat map using external tools.

References

[1] H. Bücker, H. Casanova, R. F. da Silva, A. Lasserre, D. Luyen, R. Namyst, J. Schoder, and P.-A. Wacrenier, "Peachy parallel assignments (EduPar 2022)," in *Proc. 12th NSF/TCPP workshop on parallel and distributed computing education (EduPar)*, 2022.

[2] H. Casanova, R. F. da Silva, A. Gonzalez-Escribano, H. Li, Y. Torres, and D. P. Bunde, "Peachy parallel assignments (EduHPC 2021)," in *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2021)*. St. Louis (MO), USA: IEEE, 2021.

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

[4] T. B. Schardl, I.-T. A. Lee, and C. E. Leiserson, "Brief Announcement: Open Cilk," in *SPAA*, 2018, pp. 351–353.

[5] 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, 1996.

[6] M. Feng and C. E. Leiserson, "Efficient detection of determinacy races in Cilk programs," *TCS*, vol. 32, no. 3, pp. 301–326, 1999.

[7] Y. He, C. E. Leiserson, and W. M. Leiserson, "The Cilkview scalability analyzer," in *SPAA*, 2010, pp. 145–156.

[8] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson, "The Cilkprof scalability profiler," in *SPAA*, 2015, pp. 89–100.

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

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

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

[12] E. Ayguadé, L. Alvarez, F. Banchelli, M. Burtscher, A. Gonzalez-Escribano, J. Gutierrez, D. Joiner, D. Kaeli, F. Previlon, E. Rodriguez-Gutiez, and D. Bunde, "Peachy parallel assignments (EduHPC 2018)," in *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2018)*. Dallas (TX), USA: IEEE, 2018.

[13] M. Agung, M. A. Amrizal, S. Bogaerts, R. Egawa, D. A. Ellsworth, J. Fernandez-Fabeiro, A. Gonzalez-Escribano, S. Kundu, A. Lazar, A. Malony, H. Takizawa, and D. P. Bunde, "Peachy parallel assignments (EduHPC 2019)," in *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2019)*. Denver (CO), USA: IEEE, 2019.

[14] H. Casanova, R. F. da Silva, A. Gonzalez-Escribano, W. Koch, Y. Torres, and D. P. Bunde, "Peachy parallel assignments (EduHPC 2020)," in *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2020)*. Atlanta (GE), USA: IEEE, 2020.

[15] A. Gonzalez-Escribano, V. Lara-Mongil, E. Rodriguez-Gutiez, and Y. Torres, "Toward improving collaborative behaviour during competitive programming assignments," in *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2019)*. Denver (CO), USA: IEEE, 2019.

[16] J. Fresno, A. Ortega-Arranz, H. Ortega-Arranz, A. Gonzalez-Escribano, and D. Llanos, *Gamification-Based E-Learning Strategies for Computer Programming Education*. IGI Global, 2017, ch. 6. Applying Gamification in a Parallel Programming Course.

Appendix: Reproducibility

This appendix gives details relevant to reproducibility for each assignment.

E. Simulation and Rendering of Colliding Spheres

The materials for the sphere simulation and rendering assignment, as used in the MIT course 6.172 in Fall 2021, are archived with Zenodo and available at the following URL: https://doi.org/10.5281/zenodo.7114642. The archived materials include the reference software, utility programs, and handout document that were provided to students.

Students did their development work on local virtual machines with an image of Ubuntu 18.04 that included relevant software for the course. The course staff distributed the virtual machine as a VMWare virtual disk image. The virtual machines included OpenCilk 1.0 for compilation, scalability analysis, and determinacy-race detection,[2] and the OpenGL Utility Toolkit (`freeglut3-dev`) for visualizing rendered scenes.

The student handout refers to a collection of repos on MIT's GitHub (github.mit.edu) which are not publicly accessible. They were created for each team of students, who used them for developing their projects. The initial contents of each student repo were the same as the archive materials for this assignment (sans the Peachy Parallel Assignment overview document).

For performance measurements, students ran their programs on a cluster of c4.2xlarge instances on Amazon Web Services (AWS). The AWS instances were managed by the course staff and were configured to disable simultaneous multithreading (or "hyperthreading") and reduce performance variability. Students submitted jobs through custom scripts that operate similarly to the SLURM `srun` utility but specifically target the AWS queue managed by the course staff. These scripts (`awsrun` and `awsrun8`) are mentioned in the student handout but are not publicly accessible. The performance-testing system can be replaced with any cluster of quiesced machines and corresponding job submission manager.

F. Hill Climbing with Monte Carlo

The hill climbing assignment has been used in the context of a Parallel Computing course, in the third year of the Computing Engineering grade at the University of Valladolid (Spain).

[2]Precompiled binaries for OpenCilk 1.0 can be downloaded at https://github.com/OpenCilk/opencilk-project/releases/tag/opencilk/v1.0. Installation instructions and the latest version of OpenCilk can be found at https://www.opencilk.org.

The material of the assignment, including a handout, the starting sequential code, and some example input data sets will be made publicly available through the CDER courseware repository. They can also be found at our Peachy Assignments web page: https://trasgo.infor.uva.es/peachy-assignments/.

The on-line judge used in the programming contests is named Tablon, developed by the Trasgo research group at the University of Valladolid (https://trasgo.infor.uva.es/tablon/). It uses the Slurm queue management software to interact with the machines in the cluster of our research group. During the course we used the Slurm 18.08.3 release.

The machine in the cluster used for the OpenMP contest is named *heracles*. It is a server with four AMD Opteron 6376 @ 2.3Ghz CPUs with a total of 64 cores and 128 GB of RAM.

The machines used for the CUDA/OpenCL contests are named *hydra* and *medusa*. Hydra is a server with two Intel Xeon E5-2609v3 @1.9 GHz CPUs, with 12 physical cores, and 64 GB of RAM. It is equipped with 2 NVIDIA's GPUs GTX Titan Black, 2880 cores @980 MHz, and 6 GB of RAM and 2 NVIDIA's GPUs GTX K40c, 2880 cores @745 Mhz. Medusa is a server with two Intel Xeon Silver 4208 CPU @ 2.10GHz, with a total of 16 cores con hyperthreading (32 threads) @ 1.4Ghz. It is equipped with 3 NVIDIA's GPUs GTX Titan Black 2880 cores @980 Mhz, and 1 NVIDIA's GPU GTX Titan X 3072 cores @1000 Mhz.

During the MPI contest, we used *medusa* and *hydra* in combination. They are interconnected by a 40Gb Ethernet network fabric.

All machines are managed by a CentOS 7 operating system. The compilers and system software used are GCC v10.2, and CUDA v11.3.

The assignment provides the sequential code and program arguments to be used as test-beds for the students. Other test-beds used by the on-line judge during the contest are also provided.

The results of the contests and other statistical data for the five Peachy assignments in this series are publicly available at http://frontendv.infor.uva.es.