Integrating Parallel Computing in Introductory Programming Classes: An Experience and Lessons Learned

Sheikh Ghafoor, David W. Brown and Mike Rogers
{sghafoor, dwbrown, mrogers}@tntech.edu
Department of Computer Science
Tennessee Tech University

Abstract. Parallel and distributed computing (PDC) has become ubiquitous to the extent that even common users depend on parallel programming. This points to the need for every programmer to understand how parallelism and distributed programming affect problem solving, teaching only traditional sequential programming is no longer sufficient. To address the rapidly widening gap between emerging highly-parallel computer architectures and the sequential programming approach taught in traditional CS/CE courses, the Computer Science Department at Tennessee Technological University has integrated PDC into their introductory programming course sequence. This paper presents our implementation efforts, experience and lessons learned, as well as preliminary evaluation results.

Keywords. Parallel and Distributed Computing, Introductory Programming, Undergraduate Education

1 Introduction

The widespread deployments of multicore and GPU based computing systems in recent years have changed the computing landscape. Parallel and Distributed Computing (PDC) now permeates almost all computing activities. The pervasiveness of multicore computing devices is making even common users dependent on PDC techniques. The ever-increasing use of web-based services and emerging applications, such as mobile applications, cloud computing, big data analytics, and the Internet of Things (IoT), has made high performance computing common. Therefore, the most effective programmers understand how parallelism and distributed programming affect problem solving. Acquiring only traditional sequential programming skills is no longer sufficient, even for basic programmers. These changes emphasize the need for providing a broad-based skill set in PDC technology at various levels in Computer Science (CS) and Computer Engineering (CE) programs, as well as related computational disciplines. However, the rapid changes in hardware platforms, devices, languages and supporting programming environments continue to challenge educators in ascertaining appropriate content for curriculum and how to effectively teach that material.

The computer science education community now recognizes that integrating PDC concepts in undergraduate curriculums is vital to comprehensive CS/CE education. The TCPP curriculum report [1] has identified core and elective PDC topics that a student graduating with a Bachelor's degree in CS or CE is expected to have covered. Furthermore, PDC has been designated as a new 'required knowledge' unit in the ACM/IEEE-CS Curricula 2013 [2]. However, most undergraduate CS/CE/Engineering programs still do not teach PDC concepts, and such programs typically train students to think and program exclusively in a sequential manner. Although some CS/CE programs offer PDC courses as an upper division elective, very few introduce PDC early, in the introductory programming classes (CS1 and CS2). The gap is rapidly widening between the emerging parallel computing architectures and the sequential computing approach taught in traditional undergraduate curriculums. There are currently three thousand and eleven (3011) 4-year universities in the United States [3] and most of them offer an undergraduate degree program in CS and/or CE. In addition, one thousand, eight hundred and ninety one (1891) two year community colleges offer CS/CE pre-university coursework [3]. However, while no statistics are available on how many institutions are teaching PDC concepts at the undergraduate level, the authors conservatively estimate this number at no more than 300. This estimation is based on grants sponsored by the National Science Foundation, "early adaptor" mini-grants awarded by the CDER Center [4], and faculty development workshops conducted by CS in Parallel [1].

This paper presents the PDC topics and related hands on exercises that have been integrated in traditional CS0, CS1 and CS2 classes taught in the Computer Science Department at Tennessee Technological University (TTU). The paper further describes our experiences and lessons learned from this PDC integration effort.

2 Related Works

Researchers are actively seeking methodologies and tools for introducing PDC into introductory CS courses. In [5], the authors present their effort to implement parallelism in first and second year CS courses. The authors found that students can learn the material and enjoyed the experience. However, in [6], the author suggests that CS2 is the natural place to introduce parallelism, and the author uses minimalistic parallel programming patterns, called patternlets, to teach the student in CS2.

Some researchers have focused on teaching PDC topics to students in upper division courses. For example, Geist et al. [7] describes a course for seniors and first year graduates that covers a real-world problem. Similarly, Lupo et al. [8] focusses on real world experiences with students working in teams. The authors state that eight of the ten learning objectives were met, and that the students enjoyed the real-world experience.

Researchers have also attempted to integrate PDC throughout the curriculum. Burtscher et al. [9] taught PDC in several lower division courses and a senior capstone course. The authors show encouraging empirical results that they achieve their goals in terms of student outcomes, engagement, and interest. Graham [10] used various software models and programming options to teach PDC at various levels of the curriculum. The author also states the students show interest in the topics, but that PDC must be introduced early for the concepts to take root. Neelima and Li [11] present their experiences in introducing PDC topics over 6 academic years. The authors state that the PDC topics were well received by the students. Many students implemented successful projects, and some participated in conferences. Brown, Shoop [12, 13] and Adams [14] argue that PDC concepts should be taught at all undergraduate levels. They have developed a community of PDC educators available at CSinParallel.org [1].

Foley and Hurley [15] state that complex and unfamiliar parallel computing environments, or PCEs, present a barrier to students. The authors present a web portal, called OnRamp, which allows students to interactively explore PDC concepts.

The CDER Center [4] is an NSF supported center for PDC Curriculum and educational resources development. Project personnel chair PDC educational conferences such as EduPar and EduHPC, as well as workshops. Additionally, the CDER Center provides competitive grants for early adopters of PDC in CS courses. The center also provides a book [16] for introducing concurrency in undergraduate courses and provides downloadable and searchable courseware.

3 PDC Implementation

3.1 CS Curriculum at TTU

TTU is a medium sized, accredited public university with an enrollment of approximately twelve thousand students. The Computer Science department has approximately four hundred undergraduate majors and offers BS, MS, and Ph.D. degrees in Computer Science. The introductory courses offered as part of this degree are *Introduction to Problem Solving and Computer Programming* (*CS1*), *Data Structures and Algorithms* (*CS2*), and *Object Oriented Programming and Design* (*CS3*). Multiple sections of these introductory courses are offered each semester; usually the different sections of these courses are taught independently by different instructors. To address the high DFW rates in the 1st and 2nd programming classes, a required *Principles of Computing* (*CS0*) class was added to the curriculum in fall 2013. The students in these courses are usually first or second semester freshmen and are placed in CS0/CS1 according to their math aptitude scores. If the students are able to enroll in calculus, they are allowed to take CS0 and CS1 concurrently. In

addition, CE students are required to take CS1 and CS2 but are exempted from CS0. For the majority of students involved, these courses represent their first real exposure to programming.

In addition to the introductory level coursework, required upper division courses are typically offered once each school year. Our required upper division courses for the traditional CS degree include *Assembly Language Programming, Operating Systems, Computer Networks, Computer Architecture, Database Systems*, and a two-semester capstone *Software Engineering* series.

Beginning in fall 2015, we began introducing parallel concepts into some sections of our CS0, CS1 and CS2 curriculum. One to two days of lecture per semester have been dedicated to introducing why PDC programming is necessary, parallel architecture, basic concepts and how PDC programming differs from sequential coding. Examples are provided to the student outlining parallelism, distributed computing, race conditions and concurrency. In the weeks following these lectures, hands on PDC exercises are introduced into the attached lab portion of the class, or as homework, that highlight a particular attribute of PDC development.

Following the idea of exposing the students "early and often" to the concepts of PDC, each class introduces topics that build upon previous coursework. To accomplish this, we introduce similar concepts in CS0, CS1 and CS2 but at different levels of depth. This model allows the students to practice one facet of PDC in a manner that does not lead to confusion over the complex details of any advanced techniques. Each lab exercise or homework assignment takes as part of the study is worth 8-10% of the final grade in the lab course. The following sections briefly describe the implementation in each class with concise descriptions of the hands on exercises. One of the exercises is described in greater detail for the better understanding of our reader.

3.1.1 Principles of Computing (CS0)

The concepts introduced in the CS0 lecture include *serial computing*, *parallel computing*, *concurrency*, *race condition* and *speed-up*, and the need for parallel computing. We used SNAP [17] to implement the in class examples highlighting these topics. Using animated sprites, provided in SNAP, to represent which components of the application are computing and which ones are not. To highlight the benefits of parallelism, the students are shows two lists of random numbers and the instructor will work them through a sort done in parallel. The instructor can spawn the final merge step for this application in parallel or sequentially after the parallel sort to show that synchronization is needed to overcome the race condition. The module focuses on visualization and examples of parallelism, and does *not* include

coding parallel algorithms. Once the students have been exposed to the concepts, a hands on exercise allows the students to run the sort over data collections and time their results to demonstrate *speed-up*.

3.1.2 Introduction to Problem Solving and Computer Programming (CS1)

The objective in the CS1 parallel introduction is to introduce the students to basic OpenMP coding, the *fork-join* model of parallel processing, as well as have the student become more familiar with the ideas of *shared* v. *distributed memory*, *designing parallel programs* and the differences between *concurrency* and *parallelism*. In addition, the topics covered in CS0 are restated since that course is not a requirement for all students.

Two modules have been created for use in the CS1 laboratory course. The first is a simple demonstration of fork-join summation and allows the students to create a basic parallel program and observe the speed-up PDC allows. The second more complex module walks the students through the manipulation in parallel of arrays for the means of image manipulation. Both modules help reinforce the concepts covered in the main lecture.

Parallel Sum for CS1: The parallel sum lab is designed to introduce students to the fork-join model of parallel programming. The lab begins by introducing the concepts and reasoning behind PDC programming and explaining the expected results of the experiment. The students are instructed to create a program which will create a large array, at least 1 million elements, of randomly generated integers. A function is created to process the array, adding all the elements in a standard sequential manner. A separate function is created to perform the same process but utilize fork – join through OpenMP. A timer function placed in the program allows the users to accurately determine how long each function took to arrive at the answer. The students run the program multiple times using each of the two functions and are able to see the time savings adding simple parallel code can have on their programs performance.

Parallel Image Processing for CS1: The lab describes image flipping and gray-scaling with an example, shown in Figure 1. In particular, images are represented as colored dots, known as pixels, on the monitor screen. The color of the pixel is represented as a mixture of intensities of the colors red, green and blue. Each intensity is characterized by an 8-bit number in the range from 0 to 255. For example, the value (0, 0, 0) represents the color black, the values (255, 0, 0) represents red, and the values (255, 255, 0) represent yellow. We call these intensities, the colors RGB (or red, green, blue) values.

a) Color original image b) Gray-scaled and flipped image

Figure 1: Flipping and gray-scaling an image

Gray-scaling an image represented as a series of RGB values is easy. Different methods exist, but an effective method is called the *luminosity method*. In this method, if you are given the *i*th pixel, you gray-scale that pixel with the following formula:

$$gray\_value[i] = 0.21 * pixel[i].red + 0.72 * pixel[i].green + 0.07 * pixel[i].blue \quad (1)$$

Then, for each *i*, set the red, green and blue component of *pixel*[*i*] to *gray_value*[*i*] to gray-scale the image. Flipping an image is accomplished by flipping the first pixel with the last pixel, the second pixel with the second-to-last pixel, and so on. The lab then describes how an image can be flipped and gray-scaled in parallel. An image has both a height and a width. The array of color values represents rows of pixels, where each row is a line of pixels that would appear across the screen. The size of each line of pixels is equal to the image's width, and the number of lines is equal to the images height. When writing a parallel application, the programmer must first determine how to divide the problem among the available processors. Dividing the problem requires determining 1) how much of the problem each processor should compute, and 2) determining where, in the input data, the processor should begin and end its computations. In general, when dividing the rows among processors, the programmer should divide the work equally. So, if the image consists of *n* rows, and there are *p* processors available, then each processor should get roughly *n/p* rows.

A natural division for an image is to divide the image into chunks, where each chunk consists of a number of rows of pixels. Then, each processor computes its assigned chunk. So, if the given machine has four processors and the image file is eight pixels square, each processor would compute two rows. Processor 1 would compute the first two rows, starting at index 0 and finishing with index 15, processor 2 would start at index 16 and process through index 31, and so on.

Next, the lab describes the tools needed to edit and compile a parallel program, and includes a link to download code for loading and saving images in the simple PMM format, as well as a description of the PPM libraries API. The lab also describes pseudocode gray-scaling and flipping before finally explaining how OpenMP can make writing parallel programs

easier. In fact, when using OpenMP, writing code to parallelize simple loops, such as the ones in this lab, becomes trivial.

3.1.3 Data Structures and Algorithms (CS2)

The objective in the CS2 parallel introduction is to reinforce the material the students had covered in CS1 while expanding their ability to learn and think in parallel, as well as how to design programs to effectively take advantage of the speed increases PDC provides. As with CS1, multiple modules exist to reinforce the instruction provided in the course lecture sections. The first allows the students to again observe speed-up of parallel programming by implementing a parallelized bubble sort. The second works with image modification, but this time utilizing *pipelining* and the *producer-consumer* model of parallelization

Simple Bubble Sort with Merge for CS2: Even though the student should have covered sorting before attempting this lab, the module gives a brief description of Bubble Sort with examples for review. The lab exercise then describes a simple method for parallelizing the sort using domain decomposition. The computation occurs in two phases, the first of which divides the work equally among the available processors. A second phase occurs after all of the processors are finished with the initial sort, because sorting the pieces of the array does not result in a completely sorted array. In this step, the master must merge sorted pieces to produce a completely sorted result. However, the second phase must be done in serial using a single processor.

Parallel Image Processing for CS2: The CS2 image processing lab is similar to the CS1 image processing lab but follows the producer-consumer paradigm. This module does not apply gray-scaling in parallel followed by flipping in parallel, but instead the lab describes the image processing concept of pipelining filters as shown in . By utilizing a pipeline and the producer-consumer model, the students are able to gray-scale the image and flip the pixels in the same loop. In other words, once the gray-scale filter has been applied to a single row, that row can be enqueued to the flip filter while the gray-scale filter moves to the next row.

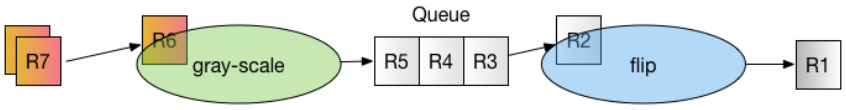

Figure 2: Implementing a pipeline with a queue

4 Evaluation

We assessed how our integration efforts affected our students' ability to think effectively using parallel concepts and the knowledge gained in PDC topics. As part of this assessment, we have conducted subjective and objective evaluations of the knowledge transfer. The objective evaluations were accomplished through quizzes, lab assignments, and homework, which is reflected in the course grade. The subjective evaluation was achieved through pre and post surveys designed to gather the students' self-evaluation of their understanding of PDC concepts. We assessed the self-evaluations on a five point Likert scale to subjectively gauge their understanding of the concepts taught during the coursework.

Results for this study were gathered from students in multiple sections of CS0, CS1 and CS2 courses over three semesters; fall 2015, spring 2016 and spring 2017. Due to time constraints with the existing curriculum and faculty capabilities this was a very sporadically applied implementation, which is something that we hope to address in the future. The class sizes for the courses under study have varied during the implementation of this study, see Table 1, but while the lecture size has fluctuated greatly, the associated lab sections have stayed around a 40 student enrollment on average.

Table 1: Enrollment in Courses

Course / Semester	Section	Lecture Size	Laboratory Size
CS0 FA15	001	44	N/A
	002	43	N/A
CS1 SP16	002	60	51
CS2 SP16	001	37	39
	002	54	49
CS1 SP17	001	103	36
	002	103	34
	003	103	36
	004	91	25

Grades for the PDC module assignments followed the general template for laboratory work in the CS1/CS2 computer classes at TTU. If the assignment is complete and on time, the user is given full credit, work with errors are reduced in score either by 25% or 50% depending on the severity of the errors present. Regardless of errors, as long as work is submitted the student scores a 25%. Based on this scale, the classes we observed have performed below average on the PDC lab. The 2016 CS1 averaged a 67.9% on the PDC lab and those same students finished the semester with an average lab grade of 78.4%.

Meanwhile, the 2017 students averaged a 72.6% on the PDC lab and finished the semester with an 80.9% average in the course. This is to be expected considering the overall lack of experience and limited time the professors were able to spend covering the PDC material prior to the work being accomplished.

Figures 3 through 5 show the results of the students' self-evaluation of their understanding of PDC concepts. These evaluations were done using a 5-point Likert scale (1 – none to 5 – a great deal). From these evaluations we can see that race conditions appear to be one of the hardest PDC topics to understand for CS0 and CS1. We can also see that the number of responses of 'None at All' and 'Little' decrease from CS0 to CS1, we can also see the responses for 'A Lot' and 'A Great Deal' increase between CS1 and CS2. While our implementation was sporadic, these changes are to be expected as the students' aptitude and exposure to programming has increased.

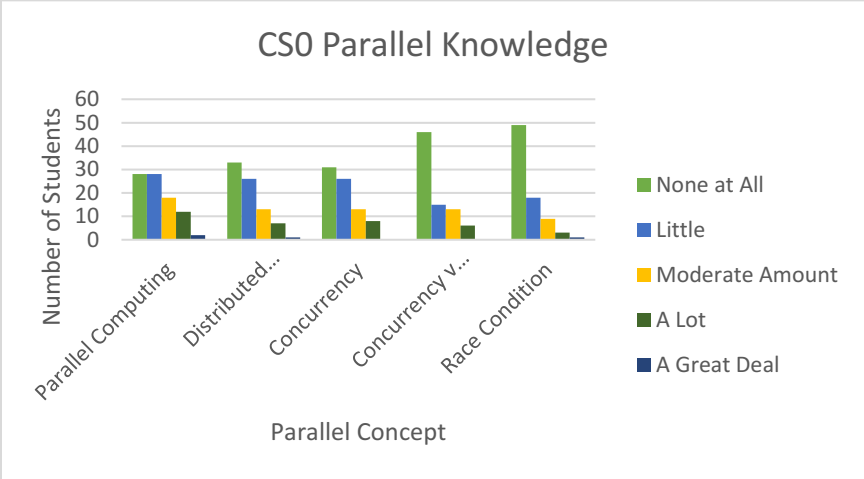

Figure 3: Learning Outcomes for CS0

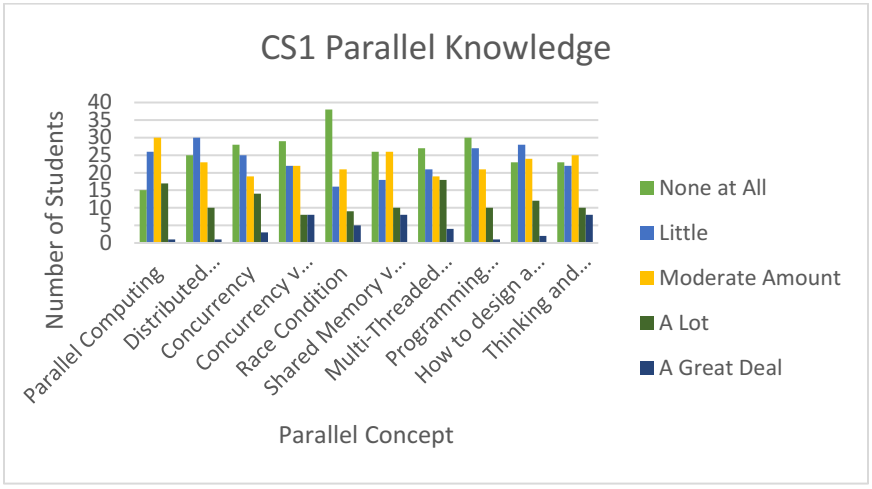

Figure 4:Learning Outcomes for CS1

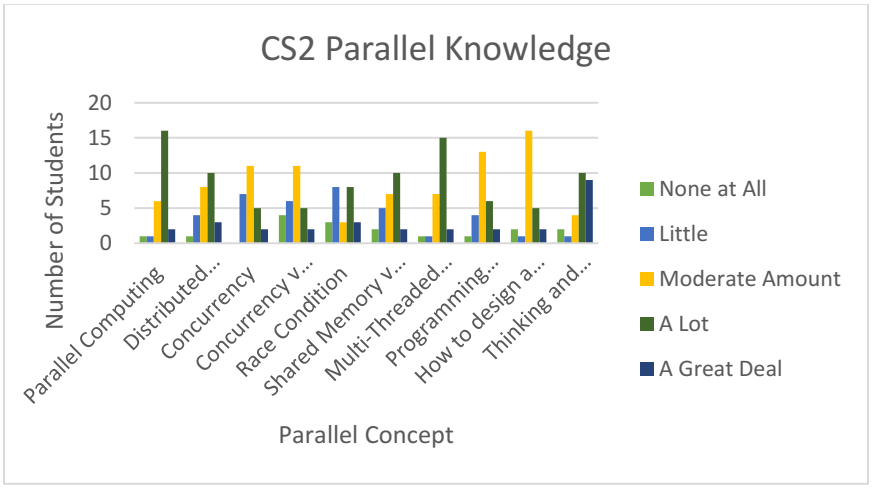

Figure 5: Learning Outcomes for CS2

5 Conclusion

Over the past two years, we have attempted to introduce PDC concepts into multiple sections of the CS0, CS1 and CS2 coursework at TTU. These implementations, limited though they may be, have been somewhat successful and point to several promising

outcomes moving forward. The biggest challenge we faced was the time constraints that were placed upon us due to the nature of these courses and the amount of material already present in their curriculum. This challenge made implementation of the necessary PDC material very difficult. Despite this, the subjective analysis of the results from the implementation show that the students can learn this material at this point in their academic careers and it is feasible to introduce these concepts in early classes.

A second lesson we learned is that students tended to learn more from doing the PDC labs and homework rather than just listening to the lectures. Part of this is the trial and error learning that occurred as the users attempted to solve the problems presented, but also that we waited too long into the semester to begin talking about the concepts. In CS1, the PDC lab was the 10th out of 13 labs, and we feel that if we could introduce the concepts sooner in the semester before the students had started tuning out the lectures, we would be more successful in imparting the necessary skills.

A third lesson is we need to formalize the introduction. The work we accomplished was only possible in a rather scattershot manner and instead we will work with the entire faculty teaching the CS0, CS1 and CS2 courses to develop lesson plans that will fit into their existing coursework and allow us to test the early and often paradigm over the course of several semesters to ensure the knowledge retention. For this to work will require coordination between all members of faculty responsible for teaching these courses and buy in to support the introduction of these topics.

Though we have not tested the theory yet, we believe including unplugged activities that demonstrate parallel concepts away from the computer will be beneficial and should be included in future implementations. We would also like to include concepts of distributed computing in future research, possibly adding them to web based activities in CS0 or coding assignments in CS2. Regardless, we still believe the topics introduced should be presented in small, bite size doses because of variations in student preparedness at this early point in their careers.

References

1. *Parallel Computing in the Computer Science Curriculum*. [cited 2017 01/11/17]; Available from: http://csinparallel.org/index.html.
2. ACM. *Computer Science 2013: Curriculum Guidelines for Undergraduate Programs in Computer Science*. [cited 2017 01/11/17]; Available from: http://www.acm.org/education/CS2013-final-report.pdf.
3. *Digest of Education Statistics, 2015*. [cited 2017 1/16/2017]; Available from: https://nces.ed.gov/programs/digest/d15/tables/dt15_317.10.asp?current=yes.

4. Sushil K. Prasad, A.G., Arnold Rosenberg, Alan Sussman, Charles Weems, *CDER Center | NSF/IEEE-TCPP Curriculum Initiative.*
5. Ko, Y., B. Burgstaller, and B. Scholz. *Parallel from the beginning: The case for multicore programming in thecomputer science undergraduate curriculum*. in *Proceeding of the 44th ACM technical symposium on Computer science education*. 2013. ACM.
6. Adams, J.C. *Injecting parallel computing into CS2*. in *Proceedings of the 45th ACM technical symposium on Computer science education*. 2014. ACM.
7. Geist, R., J.A. Levine, and J. Westall. *A problem-based learning approach to GPU computing*. in *Proceedings of the Workshop on Education for High-Performance Computing*. 2015. ACM.
8. Lupo, C., Z.J. Wood, and C. Victorino. *Cross teaching parallelism and ray tracing: a project-based approach to teaching applied parallel computing*. in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 2012. ACM.
9. Burtscher, M., et al. *A Module-based Approach to Adopting the 2013 ACM Curricular Recommendations on Parallel Computing*. in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 2015. ACM.
10. Graham, J.R., *Integrating parallel programming techniques into traditional computer science curricula.* ACM SIGCSE Bulletin, 2007. **39**(4): p. 75-78.
11. Neelima, B. and J. Li. *Introducing high performance computing concepts into engineering undergraduate curriculum: a success story*. in *Proceedings of the Workshop on Education for High-Performance Computing*. 2015. ACM.
12. Brown, R. and E. Shoop. *CSinParallel and synergy for rapid incremental addition of PDC into CS curricula*. in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. 2012. IEEE.
13. Brown, R. and E. Shoop. *Modules in community: injecting more parallelism into computer science curricula*. in *Proceedings of the 42nd ACM technical symposium on Computer science education*. 2011. ACM.
14. Adams, J., R. Brown, and E. Shoop. *Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to CS undergraduates*. in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. 2013. IEEE.
15. Foley, S.S. and J. Hursey. *OnRamp to parallel and distributed computing*. in *Proceedings of the Workshop on Education for High-Performance Computing*. 2015. ACM.
16. Prasad, S.K., et al., *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*. 2015: Morgan Kaufmann.
17. Harvey, B., et al. *Snap!(build your own blocks)*. in *Proceedings of the 45th ACM technical symposium on Computer science education*. 2014. ACM.