Peachy Parallel Assignments (EduPar 2024)

Alina Lazar*, Ethan Scheelk†, Elizabeth Shoop‡, David P. Bunde§

*Youngstown State University, USA, alazar@ysu.edu, 0000-0002-2096-1541
†Macalester College, USA, escheelk@macalester.edu, 0009-0008-9034-5358
‡Macalester College, USA, shoop@macalester.edu, 0009-0003-2871-8049
§Knox College, USA, dbunde@knox.edu, 0000-0001-6334-356X

***Abstract*—We present two new assignments in the Peachy Parallel Assignments series of assignments for teaching parallel and distributed computing. Submitted assignments must have been successfully used previously and are selected for being easy for other instructors to adopt and for being "cool and inspirational" so that students spend time on them and talk about them with others.**

The first assignment in this paper familiarizes students with the RAFT library for performing GPU-accelerated computation, part of the RAPIDS AI ecosystem. Students use this library to accelerate a Radius Nearest Neighbor computation, finding all points within a given distance from a query point. In the second assignment, students parallelize a bird flocking simulation using OpenMP or OpenACC. It is a visual assignment which allows students to readily see the performance improvement.

***Keywords*-Peachy Parallel Assignments, Parallel computing education, Parallel programming, Curriculum development, Visualizing parallelism, RAPIDS AI ecosystem, Radius Nearest Neighbor, Boid flocking algorithm**

I. Introduction

Designing engaging, high-quality assignments requires creativity and hard work. New assignments also carry risk because they don't always run as smoothly as expected. At the same time, the importance of course assignments is hard to overstate. This is where students apply what they are learning in class. In many cases, it is how they spend the majority of the time they devote to the course. Assignments can also feature prominently in students' memory of the course and sometimes even affect the reputation of a course or department when students talk about their experiences. The Peachy Parallel Assignments project aims to spread the benefits of assignment-creation work by publishing high-quality assignments, allowing others to leverage this work by adopting the assignments and also giving assignment creators public recognition for their ideas.

With these goals in mind, Peachy Parallel Assignments are solicited in the Call for Papers of the Edu* workshops (EduPar and EduHPC). Submissions are selected competitively using the following criteria:

- Tested: All assignments must have been successfully used with real students. Class testing reduces the risk of an assignment "going sideways" on an adopting instructor.
- Adoptable: The assignments must be useful to other instructors, with clear descriptions and the resources needed for adoption by others (handouts, given code, references for more information, etc.). Ideally, they focus on core PDC topics using widely-used languages and toolsets, with suggested customizations that can make the assignments suitable for students at a variety of levels. Thus, adopters have information on the assignment as it was previously used and as much guidance as possible to help them adapt it to other contexts. This information is also potentially useful as a source of "lessons learned" to others interested in creating assignments.
- Cool and inspirational: The assignments must motivate students through the artifacts they create (e.g., images) or the concepts taught. Ideally, students should be excited about the assignment and want to talk about it with friends, improving perception of the course, department, and field as a whole.

Once selected, writeups of the assignments are published in the workshop proceedings (e.g., [1], [2], [3]) and all the assignment materials are archived on the Peachy Parallel Assignments webpage (https://tcpp.cs.gsu.edu/curriculum/?q=peachy).

This paper describes the two Peachy Parallel Assignments selected for EduPar 2024. The first assignment asks students to use a library to GPU accelerate the Radius Nearest Neighbor problem. This problem has applications in machine learning and computational geometry. By using the library, the assignment is made more accessible to less experienced students or those outside the CS major. The second assignment has students parallelize a simulation of flocking behavior. This is a graphical application that produces interesting behavior using only very simple rules.

II. Fixed-Radius Nearest Neighbor Search

The rapid growth of data in various fields like geospatial analysis, bioinformatics, and social network analysis requires efficient, fast processing techniques. Nearest Neighbor (NN) search is a fundamental algorithm in these areas, and understanding its implementation is crucial for students aiming to work with large-scale data. NN search is generally classified into two categories: k-nearest neighbor search and fixed-radius neighbor search. The latter, also known as radius query, focuses on locating all data points that fall within a specified distance from a query point and does not require sorting the results. The simplest method to implement NN search is through a linear scan of the entire database, commonly named

exhaustive or brute force search. An NN search assignment using MapReduce was presented here [1]. The brute force approach remains prevalent, often used with GPU acceleration [4]. However, teaching GPU programming, especially C++ CUDA to students is a real challenge.

More generally, the widespread availability of multi-core hardware platforms and increasing need for high-performance computing capabilities across diverse application areas, presents a unique challenge for computer science educators, particularly those who teach courses in parallel and distributed computing (PDC) since a broader group of students is interested in this material. One approach to making PDC more accessible is through the use of libraries. RAPIDS AI [5] is a suite of open-source software libraries that brings GPU acceleration to data science, making tasks faster and more efficient. RAFT, as part of RAPIDS AI, provides advanced analytics and machine learning primitives implemented in both Python and C++. Exposing students to these tools prepares them for cutting-edge data science and machine learning developments. Implementing FRNN using RAFT will eliminate the overhead of learning low-level CUDA programming, allowing students to bridge theoretical knowledge of algorithms and data structures with practical, real-world data processing challenges. This hands-on approach reinforces learning and enhances problem-solving skills.

The proposed assignment involves not just an implementation but also performance analysis, teaching students to evaluate and optimize the efficiency of their code. This skill is vital for developing high-performance applications in any computing discipline. Understanding how to scale algorithms and make them more efficient is a crucial part of modern computing. This assignment introduces students to these concepts concretely, showing how algorithms can be adapted to handle larger datasets more efficiently. The assignment encourages students to innovate and experiment with different approaches to implementing FRNN, fostering creativity and critical thinking skills that are highly valued in technology fields.

A. The FRNN Assignment

This assignment aims to familiarize students with the RAPIDS AI ecosystem, particularly the RAFT library, for performing GPU-accelerated computations. Students will use RAFT to implement a Fixed Radius Nearest Neighbor (FRNN) search algorithm and analyze its performance on a given dataset. The starting point of this assignment is a brute force pseudocode, which implements a NN search. This assignment lets students choose between C++ and Python.

The instructor provides synthetic generated and real datasets of several sizes. Students will write code to load the dataset to be used for the FRNN search. They will need to make sure the data is in a format compatible with RAFT's requirements (e.g., RAFT objects or CuPy arrays).

To implement the FRNN search, the students must initialize the RAFT environment and necessary data structures before implementing the FRNN search algorithm using RAFT's nearest neighbor functionality. RAFT has a brute-force implementation, index implementations for approximate nearest neighbor search, and a graph-based nearest neighbor search implementation [6] with state-of-the-art query performance. The next step is to perform experiments that measure and record the execution time of the different FRNN search implementations. It is also possible to compare the GPU-based performance with a CPU-based implementation of FRNN search.

In addition to Python scripts or C++ code containing the FRNN search implementation, students must submit a report on their findings. They analyze the results obtained from the FRNN search and discuss the performance improvements observed with GPU acceleration. They are also asked to highlight any challenges faced during the implementation and how they were resolved. The instructor evaluates the submissions based on a rubric defined using the following criteria:

- Correct implementation of the FRNN search using RAFT.
- Effective use of acceleration and memory management.
- Clarity and efficiency of the code.
- Depth of analysis in the report, including discussions on performance and any challenges encountered.

B. Setup and Learning Outcomes

Before assigning this example as homework or in-class lab, the instructor should review fundamental concepts such as modern CPU and GPU architectures and how performance improvements are measured to help students evaluate the effectiveness of using GPUs. As technical requirements, the students should have access to a computer with a GPU, have Miniconda installed, and must be able to install RapidsAI in a conda environment. Alternative possible resources include Google Colab, which provides environments with up to 2 cores, or access to a supercomputer. For example, students and instructors in the state of Ohio have access to the Ohio Supercomputing Center [7].

In terms of learning outcomes, after completing this assignment students should be able to:

- Gain a thorough understanding of both the NN and the FRNN algorithms, including their significance, applications, and the computational challenges they address in handling large datasets.
- Develop an understanding of GPU computing's principles, including parallel processing, memory management, and the advantages of using GPUs for specific types of computational tasks.
- Acquire hands-on experience with RAPIDS AI and RAFT, understanding how to leverage these libraries for efficient data science and machine learning tasks on GPUs.
- Learn to efficiently handle, process, and manipulate large datasets using GPU-accelerated tools, which is a critical skill in big data analysis and processing.
- Develop skills in optimizing code for better performance on GPUs, including understanding how to measure, an-

alyze, and improve the execution time and resource utilization

The educational goals achieved through this assignment align with the learning objectives outlined in the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing, specifically the Core Topics for Undergraduates as referenced in Prasad et al., 2011 [8]. These objectives are particularly relevant for CS2, Data Structures and Algorithms, and Parallel Computing courses.

The main advantages of this assignment are that students can acquire crucial skills in parallel computing, an area gaining importance across various domains like scientific computing, machine learning, and data science. Furthermore, the implementations of the Nearest Neighbor (NN) search offer a concrete demonstration of how parallel computing can enhance both performance and efficiency, thereby encouraging students to explore other similar applications.

C. Conclusion

The assignment on implementing Fixed Radius Nearest Neighbor (FRNN) search using RAPIDS AI RAFT offers a practical introduction to parallel and distributed computing for computer science students. This hands-on project, requiring a fundamental understanding of Python or C++ and access to a multi-core system, provides vital experience in handling large datasets with GPU-accelerated tools. It reinforces the learning of parallel computing concepts and prepares students for the computational challenges in modern data-driven fields. This assignment extends beyond theory, enabling students to design parallel code and understand its applications in various high-computation scenarios, such as point clouds in computer vision and molecular simulations.

III. Visualizing parallelism with boid flocking

Our second assignment focuses on parallelizing for loops based on the classic work of Reynolds from 1987 that introduced autonomous agents called "boids" that model flocking behavior in nature. Using available sequential code versions that display the boids on an X graphics display or a window created from the thread-safe graphics library (TSGL), students can create one of several different parallel versions using OpenMP and OpenACC and visualize the parallelism and speedup in action.

A. Idea and Resources Available

In 1987, Craig Reynolds wrote a paper describing how to model the behavior of flocks of animals for computer graphics. He attempted to simulate birds, coining the graphic version he created bird-oids, or "boids" [9]. He pointed out that the behavior he modeled also applied to schools of fish and herds of other animals.

This groundbreaking work by Reynolds ushered in the study of individual agents that exhibit self-organizing behavior and has been cited well over 14,000 times. One such citation is in a book by Gary Flake called *The computational beauty of nature* [10]. In chapter 16, section 3, Flake describes his implementation of Reynolds' algorithm. We started with Flake's original code when formulating this assignment, but made fairly extensive changes to modernize it.

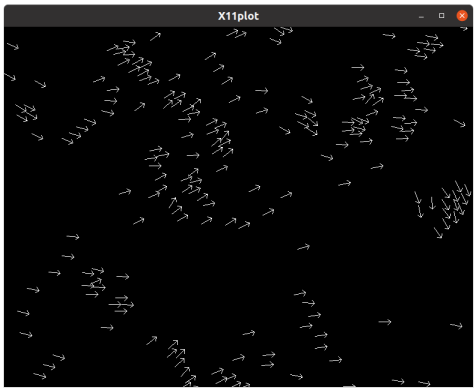

Fig. 1. Display of flocking boids after 1000 steps using 200 boids

In the program, a given number of boids are placed randomly on a canvas, each with a x and y position and initial velocity. Then they begin moving according to some rules. Flake used three 'rules' from Reynold's work to apply to each boid as it moved and added a fourth rule of his own. The movement takes place by simulating time using a for-loop for an arbitrary number of times: each time through the loop, each boid applies the rules by observing the location and speed of every other boid. According to Flake [10, pg. 272], the rules are the following:

1) "Avoidance. Move away from boids that are too close, so as to reduce the chance of in-air collisions.
2) Copy. Fly in the general direction that the flock is moving by averaging the other boids' velocities and directions.
3) Center. Minimize exposure to the flock's exterior by moving toward the perceived center of the flock.
4) View. Move laterally away from any boid that blocks the view."

Flake's original code creates a simple X window visualization that updates at each time step; the display for one run of the program after 1,000 time steps using 512 boids (as arrows) is shown in Figure 1.

This program's sequential main code file is reasonably short (approx. 500 lines including comments) and accessible, with the details of the display separated into extra files that students can treat as a black box. The code has several loops that students would need to study to decide where to place an OpenMP pragma to parallelize it, yet it's possible to determine the proper place to do this, given some preliminary activities using OpenMP. So one version of the assignment can be done in a course that uses C/C++ earlier in the curriculum, such as a systems course, provided students have access to a Linux machine.

The example also lends itself well to exploring more advanced methods of parallelization and alternate visualization. We have used it in an advanced PDC course as project where

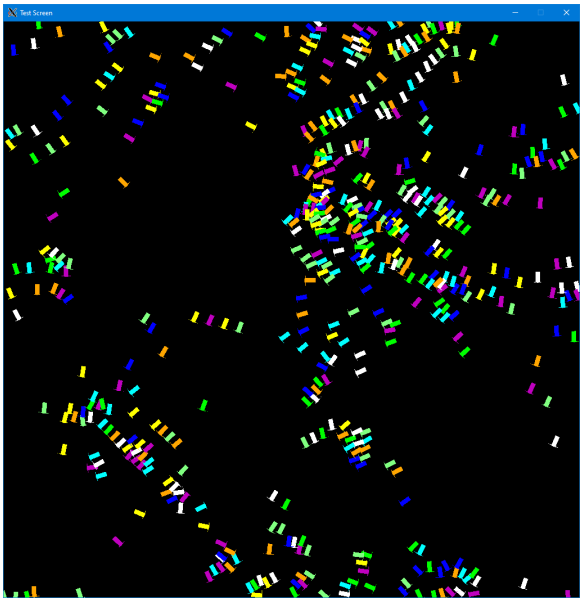

Fig. 2. The TSGL display when using 8 threads

students explore both OpenMP and OpenACC multicore and GPU solutions. We have also successfully ported the code to C++ and replaced the original X visualization with visualization using the thread-safe graphics library (TSGL) [11]. This enables students to see what threads are computing the next heading of each boid by coloring the arrows based on thread number (see Figure 2).

We supply 3 code versions on our GitHub repository (https://github.com/csinparallel/PDCAssignments/tree/main/Boids) [12]:

1) An updated version of the C code from Flake with X display, modernized to remove global variables and avoid compiler warnings. This is ready to apply OpenMP with a Makefile that includes the necessary flags for compiling it for OpenMP. It already has arguments for setting the number threads and timing the code built in.
2) A sequential version similar to 1, but with a Makefile for building a GPU version using OpenACC.
3) A new C++ implementation that uses TSGL libraries for the display. In this version, intended for advanced students, we have a single Makefile and some hints for how to use it to create 3 executables: an OpenMP version, an OpenACC version for multicore, and an OpenACC version for the GPU.

We suggest the following ways that these could be used with assignment documents that we supply:

- Start with 1 and use an assignment we have written as a guide for use in a sophomore-level course.
- Start with 2 and use it as a long assignment or project in a PDC course that introduces OpenACC. We have an example assignment for this.
- Start with 3 and use it in various contexts, adding scaffolding as you see fit in less advanced courses.

In any case, the primary positive aspect of these assignments is that the visualization is not only fun for students, it enables them to determine if their solution appears to be working and to visualize the speedup as they use more threads. When they want to time their working solution, they can choose to eliminate the display with a command line argument.

B. Concepts Covered

The primary concept of this assignment is applying parallelization to for loops in a somewhat complex program. At its core, the updates at each time step for each boid fit naturally into parallel techniques because the computations of a new headings (position and velocity) are independent.

The challenge for students is to study the code and to determine where the parallelization should take place, because the code contains multiple nested loops. This is a skill that the assignment enables the students to practice. Within those loops, students need to determine whether there are any dependencies on any of the variables. This can take some effort, because there are quite a few variables in the code that computes the new headings. Once they study the code, however, students will find there are no dependencies as long as they ensure that each thread uses private and shared variables appropriately.

C. Using the assignment

Author Shoop has used this assignment in a PDC course in a liberal arts college for several years. After first using the original Flake code, she realized that it had some problems, mainly through its extensive use of global variables. So she created a new version that eliminated this and made minor fixes to avoid compiler buffer overflow warnings in the X display code. This version was the basis for several successful student projects that emphasize not only the parallelization improvements, but also study the scalability of the program by conducting experiments and measuring its strong and weak scalability under various conditions. Students read the original paper [9] and the chapter in [10] for inspiration.

Though she hasn't used the OpenMP/X version in a sophomore level course yet, she has observed that students easily understand where to place the pragmas with just a bit of guidance and discussion before using it. With the assignment we provide, it should be possible to use it at this level.

The TSGL version of the code was recently completed by author Scheelk as part of a project for the PDC course. He completely re-wrote the code for C++, which is needed to incorporate the TSGL library for visualization. His work has thus created a new version that Shoop intends to use going forward when offering this course. It is flexible enough to have students try three versions: OpenMP or OpenACC with either multicore or GPU. We supply an assignment for this purpose.

To add more difficulty, an addition is to suggest to students that parallel random number generation could be added when updating the headings. Flake had this feature in the code, but we have commented it out to remove that complexity.

Prerequisites: Students need to have studied simple examples of OpenMP pragmas, especially the for-loop pattern. They also need to be able to read C code and use Makefiles.

Technical considerations: We have instructor notes on our GitHub site for the following: 1) Students will need the ability to display X remotely if you use a server. 2) Installing TSGL under WSL or Linux is needed, but feasible. 3) For OpenACC, installing NVDIA HPC SDK is necessary.

REFERENCES

[1] H. M. Bücker, J. Corrado, D. Fedorin, D. García-Álvarez, A. Gonzalez-Escribano, J. Li, M. Pantoja, E. Pautsch, M. Plesske, M. Ponce *et al.*, "Peachy parallel assignments (EduHPC 2023)," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 366–373.

[2] A. Lazar, V. Niculescu, and D. Bunde, "Peachy parallel assignments (EduPar 2023)," in *Proc. 12th NSF/TCPP workshop on parallel and distributed computing education (EduPar)*, 2023.

[3] R. Carratalá-Sáez, A. Gonzalez-Escribano, A.-S. Iliopoulos, C. Leiserson, C. Park, I. Rosa, T. Schardl, Y. Torres, and D. Bunde, "Peachy parallel assignments (EduHPC 2022)," in *Proc. Workshop on Education for High-Performance Computing (EduHPC)*, 2022.

[4] P. Leite, J. M. Teixeira, T. Farias, B. Reis, V. Teichrieb, and J. Kelner, "Nearest neighbor searches on the GPU: A massively parallel approach for dynamic point clouds," *International Journal of Parallel Programming*, vol. 40, pp. 313–330, 2012.

[5] T. Hricik, D. Bader, and O. Green, "Using RAPIDS AI to accelerate graph data science workflows," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–4.

[6] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang, "Cagra: Highly parallel graph construction and approximate nearest neighbor search for GPUs," *arXiv preprint arXiv:2308.15136*, 2023.

[7] Ohio Supercomputer Center, "Ohio supercomputer center," 1987. [Online]. Available: http://osc.edu/ark:/19495/f5s1ph73

[8] S. K. Prasad, A. Chtchelkanova, S. Das, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. LeBlanc, M. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, and J. Wu, "NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing: core topics for undergraduates," in *Proceedings of the 42nd ACM technical symposium on Computer science education*, ser. SIGCSE '11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 617–618.

[9] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987, pp. 25–34.

[10] G. W. Flake, *The computational beauty of nature: Computer explorations of fractals, chaos, complex systems, and adaptation*. MIT press, 2000.

[11] J. C. Adams, P. A. Crain, C. P. Dilley, C. D. Hazlett, E. R. Koning, S. M. Nelesen, J. B. Unger, and M. B. V. Stel, "TSGL: A tool for visualizing multithreaded behavior," *Journal of Parallel and Distributed Computing*, vol. 118, pp. 233–246, 2018.

[12] "CSinParallel Project Assignments Repository," https://github.com/csinparallel/PDCAssignments/tree/main/Boids, accessed: 01-24-2024.