Limited Time and Experience: Parallelism in CS1

Steven A. Bogaerts
Department of Computer Science
DePauw University
Greencastle, IN, U.S.A.
stevenbogaerts@depauw.edu

***Abstract*—With the growing recognition of the importance of parallel and distributed computing (PDC) in computer science, it is crucial to offer students early experience – even in CS1. At the same time, PDC material must be handled very carefully when presented to students with so little background, in order to avoid confusion and a reduction in interest. This paper considers two sections of CS1 that used distinct approaches in introducing the same core set of PDC concepts. A number of specific strategies are presented, as well as a detailed analysis of pre- and post-test results.**

***Keywords*-parallelism; CS1; CS education; curriculum**

I. Introduction

The computer science education community widely agrees that parallel and distributed computing (PDC) is an increasingly vital subject area. It is no longer sufficient to consider PDC only in a high-level elective course. Rather, all computer science majors should be exposed to key PDC concepts before graduation, as recommended in the ACM CS2013 curriculum guidelines [1].

The primary discussion now is *how* these concepts should be integrated into existing curricula. Options include adding a new PDC course to the core major requirements, or integrating the concepts into multiple other courses. While it is now more common to find PDC topics included in courses as early as CS2, including PDC in CS1 poses a particular challenge. The benefit of such early integration is that students will see PDC as a natural and common part of programming, instead of an advanced and rarely-used concept. The challenge, of course, is that PDC brings unique cognitive and technical problems to programming that can be particularly difficult to tackle early in a curriculum.

This paper will begin by considering related work in integrating PDC concepts into CS1. It will then examine two sections of a CS1 course offered fall 2013, sharing detailed strategies used in exploring parallelism. A pre- and post-test will be described, followed by results and a comparative analysis between the two sections. The analysis will conclude with recommendations for the successful integration of parallelism into CS1 based on these contrasting experiences.

II. Connections to Other Work

Educators are increasingly working to integrate PDC throughout the CS curriculum. Again, the ACM CS2013 curriculum guidelines [1] are an authoritative statement of recommended PDC topics for all computer science programs. Another important work in curriculum recommendations is the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing [2], which outlines many topics and recommended coverage based on Bloom's taxonomy [3].

Much work has been done in the consideration of general strategies for PDC integration into the curriculum. For example, parallelism can be used as a *medium* for the exploration of traditional course topics, in order to facilitate integration into courses that are already full of material [4], [5]. Others describe in detail a "spiral approach" for deepening exploration of PDC throughout a curriculum [6]. As described below, elements of both of these strategies are present in the work of this paper and how it fits in long term department plans.

In considering what PDC topics can be explored in CS1, both technical and non-technical approaches have been taken. Technical approaches typically choose a single technology to use, perhaps in a simplified way, as a hands-on context for exploration. For example, successful efforts have included C and OpenMP [7], core Java concurrency facilities [8], and Hadoop [9]. The work described in this paper includes some elements of this approach, using core Java concurrency facilities in a simplified way to give some hands-on experience with the course topics.

Some "non-technical" approaches place greater emphasis on broader concepts than on their actual implementation with specific technologies. For example, PDC concepts can be considered with everyday analogies like solving a jigsaw puzzle or making long-distance phone calls [10]. Students can also learn important concepts by playing games in which they play the role of processors acting in parallel [11], [12]. Some have argued that these kinds of approaches develop crucial "parallel thinking" skills while avoiding the peculiar complexities of specific tools [13]. The work described in this paper includes elements of this approach as well, with many concepts explored in a non-technical way through exercises and analogies.

It is interesting to note that some work has also been done in considering concepts of parallelism even before CS1. For example, [14] describes the use of Scratch to explore parallelism in CS0, and [15] describes the use of C and

Table I
TOPICS COVERED FOR THE TWO SECTIONS OF THE COURSE.

Topic	Section A	Section B
Architecture - Multi-core	K	K
Programming - Shared Memory	C	K
Programming - Distributed Memory	C	K
Programming - Task/Thread Spawning	A	K
Programming - Tasks and Threads	C	K
Programming - Synchronization	C	K
Programming - Concurrency Defects	C	K
Programming - Performance Metrics	K	K
Cross Cutting - Why and What Is PDC	C	C
Cross Cutting - Non-Determinism	K	K
Cross Cutting - Power Consumption	K	K

Explicit Multi-Threading (XMT) in a high school course.

III. OVERVIEW OF TWO CS1 COURSES

This section will provide an overview of how concepts of parallelism were integrated into two sections of CS1 in the fall of 2013. This CS1 course is an introductory programming course with no prerequisites and no assumed prior experience. It is taken by both majors and non-majors; in fact, a sizable number of students take the course to fulfill a general education requirement. Section A was taught by the author, and had 28 students. Section B was taught by another faculty member, with 26 students.

Students in both sections took the same pre-test and post-test, as described in Section V. The instructors worked from the same basic outline of material, independently developing lecture materials and exercises from that outline. Both instructors covered the parallelism material around week 12 of a 14 week semester. Section A ended up spending more contact time on the material: just over four hours in total. Section B, in contrast, spent about 90 minutes of contact time, with the professor choosing instead to spend additional time on other, non-parallel topics not covered in section A.

The topics covered in the two sections are summarized in Table I. The topic names are from the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing [2]. The coverage level is based on Bloom's taxonomy [3], from the lowest level, K ("Know the term"), to C ("Comprehend so as to paraphrase/illustrate") to highest level, A ("Apply it in some way"). Note that the two sections covered the same topics, only sometimes at different levels. As expected, section A, which spent more time on the material in general, covered certain topics at a higher level.

The differences in *time spent* on the material result from the way in which the material was *examined* in class. Section B was primarily a lecture-oriented format, interspersed with brief questions and a few exercises. This format was chosen so that the basic material could be covered sufficiently while still allowing time for a few additional non-parallel topics. Section A, in contrast, sacrificed these few non-parallel topics in order to include a higher number of parallelism exercises and activities, as well as an exploration of implementation in Java (though note that the pre/post test includes no Java implementation questions).

IV. SEVERAL ANALOGIES AND EXERCISES

This section will share some illustrations of the presentation of the material as done in section A, suitable for adoption into other instructors' CS1 courses. The approach in section A was characterized by many exercises and analogies offset by short periods of lecture.

A. Parallelism in Real Life

The first set of analogies were about parallelism in "real life":

1) If I can shovel my driveway in 1 hour, how long would you expect it to take if someone equally capable were helping me?
2) If one person can dig a hole with a shovel in 100 seconds, how long would it take for 10 people to dig that hole?
3) If I can run a company all by myself by working 100 hours per week, how much time do I need if I hire an assistant?

The driveway-shoveling analogy is a task that is easy to split up and requires very little communication, and so parallelism is easy to apply and highly effective. The hole-digging analogy is a task that is not easy to split up, so parallelism is less effective. The company analogy describes a task that is very challenging to parallelize, yet crucial nonetheless, due to the size of the task. It can be helpful to explain to students that in a corporate hierarchy, the principle job of those in the higher levels is to parallelize the complex task of running a business – to make sure that their subordinates collaborate effectively to achieve the goals of the company.

These analogies together illustrate that to apply parallelism, the task must be split into pieces, and communication must be organized. After these first analogies, many more can be readily considered by the students, such as multiple check-out lanes at a grocery store, assembly-line manufacturing, the interaction of various physical phenomena to create the weather, and even the interaction of cells in the human body. Further examples of parallel *computation* can then be briefly considered, such as digital signal processing, DNA sequencing, and operations research.

B. Clock Speed

Another early topic of consideration is the notion of clock speed as a measure of processor speed. A student might reasonably ask "Why not just increase the clock speed?" This can be answered with another analogy: Suppose you tell someone to make one peanut butter and jelly sandwich every hour. This would be very easy to accomplish. Every minute? Perhaps with a little practice, this would not be a problem either. Every second? This would be impossible. A person can't just be told to "speed up" beyond a certain point. Rather, some improvement in sandwich-making must be devised, such as a machine in which the operator must simply load the ingredients. Similarly, a processor can't just speed up without some improvement in processor technology. One key improvement that has brought speedup for decades has been the use of smaller and smaller transistors. This of course leads to a discussion of Moore's Law, the physical barriers of tremendous heat generation, and the eventual turning to multi-core processor designs.

C. Interprocess Communication

Another illustration is helpful in considering different kinds of interprocess communication. Suppose a friend and I want to count how many people are in a building. Consider each of three things I might say to my friend:

- "I've counted 27 people in the basement. Please go count the other floors and determine the total."
- "I'll count the basement, you count the other floors. Whenever you finish a floor, send me a text message with that result. I'll merge your results with mine."
- "I'll count the basement, you count the other floors. Whenever you finish a floor, open up our shared Google Doc. Add what you just counted to the total already in the doc, and erase the old total. I'll do the same as I go."

These communication plans correspond to passing data at process creation, message-passing, and shared memory. With these analogies, these communication strategies are considered in a very intuitive way.

Another useful way to explore interprocess communication is with a physical exercise in sorting playing cards. Some volunteers are divided into three groups A, B, and C, containing 1 person, 3 people, and 2 people, respectively. The person in group A plays the role of a single-core processor sorting the cards. That processor has a single sheet of paper serving as the boundary of the workspace – the RAM. Group B plays the role of two processors (2 students on opposite sides of the room) each with their own separate RAM (1 sheet of paper each), communicating via "message passing" (the third student, relaying messages and cards back and forth). Group C plays the role of two processors (2 students next to each other) sharing RAM (a sheet of paper) and communicating via "shared memory." The remaining students in the class are asked to observe and draw conclusions as the three groups work to sort the cards. Typical observations include that group A had no overhead, but had to work alone. Group B got to work together, but was slowed down some by passing messages back and forth. Group C also got to work together, but there was the potential to "mess up" each others' work – a foreshadowing of race conditions.

A	B
	Get the value of `x`
Get the value of `x`	
	Compute `x+1`
Compute `x+1`	
	Store the result in `x`
Store the result in `x`	

Figure 1. An interleaving example in which the two processes conflict.

A	B
Get the value of `x`	
Compute `x+1`	
Store the result in `x`	
	Get the value of `x`
	Compute `x+1`
	Store the result in `x`

Figure 2. An interleaving example in which the two processes both complete their operations successfully.

D. Race Conditions

After that first sighting of potential race conditions in shared memory situations, the students are ready to consider a number of examples of parallel code with problematic interleaving in time. In order to focus on the key concepts rather than technical details, a single increment `x = x + 1;` is considered. For the purposes of the course, this is considered as a three-step process:

1) Get the value of `x`
2) Compute `x+1`
3) Store the result in `x`

If two processes A and B are executing this increment code simultaneously, there are many possibilities for the interleaving of the code in time, such as those shown in Figures 1 and 2.

E. Locks

One solution to this kind of problem is to make the `x = x + 1;` an atomic critical section through the use of a lock. Surprisingly, an effective illustration of a lock can be found in the novel Lord of the Flies by William Golding (or the movie adaptations). In the novel, a group of boys are shipwrecked on an island with no adult survivors. With little hope of rescue, they are forced to develop whatever kind of society they can manage as they struggle to stay alive. The boys have occasional meetings in which they discuss matters of mutual interest. As one might imagine, the meetings are not inherently orderly. And so, the boys devise a rule: the only person allowed to talk at the meeting is the one holding a specially designated conch shell. When that boy finishes speaking, he releases the shell, at which time any other boy

A	B
Acquire lock `L1`	
Get the value of `x`	
	Acquire lock `L1` – **BLOCK**
Compute `x+1`	
Store the result in `x`	
Release lock `L1`	
	UNBLOCK
	Acquire lock `L1`
	Get the value of `x`
	Compute `x+1`
	Store the result in `x`
	Release lock `L1`

Figure 3. An example interleaving in which one process blocks waiting to acquire a lock.

A	B
Acquire lock `L1`	Acquire lock `L2`
Acquire lock `L2`	Acquire lock `L1`
print ”A!”	print ”B!”
Release lock `L2`	Release lock `L1`
Release lock `L1`	Release lock `L2`

Figure 4. An example in which deadlock could potentially occur.

can attempt to pick it up and thus gain the right to speak. In this way (at least, ideally), no two boys attempt to speak at the same time.

The conch in this story is a lock. It's a single shared resource that only one boy (process / thread) may hold at a time. When the speaking boy releases the conch, all waiting boys attempt to acquire it. One actually succeeds, and is able to proceed with his desired comments. He then releases the conch and another can take a turn. In the same way, a lock can enable only one process to increment `x` (or more generally, enter a critical section) at a time. Other processes must wait until they can acquire the lock themselves to be able to continue.

This illustration from Lord of the Flies takes a potentially confusing concept and puts it in a more comprehensible context. Students are then ready to consider an example like that in Figure 3. Furthermore, a first illustration of potential deadlock can be explored, as in Figure 4.

F. Join

Students sometimes have trouble with the `join` operation. In particular, if thread `A` calls `B.join()`, then it is thread `A` that will block until thread `B` completes, not the other way around. A particular short story can serve as a helpful reminder to students of how this works. Suppose an adult is taking a walk in the park with a young child on a pleasant spring day. The child decides to go pick some wild flowers beside the path. For a short time, the adult can continue walking forward on the path, but there comes

```java
public abstract class CS1Thread implements Runnable {
    private Thread t;

    public void start() {
        t = new Thread(this);
        t.start();
    }

    public void run() {
        System.err.println(”You forgot to define a public void run() method in your thread class – the class that extends CS1Thread.”);
    }

    public void join() {
        try { t.join(); }
        catch (NullPointerException e) {
            System.err.println(”Must start a thread before calling join.”);
        }
        catch (InterruptedException e) {
            System.err.println(”join interrupted.”);
        }
    }

    public void sleep(int ms) {
        try { Thread.sleep(ms); }
        catch (InterruptedException e) {
            System.err.println(”sleep interrupted.”);
        }
    }
}
```

Figure 5. The CS1Thread class

a point where the adult must wait for the child to finish picking flowers and catch up. So the adult says ”Come now, please *join* me up here.” The adult waits until the child finishes picking flowers and joins the adult on the path again. This is analogous, of course, to thread `A` (the adult) calling `B.join()` (on the child). Thread `A` blocks until `B` is finished. Only then can `A` continue with its work.

G. Java Implementation

The final material considered was a preview of the implementation of these concepts in Java. In the time spent, only a preview with a couple brief exercises was possible; more time would be necessary to develop a deeper understanding. In order to free students from some of the complexities of multi-threading in Java, a helper class was created, called CS1Thread. This small class contains some important simplifications for the students, and so it is provided in Figure 5 in its entirety.

Given this class, it is a relatively simple matter to create child thread and parent thread classes. A child thread class must extend `CS1Thread` and implement the run method. A parent thread class must instantiate the child thread objects, call `start` on each, and handle any results. In this framework, a child thread can pass results back to the parent by calling some kind of a `receiveMessage` method defined in the parent class, and ensuring that each child has access to the parent object.

Note a few features of the `CS1Thread` class. Students need not be explicitly aware of Java's `Thread` class and `Runnable` interface, nor of interfaces and inheritance in general. The default implementation of `run` helpfully reminds students that they must implement this method in a child thread class. `join` and `sleep` wrappers are defined to shield students from exception handling. These hidden topics are all worthwhile, of course, but they are not essential in order to use the `CS1Thread` class to gain some hands-on practice in parallelism in Java. As a result, the instructor is free to *choose* whether or not class time is best spent on these topics or on something else.

There are many opportunities for using the `CS1Thread` class, limited only by the amount of time available in CS1. In section A of CS1, the class was used to examine several iterations of an embarrassingly parallel strategy for adding a large array of numbers. The series of examples started with two child threads on a small array, then added timing information and a large array, then using `n` threads instead of just two, and finally using the `synchronized` keyword to prevent a race condition in updating the parent thread's total sum variable.

V. The Pre/Post Test

Both sections of CS1 under consideration here used the same pre- and post-test to measure student opinions and understanding about parallelism. The pre-test was given at the beginning of the semester, before any course material at all had been considered. The post-test was given in the last week of the semester, some time after all planned material on parallelism had been considered. The tests began with four "opinion" prompts, hereafter referred to as Opinion 1 through Opinion 4. Each was answered on a 5-point scale: 1: Strongly Disagree, 2: Disagree, 3: Neutral, 4: Agree, and 5: Strongly Agree. The prompts are as follows:

1) I think the idea of parallel computation is interesting.
2) If a friend asked me what parallel computation is, I could give a 2-3 sentence explanation.
3) I'd like to learn more about parallel computation.
4) I intend to take the computer science course that follows this one, called "Data Structures".

Note that the last opinion prompt is not specifically about parallelism. Rather, it is used as a more general measure of overall student interest and motivation for the course material.

Following the four opinion prompts are eight factual questions, hereafter referred to as Fact 5 through Fact 12. Each question has five possible answers provided, labeled (a) through (e), with only one being correct. The purpose of these factual questions is to focus on key concepts of parallelism while not requiring knowledge of specific technologies. The questions follow, with answer choices omitted for brevity:

5) Suppose a program can accomplish some task by making effective use of either one processor or two. Which of the following should typically be expected? (Choices are about comparing execution speed between the 1- and 2-processor versions.)
6) Let the variable x be set to 5. Suppose the following two statements are then executed in parallel:
   ```
   set x to (x times 2)
   set x to (x plus 2)
   ```
   What is the resulting value of x?
7) Suppose two processes X and Y are executing in parallel. They run the code seen below, in which they attempt to acquire two unique locks A and B, do some kind of calculation, and then release the locks.

X	**Y**
acquire lock A	acquire lock B
acquire lock B	acquire lock A
calculation...	calculation...
release lock B	release lock A
release lock A	release lock B

   Which of the following could **NOT** happen? (Choices are about execution order.)
8) What does Moore's Law state?
9) What is the main reason that multi-core processors are more mainstream now than in the twentieth century?
10) Suppose two processes share a variable x, which currently has a value 5. They then both execute the following statement at the same time: `Increase x by 1`. What is the resulting value of x?
11) In programming, a *join* means that...
12) Which of the following is **NOT** a standard way for processes to provide data to each other?

Most students in the course had no prior programming experience, let alone experience in parallelism. Therefore it was expected that scores on the pre-test factual questions would be quite low.

VI. Test Results

The results for the opinion questions are shown in Figures 6 through 9. A single vertical bar is divided into the number of Strongly Agree (SA) responses on top, followed by Agree (A), Neutral (N), Disagree (D), and Strongly Disagree (SD) on the bottom. Results are shown for the pre-test for sections A and B, and post-test for sections A and B. Recall that section A spent over four hours of contact time on parallelism, including many active exercises and some implementation in Java. Section B spent about 90 minutes of contact time with fewer active exercises and no Java implementation.

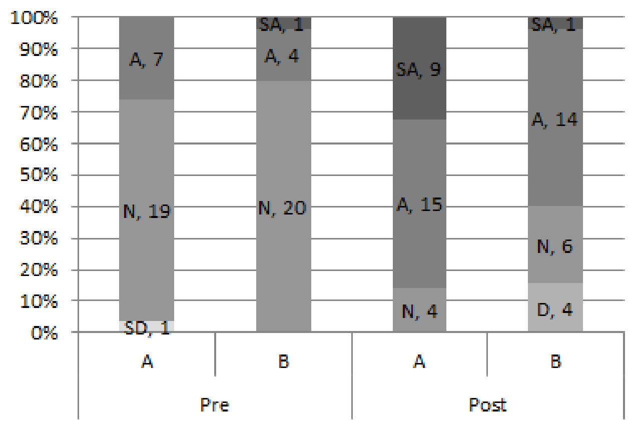

Figure 6. Opinion 1: Parallel computation is interesting.

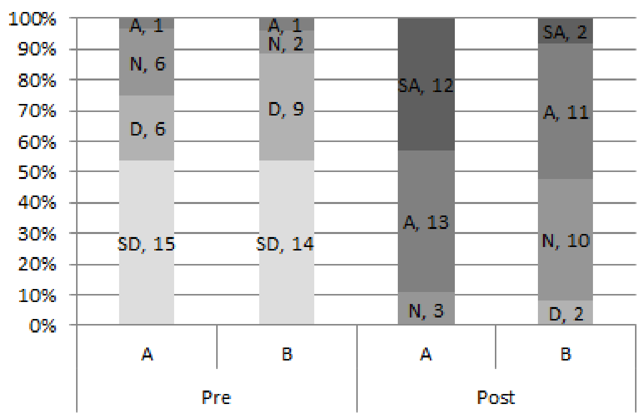

Figure 7. Opinion 2: Could give 2-3 sentence explanation.

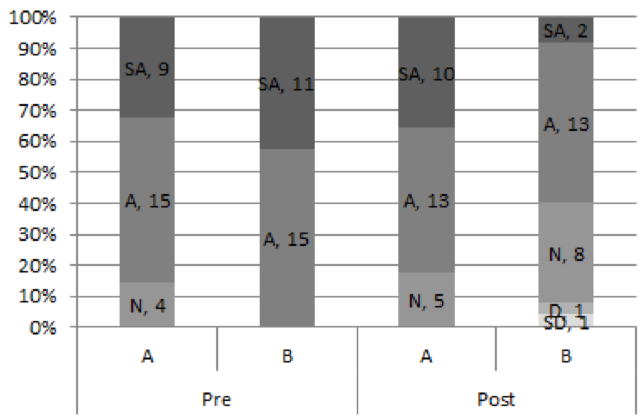

Figure 8. Opinion 3: Would like to learn more.

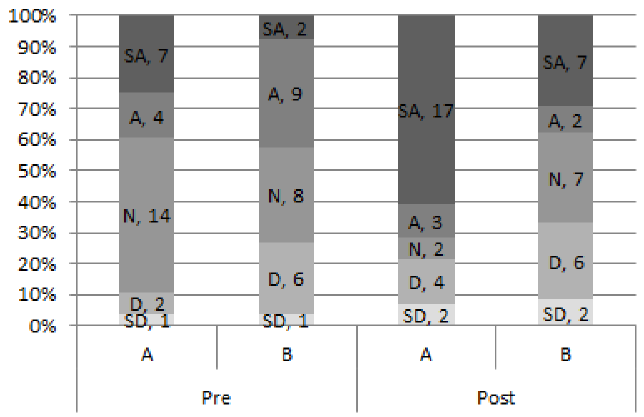

Figure 9. Opinion 4: Intend to take data structures.

Opinion 1 stated ”I think the idea of parallel computation is interesting.” Pre-test results for both sections are very comparable, with most students having a neutral opinion. Post-test results for section A show a strong increase in interest for the entire class. Post-test results for section B show a more moderate increase in interest for most students, and apparently a decrease in interest for a few.

Opinion 2 stated ”If a friend asked me what parallel computation is, I could give a 2-3 sentence explanation.” As expected, pre-test results show most students disagreeing with this statement. Post test results show widespread agreement for section A. Section B has majority agreement as well, but also with a much larger number of neutral responses.

Opinion 3 stated ”I'd like to learn more about parallel computation.” In the pre-test, students in both sections agreed, with slightly stronger agreement in section B. In the post-test, most students in section A were still interested in further study, many strongly. Section B students were also interested in further study overall, though not as strongly.

Opinion 4 stated ”I intend to take the computer science course that follows this one, called ’Data Structures.’” For the pre-test, among ”agree” and ”strongly agree” responses, sections A and B had the same total, though section A had a bit more ”strongly agree” responses. Section A had more neutral responses and fewer disagree responses than section B. For the post-test, section A interest increased significantly, though a few additional students moved to a ”disagree” response as well. Section B interest also increased, though not as strongly, with fairly comparable ”disagree” response counts.

The results of the factual questions are shown in Figure 10, for questions fact 5 through fact 12. Within the graph area for one question, bars are shown for section A's pre-test performance, then section B's pre-test performance, then section A's post-test, and finally section B's post-test. As expected, pre-test performance is quite low for both sections. In section A, post-test performance is very good overall, with typically strong majorities of the students answering a given question correctly. The one exception to this is question 6, which intentionally asked an extension question, considerably more challenging than any that had

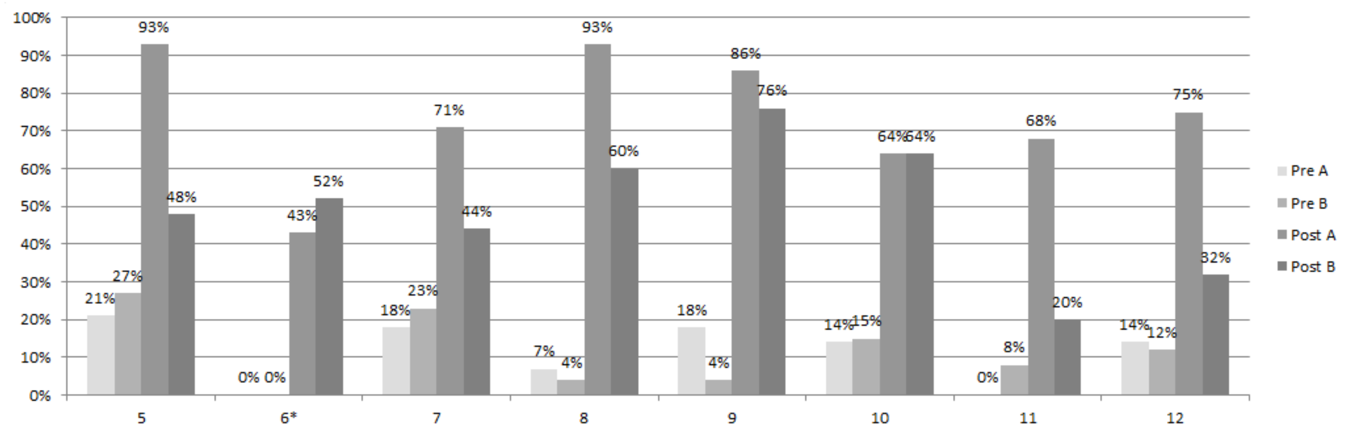

Figure 10. Results for factual questions. *The pre-test had an error on fact 6 such that there was no correct answer.

been considered in class. (Please also note that the pre-test version of question 6 had an error, such that there was no correct answer choice.) Section B performance did improve from pre-test to post-test as well, though not as strongly, usually performing well below section A.

VII. Analysis

The test results suggest conclusions both in terms of depth of learning and interest. Students in both sections appear to have learned about parallelism to some extent, as seen in Opinion 2 and the factual questions. Students in the two sections performed similarly on some of the factual questions. Specifically, questions 6, 9, and 10 show fairly comparable results, reflecting some similarities in coverage between the two sections. However, section A's learning results are much stronger overall. This is in some sense not surprising, because significantly more contact time was spent on the material (over four hours vs. 90 minutes). It is interesting to note, however, that while time spent varied, the general collection of material covered was the same. That is, both sections covered all the material on the test, except for some aspects of question 6. The difference was that while section B considered these concepts primarily in a lecture format (choosing to use the time saved on other, non-parallel topics), section A spent more time exploring the concepts with analogies and hands-on activities. The results suggest that the approach of section A, while more time-intensive, is necessary to enable students to learn these concepts more effectively as introductory students. To put it another way, if sufficient time is not available to cover deeply a given set of parallel concepts, it may be preferable to reduce the size of the set, rather than cover the entire set at a more shallow level.

The student interest questions (Opinion 1, 3, and 4) can give further insight into the experiences of the students. The two sections appear fairly comparable by these measures on the pre-test. It is also important to note that student course evaluations and student grades were similar in the two sections. That is, differences in opinion results are unlikely to be attributable to differences in overall course grades or student perceptions of the instructors.

This does then draw further attention to the difference in post-test opinion questions. On the post-test, interest in parallel computation, as measured in Opinion 1 and 3, is significantly higher in section A than section B. It is interesting to note that for section B, while interest did increase according to Opinion 1, desire to learn more decreased in Opinion 3. In section A, however, Opinion 1 interest increased more significantly, and Opinion 3 desire held steady. This suggests that a careful, deep examination of these topics increases student interest more than a quicker overview. Similar to the conclusions in the factual questions discussion above, this suggests that deep consideration of a few topics is preferable to a more shallow consideration of many topics, when faced with limited time.

VIII. Conclusion

At the introductory level, the use of analogies and hands-on activities enables students to learn about parallelism very effectively, while also stimulating greater interest in the subject. In contrast, a less time-intensive and more lecture-oriented approach, while having some success, did not produce as much learning or interest. As more educators work to integrate concepts of parallelism into introductory courses, care must be taken to explore any chosen topics with sufficient depth and practice such that students can master the material. If students fail to master a given topic, then not only will their understanding be less, but they may actually suffer from reduced interest in learning more. So in the limited time available in CS1, covering a few parallelism topics deeply appears to be preferable to covering more topics in less time.

Acknowledgment

I offer a sincere thank you to Dr. Gloria Childress Townsend for her invaluable assistance throughout the semester on all aspects of CS1.

References

[1] ACM/IEEE-CS Joint Task Force on Computing Curricula, "Computer science curricula 2013," ACM Press and IEEE Computer Society Press, Tech. Rep., December 2013. [Online]. Available: http://dx.doi.org/10.1145/2534860

[2] S. K. Prasad, A. Y. Chtchelkanova, S. K. Das, F. Dehne, M. G. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. LeBlanc *et al.*, "NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing: core topics for undergraduates." in *SIGCSE*, vol. 11, 2011, pp. 617–618.

[3] B. S. Bloom, M. Engelhart, E. J. Furst, W. H. Hill, and D. R. Krathwohl, "Taxonomy of educational objectives: Handbook i: Cognitive domain," *New York: David McKay*, vol. 19, p. 56, 1956.

[4] S. Bogaerts, K. Burke, B. Shelburne, and E. Stahlberg, "Concurrency and parallelism as a medium for computer science concepts," in *Curricula for Concurrency and Parallelism workshop at Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)*, Reno, NV, USA, October 2010.

[5] S. Bogaerts, K. Burke, and E. Stahlberg, "Integrating parallel and distributed computing into undergraduate courses at all levels," in *First NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-11), Anchorage, AK*, 2011.

[6] R. Brown, E. Shoop, J. Adams, C. Clifton, M. Gardner, M. Haupt, and P. Hinsbeeck, "Strategies for preparing computer science students for the multicore world," in *Proceedings of the 2010 ITiCSE Working Group Reports*, ser. ITiCSE-WGR '10. New York, NY, USA: ACM, 2010, pp. 97–115. [Online]. Available: http://doi.acm.org/10.1145/1971681.1971689

[7] T. J. McGuire, "Introducing multi-core programming into the lower-level curriculum: An incremental approach," *J. Comput. Sci. Coll.*, vol. 25, no. 3, pp. 118–119, Jan. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1629116.1629137

[8] K. B. Bruce, A. Danyluk, and T. Murtagh, "Introducing concurrency in cs 1," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '10. New York, NY, USA: ACM, 2010, pp. 224–228. [Online]. Available: http://doi.acm.org/10.1145/1734263.1734341

[9] R. A. Brown, "Hadoop at home: Large-scale computing at a small college," *SIGCSE Bull.*, vol. 41, no. 1, pp. 106–110, Mar. 2009. [Online]. Available: http://doi.acm.org/10.1145/1539024.1508904

[10] H. Neeman, L. Lee, J. Mullen, and G. Newman, "Analogies for teaching parallel computing to inexperienced programmers," *SIGCSE Bull.*, vol. 38, no. 4, pp. 64–67, Jun. 2006. [Online]. Available: http://doi.acm.org/10.1145/1189136.1189172

[11] A. T. Kitchen, N. C. Schaller, and P. T. Tymann, "Game playing as a technique for teaching parallel computing concepts," *SIGCSE Bull.*, vol. 24, no. 3, pp. 35–38, Sep. 1992. [Online]. Available: http://doi.acm.org/10.1145/142040.142064

[12] B. R. Maxim, G. Bachelis, D. James, and Q. Stout, "Introducing parallel algorithms in undergraduate computer science courses (tutorial session)," in *ACM SIGCSE Bulletin*, vol. 22, no. 1. ACM, 1990, p. 255.

[13] B. Rague, "Teaching parallel thinking to the next generation of programmers," *Journal of Education, Informatics and Cybernetics*, vol. 1, no. 1, pp. 43–48, 2009.

[14] S. Bogaerts, "Hands-on exploration of parallelism for absolute beginners with scratch," in *IPDPSW '13: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1263–1268.

[15] S. Torbert, U. Vishkin, R. Tzur, and D. J. Ellison, "Is teaching parallel algorithmic thinking to high school students possible?: One teacher's experience," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '10. New York, NY, USA: ACM, 2010, pp. 290–294. [Online]. Available: http://doi.acm.org/10.1145/1734263.1734363