A Module-based Approach to Adopting the 2013 ACM Curricular Recommendations on Parallel Computing

Martin Burtscher
Texas State University
burtscher@txstate.edu

Wuxu Peng
Texas State University
wuxu@txstate.edu

Apan Qasem
Texas State University
apan@txstate.edu

Hongchi Shi
Texas State University
hs15@txstate.edu

Dan Tamir
Texas State University
dt19@txstate.edu

Heather Thiry
University of Colorado
heather.smith@colorado.edu

ABSTRACT

The widespread deployment of multicore systems over the last decade has brought about major changes in the software and hardware landscape. The resulting importance of parallel computing is reflected in the 2013 Curriculum Guidelines developed by the joint ACM/IEEE taskforce. The document recommends increased coverage of parallel computing and describes a new Knowledge Area on this topic. These recommendations have already been adopted by several universities in the form of new parallel-programming courses. Implementing the recommendations in a complete curriculum, however, poses many challenges, including deciding on existing material to be removed, complying with administrative and ABET requirements, and maintaining caps on graduation credit hours. This paper describes an alternative approach for adopting the 2013 curricular recommendations on parallel computing. Specifically, we use a module-based approach that introduces parallel computing concepts and re-iterates them through a series of short, self-contained modules taught across several lower-division courses. Most of these concepts are then combined into a new senior-level capstone course on parallel programming. Each module covers parallelism aspects in the context of a conventional computer science topic, thus enabling us to include parallel computing without a major overhaul of the curriculum. Evaluations conducted during the first year show encouraging results for this early-and-often approach in terms of learning outcomes, student interest, and confidence gains.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Curriculum

Keywords

parallel computing, module-based instruction, pedagogy

SIGCSE'15 March 04-07 2015, Kansas City, MO, USA.

ACM 978-1-4503-2966-8/15/03 ...$15.00.
http://dx.doi.org/10.1145/2676723.2677270.

1. INTRODUCTION

The broad adoption of multicore-based computer systems over the last decade has introduced unprecedented challenges to programmers, who, for the first time, have to find ways to take advantage of multiple threads. Because almost all legacy software was developed to run on a single thread, most programmers lack the skills, knowledge, and experience needed to safely and effectively exploit parallelism. Thus, it is paramount that the next generation of software developers be trained in parallel programming.

In response to this shift in industry, the computer science education community has made efforts to increase the amount of parallel and distributed computing (PDC) concepts in the curriculum. New classes in parallel and distributed programming have been added and existing courses have been bolstered with PDC content. Recently and most notably, the ACM/IEEE joint taskforce on computing curricula has published its guidelines for undergraduate degree programs (henceforth referred to as ACM2013) [3]. This document puts special emphasis on PDC, which has been designated as a new Knowledge Area. Unlike all previous versions, ACM2013 now mandates the inclusion of several core hours of PDC in every CS curriculum.

There are many challenges to implementing the ACM2013 recommendations on parallel computing. Creating a new course or offering an existing course on PDC more frequently gives students the option to be exposed to these concepts. However, making such a course a required part of the curriculum is often problematic because of the cap on the number of credit hours. For instance, in Texas this cap is 120 hours for undergraduates, and at Texas State University more than half of those hours come from non-major areas. These constraints imply that another course covering some fundamental aspects of computer science would need to be eliminated before a new course can be made a requirement. Furthermore, the Knowledge Units in PDC are such that they are better covered in multiple courses for pedagogical reasons, as recommended in ACM2013.

Distributing the PDC concepts over multiple courses raises several pedagogical, curricular, and administrative challenges.

- *Identifying courses for inclusion:* The merit of teaching sequential programming as a *special case* of parallel programming is still under debate, which is why it is not advocated in ACM2013. Hence, the current direction dictates that PDC be introduced in lower-level courses within the paradigm of sequential pro-

gramming. Identifying courses where PDC material can be taught without disrupting the flow of content while satisfying the pre-requisite structure is difficult.

- *Adjusting existing content:* Introducing new material, particularly in lower-division courses, implies that some existing content needs to be eliminated or condensed. It is not readily apparent what this content should be, making it difficult to insert new topics. An even less desirable approach is to concentrate all material in a core upper-level course (e.g., Programming Languages or Operating Systems) where there is some flexibility in organizing the material. Several of the example curricula presented in ACM2013 take this approach. However, by their own admission, this does not provide the coverage of many of the Tier 2 core topics [3].
- *Faculty training:* Faculty teaching introductory courses may not have sufficient experience teaching PDC topics. In many institutions, introductory courses are staffed by a rotating pool of adjunct faculty. This makes it difficult to provide adequate training or maintain consistency across sections of lower-level courses, further complicating the adoption of PDC material.
- *Administrative issues:* At many universities, curriculum revisions require a rather involved process, moving through departmental, college, and university curriculum committees. Changes may be further restricted to satisfy the timeline for *curriculum cycles*.

This paper describes our method to adopting the ACM2013 curricular recommendations on parallel computing that addresses the above issues. In our early-and-often 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 later combined in a newly designed senior-level capstone course on parallel programming. The development and deployment of the modules are based on three key principles (described in Section 3) that provide pedagogical advantages and facilitate a gradual migration to a PDC-enhanced curriculum.

2. BACKGROUND

Although parallel computing has been an important area within computer science for decades, it did not find its way into mainstream undergraduate CS curricula until recently. Efforts at integrating parallelism can be divided into two periods, delineated by the advent of multicore processors.

2.1 Pre-multicore era integration efforts

Early efforts at integration mostly involved developing an upper-level elective covering various aspects of parallel computing [10] or an elective that focused solely on a specific parallel programming paradigm [6, 9]. These courses were primarily offered at universities that had faculty with research interests in parallel and high-performance computing. The cost of parallel computers further limited the offering of such courses to larger research-oriented universities [6]. A survey of CS undergraduate curricula at 40 universities of varied orientation (e.g., liberal arts, masters, R1 research) conducted by the authors in 2009 revealed a similar trend[1]. Prior to the multicore shift, there had been only a few efforts at a more holistic integration of parallel computing into the curriculum [12]. Most of these efforts either did not sustain or were not adopted beyond the originating college.

[1]The survey was conducted as part of a proposal to NSF and was not published independently.

2.2 Multicore era integration efforts

The industry-wide shift to multicore processors provided strong impetus for integrating PDC into the undergraduate curriculum. Many more parallel computing courses are now being offered with greater regularity. Curricular revisions are also under way to include more parallelism. The Georgia Institute of Technology has adopted a *rolling introduction* to parallel computing concepts in both its CS and ECE curricula [7]. The wide availability of multicore systems has made parallel computers much more affordable. This has allowed smaller liberal arts colleges to incorporate parallelism in their courses. In fact, many of the recent integration efforts have been initiated at such colleges [5, 8].

There are also endeavors at building communities to encourage and aid the adoption of PDC topics across universities, notably the CSinParallel project, whose aim it is to insert parallel computing concepts into various CS courses in multiple curricular contexts [2]. The formation of the TCPP Curriculum committee on PDC by the IEEE and the National Science Foundation is another notable endeavor in the same direction. This multi-institutional committee organizes many activities to encourage adoption of PDC at universities both nationally and internationally [4]. The success of this committee has led to the formation of the CDER Center, a larger body with similar goals [1].

2.3 PDC in ACM2013

ACM2013 underlines the need for integrating Parallel and Distributed Computing into undergraduate CS curricula and includes PDC as a new Knowledge Area. The topics within the PDC area are broken down into nine Knowledge Units. Furthermore, parallelism-related concepts also appear in the System Fundamentals Knowledge Area. Five Core Tier 1 hours on PDC are now recommended, compared to zero in the previous version of the guidelines. An extra 10 hours of Core Tier 2 is also recommended. ACM2013 clearly states the need for spreading PDC topics across many courses in addition to offering a dedicated course on parallelism. It includes four example curricula, all of which demonstrate increased emphasis on parallel computing.

3. OUR APPROACH

For bringing the ACM2013 recommendations to our university (and similar institutions), we employ the early-and-often approach originally proposed by Brown et al. [2]. This strategy introduces PDC concepts through a series of modules dispersed across several courses. In our implementation, we put special emphasis on the coverage of topics in each module individually *and* as a sequence. We also pay attention to *when* they are taught. As depicted in Fig. 1, the key idea is to provide sufficient intersection with PDC topics on every major path through the curriculum. This ensures that students attain a broad perspective on parallel and distributed computing, irrespective of their choice of electives. The development and deployment of the modules is based on three key principles that provide certain pedagogical advantages and facilitate a gradual migration to a PDC-enhanced curriculum. We discuss these principles next.

(1) Introduce concepts at the right level of abstraction: To gain mastery in parallel programming (and se-

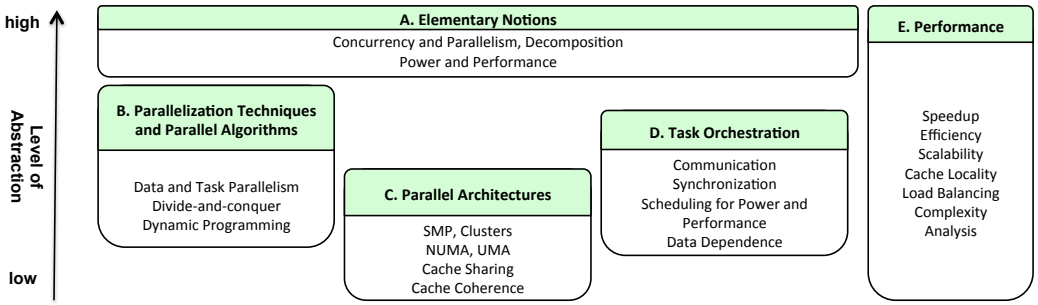

Figure 2: Module classification based on level of abstraction

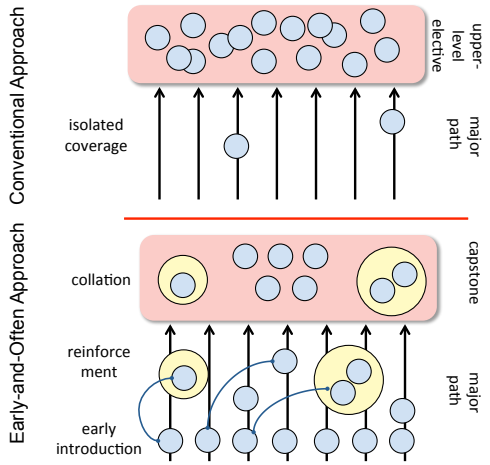

Figure 1: The one-elective (top) and our early-and-often (bottom) approach of introducing PDC

quential 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. Nevertheless, it is important to determine the right level of abstraction for introducing different aspects of parallel problem solving as exposing students to multiple levels all at once can create confusion. We advocate an approach that starts with the most abstract forms of concurrency and progressively reveals lower-level mechanisms. For example, students can learn about Amdahl's law for parallel programs without being able to write parallel programs; therefore, this topic can already be introduced in a CS I module. Fig. 2 shows a classification of module topics based on the level of abstraction. Higher-level concepts are introduced first and reinforced in subsequent years as students are gradually exposed to lower-level ideas. Some topics, such as performance of parallel programs, span multiple levels of abstraction and are covered in several modules.

(2) 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. Our approach incorporates parallel context in key topics such as divide-and-conquer algorithms. Since the modules are dispersed over several courses and do not introduce completely new concepts but rather extend topics that are already being covered, they can be introduced with minimal disruption of the original course schedule and content.

(3) Encourage adoption: For our approach to be successful, the modules need to be easy to adopt such that even an instructor with no prior experience in teaching parallel computing can seamlessly plug them into her course. With this goal in mind, the modules are designed to be short (1-3 hours) and self-contained. They include lecture notes, assignments, exercises, exam questions, solutions, and pedagogical notes. Although the modules follow a textbook treatment of the material, they are not tied to any specific textbook and are designed to be mostly programming language independent. Examples and assignments are created for multiple languages, APIs, and platforms, enabling straightforward adoption at other institutions.

4. IMPLEMENTATION

We began implementing the early-and-often approach at Texas State in Fall 2012. In this section, we describe our implementation efforts over the past two years.

4.1 Curriculum Overview

Our university offers both a Bachelor of Arts (BA) and a Bachelor of Science (BS) degree in computer science. The difference between the two programs in terms of major courses is minimal, and the described strategy applies equally to both programs. Thus, we only discuss the BS curriculum, an overview of which is shown in Fig. 3. Not all electives are listed and only Discrete Math is included among the required math courses. The students have to take an additional 46 credit hours of General Education Core, which is not shown in Fig. 3. The CS introductory sequence has a hardware and a software track. In the hardware track, the students progress through CS I, Assembly, Digital Logic, and Computer Architecture. In the software sequence, the students move through CS I, CS II, and Data Structures[2]. Software Engineering and Object Oriented Design and Implementation (OODI) serve as two required capstone courses. Embedded Systems and Human Factors are two of the three project courses that our students are required to take. Hence, most of them will take one of these courses while pursuing their undergraduate degree.

[2]This course doubles as an Algorithms course.

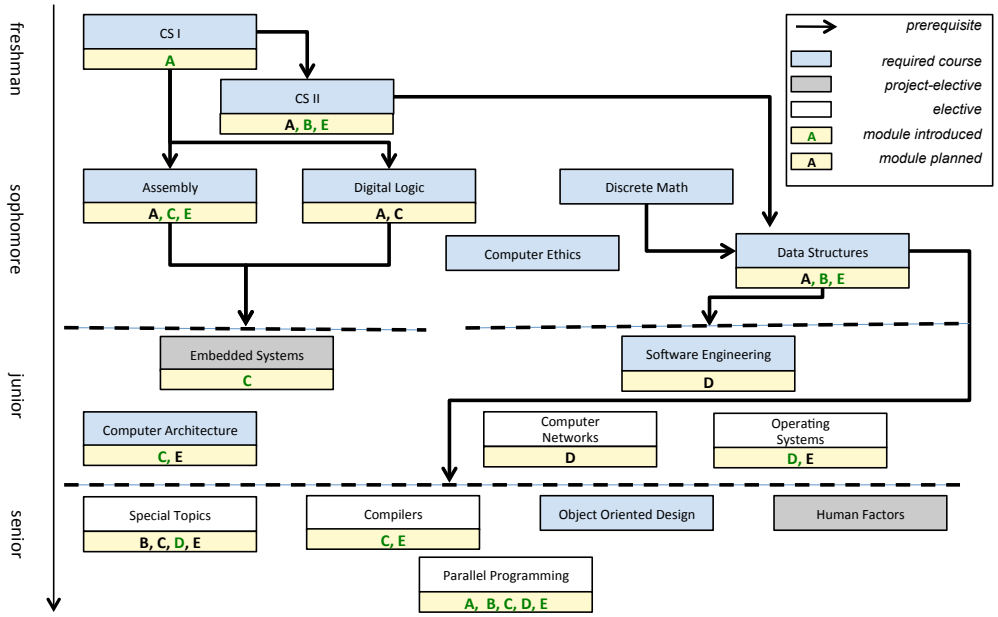

Figure 3: Computer science curriculum and mapping of PDC modules to major courses

Module Category	ACM2013 PDC Knowledge Unit
A. Elementary Notions	Parallelism Fundamentals (PF)
B. Parallel Algorithms	Parallel Decomposition (PD) Parallel Algorithms (PA)
C. Parallel Architecture	Parallel Architecture (PR)
D. Task Orchestration	Communication & Coordination (CC)
E. Parallel Performance	Parallel Performance (PP)

Table 1: Module categories and their correspondence to ACM2013 Knowledge Units

4.2 Coverage of ACM2013 Knowledge Units

To date, we have developed seven modules covering various aspects of parallel computing. These modules are divided into five categories indexed A through E. The module categories and the ACM2013 Knowledge Units (KU) they correspond to are shown in Table 1. The planning and development of the modules began prior to the publication of ACM2013. For this reason, our classification differs slightly from the ACM2013 mandated KUs. This difference is only nominal, however. As shown in Table 2, our module topics map closely to the topics of the corresponding KUs in ACM2013. For instance, module A in Elementary Notions covers all four topics (and corresponding learning outcomes) of the Parallelism Fundamentals KU. Combined, the seven modules provide coverage of all Tier 1 Core topics and most Tier 2 Core topics in the Parallel Algorithms and Parallel Performance KUs.

4.3 Module Placement

Because of the 120-hour cap on graduation hours and many of our students' dependence on financial aid, most CS majors only end up taking about four or five upper-level electives before graduating. This makes careful planning of module integration vital to the success of our strategy.

Before implementing our strategy, we first performed a detailed review and analysis of the material covered in each of our introductory sequence courses and several upper-level electives. In reviewing the existing content, we had two goals: (i) to find *context* for introducing parallel topics and (ii) to identify content that has been removed or de-emphasized in ACM2013. Based on this analysis, we constructed a mapping of our modules to various courses in the curriculum. This mapping is illustrated in Fig. 3. We found opportunities for introducing modules in four introductory courses, both in the hardware and the software track. We also discovered opportunities in four upper-level required courses. Computer Ethics is the only required course where we could not place a module. We identified several electives as good candidates for module integration as shown in Fig. 3.

Most of these opportunities have already been exploited (shown in green text in Fig. 3). We plan to introduce more modules in future years. However, as the data in Table 3 show, our current implementation already provides excellent coverage of ACM2013 PDC topics. A typical major will encounter 8.5 Tier 1 hours and 4.5 Tier 2 and elective hours even without taking the Parallel Programming capstone course. It should be noted, though, that several concepts are reiterated throughout the modules. Hence, the

	ACM2013 PDC Knowledge Unit					
Courses	**PF**	**PD**	**CC**	**PR**	**PA**	**PP**
CS I	0.75					0.25
CS II	0.5	0.5	0.25			0.25
Computer Ethics						
Assembly	0.5			0.5		
Digital Logic	0.5			0.5		
Data Structures	0.25	0.5	0.5		0.5	
Computer Architecture	0.25			1		0.5
OODI					1	0.5
Software Engineering			1			
Embedded Systems				1		
Operating Systems		1			0.5	
Other Elective						
Total	2.75	2	1.75	3	2	1.5

Table 3: PDC coverage in CS major path (grey cells refer to Tier 2 topics)

Module	Topics in ACM2013 PDC Knowledge Units																					
	1a	1b	1c	1d	2a	2b	3a	3b	4a	4b	4c	4d	5a	5b	5c	5d	6a	6b	6c	6d	6e	6f
A	x	x	x	x	x	x	x		x						x					x		
B	x	x	x	x	x	x	x		x						x					x		
C1	x	x									x	x					x	x				x
C2	x	x							x	x							x	x				x
D1	x	x		x	x	x	x	x														
D2	x	x							x	x							x	x	x			x
E	x	x			x	x											x	x		x		x

Table 2: Coverage of ACM2013 topics in PDC modules. Topics are numbered sequentially starting with first Knowledge Unit (i.e., 1a denotes the first topic in KU Parallelism Fundamentals)

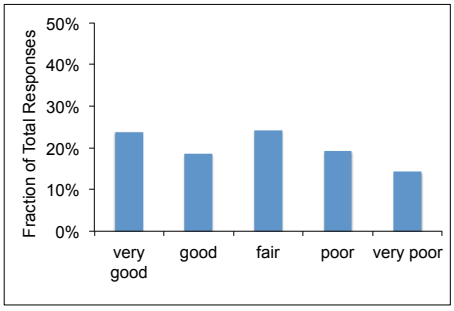

Figure 4: Student performance on PDC topics in all courses where a module was introduced (2013-14)

9.5 cumulative hours in Tier 1 do not quite represent 9.5 hours of coverage of *distinct* PDC topics.

5. EVALUATION

We instated two forms of evaluation during the first year. The assessment plan for the student learning outcomes was designed by the involved faculty whereas the teaching effectiveness and student engagement was evaluated through an independent external evaluator. Additionally, we began collecting and compiling data for a longitudinal study in which we evaluate student performance in the capstone Parallel Programming course in relation to the number of modules the students have encountered as they progressed through the program. At this time, we do not have sufficient longitudinal data to present because only a few students have completed the capstone course who have also taken a class with a PDC module. We hope to publish these results by the end of 2015.

The modules were implemented in 13 sections of six different courses over three semesters. The combined enrollment in these sections was 323. Seven instructors were involved in teaching the modules in the various sections.

5.1 Learning Outcome

For each section in which a module was introduced, a final exam question was prepared to assess student comprehension of the material covered. Although programming assignments and homeworks were associated with most modules, for consistency, we only considered student performance on final exam questions to evaluate the learning outcome. Aside from numeric scoring (which was different for different sections), a rubric was created for each question that graded the student responses on a scale of very poor, poor, fair, good, and very good. Fair or better was considered a passing grade.

Fig. 4 shows the resulting student performance on the final exam questions in all courses where a module was implemented. We observe a satisfactory learning outcome for students who were introduced to new PDC concepts. Overall, 67% of the students received a passing grade on the exam questions. This rate is slightly higher than all majors who score a C or better in these courses.

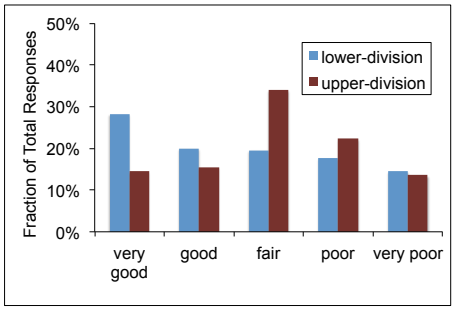

Figure 5: Comparison of student performance in lower- and upper-division courses

Fig. 5 compares the student performance in lower- and upper-division courses. We were apprehensive that the students may perform poorly in some of the lower-division courses because the topics appear to be too advanced to them. Yet, somewhat surprisingly, the passing rate was higher (about 3%) in the lower-division courses, attesting to the suitability of module placement.

After implementing a module for the first time, we obtained feedback both from the instructor and the students on the content and pedagogy. Then we revised and improved the modules based on this feedback. Fig. 6 compares the student performance in courses where a module was introduced for the first time to courses where a revised module was introduced. As we can see, there is almost a 17% increase in the passing rate when a revised module is taught, either by the same instructor or by a different instructor. These results suggest the need for iterative improvement and adjustment of each module.

5.2 Student Interest and Learning Experience

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 [11] was administered electronically at the end of the semester to the students enrolled in courses where a module was 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

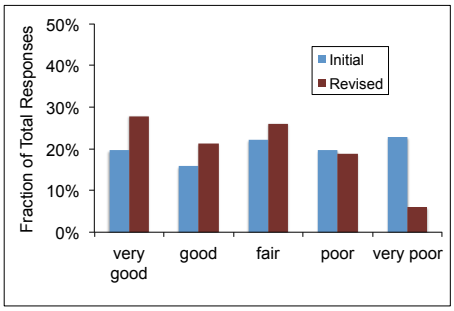

Figure 6: Comparison of student performance during first and second installment of same module

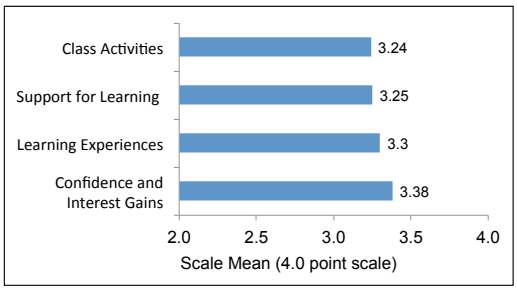

Figure 7: External evaluation summary

experiences and instructional environment in the classroom helped their learning. They rated their interactions with peers and instructors positively and 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. Next, we highlight specific aspects of this assessment.

Fig. 8 provides a course-wise breakdown of student responses with respect to their learning experience, and confidence and interest gains. The students rated the learning environment in their course positively. Fig. 8 shows the course means for the "learning experiences scale", which 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 in all 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. Overall, 75% of the students reported that the course had increased their "enthusiasm for this subject" a moderate or great amount. 92% of the students reported that the course had increased their interest in taking more CS courses.

6. CONCLUSIONS

In this paper, we present a strategy for adopting the ACM2013 recommendations that does not require a major overhaul of the curriculum. Our approach addresses several challenges commonly faced when incorporating a substantial amount

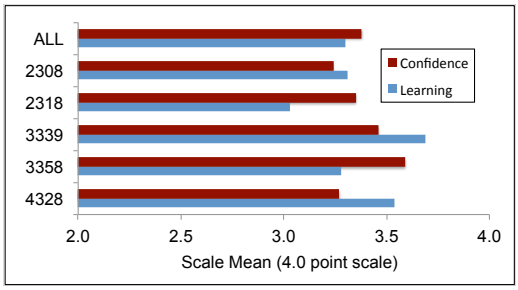

Figure 8: Student learning experience, confidence and interest gains

of new material into an existing curriculum. We have implemented this strategy at our university and so far achieved reasonable success in terms of student learning outcome, engagement, and interest.

Acknowledgements

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

7. REFERENCES

[1] Center for parallel and distributed computing curriculum development and educational resources (CDER). `http://www.cs.gsu.edu/~tcpp`.

[2] CSinParallel Project. `http://csinparallel.org/`.

[3] ACM IEEE Joint Task Force on Computing Curricula, Computer Science Curricula Recommendation and Guidelines 2013, 2013.

[4] NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing. `http://www.cs.gsu.edu/~tcpp/curriculum`, 2013.

[5] D. J. Ernst and D. E. Stevenson. Concurrent cs: Preparing students for a multicore world. *SIGCSE Bull.*, 40(3):230–234, June 2008.

[6] A. L. Fisher and T. Gross. Teaching the programming of parallel computers. In *Proceedings of the Twenty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '91, pages 102–107, New York, NY, USA, 1991. ACM.

[7] Y. S. Gavrilovska H. H., Schwan L. K. and W. M. Multi-core curriculum development at georgia tech: Experience and future steps. In *UCRC Workshop on Experimental Research in Computer Systems*, 2006.

[8] J. R. Graham. Integrating parallel programming techniques into traditional computer science curricula. *SIGCSE Bull.*, 39(4):75–78, Dec. 2007.

[9] J. Hartman and D. Sanders. Teaching a course in parallel processing with limited resources. *SIGCSE Bull.*, 23(1):97–101, Mar. 1991.

[10] D. G. Hyde. A parallel processing course for undergraduates. *SIGCSE Bull.*, 21(1), 1989.

[11] 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.

[12] W. E. Toll. Decision points in the introduction of parallel processing into the undergraduate curriculum. *SIGCSE Bull.*, 27(1):136–140, Mar. 1995.