Parallel and Distributed Computing Across the Computer Science Curriculum

David J. John
Department of Computer Science
Wake Forest University
Winston-Salem, NC 27109
Email: djj@wfu.edu

Stan J. Thomas
Department of Computer Science
Wake Forest University
Winston-Salem, NC 27109
Email: sjt@wfu.edu

***Abstract*—Two recent curriculum studies, the ACM/IEEE Curricula 2013 Report and the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing, argue that every undergraduate computer science program should include topics in parallel and distributed computing (PDC). Although not within the scope of these reports, there is also a need for students in computing related general education courses to be aware of the role that parallel and distributed computing technologies play in the computing landscape. One approach to integrating these topics into existing curricula is to spread them across several courses. However, this approach requires development of multiple instructional modules targeted to introduce PDC concepts at specific points in the curriculum. Such modules need to mesh with the goals of the courses for which they are designed in such a way that minimal material has to be removed from existing topics. At the same time the modules should provide students with an understanding of and experience employing fundamental PDC concepts. In this paper we report on our experience developing and deploying such modules.**

***Keywords*—parallel computing; distributed computing; computer science education**

I. Introduction

Both the ACM/IEEE Curricula 2013 Report [1] and the NSF/IEEE-TCPP [15] report address the urgent need for all computer science majors to meet basic proficiencies in parallel and distributed computing. The ACM/IEEE Curricula report specifically recommends that all undergraduate majors complete five Tier 1 and ten Tier 2 hours in parallel and distributed computing. These specific hours, shown in Table I, are spread among five topics: Fundamentals, Decomposition, Algorithms and Analysis and Programming, and Architecture.

One approach to satisfying the recommendations of the reports is to create a new core computer science course. For many computer science programs this is an expensive suggestion, both in terms of faculty and student workloads. An alternative plan of action spreads these recommended hours throughout existing core courses, which is the subject of this paper. Of course, elective parallel and distributed computing courses are encouraged when feasible and consistent with a program's objectives. At Wake Forest, two elective parallel and distributed computing upper level undergraduate courses, *CSC346–Parallel Computation* and *CSC391–Special Topics: CUDA Programming*, are offered regularly.

In order to successfully incorporate 15 hours of parallel and distributed computing into existing core courses, two factors must be considered. First, the incorporated material must fit appropriately into existing courses. The addition of parallel and distributed computing hours into any course must be consistent with the students' abilities and backgrounds and must not be orthogonal to achieving the existing goals of the course. Second, each department must agree upon modification of the course goals. This modification is never easy. In fact, the ACM/IEEE Curricula 2013 Report [1] recommends the inclusion of additional hours in other knowledge areas as well, which further complicates modification of the goals of core courses.

Most computer science departments offer undergraduate courses for non-majors, sometimes referred to as CS0, that provide an overview of computer science. These students, most of whom will not take additional computer science courses, also need to understand the need for, the relevancy of, and the importance of parallel and distributed computing. Developing instructional modules on PDC for this type of course is very challenging. These students will be future beneficiaries of parallel and distributed computing, and some will become policymakers. In Table I, three hours of Tier 0 instructional material on parallel and distributed computing are recommended for computer sciences courses that satisfy undergraduate general education requirements. Note that the Tier 0 moniker is used here to refer to material associated with general education, non-major, courses in a computer science curriculum. It is not used in ACM/IEEE Curricula 2013.

Table I shows the recommendation for 3 Tier 0 hours of parallel and distributed computing in CS0. Specifically, these three hours should focus on Parallelism Fundamentals (What is meant by parallel computing? What is the goal of parallel computing?), Parallel Decomposition (How do a number of processors contribute to the solution of a problem?), and Parallel Architecture (What parallel and distributed computing architectures are available today?). The CS0 student must also have a laboratory experience that makes these ideas tangible. Since most CS0 students have virtually no programming background, this parallel and distributed computing laboratory must appeal to the students' visual or auditory senses. As with core computer science courses, departments will need to modify the goals of CS0 to include these important topics.

II. Instructional Modules

The development of a library of instructional modules is a case based approach that can aid the instructor of either a core

TABLE I. RECOMMENDED HOURS OF INSTRUCTION ON PARALLEL AND DISTRIBUTED TOPICS BY KNOWLEDGE AREA

	Tier 0 hours	Tier 1 hours	Tier 2 hours
Parallelism Fundamentals	1	2	
Parallel Decomposition	1	1	3
Communication & Coordination		1	3
Parallel Algorithms, Analysis & Programming			3
Parallel Architecture	1	1	1

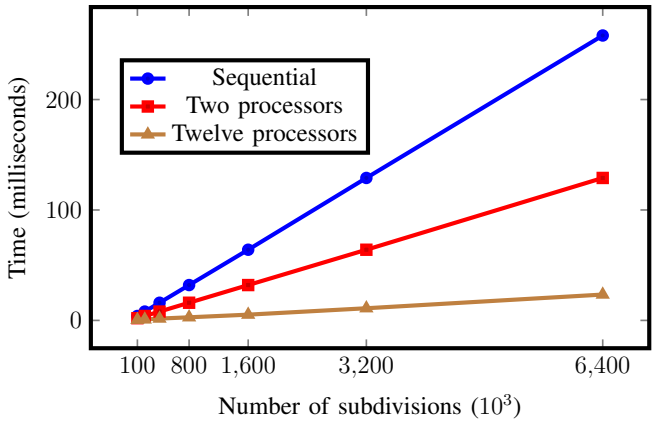
computer science course or CS0 to appropriately incorporate parallel and distributed computing. Each instructional module should consist of four components: written materials, program examples with appropriate software tools, instructor led classroom discussions, and student assignments. All together these components provide a hands-on opportunity to execute and analyze parallel and distributed programs, and a better understanding of both the impact of parallel architectures and parallel and distributed computing.

How can a program spread 15 hours of new instructional material across its core courses? If the task is spread across 5 courses, one week of each core course will need to be devoted to parallel and distributed computing. For the instructor, a typical reaction is "how can I carve out one week". For the student, the danger is that these 5 one week modules are sparsely distributed across their computer science core courses and appear to lack overall cohesion. In order to help the students, the written materials provided to the them must be written in a similar style using a common layout. Following a recommendation made at the LittleFe Buildout at Super Computing 2012, our written materials have all been developed in *reStructedText* [10] and reformatted to web content using Sphinx [3]. Encouraging all faculty to develop a module is an important goal; however, it is also important that there be an instructional module editor-in-chief to ensure consistency across the various instructional modules. The programming language and parallel and distributed paradigm used for a particular module should be the decision of the instructor of the course. This means there may be multiple versions of similar modules.

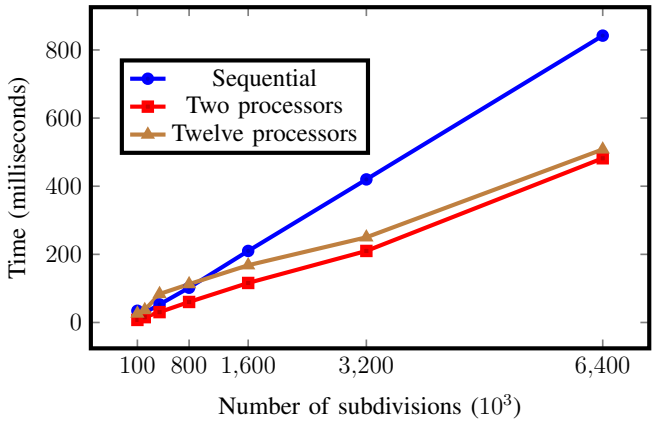
The addition of 3 parallel and distributed computing hours to CS0 is less problematic, since many of these courses consist of a number of relatively independent units. However, even though only a few of these students will take a core computer science course, it is important that the CS0 instructional modules have the same "look and feel" as the core course modules.

The designation of instructional modules as either *Introductory* or *Intermediate* indicates correlation primarily to ACM/IEEE recommended Tier 1 or Tier 2 hours, respectively. The designation of *Elementary* corresponds to the Tier 0 parallel recommendation shown in Table I. A module designation of *Advanced* implies the students have completed their Tier 1 and Tier 2 required topics and are working to develop, analyze and implement parallel programs, likely in an upper level undergraduate course.

Discussions of three instructional modules we have developed are presented in sections II-A - II-C. Students have available to them a number of computers that support parallel and distributed computing. Multicore laptops and smart phones, both with GPUs, are routinely integrated into their activities, but often students are not aware of the awsome computing potential in their hands. For PDC instruction we use a number of different devices which support parallel and distributed computing. In Sections II-A and II-B the students use a Dell PowerEdge C6145 (four 16 core computing elements connected via a 40 Gb interconnection network running Ubuntu). Often, the instructor demonstrates, in the classroom, parallel and distributed computing using a LittleFe [12] (six cards with dual computing elements connected via a 1 Gb Ethernet running BCCD [2]). The classroom use of the LittleFe provides the opportunity for students to observe and to touch a working cluster computer. Too often the only student access to a shared highly parallel computer is exclusively through a computer screen.



(a) Trapezoidal rule performances on Dell PowerEdge C6145



(b) Trapezoidal rule performances on LittleFe

Fig. 1. For the Introductory CS2 module: plots of execution time of the trapezoidal rule versus the number of subintervals for 1, 2 and 12 computing elements on two different architectures.

A. *Introductory module for CS2*

Students enrolled in CS2 (*CSC112–Fundamentals of Computer Science*) are generally good beginning programmers, but only with sequential programs. Most have participated in an *Elementary* or *Introductory* module in a previous course. An *Introductory* module designed for CS2 students must build on their backgrounds in an appropriate way. The *Introductory* module presented here focuses on numerical quadrature

because (1) all the students have seen this idea already in a calculus course, either in high school or college, and (2) the programming prerequisites for numerical quadrature have been mastered through their previous and current studies in CS1 and CS2, respectively.

Three hours of instruction are designated for this particular module which satisfies 3 of the ACM/IEEE recommended hours. Each student is provided access to a Dell PowerEdge C6145 with four 16 core computing elements running Ubuntu. All the supplied implementation codes (sequential and parallel) are stored in subdirectories within their accounts. It is important to make early experiences of compiling, running and analyzing a distributed program as successful as possible.

The implementation of this module uses *c++* with MPI. The required programming language for our CS2 is *c++*. However, as mentioned previously in Section I, the instructor can decide on any parallel paradigm for the module. Hence, another version of this module is available where the implementation is *c++* with CUDA.

Before the first module class day, students are provided a link to the instructional module written material, along with a reading assignment which covers the mathematical background and a version of the sequential algorithm. During the first class, the instructor quickly reviews the mathematics, and covers more thoroughly the sequential algorithm. The *c++* code is presented which implements the sequential algorithm; students have every opportunity to ask questions. Towards the end of the first module lecture the question is raised "If our goal is to decrease the overall execution time for this task, how should we design a new parallel algorithm?" With some guidance, the students are led to the intuitive idea of partitioning the subintervals of integration, assigning a separate computing element to integrate across each partition, and then summing all the individual integrals. The homework assigned at the end of this first day is to execute the sequential program and analyze its execution time. Specifically, the students plot the execution time as a function of the number of subintervals used in the numerical integration. It is worth noting that there will be students who have difficulties with this first assignment.

At the beginning of the second class, the students are encouraged to share observations from their homework. Most discovered that the execution time is a linear function of the number of subintervals, as shown by the blue line in Fig. 1(a). Next, the discussion leads to the development of a parallel algorithm, based on the ideas from the previous class. Then, with instructor assistance, the class develops a completed parallel implementation, based on *c++* with MPI. Sufficient class time is required to introduce the distributed computing model, and to give an overview of how the MPI functions necessary for this parallel program (*Init()*, *Get_size()*, *Get_rank()*, *Bcast*, *Reduce()* and *Wtime()*) work and interact. The next assignment is to compile, link and execute this parallel program varying the number of computing elements. Examining four executions with 2, 4, 16 and 32 computing elements, the students are to plot the execution times as a function of the number of subintervals, similar to Fig. 1(a). This assignment is started in class and completed as homework.

As an interesting complement to this module, the instructor can use the LittleFe as a classroom aid. A plot for sequential and parallel execution times with the LittleFE is presented in Fig. 1(b). This plot raises significantly interesting questions which leads into a discussion of architectural differences and and network latency.

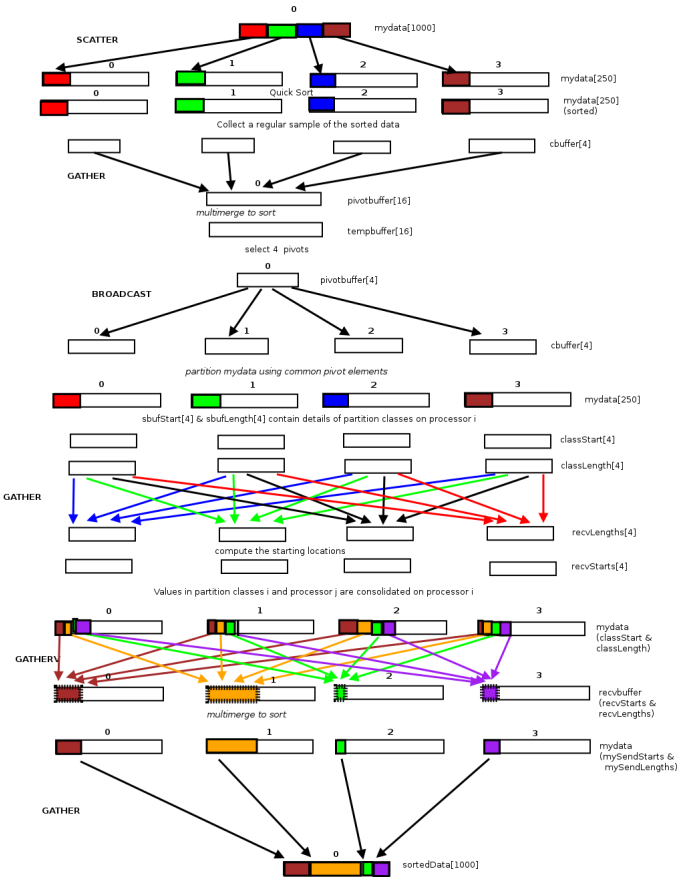


Fig. 2. For the Intermediate Algorithms module: data flow diagram of PSRS algorithm applied to a list of size $n = 1000$ with $p = 4$ processors.

Our classroom experience with this module has been quite successful. The students completed the assignments, and received a very hands-on experience in determining speed up. In terms of ACM/IEEE recommended topics, Table I, the students spent 3 hours on parallel and distributed computing: 1 hour on Tier 1 *Fundamentals*, 1 hour on Tier 2 *Communications and Coordination*, and 1 hour on Tier 2 *Algorithms, Analysis and Programming*.

B. Intermediate module for Algorithms

Students enrolled in the Algorithms (*CSC222–Data Structures and Algorithms II*) course should have some prior experiences with several parallel and distributed computing instructional modules. The instructional modules for Algorithms must take into account these previous experiences as well as the students' data structures knowledge. This instructional module highlights the Parallel Sorting with Regular Selection (PSRS) algorithm published by Li, Lu, Schaeffer, Schillington, Wong and Shi [11]. The reasons that this sorting algorithm is presented include that it is a balanced sorting algorithm, reasonable to analyze, requires exchanges of information of various different lengths, and depends upon different parallel

tasks during the distinct execution phases. This algorithm requires that the students review the QuickSort, Partition, and Merge algorithms. A sequential utility function, Multimerge, is introduced that is based on a priority queue.

As already mentioned, the students will use the Dell PowerEdge C6145 for their assignments, and the instructor will use it and also the LittleFe. This module is designed for three hours of classroom lecture and discussion, and several days of independent laboratory exercises leading to satisfaction of 3 Tier 2 hours. It is assumed that the students can modify an existing program, but are not yet ready to write a complete parallel program.

Prior to the first class meeting, the students are provided access to the written text of the instructional module. A *c++* with MPI implementation of the algorithm, along with a *Makefile*, is stored in a subdirectory of the student's home directory. Their initial assignment is to read through the entire module and to be prepared for an in class discussion on the first two pages of the module, focusing on the prerequisite algorithms of Quicksort, Partition and Merge, and on the data flow of the PSRS algorithm and the algorithmic description, Figs. 2 and 3, respectively. Included in the algorithmic description is both space and time complexity relevant information. The data flow diagram is tightly tied to MPI functions and underlying supporting data structures. The students' assigned homework after this first lecture and discussion is to determine the run time and space complexities of each of the phases, and assuming no communication costs, to approximate the speedup of PSRS over QuickSort. The students are also assigned to read again through the *c++* with MPI implementation of the algorithm and to be prepared for an inclass discussion.

The *c++* with MPI implementation is the focus of the second class meeting. The instructor's presentation includes the algorithm's implementation details, highlighting MPI functions used as well as specific MPI features such as *MPI_IN_PLACE*, and the mechanism to exchange lists of variable sizes. The instructor must ensure that the students grasp the connection between the data flow and the algorithm phases, Figs. 2 and 3, and the implementation organization. The assigned homework is to plot, for various numbers of processors, the execution times of PSRS as a function of the size of the list to sort. Depending upon the instructor, the provided code may be incomplete, which then gives the students the opportunity to develop and implement the final code fragments.

The last class meeting is used primarily to discuss the students' findings. On the Dell PowerEdge, the execution plots will strongly suggest that doubling the number of processors will correspondingly halve the execution time, yielding a speedup equal to the number of computing elements. The instructor will demonstrate a somewhat different conclusion with a machine with a different architecture, the LittleFe. When the number of processors allocated on the LittleFe is quite small the performance plots for the LittleFe suggest the same speedup as on the Dell computer. However, a very different result is suggested when using a larger number of processors on the LittleFe. These performance plots similarities and differences leads into an important discussion about the influence of multicore communication and communication complexity on the speedup on an algorithm.

Phase I: Initialization
Initialize p processors. The root processor (0) gets the list of size n.

Phase II: Scatter list, quicksort, collect regular samples
Scatter the list to the p processors. Each processor *quicksorts* its local list, roughly of size $\frac{n}{p}$. Each processor regularly chooses p sample points from its sorted list.

Phase III: Gather and multimerge sample points, choose and broadcast $p-1$ pivots
The root processor *gathers* the p sets of p sample points which are then sorted using *multimerge*. From the p^2 sorted points, $p-1$ pivot values are regularly chosen and *broadcast* to the other $p-1$ processors.

Phase IV: Local lists are partitioned
Using the $p-1$ pivots, each of the p processors *partitions* its locally sorted list, roughly of size $\frac{n}{p}$, into p classes.

Phase V: All i^{th} classes are gathered and multimerged
Processor i *gathers* the i^{th} classes from the other $p-1$ processors. These are sorted using *multimerge*.

Phase VI: Root processor assembles the sorted list
The root processor *gathers* all the data from the p processors and assembles the sorted list of size n.

Fig. 3. For the Intermediate Algorithms module: a phase oriented description of the PSRS algorithm. This description specifies the communication paradigms using MPI function names. The algorithms of *multimerge*, *partition*, and *quicksort* are used.

This instructional module provides 3 Tier 2 hours of parallel and distributed computing, 1 hour in each of *Decomposition*, *Communication and Coordination*, and *Algorithms, Analysis and Programming*. As well, the students have used their (sequential) algorithm analysis skills in the first homework, likely learned about a new application of a priority queue with MultiMerge, and extended their understanding of distributed computing by learning about exchanging variable length information in MPI.

C. Elementary module for CS0

In our curriculum, students in the CS0 (*CSC101–Overview of Computer Science*) course are expected to develop a conceptual understanding of programming concepts and computer organization but are not expected to develop programming skills. Thus a PDC module at this level can be a demonstration module with limited opportunity for modification by students rather than an implementation module. The module described here is based on the calculation and display of the familiar two-dimensional Mandelbrot set [7]. It is designed to visually convey to students an understanding of how multiple computing cores can be utilized to speed up computations, even in a commodity laptop or mobile device.

Students at this level of study may be familiar with images of the Mandelbrot set but not with the underlying iterative computation. Although Mandelbrot set images are created by sampling complex numbers and determining whether or not the value of an iterated function converges or diverges at each point, students do not need to understand complex numbers to learn from this module. The real and imaginary parts of each complex number can simply be treated as image coordinates

and the pixels colored according to how rapidly the sequence diverges, if at all.

This module is implemented in Python, for several reasons. The source code for Python tends to be more concise and more easily understood by non-programmer's than c++ code. The use of Python's multiprocessing module makes it easy to understand, at least conceptually, how a pool of processors in a multi-core system can be utilized. Students should understand that software for parallel computing can be developed in many modern programming languages. And, finally, Python is used so that this module can easily be repurposed and extended by students in CS1 (*CSC111–Introduction to Computer Science*) learning Python as a first programming language.

The module itself consists of a serial computation of the Mandelbrot set, a parallel computation of the Mandelbrot set, and accompanying lecture materials. The serial implementation is straightforward, although it does employ the Python *numpy* module for manipulating rows of pixels. The Python *animation* class from the *matplotlib* module is used to display a row at a time of the computed image, providing a visual display of the speed of the computation. Although the graphical display slows down the overall computation, the purpose of the module is not to compute the Mandelbrot set quickly and efficiently. The parallel version of the code uses the identical function as the serial version for computing a single row of pixel values. However, in the parallel version, a pool of processes, one per core by default, is used to distribute the computation of pixel row values across cores. Once again, the *animation* class is used to display the computed pixel values a row at a time. The visual impact is straightforward yet effective. It demonstrates in a visual way that a moderately intensive computational problem can be solved more quickly by distributing it across multiple cores.

The lecture material for the CS0 introduction to PDC concepts begins with a description of multicore processors and how they differ from earlier generations of CPUs. Some basic issues such as shared memory contention can be discussed at this level whereas more advanced topics, such as cache coherency, are not appropriate. Altogether this presentation and the follow up described below constitute at least one hour of Tier 0 material on Parallelism Fundamentals and one hour of Tier 0 material on Parallel Architecture. Following the discussion of this introductory material on shared memory architectures, the Mandelbrot serial and parallel code examples can be employed to motivate an intuitive understanding of how programs can utilize multicore processors. This discussion includes at least one hour of material on Parallel Decomposition of problem spaces. In some situations the instructor may want to have students experiment with different starting values for the Mandelbrot set calculation for the experience of generating visually interesting patterns. Although this hands on experience may not add greatly to student understanding of PDC concepts, it may entice them to further explore computer programming.

As a follow up to the topic of shared memory parallel computing, this module also introduces basic concepts of message passing parallelism. In this segment the instructor often uses a LittleFe unit to explain the organization and function of computing clusters. Simple demonstrations such as an n-body computation or an artificial life simulation with graphical output can be displayed while the instructor talks about important issues such as communication costs and concurrency. Students can be encouraged to consider and discuss the problems that arise when a collection of compute nodes is scaled from a single rack to a server room to the globe. This may be the first time that many students have thought about the implications of distributed computing at the cloud level.

III. Conclusion

For several years there have been, and will continue to be, many efforts to give parallel and distributed computing a more prominent role in undergraduate computer science education [4], [5], [8], [9], [13]. The current proliferation of parallel and distributed devices, along with the ACM/IEEE Curricula 2013 Report recommendations, will, in all likelihood, be the catalyst to finally make this a reality.

As articulated in the ACM/IEEE Curricula 2013 Report [1] and the NSF/IEEE-TCPP [15] report it is imperative that all computer science majors learn about parallel and distributed computing. It is equally important for students in computer science courses meeting undergraduate general education requirements to learn about the importance, potential, and relevancy of parallel and distributed computing. As seen in Table I, the Tier 0 recommendation provides the framework for PDC topics in CS0.

One approach to meeting the ACM/IEEE Curriculum 2013 Recommendations is to create a single new core course which satisfies the 15 hours of recommended curriculum and then expands upon that base significantly. However, the recommendation here is that the 15 recommended hours be integrated throughout the core courses (for many departments that would amount to 3 hours of PDC instruction per core course). This distributed approach can satisfy the ACM/IEEE Curriculum 2013 recommendation both in terms of hours and specific content. As well, quite a number of student majoring in other subjects, for example mathematics and physics, take one or two core computer science courses to complement their majors; they will significantly benefit from their, albeit limited, parallel and distributed computing experiences. For the students, seeing parallel and distributed computing in each core course can only underscore its importance to them in their future careers. Every faculty member has the opportunity to create a module for their favorite topic in a core course. They are not limited by either programming language or architecture. For the department, the ACM/IEEE Curricula 2013 recommendation for parallel and distributed computing can be incorporated into the goals of their undergraduate major without the expense of an additional core course.

Well designed and concise instructional modules can be the vehicle to successful integration of parallel and distributed computing across core courses and general education courses. For these to be successful, it is important that there be a common style and format for the modules. This facilitates the students' comprehension of the comprehensive PDC instructional goal. For this commonality to occur, there must be a departmental module editor-in-chief. Another necessary factor for success is the involvement of many faculty in the development and updating instructional modules. The most difficult module to develop is that first one; however, once

a collection of modules is developed they will be used as a template by others.

The establishment of a departmental library of instructional modules is beneficial to the faculty teaching core and general education computer science courses. The computer science community needs to become successful at sharing instructional materials such as these modules. Currently, Shodor [14] and CSinParallel [6] have established mechanisms to facilitate the sharing of PDC instructional materials. These instructional modules described herein are publicly available on the web.

Two outstanding concerns remain as major challenges. Effective assessment instruments for each module must be developed based on the specific module goals. These must include both pre-module and post-module assessment. The *Instructional module for CS2* (II-A) has been presented several times, and currently we are developing assessment tools specific for it. Table I shows not only the recommended number of hours (columns), but also the recommended topic coverage (rows). Our efforts thus far have focused directly on satisfying the recommended hours. The more difficult task using a collection of modules is satisfying the recommended hours in each topic. Few computer science undergraduate programs want to develop accounting protocols to monitor this for each undergraduate major. Certainly, as more work is invested into the development of modules, it is important for module creators to tackle these two challenges.

REFERENCES

[1] ACM/IEEE-CS Joint Task Force on Computing Curricula. Computer science Curricula 2013: Curriculum Guidelines for Undergraduate Programs in Computer Science. Available online at http://dx.doi.org/10.1145/2534860. ACM and IEEE Computer Society, December 20, 2013.

[2] BCCD group. Bootable Cluster CD. http://bccd.net

[3] George Brandl. *Sphinx: Python Document Generator*, 2013. http://sphinx-doc.org.

[4] R. Brown and E. Shoop, "Modules in community: injecting more parallelism into computer science curricula," in *SIGCSE '11: Proceedings of the 42nd ACM technical symposium on Computer science education*. New York, NY, USA: ACM, 2011, pp. 447–452.

[5] R. Brown and E. Shoop, "CsInParallel and synergy for rapid incremental addition of PDC into CS curricula," *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, vol. 0, pp. 1329–1334, 2012.

[6] CsInParallel: Parallel Computing in the Computer Science Curriculum. http://serc.carleton.edu/csinparallel.

[7] A. K. Dewdney, Computer Recreations: A computer microscope zooms in for a look at the most complex object in mathematics, *Scientific American*, 253(2):16-24, 1985.

[8] David J. John. Integration of parallel computing into introductory computer science. *SIGCSE Bulletin*, 24(1):281–285, 1992.

[9] David J. John. NSF supported projects: Parallel computation as an integrated component in the undergraduate curriculum in computer science. *SIGCSE Bulletin*, 26(1):357–361, 1994.

[10] Richard Jones. *A ReStructured Text Primer*, 2013. http://docutils.sourceforge.net/docs/user/rst.

[11] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Schillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, October 1993.

[12] LittleFe group. LittleFe: Parallel and Cluster Education on the Move. http://littlefe.net/home.

[13] Chris Nevison, Daniel Hyde, Michael Schneider, and Paul Tymann, editors. *Laboratories for Parallel Computing*. Jones and Bartlett Publishers, 1994.

[14] Shodor: A national resource for computational science education. http://www.shodor.org.

[15] TCPP Curriculum Working Group. NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing – core topics for undergraduates. Technical report, NSF/IEEE-TCPP, December 2012.