# Integrating Parallel Computing into the Undergraduate Curriculum at Texas State University: Experiences from the First Year


Martin Burtscher
*Texas State University*
*San Marcos, TX*
*burtscher@txstate.edu*

Wuxu Peng
*Texas State University*
*San Marcos, TX*
*wuxu@txstate.edu*

Apan Qasem
*Texas State University*
*San Marcos, TX*
*apan@txstate.edu*

Hongchi Shi
*Texas State University*
*San Marcos, TX*
*hs15@txstate.edu*

Dan Tamir
*Texas State University*
*San Marcos, TX*
*dt19@txstate.edu*

Heather Thiry
*University of Colorado at Boulder*
*Boulder, CO*
*heather.smith@colorado.edu*



***Abstract*—The widespread deployment of multicore-based computer systems over the last decade has brought about drastic changes in the software and hardware landscape. Yet, many undergraduate computer science (CS) curricula have not embraced the pervasiveness of parallel computing. In their first years, CS undergraduates are typically exclusively trained to think and program sequentially. However, too firm a root in sequential thinking can be a nontrivial barrier for parallel thinking and computing. Thus, there is an urgent need to teach multicore and parallel computing concepts earlier and often in CS programs.**

**This paper describes our efforts at addressing the rapidly widening gap between highly parallel computer architectures and the sequential programming approach taught in traditional CS courses. At Texas State University, we have adopted the early-and-often mode of integrating parallelism into the undergraduate curriculum. In this approach, parallel computing concepts are introduced and reiterated through a series of short, self-contained modules across several lower-division courses. Most of these concepts are then combined into a newly designed senior-level capstone course in multicore programming. Evaluations conducted during the first year show encouraging results for the early-and-often approach in terms of learning outcomes, student interest and confidence gains in computer science.**


## I. Introduction

The ubiquity of parallel computing resources presents a pressing challenge to the entire computer science discipline. The thrust of this challenge is to find ways to better equip computer science students with skills to face an increasingly parallel world. Although there is general agreement that undergraduates should learn parallel and distributed computing (PDC) concepts, there is debate about when parallel programming should be taught and to what extent. Recently, the NSF/IEEE-TCPP PDC committee and ACM have emphasized the need for integrating PDC topics across the curriculum [1], [2]. Although these initiatives have garnered strong support from the community, there remain key challenges in realizing this vision. The pedagogy of teaching current PDC topics to undergraduates is yet to mature and major curriculum revisions are problematic, particularly for departments where revisions to the curriculum require significant planning and effort, including training of faculty teaching lower-level classes, complying with administrative policies of the university curriculum board, and tracking graduation credits for majors under the revised curriculum. This paper describes our experiences in implementing the early-and-often approach to integrating parallel computing into the undergraduate curriculum.

The early-and-often approach, originally proposed by Brown et al. [3], aims to introduce PDC concepts through a series of modules dispersed across several courses in the curriculum. At Texas State University, we have adopted a similar module-driven approach with special emphasis on *how* the modules are developed and *when* they are introduced. Fig. 1 presents an overview of our integration strategy. The development and deployment of the modules is based on three key principles that provide several pedagogical advantages. We discuss these ideas next.

***Introduce parallel topics at the right level of abstraction:*** To gain mastery in parallel programming (and sequential programming, for that matter), students need to learn how to think about problems at different levels of abstraction and acquire the ability to switch between levels rapidly. It is very important to determine the right level of abstraction for introducing different aspects of parallel problem solving. Exposing students to multiple levels all at once is likely to create confusion. Moreover, choosing too low a level may hide some of the natural parallelism available in an algorithm and result in lost opportunity. We advocate an approach that starts with the most abstract forms of concurrency and parallelism, and progressively reveals lower-level mechanisms required for more complex forms of process interaction. For example, students can learn about Amdahl's law for

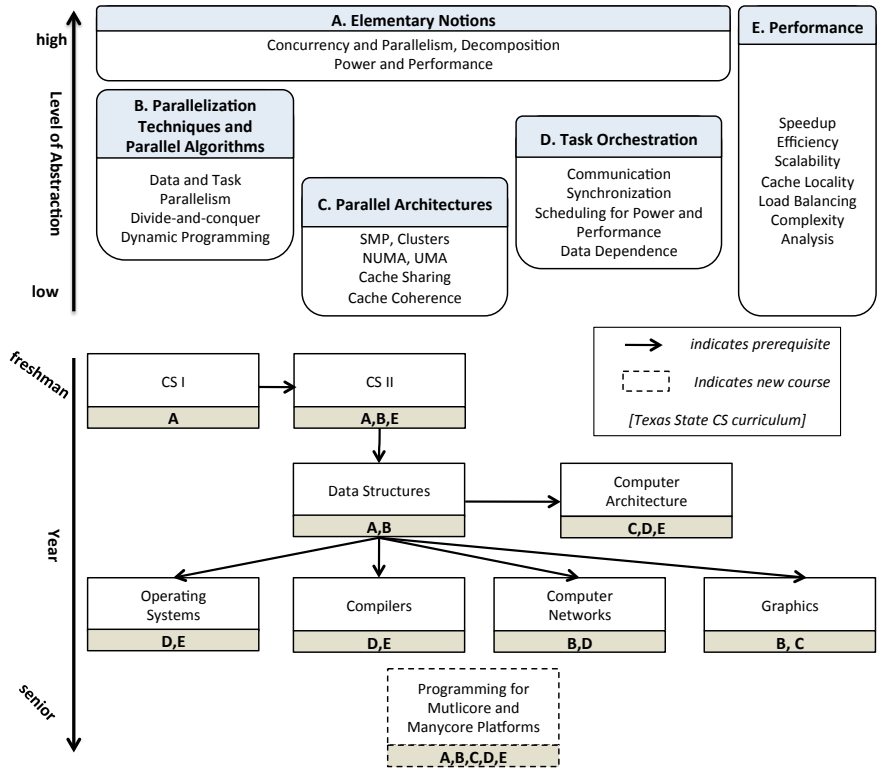


Figure 1. Parallel computing modules and their coverage in undergraduate courses

parallel programs without being able to program in parallel or having knowledge of synchronization and communication primitives. Concepts that students can grasp at a higher level of abstraction are introduced first and reinforced in subsequent years as students are gradually exposed to lower-level concepts. Finally, we tie all the ideas together in the form of a capstone course at the senior level. Some topics, such as performance of parallel programs, span multiple levels of abstraction and are therefore part of several course modules.

***Provide "parallel context" to key topics in the existing curriculum:*** Many theories and concepts covered throughout the CS curriculum can enhance a student's comprehension of parallel computing principles. However, such topics are often not taught in a parallel context. For example, almost all data structures courses introduce recursion, and in many cases, a divide-and-conquer algorithm is used as a primary example. Yet, the fact that divide-and-conquer algorithms naturally lend themselves to parallelism is rarely emphasized. Similarly, in later algorithms courses, the complexity analysis of divide-and-conquer traditionally ignores parallel implementations. Our approach incorporates parallel context to key ideas such as divide-and-conquer algorithms and recursion. Since the modules are dispersed over several courses and do not introduce completely new concepts but rather extend topics that are already being covered, we do not expect these modules to cause significant strain on covering original course content. What material, if any, needs to de-emphasized or condensed to make room for the module is left to the decision of the individual instructor.

***Encourage adoption across different institutions:*** The modules we developed are self-contained with lecture notes, assignments, exercises, exam questions, and solutions. The estimated duration for each module is between one and four 1.25-hour lectures. Some modules include lab time. Although the modules provide a textbook treatment of the material, they are not tied to any specific textbook or lab manual. This enables straightforward adoption of the modules at other institutions. The course modules are designed so that they are mostly language independent. Of course, some modules require the use of specific parallel languages or APIs. In these cases, we develop multiple modules for the same concepts using alternate language interfaces. Generally, modules do not entail any prerequisites other than the ones prescribed for the course in which the module is taught. However, the introduction of some modules may need to be postponed until later in the semester when requisite material has been covered.

Table I
PDC MODULES AND COURSE INFORMATION

| | Module | Course | Enrollment |
|---|---|---|---|
| 1. | Parallelization techniques (B) | CS II (CS2308) | 43 |
| 2. | Intra-processor parallel architecture (C1) | Assembly Language (CS2318) | 52 |
| 3. | Inter-processor parallel architecture (C2) | Computer Architecture (CS3339) | 27 |
| 6. | Performance: basic concepts (E1) | Data Structures (CS3358) | 47 |
| 5. | Task orchestration: scheduling and mapping (D2) | Operating Systems (CS4328) | 30 |

## II. First Year Implementation

### A. Module Development and Teaching

To date, we have developed five modules covering some of the core PDC concepts. The developed modules were introduced in seven sections of five undergraduate courses during the previous academic year. The modules, the courses in which they were taught, and the enrollment in each class are listed in Table I. Three of the modules were introduced in lower-level courses[1], while the other two were introduced in courses with mostly juniors and seniors. The modules were taught by four different instructors. Except for the module on Inter-Processor Architecture (C2) introduced in the Computer Architecture course, all modules were taught by faculty who were not directly involved in the development of the module. We conducted short training sessions for faculty with no prior experience in teaching parallel computing.

The teaching material associated with each module is available on a web site we set up to support our dissemination efforts [4]. To help the reader understand the structure and content of our modules, we briefly describe two sample modules.

### B. Task Orchestration: Scheduling and Mapping on Multicore Systems (Module D1)

*Description:* This module extends scheduling algorithms covered in a typical upper-level operating systems course and introduces concepts and algorithms for thread scheduling on multiprocessor systems. A priority-based algorithm for optimal throughput is presented and then modified to use power consumption as the main objective function. The notions of energy efficient computing and load balancing to improve throughput are discussed in this context. The module further includes a processor-affinity-based scheduling algorithm for multicore platforms. A producer-consumer application is used as an example to illustrate the effects of thread affinity on shared-cache locality and performance. The module briefly covers simultaneous multithreading (SMT) and issues related to scheduling of hardware threads.

*Topics and Learning outcomes (per NSF/IEEE-TCPP PDC Curriculum):*

- **[Programming]** Tasks and threads: understand what it means to create and assign work to threads in a parallel program and how this assignment affects performance; know how to assign work using OpenMP
- **[Programming]** Synchronization: understand the need for inter-thread synchronization; be able to write shared memory programs with critical regions, producer-consumer communication, and get speedup; know the notions of mechanisms for concurrency (monitors, semaphores, etc.); understand safety considerations of parallel execution, including thread-safe functions.
- **[Programming]** Load balancing: understand the effects of load imbalances on performance and power; understand ways to balance load across threads or processes
- **[Programming]** Scheduling and mapping: understand how the operating system schedules threads to computation cores; understand the performance impact of such mapping

*Recommended Length:* One lecture of approximately one hour and fifteen minutes.

*Recommended Course (and rationale):* Operating Systems. Scheduling of tasks is a topic discussed at some length in any standard Operating Systems course. This provides the ideal context for exposing students to concepts related to scheduling of tasks on multicore processors.

*Source Code:* Pthreads implementation of a producer-consumer application.

### C. Performance: Basic Concepts (Module E1)

*Description:* Improved performance is the main advantage of parallel computing over traditional sequential computing. This module introduces basic concepts and terms in performance of computing, including various techniques typically covered in computer architecture for performance enhancement, the notion of speedup, and Amdahl's law. It also introduces a basic approach to designing a parallel algorithm.

*Topics and Learning Outcomes (per NSF/IEEE-TCPP PDC Curriculum):*

- **[Cross Cutting]** Why and what is parallel/distributed computing? : Know the common issues and differences between parallel and distributed computing; history and applications. Microscopic level to macroscopic level parallelism in current architectures

[1](although Data Structures is a 3000 level course CS majors typically enroll in this course in their third semester)

- **[Programming]** Performance metrics: Know the basic definitions of performance metrics (speedup, efficiency, work, cost), Amdahl's law; know the notions of scalability
- **[Programming]** Speedup: Understand how to compute speedup, and what it means
- **[Programming]** Efficiency: Understand how to compute efficiency, and why it matters
- **[Programming]** Amdahls law: Know that speedup is limited by the sequential portion of a parallel program, if problem size is kept fixed
- **[Algorithm]** Sorting: Observe several sorting algorithms for varied platforms - together with analyses (only Quicksort is covered in this module)

*Recommended Length:* Two to three lectures, with approximately one hour and fifteen minutes devoted to each lecture.

*Recommended Course (and rationale):* Data Structures. In our curriculum, this course requires CS I and CS II as background. Most students also have completed CS2420 (Digital Logic), and some have completed CS3339 (Computer Architecture). CS II covers fundamental aspects of C++ programming. The content of this module is taught near the end of the semester. Therefore, students should have sufficient background to learn the notion of parallel computing and contrast parallel computing with sequential computing.

## III. Challenges

### A. Coverage

Module B (Parallelization Techniques) focuses on non-recursive methods of parallel decomposition. We had planned four lectures. The first lecture introduces these ideas at the algorithmic level. Scanning arrays for minimum and maximum values is used as an example. The second lecture illustrates programming issues related to data parallelism on the example of parallel Ranksort using OpenMP. Metrics for evaluating the parallel performance are also covered. The third lecture introduces parallel graph algorithms using linked-list operations as an example. The final lecture provides an overview of amorphous data-parallelism, a general, data-driven approach to parallelizing arbitrary algorithms.

We selected our equivalent of CS II for introducing this module. The first lecture on finding the maximum value in an array in parallel worked as planned except the instructor felt the slides were too dense. We have since spread out their content and added intermediate steps. As the course covers several sorting algorithms, we thought Ranksort would be a simple addition. However, this (sequential) algorithm proved more difficult for the students to understand than we had anticipated, making the module take longer and forcing the instructor to add examples. We have now updated the slides accordingly. At this point, we realized that the latter lectures would be too advanced for this course. After all, pointers and a first introduction to linked lists are only covered towards the end of the semester. As a consequence, we felt the students' grasp of this newly acquired knowledge was too basic to talk about parallel operations on linked lists or other graph-based data structures. Hence, we did not teach the last two lectures. Instead, we decided to (re-)introduce the entire module in the follow-on data-structures course, which we are teaching this semester. That course starts out with a review of pointers and linked lists, hopefully making it more suitable. Furthermore, it covers sorting algorithms and algorithm complexity as well as general graphs and graph algorithms, which should also make the final two lectures of Module B appropriate for this course.

### B. Content Substitution

Integrating parallel computing into the curriculum through a series of short, self-contained modules introduces several challenges, especially when the integration occurs early in the curriculum and involves lower division courses. Since the integration is not done in a vacuum and many of the lower-division courses are already facing the challenge of prioritizing relevant material, one has to make decisions concerning the elimination, condensing, or change of teaching pace of existing topics in order to enable introducing the relevant PDC modules.

We have identified three main remedies to the issue of integrating the new modules into existing courses that do not involve content elimination:

- Identify areas that are covered in more than one course and better synchronize the coverage.
- Identify areas where students seem to have less difficulty and increase the pace.
- Identify areas that can be partially covered by homework and self-learning.

For example, we wanted to introduce one lecture on intra-core parallelism (exploiting task-level parallelism and data-level parallelism in a single core) into our assembly language course. To be able to do so, we decided to shorten the discussion on the compiler, assembler, linker, and loader chain of operations for the following two reasons. First, we identified an unnecessary overlap of this coverage with the later computer architecture course. Second, we discovered that we could switch from the GCC tool chain, which requires a detailed understanding of this interaction, to the SPIM simulator, which does not require this understanding.

If none of the above remedies are available, one may have to make the 'painful' decision to eliminate or condense existing material. This is not an easy task and probably involves a trade-off between eliminating/condensing 'basic, potentially out-of-date' material and eliminating/condensing 'new and trendy' material. The ultimate decision will likely have to be made by the undergraduate curriculum committee.

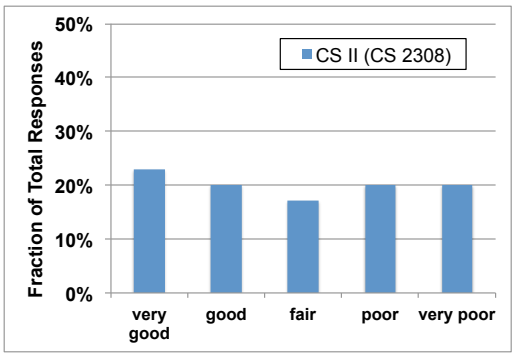


Figure 2. Student Learning Outcome in CS II (CS2308)

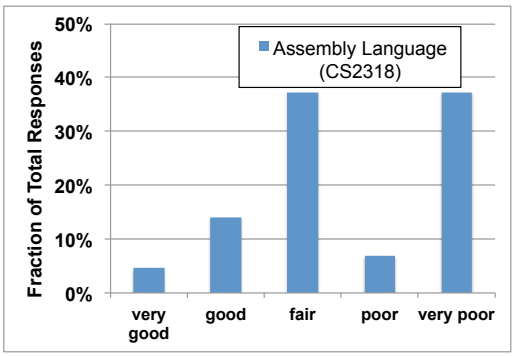


Figure 3. Student Learning Outcome in Assembly Language (CS2318)

### C. Consistency Across Sections

Texas State University has a consistently high enrollment in computer science. As the university is determined to keep class sizes small, the lower division courses in the computer science department are split into multiple sections taught by different instructors (with as many as 11 sections for the CS I course). This poses several administrative challenges in introducing new content even if the material is contained within a single module. For instance, the CS I course adheres to a strict schedule of content coverage to ensure that lecture and laboratory sections are synchronized. Introducing a new PDC module in this setup would disrupt this schedule and require significant changes in the way the course is currently being taught. Furthermore, for courses taught by multiple faculty, we need to ensure that all instructors receive adequate training so that the coverage is consistent across all sections and that the evaluation is being conducted in a uniform way. We plan to conduct a longitudinal study to evaluate the effectiveness of the developed modules. For this reason, in our first year, we chose to introduce the modules in only a subset of sections of every multi-section course. This approach creates multiple paths through the enhanced curriculum, providing better data for our final evaluation. This strategy may be reasonable during the initial phase, however, for a more complete adoption, proper co-ordination of a multi-section course will be needed.

## IV. Evaluation and Assessment

We instated two forms of evaluation during the first year. The assessment plan for student learning outcome was designed by the involved faculty whereas teaching effectiveness and student engagement was evaluated through an independent external evaluator. Additionally, we started collecting and compiling data for a longitudinal study of student understanding of parallel concepts. The initial results of that study are expected in 2015.

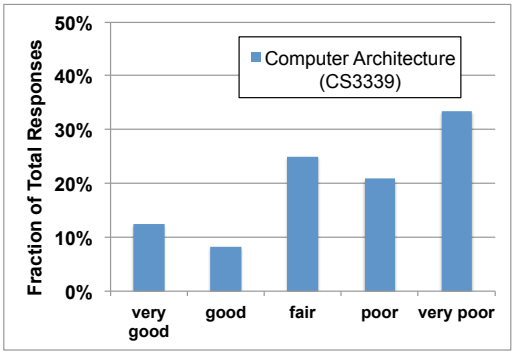


Figure 4. Student Learning Outcome in Computer Architecture (CS3339)

### A. Learning Outcome

For each module, a set of learning outcomes was identified, and these outcomes were mapped to the outcomes listed by the NSF/TCPP curriculum for PDC topics [1] (see Sections II-B and II-C). For each section, a final exam question was prepared to assess student comprehension of the content introduced by the module. Although programming assignments and homework problems were associated with the material for some modules, to be consistent, we only considered student performance on final exam questions to determine learning outcome. Aside from numeric scoring (which was different for different sections) a rubric was created for each question that graded student response on a scale of very poor, poor, fair, good and very good. A grade of fair or better was considered a passing grade.

Figs. 2-6 present student performance on final exam questions for each course where a module was implemented. Evaluation of student learning outcomes shows that 60% of the students who were exposed to the PDC module received a passing grade on the final exam question. This number indicates that our initial implementations of the modules were relatively successful. However, the individual course-based breakdown of student performance identifies problem areas that will need to be addressed in future semesters. Specifically, in *CS2318: Assembly Language*, where the

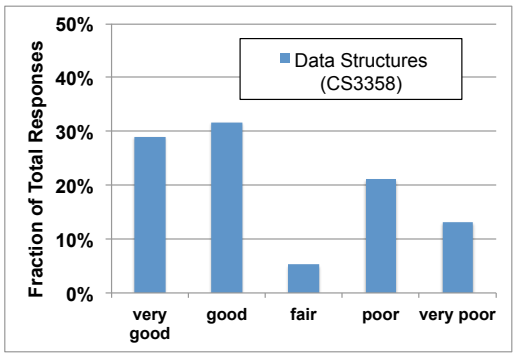


Figure 5. Student Learning Outcome in Data Structures (CS3358)

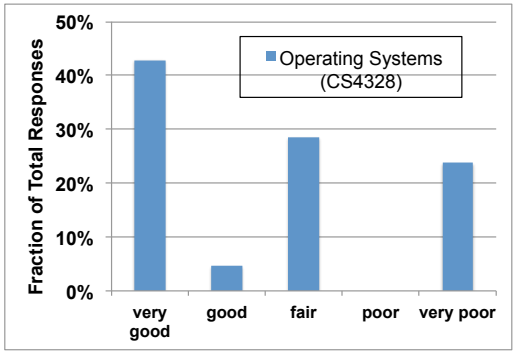


Figure 6. Student Learning Outcome in Operating Systems (CS4328)

module on intra-core parallel architecture was introduced, student performance was below par. Around 38% of the student responses fell in the *very poor* category. We surmise that this poor showing is explained by two factors. First, the material covered in this module may have been too advanced for this particular class. Second, many of the students in this class were transfer students who may not have had the required background or programming maturity (as did students from Texas State) to tackle the presented material. We intend to address both issues by re-aligning the module content for the sophomore-level courses in future semesters.

### B. Teaching Effectiveness and Student Engagement

We conducted an independent external evaluation to assess changes in student confidence and interest in computer science, as well as the students' perceptions of their classroom learning experiences. To obtain these measurements, the Student Assessment of Learning Gains (SALG) survey [5] was administered electronically at the end of the semester to the students enrolled in courses where the modules were introduced.

The strongest reported gains were in confidence and interest in computer science in general, and parallel computing in particular. The students also thought that the learning experiences and instructional environment in the classroom helped their learning. They rated their interactions with peers

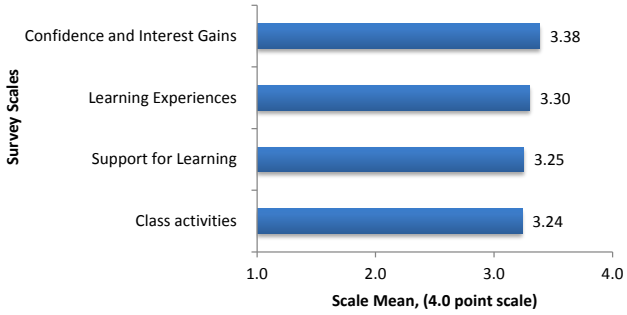


Figure 7. External Evaluation Summary

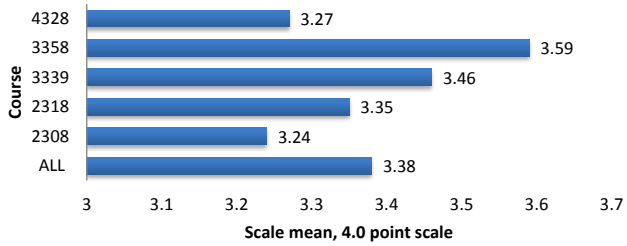


Figure 8. Student Confidence and Interest Gains

and instructors positively and also reported that specific class activities, such as lectures, examples, and asking questions, were helpful. The students were slightly positive that course assignments, projects, and tests helped their learning. Fig. 7 summarizes the scale means of the total sample of students from the four SALG survey scales. We highlight specific aspects of the assessment in the sections that follow.

*1) Student Interest:* Fig. 8 provides a course-wise breakdown of student responses with respect to confidence and interest gains. Overall, 75% of the students reported that the course had increased their "enthusiasm for this subject" a "moderate" or "great" amount. Similarly, a full 92% of the students reported that the course had increased their interest in taking more CS courses. Students overwhelmingly (86% of the students) reported that the course had increased their confidence that they can succeed in computer science.

In an open-ended question, the vast majority of the students affirmed that the courses with parallel computing modules increased their interest in the field of Computer Science. The students were asked to comment on how the course influenced their interest in the subject. Overall, 80% of them reported that the course had positively impacted their interest in the subject matter. A significant minority of the students expanded their answer and noted that they had learned new skills or gained a greater depth of knowledge about the course material. A few students stated that they were already highly interested in the material, and the course served to sustain their interest rather than increase it.

Although the students' perspectives on the modules varied

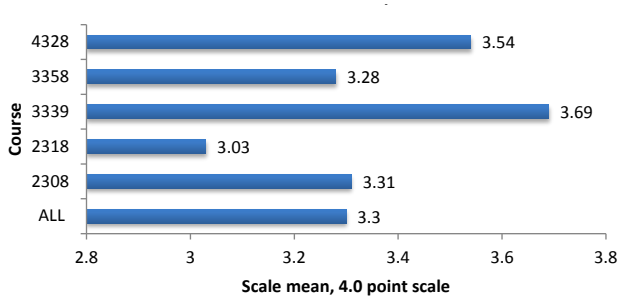


Figure 9. Student Learning Experience

by course, particularly between early and more advanced courses, more than 50% of the students in each course rated the module as "somewhat" or "very" helpful to their learning. The students' ratings of the modules increased in the more advanced courses in the major (those at the 3000 and 4000 level). The students in the lower courses may not have had the prior knowledge or experience to fully understand the parallel computing concepts, and modification of the modules in the future may be needed.

*2) Classroom Learning Experience:* The students rated the learning environment in their course positively. The "learning experiences" scale measures the efficacy of the general instructional approach and curriculum in the course. The learning environment in each course was rated between "somewhat helpful" (3.0) and "very helpful" (4.0). The mean over all courses was 3.3. The differences among the courses are not statistically significant. Thus, the students were generally satisfied with the teaching strategies used in their CS courses. For instance, 89% of the students found the "instructional approach taken in this class" to be "somewhat" or "very" helpful. Similarly, 88% of students felt that their learning was enhanced by the way that the class sessions, activities, and assignments fit together. Fig. 9 documents the course means for the "learning experiences scale" (3=somewhat helpful, 4=very helpful).

## V. Summary

Overall, we rate our first-year venture to introduce PDC topics in the undergraduate curriculum a success. The evaluation and assessment results show that the students developed a reasonable understanding of the parallel computing concepts introduced through our modules. Furthermore, the module-based approach was rated favorably by the students with respect to their learning experience. We also identified several problem areas in implementing the early-and-often approach. In one instance, the material proved to be too advanced for the course, prompting us to split the content into two separate modules. In another instance, truncating existing material to accommodate module contents proved challenging. We have identified solution strategies for these problems and intend to implement them in the coming year as we move forward with our efforts of integrating PDC into the undergraduate curriculum.

## Acknowledgements

This work was funded by the National Science Foundation under grants DUE-1141022, CNS-1253292 and CNS-1217231.

## References

[1] "NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing." [Online]. Available: http://www.cs.gsu.edu/tcpp/curriculum

[2] "ACM IEEE Joint Task Force on Computing Curricula, Computer Science Curricula 2013 Strawman Draft," 2012.

[3] "CSinParallel Project," 2013. [Online]. Available: http://csinparallel.org/

[4] "Parallel Computing in the Undergraduate Curriculum: the Early-and-Often Approach," 2013. [Online]. Available: http://tues.cs.txstate.edu/

[5] E. Seymour, D. Wiese, A. Hunter, and S. M. Daffinrud, "Creating a better mousetrap: On-line student assessment of their learning gains," in *National Meeting of the American Chemical Society*, 2000.