Teaching Modern Multithreading in CS2 with Actors

1st Mark C. Lewis
Department of Computer Science
Trinity University
San Antonio, TX, USA
mlewis@trinity.edu

2nd Lisa L. Lacher
Department of Computer Science
University of Houston-Clear Lake
Houston, TX, USA
lacher@uhcl.edu

***Abstract*—Explosive growth in multiprocessor computing and the pervasive nature of multicore processors has not only made multithreading and related topics such as parallelism, concurrency, synchronization, etc. an essential part of any undergraduate Computer Science curriculum, it has also lead to the addition of newer constructs to support multithreading in many languages. Not only is it important to motivate student interest in this topic, it is important that they are also educated in current methods used in industry. This can mean an increase in material that needs to be covered. Because of the increase in scope of a CS education, teaching topics in parallel and distributed computing in a hands-on manner is challenging, thus it is valuable for educators to explore different methods of educational delivery in order to best engage their students within the limits of curriculum timelines. The actor model is immensely popular in industry and runs some of the most important software today. In this paper, we describe how we are using Actors as a significant part of the multithreading coverage at the CS2 level, for first-year computer science majors. We also describe a semester-long project that involves the use of these concepts to help solidify student understanding and present student feedback on the project and approach.**

***Index Terms*—Computer Science Education, Parallel and Distributed Computing, multithreading, actors, futures, Scala**

I. Introduction

Virtually every computing device that people own today has at least two cores. As such, making the use of multiple threads or processes is required in order to fully utilize the potential of any modern hardware. Most CS curricula include courses that cover parallelism and concurrency in detail, but generally these occur in more advanced courses [1]–[3]. Although there is no consensus on where in the curriculum to introduce these topics, some institutions begin showing students how to parallelize their code early in CS1 or CS2 [4]. The Center for Parallel and Distributed Computing Curriculum Development and Educational Resources (CDER) also proposes integrating parallel computing "at the earliest stages of an undergraduate career." [5]. The center believes that when teaching programming, the students should begin to think of the process of applying parallization very early on in their education because it is easier to teach new programmers to think about using parallel programming concepts from the onset than it is to try to change their thinking and introduce parallel programming after sequential programming is ingrained in how the student thinks about programming. However, integrating parallel computing concepts into the early years can be difficult.

One of the standard challenges of CS instruction comes from the fact that the discipline continues to grow in its scope. The number of topics that need to be covered in a 4-year degree is daunting and it continues to grow over time [6]. Parallelism is just one of many "foundational" concepts that need to be addressed in the early courses in a CS major. There are few models and examples on how to incorporate parallel topics along with all the other necessary early computing topics [7]. Ideal approaches to teaching parallelism would give students significant practice with it without pushing other topics out of the way. This problem was well described by [8] when they said: "A typical parallel programming course focuses on using low-level libraries – MPI for message passing distributed memory systems, OpenMP and other thread tools for shared memory systems, and CUDA/OpenCL for high performance GPU computing. A course typically uses these tools to solve simple problems such as matrix multiplication and sorting. Unfortunately, this approach does not give the student programmer the skills to tackle larger problems nor skills in computational thinking for parallel applications. In addition, programmers have to deal with issues such as deadlock and mutual exclusion. A programming approach is needed that raises the level of abstraction to make parallel programming easier and also more scalable."

In this paper, we will argue for one such approach to addressing this problem.

II. Background

Trinity University has been teaching multithreading in our first-year CS2 course using various techniques since Fall 2002 when the course switched from using C++ to using Java. This course has historically introduced object-orientation, abstraction/polymorphism, an introduction to data structures, and various other topics that like multithreading, networking, and regular expressions. At the time we moved to Java, threads were introduced near the end of the course. Over the years, the coverage of threads was moved earlier and expanded to include additional elements of the Java library, such as the addition of the `java.util.concurrent` package. We found that moving it earlier in the semester gave the ability to include it in more

assignments and discuss it in a variety of contexts as the course progressed. This earlier version of our course used a semester long project that was discussed in [9].

In the 2010-2011 academic year, Trinity began teaching CS1 and CS2 using Scala as a single language for both instead of using C for CS1 and Java for CS2. While there were many pedagogical benefits to making this transition, we focus here on the options that it opened up for teaching parallelism. The first advantage is that not doing a language switch between CS1 and CS2 effectively adds 1-2 weeks in terms of what can be covered during the semester, this can enable bringing parallelism into curricula that didn't previously include it or simply provide more time to go into depth on parallelism and related topics.

While students are still shown Java threads and the underlying mechanisms that they represent, the move to Scala provided an incentive to present a more modern approach to teaching general parallelism. In particular, the course emphasizes the use of parallel collections, composable Futures, and actors as safer ways to achieve parallelism with reduced risk of race conditions and virtually no risk of deadlock. The material that is used for this instruction is covered in the textbook for the course [10].

Students enter CS2 having not covered any material on parallel programming in CS1. Instruction on multithreading in CS2 begins in the fifth week of class with a discussion of the basic concept of multithreading and the problems that are often associated with it, specifically race conditions and deadlock. We then cover parallel collections and the default Future implementation currently provided in Scala as mechanisms for adding multithreading to their programs. This is followed by coverage of the actor model using Akka [11]. We finish our coverage looking at basic Java threads and the facilities covered in `java.util.concurrent`. At each step, we look at the risks associated with each technique and when each one is appropriate to use. For example, we discuss how passing mutable messages can cause race conditions in the actor model. In addition to the discussion in class, students are assessed on these topics in short answer and paper coding questions that appear on quizzes and tests.

By using these modern mechanisms for multithreading, where students have a reduced chance of running into race conditions or deadlock, we are able to give them a significant project that is heavily multithreaded that they can complete without taking away from the other topics that are essential to cover in CS2 for our overall curriculum. Projects such as this are very valuable as projects in computer science increase student engagement and build the relationship to professional practice [12].

Note that this type of instruction is not limited to Scala. Similar curricula could be provided in other languages. Composable futures are now fairly broadly available. Java 8 introduced the `CompletableFuture` and there are similar features in C#. In addition, Akka has a Java API as well as a Scala API, though the ease of use of Futures and actors is greater with Scala thanks to the more expressive syntax. There is also a version of Akka implemented for the .NET platform [13]. Motivating students to care about Akka is fairly easy as it is used by many high-profile companies including being a significant part of the back-end for the popular video game Fortnite [14].

Our goals in sharing our experience are to convince others that Actors provide a very positive experience for students in the introduction of parallelism and concurrency and that a multi-user text game project can aid in motivating student learning. Readers can find the full course schedule, including links to project descriptions at [15].

III. The Nature of Akka Actors

The actor model of parallelism was originally developed in 1973 as an approach for doing artificial intelligence [16]. It has since been developed as a general approach to parallel computing and it was popularized by Erlang [17], [18]. The message-passing semantics make it well suited for distributed computing. When used on a single machine the implementations are multithreaded which allows them to skip the overhead of copying objects that are passed between actors as all the actors exist in the same shared memory.

The general concept of an actor is similar to object-orientation in terms of encapsulation, but instead of calling methods on actors where the current thread at the call site executes the code in the method, each actor has an inbox and messages for one actor are processed one at a time with different actors processing messages in parallel. Actors can be viewed much like objects that each have one thread for processing messages, though in practice there are fewer threads than actors and the actors share a pool of threads. The key is that only a single thread is being used by any actor at a given time, so race conditions do not occur in the mutable memory encapsulated inside of an actor.

Conceptually, the other big change between methods and actor messages is that while methods commonly return values and the code after a method call isn't invoked until the method is done, sending a message does not block the current thread and generally values are not returned. Akka provides something called the "ask pattern" which sends a message and gives back a `Future[Any]` that will have the response once one is given. Significant use of the ask pattern is considered poor practice though, so normally logic is structured so that any response is simply sent back and handled as a different message type.

One of the key aspects of the Akka actor system is that when you make a new actor, you get something called an `ActorRef` instead of the actual instance of the subtype of actor that was created. The reason for this is that it is impossible to directly call any of the methods of the actor through the `ActorRef`. All one can do is send messages. This helps to keep mutable data in an actor safe and prevent race conditions.

The following code is a short example using Akka that demonstrates two actors counting from 10 down to 0. It is one of the early examples from the course.

```
object ActorCountDown extends App {
```

```
  // Messages our actors use.
  case class StartCounting(n:Int, other:ActorRef)
  case class CountDown(n:Int)

  // The actor type for this example.
  class CountDownActor extends Actor {
    def receive = {
      case StartCounting(n, other) =>
        println(n)
        other ! CountDown(n-1)
      case CountDown(n) =>
        println(self)
        if(n>0) {
          println(n)
          sender ! CountDown(n-1)
        } else {
          context.system.terminate()
        }
    }
  }

  // Setting up the actor system and creating actors.
  val system = ActorSystem("SimpleSystem")
  val actor1 = system.actorOf(Props[CountDownActor],
    "CountDown1")
  val actor2 = system.actorOf(Props[CountDownActor],
    "CountDown2")

  // Sending a message to make them count.
  actor1 ! StartCounting(10, actor2)
}
```

Note the use of the exclamation point for sending messages. This syntax is borrowed from Erlang. In an actual project, the classes for the actors, like `CountDownActor` would be in their own files and the `case class`es associated with them would appear in the same files. While this example does not include any parallelism, it illustrates how actors communicate. Part of the beauty of the actor model is that if this were embedded in a larger actor-based application, the counting action would automatically occur in parallel with any other processing that might be happening.

IV. Project Description

The primary work evaluated for the students in this course comes from two projects that they develop individually throughout the semester. Students are provided design recommendations for each project. Both produce networked, multiplayer games that students have significant creative leeway in creating. One project is a graphical game that only has the multithreading that is required for networking and graphics using `java.net` and JavaFX. The other project is a text-based, multi-user game modeled after the MUDs that were popular on many college campuses in the early days of the internet. These multi-user, text-based games were predecessors to more recent graphical MMORPGs such as "Everquest" and "World of Warcraft". Indeed, students are encouraged to spend some time playing one of the few remaining active MUDs, SlothMUD [19], in order to get familiar with the style of play, which is quite different from the single-user text adventures that many students are more familiar with. This project is referred to in class and through the rest of this paper as the MUD project.

The original version of the MUD project, which was introduced in Spring 2011, was much like the graphical project in that it only required the most basic level of multithreading. Students don't have to write their own client as they are allowed to use `telnet` as the client. This was also typical of the original MUDs. The server required two threads because the `accept` method in `java.net.ServerSocket` blocks. This was normally written by spawning one thread in a Future for accepting new connections and processing the game in the main thread. To prevent race conditions when moving the new connections between threads, a `java.util.concurrent.BlockingQueue` was used as a thread-safe collection.

While this project worked well in many ways, the fact that neither project included significant parallelism meant that students often finished the semester without sufficient practice to retain the desired material on parallelism. For that reason, in Fall 2016 the MUD project was altered so that it includes significant parallelism through the actor model. In this modified version, many elements of the project become actors. In addition to the class that represents the players becoming an actor, the rooms and non-player characters (NPCs) are also written as actors. There are several other actors that manage parts of the program. All together, this provides a structure for the project that is highly scalable with few bottlenecks yet which can be implemented by second semester CS students.

There are eight checkpoints that the students have to turn in for assessment. These are described below and the general timeline of when they are implemented in the course can be seen in Table I. Note that the topics covered go well beyond parallelism and hit on many of the elements of a generic CS2 course. While every student works on these same tasks, they are given significant freedom in their choice of game theme, dynamics, and the details of how they implement various elements.

TABLE I
Checkpoint Tasks Course Timeline

Task	Week
Basic Setup	3
Maps(optional)	5
Actors	8
Networking	10
Priority Queue and NPCs	12
Combat	13
Shortest Path and TreeMap	14
Binary Heap	15

A. Checkpoint #1 - Basics

The initial implementation is single-threaded and non-networked. The goal is for them to implement the following commands that allow a user to move around and interact with items in a world that is read from a file.

- north, south, east, west, up, down - Moves the player in the specified direction.
- look - Displays the current room they are in and its contents.

- get *item-name* - Places an item from a room into player inventory.
- drop *item-name* - Places an item from player's inventory into room.
- inventory - Displays the items the player has.
- help - Shows the commands the player can use.
- exit - Terminates the program.

Students are given a design for this with suggestions for what classes to create and what methods go in them. Initially the only top-level constructs are a main object and classes for the player, room, and item.

B. Checkpoint #2 - Map

The second checkpoint is a basic refactoring of the initial implementation. In the initial implementation, the rooms are stored in an array and links between rooms use the numeric indices. This is challenging to maintain as the numbers aren't descriptive. Once students have been taught to use a Map, the program is changed so that rooms have keywords and links between rooms use those keywords. Note that this could easily be merged into the first checkpoint in curricula that have already covered Maps or left out completely with little effect on subsequent checkpoints.

C. Checkpoint #3 - Actors

The third checkpoint is the most significant for this paper as this is where the students refactor their program again so that instead of having a single loop that takes player input and executes the appropriate code, they change the overall design to use actors. If they have followed the suggested design, they are able to keep a large fraction of their previous code, but now the Player and Room classes are made to extend Actor and they have a `receive` method that handles messages in a manner very similar to the method calls that had been used previously.

This is one of two checkpoints in the MUD project that we tell students up front is more challenging than the others. Technically, it is just a refactoring as seen from the user side. When the code that students submit for this checkpoint is run, it will behave identically to what they turned in for the previous checkpoint. However, to get here the first thing they have to do is make a class extend `Actor`. This change breaks their code in ways that make it so they can't run it again to test it until they have made most of the changes required for this checkpoint. This prevents them from taking small steps and regularly testing their code.

There are several significant alterations that go into this checkpoint.

- The `Player` and `Room` classes extend `Actor` and get a `receive` method. Students must also add types for the messages each actor can receive.
- They add a `PlayerManager` and a `RoomManager` that are also actors that oversee actions related to players and rooms in general. For example, the code loading the rooms from file goes in the `RoomManager`.
- All references to the `Player` or `Room` classes become `ActorRef`s.
- All method calls to those types are replaced by sent messages.

To facilitate their progress on this checkpoint, we demonstrate in class the first steps in converting the `Room` class to be an actor and show how to move the existing code for loading in the world map into the `RoomManager`. Once students get their heads around the general changes required for moving to an actor-based program, there aren't many common errors they run into other than perhaps accidentally sending a message to the wrong destination. To help with diagnosing that situation in this checkpoint and all the checkpoints that follow, students are strongly encouraged to include code that prints warning messages when any actor gets a message that it doesn't know how to deal with.

D. Checkpoint #4 - Networking

In this checkpoint, students add in networking. If they have followed the recommended design for the previous checkpoints, this one is rather easy. After the actors are set up, a loop can be added at the end of the main thread that accepts connections to a `ServerSocket`. New connections are passed to the `PlayerManager` to create new instances of `Player`. In addition, because there are now multiple players, the `Room` needs to keep track of the players that are present in it and display this along with the other information for the room. Lastly, students are required to add two new commands for this checkpoint, "say" and "tell". The "say" command is used to communicate with other players in the same room. The "tell" command is used to communicate with a single user anywhere in the world.

E. Checkpoint #5 - Priority Queue and NPCs

The fifth checkpoint for the MUD project has the students add an element we call an activity manager. This is used to schedule events that will happen in the future. While Akka includes its own scheduling elements and this functionality could be achieved using them, we cover priority queues in the course and this provides a good application of them. At this point in the semester, their priority queue is written using a sorted linked-list.

To give the students something to schedule, they also add non-player characters (NPCs) in this checkpoint that are supposed to randomly roam through their world. The NPCs are themselves actors, and are treated the same as players by the rooms.

F. Checkpoint #6 - Combat

The other "large" checkpoint for the MUD project is the addition of combat. This checkpoint has the students add several more commands. They also use the activity manager from the previous checkpoint to add delays to combat that depend on the weapons being used. The new commands are as follows.

- equip *item-name* - Equips an item from the players inventory to be the active weapon for combat.
- unequip - Puts the currently equipped item back in inventory so no weapon is active.
- kill *victim-name* - Initiates combat with a player or NPC in the same room.
- flee - Attempts to leave the room to get out of combat.

The primary challenge with this checkpoint comes from the number of messages that have to be sent around and keeping track of what is going on as there are four actors that are involved: the player, the room the player is in, the activity manager, and the intended target of the attack. Students are strongly encouraged to draw out the actors as a directed graph with edges for the messages that are sent between them.

As a general rule, we allow the students to choose the exact rules and play style of their games in the class. Occasionally, we have students who want to limit the violence in their games. They can choose to disallow killing other players or even change the nature of "attacking" to make it less violent. For example, one student who created a Harry Potter themed game had wands act as the weapons and spell casting to disarm as the attack.

G. Checkpoint #7 - Shortest Path and TreeMap

When branching recursion is covered in the course, we make use of the fact that the MUD worlds are directed graphs and give them an assignment for finding a shortest path. They add a "shortestPath" command which prints out the movement directions to optimally travel to a particular room using a depth-first recursive search to find the path. This could also be done breadth-first, but in our course the breadth-first search is used in the graphical project.

The code for the shortest path winds up going in the `RoomManager` class because it is the only entity in the program that has knowledge of all the rooms. The fact that sending messages doesn't provide a return value unless you use the ask pattern, which gives you a Future, means that doing recursion directly with the rooms, as one might normally do in a standard object-oriented program, isn't feasible.

This checkpoint happens to occur just after we have finished discussing binary search trees, so students are also asked to replace the standard library `Map` that was used in checkpoint #2 with one of their own construction based on a BTS.

H. Checkpoint #8 - Binary Heap

The last checkpoint has the students upgrade their `ActivityManager` class to use a binary heap they have written for the priority queue. This is a straightforward refactoring that works well at the end of the semester when students are attempting to wrap things up.

I. Actor Hierarchy

The full actor hierarchy for the completed project is shown in figure 1. Each actor can operate in parallel processing messages, giving the full project significant potential for parallelism. Most students create fairly small worlds and never have a large number of people log in, so this parallelism isn't always exercised. We discuss one option to improve this in the future work below.

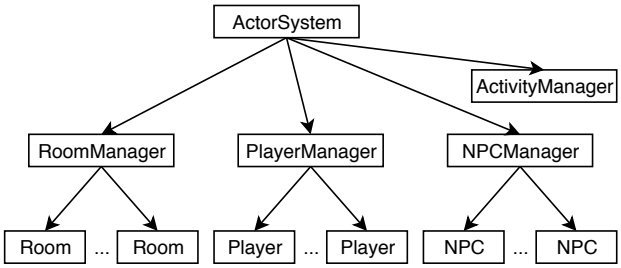

Fig. 1. This shows the actor hierarchy the project has upon completion.

V. Results

A. Student Perspective

To judge the effectiveness of this project and the use of the actor model in CS2, we requested that students fill out a survey. The questions included on the survey are listed below. For questions with limited options for the responses, the options offered are given in parentheses.

1) What semester did you take CS2? (Fall 2016, Fall 2017, Spring 2018, Fall 2018, Spring 2019, Fall 2019)
2) What are your current plans for CS study or if you have graduated, what did you do? (BS, Computing Second Major, Minor, Just taking courses)
3) How do you feel about the statement, "I enjoyed the MUD project."? (Strongly Agree, Agree, Disagree, Strongly Disagree)
4) Comments on how you felt about the MUD project.
5) Did you do anything with the MUD project after the course was over? (Y/N)
6) If you answered yes to the last question, can you please describe what you did?
7) How do you feel about the statement, "The MUD project helped me understand the actor model."? (Strongly Agree, Agree, Disagree, Strongly Disagree)
8) How do you feel about the statement, "The actor model helped me understand Object Orientation."? (Strongly Agree, Agree, Disagree, Strongly Disagree)
9) Have you done any parallel work since taking CS2 (later courses, side-projects, internships, professional, etc.)? (Y/N)
10) If you answered yes to the last question, please describe what you did and whether/how you felt that the parallelism covered in CS2 was beneficial for it. Please explain.
11) Do you have any additional thoughts related to learning Actors and the MUD project?

Questions 1 and 2 were included to give us a context for the other questions. This project has been used over multiple semester with a variety of students. The other questions were intended to provide information about the student perspective on actors and the projects. Question 8 was included specifically

because a previous student had mentioned that his work on the project in the actor model was significant to helping him understand object-orientation and encapsulation. We wanted to see if that perspective was held more broadly.

We had 44 students respond to the survey out of 99 students who took the course using the actor version of the project, giving us a 44.4% response rate. The number of responses and percentages for each of the agreement and yes/no questions are shown in table II.

TABLE II
SURVEY RESPONSES, AGREEMENT QUESTIONS

Question #	Strongly Agree	Slightly Agree	Slightly Disagree	Strongly Disagree
	Yes		No	
3	68.2%	27.3%	4.5%	0%
5	20.5%		79.5%	
7	77.3%	22.7%	0%	0%
8	52.3%	36.4%	9.1%	2.3%
9	29.5%		70.5%	

The results for questions 3 and 7 show that nearly all students enjoyed the project and felt that it helped them to understand the actor model. We also find from question 8 that the majority of students feel that working with the actor model facilitated their understanding of object-orientation.

Of the 32 answers we received to #4, all were positive with the exception of one comment that had a negative element to it and stated "Refactoring almost the entire project for actors was a low point, but overall I enjoyed it." It is somewhat surprising that students didn't mentioned that type of sentiment more given that checkpoint #3 is rather challenging as was mentioned above. Most of the other comments can be lumped into a few categories by theme with some longer ones hitting on more than one theme. Here we list the themes and how many comments hit on each theme then give some representative examples.

- Challenging/Satisfying (8)
  - The MUD project challenged me in ways I didn't think I could be challenged. Despite the frustration of solving these difficult problems, the project is very rewarding. After all of it I really feel like I did something cool
  - It was difficult especially once we got into the later projects, but building it up from nothing made it worth the effort. Also the creative aspect was enough to have fun, but not too much to make it stressful for "non-creative" types.
  - I thought it was challenging but it was also fun. It is a great feeling to interact with something you made and to improve it. I also felt like actors made a lot of sense to me
  - I thought the project was a really fun and interesting way to learn how to organize code. Once I figured out the "messaging" scheme of the actor model it was actually relatively straightforward to add all the required functionality. It was very satisfying in the end to have a playable game.
- Creative/Personalized (6)
  - It was a lot of fun! I enjoyed personalizing the game and learned a lot over the course of the project
  - I liked how much creativity was allowed so that we could make the MUD our own.
- Helped with Actors/Parallelism (5)
  - I wrote the code for pretty much the whole project and didn't even really need too much outside help. I felt more responsible for the end product and the actor system implemented felt like the most natural implementation of parallelism
  - I felt the project gave me a good understanding of actors as well as the pitfalls of running operations simultaneously.
- Favorite Project (4)
  - Probably my single favorite project so far in the degree.
  - I would say it's one of my favorite projects in our CS program. I think I still remember many details of it. Very instructive.
- Topic Coverage (3)
  - I enjoyed the project overall as it utilizes most of the concepts covered in CS2 in an actual program.
  - It was a lot, but I loved how everything we were learning tied into the game somehow
- Continued Working On It (3)
  - I loved the project. I have continued to build on it in my own time and couldn't speak more highly about how much I learned from it.
  - I enjoyed it and thought it was a great project for class. I've also included it on my resume and discussed it with interviewers. Specifically the head of engineering at Sledgehammer Games, where I'll work this summer, was very interested in the project and had a lot of questions about it.
  - It was one of my favorite projects I did while at Trinity. I continued to work on it even after I graduated.

The last theme tied in with question 5. While the responses to question 5 indicate that most students do not continue working on the project, we feel that having a solid 20% who did continue working on it after the end of the semester is pretty impressive, especially with some continuing to work on it after graduation. Most of the students who continued working on it were just playing with the world map and elements. However, a few students took it much further with one specifically mentioning that it is on their resume and has come up at a job interview.

The percent of students who answered "Yes" on question 9 might seem rather low, but as Table III shows, nearly half of the respondents only took the course in the year before the survey was given. If we throw out the students who took the

course in 2019 we find that 7 of 16 say that they have done more with parallel. In general, the comments in question 10 indicate that students felt that the coverage of parallel in CS prepared them well for future work. One student commented specifically on how the actor model helped them when they did MPI in the "Parallel and Distributed Computing" course.

TABLE III
SEMESTER ENROLLMENTS AND RESPONSE COUNTS

Semester	Enrollment	Responses
Fall 2016	9	2
Fall 2017	18	4
Spring 2018	19	8
Fall 2018	8	2
Spring 2019	35	21
Fall 2019	10	7

One of the common concerns with game-based assignments is that they don't appeal as much to female students as they do to male students. While we didn't ask about it on the survey, we can say that the students in the course sections that were included were about 33% female. Because the survey was anonymous, we don't know what the gender ratios were for the respondents. However, some of the comments included self-identifications thus we know that the respondents included females. We also know from interactions with students that there does not appear to be a significant gender difference in how students react to the MUD project. If we were to speculate on why that is, we would guess that it is a combination of the text interface and the flexibility that is provided to allow the students creative license with the project. In many ways, the creation of a text-based MUD is more of an exercise in story telling than standard games.

B. Instructor Perspective

The student perspective isn't the only one that matters. Indeed, there are many types of information that we simply can't get from students because they don't have appropriate perspective. While it was beneficial to have surveys from quite a few students who took the class in the past, including some who had since graduated and gone into the workforce, it is still important to get some information from the other side of the teaching dynamic.

As instructors, we felt that this project and the way the course was taught were highly effective in terms of achieving learning objectives. One aspect of the actor model is that when it is included in the MUD it forces the code to employ message passing and so alters almost everything at least somewhat. This is beneficial because students can't revert to writing single-threaded code and have it work after completing the third checkpoint. In addition to forcing them to write their code in a way that involves multithreading, this also provides opportunities for the rest of the semester to bring up multithreading and to mention how it is impacting what they are writing. This is significant because before bringing the actor model into the MUD we really felt that the learning objectives related to multithreading weren't being achieved. Students simply weren't being forced to think about it enough in the projects that they were working on. The benefits of this show up in both the quiz and test questions related to multithreading as well as in later courses. Students who have worked on the MUD project might not remember the details of how to set up an actor-based application two years later, but the do tend to remember what race conditions and deadlock are and have some memory of how they need to be addressed in code.

From the instructor perspective there is one significant potential drawback of this type of project: grading can be very time consuming. In our experience, this tends to be an issue with any assignments/projects that are very open ended and allow significant student creativity. If the students are allowed the flexibility to customize their projects, grading becomes a slow, manual process. The only way to test the MUD project is to run the server and connect one or more players to verify that various commands work. This process can be streamlined to some degree with various tools. We were able to improve many aspects of the flow of the projects by introducing `sbt`, the Scala Build Tool, so that compiling, testing, and running is smoother for both students and faculty. This is equivalent to using Maven or Gradle in a Java environment. In addition to helping with assessment, it also contributes to a valuable learning goal for a CS major given the ubiquitous use of build and dependency management tools in industry.

VI. CONCLUSIONS AND FUTURE WORK

Getting all the material that our students need to know before going into the workplace to fit into the limited number of hours that we have with them during a four year degree is becoming increasingly difficult as more topics, like parallelism, move from niche electives to required knowledge. Most schools don't have the option to address this by simply growing the major and adding additional required courses. Instead, we have to find creative ways to include more topics into the courses we already have without drowning students under the weight of too much material. We feel that teaching parallelism in CS2 with higher-level constructs like parallel collections, futures, and actors and giving students significant hands-on experience with the actors through the MUD assignment has worked well in this regard.

The one area that we would really like to be able to expand on is the ability to demonstrate to students that the MUD is actually using multiple threads and can handle a large load without crashing. We have done some work on a load tester but inevitably it runs into problems with the flexibility and creativity that students have in the project. While we suggest a particular format for the output for things like room descriptions, few students follow them exactly. As a result, making a program that can parse the text and give intelligent commands is challenging and this effort needs further work.

Another area of future work involves exploring the use of Akka Typed instead of the traditional, untyped actors in Akka. The strong static typing in Scala is a big motivation for using the language. However, the traditional actors in Akka receive

messages of type `Any`, the top type in the Scala type hierarchy, which are generally matched against the message types that the actor understands. By default, unhandled messages are simply thrown away. While this provides a certain level of simplicity, it means that errors occur when sending incorrect messages that lead to inappropriate runtime behavior instead of syntax errors. With Akka Typed, those mistakes would produce syntax errors. While that is certainly beneficial in many ways, it isn't clear that the use of Akka Typed would be as straightforward for students or as easy to convert to after beginning with the sequential code.

REFERENCES

[1] Andrew Danner and Tia Newhall. Integrating parallel and distributed computing topics into an undergraduate cs curriculum. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1237–1243. IEEE, 2013.

[2] Marcelo Arroyo. Teaching parallel and distributed computing to undergraduate computer science students. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1297,1303. IEEE, 2013-05.

[3] Max Grossman, Maha Aziz, Heng Chi, Anant Tibrewal, Shams Imam, and Vivek Sarkar. Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level. *Journal of Parallel and Distributed Computing*, 105:18,30, 2017-07.

[4] Joel C Adams. Injecting parallel computing into cs2. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 277–282. ACM, 2014.

[5] Sushil et. al. Nsf/ieee-tcpp curriculum initiative on parallel and distributed computing – core topics for undergraduates. http://tcpp.cs.gsu.edu/curriculum/sites/default/files/NSF-TCPP-curriculum-version1.pdf.

[6] Linda Marshall. A comparison of the core aspects of the acm/ieee computer science curriculum 2013 strawman report with the specified core of cc2001 and cs2008 review. In *Proceedings of Second Computer Science Education Research Conference*, pages 29–34. ACM, 2012.

[7] Erik Saule. Experiences on teaching parallel and distributed computing for undergraduates. pages 361–368, 05 2018.

[8] Barry Wilkinson, Jeremy Villalobos, and Clayton Ferner. Pattern programming approach for teaching parallel and distributed computing. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 409–414. ACM, 2013.

[9] Mark C Lewis and Berna Massingill. Graphical game development in cs2: a flexible infrastructure for a semester long project. In *ACM SIGCSE Bulletin*, volume 38, pages 505–509. ACM, 2006.

[10] Mark C Lewis and Lisa L Lacher. *Object-Orientation, Abstraction, and Data Structures Using Scala*. Chapman and Hall/CRC, 2017.

[11] LightBend Staff. Akka. https://akka.io.

[12] S Fincher and M Petre. Project-based learning practices in computer science education. In *FIE '98. 28th Annual Frontiers in Education Conference. Moving from 'Teacher-Centered' to 'Learner-Centered' Education. Conference Proceedings (Cat. No.98CH36214)*, volume 3, pages 1185,1191 vol.3. IEEE, 1998.

[13] Akka.NET Projet. Akka.net documentation. https://getakka.net.

[14] John K. Waters. Akka nearing 10 years hits 200k dev milestone. https://adtmag.com/articles/2019/04/24/akka-milestone.aspx.

[15] Mark C. Lewis. Csci1321-s19. https://sites.google.com/a/trinity.edu/csci1321-s19/.

[16] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[17] Joe Armstrong. erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[18] Steve Vinoski. Concurrency with erlang. *IEEE Internet Computing*, 11(5):90–93, 2007.

[19] Unknown. Slothmud. http://www.slothmud.org/wp/.