Teaching High Productivity and High Performance in an Introductory Parallel Programming Course

Vivek Kumar
IIIT Delhi, India

***Abstract*—Multicore processors are ubiquitous. Several prior research has emphasized the need for high productivity parallel programming models that require minimal changes to the sequential program and can still deliver high performance using runtimes based approaches on various architectures. In this paper, we present the structure and experience of teaching the Foundations of Parallel Programming course (FPP) at IIIT Delhi using a task-based parallel programming model, Habanero C/C++ Library (`HClib`). FPP covers a wide breadth of topics in parallel programming but emphasizes both high productivity and high performance. It is being offered at IIIT Delhi in the spring semester for undergraduate and postgraduate students since 2017. We describe our novel approach where the students start the learning process using the traditional parallel programming models, discover the underlying limitations, and build runtime solutions to achieve high performance.**

***Index Terms*—Education; Pedagogy; Task Parallelism; async-finish Programming Model; Productivity; Performance; Work-Stealing;**

I. Introduction

Multicore processors are ubiquitous. It is an unavoidable consequence of the breakdown of Dennard scaling, which has put a stop to hardware delivering ever faster sequential performance. Fugaku supercomputer that held on to the top spot in the recent Top500[1] consists of nodes with 48 cores/socket. Unfortunately, software parallelism is often challenging to identify and distribute, which means it is often hard to realize the performance potential of modern processors. Common programming models using threads impose significant complexity to organize code into multiple threads of control and balance work amongst threads to ensure effective utilization of multiple cores. Existing HPC communication models (e.g., MPI [1]) also lack tight integration with multi-threaded programming models (e.g., OpenMP [2]). It is difficult to identify and exploit opportunities for computation-communication overlap in the MPI+OpenMP programming model [3], often requiring overly coarse or error-prone synchronization between the communication and multi-threaded components of applications.

The twin challenge of achieving both high productivity and high performance has helped the evolution of several Task-based Parallel Programming Models (henceforth mentioned as TPPM), such as Cilk [4], Java fork/join [5], Intel TBB [6], Qthreads [7], Habanero-Java library [8], Habanero-C/C++ library [9], X10 [10], Chapel [11], Legion [12], Habanero-UPC++ [13], Kokkos [14], Raja [15], HPX [16], and AsyncSHMEM [17]. TPPM improves the programmer's productivity, i.e., minimizes the effort required to convert a serial application into a parallel application by offering a high-level approach for introducing parallelism. TPPM supports the creation of large numbers of lightweight tasks that can execute in parallel over worker threads bound to processor cores. While such programming models present a much higher-level and more intuitive method for designing parallel algorithms, especially for irregular and dynamic problems, they rely on a work-stealing runtime [18] to deliver high performance by dynamic load balancing of the tasks.

OpenMP and MPI have maintained their role in teaching courses on parallel computing. However, with the abundance of novel TPPM, TPPM has also made inroads in the sphere of parallel computing courses. COMP322 from Rice University [19], [20] is one such course in this space that uses Habanero-Java library (`HJlib`) for teaching the intellectual challenges in parallel software by abstracting away the low-level details of different parallel systems. This course is also offered online on Coursera [21]. COMP322 uses `HJlib` as a tool for teaching fundamental concepts in parallelism such as parallel algorithms, dynamic task parallelism, critical path length, parallel performance metrics, functional parallelism with futures, loop parallelism, data races, deadlocks, map-reduce, data-driven tasks, tasks with locality hints, etc. It also covers the basics of MPI using OpenMPI Java [22]. Students are graded in COMP322 based on homework assignments, two exams, lab exercises, quizzes, and in-class worksheets. The students can self-evaluate the homework and labs using a web-based tool, Habanero Autograder [23].

COMP322 covers a wide breadth of topics in parallel programming with a prime focus on high productivity. However, it briefly touches upon runtime techniques for achieving high performance. Also, although Java is a high-level programming language, and it allows the portable execution of a single code base across many different student laptops, it is still not a mainstream programming language in the HPC community. Apart from Fortran, C/C++ is the widely-used programming language in HPC. It is evident from the fact that a majority of the TPPM only supports C/C++ language [4], [6], [7], [9], [12], [13], [14], [15], [16], [17].

This paper presents the design and implementation of an introductory parallel programming course, Foundations of Parallel Programming (FPP) [24], offered at IIIT Delhi for the past five years for undergraduate and postgraduate students. FPP builds upon COMP322 but focuses on both productivity and performance. It uses `HClib` [9] as a teaching tool. `HClib`

[1]https://www.top500.org/lists/top500/2021/06/

```
1 int x, y;
2 finish([&]() {
3   async([&]() {
4     x = foo(n); //Task-1
5   });
6   y = bar(n);   //Task-2
7 });
8 int z = baz(x+y); //Task-3
```

Fig. 1. An example code schema with **async** and **finish**

```
1 future_t<int>* f1 = async_future([=]() {
2     return foo(n); //Task-1
3 });
4 int y = bar(n);  //Task-2
5 int z = baz(f1->get() + y); //Task-3
```

Fig. 2. An example code schema with futures

is a compiler-free C/C++ library that supports dynamic task parallelism using C++11 lambda functions. FPP uses chaining of assignments and course projects that teach students how to build their own TPPM with an underlying high performance runtime from scratch.

In the following sections, we detail several aspects of the FPP course. We start our discussion with an introduction to HClib in Section II. Section III presents the teaching methodology and course topics. Section IV details the student distribution in past offerings of FPP. Section V describes the graded components. Section VI presents a summary of the student feedback. Section VII lists out our experiences teaching this course. Finally, Section VIII presents the conclusion.

II. HClib

This section provides a brief overview of the Habanero-C/C++ library (HClib).

A. Support for multicore parallelism

HClib offers an **async–finish** programming model for exploiting shared memory parallelism. These constructs were first coined by the X10 language [10]. Now it has been adopted by several other frameworks supporting task parallelism [8], [25]. HClib is open-sourced [26] and was developed at Rice University. HClib is a native library-based implementation of the Habanero programming model that offers C and C++ APIs. It provides high productivity in writing **async–finish** programs by using C++11 lambda functions in all its asynchronous APIs. C++11 lambdas avoid the need for compiler support while still retaining the syntactic convenience of language-based approaches.

Figure 1 shows a sample code written by using **async–finish** APIs supported by HClib. The **async** API creates Task-1, which can run in parallel with the following statements, i.e., Task-2. An **async** is a powerful primitive because we can use it to enable any statement to execute as a parallel task, including statement blocks, **for**-loop iterations, and function calls. The **finish** is a generalized join operation. Task-3 will never execute until both Task-1 and Task-2 have been completed. The power of these

```
1 promise_t<int> * p1 = new promise_t<int>();
2 async([=]() {
3   int x = foo(n); //Task-1
4   p1->put(x);
5 });
6 int y = bar(n); //Task-2
7 promise_t<int> * p2= new promise_t<int>();
8 async_await([=] { //Task-3
9   int z = baz(p1->get_future()->get() + y);
10  p2->put(z);
11  delete(p1);
12 }, p1->get_future());
13 ....
14 foobar(p2->get_future()->get()); //Task-4
15 delete(p2);
```

Fig. 3. An example code schema with data-driven tasks

```
1 int array[x][y];
2 loop_domain_t loop={0, x, 1, tile};
3 forasync1D(&loop, [=](int i) {
4   foo(array[i]);
5 });
```

Fig. 4. An example code schema with loop-level parallelism

constructs comes from the ability to nest **async** and **finish** arbitrarily. Figure 2 shows the same program but written using **async_future** task. The program flow is the same as in Figure 1. However, unlike the **async**, **async_future** can return values. The **get** on the future object (f1) will block until the **async_future** returns. Figure 3 is another variation of the same program that uses futures, promises and data-driven tasks (**async_await**). The **async_await** blocks on the future p1->**get_future**(), and will be scheduled only after p1->**put**() inside Task-1. Task-4 would block on the execution of p2->**get_future**()->**get**(), until p2->**put**() is called by Task-3.

forasync1D is an API supported by HClib for loop-level parallelism. It recursively divides a **for** loop iterations into two halves that can execute in parallel. Figure 4 shows a sample program written using **forasync1D** and **finish**. Programmer can specify the tile for controlling the task granularity in **forasync1D**. Figure 5 demonstrates locality hints in asynchronous tasks (**async_at_hpt**) to improve the temporal locality for array accesses during

```
1 int array[x][y];
2 int numPl=get_num_places(place_type_t::CACHE_PLACE);
3 place_t ** cachePl = new place_t*[numPl];
4 get_places(cachePl, place_type_t::CACHE_PLACE);
5 for (int iter=0; iter<X; iter++) {
6   finish([=]() {
7     for(int i=0; i<x; i++) {
8       int pl=(i%numPl);
9       async_at_hpt(cachePlaces[pl], [=]() {
10        foo(array[i]);
11      });
12    }
13  });
14 }
15 delete(cachePl);
```

Fig. 5. An example code schema for tasks with locality hints

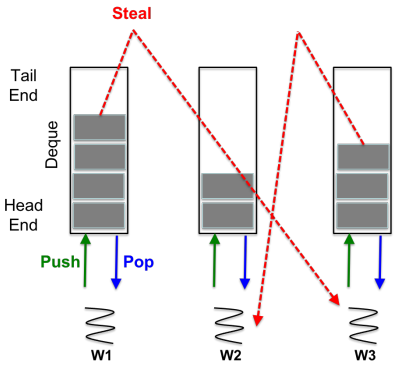

Fig. 6. Work-stealing for load balancing of **async–finish** program

```
1  finish_spmd ([capture_list]() {
2    /* local asynchronous tasks */
3    async(...);
4    async_future(...);
5    async_await(...);
6    async_at_hpt(...);
7    forasync(...);
8    /* remote asynchronous tasks */
9    async_copy(...);
10   async_at(...);
11 });
```

Fig. 7. Asynchronous calls in HabaneroUPC++

parallel loop execution. The **async_at_hpt** API creates an asynchronous task that is scheduled on a fixed worker thread bounded to a core of the processor [27]. Iterative execution of each task mapped to specific workers, thereby improve the temporal locality for array accesses. HClib also has experimental support for object-based isolation [28]. It provides an API isolated(o1, o2, ..., lambda) that allows the programmers to specify the list of objects for which isolation is required during the execution of the supplied lambda function. Any other TPPM except HJlib does not support this feature.

B. Work-stealing for high performance

HClib internally uses a work-stealing runtime for dynamic load-balancing of **async–finish** tasks. Figure 6 shows the implementation of a work-stealing scheduler. It schedules work exposed by the programmer, exploiting idle processors and unburdening those that are overloaded. In work-stealing, the worker (victim) executing the **async** in Figure 1 would push Task-1 on its deque. It will then execute Task-2. After completing the execution of Task-2, the worker will reach the end **finish** scope, where it will try to pop Task-1 from its deque. If any thief stole this task, the victim would become a thief. Otherwise, it will pop and execute Task-1 and then exit the **finish** scope.

C. Support for distributed parallelism

HClib is integrated with various HPC communication libraries for supporting task parallelism at distributed scale. HabaneroUPC++ is one such integration of HClib with UPC++ [29]. It is a compiler-free PGAS library that supports a tighter integration of intra-place and inter-place parallelism than standard hybrid programming approaches [13]. The HabaneroUPC++ library implementation is based on tight integration of the UPC++ library and HClib, with new extensions to support the integration. Figure 7 shows the APIs supported by HabaneroUPC++. It integrates the benefit of HClib's APIs in UPC++. Like UPC++, the HabaneroUPC++ program also starts in an SPMD fashion, with each place getting a copy of the main function.

III. Teaching Methodology and Course Topics

We have designed the FPP course, assuming that the students registering for this course will not have any prior experience with parallel programming. Although, they might have written a simple program using Pthread APIs in their operating systems course at the undergrad level. However, we expect the students to have familiarity with any programming language (Java / C / C++). Broadly, lecture contents in FPP can be divided into three categories: a) multicore parallel programming (70%), b) distributed parallel programming (20%), and c) student-led research seminars (10%). These topics, multicore and distributed parallel programming, first introduce the students to the traditional approaches for parallel programming, highlights the limitations in terms of productivity and performance, and teaches task parallelism and runtime techniques to overcome these shortcomings. Student seminars are designed to nurture research interest and improve the presentation skills of the students. As the access to the HPC cluster is usually limited at several institutions, we give more emphasis to multicore parallel programming in this course. Due to this reason, we only cover the basics of distributed parallel programming in FPP. It helps the students to rely on their laptops (with the multicore processor) to cover the entire course content.

Table III shows the details of lecture topics in the FPP course. Several lecture topics in FPP related to task parallelism in multicore parallel programming are borrowed from the COMP322 course, but FPP uses HClib as a teaching tool instead of HJlib in COMP322. FPP also borrows a few other topics from other courses and online materials [32], [30], [33], [31]. However, giving importance to both productivity and performance is the key differentiating factor in the FPP course. Using HClib as a teaching tool in FPP further helps the students understand *how* to achieve high productivity in parallel programming and *what* happens behind the scenes to deliver high performance.

A. Topics on multicore parallel programming

FPP course starts with a refresher on Pthread programming, reminding students that although Pthread programming is now not mainstream, it is still the building block of all the TPPM. FPP uses Pthread programs to demonstrate how difficult it is to achieve productivity and performance using Pthreads [30]. After this, we introduce them to **async–finish** programming in HClib and its support for serial elision. We then

Lecture topics	Content	Focus
Introduction to parallel programming	Why multicore processors, refresher on Pthread programming [30], concurrency decomposition [31]	Productivity
Dynamic task parallelism [19]	async-finish programming model, serial elision, computation graphs, ideal parallelism, greedy scheduling of computation graphs on fixed number of cores	Productivity
Design and implementation of thread pools	Mapping async tasks to library-based thread pool runtime, work-sharing and work-stealing scheduling, work-first and help-first work-stealing	Performance
Loop level parallelism [19]	Loop parallelism using forasync, divide-and-conquer parallelism and its impact on critical path	Productivity
Locality aware task parallelism	Assigning task affinity using async_at_hpt, design of hierarchical work-stealing, false sharing	Productivity and performance
Mutual exclusion	Object-based isolation in HClib async-finish program	Productivity and performance
Functional parallelism [19]	Futures, promises, data driven tasks using HClib	Productivity
Cilk language and runtime [32]	Terminally strict computation in async-finish v/s fully strict computation in Cilk, cactus stack abstraction, inlet and abort	Productivity
OpenMP parallel programming model [32], [33]	Work-sharing constructs, task directives, data scoping, comparison with async-finish programming model	Productivity
Message Passing Interface [32]	Point to point communication, collective communication, hybrid parallelism using MPI+OpenMP	Productivity
Distributed task parallelism	Challenges with MPI+OpenMP programming model, PGAS programming using HabaneroUPC++, communication and computation workers in work-stealing, distributed work-stealing	Productivity and performance
Research seminars	Student led seminars on project work	Performance

TABLE I
DETAILED DESCRIPTION OF LECTURE TOPICS IN FPP ALONG WITH THEIR FOCUS AREA

train them with the computation graphs. Students can use a tool developed at IIIT Delhi to improve their understanding by self-generating the computation graphs of **`async`–`finish`** programs are written using `HClib`. Before introducing them to any other tasking techniques, we discuss the design and implementation of thread pools. We teach them how we can map **`async`** tasks generated by the programmer to a thread pool implementation. We discuss the implementation of both work-stealing and work-sharing based thread pools and their merits and demerits. After this, we teach them the rest of the tasking techniques for multicore parallelism (**`forasync1D`**, **`async_future`**, **`async_await`**, and **`async_at_hpt`**). Introducing these topics after teaching thread pool helps them understand how the runtime handles these tasks internally. As locality aware tasks (**`async_at_hpt`**) are handled differently by the thread pool, we also teach them the design of hierarchical work-stealing runtime in `HClib`. We introduce them with mutual exclusion by using Pthread mutex locks and demonstrate how difficult it is to avoid deadlocks in **`async`–`finish`** programs using mutex locks. We use the experimental support for object-based isolation in `HClib` to teach how productivity and performance go hand in hand. We conclude the topics on multicore parallel programming by introducing the students with Cilk and OpenMP parallel programming models and contrasting them with the `HClib`.

B. Topics on distributed parallel programming

We start this module by introducing the students to basic topics in MPI, followed by the MPI+OpenMP hybrid programming model. Here, we teach them the challenges in overlapping computations and communication in MPI+OpenMP. It serves as a motivation to introduce task parallelism even for distributed computing. We use `HabaneroUPC++` to teach the basics of distributed task parallelism in PGAS using APIs such as remote asynchronous tasks (**`async_at`**), remote asynchronous copy (**`async_copy`**), locality free tasks that can participate in distributed work-stealing (`async_any`) [34]. At the end of this module, we also briefly cover the design of a distributed work-stealing runtime.

C. Student seminars

Here, the students present their course project, which is essentially an implementation of a pre-published conference paper related to work-stealing (details in Section V-D). These seminars help improve student's presentation skills by preparing slides and delivering the presentation in a conference-like setting. We also organized a guest lecture from academia in some of the FPP batches to motivate the students further to pursue research.

IV. STUDENTS DISTRIBUTION

FPP course is being offered at IIIT Delhi to undergraduate (B.Tech.) and postgraduate (M.Tech. and Ph.D.) students since 2017 (five batches till now). It was offered in online mode in 2021 due to the Covid pandemic. Figure 8 shows the student distribution over the past five batches. Postgraduate students primarily consist of M.Tech students, and there has been at most single Ph.D student in any batch. We saw fewer registrations in 2021 (online mode), and it was limited to undergraduate students. In its existing form, FPP is a programming heavy course and is designed primarily to be offered in classroom settings (offline mode). It has a project component (Section V) where students work in pairs. Although FPP has consistently received positive responses from the

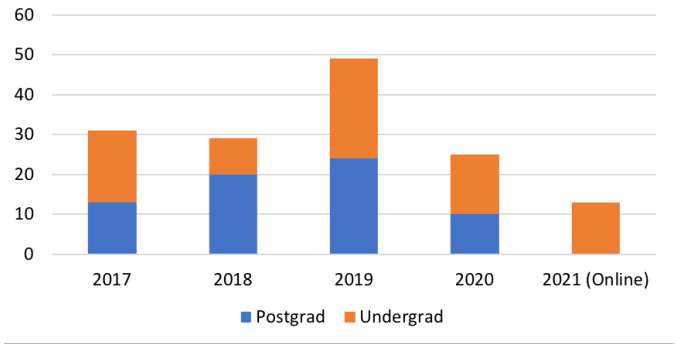

Fig. 8. Student distribution over the years

students (Section VI), we believe students would face difficulty if they had to take this course in its current form in an online mode. For offering FPP in online mode, we can remove pair programming based course projects and student seminars.

V. Course Evaluation

We follow regular and frequent evaluations in FPP to give students early feedback and judge their level of understanding. FPP has several evaluation components, as mentioned below, along with their weightage.

1) Labs (5%)
2) Quizzes (10%)
3) Take-home assignments (10%)
4) Course project (25%)
5) Midterm written exam (20%)
6) Endterm written exam (30%)

Below sections discusses these components in detail.

A. Labs

Labs in FPP are short programming exercises that the student has to complete within 90 minutes in the presence of TAs. These labs aim to give them hands-on experiences with programming APIs taught in lectures. We conduct a total of seven labs throughout the semester. Topics covered in these labs are as mentioned below.

1) Pthread programming
2) **`async`–`finish`** and **`forasync1D`** using `HClib`
3) Locality using **`async_at_hpt`** in `HClib`, and removing false sharing
4) **`async_future`** tasks in `HClib`
5) Work-sharing pragmas in OpenMP
6) Tasking pragmas in OpenMP
7) Hybrid parallel programming using MPI+OpenMP

In each lab, we provide the student with a partially incomplete program. Students have to add parallelism to this program as per the supplied description. The maximum lines of code they ever write are around 50 (lab topics 1 and 7). The rest of the labs require even less coding. Hence, we give only 5% weightage to labs. We chose the best five labs out of seven for grading, where the student obtained the maximum

```
1 loop_domain_t loop={low, high, stride};
2 void parallel_for(&loop, int numThreads,
3              std::function<void(int)> &&lambda);
```

(a) One dimensional `parallel_for`

```
1 loop_domain_t loop_1={low1, high1, stride1};
2 loop_domain_t loop_2={low2, high2, stride2};
3 void parallel_for(&loop_1, &loop_2, int numThreads
4              std::function<void(int, int)>  &&lambda);
```

(b) Two dimensional `parallel_for`

Fig. 9. StaMp APIs for parallelizing one and two dimensional for loops. These APIs runs the loop body (lambda) in parallel by using 'numThreads' number of Pthreads

marks. They have to write their programs on IIIT Delhi's lab Desktops. Students are not allowed to access the internet during lab time to avoid plagiarism. They have access to the lecture materials during the labs.

B. Quizzes and term exams

Quizzes, midterm, and endterm exams are the three written exams in the FPP course. Each of these is a closed-book exam. FPP has a total of six quizzes, and they happen roughly after every four lectures. Each quiz is of 20 minutes duration and is scheduled during the lecture hours towards the end of the lecture. The quiz consists of multiple-choice questions, fill in the blanks, and reasoning based questions. Midterm and endterm exams together have 50% weightage. These exams include both theoretical and programming related questions.

C. Take-home assignments

The design of assignments and projects is a key factor in differentiating FPP from other similar parallel programming courses. As the motto of FPP is to teach both productivity and performance, we have taken a novel approach to chain assignments and projects in the FPP course. The output of assignment-1 serves as the input of assignment-2. The output of assignment-2 serves as an input for the project implementation.

1) Assignment-1: runtime for Static Mapping of tasks to threads (StaMp): This is the first assignment in the FPP course, and it is released in the second week of the course after covering Pthread programming and introduction to **`async`–`finish`** programming. This assignment aims to introduce the importance of productivity in parallel programming. In this assignment, students are asked to develop a shared library to improve the productivity of the Pthread programmers by abstracting away some of the basic programming efforts. The StaMp is intended to help programmers in parallelizing for-loop based algorithms, such as the addition of two vectors and the multiplication of two matrices. The signature of the linguistic interfaces for exposing in StaMp are as shown in Figure 9. This assignment has a 3% weightage and a deadline of four days.

2) Assignment-2: A light-weight work-stealing runtime for async-finish parallelism (Cotton): This is the second and the last assignment. It is released in the fourth week after teaching the design and implementation of thread pools. It is a group assignment where each group comprises a maximum of two

```
/* cotton APIs are declared here */
#include ``cotton.h''
int main(int argc, char** argv) {
    /* initialize cotton runtime */
    cotton::init_runtime();
    int x;
    /* start a flat-finish scope */
    cotton::start_finish();
    /* spawn an async */
    cotton::async([&]() {
        x = foo(n);
    });
    int y = bar(n);
    /* end the flat-finish scope */
    cotton::end_finish();
    int result = x + y;
    /* release runtime resources */
    cotton::finalize_runtime();
    return 0;
}
```

Fig. 10. Pseudocode of a parallel program written using Cotton

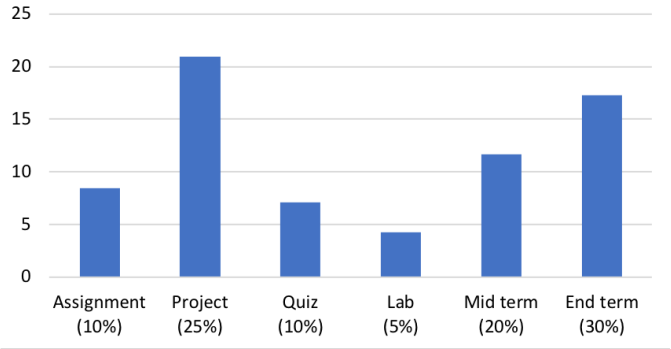

Fig. 11. Average marks distribution

students. This assignment aims to teach both productivity and performance. In this assignment, each group has to implement a library-based new concurrency platform called Cotton. It should support basic **async–finish** based task-parallelism with flat finish scopes. We ask the students to implement the Cotton runtime by reusing the code of StaMp runtime. We don't ask them to support nested finish scopes to avoid complexities, but they should support nesting of **async**. They should implement Cotton as a library-based thread pool implementation that would use work-stealing for dynamic task scheduling. We provide them with a recursive `nqueens` program as a test case that uses Cotton APIs for achieving parallelism. This assignment has 7% weightage and is given a deadline of ten days.

D. Course project

The project in FPP is also a group activity similar to assignment-2. Projects are released immediately after the midterm exams. The projects are aimed to teach runtime techniques for achieving high performance in TPPM. We provide a set of pre-published conference papers related to work-stealing runtimes and allow each group to choose a research paper from these options. Some of the conference papers that we have given in past as FPP course projects are mentioned here: [35], [36], [37], [38], [39], [40]. The course project has three milestones, spanning across the entire semester post-midterm exam.

1) Milestone-1: In this milestone, each group has to demonstrate their understanding of the project by submitting a PowerPoint presentation. These slides have to be prepared so that the student group is the original author of that paper, and they have to use these slides for a conference presentation of 30 minutes. This milestone is of 5% weightage and is given a deadline of two weeks. We choose one group in each project category with the best slides and ask them to present their slides as a student seminar during the lecture. Some extra marks are awarded to these groups presenting the slides.

2) Milestone-2: Each group has to turn in their in-progress project implementation for this milestone. Students are asked to implement their project in their Cotton runtime implementation (Section V-C2). This milestone is of 5% weightage and is given a deadline of three weeks. Here, we judge the in-progress implementation based on: a) designing all required data structures and b) identifying all the required methods and their interaction in work-stealing runtime.

3) Milestone-3: This is the last milestone where each group has to submit the fully working implementation of their project along with a report that: a) demonstrates the experimental analysis of their project as per the original paper, but by using a set of benchmarks provided by us, b) details the challenges faced by the students during the implementation (if any), and c) any novel research ideas. This milestone is of 15% weightage and is given a deadline of four weeks.

E. Grading summary

Figure 11 shows the average marks obtained by students in each evaluation component over the years. This graph does not include the data for the years 2017 and 2021. As FPP was a newcomer in 2017, we kept the course components relatively more straightforward and hence did not have labs, projects, and assignments on StaMp and Cotton. In 2021 we had to drop the project and endterm exam midway due to the raging second Covid wave in Delhi.

From Figure 11, we can observe that the students scored relatively higher in the programming based components. It is because, for 32% weightage, students worked in groups (assignment-2 and project). It helped them in clarifying their doubts and resolving the bugs in their code much more manageable. Lab questions are already designed so that they are easy to score (Section V-A).

VI. Student Feedback

Student feedback is essential for the instructor to understand the shortcomings in the course, improve the course quality, and know if the course met the student's expectations. IIIT Delhi floats an anonymous feedback form to students towards the end of a course, where the students have to provide a

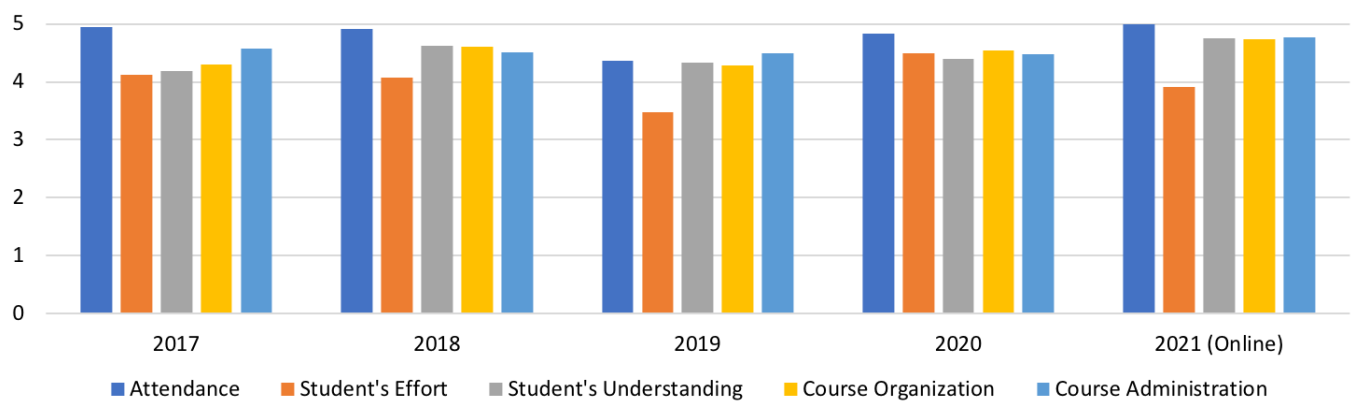

Fig. 12. Student feedback on a scale of 1–5 (higher the better)

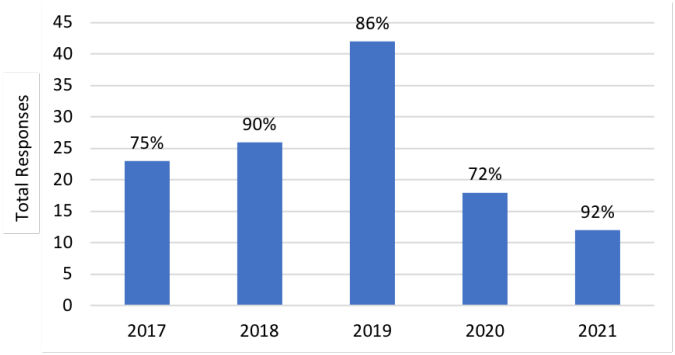

Fig. 13. Total number and percentage of students who participated in the feedback shown in Figure 12

rating on a scale of 1–5 for various questions related to the course. Figure 12 summarises the students feedback in all five offerings of the FPP course, and Figure 13 shows the total percentage of the students who participated in the feedback. We can observe that a majority of the students regularly attended the lectures. As FPP is a programming heavy course, we can observe that the students had to put more effort into this course. Despite this, consistently high ratings for student's understanding, course organization, and course administration demonstrate that FPP successfully achieved its goal of teaching high productivity and high performance in parallel programming.

VII. Discussion

In this section, we discuss the challenges faced while teaching this course. As FPP is a programming heavy course and requires programming in C/C+ on Linux, we found it is essential to provide students with basic training on Unix commands, shell scripting, and debugging C/C++ programs. Hence, at the start of the FPP course, we conduct tutorials on GDB where we teach them how to debug a multithreaded program. We also ask the students to go through this online course material [41]. We also gave tutorials on a version control system (Git) and deducted a few marks during assignment/project demos if we found the student not using version control.

As we use chaining of assignments and projects in FPP, it is crucial to help the students remove all the bugs in their StaMp and Cotton runtimes before using them in the next chain. We found that allowing the students to work in pairs for the assignment on Cotton runtime was helpful as they could resolve most of the bugs on their own. They usually don't face such issues in the first assignment as it is much simpler and straightforward.

VIII. Conclusions

This paper presented the outline and methodology of the Foundations of Parallel Programming (FPP) course taught at IIIT Delhi for the past five years. We discussed that having a blend of topics in productivity and performance differentiates FPP from other existing courses on parallel programming. We presented the novel technique of chaining assignments and projects in the FPP course. It teaches students how to build their own task-based parallel programming model and the underlying load-balancing runtime from scratch. We hope that this course makes an impact in teaching parallel programming at other universities.

Acknowledgments

The author is grateful to the instructors of the COMP322 course for sharing the course materials and to the Habanero team [42], [43] for open-sourcing the HClib library. The author is also thankful to the anonymous reviewers for their suggestions on improving the presentation of the paper.

References

[1] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*, 1995.

[2] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[3] S. Chatterjee, S. Tasırlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 712–725.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1998, pp. 212–223.

[5] D. Lea, "A Java Fork/Join framework," in *Proceedings of the ACM 2000 Conference on Java Grande*, 2000, p. 36–43.

[6] J. Reinders, *Intel Threading Building Blocks*, 1st ed. O'Reilly & Associates, Inc., 2007.

[7] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[8] S. Imam and V. Sarkar, "Habanero-java library: A Java 8 framework for multicore programming," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 2014, pp. 75–86.

[9] M. Grossman, V. Kumar, N. Vrvilo, Z. Budimlic, and V. Sarkar, "A pluggable framework for composable hpc scheduling libraries," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 723–732.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005, pp. 519–538.

[11] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[12] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for hpc with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015.

[13] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, "HabaneroUPC++: A compiler-free PGAS library," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, 2014, pp. 5:1–5:10.

[14] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, p. 3202–3216, 2014.

[15] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[16] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14, 2014.

[17] M. Grossman, V. Kumar, Z. Budimlić, and V. Sarkar, "Integrating asynchronous task parallelism with openshmem," in *Workshop on OpenSHMEM and Related Technologies*. Springer, 2016, pp. 3–17.

[18] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[19] "COMP 322: Fundamentals of Parallel Programming," 2014. [Online]. Available: https://wiki.rice.edu/confluence/display/PARPROG/COMP322

[20] M. Grossman, M. Aziz, H. Chi, A. Tibrewal, S. Imam, and V. Sarkar, "Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level," *Journal of Parallel and Distributed Computing*, vol. 105, pp. 18–30, 2017, keeping up with Technology: Teaching Parallel, Distributed and High-Performance Computing.

[21] V. Sarkar, M. Grossman, Z. Budimlić, and S. Imam, "Preparing an online java parallel computing course," in *IPDPSW '17*, 2017, pp. 360–366.

[26] "HClib." [Online]. Available: https://github.com/habanero-rice/hclib

[22] O. Vega-Gisbert, J. E. Roman, and J. M. Squyres, "Design and implementation of java bindings in open mpi," *Parallel Computing*, vol. 59, pp. 1–20, 2016.

[23] "An autograding framework for parallel JVM programs," 2015. [Online]. Available: https://github.com/agrippa/habanero-autograder

[24] V. Kumar, "Foundations of Parallel Programming." [Online]. Available: https://hipec.github.io/courses/fpp.html

[25] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu, "Work-stealing without the baggage," in *OOPSLA '12*, 2012, pp. 297–314.

[27] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *Languages and Compilers for Parallel Computing*, 2010, pp. 172–187.

[28] J. Zhao, R. Lublinerman, Z. Budimlić, S. Chaudhuri, and V. Sarkar, "Isolation for nested task parallelism," in *OOPSLA '13*, 2013, pp. 571–588.

[29] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS extension for C++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1105–1114.

[30] I.-T. A. Lee, "CSE 539: Concepts in Multicore Computing." [Online]. Available: https://classes.engineering.wustl.edu/cse539/web/

[31] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Pearson Education, 2003.

[32] V. Sarkar, "COMP422: Parallel Computing." [Online]. Available: https://www.cs.rice.edu/~vs3/comp422/

[33] T. Mattson, A. Koniges, C. Breshears, and J. Kemp, "Programming Irregular Applications with OpenMP." [Online]. Available: https://www.nersc.gov/assets/Uploads/SC16-Programming-Irregular-Applications-with-OpenMP.pdf

[34] V. Kumar, K. Murthy, V. Sarkar, and Y. Zheng, "Optimized distributed work-stealing," in *Workshop on Irregular Applications: Architectures and Algorithms (IAAA)*, 2017, pp. 74–77.

[35] S. Shiina and K. Taura, "Almost deterministic work stealing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[36] S. Sridharan, G. Gupta, and G. S. Sohi, "Holistic run-time parallelism management for time and energy efficiency," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, 2013, p. 337–348.

[37] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Steal tree: Low-overhead tracing of work stealing schedulers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, p. 507–518.

[38] U. A. Acar, A. Charguéraud, and M. Rainey, "Scheduling parallel programs by work stealing with private deques," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013, p. 219–228.

[39] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy binary-splitting: A run-time adaptive work-stealing scheduler," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, p. 179–190.

[40] A. Duran, J. Corbalan, and E. Ayguade, "An adaptive cut-off for task parallelism," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–11.

[41] MIT, "The Missing Semester of Your CS Education." [Online]. Available: https://missing.csail.mit.edu/

[42] Rice University, "Habanero extreme scale software research project." [Online]. Available: http://habanero.rice.edu/

[43] Georgia Tech, "Habanero extreme scale software research lab." [Online]. Available: https://habanero.cc.gatech.edu/