Peachy Parallel Assignments (EduHPC 2018)

Eduard Ayguadé
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
eduard.ayguade@bsc.es

Lluc Alvarez
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
lluc.alvarez@bsc.es

Fabio Banchelli
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
fabio.banchelli@bsc.es

Martin Burtscher
Texas State University
San Marcos, TX, USA
burtscher@txstate.edu

Arturo Gonzalez-Escribano
Universidad de Valladolid
Valladolid, Spain
arturo@infor.uva.es

Julian Gutierrez
Northeastern University
Boston, MA, USA
jgutierrez@ece.neu.edu

David A. Joiner
Kean University
Union, NJ, USA
djoiner@kean.edu

David Kaeli
Northeastern University
Boston, MA, USA
kaeli@ece.neu.edu

Fritz Previlon
Northeastern University
Boston, MA, USA
previlon.f@husky.neu.edu

Eduardo Rodriguez-Gutiez
Universidad de Valladolid
Valladolid, Spain
eduardo@infor.uva.es

David P. Bunde
Knox College
Galesburg, IL, USA
dbunde@knox.edu

***Abstract*—Peachy Parallel Assignments are a resource for instructors teaching parallel and distributed programming. These are high-quality assignments, previously tested in class, that are readily adoptable. This collection of assignments includes implementing a subset of OpenMP using pthreads, creating an animated fractal, image processing using histogram equalization, simulating a storm of high-energy particles, and solving the wave equation in a variety of settings. All of these come with sample assignment sheets and the necessary starter code.**

***Index Terms*—Parallel computing education, High-Performance Computing education, Parallel programming, OpenMP, Pthreads, CUDA, Compiler and runtime systems, Fractals, Image processing, Particle simulation, Wave equation, Peachy Assignments**

I. Introduction

A key part of teaching a course on parallel and distributed computing or computational science is creating great programming assignments. Students likely spend more time working on these assignments than engaging with other aspects of the course, making the assignments integral both to student learning and student perceptions of the subject matter. That said, creating great assignments is a large time commitment and is not guaranteed to succeed; sometimes even seemingly-great assignment ideas turn out to have flaws when implemented and given to students. To help overcome these issues, we are presenting Peachy Parallel Assignments at the Edu* series of workshops. These assignments all go through a competitive review process based on the following criteria:

- Tested — All Peachy Parallel Assignments have been successfully used in a class.
- Adoptable — Peachy Parallel Assignments are easy to adopt. This includes not only the provided materials, but also the content being covered. Ideally, the assignments cover widely-taught concepts using common parallel languages and widely-available hardware, have few prerequisites, and (with variations) are appropriate for different levels of students.
- Cool and Inspirational — Peachy Assignments are fun and inspiring for students. They encourage students to spend time with the relevant concepts. Ideal Peachy Assignments are those that students want to demonstrate to their roommate.

This effort is inspired by the SIGCSE conference's Nifty Assignment sessions, which focus on assignments for introductory computing courses. (See http://nifty.stanford.edu for more details.)

In this paper, we present the following Peachy Parallel Assignments:

- Implement a subset of OpenMP in pthreads to gain a deeper understanding of both libraries.
- Create a movie that zooms into a fractal. Appropriate for MPI, pthreads, OpenMP, or CUDA.
- Implement histograph equalization to sharpen images using CUDA.
- Simulate a storm of high-energy particles using OpenMP, MPI, CUDA, or OpenCL.
- Implement and parallelize code to solve the wave equation in multiple contexts (serial, OpenMP, MPI, and CUDA) to illustrate principles of performance optimization.

See the Peachy Parallel Assignments webpage (https://grid.cs.gsu.edu/~tcpp/curriculum/?q=peachy) for the materials (e.g. assignment handouts and starter code) for these assignments.

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (contracts TIN2015-65316-P and FJCI-2016-30985), and by the Generalitat de Catalunya (contract 2017-SGR-1414).

978-1-7281-0190-3/18/$31.00 ©2018 IEEE
DOI 10.1109/EduHPC.2018.00012

The website also lists Peachy Parallel Assignments presented previously. We hope that you find these assignments useful and encourage you to consider submitting your own great assignments for future presentation!

II. Mini-OpenMP — Ayguadé, Alvarez, Banchelli

This section presents the *"Implementing a minimal OpenMP runtime using Pthreads"* assignment that is offered to students of *Parallel Programming and Architectures (PAP)*, a third-year elective subject in the Bachelor Degree in Informatics Engineering at the Barcelona School of Informatics (FIB) of the Universitat Politècnica de Catalunya (UPC) — BarcelonaTech. In this assignment, students open the black box behind the compilation and execution of OpenMP programs, exploring different alternatives for implementing a minimal OpenMP-compliant runtime library, providing support for both work-sharing and tasking models. In addition, the assignment allows the students to learn the basics of Pthreads in a very interesting and applied way, as well as the low-level atomic mechanisms provided by the architecture to support the required thread and task synchronisations.

A. Motivation

OpenMP is the de-facto standard API for programming shared-memory parallel applications in C/C++ and Fortran. From the programmers' perspective, OpenMP consists of a set of compiler directives, runtime routines and environment variables. This programming model is ideally suited for multicore and shared-memory multi-socket architectures, as it allows programmers to easily specify opportunities for parallel execution in regular (e.g. loop-based) and irregular (e.g. recursively traversing dynamically created data structures) applications.

The two components that support the execution of OpenMP programs are the compiler and the runtime system. The OpenMP compiler transforms code with OpenMP directives into explicitly multithreaded code with calls to the OpenMP runtime library. The OpenMP runtime system provides routines that support thread and task management, work dispatch, thread and task synchronisation, and intrinsic OpenMP functions. Most OpenMP runtimes are implemented using Pthreads (also named POSIX threads), the threads programming interface specified by the IEEE POSIX 1003.1c standard.

B. The `miniomp` Assignment

The assignment is to build a minimal OpenMP runtime system. This simplified runtime system has to support the code generated by the GNU `gcc` compiler and to correctly work as an alternative to the `libgomp` runtime system that is distributed as part of `gcc`. Figure 1 shows a flowchart with the main components involved in the assignment. The `gcc` compiler is used unmodified and students analyse how the compiler translates the different directives to invocations of services in the runtime system. The `libgomp` library is replaced by the minimal runtime library (`libminiomp`) using the `LD_PRELOAD` mechanism, making use of the library constructor and destructor mechanisms to setup the OpenMP environment necessary to execute the OpenMP programs. The implementation of the simplified runtime is based on the POSIX Pthreads standard library, which provides the basic support for thread creation and synchronisation on top of which the work-sharing and tasking models can be built. So, in addition to the internal organization of OpenMP runtime systems, the students also learn the main features of the Pthreads library and they put into practice the use of dynamically linked shared libraries.

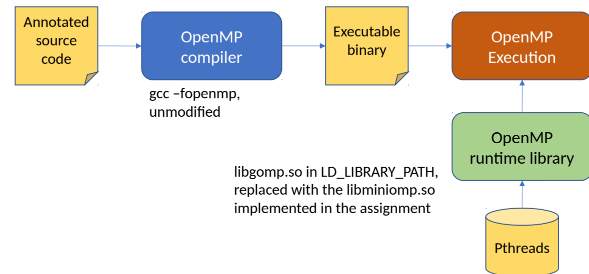

Fig. 1. The OpenMP compilation and execution flow.

In this scenario, our students usually ask the same questions: *"Why do we want to study how an OpenMP program is actually executed on our parallel machine? Is it not enough to be able to write correct OpenMP programs?, ..."*. We believe that this assignment helps students understand program performance and possible sources of overhead so they can write more efficient code. It also teaches the basics of developing a parallel runtime system on top of low-level threading offered by the OS, targeting new programming models and/or multicore architectures.

C. Two Versions

We propose two versions of the assignment that ask students to implement different functionality of the OpenMP runtime:

- **Version 1 — Basic implementation for the work-sharing model**: `parallel` regions, work distributors (`single` and `for`) and synchronisations (`barrier` and `critical`, both unnamed and named).
- **Version 2 — Basic implementation of the tasking model**: unified `parallel-single` region, `task` (with no support for nesting), `taskloop` and synchronizations (`critical` and `taskwait`).

Both versions are developed in an incremental way, so students can always generate a working runtime library with the functionality implemented so far. For example, for version 1 students first implement `parallel` and intrinsic functions `omp_get_thread_num`, `omp_get_num_threads` and `omp_set_num_threads`, learning how to access to environment variables such as `OMP_NUM_THREADS`; second they implement `barrier` and `critical` (both named and unnamed) and take a look at how the compiler implements `atomic`; then they implement the `single` work-sharing construct, paying attention to the fact that there may be multiple instances of `single` active at the same time; and finally they implement the `for` work-sharing construct.

```
miniomp                          miniomp
    src                              test
        Makefile                         Makefile
        env.{c|h}                        run-omp.sh
        intrinsic.{c|h}                  extrae.xml
        libminiomp.{c|h}                 run-extrae.sh
        parallel.{c|h}                   tparallel.c
        synchronization.{c|h}            tsynch.c
        single.{c|h}                     tworkshare.c
        loop.{c|h}                       ttask.c
        task.{c|h}                       ttaskloop.c
        taskloop.c                       tsynchtasks.c
        tasksync.c                       mandelbrot-gui.h
    lib                                  tmandel1.c
        libminiomp.so                    tmandel2.c
    …                                    tnested.c
```

Fig. 2. Directory structure for the miniomp assignment.

Once the selected version is implemented, students are encouraged to implement some optional functionality. For example, in both versions the support for nested parallel regions, implementation of a thread pool (i.e. threads not created and finished in each parallel region) or thread binding policies are not initially considered. Also, students are allowed to initially use the synchronisation constructs provided in Pthreads (locks, barriers, ...). As part of their work, and based on their own skills, interests and time, students decide to relax some of these constraints, make use of atomic instructions to implement synchronisation primitives and/or consider some additional OpenMP features initially not considered in the specification of the assignment.

A number of simple test cases are provided with the aim of testing the functionality as it is incrementally implemented. For functional validation, students can always switch back to the original `libgomp` library to know the output that each test case should produce. We encourage students to extend the basis test suite to improve coverage and better validate the functionality implemented. Students are also motivated to compare, in terms of performance, their implementation with the original `libgomp` library; although unfair, since the original `libgomp` library supports much more functionality, this gives them an idea of what performance should be expected and motivates them to improve their own implementation.

D. Resources and Duration

The proposed assignment can be done in any system booted with Linux, preferably multicore. Although students could use their own laptops, we strongly suggest they use a departamental cluster composed of several nodes, each with a certain number of cores. One of the nodes is used as login node, in which students can edit and compile their code, as well as do some functional tests. The rest of nodes are accessible through execution queues and can be used for performance evaluation/comparison purposes.

In order to give an idea of the number of files that need to be modified, Figure 2 shows the directory tree and files initially provided to students; dummy implementations for all runtime functions that need to be implemented are provided. In terms of duration, the assignment is designed to be done in 6 laboratory sessions (one per week), each taking 2 hours; in addition, students usually spend between 2 and 4 additional hours at home per week. So the total load is estimated between 24 and 36 hours, depending on how far students go with optional parts, functional validation and performance evaluation and comparison against the original `libgomp`.

The assignment has been tested (and evolved) in the past 3 academic years, making it readily adoptable by other educators.

III. Fractal Movie — Burtscher

This project is to parallelize a provided serial program that computes multiple images of a fractal, each at a higher zoom level. The resulting images can be viewed individually or combined into a movie, e.g., with the free convert tool.

The part of the code that needs to be parallelized is short and relatively simple. It consists of three nested loops. The outer loop iterates over the images and the inner two loops iterate over the x and y coordinates. The fractal code that computes a pixel's gray-scale value comprises only six statements but is complex, which is why I explain it in class before handing out the project.

A. Target Audience and Context

I have used this assignment for many semesters in a senior-level undergraduate course on parallel programming as well as in a masters-level graduate course on parallel processing. In both courses, the fractal is part of multiple bi-weekly programming projects. Each project focuses on a different parallelization approach (MPI, Pthreads, OpenMP, and CUDA). In addition, I typically include at least one variation per project.

B. Interesting Variations

In the most basic version, only the outer image-loop is parallelized. However, the project can be made more challenging by including some of the following additions: 1) If the zoom factor is computed iteratively (rather than using a closed-form function), there is a loop-carried dependency that the students must eliminate. 2) If the loop does not execute enough iterations to yield sufficient parallelism (e.g., for GPUs), multiple loops must be combined and parallelized together. 3) If the order of the inner two loops does not match the memory layout of the image, the loops must be swapped to improve locality and enable coalescing. 4) Depending on the selected fractal, there may be substantial load imbalance, which can be alleviated by modifying the schedule.

C. Prerequisites

At a minimum, the students must have been taught basic loop-parallelization strategies. If the above-mentioned variations are used, knowledge about dependency elimination, loop fusion, loop interchange, and scheduling techniques is also required.

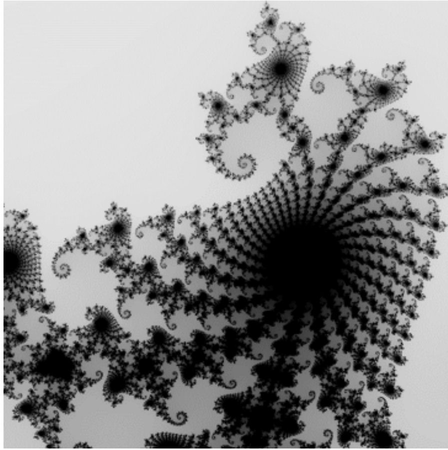

Fig. 3. A frame of the fractal movie

D. Covered Concepts

The covered concepts include basic loop parallelization and load imbalance. Optionally, they include which loop to parallelize (to avoid dependencies or to lower overhead), loop interchange (to enhance locality), loop fusion (to increase parallelism), dependency elimination, and scheduling (to reduce imbalance).

E. Strengths

I have had great success with this assignment because of the following benefits: 1) Most students enjoy creating the fractal image/movie. 2) The code is quite short but not trivial, and the part that needs to be parallelized easily fits on a single screen. Not counting writing out the BMP, there are only about 40 statements, including timing code and code for checking the command-line parameters. 3) The program requires no input files, just the size of the image and the number of images to compute. 4) The output can be viewed image-by-image or converted into a movie, the latter of which is great for demonstrating the result to other people. 5) The images are not only visually pleasing but also useful for debugging, e.g., they typically expose which thread or process is off. 6) The fractal can easily be changed from one semester to the next to modify the runtime and the load imbalance.

Probably the most distinctive feature of this assignment is that the brightness of each pixel is determined by the amount of computation performed. Hence, the fractal is a visualization of the workload and thus enables the students to see the load imbalance. For example, when generating Figure 3 with two threads, there will be substantial load imbalance if one thread computes the less work intensive, brighter top half of the image and the other thread the more work intensive, darker bottom half.

F. Weaknesses

The main downsides of this project are that the computation is not particularly useful, that it is hard to understand (though asking students to parallelize code they do not grasp completely may still be useful as this situation does occur in practice), that the provided BMP-writing code is not pretty because of various quirks of the BMP format (which are hidden in a header file), and that a third-party viewer or movie maker is required.

G. Final Comment

Most students, including underrepresented students, like the fractal very much. For example, some of my students have used it as background for talk slides, web pages, and even personalized credit cards. One student liked it so much that she gave me a jigsaw puzzle of the fractal as a thank-you gift.

IV. Image Processing — Gutierrez, Previlon, Kaeli

Many textbooks rely on classical linear algebra examples to illustrate best practices in parallel programming (e.g., matrix multiplication and vector add). Despite their common use in class, these examples lack the sophistication of a complete application. We have found that students seem to be more motivated to work with imaging processing algorithms, where the student can view the before and after image, visually inspecting the results of their processing.

This assignment focuses on improving the performance of the histogram equalization algorithm applied to an image. Histogram equalization is a popular image processing algorithm used to increase the contrast of an image to better highlight its features. It is a common algorithm used in many scientific applications such as x-ray imaging, thermal imaging, and as a pre-processing task for multiple computer vision/deep learning algorithms.

The following guidelines are used for this assignment:

- All students work on the same project.
- The project can be developed in groups of (at most) 3 students.
- The students need to study the application to understand how the algorithm works.
- They develop an optimized GPU implementation using CUDA, providing the same functionality (but faster) of the histogram equalization found in the OpenCV package.
- Students are encouraged to explore their own unique algorithm if the output result is comparable to that of the OpenCV library (+- 5% difference allowed).
- Students obtain all the points if they have successfully completed the assignment.
- Extra points (or rewards) are given to those who achieved the best-performing implementation.

Students are given a baseline code that works with a CPU-based OpenCV interface, and a simple CUDA kernel which reads the input image and provides the necessary structure to modify the image. This structure allows the students to focus on the algorithm implementation and performance, without having to develop the initial program structure. Additionally, evaluating the final code from students becomes easier if they all follow the same coding structure. A key factor in the effectiveness of this assignment is dedicating a class session

Fig. 4. Example input image and output for the histogram equalization algorithm.

discussing their implementations, sharing the different ways of optimizing their resulting CUDA code with their classmates.

Concepts covered within this assignment include: 1) CUDA programming in C, 2) performance tuning with profiling tools (`nvprof` and `nvvp`), and 3) algorithmic optimizations specific for image processing algorithms (including memory performance improvement through coalescing reads and shared memory, and efficient reduction schemes for histograms).

The assignment is appropriate for students at all academic levels, as long as they have a passing knowledge of CUDA as part of their past coursework (with most CUDA and GPU architecture concepts covered before the assignment). This assignment has been used as a final project for a free GPU programming class offer to undergraduates and graduate students at Northeastern for the past 5 years. A GPU was awarded to the best performing project. Additionally, multiple image processing algorithms can be used as variations for this assignment, such as the Sobel filter.

The strengths of this assignment include:

- Motivational (we received positive feedback from students).
- Challenging, yet doable.
- Encompasses multiple GPU programming optimization concepts.

The weaknesses of this assignment include:

- Oriented specifically toward the CUDA programming language (although the assignment can be adapted to other parallel programming languages).
- Requires CUDA-enabled hardware.

V. Storms of High-Energy Particles — Gonzalez-Escribano, Rodriguez-Gutiez

We present an assignment used in a Parallel Computing course to teach the approaches to the same problem in different parallel programming models. It targets basic concepts of shared-memory programming with OpenMP, distributed-memory programming with MPI, and GPU programming with CUDA or OpenCL. This assignment is based on a highly simplified simulation of the impact of high-energy particle storms in an exposed surface. The idea is inspired by simulations to test the robustness of material used in space vessels. The program is designed from scratch to be easy to understand by students, and to include specific parallel structures and optimization opportunities. A simple parallelization in the three models considered is quite direct. But the program has plenty of opportunities for further improvements, optimizations, and the usage of more sophisticated techniques. It has been successfully used in parallel programming contests during our course, using the performance obtained by the students' code as a measure of success.

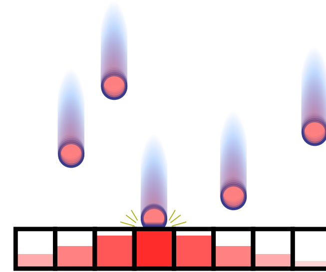

Fig. 5. Graphical representation of the effect of particles impact in a discretized 1-dimensional space

A. Idea and Context

Different programming models use different approaches for the parallelization of basic application structures. Understanding these differences is key for students to get into more advanced techniques, and to face parallel programming on current heterogeneous platforms. We have designed a course on parallel programming that introduces the basic concepts and techniques for programming with OpenMP, MPI, and CUDA or OpenCL.

We use the same simple but inspirational application as an assignment for each programming model. The provided material includes a sequential code, a test-bed of input files, and a handout explaining the assignment. After the lectures and laboratory sessions dedicated to teach the basic concepts and tools for each programming model, we start a programming contest lasting one week. During that week, the students parallelize the code with the target programming model, and compete to obtain the best performance. The students can use common compilers and PC platforms to develop and test their codes. An automatic judging tool with an on-line public ranking is used to provide a fair arena, and to keep the students engaged during the contest. Other gamification tools are also included [1].

B. Concepts Covered

For simplicity, the program simulates the effect of particles impacting on a cross section of the surface. It uses 1-dimensional arrays to represent the discretized space (see Fig. 5). Each storm is represented as an unordered collection of pairs of numbers. Each pair indicates the impact position and the energy value of a particle. For each storm, the program applies three stages: (1) Update the array cells depending on particle energy and distance to the impact point; (2) Compute a relaxation using a stencil operator; and (3) Compute a reduction to obtain the maximum energy value and its position in the array. The output of the program is the list

of maximum values and positions after each storm. In debug mode, the program also writes the final energy values in all array positions in a graphical plain text representation. More sophisticated representations of the evolution of the energy after each storm can also be obtained with simple tools such as `gnuplot`.

The basic concepts covered in OpenMP are parallelization of loops, targeting the outer/inner loop to avoid race conditions, and non-trivial reductions. Loop reordering can also be applied. In the case of MPI, the students need to understand how to compute the limits of a distributed array in terms of the process index, using halos and neighbor communications for the stencil part, and generic reductions. For GPU programming, the students work with the concepts of memory management in the co-processor, reducing host-to-device and device-to-host communication, embarrassingly-parallel kernels, choosing proper thread block sizes, and simple reductions.

Plenty of further code optimizations can be discovered and applied, such as pointer swap to avoid array copies, reverse threshold calculations to narrow loop limits, using wider halos to reduce synchronization stages in MPI, using shared memory in GPUs to implement faster reductions, etc.

VI. Wave Equation – Joiner

Topics for use in a classroom setting for high performance computing suffer from competing challenges of (1) showing reproducible speedup and scalability, (2) having authenticity, and (3) being able to present and complete in a standard classroom period. In CPS 5965 High Performance Computing at Kean University, we use the numerical solution of the wave equation as a motivating example. The problem is used in multiple contexts across different course meetings, and provides an example giving continuity across 4 key topics in my course (performance programming, threaded parallelism, accelerated parallelism, and distributed parallelism).

A. Initial Use, Performance Computing

The problem posed to students is to solve for a solution of the equation

$$\frac{\partial^2}{\partial t^2} A = \nabla^2 A$$

given fixed boundary conditions at the extremes and an initial perturbed (typically Gaussian) configuration between the extremes. Students are provided with an algorithm and pseudocode for solving the problem using a Leapfrog method. They are asked to implement the pseudocode, which includes initial conditions and sample output, in their language of choice. This allows us to begin one of the first key discussions in the course, choice of language. Most students entering in the course have backgrounds in languages other than C, C++, or Fortran, and will typically write their code in Java, Python, or MATLAB. The student implementations are compared to each other, as well as solved examples in those languages plus C, C++, Fortran, JavaScript, C#, and PERL. For languages with an available optimizing compiler, a range of optimization options are used. We also use commercial compilers if available. The wall time of the solutions are sorted by language given a simple student-created benchmark of available languages for HPC use. Results can vary by platform and installation, but generally students will see significant performance differences between interpreted languages and compiled languages, as well as a significant performance difference between architecture-specific optimizing compilers and platform-independent non-optimizing compilers (e.g. PERL, Python, MATLAB typically perform slower than C#, Java, and JavaScript, which in turn typically perform slower than C, C++, and Fortran). This helps to motivate the use of C/C++ as the language in which my course is taught, and leads into subsequent discussions of performance tuning, particularly the importance of optimizing for sequential access through contiguous memory. Other elements that can be worked into this activity is the use and meaning of the Unix `time` command, as the timing of each version of the code is shown to the students in class. Students are not given the C solution at this time, but rather shown it in demonstration. The next lesson that follows is a primer in the C language, as typically the students in the class have not previously had experience with it. As a homework assignment, students are required to write their own version of the wave code in C.

B. Reprise of Activity, Parallelization

We return to the wave equation throughout the class when discussing different types of parallelism. Our first parallel unit covers threaded parallelism in OpenMP. After an initial lecture where students are shown typical demonstrations of OpenMP (e.g. trapezoidal rule parallelization from Pacheco), students are asked to use OpenMP in their C version of the wave equation code. In class, students discuss different options for choices of loops to parallelize, and scheduling options to be used. The implementation is done as a lab activity in class, and students who do not complete it in the alloted time are assigned it as homework. Students are asked to compare speedups for increasingly higher resolutions of their wave, and to determine how large their wave array needs to be before any performance improvement is seen. Students typically will quickly see that the size of the array required to show speedup for the 1D solution is substantially greater than needed for an accurate solution to the given problem, and also that by making the problem higher resolution than needed, the wall time for the parallel implementation is greater than the simpler, serial approach that they started from. The follow-up activity is the students being given starter code for a 3D version of the same problem, which they are required to parallelize and profile. They typically will determine that the greater amount of work required to solve the 3D version of the problem make this version more suitable for a parallel solution.

For the MPI unit, the wave equation is used as an example of a domain decomposition, as block scheduling of the work in updating the arrays requires that each rank be updated with a ghost cell of the data from a nearby rank, in order to calculate $\nabla^2 A$. Best practices in determining block scheduling in MPI

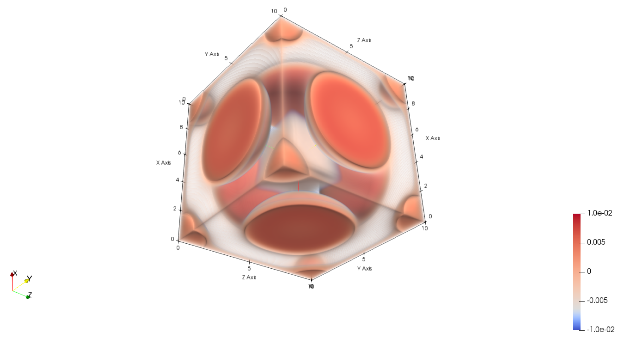

Fig. 6. Paraview visualization of a time step in the 3D wave solution.

are discussed in class, and code for computing ranges of blocks that are as evenly spaced as possible is provided in an earlier activity. Students then are asked to implement block sceduling with ghost cells in MPI. Solved code, including the Block-Range structure provided to students, is available to instructors by request to djoiner@kean.edu. Additionally, during the MPI unit the benefit of structuring multi-dimensional arrays as single flat arrays with stride determined during array access (e.g. x[i*ny+j] instead of x[i][j]) is discussed, and starter code for the students includes a rewrite of the wave3d code with flattened arrays.

The wave code is also used as a lab activity for CUDA parallelization.

C. *Visualization of Results*

Throughout the semester, in discussions of this and other activities, code examples are often used to showcase best practices in other areas of high performance computing other than parallel programming. Visualization, in particular, gets discussed in class, including different programs for graphing results and for creating graphics during a run for rapid visualization and debugging. 1D solved examples in C provided with this paper include simple line graphing using GD, 2D solved examples in C include a heatmap in GD, and 3D solved examples include output in CSV and NetCDF formats that can be viewed post-run using visualization tools such as ParaView (shown in Figure 6).

References

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

Appendix: Reproducibility

It is certainly our hope that others will use these assignments and reproduce the success we have had with them. The main materials needed for this are the assignments and supporting materials available on the Peachy Parallel Assignments webpage, https://grid.cs.gsu.edu/~tcpp/curriculum/?q=peachy. The bulk of the resources and context for each assignment is presented in the section on that assignment. Where the authors thought that additional information on the resources or staging of an assignment would be useful, it is presented below.

A. *Storms of High-Energy Particles*

The particle storm 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).

The material of the assigment, including a handout, the starting sequential code, and some input data sets to be used as examples are publicly available (https://trasgo.infor.uva.es/peachy-assignments/).

The on-line judge program used in the programming contests is named *Tablon*, and it was developed by the Trasgo research group at the University of Valladolid (https://trasgo.infor.uva.es/tablon/). The contest software uses the Slurm queue management software to interact with the machines in the cluster of our research group. During the course we used Slurm 17.02.7.

The machine of the cluster used for the OpenMP and CUDA/OpenCL contests is named *hydra*. It 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 4 NVIDIA's GPUs (CUDA 3.5), GTX Titan Black, 2880 cores @980 MHz, and 6 GB of RAM.

During the MPI contest we use *hydra* in combination with two other servers to create a heterogeneous cluster. The other two machines are: *thunderbird*, with an Intel i5-3330 @2.4 GHz CPU and 8 GB of RAM; and *phoenix*, with and Intel QCore @2.4 Ghz CPU with 6 GB of RAM.

All machines are managed by a CentOS 7 operating system. The compilers and system software used have been GCC v6.2, and CUDA v9.0.

The assignment provides the sequential code and the input files of the 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 are publicly available until the start of the next semester at http://frontendv.infor.uva.es.

B. *Wave Equation*

The wave examples shown are used in CPS 5965 at Kean, which is a semester long course on high performance computing. Classes run in a double period format over 2.75 hours with a 15 minute break, over the course of 16 weeks. The class is set up with roughly half of the time devoted to a combination of lecture and interactive demonstrations, for which the students are expected to follow along on their own laptops, and half for lab-based time where students work independently and with each other while the instructor is available to assist. Examples are run on one of two machines available to the students. Having the students create the code in their language of choice is implemented as one lab activity. The comparison between languages is done during the lecture portion of the following class, along with a primer in C for students with prior programming knowledge. The lab activity following the C primer is for the students to translate their code into C. Each

of the other uses of the example are done as lab activities during the OpenMP, CUDA, and MPI units in the course. The serial codes as well as OpenMP and MPI versions of codes are run on a 130-node cluster housed on campus, available to students throughout the run of the course, however all of the examples can be configured to run in different amounts of time by either changing the resolution or end time of the simulation, and thus can run on a variety of hardware options. Serial codes for benchmarking are run on a single core of a single node, each of which has 8 2.6GHz CPU cores available, with 16 GB RAM and 300 GB storage per node, connected via Gigabit ethernet. For C/C++ and Fortran, students have both Gnu and Intel compilers available. Scheduling is done using Torque/Maui for all serial, OpenMP, and MPI examples. Students run OpenMP examples with dedicated use of all cores on a single node, and students can request all available nodes for MPI examples, with a subset of nodes set aside for short classroom jobs. For CUDA-based examples, students have access to a shared server with 2 K20c Tesla GPGPUs.