Towards a Programming-Free, Hands-On Introduction to Concepts in High-Performance and Cloud Computing for First-Year University Students

Prasun Dewan
Computer Science Department
University of North Carolina
Chapel Hill, NC, USA
dewan@cs.unc.edu

Samuel George
Computer Science Department
University of North Carolina
Chapel Hill, NC, USA
sdgeorge@cs.unc.edu

Yuvraj Jain
Computer Science Department
University of North Carolina
Chapel Hill, NC, USA
yjain@unc.edu

Mason Laney
Computer Science Department
University of North Carolina
Chapel Hill, NC, USA
mlaney@cs.unc.edu

We explore how major concepts in high-performance, distributed, and cloud computing can be introduced to students both within and outside computer science. We developed a new pedagogical framework that requires neither programming nor prerequisites and integrates hands-on interaction with realistic workflows on the CyVerse infrastructure, Socratic, LLM-mediated Zoom-based discussions, and a rich conceptual and question-driven design spanning multiple Bloom levels. This framework was implemented in a two-month module of an Honors Freshman Seminar taken by twenty-one students from nine different majors. The module covered high-level concepts from topics introduced today in multiple prerequisite-laden courses. These topics included parallelizable RNA sequencing, processes, client-server and producer-consumer models, virtual computers, processing workflows, threads, scheduling, caching, recursion, and data and pipeline parallelism. Evaluation of more than two hundred student responses to questions in assignments and exams showed strong performance across concepts and Bloom levels, despite diverse majors and no prior background in these topics. These results suggest that the proposed framework can introduce a broad range of parallel, distributed, and cloud-computing ideas in a CS-0 course while maintaining conceptual rigor, supporting the feasibility of extending such modules to larger and more varied student populations.

Keywords—HPC Education, CS-0, Socratic Teaching, Programming-Free Learning, Cloud Computing, PDC Education

I. Introduction

As a concrete motivation for this work, consider the following message posted on a discussion forum [1]:

I'm currently working on a beefy machine with 80 cores and half a terabyte of RAM, with all files on a RAM disk. I executed Kallisto with the --threads=60 option, but it is still chugging away, six hours later.

Kallisto [2] is a tool for RNA-sequencing —specifically, RNA fragment alignment—which can be executed in the NSF-supported cyberinfrastructure CyVerse [3, 4]. The replies to the above message involve discussions of non-linear parallel scaling, bottlenecks, and thread scheduling—concepts typically taught in university computer science courses on parallel programming and operating systems.

These courses, however, have a slew of prerequisites such as programming, data structures, and computer architecture. Even for computer science students, understanding how to analyze the above problem requires climbing a steep conceptual ladder. For biologists or students from other disciplines, the hurdle of taking multiple formal courses simply to grasp such performance issues is effectively insurmountable.

Therefore, it is attractive to explore the idea of teaching core concepts in high-performance and cloud computing—the domain to which the above problem belongs—within a short, prerequisite-free module of a first-year university course open to students from all disciplines. Ideally, such a module should include hands-on work that provides concrete, motivating examples of the covered problems and solutions and helps students retain the introduced concepts. This goal aligns with the fact that the motivating problem above itself arises from hands-on interaction with the Kallisto tool on CyVerse.

The idea of teaching high-performance or cloud-computing concepts in introductory courses is not entirely new. There have been notable efforts to expose students to some of these topics in introductory programming courses [5, 6]. However, because those courses focus on programming, students encounter only a narrow slice of the conceptual space. For example, they typically do not learn about cloud computing or thread scheduling—two key issues in the Kallisto example above. Expanding the conceptual scope of programming courses is difficult, as most are already overburdened and omit even the innovations introduced by the cited pioneering efforts.

To enable broader conceptual coverage, an introductory module meeting our objectives must adopt a radically different approach—one that is programming-free and therefore does not require a semester-long programming prerequisite. Moreover, by removing time-consuming programming from the course, a wider range of conceptual topics can be addressed for two reasons. First, there is more time to cover additional topics. Second, there is less interdependence among high-level conceptual ideas (the *what*) than among their implementations (the *how*). Thus, even for students who already know programming, a programming-free design remains worthwhile because it allows broader—though less deep—exploration of the intended concepts.

This shift is further motivated by the growing ability of Large Language Models (LLMs) to solve academic and industrial programming problems. Preliminary studies suggest that students who rely heavily on such models tend to retain and understand less than those who engage more directly with

This research was funded in part by NSF grant OAC 1829752.

underlying concepts [7]. The suggested response is to reinforce conceptual understanding and critical thinking rather than focus on implementation details [8].

A related motivation for this approach is that it represents an extreme extension of efforts to introduce high-performance and cloud-computing concepts early in the curriculum. Much of the existing work in this area has focused on integrating such topics at the CS-1 [5, 6], CS-1+ [9], or CS-2 [10, 11] levels. By making this module both background- and programming-free, we bring these ideas into the curriculum at the CS-0 level. We also build on modules from some of these courses that introduce unplugged concurrency—programming-free, real-world activities that do not require computers. Like those modules, our approach is programming-free but involves hands-on interaction with plugged computers. This plugged approach thus extends unplugged learning into authentic computing environments, allowing students to engage directly with real systems while maintaining conceptual accessibility.

This paper reports on our effort to design and implement such an approach within a two-month module. It describes the challenges we encountered, the strategies we employed at multiple levels of abstraction, and preliminary data from our experience. The first author taught this module as part of a new Honors Freshman Seminar titled "A Technological Preview of Computer and Data Science".

II. Methodology Overview

We provide an overview of our approach by characterizing it along the following dimensions: target audience, assumed background, mechanisms for hands-on work, worked examples and assignments, conceptual framework, and question framework and active-learning techniques

A. Target Audience

The target audience is motivated by our illustrative example, which involves the general domain of cyberinfrastructures—integrated systems that connect high-performance computing, data storage, instruments, visualization tools, and people networks to enable complex research that would not otherwise be possible [12, 13]. The U.S. National Science Foundation (NSF), under the leadership of Dan Atkins, pioneered this concept [13]. CyVerse, the system used in our illustrative example, is an NSF-supported implementation of such a cyberinfrastructure. Other similar examples include [14-16].

Recognizing the importance of these systems and the shortage of associated training resources, NSF launched the Cybertraining Program [17] to promote the development of educational materials in this space. This program distinguishes three kinds of target audiences: (a) Cyberinfrastructure Users: those who use these systems to solve domain-specific problems; (b) Cyberinfrastructure Contributors: those who create new capabilities such as algorithms, software, and tools; and (c) Cyberinfrastructure Professionals: those who maintain, support, and operate these systems.

Our module is designed to teach domain-independent concepts relevant primarily to cyberinfrastructure users. By definition, these concepts are also relevant to cyberinfrastructure contributors and professionals: contributors must understand these ideas to enhance the systems, and professionals must understand them to assist users effectively. Thus, while the course is designed for the users, it also serves as a stepping stone toward the other two communities by establishing the shared conceptual foundation on which their specialized work depends.

B. Background

The module assumes only the ability to use a personal computer to download, install, and run software, as well as a basic familiarity with mathematical functions and genetics. Because first-year university students generally possess this background, the course is well suited for them. It makes no assumptions about intended major or prior exposure to computing, programming, or data analysis.

C. Mechanisms for Hands-On Work

Hands-on work forms the experiential backbone of the module. Given the goal of teaching concepts related to cyberinfrastructure use, the mechanism for hands-on work was an operational instance of such a system. We selected CyVerse, a well-supported and sophisticated NSF-funded cyberinfrastructure, as the platform for this purpose. CyVerse consists of two major components that together provide a rich, conceptual bridge between abstract computing ideas and practical experience. The first component, *DNA Subway,* allows users to execute predefined scientific workflows on their own data. The second component, *Discovery*, provides a graphical, cloud-based operating system.

D. Implemented Worked Examples and Assignments

Worked examples were used to demonstrate how the underlying concepts could be applied to concrete problems solved by the instructor. Assignments then asked students to apply these same concepts independently to new problems designed for them.

Given the target audience—cyberinfrastructure users, contributors, and professionals—and the mechanism for hands-on work (the CyVerse system), the worked examples and assignments involved running predefined parameterized cloud applications and workflows, as well as creating and running user-defined workflows.

The main theme for the examples and assignments was RNA sequencing. This analysis involves processing files of RNA fragments collected from biological samples and matching each fragment to a known gene to determine how many times the gene was expressed—that is, how many RNA molecules were created from the gene. The greater the number of RNAs created, the higher the gene expression and the greater the number of proteins ultimately synthesized from that gene. The overall purpose of RNA analysis is often to compare gene expression under different conditions, such as the presence of various drugs.

RNA sequencing consists of the following sequence of steps, carried out in this order: (1) *Trimming*: removing artificially inserted ends from the RNA fragments in the sample data. (2) *Filtering*: removing low-quality fragments from the set of trimmed fragments. (3) *Alignment*: matching the filtered RNA segments to known genes to determine the expression of these genes. The Kallisto tool mentioned in the introduction performs

this last step. We selected RNA analysis as the central theme for three main reasons.

First, it is a fundamental scientific concept that students of all majors can benefit from understanding, as it builds on the biology knowledge they typically acquire in high school. Second, the biological process and its computational implementation have many analogues to parallel and distributed computing. A gene is a recipe for creating proteins, much as a program is a recipe for creating processes. Creating multiple copies of the gene in the form of RNAs is analogous to creating multiple processes from a program. Third, understanding this biological process through sequencing on a computer is inherently a parallel computing task. Every stage of the RNA sequencing pipeline lends itself to data parallelism: multiple RNA fragments can be trimmed, filtered, or aligned independently and concurrently. The most computationally expensive stage—alignment—particularly benefits from such data-parallel execution, as suggested by the motivating example in the introduction. Moreover, when these stages are combined into a composite workflow, they exhibit pipeline parallelism: while one stage processes new input, the next concurrently processes its previous output.

This theme guided all of our worked examples and assignments. Students were first shown how to run an RNA sequencing workflow within the DNA Subway system to understand the overall process. The worked examples and assignments then transitioned to the Discovery environment. One example showed how the output of RNA sequencing could be searched for specific genes using the standard Unix grep tool. Another demonstrated how the output of grep could be sorted by different columns using a Discovery sort application. A third example showed how the two steps could be combined into a single composite workflow in which the output of grep was automatically fed as input to sort. The steps for these examples were written down for students, who were asked to execute them on CyVerse Discovery.

The assignments extended these examples. Students were first asked to run the filter and trimmer steps serially. They were then asked to create a composite workflow that automatically passed the output of trimmer to filter. Finally, they executed the workflow as a composite. They performed these tasks on their own, drawing on the knowledge and procedural understanding gained from the worked examples.

Unlike command-line interfaces such as bash, Discovery does not support pipeline parallelism during workflow execution. Moreover, a single-line command is simulated by filling out multiple forms, making the interface more cumbersome. To overcome these limitations and help students better understand the distributed and cloud-computing principles underlying a cyberinfrastructure, we developed a Docker application that created a Docker container populated with the RNA input and output data used in their Discovery exercises. This application took the form of a Jupyter web server. Students were asked to use the browser-based bash interface on both the cloud container and the local containers. Continuing with the theme of RNA sequencing, the commands involved browsing and processing RNA data—this time using the command line rather than Discovery.

These were the implemented worked examples and exercises. Like any hands-on course, we also used a wide range of discussed examples and exercises. Because these can be easily modified or replaced, we do not present them exhaustively here; instead, we refer to selected ones later when discussing the concepts covered.

E. Conceptual Framework

In general, the choice of concepts covered in a hands-on course depends on the target audience and on the tools and exercises used for hands-on work. For instance, a Java-based introductory course may cover static typing, while a Python-based course will instead address dynamic typing. Similarly, a Java course in which worked examples and assignments use the console for program input and output will cover stream I/O, whereas one using a graphical user interface will introduce event handling. In all of these cases, no prior background is assumed in selecting the concepts to be taught.

Our choice of concepts followed the same principle but was tailored to our specific audience and tools. Because these differ from those in typical introductory computing courses, the resulting set of covered concepts is correspondingly distinct (Table I). More specifically, a concept was included in our module if it met one or more of the following criteria:

- *No background or programming*: it could be introduced without requiring prior knowledge or programming skills, allowing students to understand it at a theoretical rather than implementation level.
- *Abstraction of hands-on work*: it abstracted an idea embodied in the hands-on exercises or in the cyberinfrastructure on which they were performed.
- *Foundational concept*: it served as a basis for other concepts embodied in the hands-on work.
- *Traditional concept*: it represented a well-established topic in traditional computing courses.

TABLE I. CONCEPTS COVERED

Concept	Area/Traditional Courses	Source
OS Services*	Intro Comp./Operating Sys.	Handouts
RNA Sequencing	Genetics/Bioinformatics	Handouts
Shared File System*	Cloud Computing/OS	Handouts
Virtual Computers*	Cloud Computing/OS	Handouts
Interprocess communication*	Distributed Computing	Handouts
Processes*	Distributed Computing/OS	Lectures
Threads*	Parallel Computing/OS	Lectures
Scheduling*	Operating Systems	Lectures
Trees	Data Structures	Handouts
Tree-based Recursion	Data Structures	Lectures
Number-based Recursion	Intro. Prog./Data Struct.	Lectures
Caching	Comp. Arch/OS	Lectures
Data Parallelism*	Parallel /Dist. Computing	
Pipeline Parallelism*	Parallel /Dist. Computing	Handouts

Each row in Table I identifies a covered concept, the area (or traditional course) in which it is typically taught, and whether it was introduced primarily through lectures or assignment handouts. In all cases, the explicitly conveyed knowledge about a concept was augmented—and, in some cases, initially formed—through problems students solved in open-book homeworks and closed-book quizzes and exams. In the case of

data parallelism, problem solving was the only mechanism used to convey the concept; there were no lectures or handouts that explicitly addressed this topic, although related real-world concurrency analogies had been covered in lectures. A concept is marked with an asterisk (*) if, in our university's curriculum, students would not encounter it through the required courses. Thus, students outside computer science would not encounter any of these concepts, while those completing only the required CS courses would never encounter the starred concepts and would see the non-starred concepts only later in their studies. Below we discuss, for each concept, the specific topics that were covered through formal material and problem solving.

Processes: While running and connecting programs in CyVerse, they saw that a process is an execution of a program with a unique identifier that can be terminated and that has standard input and output streams. These streams can be connected to those of other processes to form a pipeline. They also saw that a process provides a memory space shared by its threads.

Threads: Although the exercises and the underlying systems did not require direct manipulation of threads, they saw their relevance through references to cores. DNA Subway displays both biological and system parameters for a workflow run, including the number of cores used. This motivated a discussion of threads as concurrent units of execution within a process that share the same memory.

Scheduling. To reinforce the concepts of processes and threads, they viewed process statistics on a personal computer showing that a small number of cores support hundreds of processes and thousands of threads. When they ran programs in the cloud, they saw that a "queued job" was created and that they received a notification when the job was scheduled. This experience introduced the ideas of batch computing and the sharing of limited cores among many threads and processes. They also saw that scheduling policies can differ in purpose and fairness—for example, prioritizing urgent tasks such as braking a car to avoid an accident—leading to the concepts of task deadlines and real-time scheduling.

Inter-process communication: Because the Docker containers used in our exercises ran Jupyter servers, students learned that processes communicate by sending messages to other processes through a (host, port) address. They also learned that a server publicly exposes such an address, which can then be used by multiple clients to connect to it. They also encountered bind errors when two server processes attempted to use the same port and saw the phenomenon of server switching, in which a new server process successfully used the port previously bound to a terminated process.

Virtual Computers: To explain the abstraction underlying Docker containers, they were introduced to the idea of dynamically creating one or more virtual computers on top of a physical computer—either in the cloud or on a desktop. They saw that these virtual computers share the hardware resources of the underlying host machine. Docker containers were described as delegating virtual machines that delegate all operating system services, such as process and file manipulation, to the host operating system. In contrast, they were told that conventional virtual machines implement these services directly using the underlying hardware. In the context of delegating virtual computers, they also learned how ports and directories on the virtual machines can be mapped to ports and directories on the host machine.

Operating System Services. By interacting with CyVerse, they saw that many of the functions available on their personal computers are also present in a cloud operating system, albeit through a different user interface. By working with the virtual computer, they encountered yet another interface—the bash command line. They also saw important differences among these systems. This comparison helped them understand that operating systems can vary in the services they provide, particularly in their user interfaces. The discussion emphasized both the similarities and the differences among these systems.

In addition, we introduced several key concepts exposed by the bash command line. They learned the notion of parameterized operations—illustrated by commands like *cd* and *ls*—which appears not only in program invocation but also in program implementation, where functions and methods are similarly parameterized. We also discussed the distinction between built-in operations such as *cd*, which do not create new processes, and external operations such as *ls*, which do and can be used by programmers to extend the vocabulary of the command line. They also learned about I/O redirection and pipelining. Finally, they were introduced to operating system file-management services, discussed below as a separate topic.

Shared File System: In their CyVerse exercises, students used predefined sample files located in a directory provided by the second author. Through this experience, they saw how the personal file system familiar to them was extended to a shared file system in the cloud. Their interactions on both personal and shared systems were used to formalize several key file-system concepts, including file and directory ownership, home directories, local versus absolute and relative paths, hard links to files, symbolic links to files and directories, and the recursive display, deletion, and copying of directories.

Trees: The concept of a shared file system was further abstracted through the more general notion of trees. Students learned about parent, child, root, internal, and leaf nodes, subtrees, and about operations such as changing directories to move up and down a tree. As discussed below, this abstraction was foundational for introducing recursion, which in turn motivated the high-performance computing concepts of caching and data parallelism.

Tree-based Recursion: Understanding high-performance computation first requires an understanding of basic computation. Put another way, *high-performance* is an adjective that qualifies the noun *computation*—and one cannot meaningfully understand adjectives without understanding the nouns they modify. More concretely, caching optimizations cannot be understood or evaluated without understanding non-cached computation, and parallel or distributed computation cannot be understood or evaluated without understanding the sequential computation executed by each thread in a process.

The traditional approach to introducing high-performance computing is to layer it on top of number-based iterative computation, such as finding the sum of a set of numbers. Our

tree-based exercises allowed us to challenge this convention by introducing basic computation through examples of tree recursion. Before object-oriented programming became commonplace, many curricula used functional, recursion-based computation to introduce basic computation. Our approach was motivated by this experience, but it differed in two key ways: it used tree manipulation rather than number manipulation as examples, and the algorithms were expressed in unconstrained natural language rather than within the syntactic constraints of a functional language.

There were two related reasons for choosing tree-based recursion. First, it allowed us to explain recursive listing, copying, and deletion of files—operations symbolized by the *-R* option in bash commands. More generally, the tree-based context of file manipulation provided concrete examples of functions used in these operations that could be defined recursively. To illustrate, our first example of recursion was not the traditional factorial but the definition of the absolute name of a file node in the CyVerse file system. It was given as follows

If the node is the root, then its absolute name is "/home"; otherwise, it is the local name of the node + "/" + the absolute name of the node's parent.

The file system provided many functions that could be recursively defined, including determining the name of a node relative to an ancestor, the number and set of nodes in a subtree, and the number and set of ancestors of a node. Students saw that in some of these examples, recursion climbed up the tree, inheriting information from ancestor nodes, while in others it descended down the tree, synthesizing information from descendants. In all cases, there was a base case where the climb or descent stopped and a recursive case where the function was applied to a parent or child of the larger case. Unlike traditional mathematical examples, in all of these cases recursion was grounded and made concrete by a notion they had encountered directly in their hands-on exercises, rather than by a mathematical function unrelated to any application area covered by the course.

Number-based Recursion: Once tree-based recursion was understood, number-based recursion followed readily. It was introduced as the process of increasing or decreasing a numerical parameter in the recursive case and stopping at a boundary value in the base case.

Caching: Caching is usually introduced in the abstract context of computer memory access. Our examples based on the persistent file-system tree provided a more concrete context—one students had already explored in depth through recursive problems—for understanding this general concept. The result of invoking a function on a persistent file-system node could be saved and reused the next time it was needed, avoiding re-computation. For example, the result of computing the absolute name of a node could be reused when the absolute name of a descendant node was subsequently required.

Using extra space for caching led to the introduction of the general concept of the space–time tradeoff. It further motivated discussion of caching-specific challenges. While caching can save time, it can also lead to incorrect results due to stale data—for example, a cached absolute name of a node becomes invalid if any ancestor's name or position in the hierarchy changes. Students also saw that the time required to compute a function can vary dramatically depending on whether a cached value is used. This, in turn, led to discussion of accounting problems in real-time scheduling policies that must meet strict deadlines, such as braking in time to avoid an accident. Thus, through caching, we addressed and linked the concepts of correctness, performance variability, and real-time scheduling.

The space–time tradeoff, in turn, led to a discussion of the design space of cache-eviction policies. Three such policies were discussed. The first was an algorithmic policy in which inheritance-based tree algorithms gave preference to cached values of higher-level nodes, since these values are relevant to all descendants of the node. The other two were data-driven policies: one that gave preference to values used more frequently, and another that gave preference to values used more recently. Students were then introduced to a more abstract and unfamiliar context for caching—memory caching—and learned that the third policy, eviction of Least Recently Used (LRU) items, is generally believed to perform best in this context. The distinction between data-driven and algorithmic policies was a recurring theme in the course and was also introduced in the context of parallelization of work, discussed next.

Data Parallelism: The traditional approach to motivating data parallelism is to have students examine or implement toy problems such as the parallel summation of a set of numbers. However, it is difficult to make such examples large enough to demonstrate measurable performance improvement from multithreading or distribution. In contrast, the alignment step in the RNA sequencing workflow provided a more compelling motivation in our course, as illustrated by the Kallisto complaint in the introduction. Students saw that this step took a long time to execute even when the set of RNA samples was artificially reduced.

Our handout on RNA sequencing explained why, describing the alignment process and the large sizes of both the RNA fragment set and the gene database against which the fragments had to be matched. The tree-based synthesizing recursion problems provided another opportunity for concurrency that students had already seen in their hands-on work. In both contexts, the same computation could be performed concurrently on different subsets of a set or on different subtrees of a tree. This form of concurrency corresponds to data parallelism, though the term itself was never introduced formally, as no lecture or handout explicitly addressed such parallelization of work.

Given a problem involving a set or a tree and a specified number of available threads or cores, students were asked to determine how the data could be divided among them. They had to rely on their formal understanding of threads, cores, and the problem itself to make this assignment. As in OpenMP-based programming exercises, how the threads were created or assigned to the selected data was treated as "magic."

Pipeline parallelism: The RNA sequencing and grep-sort workflows provided a concrete setting for pipeline parallelism, which was introduced through the producer–consumer model.

RNA Sequencing: As seen in this section, RNA sequencing served as a cross-cutting theme that provided both the conceptual motivation and the unifying context for the module. This theme not only anchored the hands-on work in DNA Subway and Discovery but—like tree manipulation—also provided a coherent narrative linking diverse computing topics.

F. *Question Framework Active Learning and Assessment*

The class lectures adopted a Socratic style of instruction, encouraging students to derive general concepts from specific examples. For instance, to elicit the concept of caching, students were shown two queries to find the absolute names of a parent node and its child, and asked how the second query could be made faster given that the first had already been executed. Similarly, to introduce caching policies, they were presented with a cache containing three items and a sequence of access requests, and asked to propose and defend three different policies that would favor each of the three items. Some of these inductive Socratic questions were followed by deductive ones in which the derived abstract concept was applied to new concrete examples. For instance, after discussing caching, students were asked to determine the cache contents for a given sequence of absolute-name queries.

Some questions followed a traditional, audio-based discussion format in which questions and answers were spoken by the instructor and students, respectively. In this serial mode, typically only a few confident students volunteered answers, and others often withdrew their raised hands once a similar answer had been given. These spoken responses were ungraded and intended solely to stimulate discussion.

Drawing on our experience during the COVID pandemic, we adopted a more concurrent, text-based style in which questions were posed orally but answers were composed in parallel and submitted privately to the instructor through Zoom chat. Once all submissions were received, they were publicly analyzed using the GPT LLM. This approach not only broadened participation but also mirrored the course's subject matter—moving from serial to parallel modes of processing and interaction. Unlike stylized question–answer systems such as Poll Everywhere, which are optimized for short, multiple-choice responses, Zoom chat was designed for natural-language communication and was therefore well suited to our pedagogy: it allowed students to express answers in natural rather than programming language, reinforcing the course's focus on concepts over implementation, and it supported inductive, concept-forming questions rather than deductive, concept-application ones. Because these responses were automatically recorded, 10% of the course grade was allocated to them, with full credit awarded for any earnest attempt regardless of correctness. Only students attending the lecture in person were eligible to participate.

Active learning in class also included two in-class hands-on exercises using Docker and bash. Progress on these exercises was assessed through Google Form questions. The same question-based format was used for two homework assignments and two in-class exams: a short, eight-minute preview quiz and a full-period, seventy-five-minute midterm. Students were encouraged to check and correct their homework answers by collaborating with peers or consulting an LLM. Exams and the two in-class exercises were closed-book.

III. Experience

The freshman seminar was taken by twenty-one students from diverse majors, spanning a wide technical spectrum from Political Science to Computer Science: Computer Science (eight), Data Science (two), Political Science (one), Biology (two), Economics (two), Physics (one), Mathematics (one), Geography (one), and Business (three). Eight of the students were women. The module described here consisted of fourteen seventy-five–minute lectures. All topics taught in this module were unfamiliar to the students, except one Computer Science student had previously heard of caching.

We gathered a variety of statistics to gauge student learning from their responses to more than two hundred questions, distributed as follows by source: Zoom (17), homework assignments (123), in-class exercises (29), and exams (36). Figure 1 shows the distribution of these questions by concept and source. Concepts that were either more complex (e.g., tree recursion) or directly referenced in the hands-on work (e.g., files, processes, operating-system services) were associated with higher question counts. Not all concepts were covered by every source; for instance, data parallelism was not addressed in class.

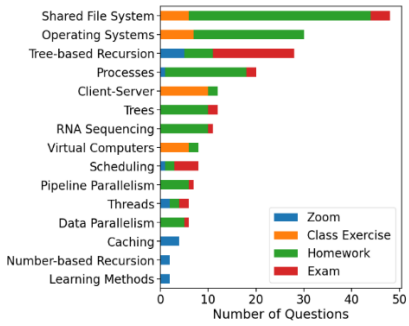

Fig. 1. Question Distribution by Source and Concept

Figure 2 shows the distribution of questions by source and by level in Bloom's taxonomy [18]. The middle levels were the most represented overall. The relatively large number of "remember" questions reflects the fact that many time-limited, in-class exercises required students to simply report the outcomes of their experiments—verifying task completion rather than explaining or analyzing the results.

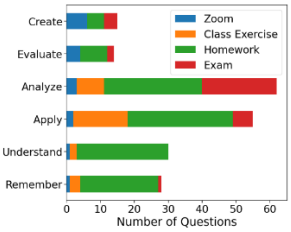

Fig. 2. Question Distribution by Source and Bloom Level

The classification of questions by concept and Bloom's taxonomy level was performed by the fourth author with assistance from the Gemini LLM. He first prompted Gemini to generate the initial categorizations and then manually corrected

its errors. The weighted macro-F1 agreement between the LLM's classifications and his final corrections was 0.96 for concept categories and 0.87 for Bloom's levels.

How well did students answer these questions? We first consider the Zoom responses. Recall that these answers were not graded for correctness, as they were primarily Socratic in nature and targeted concepts that students had not yet formally learned. Every question elicited a response from each attendee, ensuring universal engagement and giving the instructor a real-time gauge of each student's conceptual understanding. Except for one student, who attended fewer than five lectures, all students attended every class unless they were sick.

To evaluate how well this learning method worked—and how intuitive and accessible each concept was—we classified the answers into three categories: almost or fully correct, along the right track, and off track. The third author segmented the chat logs by question, used the GPT LLM to perform the initial classification, and then manually corrected any misclassifications. The weighted F1 agreement between the LLM's output and the author's final coding was 0.91.

Figure 3 shows the distribution of response correctness after the third author's manual corrections. The figure includes one non-technical category—learning methods—not represented among the technical topics in Table I. In this category, no responses were off track, as students could be expected to have prior exposure to general learning strategies. The predominance of "right-track" answers across nearly all topics indicates that even when students did not reach full correctness, they were actively reasoning about the underlying ideas and moving toward conceptual understanding. The relatively low proportion of off-track answers further suggests that even technically novel topics were accessible through the module's programming-free, concept-focused design, reinforcing the effectiveness of the approach for a diverse audience.

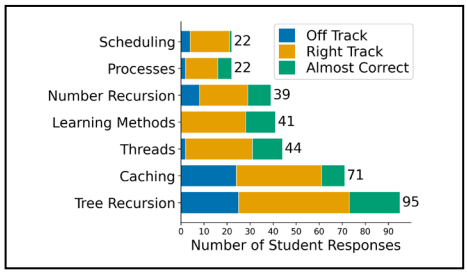

Fig. 3. Off-Track, Right-Track and Almost-correct Zoom Answers

This overall accessibility pattern also allows finer-grained insights into relative concept difficulty. Our previous work on a programming-based introduction to sequential and concurrent programming showed that the basic idea of threads was easier for students to comprehend than several taught sequential concepts [9]. This trend appears here as well. Students struggled relatively little with threads compared to more abstract sequential topics such as caching and recursion. Consequently, more questions—and lecture time—were devoted to these concepts, as reflected in the higher number of responses along the X-axis. As seen in the figure, the processes and threads bars have similar shapes, which is intuitive, as the only difference is that processes, unlike threads, have their own memory. Scheduling was more difficult than these two concepts probably because it builds on them.

To investigate the reasons behind the observed distribution of answer quality, we examined whether off-track responses reflected lower engagement and whether nearly correct answers resulted from prior knowledge that could be applied directly. If this reasoning were correct, students would spend less time composing both off-track and almost-correct responses. We tested this hypothesis using Zoom chat timestamps to measure response times. However, the violin plots in Figure 4 contradict this expectation: off-track answers actually took the longest to compose, while almost-correct answers required slightly less time than right-track ones.

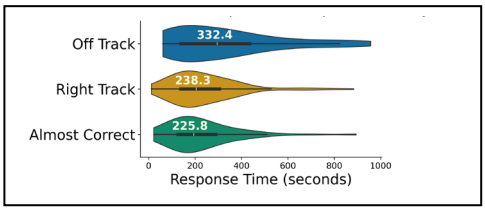

Fig. 4. Time Taken by the Three Kinds of Answers (Violin Plots)

How did students perform on the graded Google Form questions? Figure 5 shows violin plots by concept for the average score across all sources, while Figure 6 focuses on total scores for the two exams. These questions were graded manually by the last author, who served as the teaching assistant for the course. All three plots indicate that the vast majority of students answered these questions correctly. We hypothesize that the highly engaging, Socratic, Zoom-style questioning—allowing the instructor to address each student's misunderstandings in real time—was a contributing factor. Another likely factor is that broad conceptual questions tend to elicit fewer mistakes than implementation-oriented ones, which often involve numerous subtle boundary conditions. Interestingly, RNA Sequencing and Scheduling produced the lowest scores: the former likely reflects the fact-based nature of RNA Sequencing, whereas the latter may stem from the high level of abstraction involved in Scheduling.

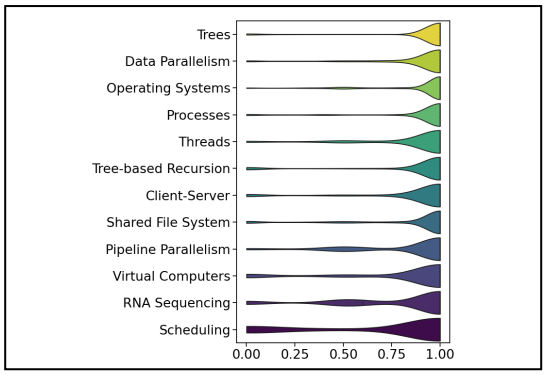

Fig. 5. Violin Plots of Scores By Concept for All Questions

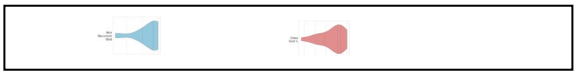

Fig. 6. Violin Plots of Total Score on Short Quiz (left) and Midterm (right)

As part of the course, the second author developed a Chrome plugin that recorded when a Google Form was opened and submitted, allowing us to estimate how much time students spent answering each form. Installation of the plugin was voluntary, and we obtained anonymous logs from about half of the class. Figure 7 shows the results for the six problem sets. As seen in the figure, there was greater variation in the time taken than in the scores earned, possibly reflecting that some students worked in a more "slow and steady" manner than others. The in-class short quiz lasted eight minutes; hence, the time scale does not capture the fact that nearly all students took approximately the same amount of time on it. These plots also show that students spent considerably less time on homework than the six to nine hours per week typically expected for a three-credit course, suggesting that additional or more challenging issues could be incorporated into future assignments.

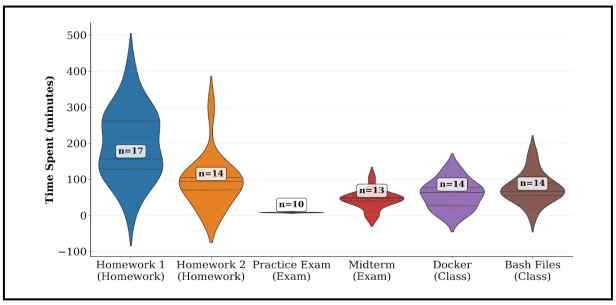

Fig. 7. Violin Plots of Time Taken (from left to right) by HW1, HW2, Short Quiz, Midterm, Class Exercise 1, and Class Exercise 2.

IV. Conclusions and Future Work

The opportunity to offer a Freshman Seminar without traditional curricular constraints allowed us to experiment with a background- and programming-free approach for introducing a wide range of concepts relevant to users, contributors, and administrators of cutting-edge cyberinfrastructures. The vast majority of these concepts receive no exposure in our required courses. We developed a new pedagogical framework that integrates hands-on interaction with realistic workflows on the CyVerse infrastructure, Socratic, LLM-mediated Zoom-based discussions, and a rich conceptual and question-driven design spanning multiple Bloom levels.

We evaluated our work not only through traditional metrics such as exam and homework scores but also by analyzing the quality of in-class responses and the time students took to compose them. Further, we provided fine-grained analyses of these metrics, showing their distribution by concept and Bloom's taxonomy level. Despite the diversity of student backgrounds and the breadth of topics covered, students performed well across all concepts and Bloom levels. These preliminary results give us confidence in our goals, conceptual framework, and pedagogical techniques.

While diverse in major, the small Honors cohort studied here may not represent the broader undergraduate population. Future work includes offering the module to non-Honors students and scaling it to larger class sizes. Additional studies can compare students' conceptual understanding under our approach with that of students in traditional programming-based courses. Longitudinal studies can also assess how well students retain these concepts over time and apply them in subsequent coursework or real-world contexts. While programming-based approaches allow automatic grading [19, 20], developing analogous methods for evaluating natural-language responses—potentially through AI-based analysis—represents another promising direction for future investigation. Extending students' understanding and knowledge through programming-based instruction of the covered topics is another important direction.

Acknowledgment

Montek Singh served as a sounding board and advisor through all stages of the course design—from crafting its title to discussing its implementation details. The comments of the referees helped better position this work and identify limitations to be addressed in future iterations.

References

[1] Various, G. kallisto quant --threads=60' runs on one core for hours. 2017; Available from: https://github.com/pachterlab/kallisto/issues/128.

[2] Bray, N.L., H. Pimentel, P. Melsted, and L. Pachter, Near-optimal probabilistic RNA-seq quantification. Nature biotechnology, 2016. 34(5):

[3] Merchant, N., et al The iPlant collaborative: cyberinfrastructure for enabling data to discovery for the life sciences. PLoS biology, 2011. 1

[4] CyVerse, Cyverse Learning Center. 2016. https://cyverse.org/learning.

[5] Bogaerts, S., Hands-on Parallelism with no Prerequisites and Little Time Using Scratch, in Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses,. 2019,

[6] Bruce, K.B., A. Danyluk, and T. Murtagh, Introducing Concurrency in CS 1, in Proc. ACM SIGCSE'10. 2010, ACM.

[7] Jošt, G., V. Taneski, and S. Karakatič, The impact of large language models on programming education and student learning outcomes. Applied Sciences, 2024. 14(10): p. 4115.

[8] Lohr, S., How Do You Teach Computer Science in the A.I. Era?, in New York Times. 2025: New York Times.

[9] Dewan, P., S. George, A. Wortas, and J. Do, Techniques and Tools for Visually Introducing Freshmen to Object-Based Thread Abstractions Journal of Parallel and Distributed Computing, 2021. 157.

[10] Grossman, D., Fork-Join Parallelism with a Data-Structures Focus, in Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses, 2019, Morgan Kaufmann.

[11] Adams, J.C., Injecting parallel computing into CS2, in Proceedings of the 45th ACM technical symposium on Computer science education. 2014,

[12] Stewart, C.A., S. Simms, B. Plale, M. Link, D.Y. Hancock, and G.C. Fox. What is cyberinfrastructure. in Proceedings of ACM SIGUCCS 2010.

[13] Atkins, D.E., et al Revolutionizing Science and Engineering Through Cyberinfrastructure: Report of the National Science Foundation Blue-Ribbon Advisory Panel on Cyberinfrastructure. 2003; Available from: https://www.nsf.gov/cise/sci/reports/atkins.pdf.

[14] Deelman, E., et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific programming, 2005..

[15] Authors, V. OneSciencePlace. 2025; Available from: https://onescienceplace.org/.

[16] Herre, C., et al, Introduction of the Capsules environment to support further growth of the SBGrid structural biology software collection. Biological Crystallography, 2024. 80(6): p. 439-450.

[17] NSF, Office of Advanced Cyberinfrastructure, Blueprint for Cyberinfrastructure Learning and Workforce Development. 2021,

[18] Bloom, B., Taxonomy of Educational Objectives Book 1: Cognitive Domain. 1956: Addison Wesley.

[19] Dewan, P. Infrastructure for Writing Fork-Join Tests. in Proceedings of the SC'23 Workshops 2023.

[20] Dewan, P. and N. Gaddis, Leveraging Valgrind to Assess Concurrent, Testing-Unaware C Programs, in HiPCW. 2024, IEEE. p. 28-35.