Teaching Task-Based Parallel Programming with a Runtime Systems-Aware Perspective

Vivek Kumar
Department of Computer Science and Engineering
IIIT Delhi
New Delhi, India
vivekk@iiitd.ac.in

Abstract

Conventional parallel programming using explicit multithreading over modern multicore processors imposes significant complexity in organizing and balancing work across threads. Task-based models simplify parallel programming using runtimes that handle task scheduling and resource management, improving scalability and reducing developer effort.

This paper presents the structure and experiences of teaching the Parallel Runtimes for Modern Processors course (PRMP) at IIIT Delhi. The course introduces a basic task-based parallel programming model in the async–finish style. Students implement this programming model together with a general-purpose dynamic load-balancing runtime system. As the course advances, students gradually improve both the parallel programming model and the runtime to overcome limitations and challenges of modern processor architectures. We conclude with a qualitative and quantitative evaluation of the three offerings of PRMP to date, showing that the course has significantly improved student's understanding of how to write and execute parallel programs effectively.

CCS Concepts

• **Software and its engineering → Software notations and tools**; **Runtime environments**;

Keywords

Task parallelism, async-finish, work-stealing, education, teaching

ACM Reference Format:
Vivek Kumar. 2025. Teaching Task-Based Parallel Programming with a Runtime Systems-Aware Perspective. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3731599.3767387

1 Introduction

Computing hardware is becoming increasingly complex. Modern multicore processors used in the cloud, data centres, and supercomputers now scale to over a hundred cores, have wide vector units, maintain a complex memory hierarchy and often share memory with accelerators such as GPUs. For example, the top three entries in the Top500 list [3] include nodes with up to four sockets, 104 CPU cores, and many GPU cores. However, achieving high performance on such systems through explicit multithreading is difficult, primarily because it is hard to identify and express enough parallelism in software. OpenMP [13] is the de facto parallel programming model for multicore parallelism and provides a high-level way of expressing shared-memory parallelism. However, its performance is often limited on modern processors [21]. Several asynchronous, dynamic task-based parallel programming models have emerged to address the growing demand for portability and scalability in parallel applications on modern processors by treating *tasks* as first-class citizens [8, 20, 22, 33, 42, 51]. These programming models enable the creation of a large number of lightweight asynchronous tasks (**async**), which are efficiently executed on multicore processors using an underlying work-stealing runtime system [16]. Some of these frameworks focus mainly on node-level parallelism, such as Cilk [8], Java fork/join [33], Intel TBB [42], Qthreads [51], and the Habanero Java/C/C++ libraries [20, 22]. Others are designed to support both intra-node and inter-node parallelism, such as X10 [11], Chapel [10], Legion [46], Kokkos [9], Raja [6], HPX [24], and Habanero-UPC++ [32].

Due to the ubiquity of multicore processors and the growing adoption of task-based parallel programming models, maximizing performance has become a critical skill for today's software developers. Hence, several universities worldwide have integrated task-based parallel programming into their parallel computing curricula [18, 19, 25, 29, 43]. One such course is Foundations of Parallel Programming (FPP), which we have developed and offered at IIIT Delhi for over seven years to undergraduate and postgraduate students [29]. FPP adopted a task-based parallel programming approach to balance performance with programmer productivity. It first introduces explicit multithreading and its limitations on multicore processors. Then it covers a variety of tasking APIs suitable for such systems, along with a brief introduction to distributed parallel programming. As an introductory course, FPP primarily focused on enhancing student's parallel programming skills, with limited focus on teaching runtime engineering to optimize performance and scalability on increasingly complex multicore processors.

HPC relies on skilled professionals to design and optimize parallel applications. As multicore processors grow more complex, training students in both parallel programming and performance engineering becomes essential [50].To meet this need, we developed the Parallel Runtimes for Modern Processors (PRMP) course at IIIT Delhi [30], a 4-credit offering taught over the past three years to undergraduate and postgraduate students. Building on the foundation of the FPP course, PRMP focuses on tackling the

This work is licensed under a Creative Commons Attribution 4.0 International License.
SC Workshops '25, St Louis, MO, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767387

challenges of parallel programming on modern processor architectures. It combines theoretical foundations with hands-on experience in task-based parallel programming and performance tuning via runtime techniques. The course introduces **async–finish** style parallelism, implemented by students alongside a work-stealing runtime. It then explores advanced topics in processor-aware runtime design and load balancing. A semester-long project reinforces these concepts by having students iteratively enhance their task-parallel APIs and the underlying load-balancing runtime systems. This paper presents the design, structure, and outcomes of PRMP in pursuing the twin goals of teaching task-based parallelism and performance engineering.

The following sections cover different aspects of the PRMP course. We begin with an overview of the **async–finish** style parallel programming model in Section 2. Section 3 outlines the teaching methodology and course content. Section 4 explains the evaluation components. Section 5 summarizes student feedback. In Section 6, we share our teaching experiences, and finally, Section 7 concludes the paper.

2 Task parallelism using async–finish

Task parallelism gained popularity with the introduction of the Cilk language [8]. Cilk, an extension of the C programming language, introduced the **spawn** and **sync** keywords for using task parallelism. The **spawn** keyword indicated a function (task) that could execute asynchronously relative to the subsequent statements, while **sync** ensured that all previously spawned tasks completed before continuing. Several other programming models and libraries later adopted task parallelism, each introducing their nomenclature and abstractions for tasking APIs [6, 9–11, 20, 22, 24, 33, 39, 42, 51]. The **async–finish** APIs were originally introduced by the X10 programming language [11], and have since been adopted by the Habanero Java/C/C++ libraries [20, 22]. Figure 1 shows how to use the C++11 lambda function-based **async–finish** APIs to parallelize two simple operations on an unbalanced tree with an irregular execution DAG [20]. Both constructs allow nesting. For example, Figure 1(a) demonstrates a recursive use of **finish**, where each **async** task creates new **async** calls within its own **finish** scope, whereas Figure 1(b) demonstrates a flat **finish** structure, where all **async** tasks are launched within a single global **finish** scope. The benefit of a flat **finish** structure is the elimination of nested synchronization.

```
1  int recurse(Node* node) {
2    std::vector<int> counts(node->numChild);
3    finish([&]() {
4      for(int i=0; i<node->numChild; i++) {
5        async([&, i]() {
6          counts[i] = recurse(node->child[i]);
7        });
8      }
9    });
10   int sum=1;
11   for(int i=0; i<node->numChild; i++) {
12     sum+=counts[i];
13   }
14   return sum;
15 }
16
17 int countNodes(Node* root) {
18   int total = recurse(root);
19   return total;
20 }
```

(a) Couting total nodes in an unbalanced tree

```
1  std::atomic<bool> found;
2  void DFS(Node* node, int goal) {
3    if(node->value == goal) {
4      found = true;
5      return;
6    }
7    for(int i=0; i<node->numChild; i++) {
8      if(found) return;
9      async([=]() {
10       DFS(node->child[i], goal);
11     });
12   }
13 }
14
15 bool search(Node* root, int goal) {
16   found = false;
17   finish([=](){
18     DFS(root, goal);
19   });
20   return found;
21 }
```

(b) Searching a node in an unbalanced tree

Figure 1: Examples to demonstrate the task-based parallel programming model by using the async–finish APIs

Task-based parallel programming languages and libraries generally use a work-stealing runtime for the load balancing of dynamically generated **async** tasks. The runtime employs a pool of worker threads, each maintaining a double-ended queue (deque). Tasks created by a worker (called the `victim`) are pushed to the LIFO end of its deque. The victim continues executing tasks by popping from the same end. When no more tasks are left at a worker's deque, it becomes a `thief` and randomly selects another worker to steal a task from the FIFO end of that worker's deque. If the steal attempt fails, the thief continues searching for a task at another worker until the program terminates. Although this straightforward implementation of **async–finish** improves productivity and performance over explicit multithreading, it encounters several challenges over modern multicore processors. These include sequential overheads from fine-granular **async** [4, 14, 23, 49], the lack of co-location of **async** and its data on non-uniform memory access architectures [28, 41], difficulties in tracing fine-grained nested **async** calls [35], false sharing between tasks running on different workers [36], achieving energy efficient execution [47], enabling fault-tolerance [40], and extending task parallelism for hybrid computing across CPU and GPU platforms [5]. PRMP course takes a deep dive into all these issues and teaches students how to address them by extending tasking APIs and enhancing the work-stealing runtime.

3 Teaching Methodology and Course Topics

The PRMP course is designed to be suitable for both undergraduate and postgraduate students. The students must have completed an operating systems course and at least one programming-intensive course. We expect the students to have prior experience with C/C++ programming, and completing a computer architecture course is recommended, though not mandatory. The PRMP course has four

Lecture No.	Lecture Topics	Short Description	COs Fulfilled
1	Introduction to the course	Course structure, evaluation criteria and logistics	–
2-3	Introduction to parallel programming	Design of modern multicore processors, refresher on Pthread programming [34], concurrency decomposition [17]	CO1
4-5	Dynamic task parallelism [2]	async-finish programming model, serial elision, computation graphs, mapping async tasks to library-based thread pool runtime, work-stealing	CO1 and CO2
6-7	Sequential overheads of work-stealing	Techniques for controlling task granularity and reducing overheads arising from concurrent deque accesses	CO2 and CO3
8-9	Task parallelism over NUMA machine	Page allocation policies on NUMA machines, libnuma library hierarchical work-stealing, recursive task parallelism using data-affinity hints in async	CO2 and CO3
10	Trace and replay	Profiling of async-finish program, steal trees, trace/replay of async tasks in iterative algorithms	CO3
11-12	User level threads	Context switching inside the userspace using Boost.Context library [26], programming with user-level threads using Boost.Fiber library [1]	CO1 and CO3
13	Mid semester exam review	Review of concepts taught in Lectures 02–12	CO1–CO3
14-15	Memory consistency and synchronization	Sequential consistency, x86 TSO memory model, store buffers, C++11 memory model, atomic operations, lock-free work-stealing deques	CO3
16-17	Cache coherency and false sharing	Cache coherency protocols, writing cache friendly code, runtime solutions for detecting/repairing false-sharing	CO3
18	Achieving energy efficiency	Dynamic concurrency throttling, processor's core and uncore frequency scaling	CO4
19-21	Heterogeneous parallel programming	Programming SIMD vector units using Vector Class Library [15], GPU programming using Boost.Compute library [38], runtime solutions for hybrid CPU-GPU task parallelism	CO1, CO3 and CO4
22	Resiliency	Runtime solutions for resilient task parallel programs	CO2 and CO3
23-25	Research seminars	Student led seminars on project work, and presentation of recently published research papers related to task parallelism	CO1–CO4
26	End semester exam review	Review of concepts taught in Lectures 14–22	CO1–CO4

Table 1: Detailed description of lecture topics in PRMP along with focus area and CO mapping

Course Objectives (COs), defined using Bloom's taxonomy [7]. These are:

- CO1: Students will be able to explain parallel programming abstractions and their underlying hardware and software implementations.
- CO2: Students will be able to design and implement a low-overhead load balancing runtime for task-based parallel programming for modern multicore processors.
- CO3: Students will be able to implement and evaluate runtime optimizations to improve the performance of parallel applications on modern processors.
- CO4: Students will be able to implement and evaluate runtime techniques for improving the energy efficiency of parallel applications on modern processors.

Table 1 presents the details of the lecture topics covered in the PRMP course, with each lecture lasting 90 minutes. The course content can be broadly grouped into four modules, as described in the following sections.

3.1 Foundations of Task Parallelism

Lectures 2–5 cover the fundamentals of parallel programming and task-based parallelism, and are the only lecture topics adapted from the Foundations of Parallel Programming (FPP) course offered at IIIT Delhi [29]. Since students have already completed an Operating Systems course, they are familiar with explicit multithreading in C/C++ using Pthreads. However, we start the course with a refresher on Pthread programming, highlighting the productivity and performance challenges it poses on modern multicore processors (Lecture 2). Students are taught a variety of concurrency decomposition techniques along with practical use-cases (Lecture 3). Students are then introduced to **`async–finish`** style task parallelism and dynamic load balancing of **`async`** tasks using a work-stealing runtime (Lectures 4–5). Instead of using an existing dynamic tasking library, we explain the design and implementation using pseudocode for C++11-based **`async–finish`** APIs and a lightweight, library-based work-stealing runtime. Students use this pseudocode in their project milestone-1 as explained in Section 4.2.

3.2 Runtime Optimizations

Several prior research works have highlighted the scalability issues of a straightforward implementation of the work-stealing runtime over modern multicore processors [4, 14, 23, 28, 35, 41, 49]. Lectures 6–10 cover these scalability issues and discuss solutions to overcome these limitations. We start this module with the sequential overheads associated with a work-stealing runtime [31] (Lectures 6–7). These overheads arise from task creation overheads associated with fine-granular **`async`** [14, 23] and expensive memory fences incurred during concurrent deque accesses [4, 49]. Students measure these overheads by changing the task granularity of recursive **`async–finish`** programs, and observing the effect on total execution time. We cover the design and implementation for automatic

task granularity control using dynamic task aggregation [14], dynamically switching to an iterative version of recursive code upon reaching a depth threshold in the recursion tree [23], and task memoization. To address deque access overheads, we discuss techniques such as using a mix of private and non-concurrent data structures with regular work-stealing deques [4, 49].

Modern servers and HPC nodes increasingly employ multi-die and multi-socket designs with multiple memory banks and cache hierarchies to improve memory bandwidth. While this design enables cache-coherent Non-Uniform Memory Access (NUMA), it hampers the scalability of memory-bound task-parallel programs due to increased cache misses and the latency of remote memory accesses caused by a random work-stealing runtime. We discuss these issues in Lectures 8–9, where we first cover the design of a hierarchical work-stealing, but with manual task partitioning over the NUMA domains [41]. Students learn that while this approach works fine for a regular execution DAG, the productivity and performance will suffer with an irregular execution DAG. If a DAG has the same branching degree for all its non-leaf nodes, it is a regular DAG and an irregular DAG if different branching degrees. Students are then introduced to extending the **`async`** API with data-affinity hints [28], where each **`async`** task explicitly specifies the arrays and corresponding access ranges it will use. It lets the runtime automatically schedule the task on a NUMA node that contains the specified memory regions, thereby improving data locality and overall performance. To further improve locality in iterative algorithms, Lecture 10 covers the design and implementation of a low-overhead trace/replay mechanism for **`async`** tasks [35], ensuring that each worker consistently executes the same set of **`async`** tasks across each successive iteration of the task-parallel compute kernel.

3.3 System-level Concurrency Mechanisms

Lectures 11–12 and 14–17 cover low-level parallel programming and runtime mechanisms that extend beyond task-based parallel programming. For example, removing the **`finish`** call at Line 3 in the recursive **`async–finish`** program (Figure 1(a)), and replacing each **`async`** call (Line 5) with C++11 `std::thread T` and the end of the finish scope (Line 9) with a corresponding `T.join()` for each thread, results in nested thread creation instead of lightweight tasks. Such a program can incur heavy overheads and, for larger input sizes, may even fail to terminate because it creates millions of threads. In Lectures 11–12, we demonstrate how to reduce the abovementioned overheads in the modified program using user-level threads from the Boost library [1, 26] instead of kernel-level threads (Pthreads). Further, when writing lock-free code in C/C++, it is essential to ensure correct memory ordering to prevent unexpected behaviours in parallel programs. These concepts are covered in Lectures 14–15, beginning with an introduction to sequential consistency and the x86 Total Store Order (TSO) memory model. We then discuss the synchronization features introduced in C++11, including mutex locks and atomic operations. Students are then introduced to the C++11 memory model and learn how to implement a lock-free Chase-Lev deque for work-stealing runtime [12]. Finally, we teach how to achieve high performance and improve scalability in parallel programs by automatically minimizing false sharing through runtime-based techniques [36], thereby reducing contention for shared resources.

3.4 Advanced Topics

The debut of the Frontier supercomputer in 2022 marked the beginning of the exascale computing era. As of today, three exascale supercomputers hold the top spot in the Top500 [3], having multiple sockets and many CPU cores per node. These systems include multiple GPUs per node, while each socket has wide SIMD vector units. With hundreds of cores per node and multiple execution units, exascale systems have introduced four key challenges for HPC [44], such as: a) scalable runtimes to manage massive number of threads, b) reducing memory access latency and ensuring locality over deep memory hierarchies, c) achieving energy efficiency within a 20–30 MW/exaflops power envelope [37], and d) fault detection and recovery mechanisms to address increased failure rates. Lectures grouped under the first two modules (Sections 3.1 and 3.2) addressed challenges related to scalable runtimes and locality awareness, whereas this module of the PRMP course focuses on topics targeting energy efficiency and resilience (Lectures 18–25).

Lecture 18 focuses on techniques for achieving energy efficiency in runtime systems, including dynamic concurrency throttling [48], as well as processor frequency control using Dynamic Voltage and Frequency Scaling (DVFS) and Uncore Frequency Scaling (UFS) [27], by making use of Linux hardware performance counters. Lecture 19 covers writing SIMD-parallel programs for the vector units available on multicore processors. We use the C++17 Vector Class Library [15], a header-only implementation that avoids compiler intrinsics and assembly language to ease the programming. Students also learn hybrid parallel programming using multicore and vector units concurrently. Lectures 20–21 expand the scope of heterogeneous programming by introducing hybrid task parallelism across CPUs and GPUs. Students learn how to extend work-stealing runtime systems to support dynamic load balancing between multicore CPUs and GPUs. We use C++ Boost.Compute library [38] for GPU programming to avoid focusing on any specific GPU. It is also a header-only implementation that integrates seamlessly with the C++11 lambda function-based APIs students use throughout this course. Since access to discrete GPUs is limited at several educational institutions, we encourage the students to experiment using the integrated Graphics Card (iGPU), widely available on desktop/laptop processors. Finally, before starting the student-led seminar series, we briefly introduce resilience in Lecture 22, where students learn how to extend the regular **`async–finish`** programming for some widely used fault tolerance techniques [40]. The student-led research seminars in Lectures 23–25 aim to improve student's presentation skills and inspire them to pursue research. Students present the evaluation of their project implementation and summarize a recently published conference or journal paper of their choice, related to topics covered in the PRMP course.

4 Course Evaluation

The PRMP course includes the following evaluation components, along with their respective weightage.

(1) Quizzes (10%)
(2) Midterm written exam (20%)

Milestone	Description	Total Marks (out of 100)	Deadline (Weeks)
1	C++11 lambda based **async–finish** implementation with a light-weight work-stealing runtime	10	2
2	Minimizing the sequential overheads in work-stealing runtime developed in Milestone-1	10	1.5
3	NUMA-aware hierarchical work-stealing for regular DAG	8	1
4	Improving locality of **async** tasks in iterative computation using trace and replay	10	1.5
5	Achieving energy efficiency by dynamic controlling worker count	6	1
6	Summarizing a research paper and experimental evaluation of Milestones 1–5 on a NUMA server	6	4

Table 2: Short summary of project milestones in PRMP course

(3) Endterm written exam (20%)
(4) Group project (50%)

4.1 Proctored Exams

Quizzes, the midterm, and the end-term exams are the three proctored evaluation components in the PRMP course. Quizzes are conducted during lecture hours, towards the end of a class session (roughly 20 minutes). They are scheduled approximately every two weeks, covering topics from the previous two weeks. We inform the students in advance about the quiz schedule. Each quiz includes multiple-choice questions (MCQs), fill-in-the-blank questions, and partially written code that students are required to complete. The midterm and endterm exams are one and two hours long, respectively, and include theoretical and programming-related questions.

4.2 Group Project

The course project in PRMP has six milestones and runs throughout the semester, with deadlines closely aligning with the lecture topics. Each milestone focuses on runtime optimization and performance evaluation. Due to the programming-intensive nature of the project, which focuses on enhancing **async–finish** programming implementations for modern processors, we follow a pair programming approach where up to two students are allowed to form a group and submit the project jointly. Table 2 summarizes the PRMP project milestones. We use a deadline chaining approach, where each milestone builds on the previous one. It reinforces learning and helps students improve their resumes by demonstrating experience with a large-scale project. Students use their laptops or desktops to develop and test Milestones 1–5.

4.2.1 Milestone-1. The first milestone is adapted from the FPP course and aims to deepen the student's understanding of the design and implementation of the **async–finish** style programming model. We provide skeleton code for a work-stealing runtime, which they are required to implement and test by the deadline. To reduce concurrency-related bugs, we ask them to support only a flat **finish** construct and to use mutex locks for implementing the pop and steal APIs in the work-stealing runtime. We also provide recursive task-parallel programs, such as the N-Queens problem, to help them test their implementation. This project milestone is released immediately after the completion of Lecture 5.

4.2.2 Milestone-2. In this milestone, students are given the option to either implement dynamic task aggregation to reduce the task creation overheads associated with fine-grained **async**, or implement private-deque-based work-stealing to minimize the overheads arising from concurrent deque accesses (see Lectures 6–7 in Section 3.2). Students build upon their Milestone-1 implementation and evaluate the resulting performance improvements. Additionally, a bonus of 2 marks is awarded to those who implement both optimizations for minimizing the sequential overheads. This milestone is released three weeks after the first milestone is announced.

```
1 template <typename T>
2 void prmp::numa_parallel_for(int low, int high, T&& lambda_loopBody);
```

Figure 2: A NUMA-aware API in Milestone-3 that internally uses divide-and-conquer style recursive task parallelism

4.2.3 Milestone-3. This milestone builds upon the concepts taught in Lectures 8–9 and is released immediately after the midterm exams. Students extend their Milestone-2 implementation by introducing support for NUMA architectures. They implement a `numa_parallel_for` API (Figure 2) that performs block-cyclic division of loop iterations, where each block is executed on a separate NUMA node. Each block then uses a divide-and-conquer style of recursive task parallelism. To exploit locality, students implement a basic hierarchical work-stealing runtime in which worker threads first attempt to steal tasks from other workers within the same NUMA domain, and carry out limited remote steals only upon failing locally. Students use the Linux `libnuma` library to perform block-cyclic memory allocation across NUMA nodes.

```
1  bool replay=false;
2  for(int i=0; i<NUM_ITERS; i++) {
3    if(!replay) prmp::trace_asyncs();
4    else prmp::replay_asyncs();
5    /*
6     * The compute_kernel is an iterative style
7     * computation such as iterative averaging.
8     * It contains async-finish parallelism.
9     */
10   compute_kernel();
11   if(!replay) {
12     prmp::stop_tracing();
13     replay = true;
14   } else prmp::stop_replay();
15 }
```

Figure 3: Improving locality in Milestone-4 using trace/replay

4.2.4 Milestone-4. This milestone focuses on further improving the data locality for iterative algorithms on NUMA architectures

using a trace/replay enabled work-stealing runtime (see Lecture 10 description in Section 3.2). Students implement the trace/replay feature on top of their Milestone-1 deliverable. A task-parallel implementation of an iterative algorithm using the `async-finish` model is provided as a testcase that uses the trace/replay feature as shown in Figure 3. In this milestone, students use the trace/replay mechanism to improve task placement and execution locality. Additionally, students receive a bonus of 1 mark if they successfully integrate the implementations of Milestones 3 and 4. This milestone is released one week after the deadline of Milestone-3.

4.2.5 Milestone-5. This milestone focuses on achieving energy efficiency in the work-stealing runtime by dynamically controlling the number of active worker threads. Students use hardware performance counters to measure energy consumption and instructions executed to decide the number of active threads at any instant during program execution. Since this milestone is announced toward the end of the semester, we provide pseudocode for implementing dynamic thread throttling to ease the student's workload. It also gives them sufficient time to revisit previous milestones and prepare for Milestone-6. Milestone-5 is released one week after the deadline for Milestone-4.

4.2.6 Milestone-6. For evaluating Milestones 1–5, we mainly check whether each student's implementation is correct and runs successfully on the provided test cases using their laptop or desktop. We evaluate each milestone immediately after its deadline is over. However, since student laptops/desktops have a limited number of cores and lack NUMA support, they cannot evaluate the actual performance of their implementation locally. Hence, we provide students access to a dedicated multi-socket, multicore NUMA server for the performance evaluation of Milestones 1–5. This server is not time-shared, and students must reserve access in advance through a reservation system. Milestone-6 includes two deliverables: a) students have to submit performance results obtained by running their Milestones 1–5 implementations on the NUMA server using a set of benchmarks we provide, and b) they also need to submit a summary slide deck of a recently published conference or journal paper (of their choice) relevant to the topics covered in the PRMP course. Students present their experimental evaluation and research paper summary during the student seminar sessions (Lectures 23–25). As students need sufficient time to prepare for this milestone, we release Milestone-6 immediately after the midterm examinations, along with Milestone-3. Its submission deadline is immediately after the Lecture 22 delivery date.

4.3 Grading Summary

Figure 4 presents the geometric mean of marks obtained in each evaluation component across the three offerings of PRMP. Overall, students performed best in the project component, followed by the quizzes, the midterm and the endterm exams. The pair programming-based project structure enables students to clarify doubts and debug code more effectively, resulting in better scores. Quizzes are intentionally kept easy, as their primary objective is to encourage students to attend lectures.

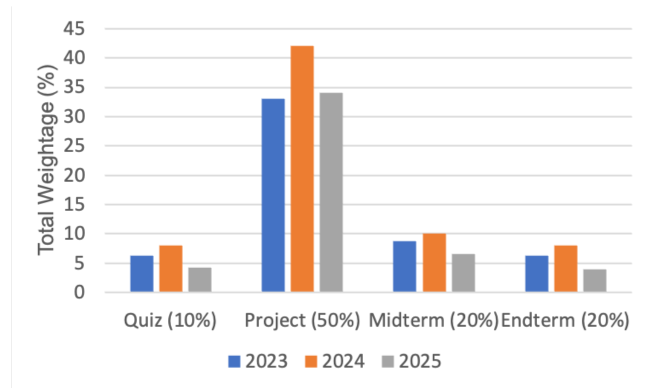

Figure 4: PRMP batch-wise mark distribution (Geomean)

Year	Undergrad	Postgrad	Total Students
2023	13	3	16
2024	17	1	18
2025	21	2	23

Table 3: Distribution of PRMP students across three offerings

5 Student Feedback

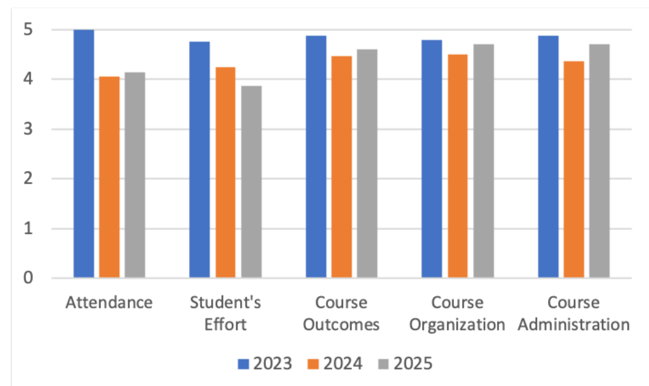

Figure 5: Student feedback on a scale of 1–5 (higher the better)

Table 3 presents the student demographics, and Figure 5 summarizes the student feedback across the three offerings of the PRMP course. Students receive an anonymous feedback form from the IIIT Delhi towards the end of each course, where they have to provide a rating on a scale of 1–5 for various questions related to the course. Higher scores indicate a more positive response and stronger agreement from the student. The course feedback summary for the past three offerings of PRMP is shown in Figure 5. The percentage of students who submitted the feedback form across the three offerings was 50%, 94%, and 91%, respectively. We can observe that most students agreed that they regularly attended the lectures. They had to put more effort due to the programming-heavy nature of the course project, which also has significant weightage (50%). The

effort reported in 2023 was highest, as students of this batch implemented their project milestones using Argobots runtime instead of the 2024 and 2025 batch, who used their own custom work-stealing runtime (discussed in Section 6.2). A high rating for course outcomes, course organization, and course administration indicates that students could easily follow the course by regularly attending the lectures. There is no textbook in the PRMP course, but we provide relevant online materials (e.g., research papers and tutorials) in each lecture, which the students read for deeper understanding.

6 Challenges and Lessons Learned

6.1 Lecture Layout

The PRMP course project has a significant weightage, and Milestones 1–5 are based on concepts covered in the lectures. Hence, completing the relevant lectures before releasing the corresponding project milestones is essential. Due to this reason, except for Lecture 18 (for Milestone-5), all other relevant lectures are covered before the start of the midterm examination.

6.2 Use of Argobots Runtime

In the first offering of the PRMP course, we taught Argobots [45], a lightweight, low-level threading and tasking framework, along with Boost.Fibre in Lecture 12. We covered this lecture content in the first two weeks of that offering and provided an option for students to implement Milestone-1 using the Argobots backend. We floated an anonymous form for students to share their feedback in the middle of the course (separate from the course feedback form shared by IIIT-Delhi). Students reported that they enjoyed working with the Argobots option as it exposed them to an existing low-level runtime. However, it required more effort than implementing a custom work-stealing runtime from scratch. Hence, we provided bonus marks to the groups who successfully implemented their Milestone-1 using Argobots. Later, we realized that grading the Argobots-based solutions was challenging for the Teaching Assistant. Hence, we removed this option from the 2024 offering onwards.

6.3 Inclusion of Labs

While the project milestones require students to apply the concepts learned from several lectures covered before the midterm examination, Lectures 11 and onwards do not contribute to any project deadlines (except Lecture 18). We have observed that, for this reason, student performance tends to drop in the end-term exam and quizzes conducted during the second half of the course. This trend is also evident from the marks distribution for the end-term exam compared to the midterm exam in Figure 4. In future offerings of PRMP, we plan to include mandatory lab components in the second half of the course with 10% weightage and reduce the course project's weightage by the same amount. These labs will cover concepts such as user-level threads using the Boost.Fiber library, achieving synchronization using atomic variables, applications of C++11 memory models, SIMD programming using Vector Class Library, and GPU programming using the Boost.Compute library.

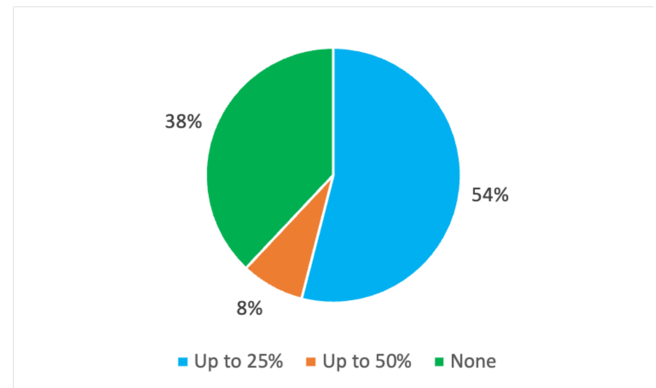

Figure 6: Student-reported AI tool usage (course project)

6.4 Use of AI Tools for PRMP Milestones

We request that students avoid open-sourcing their project implementation after completing the PRMP course. However, with the advent of LLMs such as ChatGPT, students increasingly rely on AI-generated code for programming-related evaluation components. We asked students from the last two offerings of the PRMP course (41 students) to anonymously share with us the amount of help they took from ChatGPT (and related AI tools) for solving PRMP project milestones. We received responses from 24 students. Their responses are summarised in Figure 6. We can observe that only 8% of the students reported taking help in more than 25% of their submitted code. However, as LLMs continue to advance, we plan to allow students to seek help from LLMs with mandatory disclosure of the AI-assisted code in the future. To conduct a fair evaluation in such cases, apart from including an exhaustive evaluation rubric for each milestone, we also plan to interview students after each deadline to assess their understanding of the code they submitted.

7 Conclusions

This paper outlines the structure and methodology of the Parallel Runtimes for Modern Processors (PRMP) course being offered at IIIT Delhi. We discussed the project-oriented course design that introduces task-based parallel programming, providing students with a deeper understanding of the underlying parallel runtime systems. We anticipate that this course will positively influence the teaching of parallel programming and runtime systems at other universities.

Acknowledgments

The author is grateful to the anonymous reviewers for their suggestions on improving the presentation of the paper. ChatGPT was used only for improving grammar and spelling and for minor language polishing of the author's original text in this paper.

References

[1] [n. d.]. Boost.Fiber. https://www.boost.org/doc/libs/release/libs/fiber/. Accessed 2025.
[2] 2014. COMP 322: Fundamentals of Parallel Programming. https://wiki.rice.edu/confluence/display/PARPROG/COMP322

[3] June 2025. TOP500. https://top500.org/lists/top500/2025/06/
[4] Umut A. Acar, Arthur Chargueraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private deques. In *PPoPP'13*. 219–228. doi:10.1145/2442516.2442538
[5] Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A black-box approach to energy-aware scheduling on integrated CPU-GPU systems. In *CGO'16*. 70–81. doi:10.1145/2854038.2854052
[6] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *P3HPC'19*. 71–81. doi:10.1109/P3HPC49587.2019.00012
[7] Benjamin S Bloom, MD Engelhart, EJ Furst, WH Hill, and DR Krathwohl. 1956. Taxonomy of educational objectives: Theclassification of educational goals. *Handbook I: Cognitive domain. New York: David McKay Company* (1956).
[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. In *PPoPP'95*. 207–216. doi:10.1145/209936.209958
[9] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos. *J. Parallel Distrib. Comput.* 74, 12 (2014), 3202–3216. doi:10.1016/j.jpdc.2014.07.003
[10] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.
[11] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *OOPSLA'05*. 519–538. doi:10.1145/1094811.1094852
[12] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA'05*. 21–28. doi:10.1145/1073970.1073974
[13] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. doi:10.1109/99.660313
[14] Alejandro Duran, Julita Corbalan, and Eduard Ayguade. 2008. An adaptive cut-off for task parallelism. In *SC'08*. 1–11. doi:10.1109/SC.2008.5213927
[15] Agner Fog. [n. d.]. Vector Class Library. https://github.com/vectorclass/version2. Accessed 2025.
[16] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *PLDI'98*. 212–223. doi:10.1145/277650.277725
[17] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. 2003. *Introduction to parallel computing*. Pearson Education.
[18] Dan Grossman and Ruth E. Anderson. 2012. Introducing parallelism and concurrency in the data structures course. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. 505–510. doi:10.1145/2157136.2157285
[19] Max Grossman, Maha Aziz, Heng Chi, Anant Tibrewal, Shams Imam, and Vivek Sarkar. 2017. Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level. *J. Parallel and Distrib. Comput.* 105 (2017), 18–30. doi:10.1016/j.jpdc.2016.12.026 Keeping up with Technology: Teaching Parallel, Distributed and High-Performance Computing.
[20] Max Grossman, Vivek Kumar, Nick Vrvilo, Zoran Budimlic, and Vivek Sarkar. 2017. A pluggable framework for composable HPC scheduling libraries. In *IPDPSW'17*. 723–732. doi:10.1109/IPDPSW.2017.13
[21] Max Grossman, Jun Shirako, and Vivek Sarkar. 2016. OpenMP as a High-Level Specification Language for Parallelism. In *OpenMP: Memory, Devices, and Tasks*. Springer International Publishing, 141–155. doi:10.1007/978-3-319-45550-1_11
[22] Shams Imam and Vivek Sarkar. 2014. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *PPPJ'14*. 75–86. doi:10.1145/2647508.2647514
[23] Shintaro Iwasaki and Kenjiro Taura. 2016. A static cut-off for task parallel programs. In *PACT'16*. 139–150. doi:10.1145/2967938.2967968
[24] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *PGAS'14*. Article 6, 11 pages. doi:10.1145/2676870.2676883
[25] Tim Kaler, Xuhao Chen, Brian Wheatman, Dorothy Curtis, Bruce Hoppe, Tao B. Schardl, and Charles E. Leiserson. 2024. Speedcode: Software Performance Engineering Education via the Coding of Didactic Exercises. In *IPDPSW'24*. 391–394. doi:10.1109/IPDPSW63119.2024.00087
[26] Oliver Kowalke. [n. d.]. Boost.Context. https://www.boost.org/doc/libs/release/libs/context/. Accessed 2025.
[27] Sunil Kumar, Akshat Gupta, Vivek Kumar, and Sridutt Bhalachandra. 2021. Cuttlefish: Library for Achieving Energy Efficiency in Multicore Parallel Programs. In *SC '21*. Article 81, 14 pages. doi:10.1145/3458817.3476163
[28] Vivek Kumar. 2020. PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks. In *HiPC'20*. 251–260. doi:10.1109/HiPC50609.2020.00039
[29] Vivek Kumar. 2021. Teaching High Productivity and High Performance in an Introductory Parallel Programming Course. In *HiPCW'21*. 21–28. doi:10.1109/HiPCW54834.2021.00010
[30] Vivek Kumar. Accessed 2025. Parallel Runtimes for Modern Processors (PRMP). https://hipec.github.io/courses/cse513.html
[31] Vivek Kumar, Daniel Frampton, Stephen M. Blackburn, David Grove, and Olivier Tardieu. 2012. Work-stealing Without the Baggage. In *OOPSLA'12*. 297–314. doi:10.1145/2398857.2384639
[32] Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. 2014. HabaneroUPC++: A Compiler-free PGAS Library. In *PGAS'14*. Article 5, 10 pages. doi:10.1145/2676870.2676879
[33] Doug Lea. 2000. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*. 36–43. doi:10.1145/337449.337465
[34] I-Ting Angelina Lee. [n. d.]. CSE 539: Concepts in Multicore Computing. https://classes.engineering.wustl.edu/cse539/web/
[35] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2013. Steal Tree: low-overhead tracing of work stealing schedulers. In *PLDI'13*. 507–518. doi:10.1145/2491956.2462193
[36] Tongping Liu and Emery D. Berger. 2011. SHERIFF: precise detection and automatic mitigation of false sharing. In *OOPSLA'11*. 3–18. doi:10.1145/2048066.2048070
[37] LLNL. Accessed 2021. Exascale Computing Project. https://exascale.llnl.gov/
[38] Kyle Lutz. [n. d.]. Boost.Compute. https://www.boost.org/doc/libs/release/libs/compute/. Accessed 2025.
[39] OpenMP ARB. November 2018. *OpenMP Application Programming Interface Version 5.0*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
[40] Sri Raj Paul, Akihiro Hayashi, Nicole Slattengren, Hemanth Kolla, Matthew Whitlock, Seonmyeong Bak, Keita Teranishi, Jackson Mayo, and Vivek Sarkar. 2019. Enabling Resilience in Asynchronous Many-Task Programming Models. In *Euro-Par'19*. 346–360. doi:10.1007/978-3-030-29400-7_25
[41] Jean-Noël Quintin and Frédéric Wagner. 2010. Hierarchical Work-Stealing. In *Euro-Par'10*. 217–229. doi:10.1007/978-3-642-15277-1_21
[42] James Reinders. 2007. *Intel Threading Building Blocks* (first ed.). O'Reilly & Associates, Inc.
[43] Vivek Sarkar, Max Grossman, Zoran Budimlić, and Shams Imam. 2017. Preparing an Online Java Parallel Computing Course. In *IPDPSW'17*. 360–366. doi:10.1109/IPDPSW.2017.162
[44] Vivek Sarkar, William Harrod, and Allan E. Snavely. 2009. Software Challenges in Extreme Scale Systems. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012045. doi:10.1088/1742-6596/180/1/012045
[45] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE TPDS'18* 29, 3 (2018), 512–526. doi:10.1109/TPDS.2017.2766062
[46] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: A High-Productivity Programming Language for HPC with Logical Regions. In *SC'15*. Article 81, 12 pages. doi:10.1145/2807591.2807629
[47] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2014. Adaptive, efficient, parallel execution of parallel programs. In *PLDI'14*. 169–180. doi:10.1145/2594291.2594292
[48] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2014. Adaptive, efficient, parallel execution of parallel programs. In *PLDI'14*. 169–180. doi:10.1145/2594291.2594292
[49] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. 2010. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP'10*. 179–190. doi:10.1145/1693453.1693479
[50] Ana-Lucia Varbanescu, Stephen Nicholas Swatman, and Anuj Pathania. 2023. Performance Engineering for Graduate Students: a View from Amsterdam. In *SC-W'23*. 357–365. doi:10.1145/3624062.3624102
[51] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS'08*. 1–8.