Using Actors and the SALSA Programming Language for Introducing Concurrency to Computer Science II Students

Travis Desell
Department of Computer Science
University of North Dakota
Grand Forks, ND, USA
Email: tdesell@cs.und.edu

***Abstract*—This paper presents an evaluation of using the SALSA programming language in a two week module to introduce concurrent and distributed programming concepts to computer science II students at the University of North Dakota. The computer science II course was taught using Java, which allowed students to easily use SALSA for concurrent programming as it has a similar syntax and allows the use of Java objects. This introduced the actor model, and the necessary concepts of concurrency, asynchronous message passing and distributed memory that the actor model uses. To evaluate knowledge gains, a survey was given to students before and after the module, with minor positive gains being shown in student interest and knowledge. The results of the survey highlight the fact that early computer science students do have a natural understanding of many concurrent and distributed programming concepts. Further, they can make their minds up and gain confidence much easier than they gain actual knowledge. The results also show that students learned concepts better by applying them in programming assignments than by being presented them in lecture. This work provides motivation for longer, applied learning modules on concurrent and distributed programming in future early programming courses, like computer science I, II and data structures and algorithms.**

***Keywords*-parallel and distributed computing education; actor model; computer science II**

I. Motivation

The object and threads programming model used by Java and many other programming languages may present some difficulties for beginning computer science students, namely how to appropriately deal with deadlocks and how to synchronize data access (to prevent memory errors). Further, programs written using objects and threads can be difficult to debug, as it is challenging to determine which threads are causing these problems as there is no representation of this in the source code. These challenges are reflected in the NSF/TCPP Curriculum Initiative document [1], where concurrency defects and tools to detect them are suggested to be taught at the Data Structure and Algorithms level. Further, the object and threads model also is not particularly intuitive, as it is not representative of how things happen concurrently in the real world. Objects and threads have a module where multiple tasks are done to the same object at the same time, whereas the actor model has multiple agents which perform single tasks at a time.

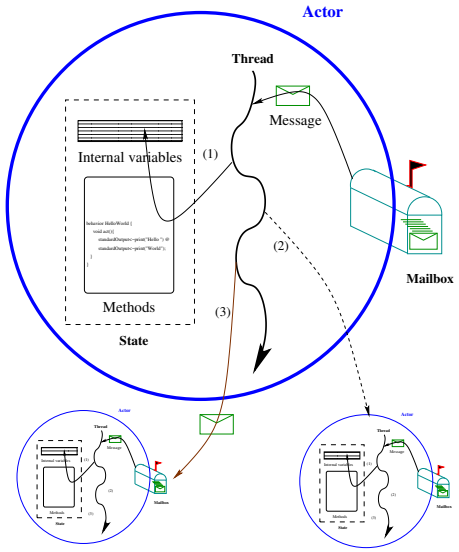

Figure 1. Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) create new actors, and/or (3) send messages to other actors (*image used with permission*) [2].

As such, *the hypothesis is that using the actor programming model, which is more "true to life", may prove more intuitive for beginning students*. This has some support in the education community as, *e.g.*, MIT's Scratch programming language used for education is also based on the actor model [3]. An actor combines a thread of control, along with its state and behavior into a single unit of concurrency [4] (see Figure 1). Each actor maintains a mailbox of messages and will process them one after the other. As the actor model uses distributed memory this prevents concurrent memory accesses, and actors communicate via asynchronous message passing which makes it very difficult to program deadlocks. This can make concurrent programming significantly simpler, as shown in Figure 2, which demonstrates a concurrent Fibonacci number calculator in the SALSA programming language.

```
 1: behavior Fibonacci {
 2:     int n;
 3:
 4:     Fibonacci(int n) {
 5:         self.n = n;
 6:     }
 7:
 8:     Fibonacci(String[] arguments) {
 9:         n = Integer.parseInt(arguments[0]);
10:
11:         self<-finish( self<-compute() );
12:     }
13:
14:     int compute() {
15:         if (n == 0) {
16:             pass 0;
17:         } else if (n <= 2) {
18:             pass 1;
19:         } else {
20:             pass   new Fibonacci(n-1)<-compute()
21:                  + new Fibonacci(n-2)<-compute();
22:         }
23:     }
24:
25:     ack finish(int value) {
26:         System.out.println(value);
27:     }
28: }
```

Figure 2. A simple concurrent Fibonacci program in SALSA. The SALSA syntax is extremely similar to Java's syntax, and it can utilize all of Java's libraries (lines 9, 26). The `new` command creates a (concurrent) actor (lines 20 and 21), and `<-` sends asynchronous messages (lines 11, 20, 21). If a message or result of a message requires the result of another message (lines 11, 20, 21) it will not be sent until the required result has been sent with the `pass` statement (lines 16, 18, 20, 21), similar to a `return` statement. The constructor taking a array of arguments serves as an actor's `main` method. This example is also a simple introduction to concepts of divide & conquer, recursion and reduction.

II. Methodology

Computer Science II has been traditionally taught in Java at the University of North Dakota, to introduce students to concepts of object oriented programming such as inheritance, polymorphism, generics and software design. Other topics of the course include, in addition to an overview of Java's libraries and syntax, an introduction to data structures such as stacks, queues and trees, as well as sorting algorithms such as quicksort and mergesort, which also introduce the students to recursion, and their analysis using big O notation.

Computer Science II is taught with a strong focus on applied learning by using extensive in-class programming assignments to reinforce the information taught in the lectures. One class a week is lecture style, while the remaining two classes are in the lab, along with an additional 2 hour evening programming lab. Because of this, students typically complete 2-3 programming assignments a week based on applying the information taught in the lecture. In general, while students have found the amount of programming to be challenging, they prefer the active learning and recognize the necessity of becoming stronger programmers to succeed in the computer science field.

In this course structure for Computer Science II, the goal was to develop a course module (over a period of two weeks) to begin to introduce concurrent programming topics using the SALSA programming language. A survey was given to students pre- and post- module to measure knowledge gains, changes in student confidence as well as student interest in concurrent and distributed programming.

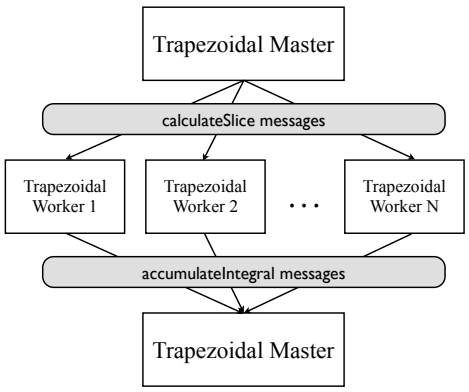

Figure 3. The trapezoidal master creates N trapezoidal worker actors, sends messages to them to calculate their slice of the integral (which are done in parallel), then combines the results when the workers are done.

III. Teaching Concurrency Using SALSA

With the structure of Computer Science II, there was time for one hour lecture and 4 lab hours each week. As opposed to most other weeks in the course, where students had 2-3 programming assignments to complete in the lab sessions, students were given a single programming assignment due to the fact that the concepts involved were more advanced and that they were using a new programming language.

The first week's lecture focused on introducing the syntactic differences between SALSA and Java, as well as the concepts of concurrency, asynchronous message passing and distributed memory. Students were given skeleton code for a parallel trapezoidal rule solver and asked to implement a version which could run any number of worker actors which could calculate slices of the trapezoidal rule, whose results would be summed up by the master actor (see Figure 3).

The second week focused on introducing non-determinism and concurrency defects. In actor programming languages, most concurrency defects occur due to the fact that when messages are sent asynchronously, the order they are received in is not necessarily the order they were sent in, which is a direct example of non-determinism. Students were given a faulty dining philosophers SALSA program, which resulted in deadlock, and were asked to fix the program so that no philosopher starved.

On average, students performed well on the programming assignments, and in general, students responded positively to using the SALSA programming language. Of particular note, the lack of static methods and fields (SALSA uses a non-static main constructor, as opposed to a static main

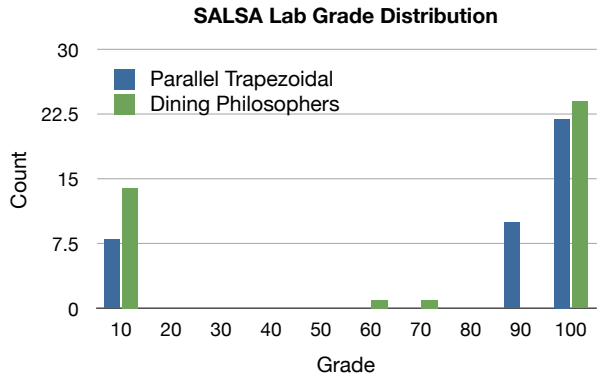

Figure 4. Histogram showing the grade distribution of the two programming assignments. Grades were very bimodal – either students successfully completed the assignments or did not do them at all.

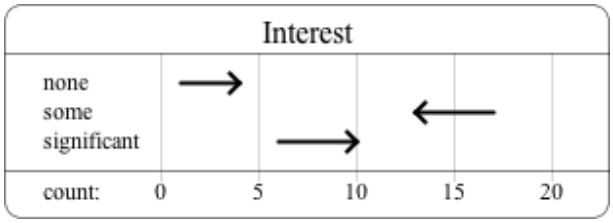

Figure 5. The change in student interest in distributed and concurrent programming, pre- and post-module.

method), which many students struggled with of the course, aided many students in quickly picking up the language.

The average grade over all students for the non-SALSA lab assignments was a 78.3, while the trapezoidal assignment had an average grade of 76.8 and the dining philosophers had an average grade of 63.2 – however for both assignments the grade distribution was very bimodal (see Figure 4), most students successfully completed the lab and received a 100, while the rest did not submit the lab and received a 0. As the dining philosophers assignment was the last of the semester, the lower grade may be due to the fact that students simply did not do the assignment due to studying for tests in other courses, rather than the fact they struggled with the material.

IV. Survey Examination

A survey was given to students before and after the module on programming in SALSA. The Computer Science II course consisted of 34 students (22 were Computer Science majors). Of these students, 24 students took the pre-survey and 27 students took the post-survey, with 21 students taking both the pre- and post-survey. The first part of the survey addressed student interest and experience in concurrent and distributed programming. Figures 5 and 6 show how student interest and experience changed during the course of the module, for all students surveyed.

Interest: Pre-survey, 1 student reported no interest, 17 reported some interest, and 6 reported significant interest.

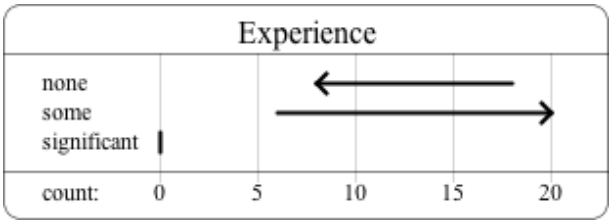

Figure 6. The change in (self-assessed) student experience, pre- and post-module.

Post-survey, 4 students reported no interest, 13 reported some interest, and 10 reported significant interest. While unfortunately the interests of all students weren't increased by the module, but more students had an increase in interest than did not (although this is most likely not statistically significant). However, the most important thing to note here is the importance of how these topics are taught to beginning computer science – *a two week module was sufficient to some students for making up their minds about further studying this subject*. Student major may have also had some effect on student interest, as 3 of the 5 students who reported no interest in the post survey were not computer science majors.

Experience: Pre-survey, 18 reported no experience, 6 reported some experience and 0 students reported significant experience, and post-survey 7 reported no experience, 20 reported some experience, and 0 reported significant experience. Again, a short two week module can have a significant effect on beginning computer science learners, as most students reported that the module gave the some experience in concurrent/distributed programming. The students were also reasonable about their self assessed knowledge, as none reported that they gained significant experience from the module.

The rest of the survey consisted of questions divided into five categories: concurrency, asynchrony vs. synchrony, determinism, distributed vs. shared memory, and concurrency defects. The categories were chosen due to their relatedness to programming in the SALSA programming language and what was taught during the two week module. Each category consisted of 3 to 4 (generally real-world) scenarios which addressed concepts for those categories. The rationale behind this approach was that many concurrent and distributed computing topics generally occur in the real world, so students may actually have some understanding of them, but not knowledge of the appropriate terminology.

For each of the scenarios given, students were asked to mark if the scenario was an example of the topic in question (for more details see the section on each topic). Students were also asked to provide their confidence in each answer (from 0 for no confidence to 5 for extremely confident), however for some reason most students only reported their confidence for scenarios that were marked. Unfortunately, this makes the confidence data of limited use. Further, this

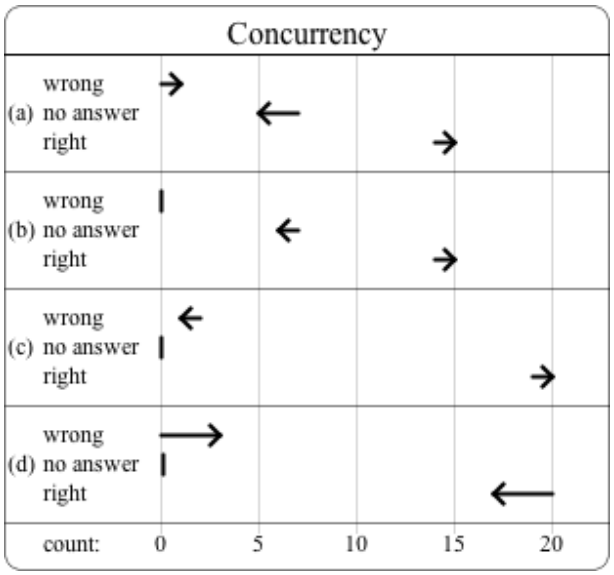

Figure 7. The change in student knowledge for the concurrency section. (a), (b), (c), and (d) correspond to the scenarios described in Section IV-1, and the arrows show the change the counts of wrong, right and no answers for students which took both the pre- and post-surveys.

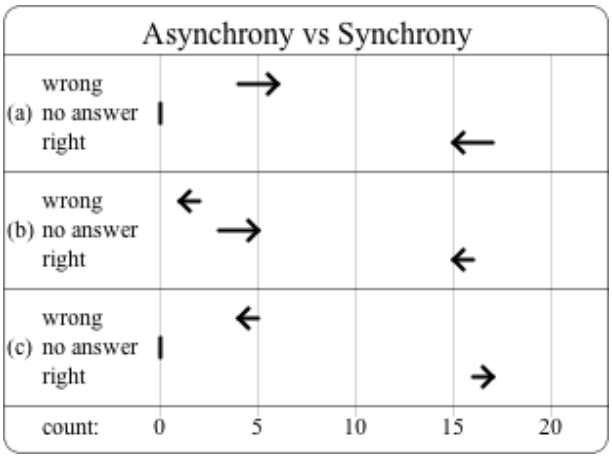

Figure 8. The change in student knowledge for the asynchrony vs. synchrony section. (a), (b), and (c) correspond to the scenarios described in Section IV-2, and the arrows show the change the counts of wrong, right and no answers for students which took both the pre- and post-surveys.

made it unclear if a student left a scenario unmarked if they actually giving an answer, or not answering at all. Because of this, data is presented for correct, wrong and no answers (as opposed to just correct and wrong). A response was scored wrong if the student checked the scenario when it was not an example of the topic. A response was scored correct if the student marked (or did not mark) the scenario correctly. A response was scored as no answer if it should have been marked and was not (and no confidence was given).

Survey results for these five categories are presented in the following sections:

1) Concurrency: Students were asked to check which of the following scenarios were examples of concurrency:

- *(a) Having each member in a group programming project work on a different section of code before combining them all at the end.*
- *(b) Having one roommate wash the dishes while the other dries the washed dishes.*
- *(c) Answering the questions in this survey sequentially, one after the other.*
- *(d) Answering the questions in any order, returning later to questions you may have skipped.*

The first two scenarios are examples of concurrency while the last two are not. Figure 7 shows how the responses of the students changed over the course of the module. For scenarios (a) and (b), average student confidence increased from 3.1 to 3.6 and 3.0 to 3.7, respectively. Not enough confidence data was gathered for scenarios (c) and (d) as students did not report confidence for unmarked scenarios. There were mild improvement for scenarios (a), (b) and (c), but it is not clear why less students were correct on scenario (d) post module. One potential reason is the fact that as SALSA uses asynchronous message passing, when messages are sent they can be received and processed in different orders than they were sent. Students may have confused this behavior of SALSA with this scenario. The results show that the students do have a general understanding of the topic, as the answers were mostly correct both pre- and post-module.

2) Asynchrony vs Synchrony: Students were asked to mark which of the following scenarios were examples of asynchronous communication (as opposed to synchronous communication):

- *(a) Invoking a method on a Java object.*
- *(b) Sending someone an email and then working on something else while waiting for the response.*
- *(c) Hitting the submit button on a webpage, which updates a database on the web server before displaying a success webpage.*

The second scenario was an example of asynchronous communication, while the others were not. Figure 8 shows how the responses of the students changed. For scenario (b), student confidence increased from 2.5 to 3.7. Interestingly, teaching a module using SALSA should have highlighted the difference between object method invocation (as done in Java) vs. asynchronous message passing (as done in SALSA), however less students answered scenario (a) correctly after the module, and more students did not provide an answer for scenario (b). So while the students who answered correctly had greatly improved confidence, for other students either the scenarios were confusing or something went awry in the course module (perhaps due to the fact that Java objects can be used within SALSA programs; albeit still by synchronous method invocation). As with the previous topic, the students generally had a good understanding of

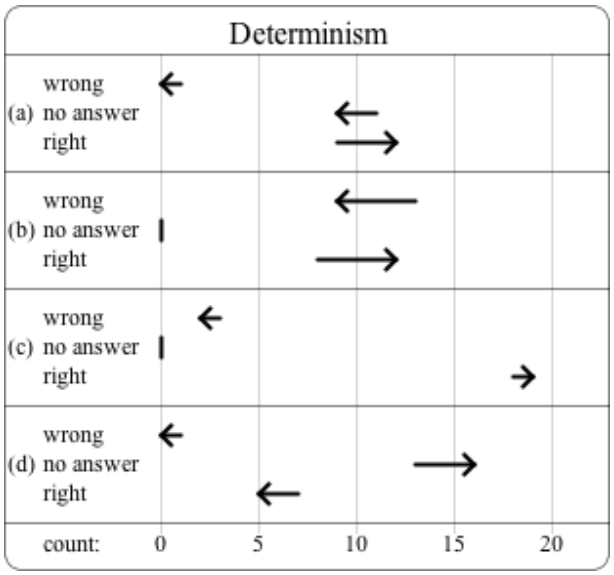

Figure 9. The change in student knowledge for the determinism section. (a), (b), (c), and (d) correspond to the scenarios described in Section IV-3, and the arrows show the change the counts of wrong, right and no answers for students which took both the pre- and post-surveys.

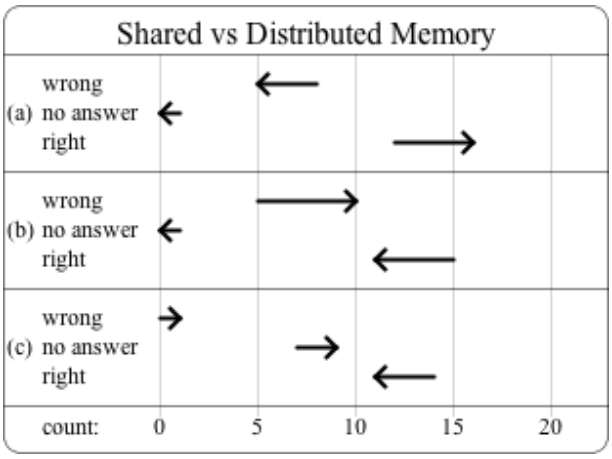

Figure 10. The change in student knowledge for the distributed vs. shared memory section. (a), (b), and (c) correspond to the scenarios described in Section IV-4, and the arrows show the change the counts of wrong, right and no answers for students which took both the pre- and post-surveys.

these topics both pre- and post-module.

3) Determinism: Students were asked to mark which of the following scenarios were deterministic (as opposed to non-deterministic):

- *(a) Making ten six-sided dice rolls and calculating their sum.*
- *(b) Having two computer processes continuously send messages between each other for 10 seconds, and calculating the total number of messages sent.*
- *(c) Performing Quicksort where the pivot is chosen randomly, using a random seed.*
- *(d) Performing Quicksort where the pivot is chosen randomly, using the same seed.*

Scenario (a) and (d) were deterministic, while the others were not. Student confidence increased from 2.0 to 3.0 for (a) and 1.8 to 2.8 for (d). However, unlike the previous topics, student confidence was lower, along with the correctness of their answers both pre- and post-module. This does make some sense as determinism is less readily apparent in real life situations as the previous topics. The module improved scores for scenarios (a), (b) and (c), but decreased answers for scenario (d). It is not readily apparent why students would have performed worse in scenario (d), other than previous to the SALSA module students were programming quicksort as part of their lab programming assignments (where random number generation seeding was taught), and may have forgotten about this during the SALSA module.

4) Distributed vs Shared Memory: Students were asked to mark which of the following scenarios used distributed memory, as opposed to those that used shared memory:

- *(a) Using software like Dropbox to save copies of your homework and class materials to a remote cloud storage service which will sync the same files on your laptop and desktop in case one breaks.*
- *(b) Running a parallel mergesort on a computer with 4 processors, all of which have access to the same memory which stores the array being sorted.*
- *(c) Two teaching assistants have a stack of papers to grade. One TA will grade the first half of the papers and the other TA will grade the other half of the papers.*

Scenario (c) is an example of distributed memory, while the others are not. Student confidence for scenario (c) decreased from 2.9 to 2.6, and increase for scenario (b) from 1.8 to 3.1. These results are particularly interesting, as scenario (c) is the only scenario where student confidence decreased. This makes the results for scenario (c) not as bad, as while more students were incorrect, at least they were less confident about it. However, more students were wrong about scenario (b) and they were significantly more confident about it which is a problem. Actors explicitly use distributed memory and students had to deal with this in their programming assignments, so it is not quite clear why the students' answers changed in this manner.

5) Concurrency Defects: Lastly, students were asked to mark which of the following were an example of a defective concurrent system:

- *(a) Three philosophers sit at a circular table with three forks. One for is in between each philosopher. Each philosopher will attempt to pick up the fork to the left of them and hold it, while trying to pick up the fork to the right of them. After a philosopher has both forks they will eat their meal and put the forks down (so the other philosophers can pick them up).*

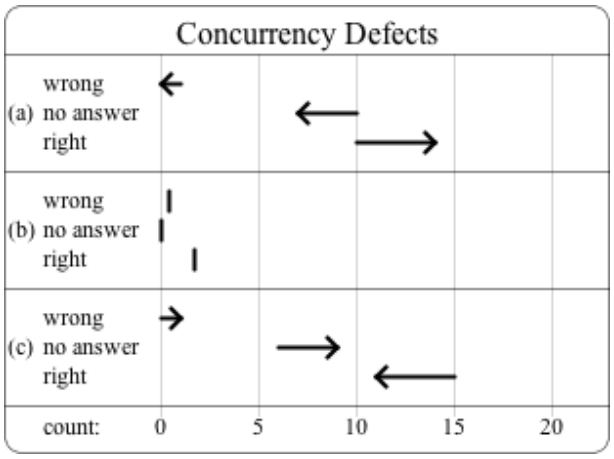

Figure 11. The change in student knowledge for the concurrency defects section. (a), (b), and (c) correspond to the scenarios described in Section IV-5, and the arrows show the change the counts of wrong, right and no answers for students which took both the pre- and post-surveys.

- *(b) A group of athletes are running in a line around a track. Every minute, the athlete in the back of the line will sprint to the front of the line.*
- *(c) Two robots are coming at each other from opposite directions down a hall. The first robot is programmed to move 1 meter to the left if something is in its way. If something is still in its way it will attempt to move 1 meter to the right. Robot 1 will repeat this left-right process until it can move forward. The second robot is programmed to move 1 meter to the right if something is in its way. If something is still in its way it will move 1 meter to the left. Robot 2 will repeat this right-left process until it can move forward.*

Both scenarios (a) and (c) are defective concurrent systems, providing examples of potential deadlock and livelock, respectively. Student confidence increased in scenario (a) from 2.0 to 3.5 and in scenario (c) from 2.0 to 3.4. Scenario (a) is the dining philosophers problem, and fixing a dining philosophers program which resulted in deadlock was the second SALSA programming assignment in the module, so the improvement in correct answers is positive result for this assignment. However, scenario (c) which was presents an example of livelock (and which was explicitly lectured on) resulted in an increase in wrong or no answers. While this is not a positive result, it does provide more evidence that having students actively work on a problem (as opposed to just being lectured on it) does a better job at promoting learning.

V. Future Work

Given that this was the author's first time performing a more rigorous evaluation of course content, there were a number of pitfalls which could have been avoided and in doing so could have provided more information about the effectiveness of the teaching methods used. First, it became apparent that in developing a survey, instructions must be very explicit (and perhaps gone over multiple times) as the students did not mark their confidence for each scenario, but rather only the ones they marked as an example of the given topic. In future surveys, this will be addressed to ensure that the most information can be gathered about the teaching methods, and also so it can be differentiated between a student giving a negative answer to a scenario and a student simply not answering the question at all. Second, more input on the scenarios presented in the survey by other teachers could result in scenarios that can more accurately access a students knowledge and learning. Some of the scenarios (in particular 3a, 4a, 5b, and 5c) could also be improved or replaced with ones with more clear cut answers.

It will also be interesting to try a similar approach in future courses, using different concurrent programming paradigms and languages, such as OpenMP [5] and Scala [6], to evaluate their effectiveness in teaching new computer science students. With a more refined survey, and investigation in multiple classes, it will be possible to gather more robust and informative data.

VI. Discussion

Perhaps the most important information to take from this work is that early programming students do have a basic understanding of concurrent and distributed programming topics, without having any formal instruction on them, as most things in the real world operate concurrently. Second, and as a warning to those seeking to introduce these topics into their curriculum, increasing student confidence in a topic is significantly easier than increasing their actual knowledge – which can prove dangerous as students can easily become more sure of their incorrect knowledge.

While the effect of using SALSA for the course module was only moderately positive, there is reason to believe that with some changes it could be a very successful method for introducing concurrent and distributed programming to early programmers. First, the module only lasted for two weeks, and while this was sufficient to increase student confidence, the level of student knowledge was not improved on a similar level. On the other hand, topics which students directly had to apply via programming assignments showed the largest improvement (as in scenario 5a).

This is positive in that it does show that students at this level can learn and understand these topics – however it should be noted that some students may still be at the level of code modification and testing, so they can simply get things to work without understanding the *why* of how they worked. However, it does mean that more time (especially applied time) must be devoted to these topics if students are going to understand them at this level. As such, future examination of this subject using SALSA and other distributed/concurrent programming languages/paradigms should involved more

applied programming, to avoid the pitfall of students gaining confidence than actual knowledge.

ACKNOWLEDGMENT

I would like to thank the students of the fall class of Computer Science II at the University of North Dakota for their patience and enthusiasm in participating in this research. This work was funded by the NSF/IEEE-TCPP Curriculum Initiative Early Adopter Program.

REFERENCES

[1] "NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates," http://www.cs.gsu.edu/ tcpp/curriculum/?q=home, accessed: 07/10/2012.

[2] C. Varela, "Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination," Ph.D. dissertation, U. of Illinois at Urbana-Champaign, 2001, http://osl.cs.uiuc.edu/Theses/varela-phd.pdf.

[3] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: programming for all," *Commun. ACM*, vol. 52, no. 11, pp. 60–67, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1592761.1592779

[4] G. A. AGHA, I. A. MASON, S. F. SMITH, and C. L. TALCOTT, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 01, pp. 1–72, 1997. [Online]. Available: http://dx.doi.org/10.1017/S095679689700261X

[5] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan Kaufmann, 2000.

[6] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2, pp. 202–220, 2009.