Peachy Parallel Assignments (EduHPC 2023)

H. Martin Bücker
Friedrich Schiller University Jena
Jena, Germany
martin.buecker@uni-jena.de

Jeremiah Corrado
Hewlett Packard Enterprise
Houston, USA
jeremiah.corrado@hpe.com

Daniel Fedorin
Hewlett Packard Enterprise
Houston, USA
daniel.fedorin@hpe.com

Diego García-Álvarez
Universidad de Valladolid
Valladolid, Spain
dieggar@infor.uva.es

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

John Li
University of California San Diego
San Diego, USA
jzl011@ucsd.edu

Maria Pantoja
CalPoly
San Luis Obispo, USA
mpanto01@calpoly.edu

Erik Pautsch
Loyola University Chicago
Chicago, USA
epautsch@luc.edu

Marieke Plesske
Friedrich Schiller University Jena
Jena, Germany
marieke.plesske@uni-jena.de

Marcelo Ponce
University of Toronto Scarborough
Toronto, Canada
m.ponce@utoronto.ca

Silvio Rizzi
Argonne National Lab
Chicago, USA
srizzi@anl.gov

Erik Saule
University of North Carolina
Charlotte
Charlotte, USA
esaule@uncc.edu

Johannes Schoder
Friedrich Schiller University Jena
Jena, Germany
johannes.schoder@uni-jena.de

George K. Thiruvathukal
Loyola University Chicago
Chicago, USA
gthiruvathukal@luc.edu

Ramses van Zon
SciNet HPC Consortium
University of Toronto
Toronto, Canada
rzon@scinet.utoronto.ca

Wolf Weber
Friedrich Schiller University Jena
Jena, Germany
wolf.wilhelm.stephan.klaus.weber@uni-jena.de

David P. Bunde
Knox College
Galesburg, 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 six new assignments. Students completing them will use *k*-Nearest Neighbor for classification, cluster using *k*-means, implement a data science pipeline of their choice, model traffic jams, apply parallel language features to solve the heat equation, and speed up a machine learning classification system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC-W 2023, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3625541

CCS CONCEPTS

• **Applied computing → Education**.

KEYWORDS

Peachy Parallel Assignment, Parallel computing education, Parallel programming, Distributed processing, k-Nearest Neighbor, k-means clustering, Data sciende pipeline, Nagel-Schreckenberg traffic model, reproducibility, random numbers, Chapel programming language, heat equation, ensemble classification, Spark, MapReduce, MapReduce MPI

ACM Reference Format:
H. Martin Bücker, Jeremiah Corrado, Daniel Fedorin, Diego García-Álvarez, Arturo Gonzalez-Escribano, John Li, Maria Pantoja, Erik Pautsch, Marieke Plesske, Marcelo Ponce, Silvio Rizzi, Erik Saule, Johannes Schoder, George K. Thiruvathukal, Ramses van Zon, Wolf Weber, and David P. Bunde. 2023.

Peachy Parallel Assignments (EduHPC 2023). In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3624062.3625541

1 INTRODUCTION

Peachy Parallel Assignments are a series of high-quality assignments in Parallel and Distributed Computing (PDC). The goal is to provide instructors with pre-tested assignments that will motivate their students while reinforcing important concepts. Publicizing and spreading these assignments recognizes for the effort instructors devote to creating them and saves time for those adopting them.

Assignments are solicited via a public call for submissions and presented at the EduHPC and EduPar workshops. Selection is competitive based on the following criteria:

- Tested: They have been successfully used with real students
- Adoptable: They are useful to other instructors, with clear descriptions and the resources needed for adoption.
- Cool and Inspirational: They motivate students through the artifacts created (e.g., images) or the concepts taught.

The selected assignments are archived on the Peachy Parallel Assignments webpage (https://tcpp.cs.gsu.edu/curriculum/?q=peachy).

This paper describes 6 assignments selected as Peachy Parallel Assignments. Due to space limitations, the assignment descriptions have been abbreviated, but full versions are available online [4, 20].

2 K-NEAREST NEIGHBOR

We begin with an assignment to classify objects based on a database of preclassified objects using the k-Nearest Neighbors method. It takes advantage of the increased popularity of data science and machine learning. For a large database, the algorithm runs slowly, making it natural to consider parallel computing. This assignment uses Map Reduce MPI. It was assigned in an upper-division parallel computing class, but can be adapted for CS1/CS2 or Data Structures.

Problem Description. Students are asked to implement a classifier on a parallel system. The problem assumes that n objects in a database are represented as d-dimensional points and that each has been assigned a class. The goal is to classify each of q query points.

The k-Nearest Neighbor (kNN) method assumes that nearby points in d-dimensional space likely belong to the same class. For each query point, the algorithm finds the k database points closest to it and guesses that it belongs to the class most often represented in those points.

The direct implementation of that algorithm computes $\Theta(nq)$ Euclidean distances in d dimensions for a cost of $\Theta(nqd)$. For each query, identifying the k nearest neighbors by sorting costs $\Theta(n \log n)$, but a heap-based implementation [8] reduces this to $\Theta(n \log k)$, making the overall complexity $\Theta(qn(d + \log k))$.

This algorithm takes significant time even on relatively small instances; a 40-dimensional test case with 5,000 database points and 5,000 queries takes about 5 seconds sequentially. This easily motivates the use of parallelism.

Datasets suitable for this assignment are online; https://datahub.io/machine-learning has 91 instances for classification problems, from leaf identification to detecting forged bank notes. The variety of applications helps show the problem's relevance.

Usage. This assignment was deployed in an undergraduate Parallel and Distributed Computing course at UNC Charlotte (ITCS 3145, described in [17]) and in a similar course for Masters students (ITCS 5145). k-Nearest Neighbor was used to highlight the features of MapReduce, a programming paradigm for BigData contexts [9].

The classes are taught in C++ so we do not use Hadoop's MapReduce framework. Instead, we use MapReduce MPI [15], a C++ library that sits atop MPI and provides MapReduce functionality. This assignment came late in the term: it was the last assignment for undergraduates and the last distributed memory assignment for graduate students. For both, it followed multiple MPI assignments.

Assignment materials are at https://webpages.uncc.edu/esaule/classes/2019_08_ITCS3145/assignment-mrmpi.tgz. These include a classic problem, Word Counting, to familiarize the students with programming using MapReduce MPI. Students are given a sequential implementation of k-NN. Their task is to adapt the code using MapReduce MPI and obtain speedup. Solutions available by request.

In a typical implementation, all processes load the query set since it is assumed not to be large. Then the database file is parsed in parallel by multiple map tasks which compute distances and generate (key:query, value:(distance, class)) pairs. Then a reduction phase takes the pairs for each query, extracts the nearest neighbors' classes, and generates (key:query, value:predicted_class) pairs.

This assignment uses many concepts. It highlights MapReduce and its use to solve BigData problems. It also demonstrates parallel IO since multiple MPI ranks perform IO in MapReduce MPI. MapReduce is a case of load balancing through hashing. It also shows how architectural knowledge can help design faster code since adding local reductions at each rank and again at each multicore node noticeably improves the communication cost.

Assessment. In two semesters of ITCS 3145 and one semester of ITCS 5145, students who submitted this assignment performed reasonably well, but the assignment came at the end of the semester and many students skipped it, knowing they would pass anyway.

One challenge for students was that they needed time to understand how the problem works since they didn't write the sequential code, They also found the API for MapReduce MPI confusing, mostly because it relies on C-like interfaces, but also because the map functions have many complex parameters.

Most graduate students were excited to work on a classic data mining kernel, but some undergraduates did not fully embrace the application because the vector space mapping is already done and they only work with the data mining kernel. To address this, we suggest modifying the assignment so they see a full application.

Adapting the Assignment. The assignment could be adapted to shared memory programming models like OpenMP, other distributed memory programming models like MPI, or accelerator programming models like CUDA.

It could also be adapted to an early programming course. The new assignment would be to write the whole application: parsing the database and queries from a CSV file, implement the distance function with a loop and use the language's built-in sorting function

to find a query point's nearest neighbors. To add parallel computing, have the program find neighbors for each query independently.

For Data Structures, the assignment could focus on space partitioning trees like quad-trees. These can accelerate spatial search [19]; for a "box" of the search space, compute a lower bound on the distance from its points to a query point and decide whether to examine any point in the box. More challenging would be to build the tree in parallel.

3 *K*-MEANS CLUSTERING

Our second assignment is based on the well-known *K*-means clustering algorithm. It is notable as the 6th assignment in a series of Peachy Parallel Assignments [2, 3, 5–7] that ask students to solve one problem on shared-memory using OpenMP, distributed-memory with MPI, and also GPUs using CUDA (or OpenCL). The supporting material for all assignments in this series is available at https://trasgo.infor.uva.es/peachy-assignments/.

The idea behind this series is that different programming models use different approaches to parallelize applications and that students must understand these variations to effectively tackle parallel programming on modern heterogeneous platforms. The assignments have been used in an elective Parallel Programming course for third-year Computer Engineering students at the University of Valladolid. Implementing each assignment in different programming models not only teaches each model, but also helps students focus on their similarities and differences.

This assignment is simpler than the others in this series, focusing mainly on key base concepts: race conditions, reductions, and collective operations. The assignments for each programming model can be completed in one week each. The students familiarity with operating systems, concurrency concepts, and C is assumed.

K-means. The *K*-means algorithm is an unsupervised clustering method that groups data into *K* clusters by minimizing the distance between each data point and the centroid of its cluster. It is a powerful and popular data mining algorithm used by the research community [10, 13], with applications such as data segmentation, pattern analysis, image compression, and fault detection.

Students begin with a sequential program that is intentionally designed to be understandable. It reads a cloud of points into an array. Initially, centroid positions are chosen randomly. The main clustering loop has two phases. First, each point is re-assigned to the cluster with the closest centroid. The code tracks the centroid assigned to each point and the number of cluster changes, i.e. points switching clusters. These operations can cause race conditions when they are parallelized. Second, each cluster's new centroid location is calculated as the arithmetic mean of its points. This requires counting the number of points classified in each cluster and summing their coordinate values. As the number of points assigned to each centroid can be different and they are not contiguous, this stage presents issues of load-balance and cache management. In our implementation, the program ends if thresholds on the number of iterations, number of cluster changes, or centroid displacement are reached. Figure 1 illustrates the point assignment to centroids.

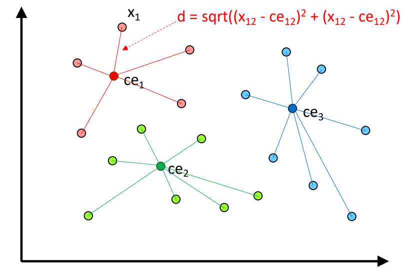

Figure 1: *K*-means clustering based on Euclidean distance for a 2D dataset with three centroids ($K = 3$)

Concepts covered. Previous approaches to parallelize the *K*-means algorithm (e.g. [11]) skip introducing parallelism to the centroid position calculations, as it leads to load balancing problems. However, this leads to a loss of potential parallelism and costly communications and synchronizations. Other educational proposals (e.g. [1]) use dynamic buffers to store the points in each cluster. This achieves better locality when traversing buffers in the second step, but adds complexity.

For this assignment, we provide code with static data structures and identify a parallelization strategy to help students apply the theory systematically. They must detect and solve both write and update race conditions as well as use other collective operations.

The parallelization strategy for this code in OpenMP has four stages: (1) Detect potential race conditions and their type; (2) Solve them with critical regions; (3) Improve efficiency by substituting them with atomic operations; and (4) Detect situations where a reduction can eliminate a race condition. The code presents opportunities for further optimizations based on cache effects, etc.

Later, students adapt this strategy to MPI and CUDA/OpenCL. In MPI, the data structures should be distributed. The initial data and results can be communicated with collective communication operations. Students who reach the fourth step in OpenMP, solving the race conditions with reductions, find MPI easier since a distributed reduction is needed in any case. For CUDA/OpenCL, students should use thread-blocks and coalesced memory accesses. They then determine the situations when atomic operations or reductions are more profitable. The second phase of the main loop also introduces load balancing issues that can be tackled by advanced students.

Using the Assignment. To introduce the assignment, we present practical applications of *K*-means. We also give an initial sequential implementation and summarize it. The parts and functions of the program to be modified are clearly marked. Example files with input point clouds of different sizes and dimensions are provided for training. The only required software is a modern C compiler with OpenMP support, any MPI library, and a CUDA or OpenCL toolkit. Although the OpenMP or MPI assignments can run on any multicore computer, the best experience requires a cluster so the students can compare performance. In our course for OpenMP, the target platform is an AMD server with 64 cores that shows interesting effects related to its 4 NUMA nodes. For MPI, we leverage two interconnected servers, one with 12 physical threads and the other with 32. During the CUDA/OpenCL evaluation, we use the

same servers as in the MPI contest, which are equipped with several NVIDIA GPUs with CUDA 3.5 architecture.

Evaluation. The course enrolled a total of 48 students, who worked in pairs. A one-week time period is given for each programming model. The students submit their best version for evaluation. They are also scheduled for an interview with the teacher, during which they are questioned about their strategies, the results, and their level of satisfaction with the assignment. All students said the project significantly enhanced their understanding of the concepts covered. They were impressed by the reduction in the program's run time.

An interesting observation is that approximately 60% of the students showed a preference for the MPI model over OpenMP. In all the previous assignments in this series, the trend was the opposite. We conclude that this assignment is easier in MPI than our previous assignments because it requires simpler collective communications and static data structures management.

4 DATA SCIENCE PIPELINE

Our third assignment is an open-ended three-week programming project where students learn to design, construct, and improve data analysis and machine learning pipelines using Hadoop, MapReduce, and Spark on the university's central compute cluster.

Wider Context. In October 2014, Friedrich Schiller University Jena began offering a two-year master's degree program in "Computational and Data Science" to bring together computational science and data science. This program is open to students with a bachelor's degree in various scientific disciplines, including computer science, mathematics, the natural sciences, and engineering. It is intended to train students not only to apply existing techniques for simulation and data analysis, but also to understand the underlying principles needed to create techniques of their own.

From its beginning, parallel and distributed computing was an integral part of this degree program. It includes a mandatory course on the design and analysis of scalable MapReduce algorithms as well as hands-on experiences with Hadoop, MapReduce, and Spark. This course is also attended by Computer Science, Mathematics, and Business Information Systems students enrolled in other master's degree programs. Thus, the course's target audience is a group of students with diverse backgrounds and some previous experiences in serial programming.

Programming Project. For practical training during the course, students solve a large number of small programming assignments. Additionally, there is a single three-week programming project toward the end of the course where students implement a more comprehensive project, typically in Spark. This project is our Peachy assignment. It is completed in teams of up to three students. Each team implements their favorite data science workflow, drawing on multiple datasets. Moderate programming skills in Python and prior knowledge of Spark are required. The instructor gives students access to a Spark framework running on a compute cluster. The assignment is not graded, but its successful completion is the prerequisite for gaining admission to the exam.

The project is designed to deepen the students' understanding of managing data-intensive and computationally-intensive applications. It addresses not only data parallelism, data replication, and storage management in distributed file systems, but also job scheduling, and resource management. However, the project's main goal is to gain practical experiences in designing, constructing, and improving true data analysis pipelines. Analogous to the "Computational and Data Science" program, the project aims to encourage critical and independent thinking. Thus, teams are given a completely free choice of topic. The only prerequisites are to (i) use at least two real-world datasets, (ii) formulate at least three different data analysis problems to be solved using these datasets, (iii) implement the solution in Spark or MapReduce, (iv) go through multiple steps of a typical data analysis workflow (data aggregation, cleaning, analysis, communication of findings using visualization), (v) present the difficulties and findings in class, and (vi) submit the code and a final project report.

To demonstrate the kind of work students do, we now discuss a submission from a pair of students who took the course in winter 2021/22. Their project considers aspects of crime in New York City. One of their data analysis problems asks for the number of arrests in distinct neighborhoods of New York City. To do this, the students used four datasets published by https://data.cityofnewyork.us, on arrests (historic and current year) and Neighborhood Tabulation Areas (boundaries and population). This pipeline identifies the spatial positions of all arrests, accumulates the number of arrests in each neighborhood, and plots a heat map. A conceptual view of the resulting pipeline is displayed in Figure 2. More detailed information is available from a public repository [18].

Classroom Experiences. Starting in winter 2015/2016, the programming project was assigned annually. Each year, the course is taken by 20–30 students. This course has a standardized survey to collect anonymous student feedback, but the response rate is around half and none of the questions directly focus on the project. Thus, we focus on some open-ended questions that students often use to refer to the programming project.

The results of these questions are aggregated in Table 1 for the four years associated with the winter terms 2019/2020 to 2022/2023. This table shows the number of students who took an exam and who completed the final survey. On the one hand, students were explicitly asked about positive feedback "What did you particularly like about this course?" On the other hand, they were asked to suggest further improvements to the course. These two categories of questions are referred to as positive and negative items (by subcategories total number of items related to the whole course and number of items explicitly addressing the programming project).

Forty-three students contributed 33 positive items about the course, 13 of them specifically about the project. Students appreciated (paraphrased) *practical experiences with Spark*, *gaining practical experiences using a cluster*, and *high relevance for future data scientists.* One student also appreciates the *improvement in my scientific writing skills.* The majority of positive items are related to *the flexibility to formulate the research problem and design the solution.* The five negative items raised in the last two years are concerned with *increasing or decreasing the size of the programming project* and the suggestion to *grade the project and let that grade constitute a*

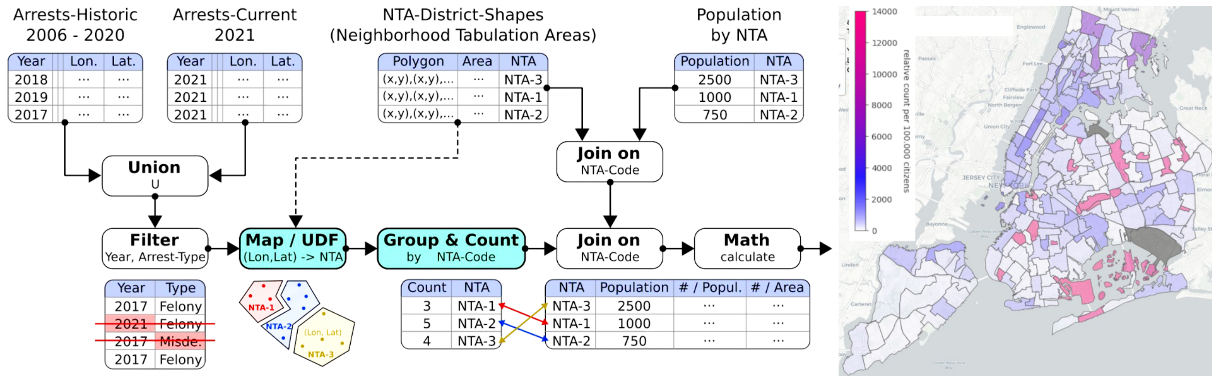

Figure 2: An analysis pipeline that combines datasets from different sources to produce a spatial heat map displaying the number of arrests per 100,000 citizens in Neighborhood Tabulation Areas (NTAs) of New York City in 2021.

Table 1: Summary of Evaluation Results

	# Students		# Pos. Items		# Neg. Items	
Winter	Exam	Survey	Total	Proj.	Total	Proj.
2022/23	22	11	14	8	8	4
2021/22	11	12	12	3	8	1
2020/21	18	9	5	2	4	0
2019/20	21	11	2	0	4	0

percentage of the final grade. We observe that students regularly exceed the project's requirements.

Finally, it is also noteworthy that, given multiple separate teams with different topics, instructors must spend substantial effort in providing useful advice and specific guidance on this project.

5 TRAFFIC MODEL

Our fourth assignment is based on the Nagel-Schreckenberg model, a stochastic one-dimensional traffic model [14]. In this assignment, we guide students through creating a shared-memory parallel and reproducible version of a serial code implementing this model.

Rationale. One of the key elements in the Nagel-Schreckenberg traffic model is the presence of randomness, without which it would lack realistic phenomena such as traffic jams. Simulating this model thus requires using pseudo-random number generators [16] in parallel, a tricky and often-overlooked scientific computing topic.

Several variations of this assignment have been used in the graduate course *PHY1610 Scientific Computing for Physicists* at the University of Toronto. This course aims to teach students the skills needed to develop scientific applications: C/C++, best practices in software engineering, use of established libraries, and experience with OpenMP and MPI. The course consistently gets positive course evaluations, is highly practical and applied, and requires students to develop code on our teaching cluster. It originated in the training program of the SciNet HPC Consortium (https://scinet.courses). Because of this, it is also suitable for other scientific disciplines and many of its topics also fit in an undergraduate curriculum.

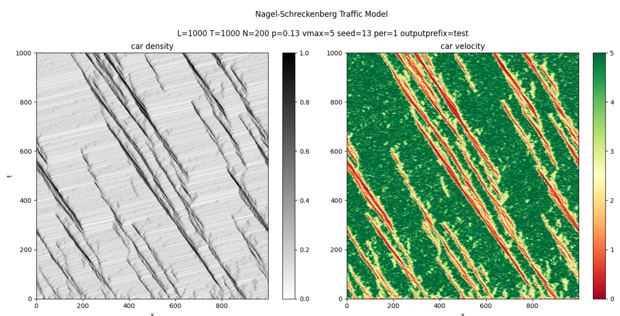

Figure 3: One-dimensional simulation of the Nagel-Schreckenberg traffic model (200 cars, length 1000, probability $p = 0.13$ and maximum velocity 5) that shows irregularities ("traffic jams") in the flow of vehicles and how they propagate. Without randomness, these do not occur.

Concepts Covered. Implementing of the Nagel-Schreckenberg traffic model requires a pseudo-random number generator (PRNG). We use this model as an excellent and easily-relatable example of a stochastic simulation. The starter code for the assignment is in C++ and can be accessed from https://github.com/Practical-Scientific-and-HPC-Computing/Traffic_EduHPC-23. Students will develop their own parallel version using OpenMP for shared-memory multi-core computers.

One nice feature of this simulation is that it can be solved using either a grid representation or an agent-based one. The grid representation assigns a value to every point on the circular road, while the agent-based implementation stores the positions and velocities of the N cars as (two) vectors of length N. Each implementation has its advantages, but the agent-based approach significantly simplifies the parallelization of PRNG.

Pseudo-random numbers are generated by sequentially deriving a number from an internal state that gets updated with every next number. Before drawing the first number, the state is initialized with a seed value, often a single integer. The state update algorithm is deterministic, and therefore the sequence is reproducible if the same

seed is used. The resulting sequence of numbers should nonetheless be nearly indistinguishable from being uniformly distributed.

In the course, students are made familiar with programming in C++, best practices in software development such as modularity, version control, unit testing, documentation, use of external libraries, make, file formats such as ASCII, binary, self-describing formats, etc. For this assignment, students should already have good working knowledge of C++ and how to use the C++11 standard random library. The starting code is fairly modular, so familiarity with the concepts of C++ headers and implementation files is helpful. Knowledge of OpenMP is required to do the assignment, including the `parallel`, `for`, and `threadprivate` compiler directives. The code does not require external libraries.

One of the trickiest parts in the parallel implementation of this model, and the one highlighted in this assignment, is managing the PRNG in parallel so that the output of the parallel code is exactly the same as the serial code. Scientific reproducibility is a critical topic nowadays. Without this requirement, one could parallelize the code by giving each thread its own PRNG, starting from different seeds. However, this gives different results when the number of threads changes. Although this may be allowed sometimes, this assignment requires identical behavior on different numbers of threads.

Reproducibility requires using a shared sequence of random numbers. While generating a random number sequence is generally a serial process, several random number generators have algorithms for quickly "moving ahead". Because these are not implemented in the C++ standard random library, the assignment starter code implements a fast-forward algorithm for one of the C++ linearly congruent generators.

Limitations. The scaling that students achieve depends highly on how well they reduced the cost of fast-forwarding the random number generators and other serial parts of the code. Scaling beyond a single socket is not ideal due to NUMA effects. Finally, one should not use more virtual cores than physical ones; even if there is a small benefit, the timing results are hard to interpret.

Variations. In this assignment, we focused on the parallelization of the algorithm, particularly the PRNG implementation using a shared-memory approach. In other variations we have used in the past, we have asked students to create their own serial implementation from scratch, or to adapt the output to use the NetCDF library. This problem offers many other opportunities for variation that address other HPC aspects. Students could implement a distributed-memory parallel code using MPI, port the code to use GPUs, run a series of parameter study cases and take advantage of embarrassingly parallel jobs, perform scaling analysis, do a performance analysis by profiling the code, change boundary conditions, etc.

6 1D HEAT EQUATION IN CHAPEL

Our fifth assignment uses the Chapel programming language to create a 1D heat equation solver. This is done in two ways, exposing various parallel programming concepts. The first part of the assignment uses high-level parallel constructs, namely Chapel's `forall` loop and `Block` distribution, to create a simple distributed-memory solver. Here, students are asked to think about what it means for an array to be split across the memory in multiple compute nodes while relying on the language to handle the details of communication and synchronization. The second part of the assignment uses low-level parallel constructs such as Chapel's `coforall` loop (used to manually spawn threads), `barriers`, and explicit communication. Here, the goal is to create a more efficient solver by reducing overhead, while also introducing students to explicit communication and synchronization. For each part, students are provided with a non-distributed version of the solver and asked to create version that runs across multiple compute nodes. The assignment materials are at https://github.com/jeremiah-corrado/Chapel-Heat1D-PPA.

Motivation. The heat equation is a simple partial differential equation (PDE) that can be solved using the finite difference method. It is highly amenable to parallelization and distribution while exposing enough complexity to motivate interesting discussion of parallel programming concepts.

Here, we will solve the PDE using the Chapel Programming Language. Chapel is a modern general-purpose language designed to make parallel and distributed programming highly productive while achieving performance similar to other HPC stacks such as C++/MPI/OpenMP. As such, it allows us to introduce some PDC concepts to new HPC programmers. Specifically, this assignment is aimed at students with some solid programming experience who are interested in scientific computing, have a need for HPC, and are familiar with at least one high-level language like Python or Matlab. Students will learn about parallelizing order-independent loops, writing distributed memory patterns, and reasoning about synchronization and communication in a multi-node setting.

Assignment. This assignment uses this form of the 1D heat equation:

$$\frac{\partial u}{\partial t} = \alpha * \Delta u = \alpha * \frac{\partial^2 u}{\partial x^2}$$

which can be converted into the following discretized form:

$$u^{n+1}[x] = u^n[x] + \alpha * (u^n[x-1] - 2 * u^n[x] + u^n[x+1])$$

Here, u is an array of values defined at discrete points in space ($x \in \Omega$) and time (n, $n+1, \ldots$). Given initial conditions and forcing values on the edges (Dirichlet boundary conditions), we will approximate u using the following algorithm:

1. define Ω to be a set of discrete points on the x-axis, and $\hat{\Omega} \subset \Omega$ to not include boundary points ($\partial\Omega \not\subset \hat{\Omega}$)
2. define an array: `u` over Ω with some initial conditions
3. create a temporary copy of `u` named `un`
4. for `nt` time steps do the following:
   1. swap `u` and `un`
   2. compute `un` in terms of `u` over $\hat{\Omega}$

The given file `Example1.chpl` has an implementation of this algorithm. An important aspect of this computation is that `un` can be computed in parallel, as each of its values depends strictly on the previous time step's values. The given code uses a `forall` loop to automatically split step 4.2 across multiple tasks. Chapel's runtime will execute tasks on all the available cores concurrently. In a distributed setting, the `forall` loop can also be used to parallelize computations across multiple compute nodes and across the cores of each node with a single loop.

The first part of the assignment is to convert `Example1.chpl` into distributed code, i.e., a version that splits the problem across

multiple compute nodes, taking advantage of the collective memory and processing capacity of a cluster. To do this, students will use Chapel's concept of a *distribution.* A distribution maps the entries in a *domain* (a set of indices, like Ω) to a memory layout across a group of nodes (called *locales* in Chapel). See the following snippet that uses the Block distribution to create a distributed 1D domain:

```
use BlockDist;
config const n = 1000;
const D = Block.createDomain({0..<n});
var a = [i in D] i;
forall i in D do a[i] *= i;
```

It initializes an array a over that domain, setting the value of each element to its index, then squares each value in parallel using a `forall` loop. Running this program across multiple compute-nodes (or locales) will split a into evenly-sized contiguous blocks in the memory across the nodes. The computation on each block will be executed on their respective locales.

Second part: One drawback of the above approach is that new tasks are created and destroyed by the `forall` loop at each time step, which incurs overhead. To avoid this, we introduce a modified version of the code that re-uses the same set of tasks throughout.

The associated file, `Example2.chpl`, makes use of several new constructs. A barrier is used to manage synchronization among tasks, something the `forall` loop did for us automatically. The `coforall` loop spawns exactly one task per iteration (unlike the `forall` loop that evenly splits work across available cores). A procedure (`taskSimulate`) is used to abstract away the computation for a single task. Array and range slices are used to copy the initial conditions into each task's local array. The `foreach` loop is used to express order-independent parallelism without creating new tasks.

In addition synchronizing via the barrier, we also need to manage the sharing of edge-values between tasks that own neighboring regions of the global array. The code does this by creating a global array of "halo" cells. At each time step, tasks store the values along their edges in their neighbors' halo cells. They then copy the neighbors' values into their own local array.

The assignment's second part is to convert `Example2.chpl` into a distributed code. To do this, we need to understand how to specify where a computation (`taskSimulate` in our case) should be executed and where memory should be allocated. In part 1, locality was specified behind the scenes by the `Block` distribution's implementation. For this part, we need to use an `on`-statement to explicitly designate which locale each task should run on, and by extension, where its variables should be stored in memory.

The following code snippet executes 10 tasks on each locale in the global `Locales` array (an array of available locales):

```
coforall loc in Locales do on loc {
  coforall tid in 0..<10 {
    var a: [1..5] int = loc.id * tid;
    writeln("Task ", tid,
        " of 10 on Locale", here.id,
        ". 'a' is on locale: ", a.locale.id);
  }
}
```

The outer `coforall` loop, creates one task per locale and uses an `on`-statement to specify the locale on which it should run. The inner loop spawns 10 more tasks, declares an array, and prints a message. This code prints one line for each task in arbitrary order.

The second part of the assignment is to use these concepts to complete the explicit task-parallel solver.

7 HYPER-PARAMETER OPTIMIZATION

Our final assignment involves Deep Learning (DL), part of Machine Learning (ML). Most DL models assume that the input data distribution is identical between testing and validation, though they often are not. For example, if we train a traffic sign classifier, the model might incorrectly classify a graffitied stop sign as a speed limit sign. Often ML provides high-confidence (softmax) output for out-of-distribution input that should have been classified as "I don't know". By adding the capability of propagating uncertainty to its result, the model can provide not just a single prediction, but a distribution over predictions that will help the user determine the model's reliability. Uncertainty estimation is computationally expensive; in this assignment, we accelerate the calculations using divide-and-conquer techniques.

An example of the desired behavior is given in Figure 4, which shows a handwriting recognition system with uncertainty estimation. The handwritten digit can be confusing even for humans. Our model outputs a value of 4, but also provides a relatively high uncertainty of 0.4. Meanwhile, a clear image produces very low uncertainty, as seen in Figure 4b. Depending on the application, the user can decide what to do with outputs of high uncertainty.

This assignment is from an undergraduate Distributed Computing (DC) class where most students have no experience in ML. We explain the ML concepts necessary to understand the problem and then explain where in the code the independent tasks are generated and how they can be distributed among nodes using MPI4Py. Necessary prerequisites include introductory python programming and an understanding of threads. The students are given slides explaining the problem and starter code. These can be found at https://drive.google.com/drive/folders/1KrxWlMZpoJzph0Y7VbZj_yYyACK-Jusl?usp=sharing)

The PDC concept covered is how to distribute independent tasks to different nodes in MPI when the number of nodes is not evenly divisible by the number of tasks. By using an assignment related to distributed ML, this concept is made more appealing to students.

How it works. To quantify uncertainty we use an ensemble, in which several models (an ensemble) are trained independently with the same data. When an ensemble is run, the result is an aggregation of the individual model results. Ensembles perform model combinations, combining weak models to obtain a more powerful one. For our project, each model is a neural network (NN). We generate these intermediate models while performing Hyper-parameter Optimization (HPO) so uncertainty evaluation is essentially free (in execution time). We use the best-performing models to identify both the uncertainty and optimal hyperparameters. For classification problems with input x and label y, the NN models the probabilistic predictive distribution of $p_\theta\{y \mid x\}$, where θ is the parameters of the NN. Let M be the number of NNs in the ensemble, with $\{\theta_m\}_{m=1}^{M}$ being the corresponding hyperparameters. Each NN is

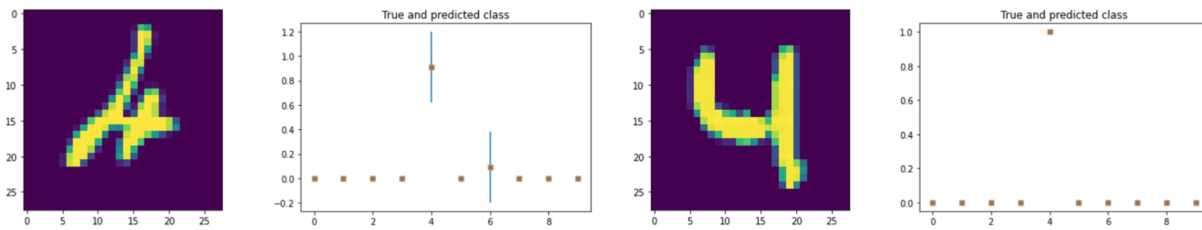

Figure 4: Image with: A) High Uncertainty. B) Low Uncertainty output is always 4 for all the ensembles

trained in parallel using the entire training set and the predictions are aggregated by averaging the predicted probabilities.

Giving the assignment: Since students are not assumed to know ML, we begin the assignment by briefly introducing it in class. We then spend 15–30 minutes of lab time explaining the provided code to do a simple Fully Connected Neural Network that classifies the MNIST handwritten digits [12].

For the application, the idea is to run each model as a tasks; this results in independent tasks whose results must then be aggregated. Once they can identify the given code, the students are asked to write the code to map the tasks to the nodes using MPI4Py, an MPI package for Python.

Interesting variations of this assignment include adding the ability to check the accuracy of the model at regular intervals or killing some of the lowest performing nodes and reassign their resources.

ACKNOWLEDGMENTS

Development of the k-nearest neighbor assignment was partially supported by NSF grants CCF-1652442 and DUE-1726809.

The k-means assignment was developed in the context of the GAMUVa group (https://gamuva.infor.uva.es/), and it has been partially supported by Vicerrectorado de Innovación Docente y Transformación Digital de la Universidad de Valladolid, Proyectos de Innovación Docente, PID2122_65, and PID2223_58, and by the NVIDIA Hardware Grant Program, which provided GPU devices used during the assignments.

Development of the pipeline assignment used the Ara cluster at Friedrich Schiller University Jena, which is supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants INST 275/334–1 FUGG and INST 275/363–1 FUGG.

Development of the hyperparameter optimization assignment was supported by Sustainable Horizons Institute, part of the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and by Argonne National Laboratory. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] Aawright, Abunch, Nslobody, and Yingchal. 2013. Decomposition, Assignment, and Orchestration of K-Means Clustering, Parallel Computer Architecture and Programming (CMU 15-418). On http://15418.courses.cs.cmu.edu/spring2013/article/10 (last visit Aug 2023).

[2] Mulya Agung, Muhammad Alfian Amrizal, Steven Bogaerts, Ryusuke Egawa, Dani el A. Ellsworth, Jorge Fernandez-Fabeiro, Arturo Gonzalez-Escribano, Sukhamay Kundu, Alina Lazar, Allen Malony, Hiroyuki Takizawa, and David P. Bunde. 2019. Peachy Parallel Assignments (EduHPC 2019), Fire extinghuising. In *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2019)*. IEEE, Denver (CO), USA. https://doi.org/10.1109/EduHPC49559.2019.00015

[3] E. Ayguadé, L. Alvarez, F. Banchelli, M. Burtscher, A. Gonzalez-Escribano añd J. Gutierrez, D.A. Joiner, D. Kaeli, F. Previlon, E. Rodriguez-Gutiez, and D.P. Bunde. 2018. Peachy Parallel Assignments (EduHPC 2018), Energy Storms. In *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2018)*. IEEE, Dallas (TX), USA. https://doi.org/10.1109/EduHPC.2018.00012

[4] H. Martin Bücker, Jeremiah Corrado, Daniel Fedorin, Diego García-Álvarez, Arturo Gonzalez-Escribano, John Li, Maria Pantoja, Erik Pautsch, Marieke Plesske, Silvio Rizzi, Erik Saule, Johannes Schoder, George K. Thiruvathukal, Wolf Weber, and David P. Bunde. 2023. Full versions of Peachy assignments from EduHPC 2023. (2023). https://doi.org/10.6084/m9.figshare.c.6860107

[5] Rocío Carratalá-Sáez, Arturo Gonzalez-Escribano, Alexandros-Stavros Iliopoulos, Charles E. Leiserson, Charlotte Park, Isabel Rosa, Tao B. Schardl, and Yuri Torres David P. Bunde. 2022. Peachy Parallel Assignments (EduHPC 2022), Hill Climbing with Monte Carlo. In *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2022)*. IEEE, Dallas (TX), USA. https://doi.org/10.1109/EduHPC56719.2022.00012

[6] Henri Casanova, Rafael Ferreira da Silva, Arturo Gonzalez-Escribano, William Koch, Yuri Torres, and David P. Bunde. 2020. Peachy Parallel Assignments (EduHPC 2020), Life Evolution. In *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2020)*. IEEE, Atlanta (GE), USA. https://doi.org/10.1109/EduHPC51895.2020.00012

[7] Henri Casanova, Rafael Ferreira da Silva, Arturo Gonzalez-Escribano, Herman Li, Yuri Torres, and David P. Bunde. 2021. Peachy Parallel Assignments (EduHPC 2021), Wind Tunnel. In *IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC 2021)*. IEEE, St. Louis (MO), USA. https://doi.org/10.1109/EduHPC54835.2021.00012

[8] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein. 2009. *Introduction to Algorithms* (third edition ed.). MIT Press.

[9] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communication of the ACM* 51, 1 (Jan. 2008), 107–113.

[10] Abiodun M. Ikotun, Absalom E. Ezugwu, Laith Abualigah, Belal Abuhaija, and Jia Hemi ng. 2023. K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences* 622 (2023), 178–210. https://doi.org/10.1016/j.ins.2022.11.139

[11] N. Mi J. Bhimani, M. Leeser. 2015. Accelerating K-Means Clustering with Parallel Implementations and GPU computing. In *2015 IEEE Conference on High Performance Extreme Computing (HPEC)*. IEEE, Waltham (MA), USA. https://doi.org/10.1109/HPEC.2015.7322467

[12] Yann LeCun, Lawrence D Jackel, Léon Bottou, Corinna Cortes, John S Denker, Harris Drucker, Isabelle Guyon, Urs A Muller, Eduard Sackinger, Patrice Simard, et al. 1995. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective* 261, 276 (1995), 2.

[13] Mitchell, Tom M. 1997. *Machine Learning*. McGraw-Hill, Boston, MA.

[14] Kai Nagel and Michael Schreckenberg. 1992. A cellular automaton model for freeway traffic. *Journal de physique I* 2, 12 (1992), 2221–2229.

[15] Steven J. Plimpton and Karen D. Devine. 2011. MapReduce in MPI for Large-scale Graph Algorithms. *Parallel Computing (ParCo)* 37, 9 (Sept. 2011), 610–632.

[16] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. 2007. *Numerical Recipes*. Cambridge University Press, Chapter 7.

[17] Erik Saule. 2018. Experiences on Teaching Parallel and Distributed Computing for Undergraduates. In *Proc of IPDPSW 2018*.

[18] J. Schoder, M. Plesske, W. Weber, and H. M. Bücker. 2023. *Project: Program Your Favorite Data Science Pipeline*. Friedrich Schiller University Jena. Retrieved August 18, 2023 from https://git.uni-jena.de/big_data_assignments/projects.git

[19] Clifford A. Shaffer. 2011. *Data Structures & Algorithm Analysis in Java* (3rd edition ed.). Dover.

[20] Ramses van Zon and Marcelo Ponce. 2023. Parallelizing a 1-Dim Nagel-Schreckenberg Traffic Model. (2023). https://doi.org/10.48550/arXiv.2309.14311