Programming with Concurrency: Threads, Actors, and Coroutines

Zhen Li and Eileen Kraemer
Department of Computer Science
The University of Georgia
Athens, GA USA
e-mail: janeli@uga.edu, eileen@uga.edu

***Abstract*—We describe an elective upper-division undergraduate / graduate course that focuses on programming with concurrency and puts into practice topics from the PDC curriculum. We introduce three approaches to concurrent programming: threads (using Java), Actors (using Scala) and Coroutines (using Python) for both shared memory and message passing applications. We also address synchronization issues such as race conditions, conditional synchronization, deadlock and fairness. We use a pseudocode notation to support language-independent evaluation of students' comprehension of concurrency concepts. Students engage in intensive lab sessions to implement solutions to classical problems in concurrency. We present data analyses that we hope will provide insight and inform the pedagogy associated with concurrent programming.**

Concurrency; programming; undergraduate curriculum

I. INTRODUCTION

A recent report of the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing on Core Topics for Undergraduates correctly states that "Parallel and Distributed Computing (PDC) now permeates most computing activities," impacting both programmers and users [1]. The report further emphasizes the "imperative that users be able to depend on the effectiveness, efficiency, and reliability of this technology".

The NSF/IEEE-TCPP report identifies a number of topics that could be included in advanced and/or elective curricular offerings. In this paper we describe our current efforts to further our students' understanding of PDC via a new course for upper-division undergraduates and graduate students, titled "Programming with Concurrency."

This four-hour course covers programming techniques used in building concurrent systems: the basics of multi-core architectures, and concurrency and synchronization issues in both shared memory and message passing concurrent systems using three approaches: (1) Threads in Java, (2) Actors in Scala, and (3) Coroutines in Python. Students explore the features and libraries available, and investigate the efficiency of these implementations. In this context, students assess the advantages and disadvantages, including performance and the ease of programming and debugging for each approach.

II. BACKGROUND

The pervasive growth in programming concurrent and parallel systems has led to the development of different programming languages and programming models in academia and industry. In this course, we teach the threads model using Java, the Actors model using Scala and the Coroutine model using Python.

A. Java and the Thread Model

We introduce threading at the programming level of abstraction and choose Java because of its pervasive use over a large array of devices and the fact that it is a popular introductory programming language in many CS curricula. Java synchronization syntax including the *synchronized* keyword, *wait()*, *notify()* and *notifyAll()* functions, together with the thread package provides a good set of coding schemes for the concurrency issues being covered and discussed in this course. Also, Java provides a collection of well-defined and easy-to-use concurrent data structures for advanced programming requirements.

B. Scala and the Actors Model

Scala is a recently popularized general purpose programming language that integrates features of object-oriented and functional languages. Scala programs also run on Java virtual machines and the program byte code is compatible with Java. Therefore, Scala allows usage of existing Java libraries and application packages. Scala programs may be called from Java and vice versa, with seamless integration. Accordingly, it is possible to implement concurrency and synchronization in Scala by using Java threads artifacts with the *java.lang.Thread* and *java.util.concurrent* libraries, which provide several thread definition mechanisms, inter-thread communication mechanisms and some high-level synchronized object classes. The thread synchronization and monitor models available in Java are also fully accessible in Scala.

However, Scala differs from the Java programming language in that it provides another means to implement concurrency – the Actor model. An Actor model is a mathematical theory of computation that treats "Actors" as the universal primitives of concurrent digital computation [2]. An Actor is a computational entity that, in response to a message it receives, can concurrently:

- send messages to other Actors;
- create new Actors;
- designate how to handle the next message it receives

No assumed order exists for the above actions and they may be carried out concurrently. In addition, two messages sent concurrently can arrive in either order. The Actor model enables asynchronous communication and control structures as patterns of message processing.

The Actor model illustrates a fundamental concept of the "happened before" relation, a relation among distinct events in a universe that defines the concept of time [3]. This partial relation may be extended to a full relation with an algorithm that results in a non-deterministic event sequence in a distributed system. In such a distributed system, tasks may be carried out on computational units that are either spatially separated or on a single processor. These fundamental notions characterize a concurrent system with non-deterministic ordering of task executions. Therefore, the Actor model illustrates a way of implementing concurrency.

To support the Actor model, Scala provides a set of language utilities to deal with sending, receiving, and handling messages and the creation and recognition of different Actors. Due to the relationship between Java and Scala, their similarities and differences, as well as the Actor features provided by Scala, we choose it as a contrast to Java and its thread model.

C. Python and Coroutine Model

Coroutines differ greatly from both the Java thread model and the Scala Actor model. The concept of coroutines was introduced in the early 1960s and constitutes one of the oldest proposals of a general control abstraction. The fundamental characteristics of a coroutine are introduced in [4] as follows: 1) The values of data local to a coroutine persist between successive calls to that coroutine; and 2) The execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.

In addition to this fundamental description, three further issues are identified in [5] for a coroutine: 1) the control-transfer mechanism, which can provide symmetric or asymmetric coroutines; 2) whether coroutines are provided in the language as first-class objects, which can be freely manipulated by the programmer, or as constrained constructs; and 3) whether a coroutine is a stackful construct, i.e., whether it is able to suspend its execution from within nested calls.

Based on these three issues, the authors of [5] classify coroutines into different categories and claim that a first-class stackful coroutine provides the same expressiveness as obtained with one-shot continuations, which support concurrency as stated in [6]. Therefore, a system that supports coroutines is capable of defining a concurrent system.

We choose Python not only because it is one of the most popular languages that take the advantage of coroutine concepts, but also because Python provides language features to implement concurrency with traditional thread models as well. Therefore, we expect to bring another dimension to expand students' thinking about the nature of concurrency.

III. Rationale

Current trends in multi-core and multi-processor architectures demand that students in Computer Science and Computer Engineering develop substantial practical skills in concurrent and parallel programming. However, even with recent updates to the undergraduate curriculum to include PDC concepts, Computer Science students are not systematically introduced to these concepts. Difficulties in programming such systems correctly and efficiently are seen in both academia and industry. Improved understanding of human comprehension of PDC systems and a comprehensive study of how programmers use different programming language approaches to concurrency may help to provide guidance in solving these difficulties.

This new course provides a systematic introduction to concurrent programming issues and corresponding practical programming experience in working with these systems. The course is designed to not only emphasize concepts in concurrency and concurrent systems, but also to provide hands-on programming practice and experience. We have designed the course so that data collected from integrative course activities may provide meaningful data in our ongoing study of how programmers comprehend different types of concurrent systems and the costs and benefits of different programming approaches for concurrent systems.

IV. Course Content

A. Multi-core architecture and Overview of Parallel and Concurret Programming

During the first two weeks of the course we briefly introduce students to modern computer architectures, including multi-processor and multi-core architectures. We then provide an overview of parallel and concurrent programming, introducing two basic types of concurrent systems: shared memory systems and distributed memory systems.

A primary learning objective of this portion of the course is for students to know the history of parallel and distributed computing and to comprehend the growing importance of parallel and concurrent programming given current trends in hardware development.

The lab assignment in this portion of the course involves an observation of the architecture of the student's personal

computer, in which students run two pre-compiled multi-threaded Java programs (a thread pool arithmetic program and a dining philosopher program) and are asked to report on both the nature of the dining philosophers problem and the utilization of CPU, RAM, and other resources during each of these programs.

B. UML and UML Modeling of Concurrent Systems

Next, we spend 1 to 1.5 weeks introducing UML 2.0 class, state and sequence diagrams and studying how to use these diagrams to model concurrent systems. In particular, we study the well-defined transformation from state diagrams to threads-based implementations of monitor constructs and condition variables, and a corresponding transformation to a message-passing implementation. The goal of this module is for students to gain experience in applying abstraction and modeling to the problem of reasoning about concurrent systems and in mapping from models to code.

The lab assignment here is to model a book inventory system using UML class diagrams. Later in the course students implement both shared memory and message passing solutions for this system.

Simple Statement	
variable = expression Simple statements are executed atomically. Assignment is an example of a simple statement	total = 0 name = "John Smith" condition = True height = 3.3

Figure 1. Pseudocode (Assignments)

If Statement (Conditional)	IF testScore >= 90 THEN PRINTLN "A" ELSE IF testScore >= 80 THEN PRINTLM "B" ELSE IF testScore >= 70 THEN PRINTLN "C" ELSE PRINTLN "F" ENDIF testScore = 88 **Output** B
IF *condition* THEN statement(s) ELSE IF *condition* THEN statement(s) ELSE statement(s) ENDIF The calculation of *condition* is not necessarily atomic if it involves function call statements. However, the choice of branch based on a calculated *condition* value is executed atomically.	

Figure 2. Pseudocode (Contional Statement If)

C. Comprehension and Pseudo Code Modeling

In the next 3-4 weeks of the course we introduce concurrency issues including race conditions, conditional synchronization, deadlock, and fairness with both shared memory and message passing approaches. In prior work, Tew [7] developed and validated pseudocode that supports language-independent measurements of CS1 knowledge. We have extended this pseudocode to incorporate elements related to the design and modeling of both shared memory and message passing approaches. A selected subset of this pseudocode can be seen in the figures. Figure 1 shows the pseudocode notation associated with assignment statements and Figure 2 shows the pseudocode notation associated with conditional statements. In Figure 3 we provide an example of the pseudocode we have devised for representing concurrent execution. Pseudocode designed to represent constructs in shared memory approaches are seen in Figure 4 and pseudocode designed to represent constructs in message passing approaches is seen in Figure 5.

Parallel Execution Statements	
PARA *statement(s)* ENDPARA Statements within the PARA/ENDPARA block are executed concurrently. Atomic statements within PARA/ENDPARA are executed in any order.	PARA PRINT "hello " PRINT "world " ENDPARA **Output** possibility 1: hello world possibility 2: world hello
Statements defined in a function that is called within the PARA/ENDPARA block are executed sequentially.	DEFINE print() PRINT "hi" PRINT "there" ENDDEF PARA print() ENDPARA **Output** hi there
Statements defined in functions that are called within a PARA/ENDPARA block are executed in any order of interleaving with simple statements within the same PARA/ENDPARA block. Statements defined in two functions that are called within the same PARA/ENDPARA block are executed in any order of interleaving while statements from any one of the functions are executed in their order of definition.	DEFINE print() PRINT "hi" PRINT "there" ENDDEF PARA print() PRINT "world" ENDPARA **Output** possibility 1: world hi there possibility 2: hi world there possibility 3: hi there world

Figure 3. Pseudocode (Concurrent Execution)

Use of this pseudocode allows us to evaluate student comprehension of concurrency concepts in a language-independent manner. While Tew's pseudocode has been validated for language-independent measurement of CS1 knowledge, our extensions and their use for purposes of evaluating understanding of concurrency concepts are exploratory.

Shared Memory Concurrency	
Exclusively Accessed Statement EXC_ACC *statement(s)* END_EXC_ACC Only appears within a function definition. When one function call executes statements inside an EXC_ACC/END_EXC_ACC block, other function calls that read or modify the same variables that appear inside the markers may not execute until the first function call completes or executes a WAIT function.	x = 10 DEFINE changeX(diff) EXC_ACC x = x + diff END_EXC_ACC ENDDEF PARA changeX(1) changeX(-2) ENDPARA PRINTLN x **Output** 9
Wait and Notify Functions WAIT() NOTIFY() Only be called inside a EXC_ACC/END_EXC_ACC block. Once a WAIT() function starts execution, another function call that reads or modifies variables inside the EXC ACC/END EXC ACC block may execute. Once a NOTIFY() function is executed, all WAIT() functions finish their execution. Both WAIT() and NOTIFY() functions are atomic.	x = 10 DEFINE changeX(diff) EXC_ACC WHILE x + diff < 0 DO WAIT() ENDWHILE x = x + diff NOTIFY() END_EXC_ACC ENDDEF PARA changeX(-11) changeX(1) ENDPARA PRINTLN x **Output** 0

Figure 4. Pseudocode (Shared Memory)

The pedagogical objective of this portion of the course is for students to know the two types of concurrent programming systems (shared memory vs. message passing), to comprehend the related concurrency issues (race conditions, conditional synchronization, deadlock and fairness), and to comprehend and apply the corresponding solutions to these issues (lock mechanisms vs. private data, wait and notify vs. message protocol design, and asymmetric design in concurrent systems). Another pedagogical objective is to familiarize students with the pseudocode notation so that they can use this notation to comprehend and reason about various concurrency problems and scenarios.

Students complete several in-class quizzes to practice using the pseudocode notation to create or enhance models of different concurrent scenarios such as a sum & workers system, a bounded buffer system, a dining philosophers system and a readers-writers system. Students also model a book inventory system with pseudocode and use sequence diagrams to depict and reason about some critical scenarios of the system with their model. In a homework assignment, students search for and study different concurrency-related bugs (mainly through the open source MySQL bug report database). The goal of this assignment is to promote students' understanding of concurrency concepts via these practical examples.

Message Passing Concurrency	
Message Variable MESSAGE.*message-name*(*value*...) A special message variable that carries a collection of values. The *message-name* is used to distinguish message variables from one another.	m1 = MESSAGE.h("hello") m2 = MESSAGE.w("world")
Send Statement Send(*message variable*).To(*object*) Send a message specified by message variable to a receiver object. A send statement is asynchronous, which means that the order in which messages are received may differ from the order in which they were sent.	m1 = MESSAGE.h("hello") m2 = MESSAGE.w("world") Send(m1).To(r1) Send(m2).To(r1)
Receive Statement ON RECEIVING *message* *statement(s)* *message* *statement(s)* ... Accept the next message and execute statement(s) according to the type of the message.	CLASS Receiver DEFINE receive ON_RECEIVING MESSAGE.h(var) PRINT var MESSAGE.w(var) PRINTLN var ENDDEF ENDCLASS m1 = MESSAGE.h("hello") m2 = MESSAGE.w("world") r1 = new Receiver() r1.receive() Send(m1).To(r1) Send(m2).To(r1) **Output** possibility1: hello world possibility2: world hello

Figure 5. Pseudocode (Message Passing)

D. Implementation of Concurrent Systems

This portion of the course takes about 8-10 weeks and has three major phases. First, we introduce students to general knowledge about the Java, Scala and Python programming languages. Students at UGA are already familiar with Java, but Scala is new to most students and Python is new to many. We then focus on the threading elements of Java, the Actors elements of Scala, and the Coroutine elements of Python. Finally, we look at some of the advanced concurrency programming elements in each of these languages. During this portion of the course we employ a "flipped classroom" approach, meaning that students learn about programming in these languages by reading and making use of online resources while at home and then engage in actual coding in the classroom.

Students then complete labs that employ basic Java, Scala and Python programming elements to become familiar with these three languages. Next, students implement the party-matching and sleeping barber problem with Java threads, Scala Actors and Python Coroutines during in-class lab projects. Finally, students implement the book inventory system as both a shared memory system and a message passing system.

The learning objectives of this portion of the course are for students to know, comprehend, and apply knowledge of these programming languages and their concurrency constructs to implement solutions to concurrent problems.

E. Research in Human Factors and Software Engineering of Concurrent Systems

This element of the course is conducted in parallel with the implementation components. The pedagogical objective of this portion is to make students aware of the difficulties inherent in programming concurrent software, the historical and practical concerns of designing development environments for these programming activities and the human factors issues involved. Paper presentations and in-class discussions are the means by which the objective is achieve. Students choose a paper that addresses concurrent or parallel software engineering issues or human factors in programming and present it to the class. Each student reads every paper and participates in the discussion of all presented papers.

V. Study Design

In the context of teaching this course, we observe and collect data on characteristics of students' comprehension of shared memory and message passing concurrent systems, the impact of learning each approach on students' comprehension of concurrency concepts, and the ease or difficulty with which students are able to apply these approaches (design, implement, and debug) to the solution of classical problems in concurrency.

As described in sections III.A, III.B and III.C all subjects receive, as a group, the same instruction on 1) general knowledge about multi-core architectures and parallel and concurrent programming; 2) UML and UML modeling of concurrent systems; 2) the threading elements of Java, the Actors elements of Scala, and the Coroutine elements of Python; and 3) Comprehension of shared memory and message passing systems and pseudocode modeling.

For purposes of Test 1, subjects are separated into two groups, a shared memory group (S) and a distributed memory group (D), such that the groups have equivalent performance on previous homeworks, labs and quizzes. The test contains two sections of questions, and both groups take both sections of the exam. However, to account for any practice/learning effect that may result from answering similar questions on the two sections we had group S take the shared memory section first and group D take the message passing section first.

In this test, we expect to learn about and compare student difficulties in comprehending the same problem in two forms (shared memory vs. message passing), regardless of actual programming models.

The program used in the test is the single-lane bridge problem in which cars travel in two directions using the same single-lane bridge. In the test, we describe the problem using both natural language (English) and the pseudocode notation described above. We give descriptions of what has already happened in the system and ask students to predict what could happen next, and to explain their reasoning. Figures 7-8 illustrate sample test questions in shared memory and message passing sections in a scenario consisting of a bridge, two red cars and a blue car.

Students' answers to these test questions are graded according to the misconceptions apparent in their explanations. In an earlier study [8], we identified and categorized concurrency-related misconceptions about shared memory systems into a hierarchy of 5 categories. In this study, we identify misconceptions about message passing system and combine this into the hierarchy of 5 categories, as depicted in Table II and discussed in Section VI.

Next, as a group, subjects receive instruction in programming with basic elements of Java, Scala and Python (without the concurrency-specific programming features). We then review solutions to Test 1 and instruct subjects to implement two concurrent programs, a party-matching problem and a sleeping-barber problem in three different forms: shared memory (with Java Threads), message passing (with Scala Actors) and cooperative (with Python Coroutines).

In the party-matching problem, boys and girls come to a party individually, but may only leave with a partner of the opposite sex. In the sleeping-barber problem, customers

come to a barber's shop with a limited waiting area, wait if all barbers are busy or are served if one of the barbers is available. The barbers keep working when customers are waiting or take a rest when no customer is in the shop.

```
PARA
  redCarA.run()
  redCarB.run()
  blueCarA.run()
END_PARA
```

Suppose **redCarA** has called the *redEnter()* method on line 9 but has not returned. Then **redCarB** invokes its *run()* method and calls the *redEnter()* method but also has not returned.

Decide if each of the scenarios below (k-t) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(m)**redCarB** returns from the *redEnter()* method, then calls the *redExit()* method on line 19 and blocks on the `EXC_ACC` marker on line 20.

YES NO

Explanation:

Figure 6. A Sample Question in Shared Memory Section of Test 1

```
PARA
  bridge.start()
  redCarA.start()
  redCarB.start()
  blueCarA.start()
END_PARA
```

Suppose **redCarA** has sent the *redEnter* message but has not yet received any messages. Then **redCarB** invokes its *start()* method, and sends the *redEnter* message but has not yet received any messages.

Decide if each of the scenarios below (k-t) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(m)**redCarB** receives a *succeedEnter* message, then sends a *redExit* message and receives `MESSAGE.succeedExit(2)`.

YES NO

Explanation:

Figure 7. A Sample Question in Message Passing Section of Test 1

TABLE I. CONCURRENCY-RELATED MISCONCEPTIONS IN HIERARCHY

Description Level	
D1	Misconceptions of the system and/or problem descriptions
Terminology Level	
T1	Misinterpretation of a term that describes thread or process behavior
Concurrency Level	
C1	Misconceptions about thread or process behaviors
Implementation Level	
I1	Misconceptions about synchronous mechanisms
I2	Misconceptions about asynchronous mechanisms
Uncertainty Level	
U1	Confusion about space of executions; include impossible execution sequences or fail to consider possible execution sequences

During this period of time, subjects are required to finish online reading materials at home and complete lab assignments in class. Therefore, they learn to use the three programming languages to program two different concurrent scenarios in three different forms. Next, students all attend the same Test 2, which is a computer-based practical programming test. In this test, students are required to implement the single-lane bridge problem with Java threads, Scala Actors and Python Coroutine models in shared memory, message passing and cooperative forms. This test provides information on the costs and benefits of implementing the same problem in three forms with three different approaches.

Next, we review solutions to Test 2 and separate students into two groups with equivalent performance in the prior assignments and tests: a pair programming group (PP) and a solo programming group (SP). Students in both groups then finish the same labs involving programming the book inventory system in shared memory and message passing forms. Students in the PP group work on these lab assignments with their designated pair partner and students in the SP group work on these lab assignments individually. According to our previous study[9], students in the PP group and SP group likely experience basically the same level of challenge in finishing such programming labs.

Using the data collected during this period, including survey answers and lab submissions, we hope to evaluate the benefits and costs of pair programming in programming concurrent systems. Throughout the semester we collect survey data on the time required to complete each assignment, perceived time pressure, perceived performance, and other evaluations of preferences and subjective satisfaction. These survey data, in combination with the objective data on performance, are used to empirically evaluate the costs and benefits of the different programming models and languages employed in this course.

VI. INITIAL DATA

Surveys on effort and preferences were collected with each lab and homework assignments. Students consistently reported difficulties with shared memory systems. In homeworks 2 (shared memory) and 3 (message passing), students were asked to write pseudocode for the bounded-buffer and dining-philosopher problems discussed in class. In a survey conducted after homework 3, only 1 student indicated that message-passing is more difficult, and 10 indicated that shared memory is more difficult. The remaining students either indicated that the two approaches were equally difficult, or they did not respond to the question.

In lab 2 (shared memory) and lab 3 (message passing) students were asked to design a book inventory system. In the post-lab survey, 8 of 11 students who responded indicated that shared memory is more difficult, 1 indicated

that message passing is more difficult, and 2 students found the assignments equally difficult.

TABLE II. PERFORMANCES ON TEST 1

Group	Shared Memory Section Mean	Message Passing Section Mean	Overall Mean
S (9 students)	56.67 / 100 (1st)	81.72 / 100 (2nd)	138.39 / 200
D (7 students)	76.14 / 100 (2nd)	65.93 / 100 (1st)	142.07 / 200
All	65.19 / 100	74.81 / 100	

TABLE III. MISCONCEPTIONS SHOWN IN TEST 1

Message Passing
[D1]M1: Question setting (#students: 6)
[T1]M2: Misinterpret "race condition" as "different order of messages" (#students: 1)
[C1]M3: Send semantics : assume ability to send depends on condition at receiver or interpret send as a synchronous method call (#students: 7)
[C1]M4: Receive semantics: assume receipt of acknowledgement message is synchronous with the occurrence of the event ((bridge entered or exited) (#students: 7)
[I2]M5: Conflate message sending order with receiving order (#students: 6) Four scenarios: 1) different senders, same receiver (covered by test problem) 2) different senders, different receivers 3) same sender, different receivers (covered by test problem) 4) same sender, same receiver
[U1]M6: Uncertainty (#students: 7) Increased size of state spaced causes illogical (self-contradictory) reasoning or occurrence of misconceptions not seen in simpler scenarios
Shared Memory
[D1]S1: Conflate order of cars with their thread's name (#students: 3)
[T1]S2: Misinterpret "race condition" as "different interleaving" (#students: 1)
[T1]S3: Misinterpretation on terminology "block on" (#students: 2)
[C1]S4: Conflate order of method return with order of entering/exiting bridge (#students: 4)
[C1]S5: Conflate locking with conditional waiting (#students: 9)
[I1]S6: Misinterpretation of WAIT() function's effect and conflate wait with continuous execution of the enclosing while loop (#students: 1)
[I1]S7: Conflate order of method invocation/return with get/release lock (#students: 10)
[U]S8: Uncertainty (#students: 2) Increased size of state spaced causes illogical (self-contradictory) reasoning or occurrence of misconceptions not seen in simpler scenarios

In the cases of both the homeworks and the labs, students were asked to first solve the problem for the shared-memory case and then for the message-passing case. Thus, ordering effects could explain the preference for message-passing. Therefore, for Test 1, students were assigned into two groups S and D such that the groups had equivalent performance on previous assignments and asked to complete the sections of the exam in opposite orders. In the 1st session group S took the shared-memory section of the exam and group D took the message-passing section of the exam. In the 2nd session, each group took the remaining section of the exam. The testing order is listed in Table I.

After test 1, we again surveyed students on their perceived difficulty of the two different systems. In this survey, 11 of the 15 students who responded indicated that questions in the shared memory section were harder to answer than those in the message passing section. In the same survey, students were given the opportunity (without knowing their scores) to choose which of the two sections of the exam would count as their midterm grade. (In fact, we always used the higher-scoring section to count toward their class grade). Of the respondents, 10 of the 15 chose the message passing section. Of the 5 students who chose the shared memory section, 4 took the shared memory portion in the 2nd session. Of these 15 students, 13 chose correctly, in that they selected the section in which they actually scored higher. The two students who chose incorrectly chose the shared memory section but actually scored slightly higher on the message-passing section.

Test results are listed in Table III. We found no significant difference in performance between the shared-memory and message-passing sections. However, we did find that students performed better in the 2nd session (79.20%) than in the 1st session (60.71%) (p=0.005), likely as a result of learning that occurred during the exam and/or additional studying that may have occurred between sessions. However, the students' better raw scores on the message passing section than on the shared memory section supports the survey result that students found the shared memory model more difficult to understand.

By analyzing students' explanations for each test question, we identified some frequently seen misconceptions about shared memory and message passing concurrent systems, as illustrated in Table IV.

One major misconception seen with message passing is a misunderstanding of the send function. In [C1]M3, we saw some students interpret a message send as a method call that could not happen unless the condition were satisfied at the receiver. For example, in a scenario in which redCarA successfully entered the bridge, a student indicated that redCarB could enter the bridge but could not exit because "redCarB cannot send the redExit message until redCarA sends redExit". Some students interpret a message send as a synchronous call, writing "redCarA calls redEnter first and the bridge has to process its message first before any other messages."

The next major misconception, seen in [C1]M4, is the assumption that the occurrence of an event (entering/exiting the bridge) implies that an acknowledgement message has been received. For example, one student wrote, "redCarA is not on the bridge since it has not received any message yet".

Students exhibited difficulty in fully managing the asynchronous nature of message passing systems. Table IV lists four scenarios that may actually happen in asynchronous systems, but due to the nature of single-lane bridge problem, students were only tested on scenario 1 (different senders, same receiver) and 3 (same sender, different receivers). Looking closer into students' explanations, we see that student understanding is quite unreliable – among the six students who displayed the misconception that messages are

necessarily received in the order sent, two of the six applied this only to messages from the same color cars but correctly reasoned about messages from different color cars.

The major misconception in reasoning about shared memory was a conflation of the order of method invocation/return with the order of obtaining/releasing the lock (e.g. "redCarA has not returned from the redEnter method so it must still hold the lock"), likely because most students had prior experience in Java, in which entry to a *synchronized* method may be thought to occur simultaneously with obtaining the lock and release of the lock may be thought to occur simultaneously with return from the *synchronized* method.

Also, some students showed misconceptions in differentiating lock mechanisms from wait/notify mechanisms. When the question asked whether a particular thread will be blocked on the acquisition of lock, the students explained that "the condition is not satisfied yet for the thread to get the lock" or "the first red car has not exited yet, so the second red car cannot get the lock and execute redExit function". This misconception is similar to that in which a message send is interpreted as a method call that cannot happen unless the condition is satisfied at the receiver. In both cases, the student's incorrect reasoning is based on global knowledge not actually available to the current thread or process.

Some students performed quite well on the test. However, even the most advanced students had difficulty when reasoning about a large space of possibilities. When students are not quite able to manage the execution space (usually over 3-4 possibilities), they tend to reduce the complexity by falling back into one of the lower level misconceptions, perhaps as result of increased cognitive load. At this time, they either give a correct explanation but choose an incorrect answer or conflate two concepts in a way that reduces the execution space.

VII. Conclusions

Prior to the efforts of the NSF/IEEE-TCPP, knowledge and practical skills related to parallel and distributed computing have been under-represented in CS curricula. In this paper, we describe a course that focuses on systematically introducing concepts and programming tactics with concurrency at the application level of abstraction. We introduce two types of concurrent applications, shared memory and message passing systems. We introduce various concurrency issues such as race conditions, conditional synchronization, deadlock and fairness in concurrent systems. We also introduce three typical programming models used in the programming of concurrent applications, threads, Actors and Coroutines by using the Java, Scala and Python programming languages. We employ language-independent evaluation of students' understanding of concurrency concepts to provide information for further course design. We also emphasize intensive lab and research activities to promote the development of actual programming skills and critical thinking. With careful organization and arrangement of the course, we also collect data from which we begin to gain insight to inform the pedagogy associated with concurrent programming. Through the first one-third of the course, we made following observations: 1) This class is challenging, especially to undergraduate students who have limited knowledge of concurrency and are inexperienced in programming. Some students report time pressure on completing homework and lab projects. From the feedback of students who withdrew from the course, 2 of 3 expressed unmanageable course workload as their major reason for dropping; 2) The pseudocode system is useful for students to comprehend and reason about concurrent systems, but it requires further refinements on wording and validation; 3) A standard glossary of well-defined terminology is essential; 4) Shared memory is harder for students to understand, design, write pseudocode for, and reason about.

The described course and the study are in progress during the Spring semester of 2013 at the University of Georgia. We expect to form more solid conclusions after carrying out the whole course and study plans.

VIII. References

[1] Sushil K Prasad et al., "NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing: core topics for undergraduates," , 2011, pp. 617--618.

[2] Carl Hewitt, "Actor Model of Computation," *Arxiv preprint arXiv10081459*, pp. 1-29, 2010.

[3] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558--565, jul 1978.

[4] Chris D. Marlin, *Coroutines: A Programming Methodology, a Language Design and an Implementation*.: Springer, 1980, vol. 95.

[5] Ana Lucia de Mour and Roberto Ierusalimschy, "Revisiting Coroutines," techreport 2004.

[6] R Hieb and R. K. Dybvig, "Continuations and concurrency," *SIGPLAN Not.*, vol. 25, pp. 128--136, feb 1990.

[7] A Elliott Tew, "Assessing Fundamental Introductory Computing Concept Knowledge in a Language Independent Manner," 2010.

[8] Zhen Li, Zhe Zhao, and Eileen T. Kraemer, "Characterizing Comprehension of Concurrency Concepts," in *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group*, Madrid, Spain, 2010, pp. 88-99.

[9] Zhen Li, Christopher Plaue, and Eileen T. Kraemer, "A Spirit of Camaraderie: The Impact of Pair Programming on Retention (unpublished)," University of Georgia, Athens, GA, 2012.