May 2016

Empirical Study of Concurrent Programming Paradigms

Patrick M. Daleiden
University of Nevada, Las Vegas, daleiden@unlv.nevada.edu

EMPIRICAL STUDY OF CONCURRENT
PROGRAMMING PARADIGMS

By

Patrick M. Daleiden

Bachelor of Arts (B.A.) in Economics
University of Notre Dame, Notre Dame, IN
1990

Master of Business Administration (M.B.A.)
University of Chicago, Chicago, IL
1993

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2016

www.manaraa.com

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

April 15, 2016

This thesis prepared by

Patrick M. Daleiden

entitled

Empirical Study of Concurrent Programming Paradigms

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Andreas Stefik, Ph.D.
Examination Committee Chair

Ajoy Datta, Ph.D.
Examination Committee Member

Jan Pedersen, Ph.D.
Examination Committee Member

Matthew Bernacki, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

ii

Abstract

Various concurrent programming paradigms have been proposed by language designers in an effort to simplify some of the unique constructs required to handle concurrent programming tasks. Despite these different approaches, however, there has been no general clear winner accepted by software developers and different paradigms are regarded to have strengths and weaknesses in certain areas. This thesis was motivated by the desire to investigate the question of whether or not there are measurable differences between two widely differing paradigms for concurrent programming: Threads vs. Communicating Sequential Processes. The mechanism for observing and comparing these paradigms was a randomized controlled trial of two groups of participants who completed identical tasks in one of the two paradigms. The study was run in Fall 2015 with 88 student participants primarily from the Department of Computer Science at UNLV. I examined programming accuracy and comprehension rates among participants in three different common shared memory problem areas introduced by concurrent programming. The results were measured using a token accuracy map algorithm which matches the token strings of a participants answer compared to a correct solution. The overall results show that for two relatively straightforward tasks using shared processes and memory, both paradigms were reasonably well understood, with a possible small learning advantage in favor of CSP in two of the tasks. In a more complex example combining task co-ordination and memory sharing, however, the participants in the CSP group struggled to grasp the guarded blocking and communication channels needed in the CSP model and performed measurably worse.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The natural world we live in is filled with parallel systems, from weather and climate, to physics, astronomy or biological analysis. Within these systems, complex, simultaneous and related events interact many ways. Scientists aim to understand, predict and sometimes alter natural systems by creating complex simulation models, measuring natural phenomenon or examining the building blocks of systems. There has been a proliferation in the amount of scientific data generated from digitally monitoring and analyzing these real world systems and it is growing at an increasing pace. For example, in bioinformatics, a single sequenced human genome is about 140 gigabytes in size and since sequencing machinery is becoming more low cost and accessible, there is more data generated every year. The European Bioinformatics Institute today has a database of around 20 petabytes of data on proteins and molecules, which will continue to grow in the future. These data sizes pale by comparison to some fields, such as physics, where The Large Hadron Collider at CERN currently generates around 15 petabytes of data every year. [13] By 2020, the proposed international SKA radio telescope project is expected to come online and will be capable of producing 700 terabytes of data per second, an amount which would eclipse the size of the internet in a few days, while its central computer will have the processing power of 100,000,000 PC's. [1][14]

Is it often not feasible to answer research questions using these databases on single machines using traditional sequential methods. It requires specialized concurrent approaches to breaking down the problem and performing computation in parallel. Concurrency is required in these cases not just to perform computation in reasonable times, but to perform analysis which might otherwise be impossible. Computer architectures have developed to support ubiquitous parallelism in various ways to aid with this class of problem, but the issues in programming and data management are complex and difficult to manage. Parallelism takes many forms in these architectures including

1

multicore processors, multithreaded cores, parallel co-processors, and vector processing units (both on the CPU and in graphics cards). These individually parallelized elements are also commonly combined in multiprocessor environments with shared and distributed memories.

There are various alternative paradigms for solving these systems on various architectures, but the computer science community has differing opinions on the best approaches. These thesis was motivated by the desire to investigate the question of whether or not there is a difference in programming performance from competing paradigms for certain types of problems. This thesis reports on a randomized controlled trial examining the human factors impacts of two popular paradigms for concurrent programming: Threads and Communicating Sequential Processes. The goal was to to draw inferences about the intuitiveness and understandability of each paradigm. The study was run in Fall 2015 with over 90 student participants primarily from the Department of Computer Science at UNLV. The study examined programming accuracy and comprehension rates among participants in three different common shared memory problem areas. The results were measured using a token accuracy map algorithm which matches the token strings of a participant's answer compared to a correct solution.

Today, parallel hardware architectures are ubiquitous in everything from cell phones to super-computers. This transition has been forced by the desire of hardware manufactureres to increase processing power steadily in the face of physical limitations on clock speeds from heat dissipation and differences in memory access times, as well as the exhaustion of automatic scaling from instruction level parallelism. Parallelism takes many forms including multicore processors, multithreaded cores, parallel co-processors, and vector processing units (both on the CPU and in graphics cards). Individually parallelized computers are also commonly combined in multiprocessor environments with either shared or distributed memory designs. [15]

While it is true that computer archittures have transitioned almost exclusively to parallel approaches, it is not necessarily true that *all programming* must also transition in every case. Often times performance scaling is achieved with sequential programming by co-ordinated horizontal scaling approaches to break down the problem with very little co-ordination required. In many cases, performance is sufficient using sequential algorithms and adding in parallel processing support creates an unnecessary complication or even a performance decrease from co-ordination overhead. In cases where scaleable future computation and performance is required, however, concurrent programming is likely the only avenue to gain increasing performance and so, therefore, must be understood.

The motivation for this thesis is to examine the human factors impacts of concurrency with a goal of better designing programming languages and approaches that can be integrated naturally and intuitively by programmers. Since the issues are difficult to solve with concurrency, a better understanding of precisely what areas are stumbling blocks may help us understand why and design approaches around them.

1.1   Concurrent Paradigms

Various concurrent programming paradigms have been proposed over the years and with the widespread adoption of parallel hardware architectures, the issue is frequently discussed. The base model for comparison of any approach is usually a threads model. The reason is that aside from the conceptual appeal of simply extending the common sequential model, threads are easily implemented in most languages, operating systems and hardware because they require only minor additions and changes in syntax. As a result, most languages and computing systems support some form of threading and it remains the most popular and familiar model of concurrency. It is a good basis for comparison because it is well known and widely used.

In scientific computing, concurrency is often essential for data parallelism and scientific users have adapted to concurrency in alternate ways that include message passing and vector operations. The hardware architectures in use for these types of parallel applications are different than general purpose single computers, however, and a simple thread translation is not as logical. Parallel programs for these architectures do not generally rely on shared memory abstractions since communication costs between processes must be considered and designed into the programs. In this thesis, I investigate the human factors impacts of a basic threads approach compared with a CSP-style process approach.

1.2   Threads

The fundamental approach to parallelism in a threads-based paradigm is the concept of multiple processes in a program operating in parallel sharing a single memory space. This is often viewed as being intuitively appealing since it mimics the computer hardware architecture with shared hardware threads of execution and multiple cores which share memory at various caching levels. Threads conceptually are a logically appealing extension of the basic sequential model of computing to allow for concurrent activity and apply naturally to embarassingly parallel-type algorithms where

3

work is simply divided into segments. In this paradigm, there is no additional level of abstraction for concurrency introduced beyond threads.

Since shared memory can be accessed by all threads simultaneously, the paradigm includes various mechanisms to lock memory during critical sections to prevent conflicts. Additionally the model must provide mechanisms for synchronization to allow a global sequential ordering of all threads at certain points where that is required. This model is also referred to as fork-join parallelism, since the program design involves threads forking execution at points in a program where tasks can be parallelized and then joining back again for sequential portions.

In his paper, "The problem with threads", Lee [11] argues that "Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism." His position is that by using a threading paradigm, concurrent programmers fundamentally assume the stance of pruning nondeterminism in their code to achieve the desired behavior. This pruning is done with tools like semaphores, monitors and overlays, but this activity becomes increasingly more complex and process interaction escalates. By contrast, he argues that a better approach is the reverse, where deterministic composable components are constructed and nondeterminism is introduced intentionally where it is needed instead of removed where it is not. He makes the case that "achieving reliability and predictability using threads is essentially impossible for many applications."

The major complication for programmers using threads comes not from understanding the basic concept of threads or how to launch and use them, but in managing the complications introduced by interacting processes, especially with the need to lock and unlock shared memory to avoid race conditions and deadlock. Deadlock situations become increasingly common and difficult to debug and discover in complex systems with many interacting threads and locks. Even simple common examples create so many opportunities for behavior that is not thread safe, it is difficult to imagine that concurrent system designers have properly anticipated all the possible interactions. Lee takes the position that these complex interleavings are so common that it is likely most general purpose multithreaded applications are filled with latent concurrency bugs that will only be discovered as the degree of multi-core parallelism increases in computing.

The coding sample and task approach in this study uses a Java-style syntax adapted to the Quorum Programming Language [2] for the thread paradigm. The required keyword additions include just `synchronized` which creates an implicit lock on a method, variable or class in the

4

same way as Java. A `Thread` class is assumed and also behaves like Java with `Run()` and `Join()` methods for thread management.

1.3 Communicating Sequential Processes

The fundamental approach to parallelism in a process-based paradigm is the concept of multiple processes having local memory and operating independently and sequentially on their own, but sharing data by explicitly passing messages over channels. A key feature of this approach is a lack of reliance on shared memory and the prevention of conflicts resulting from that. Message passing is done explicitly by processes and is non-buffered and blocking. The message passing is done on dedicated one-way channels that effectively wire processes synchronously together. Since channel reads and writes block because they are synchronous, they also provide synchronization points in the overall system and guarantee deterministic behavior. The result of this is the creation of composable components that can be linked together. The message passing infrastructure prevents memory conflicts and provides built-in synchronization. Synchronous reads ensure that deterministic composable building blocks are constructed to avoid deadlock and race conditions.

Nondeterminism is deliberately built in to the program through some form of `alt` block (labeled `choose` in this study) at specific points in the deterministic execution of some component. The `alt` block essential allows the program to probe a set of channels for messages before it commits to a synchronous (blocking) read. If none are available, the program blocks until one is available to read. This is called a guarded block and it enforces the condition that a read only occurs on a valid channel when that behavior is desired. The guard prevents deadlock from inadvertant blocking and allows non-determinism at a deliberate point in the program execution. A `par` block (labelled `concurrent` in this study) is used conceptually similarly to the way a thread is launched in other languages. All statements and method calls inside the `par` block are launched "simultaneously" and executed concurrently as determined by the underlying OS and compiler. The non-deterministic `alt` block applies not only to channel reads and writes, but to any internal conditions or variables in the composable process and the decision which statement to execute is made within the process and not subject to race condition or other outside interference.

Overall, this approach retains correctness of the program and manages the nondeterminism proactively to prevent harm or system failure. Determinism in this case does not refer to the timing or ordering of the particular execution history of the processes, but to the computation being performed safely and correctly *regardless* of ordering. Non deteriminism is introduced at a

5

point where it is expected and can not create unintended consequences.

One of the key advantages of using a process-based approach is the ability to take advantage of the formal CSP process algebra which can be used to reason about problems and prove the correctness of solutions. The approach was originally developed by Hoare [6] as a prototype to describe concurrent problems from the perspective of processes and communications. It was later expanded it into a full process algebra and formal verification system. [7][3]

The coding sample and task approach in this study uses an `occam`-style syntax adapted to Quorum for the CSP paradigm. The required keyword additions include: `choose` and `or` for an `alt` block and `concurrent` for a `par` block. Channels are typed using the generics system in Quorum (e.g `Channel<integer>`) and each Channel has methods for `GetReader()` and `GetWriter()` to create the proper end of the channel which can then be passed to the appropriate processes according to the program logic. Synchronous reads and writes are invoked by calling `Read()` and `Write()` on the channel ends.

Chapter 2

Related Work

Human factors evidence in programming language design has been generally deficient, however there is a growing community of scholars advocating for the increased use of replicable scientific evidence in studying programming languages in regards to human factors. The intent of this thesis is to build on this growing area of research in empirically examining these issues as they relate to concurrency.

I reviewed the literature with a view to finding papers or studies regarding human factors evidence comparing the difficulty of parallel programming to sequential programming, but was not able to find any data to provide a quantification of any differences. Many authors tend to mention or acknowledge that parallel programming is "harder" in various qualitative ways, however, I did not find any specific evidence to support these claims. Intuitively this seems to be a logical statement to make, but it would probably be beneficial for the software engineering or language design community to have some measure of how much harder it might actually be and in what specific areas.

One relevant study to the intuitive ability of students to comprehend parallel concepts in general was conducted by Lewandowski et al. [12] The authors examined what they call "commonsense computing," defined as knowledge students have before formal study of computing. The study involved 66 students in the first week of a computer science class in college at five different venues. The students were presented with a scenario involving multiple sellers of a fixed number of assigned seating concert tickets. They were asked to write an answer in English about what problems might be encountered and how they might be solved. The results demonstrated that 97% of the students identified that a race condition may occur and 73% of the students identified at least one possible solution. These results indicate that even untrained novices seem to have an intuitive grasp on parallel concepts.

7

Rossbach et al. [21] examined results of programming assignments for 237 students and 1,323 programs in an operating systems course over a three year period and identified and classified the types of synchronization errors that students had, as well as their frequency. In total, the authors identified ten types of errors that occurred. In the study, at least 50% of the students made at least one of these errors in an example requiring locking.

2.1   Literature Reviews

In an effort to locate other empirical studies related to concurrency issues, I examined a number of published literature reviews in the software engineering area. The most comprehensive review of empirical studies that utilize human factors evidence in programming language design was recently the subject of a dissertation by Kaijanaho. [8] Kaijanaho reviewed and scored peer-reviewed papers from the 1950s and discovered only 156 papers that compared programming language designs with human factors measures. Of those 156 papers, only 35 used controlled experiments and only 22 of these used formal randomized trials, the typical standard for scientific research. [24] Of these 22 papers, there were 4 experiments related to concurrency, specifically Software Transactional Memory. Other systematic literature reviews identifying empirical studies in software engineering generally include those performed by Kitchenham et al. [10] and Zhang et al. [26] make no specific mention of papers on concurrency.

2.2   Software Transactional Memory Studies

The most popular topic for empirical studies in concurrency has been the Software Transactional Memory (STM) method. The 4 studies identified in the Kaijanaho disseration all studied STM in some empirical manner compared to the traditional Threads/Locks paradigm. The main concept of STM that most of these researchers claim is the most appealing aspect of the STM model is the replacement of locks with atomic transactions. In general, the studies found some advantages and disadvantages of each approach with no conclusive answer as to which is better.

Rossbach et al. [21] conducted the first study on STM using 237 students and 1,323 programs in an operating systems course over a three year period where the students implemented solutions to the same problem using multiple paradigms, including coarse, grain locking, fine-grained locking, and STM transactions. They found that although inexperienced programers found difficult syntax of STM to be a barrier to entry, on the whole, the number and types of programming errors was

8

much lower for a transaction-based approach compared to fine-grained locking. They also found that development time was lower for the STM approach. The study has been criticized by some researchers [18][4] for methodological errors in group assignment, however, the sample size is much larger than other studies.

Pankratius and Adl-Tabatabai [18] performed a study on six teams of students in a graduate level computer science course which claimed to be the first study documenting how teams used STM in a realistic example over an extended period of time compared to Threads/Locks. The empirical study examined quantitative and qualitative data on performance, hours spent on phases of development, code metrics and easy of understanding of code. The study found that the STM teams were the first to achieve a working prototype and they spent less than half the time debugging segmentation faults in their code. Additionally, the STM teams' code was judged to be easier to understand by a group of outside industry experts. The downside for the STM groups was that the had more difficulty with performance and implementing queries. The major drawback to this study was the very limited participant size of 12 students in 6 teams.

Castor et al. [4] performed a comparative study with 51 undergraduates on the STM approach compared to locks using Haskell with the stated goal of evaluating whether STM "delivers on its promises of avoiding common concurrent/parallel programming pitfalls." Haskell, a purely functional programming language, was chosen because of it contains an implementation of an STM approach in its general distribution, as well as a lock-based control mechanism. Overall, the study concluded that number of errors, size of the solutions (measured in lines of code) and development time did not differ significantly between the two paradigms. They did find that STM programmers finished the assignments quicker where mistakes were non-concurrency related.

Nanz et al. [16] examined Threads/Locks compared to alternate methods including both STM and an Actor model similar to CSP. The primary aim of the paper was to describe in detail a methodology for empirical studies to compare concurrent programming languages. Although a pilot study was performed using the technique, the primary contribution of the paper is in examining the issues of evaluation including comprehension, debugging and correctness.

2.3   C/C++ Language Extensions for Parallelism

A more recent empirical study examined a different aspect of concurrency in comparing the C/C++ language extensions Cilk Plus to OpenMP [5]. The study investigated the usability and correctness tradeoffs of using *reducers* from a learnability perspective in the classroom. This was a preliminary

9

study involving eight students in a masters level course in Secure Coding. The authors focused on human factors issues instead of just performance. The study involved each student completing the same assignment to parallelize an existing sequential program in both methods with the order randomized in two groups. Overall the students had considerable difficulty in writing correct programs with either method, despite a relatively simple task as well as course lectures and reading assignments on the specific topics in advance. Many students submitted solutions with race conditions or a failure to use the reducer as instructed. The sample size was small, but the researchers conclude that new instructional techniques or tools are needed to improve the students' performance.

Chapter 3

Experiment

In this chapter, I will review the design of the randomized controlled trial that was conducted on these two concurrent programming paradigms. In the study, the participants are asked to complete three programming tasks which involve issues of concurrency in one of the two paradigms. The central research question that the study attempts to address is:

RQ1: Which concurrent programming paradigm do students find intuitive in solving common problems in parallel computing?

Additionally, the study was designed to shed light on the question of whether programmer experience was a factor in participants' performance in completing the tasks. A second research question is therefore posed in the study:

RQ2: Does programmer experience contribute to participant performance in the programming tasks.

The randomized controlled trial used for the experiment adheres to the design and properties specified in the WWC Handbook [24] for evaluating the integrity of studies of this kind. Kaptchuk [9] provides a detailed history of randomized controlled trials including the reasons and decisions made by the scientific community in their development over time. A further discussion of the statistical analysis and procedures used in the evaluation of the results can be found in Vogt [25].

3.1 Hypotheses

The primary purpose of this experiment is to find evidence of the impact of two programming paradigms on the ability of students to solve common problems in concurrent computing. The null hypotheses for our investigation of the previously identified research questions are:

Null hypothesis H0-1: There is no impact of programmers ability to accurately complete concurrent programming tasks using a Threads paradigm compared to CSP.

Null hypthesis H0-2: The programmer's education level, as a proxy for programming experience, has no impact on the ability of the programmer to complete the programming task.

We will reject the null hypothesis HO-1 if we find statistically significant differences in the accuracy rates of participants between the different paradigms. The overall accuracy rates are measured by a token accuracy mapping algorithm that compares the actual code submission of the participant with a correct solution. Common statistical techniques will be used to compare the scored results. Similarly, we will compare the scored results for participants at different positions in the academic pipeline and programming experience across each group to determine if the null hypothesis H0-2 should be retained or rejected.

The dependent variable in this experiment is the accuracy score for a participant on a task as measured by the overall token accuracy mapping algorithm. The accuracy score was determined automatically by the scoring algorithm and is reflected as a percentage of correctness from 0% to 100%. The independent variables are i.) the paradigm group (Thread or CSP), ii.) the level of education and iii.) the task. The level of education was self reported as Freshman, Sophomore, Junior, Senior or Graduate and all student participants were from courses offered at the University of Nevada, Las Vegas. The tasks were numbered from 1 to 3 and were the same for each group.

The token accuracy mapping approach also enables us to examine the comprehension and use of particular tokens across groups in addition to the overall accuracy. This data can be used to determine which individual elements of the different paradigms are most intuitively understood or likely to be correct. Additionally, the automated testing system recorded snapshots of the input area every 10 seconds until the participant made a final submission. These snapshots enable the token accuracy mapping scoring algorithm to be run on the code to provide information on the overall accuracy score over time, as well as data on the individual token accuracy.

3.2 Experimental Design

The participants in this study were assigned to one of two paradigm groups for their tasks. The first group used the Thread paradigm and the second used the CSP paradigm. Participants were randomly assigned to one of the groups automatically by the web-based program used to administer the study. In order to balance the participants in each group, the program randomly selected the ordering of the next two participants at each academic pipeline position so that one was assigned to the Thread group and one to the CSP group. After two participants entered the study, the program again randomly selected the ordering of the next two participants. The task descriptions and instructions were identical for each group. In an attempt to filter out the effect of language syntax between the groups, the language used in the code samples for each paradigm was also the same. Minor syntatical differences were required to implement the different programming paradigms (for example, the inclusion of the keyword `concurrent` in the CSP group), but otherwise the languages were the same. The language used in the study, Quorum, was chosen as a neutral language with minimal syntactic elements in a further effort to reduce the impact of language specific features on the results.

We chose to evaluate the Threads paradigm compared to the CSP paradigm because they represent fundamentally different approaches to solving parallel computing problems. The strengths and weaknesses of these two paradigms have been debated in academic literature and textbooks for many years and have been implemented in different programming languages and styles, however, to our knowledge, there has not previously been any kind of randomized controlled trial attempting to independently measure whether one style results in more accurate programming than the other. The tasks were selected to measure the ability of programmers to solve simple forms of building block level parallel tasks to determine the relative accuracy rates for each paradigm.

3.3 Testing Application

In order to administer this experiment, I designed and built a web-based testing application ("WebTester") which we anticipate using as the basis for other randomized controlled trials in the future. The WebTester uses a standard Apache webserver configuration with an SQL database and was written in HTML, PHP, SQL and JavaScript. It runs in a standard browser and was tested in Chrome, Firefox and Safari with no noticeable differences between them. The URL used during the time when the study was run was `http://jedi.cs.unlv.edu/langstudy/`, **not**

13

`https://www.quorumlanguage.com/langstudy/` as shown in screenshots in this document. This is important to note because there was no identification of the Quorum Programming Language presented to the participants. Throughout the study, the participants did not have any knowledge of what language was being used, consistent with the design intention of having the study be language syntax neutral so much as is possible.

The advantage of using an automated testing application compared to a proctor administered study is primarily in the consistency of test conditions for all the participants. Each participant was presented with exactly the same instructions and tasks (except for the controlled difference of programming paradigm) and for the same amount of time. The program logs events and periodic code snapshots with a timestamp to confirm this standardization. Additionally, based on previous studies we have run, we found that having the study administered through the web enhanced our participation rate among students who were asked to complete the study. Since it is impractical to schedule times when many students can attend a session together and still get large numbers of participants, this approach also allowed students to complete the study at a convenient time and still experience the same test conditions.

3.4   Study Protocol

As a result, the standardization introduced by the WebTester, the experiment protocol and IRB requirements were strictly enforced and tracked to ensure consistent instruction and timing for each section among all participants. Upon entering the URL in their browser, each participant was introduced to the study with a start screen shown in Figure 3.1. As the screenshot shows, the participant was given some introductory information about the study, how long it was expected to take and the general requirement that it be completed in a single sitting because of its timed nature. The introductory page also explained the rules of the study and the overall intent of this information was to give a general overview of what the participant should expect so they could make a decision whether or not they were interested in proceeding and how much time it would take so that could determine when they should start. The tracking only began after this point, so participants could come to this page more than once if they chose.

In order to proceed, the participant was required to submit an email address to uniquely identify themselves. Since extra credit was offered by many of the course instructors as an inducement to complete the study, it was necessary to track this information. Since we conducted this study remotely and without supervision, it was also necessary to use this identification as a way to de-

14

termine if a person was completing the study for a second time or if it was restarted (since this would invalidate the result). The email address was recorded in the participant database as soon as the user clicked the "Begin Study" button to leave the screen and if the address was already there, the user was not permitted to advance. Once this check was made, the participant was internally given an ID number and all responses and events were recorded in the primary database tables by that ID number, not by their personally identifying information. The personal information was stored in a separate table and used solely for the extra credit purposes, as required by the IRB, so eliminating it is possible.
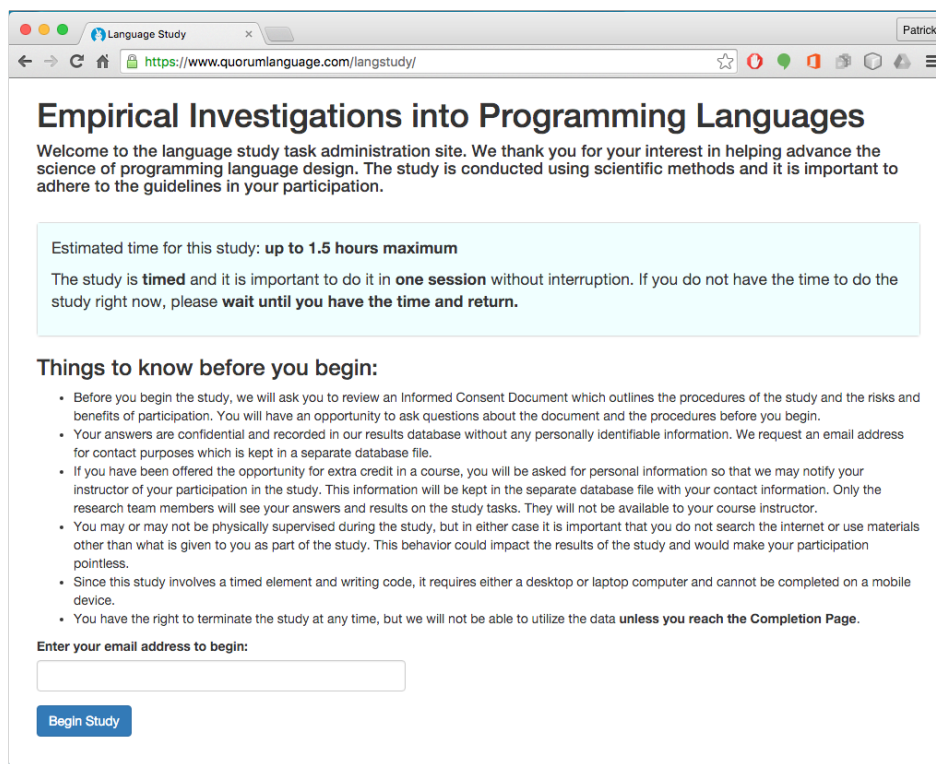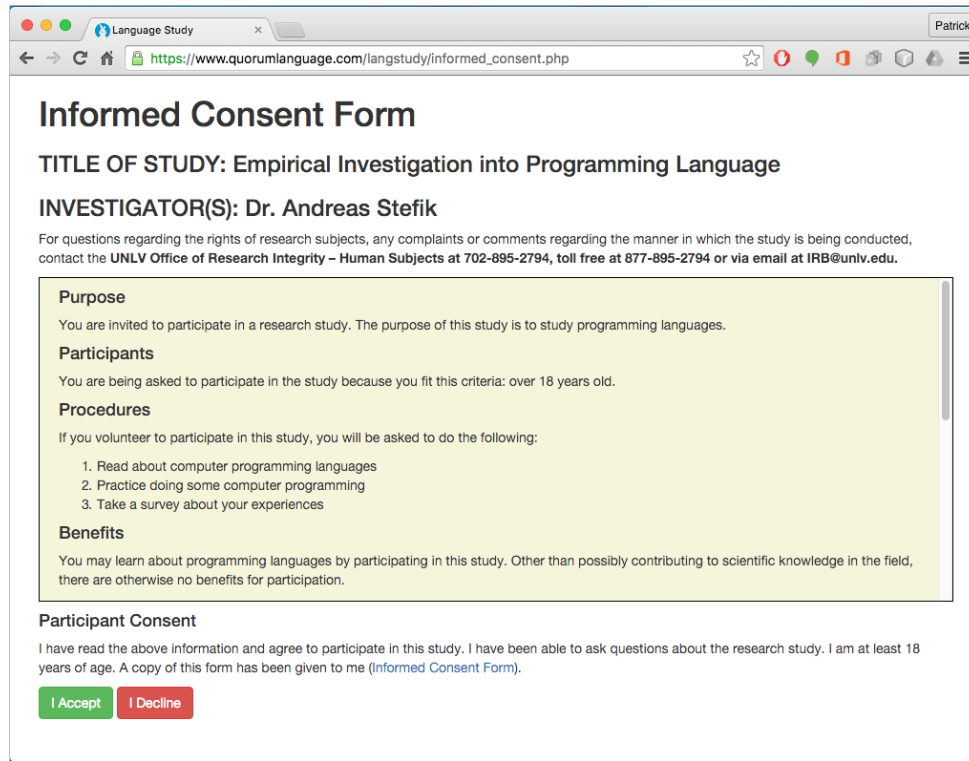
Figure 3.1: Start screen.

After entering their email address and leaving the introductory screen, the participant was im-mediately presented with a digital copy of the IRB approved Informed Consent Form. The form contained all of the information on the IRB stamped form, to which the page included a link to a downloadable copy for the student's records. Per federal regulations for human subjects research, the participants are required to be appraised of all the information contained on the form and have the ability to ask questions. After reviewing the form, the participant must then affirmatively accept the agreement in order to provide consent and get permission to proceed. If the participant

15

declined, they were thanked and the study was terminated. Figure 3.2 depicts a screenshot of the visible portion of the form and a full copy of the form is included with the IRB approval documents in Appendix A.


Figure 3.2: Informed Consent screen.

After the participant successfully gave their consent, the study moved into the regular protocol, which first entailed giving instructions about how the study would work. It begins by postulating a scenario for the student to imagine that they are in a new job and being asked to write a program in an unfamiliar language. They are told they will be given code samples in a particular language and then instructed to write programs for solutions to similar tasks in the given language. As shown in Figure 3.3, the page clearly explains that they will be shown a series of three code samples for a timed period, after which they will be given a task and asked to type a solution in the form of a computer program in an editor box on the same screen. They are advised that they will have access to the code sample while working and that they can freely copy relevant sections to their solution because the sample is correct.

The participant was also given detailed rules to adhere to, such as reducing distraction, not using the internet, trying to complete as much as possible and using as much of the allotted time

16

as they need. We reiterate in several places that the participant use their best effort to complete as much of the solution as possible, even if they do not have a full understanding. This is important because in the results we will look at the accuracy mapping of individual tokens as well as overall understanding in order to try to determine which particular parts of the programming paradigms were confusioning or intuitive. Obviously, since the study was not proctored or recorded, we have no way of knowing if the participant followed the rules, so this is one drawback of the automated testing approach.
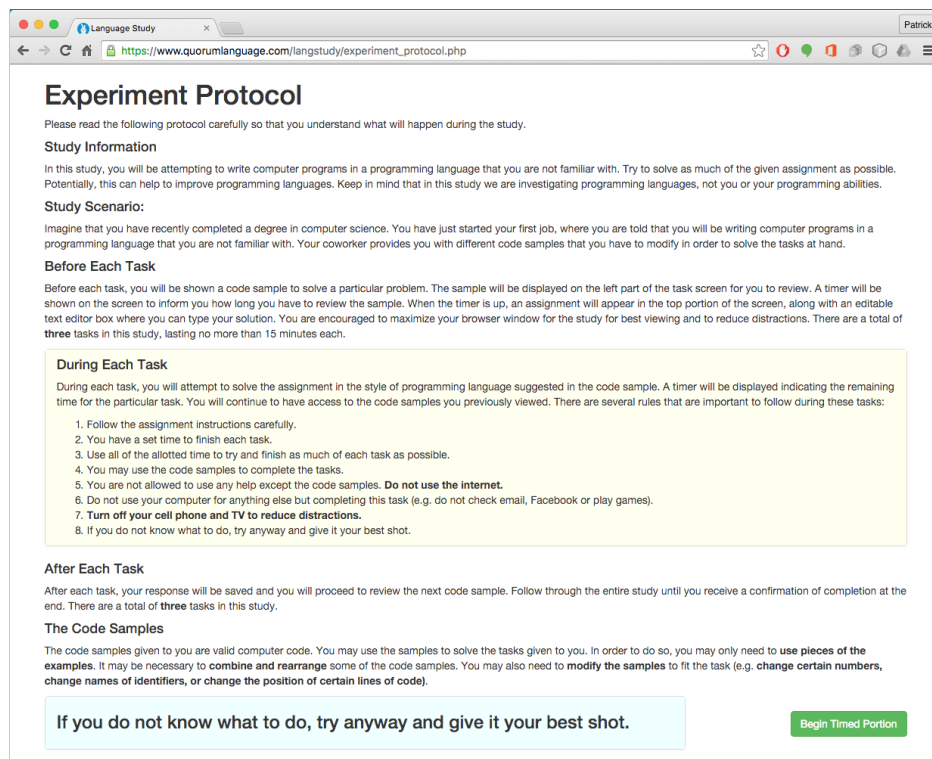
Figure 3.3: Experiment Protocol screen.

The classification page shown in Figure 3.4 included most of the relevant demographic information that we chose to collect for this study (with the exception of course/programming experience and potentially biasing classification information). Most of the questions on this page were directed to understand which academic group they would be placed in. This was necessary prior to the commencement of the study for group balancing.

The logic for determining which paradigm group the participant was assigned to was based on random assignment. At the beginning of the study, participants were randomly ordered for the group assignment at each level (which in this case was only 2 paradigms). Once a participant moved

17

Figure 3.4: Classification screen.

into the active part of the study after this page they were assigned to either the CSP or Threads group based on this random initial ordering. The next participant at that academic position was then assigned to the other group. After two participants entered at that academic position, the computer selected a new random ordering for the next two participants at that academic position. This assignment technique ensured random assignment, while balancing the numbers assigned to each group.

The assignment technique was not intended to ensure perfect balancing, since invalid or incomplete attempts "used up" the ordered group assignment. It was thought that these invalid responses would balance each other out however and that the overal approach should yield roughly similar group sizes at each level. As later shown in Table 5.1, the final tally of participants with valid responses across each academic level were similar across the language group paradigms, so the assignment methodology apparently worked.

The balance of the demographic data collected on this page was used to run our statistical analysis in various ways, examining different independent demographic variables, such as performance by native language, country of origin or programming experience. The disability data is collected as part of long term efforts in the lab to analyze data and test impacts of these language studies

18

on people with disabilities. Although this study was initially done just with students at UNLV, the testing application and demographic forms anticipate future studies in other locations.
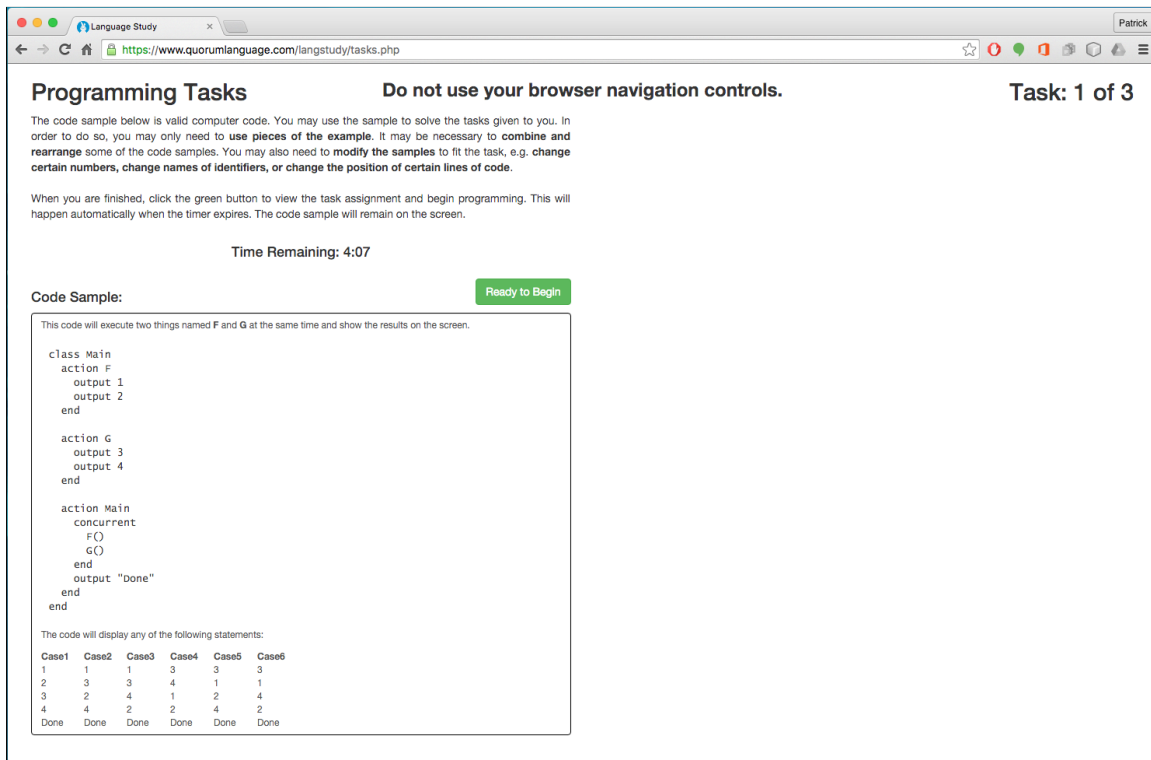
Figure 3.5: Code Sample screen.

The next page of the application, shown in Figure 3.5 finally brings the participant to the actual programming portion of study. It shows the code sample page, which includes the following things:

- Text-based instructions at the top left explaining what they are looking at and instructions on how to proceed when ready.

- The timer above the code sample showing the time remaining, which is a dynamic html field that counts down.

- The actual code sample in the box on the bottom left which describes what the code does, the actual code and the possible output.

- A reminder in the top center not to navigate away from the page with the brower controls.

- A status indicator in the top right showing which task the participant is on.

19

- A single green button marked "Ready to Begin" which pretty clearly highlights what the user should do next.

After the user clicks the "Ready to Begin" button or the time expired on the Code Sample page, the user is brought to they main task page of the study, show in Figure 3.6. In fact it is still the same `tasks.php` page controlled by JavaScript commands to dynamically display the only possible correct information to the user. By programming everything on to the same page, I could ensure proper timing and prevent a participant from going backwards. Even if they hit the back arrow in the browser, they were not brought back to the previous sample or task, just to the Classification Screen. We continue to display the reminder at the top of the page not to use the browswer navigation controls, but this methodology ensured that the user could not disrupt this.
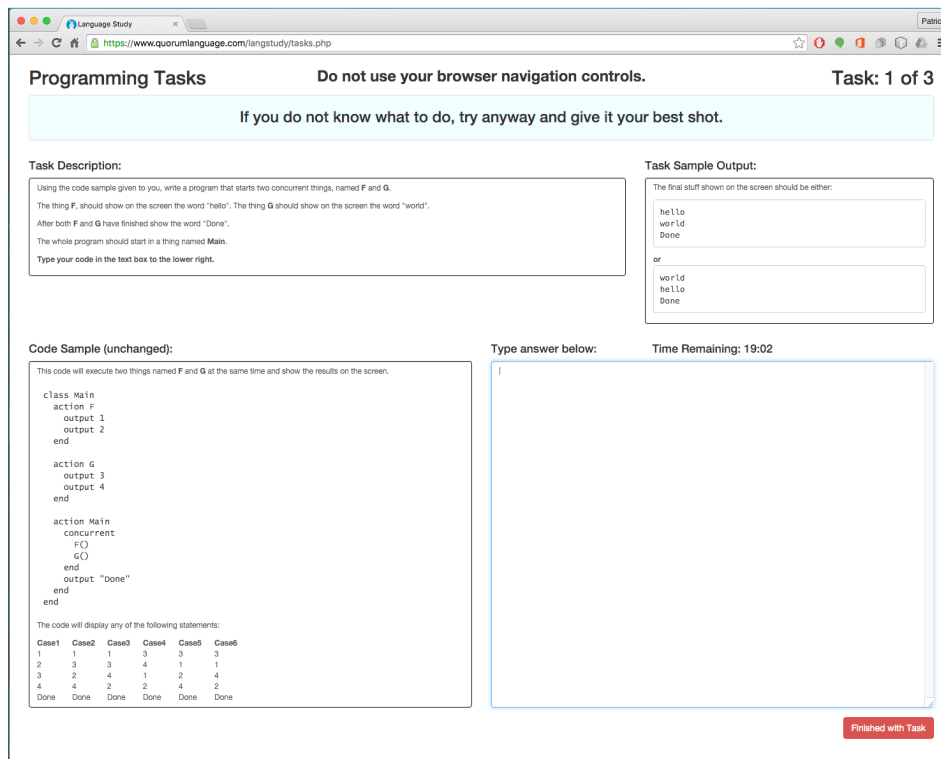
Figure 3.6: Task screen.

The key parts of this page are:

- The top left box is a decription of the task the participant is requested to write a solution for. It contains a detailed descriptions of the components expected in the solution and what they should do.

20

- The top right box shows sample possible outputs for the expected task.

- The code sample box, which is identical to the code sample from the previous screenshot.

- An empty text box, highlighted by default where the user can type their solution.

- A dynamic countdown time showing the time remaining on the task.

- A reminder to do as much of the solution as possible.

- The task status indicator on the top right (Task: 1 of 3).

- A Red submit button to indicate the completion and move on to the next page.

When the Red button is pushed or time expires, the contents of the participants entry box is recorded in the database with an event code indicating task completion. The WebTester also recorded snapshots of the users entries at 10 second intervals with a different code indicating a task in process. These entries will be examined at a later date to determine if there were any interesting patterns in the users responses over time. In total there were 17,786 code events in the database, including the 88 valid participants and the 10 invalid participants.

The participant spends the bulk of their time in the study on these coding pages going through the three successive tasks for the paradigm group they were randomly assigned to. After the third task, they are brought to the final survey page shown in Figure 3.7. The blue section is optional and was used to collect information that was passed to the instructor to give the participant extra credit in their course. The other survey data collected allowed the user to self report their experience level in various programming languages and courses. The age and gender questions were asked on this page after the study instead of before in order not to bias the result of the study based on the reported answer. Since we did not use the information for group classification, it will just be looked at with other independent demographic data to look for correlations.

3.5 Programming Tasks

In this section of the discussion of the experiment, the three tasks are described as well as the rationale for their inclusion in this study. First, the code samples shown to each of the participant groups is presented, followed by a discussion of the thought process used for selecting the particular example. Then, the task description and expected possible outputs, which were the same for both

21

Figure 3.7: Survey screen.

groups, is presented and discussed. Finally, the intended *"correct"* solutions to the task for each the paradigms is shown.

For this experimental design we selected tasks that highlight three core fundamental issues of concurrent programming: i) concurrent execution of objects, ii) producer-consumer problems with concurrent objects and iii) shared memory contention with concurrent objects. This experiment is intended to be the first of several randomized controlled trials of various aspects of concurrency in programming language design and since these issues are addressed at a rudimentary level in most textbooks on the topic, they were selected as the starting point for my planned doctoral dissertation research on concurrency. Our goal in this first experiment is to get a sense of the intuition involved for students under two different paradigmatic approaches to solving these common problems.

3.5.1 Task 1: Two Concurrent Objects

Task 1 begins with the code sample shown in Figure 3.8. It is a rudimentary warm up task designed to show the basic mechanics for a concurrent execution of two objects, which we label **F** and **G** in the code sample. We started with this basic example expecting a very high success rate, but with the intent of highlighting the essential concurrency syntax and program structure for the particular

22

paradigm so that the participants could gain some experience with the format of the paradigm and language before we added more complicated shared memory aspects.

This code will execute two things named **F** and **G** at the same time and show the results on the screen.

(a) CSP.

```
1  class Main
2    action F
3      output 1
4      output 2
5    end
6
7    action G
8      output 3
9      output 4
10   end
11
12   action Main
13     concurrent
14       F()
15       G()
16     end
17     output "Done"
18   end
19 end
```

(b) Threads.

```
1  class F is Thread
2    action Run
3      output 1
4      output 2
5    end
6  end
7
8  class G is Thread
9    action Run
10     output 3
11     output 4
12   end
13 end
14
15 class Main
16   action main
17     F f
18     G g
19     f:Run()
20     g:Run()
21     check
22       f:Join()
23       g:Join()
24     detect e
25     end
26     output "Done"
27   end
28 end
```

The code will display any of the following statements:

1	1	1	3	3	3
2	3	3	4	1	1
3	2	4	1	2	4
4	4	2	2	4	2
Done	Done	Done	Done	Done	Done

Figure 3.8: Task 1 - Code Sample.

This code sample runs two concurrent processes which each print out two statements. The intention is that the participant should recognize that the print order of the two statements within the object will remain in a fixed local ordering, but that the global ordering between the two threads will vary depending on the runtime execution and thus create different printed outputs. Each sample output is shown to highlight this point and reinforce the notion that the local ordering is fixed (i.e. 1 always prints before 2 and 3 always prints before 4), but that the interleaving of the two print statements can vary to every permutation.

Figure 3.9: Task 1 - Description.

The description for Task 1, shown in Figure 3.9, asks the participant to write a program to launch two concurrent object which each print a statement. This task is a slightly simplified version of the code sample (using a single print statement in each concurrent object instead of two). This task was intended to be a warmup task to give the participant practice in setting up the code to execute the two tasks concurrently in the paradigm.

Since the operating segment of the code for each object is trivial and there are no ordering issues, shared resourced or dependencies in this task, any programming errors should be solely due to the structure and requirements of the paradigm or unfamiliar language syntax. To complete this task, the participant can copy and paste the code sample into the entry box and change the print statements. The solution is shown in Figure 3.10.

3.5.2 Task 2: Producer-Consumer

The second problem category we selected for study is the Producer-Consumer problem where multiple concurrent objects are producing or consuming data for/from each other. Our second code sample in Figure 3.11, focuses on showing the participant the infrastructure for running three concurrent objects: two identical objects of the same class that send data (**F**) and a third receiving (**G**) object that receives and prints the data. We do not show the participant specifically how to solve the Producer-Consumer problem in the sample, but we give them the scaffolded code they

24

```
1  class Main
2    action F
3      output "hello"
4    end
5
6    action G
7      output "world"
8    end
9
10   action Main
11     concurrent
12       F()
13       G()
14     end
15     output "Done"
16   end
17 end
```

```
1  class F is Thread
2    action Run
3      output "hello"
4    end
5  end
6
7  class G is Thread
8    action Run
9      output "world"
10   end
11 end
12
13 class Main
14   action Main
15     F t1
16     G t2
17     t1:Run()
18     t2:Run()
19     check
20       t1:Join()
21       t2:Join()
22     detect e
23     end
24     output "Done"
25   end
26 end
```

Figure 3.10: Task 1 - Solutions.

need to create a solution for a dual producer, single consumer problem.

The sample output shows all six possible cases of output for the sample code. The output highlights the fact that the sending threads can run in any order (interleaved or sequential) and the receiver will print the output as the messages are received. Unlike the first sample, the amount of code to execute this in the Threads paradigm is significantly longer than CSP (71 vs. 32 lines). Additionally, the Threads paradigm requires an additional shared object (**N**) to hold the values being passed and which can be locked using the Java-style `synchronized` keyword.

The task description shown in Figure 3.12 lays out the detailed request to create a dual producer, single consumer system controlled by a driver program called **Main**. We deliberately called the objects "things" in the description since the "things" are *methods* in the CSP paradigm and *classes* in the Threads paradigm. We create the complication of incrementing successive values from each producer with the restriction that no values can be skipped. These additions to the code sample require that the shared memory variable used by each producer must be locked after it is generated until it is consumed. An implied constraint is that the consumer must not attempt to consume when there is no data available. Together these requirements and constraints form the invariant for a Producer-Consumer problem.

25

This code will execute three things at the same time, including two copies of **F** and 1 copy of **G** and show the results.

(a) CSP.

```
1  class Main
2    action F(Writer<integer> c, integer x)
3      c:Write(x)
4      c:Write(x)
5    end
6
7    action G(Reader<integer> c1, Reader<
         integer> c2)
8      integer i = 0
9      repeat while i < 4
10       integer x = 0
11       choose
12         x = c1:Read()
13           output x
14       or
15         x = c2:Read()
16           output x
17       end
18       i = i + 1
19     end
20   end
21
22   action Main
23     Channel<integer> c1
24     Channel<integer> c2
25     concurrent
26       F(c1:GetWriter(), 1)
27       F(c2:GetWriter(), 2)
28       G(c1:GetReader(), c2:GetReader())
29     end
30     output "Done"
31   end
32 end
```

(b) Threads.

```
1  class N
2    integer value = 0
3  end
4
5  class F is Thread
6    N n = undefined
7    integer x = 0
8    action Set(N n, integer x)
9      me:n = n
10     me:x = x
11   end
12   action Run()
13     i = 0
14     repeat while i < 2
15       synchronized(N)
16         if n:value = 0
17           n:value = x
18           i = i + 1
19         end
20       end
21     end
22   end
23 end
24
25 class G is Thread
26   N n1 = undefined
27   N n2 = undefined
28   action Set (N n1, N n2)
29     me:n1 = n1
30     me:n2 = n2
31   end
32   action Run()
33     i = 0
34     repeat while i < 4
35       synchronized(N)
36         if n1:value not= 0
37           output n1:value
38           n1:value = 0
39           i = i + 1
40         elseif n2:value not= 0
41           output n2:value
42           n2:value = 0
43           i = i + 1
44         end
45       end
46     end
47   end
48 end
49
50 public class Main
51   action Main
52     F  f1
53     F  f2
54     G  g
55     N  n1
56     N  n2
57     f1:Set(n1, 1)
58     f2:Set(n2, 2)
59     g:Set(n1, n2)
60     f1:Run()
61     f2:Run()
62     g:Run()
63     check
64       f1:Join()
65       f2:Join()
66       g:Join()
67     detect e
68     end
69     output "Done"
70   end
71 end
```

The code will display any of the following statements:

1	1	1	2	2	2
1	2	2	1	1	2
2	1	2	1	2	1
2	2	1	2	1	1
Done	Done	Done	Done	Done	Done

Figure 3.11: Task 2 - Code Sample.

Figure 3.12: Task 2 - Description.

The solutions to this problem in the style of the code samples we offered is shown in Figure 3.13. The Threads paradigm, like the code sample is considerably longer (over twice as many lines of code) and involves the shared object **N** for synchronization. Like the scaffolded code sample, the Producer and Consumer are also *classes* and not *methods*. The differences between the two paradigms are clearly on display in this example.

The CSP paradigm uses a straightforward approach to the driver method **Main**. The solution requires two communication channels, one for each producer to communicate to the single consumer. The two producer and consumer methods are initialized and launched by the driver inside a concurrent block which sets each of the three methods running. In this case, since there is no termination of any of the methods, the producers and consumers will run indefinitely. Each producer gets passed the *Write* end of a channel and the consumer gets passed the *Read* end of both channels it will consume from.

The mechanics of the runtime are simple on the producer side, which starts a counter at zero and enters an infinite loop where it writes the counter value to the channel (where it blocks until the value is removed) and then increments the counter and repeats. The automatic synchronizaton at the channel write (line 5 in the solution code for CSP-3.13a) ensures that the producer does not

(a) CSP.

```
1  class Main
2    action Producer(Writer<integer> c)
3      integer i = 0
4      repeat while true
5        c:Write(i)
6        i = i + 1
7      end
8    end
9
10   action Consumer(Reader<integer> p1,
         Reader<integer> p2)
11     repeat while true
12       integer x = 0
13       choose
14         x = p1:Read()
15           output "Received Producer 1: " +
           x
16       or
17         x = p2:Read()
18           output "Received Producer 2: " +
           x
19       end
20     end
21   end
22
23   action main
24     Channel<integer> c1
25     Channel<integer> c2
26     concurrent
27       Producer(c1:GetWriter())
28       Producer(c2:GetWriter())
29       Consumer(c1:GetReader(), c2:
           GetReader())
30     end
31   end
32 end
```

(b) Threads.

```
1  class N
2    integer n = 0
3  end
4
5  class Producer is Thread
6    N n = undefined
7    action Set(N n)
8      me:n = n
9    end
10   action Run
11     integer i = 0
12     repeat while true
13       synchronized(N)
14         if n:n = 0
15           n:n = i
16           i = i + 1
17         end
18       end
19     end
20   end
21 end
22
23 class Consumer is Thread
24   N n1 = undefined
25   N n2 = undefined
26   action Set (N n1, N n2)
27     me:n1 = n1
28     me:n2 = n2
29   end
30   action Run
31     repeat while true
32       synchronized(N)
33         if n1:n not= 0
34           output "Received p1: " + n1:n
35           n1:n = 0
36         elseif n2:n not= 0
37           output "Received p2: " + n2:n
38           n2:n = 0
39         end
40       end
41     end
42   end
43 end
44
45 class Main
46   action Main
47     N n1
48     N n2
49     Producer p1
50     Producer p2
51     Consumer c
52     p1:Set(n1)
53     p2:Set(n2)
54     c:Set(n1, n2)
55     p1:Run()
56     p2:Run()
57     c:Run()
58     check
59       p1:Join()
60       p2:Join()
61       c:Join()
62     detect e
63     end
64   end
65 end
```

Figure 3.13: Task 2 - Solution.

write to a full consumer and therefore never skips a value. In this solution, there is no guarantee of a service from the consumer or bounded waiting compared to the other thread, but it meets the invariants for a producer in the definition of this problem.

The consumer side of CSP is slightly more complicated, but the structure is similar to the producer. The consumer is passed the *Receive* ends of both channels and enters an infinite loop where it picks an available channel to read from (randomly), assigns the channel value to a memory

location and prints the value to the screen based on which producer it selected. The `choose` block uses this logic to select which channel to read from: i) if only one channel has data, it selects that channel, ii) if both channels have data, it randomly selects a channel, and ii) if no channel has data, it blocks until one receives a message based on the notion of synchronous communication. After the `choose` block, the program repeats. Since the `Read()` method is only invoked when the channel is full, the consumer will never attempt to read from an empty channel, thus fulfilling that aspect of the invariant of the problem.

The Threads model also solves the task requirements, but uses locks (through the `synchronized` command) to implement the reading and writing to ensure no values are lost or that values are not read when empty or written when full. The driver method initializes two **N** objects for the shared memory and then creates two producer objects and one consumer object. Each producer object is initialized with one of the **N** objects and the consumer is initialized with both, similar to the channels in the CSP model. The producer and consumer objects are declared using the `Thread` interface and are required to have a `run` method. After the objects are set up, this method is called. The `check-detect` structure is required when the `join()` methods are called. In our example, the threads run in infinite loops, but the solution contains the call for good form and to ensure the driver program does not terminate while the threads are running.

The producer's run method contains the infinite loop to check if a value needs to be written, and if so, write the next value. The mechanics of this in the Threads paradigm requires the shared object to be locked (with `synchronized`) to prevent the consumer from reading or changing the data while it is operated on by the consumer, so the first step is to wait to aquire the lock (line 13 in the solution code for Threads-3.13b). Once the lock is aquired, the producer can check if it is time to produce another value or not. In our solution, a value of "0" in the shared object indicates that there is nothing for the consumer to consume, so the producer must produce a new value. So, the producer checks the shared object and if it is "0", then it writes the curremt value of its counter variable and increments it. If the value not "0", then it takes not action on the shared object, releases the lock and restarts the loop. This protocol ensures that no values are skipped, since a new value will be produced only once the last value is consumed.

The consumers run method operates under a similar protocol where it first acquires the lock on `class N` (line 32 in the solution code for Threads-3.13b) which ensures that it is the only object in the system with permission to read and write the shared objects. After the lock is acquired, it checks both objects sequentially and if either is full, it prints the value and resets the object's value

to 0 indicating that the producer with that shared object can generate a new value. If the value of the object is still 0, it takes no action on the object. After checking both objects, the consumer releases the lock implicitly when the `synchronized` block ends and the loop is repeated. By locking the object while it works and only reading the value if it is non-zero, the consumer ensures that there is no attempt to read an empty value.

The Thread model also does not guarantee that a particular thread's value is ever printed since the competition for the shared class lock will be repeated at each iteration of the loop by both producers and the consumer. This could be solved by putting a producer thread to sleep after a value is produced and having the consumer signal threads after it read, but this was beyond the scope of the experiment's task.

3.5.3 Task 3: Readers-Writers

The third example involves accessing a shared memory location used as a counter across multiple concurrent objects/processes and is a variation of the Readers-Writers problem. The main concept being examined concerns the different paradigmatic approaches to accessing the shared variable and ensuring mutual exclusion during the reading and modification of its value. Like the Producer-Consumer problem, this problem identifies a common core concurrency issue. Our goal was to test the programming language design implications of the two paradigms.

To introduce the structure required to solve this problem, we used the scaffolded code shown in the code sample in Figure 3.14. The code sample showed a program where two concurrent processes could communicate with a shared value to provide an increasing list of even numbers. The CSP version of the solution shown in Figure 3.14a used two process **A** and **B** and two communication channels to pass a shared value. The sample is designed to demonstrate a concept from the last sample, where the channel blocks when a `Read()` or `Write()` is called on an empty channel. In the sample, process **A** enters its infinite loop with a synchronizing request to read the shared value (line 4). It either finds a value in the channel and proceeds to write (line 5) or it waits until **B** puts a value in the channel. Process **B** is the converse. It begins by writing the shared value to the `request` channel (line 12) and then waiting for a response from process **A** (line 13). Once it has the revised value, it prints it, increments the counter and restarts the loop. Since the channels are sychronizing on the reading and writing, the value is guaranteed to be mutually exclusive. This is a simplistic example with only two threads, but the sample provides the structure required to implement the actual Task shownn in Figure 3.15 involving a third thread competing to access the

30

This code will execute two things named **A** and **B** at the same time and show the results on the screen.

(a) CSP.

```
1  class Main
2    action A(Reader<integer> request, Writer<
         integer> reply)
3      repeat while true
4        integer x = request:Read()
5        reply:Write(x * 2)
6      end
7    end
8
9    action B(Writer<integer> request, Reader<
         integer> reply)
10     integer i = 0
11     repeat while true
12       request:Write(i)
13       integer y = reply:Read()
14       output y
15       i = i + 1
16     end
17   end
18
19   action Main
20     Channel<integer> c1
21     Channel<integer> c2
22     concurrent
23       A(c2:GetReader(), c1:GetWriter())
24       B(c2:GetWriter(), c1:GetReader())
25     end
26   end
27 end
```

(b) Threads.

```
1  class N
2    integer value = 0
3    boolean newItem = false
4  end
5
6  class A is Thread
7    N n = undefined
8    action Set(N n)
9      me:n = n
10   end
11   action Run()
12     repeat while true
13       synchronized(N)
14         if n:newItem = true
15           n:value = n:value * 2
16           n:newItem = false
17         end
18       end
19     end
20   end
21 end
22
23 class B is Thread
24   N n = undefined
25   action Set(N n)
26     me:n = n
27   end
28   action Run()
29     integer i = 1
30     repeat while true
31       synchronized(N)
32         if n:newItem = false
33           integer y = n:value
34           output y
35           n:value = i
36           i = i + 1
37           n:newItem = true
38         end
39       end
40     end
41   end
42 end
43
44 class Main
45   action Main()
46     N n
47     A a
48     B b
49     a:Set(n)
50     b:Set(n)
51     a:Run()
52     b:Run()
53     check
54       a:Join()
55       b:Join()
56     detect e
57     end
58   end
59 end
```

The code will display a list of even numbers starting at 0 forever:
```
0
2
4
6
...
```

Figure 3.14: Task 3 - Code Sample.

counter.

The Thread based protocol is similar to Task 2 with the initialization of a shared variable **N** and two threads competing to increment and multiply the value. There is an additional flag variable

Using the code sample given to you, write a program that synchronizes counting across multiple things. In the thing **Main**, the goal is to use other things that facilitate counting. In this program, **N** contains an integer value that starts at 0. In two copies of the thing named **P**, facilitate the incrementation of the value held by **N**. When the program finishes, the final value in **N** should be equal to 20. Show on the screen the final value of **N**. The following image may help describe what we want you to program in this task. There must be two **P** things and one **N** thing started by **Main**.
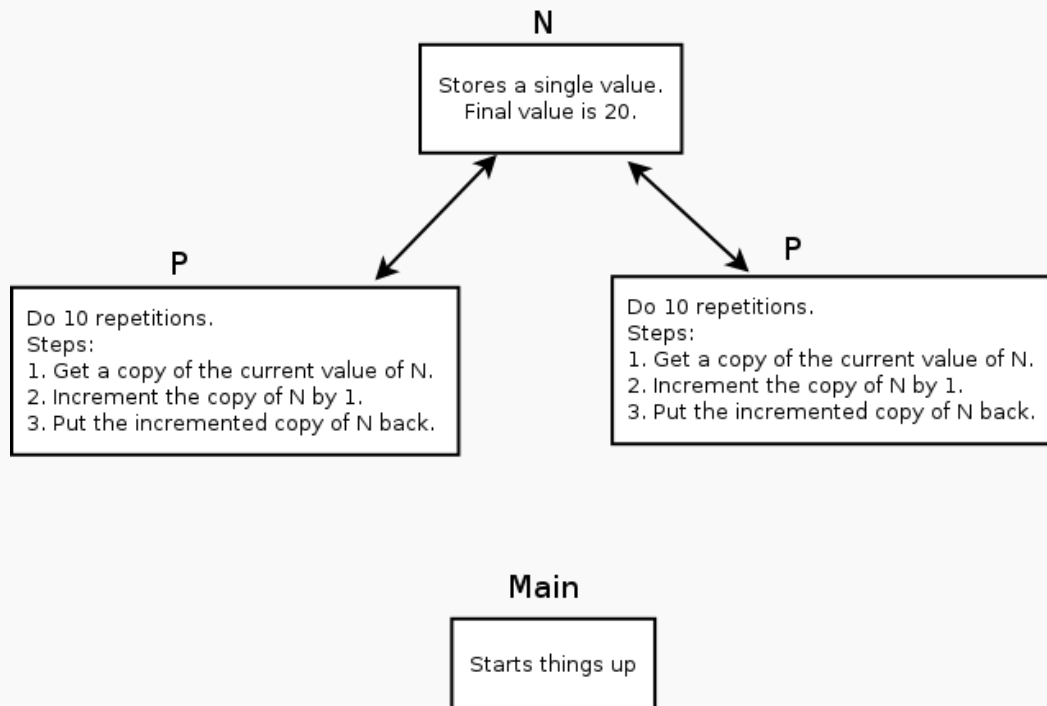
Type your code in the text box to the lower right.

Task Sample Output
The final stuff shown on the screen should be:

    20

Figure 3.15: Task 3 - Task Description.

(line 3) in the **N** object in this case to indicate that the value is newly produced from the counter. The threads are launched in similar fashion in the driver class **Main** with the `check-detect` structure for proper form again, despite the infinite loop. Thread **A** runs a loop where it competes for a lock on the shared object (line 13) and then procees to check if the object contains a new value by examining the `newItem` flag. If the value is new, it updates the value, sets the flag to false, releases the lock and restarts the loop; if not, it just releases the lock and restarts the loop. Thread **B** operates in a similar way, where it competes to acquire the lock (line 31), and then if the flag is

false, it updates the value with the counter value, sets the flag to true and increments the counter. In both cases, mutual exclusion is assured on the shared variable so that only one of the processes can be updating it at a time and the other process waits to do anything until there is a new value. The second condition ensures that the counting is done sequentially and no values are missed.

The formal task description modifies the sample by introducing another process/thread to the problem and asking each to read and write 10 times to the shared variable in any order. The simple example in the sample requires a couple of important changes to demonstrate the understanding of the nuances of this problem. The instructions clearly indicate that the value should be incremented one at a time (10 times by each thread) and the final value must be 20. This constraint retains the requirement for mutual exclusion of the shared variable or each process/thread would read or write duplicate values and the final answer could be something less than 20.

The solutions shown in Figure 3.16 demonstrate that the CSP-based solution has roughly the same number of lines of code as the Threads-based solution (43 compared to 40). The major addition to the CSP-solution is the requirement for two communication channels for each of the processes (one `in` and one `out`), plus an additional request channel for each process to indicate that it is OK to write to the channel. Without this additional channel, mutual exclusion could not be guaranteed. Examining this solution in more detail, we can observe that there are two **P** channels which process the numbers and one **N** channel which accepts the values and writes the final value. The CSP paradigm therefore requires an integration of the Producer-Consumer model to implement this solution.

Each consumer executes its read-write loop 10 times. It begins by informing the producer that it wants to read the next value by writing `true` on the request channel (line 23). Consistent with the basic CSP paradigm for synchronization, the process then waits until the channel is read by **N**. Once this happens, it knows that a new value is about to be produced and it will have the exclusive next value on its inbound channel. It then processes the value and writes the incremented value to the out channel and restarts the loop.

The consumer has to balance the demands of both processes to access the shared variable `n` to ensure mutual exclusion. It executes its main loop until the shared value is 20 (per the task instructions). In the main loop, it randomly selects one of the two request channels if both are full and available and then proceeds to write the current counter value (line 8 or 12) and synchronously waits until an update value is returned (line 9 or 13). If only one channel has data available, it reads that channel and if none are available yet, it blocks on the `choose` in line 6. Once the two

```
1  class Main
2    action N(Reader<integer> in1, Reader<
          integer> in2, Writer<integer> out1,
           Writer<integer> out2, Reader<
          boolean> req1, Reader<boolean> req2
          )
3      integer n=0
4      repeat while n < 20
5        boolean x = false
6        choose
7          x = req1:Read()
8            out1:Write(n)
9            n = in1:Read()
10       or
11         x = req2:Read()
12           out2:Write(n)
13           n = in2:Read()
14       end
15     end
16     output n
17   end
18
19   action P(Writer<integer> out, Reader<
          integer> in, Writer<boolean> req)
20     integer i=0
21     repeat while i<10
22       integer myN
23       req:Write(true)
24       myN = in:Read()
25       myN = myN + 1
26       out:Write(myN)
27     end
28   end
29
30   action Main
31     Channel<integer> c1a
32     Channel<integer> c1b
33     Channel<integer> c2a
34     Channel<integer> c2b
35     Channel<boolean> r1
36     Channel<boolean> r2
37     concurrent
38       N(c1a:GetReader(), c2a:GetReader(),
           c1b:GetWriter(), c2b:GetWriter(),
          r1:GetReader(), r2:GetReader())
39       P(c1a:GetWriter(), c1b:GetReader(),
           r1:GetWriter())
40       P(c2a:GetWriter(), c2b:GetReader(),
           r2:GetWriter())
41     end
42   end
43 end
```

```
1  class N
2    integer value = 0
3  end
4
5  class P is Thread
6    N n = undefined
7    integer i = 0
8    action Set(N n)
9      me:n = n
10   end
11   action Run
12     repeat while i < 10
13       synchronized(n)
14         integer temp = 0
15         temp = n:value
16         temp = temp + 1
17         n:value = temp
18       end
19       i = i + 1
20     end
21   end
22 end
23
24 class Main
25   action main
26     N n
27     P p1
28     P p2
29     p1:Set(n)
30     p2:Set(n)
31     p1:Run()
32     p2:Run()
33     check
34       p1:Join()
35       p2:Join()
36     detect e
37     end
38     output "Final value of N: " +
          n:value
39   end
40 end
```

Figure 3.16: Task 3 - Solution.

processes have incremented the value 20 times, the main loop ends and the final value is output.

The Threads-based solution is similar to the code sample and previous solution with some important distinctions. First of all, it should be noted that there is only a single shared object of class **N** according to the instruction and a reference to the same object is passed to each of

34

the threads. Since it is a single object being shared, `synchronized()` is called on the object itself instead of the class in the previous solution. Although calling it on the class would have the same effect in this example, this version ensures the mutual exclusion for the particular shared copy. The `run()` method of the object is the critical portion of the code in this solution and causes the variable to be read and incremented a total of 10 times, locking the object in each case. For the first time, the `check-detect join` structure is used to ensure the threads synch up before the driver program prints the final value and terminates.

Chapter 4

Analysis Methodology

The scoring model used to evaluate the users responses to these programming tasks is based on the Token Accuracy Map (TAM) approach described by Stefik. [22] The basic concept of the TAM, is to parse a code sample into a sequence of tokens using a lexer, aligning the tokens to a correct solution and then comparing the two token arrays to determine the percentage of correctness in the overall response. In addition to a total overall score, the TAM yields data on the accuracy rate of individual tokens and groups of tokens in the participants' code. Examining the patterns in this data allows us to draw inferences about both the overall impact of the paradigm and specific elements.

4.1 Token Accuracy Maps

I will review the process used to generate a Token Accuracy Map in a step by step example in this section showing details of the various stages of interim work for clarity. Overall, the process has the following steps:

1. Lex tokens into a Token array

2. Run string alignment algorithm

3. Post-processing of Token array

4. Scoring against solution

4.1.1 Step 1: Lexing tokens

The first step in the token accuracy mapping is to lex the user responses into a token array for further processing. Since the tasks were presented in the style of the Quorum Programming Language, I was able to use the open source ANTL [19][20] lexer and Quorum grammer used in the compiler to process the files and extract the tokens. This lexing made it convenient to recognize language keywords and categorize them by type. I had to add the keywords to the grammar discussed earlier based on the parallel paradigm concepts we tested for this experiment, including the words `concurrent`, `choose`, and `synchronized`. I also used the ANTLR channels to conveniently ignore whitespace and comments placed by certain users. The result was an array of `Token` objects (as defined by ANTLR). The key methods of the `Token` class which I used for this analysis were `getText()`, which returns the actual text parsed for the token and `getType()`, which returned an integer designating the type (e.g, an ID, the `class` keyword, etc.).

Figure 4.1 shows an example of the token stream for the solution for Task 1, Language 1 parsed by the Quorum lexer at this stage of the process (with whitespace and comment channels hidden). Note that all identifiers have type 67 and will be treated similarly in the alignment algoritm.

```
 1 class         63
 2 Main          67
 3 action        35
 4 F             67
 5 output        1
 6 "hello"       68
 7 end           62
 8 action        35
 9 G             67
10 output        1
11 "world"       68
12 end           62
13 action        35
14 Main          67
15 concurrent    8
16 F             67
17 (             58
18 )             59
19 G             67
20 (             58
21 )             59
22 end           62
23 output        1
24 "Done"        68
25 end           62
26 end           62
```

Figure 4.1: Sample Token Array.

4.1.2 Step 2: Token Alignment Algorithm

In order to automatically score participant results, I implemented the Needleman-Wunsch (NW) [17] sequence alignment algorithm on the token arrays. The NW algorithm is a dynamic programming algorithm based on the longest common substring algorithm often used in DNA sequencing for global alignment problems.

The implementation of the algorithm involves making an `n X m` array where `n-1` is the number of tokens in the user response array and `m-1` is the number of tokens in the solution token array. The first column and first row of the NW array are reserved for an empty token for matching.

In the NW algorithm, each cell is calculated consecutively by row from `[0][0]` to `[n][m]`. The cell value is computed based on the values of three cells:

- Cell Left (CellLeft)

- Cell Above (CellAbove)

- Cell Above and Left (CellDiag)

The cell value is computed based on the lesser of these values:

- Value of CellLeft - 2

- Value of CellAbove - 2

- Value of CellDiag: +1 if tokens match or -1 if they do not

The concept is that you can arrive at each cell in the matrix in one of three ways:

- by accepting a token from the top and skipping one from the left (CellLeft)

- by accepting a token from the left and skipping one from the top (CellAbove), or

- accepting a token from both strings (CellDiag)

In the algorithm accepting a space has a penalty of -2 for both CellLeft and CellAbove. Accepting both tokens means a score of -1 if they do not match and +1 if they do, indicating that the string is aligning better than if spaces were being inserted from one of the strings. In our example we match just the token type at this stage in order to not penalize a score for things like different variable names.

	◇	class	Main	action	F	integer	a	=	1	output	a	end	action	Main	concurrent	F	(	)	end	end	end
<>	0	-2	-4	-6	-8	-10	-12	-14	-16	-18	-20	-22	-24	-26	-28	-30	-32	-34	-36	-38	-40
class	-2	1	-1	-3	-5	-7	-9	-11	-13	-15	-17	-19	-21	-23	-25	-27	-29	-31	-33	-35	-37
Main	-4	-1	2	0	-2	-4	-6	-8	-10	-12	-14	-16	-18	-20	-22	-24	-26	-28	-30	-32	-34
action	-6	-3	0	3	1	-1	-3	-5	-7	-9	-11	-13	-15	-17	-19	-21	-23	-25	-27	-29	-31
F	-8	-5	-2	1	4	2	0	-2	-4	-6	-8	-10	-12	-14	-16	-18	-20	-22	-24	-26	-28
output	-10	-7	-4	-1	2	3	1	-1	-3	-3	-5	-7	-9	-11	-13	-15	-17	-19	-21	-23	-25
hello	-12	-9	-6	-3	0	1	2	0	-2	-4	-4	-6	-8	-10	-12	-14	-16	-18	-20	-22	-24
end	-14	-11	-8	-5	-2	-1	0	1	-1	-3	-5	-3	-5	-7	-9	-11	-13	-15	-17	-19	-21
action	-16	-13	-10	-7	-4	-3	-2	-1	0	-2	-4	-5	-2	-4	-6	-8	-10	-12	-14	-16	-18
G	-18	-15	-12	-9	-6	-5	-2	-3	-2	-1	-1	-3	-4	-1	-3	-5	-7	-9	-11	-13	-15
output	-20	-17	-14	-11	-8	-7	-4	-3	-4	-1	-2	-2	-4	-3	-2	-4	-6	-8	-10	-12	-14
world	-22	-19	-16	-13	-10	-9	-6	-5	-4	-3	-2	-3	-3	-5	-4	-3	-5	-7	-9	-11	-13
end	-24	-21	-18	-15	-12	-11	-8	-7	-6	-5	-4	-1	-3	-4	-6	-5	-4	-6	-6	-8	-10
action	-26	-23	-20	-17	-14	-13	-10	-9	-8	-7	-6	-3	0	-2	-4	-6	-6	-5	-7	-7	-9
Main	-28	-25	-22	-19	-16	-15	-12	-11	-10	-9	-6	-5	-2	1	-1	-3	-5	-7	-6	-8	-8
concurrent	-30	-27	-24	-21	-18	-17	-14	-13	-12	-11	-8	-7	-4	-1	2	0	-2	-4	-6	-7	-9
F	-32	-29	-26	-23	-20	-19	-16	-15	-14	-13	-10	-9	-6	-3	0	3	1	-1	-3	-5	-7
(	-34	-31	-28	-25	-22	-21	-18	-17	-16	-15	-12	-11	-8	-5	-2	1	4	2	0	-2	-4
)	-36	-33	-30	-27	-24	-23	-20	-19	-18	-17	-14	-13	-10	-7	-4	-1	2	5	3	1	-1
G	-38	-35	-32	-29	-26	-25	-22	-21	-20	-19	-16	-15	-12	-9	-6	-3	0	3	4	2	0
(	-40	-37	-34	-31	-28	-27	-24	-23	-22	-21	-18	-17	-14	-11	-8	-5	-2	1	2	3	1
)	-42	-39	-36	-33	-30	-29	-26	-25	-24	-23	-20	-19	-16	-13	-10	-7	-4	-1	0	1	2
end	-44	-41	-38	-35	-32	-31	-28	-27	-26	-25	-22	-19	-18	-15	-12	-9	-6	-3	0	1	2
end	-46	-43	-40	-37	-34	-33	-30	-29	-28	-27	-24	-21	-20	-17	-14	-11	-8	-5	-2	1	2
end	-48	-45	-42	-39	-36	-35	-32	-31	-30	-29	-26	-23	-22	-19	-16	-13	-10	-7	-4	-1	2

Figure 4.2: Needleman-Wunsch Sample Table.

After a cell value is calculated, a backpointer value is filled in for that cell indicating which cell was selected to arrive at that cell. The algorithm continues, row by row until the last cell if filled. To construct the optimal sequence alignment, we take the maximum cell, `table[n][m]`, and look at the backpointer cell and calculate the row and column offset to determine its position relative to the current cell. We construct two new arrays of aligned tokens matching each backpointer cell in succession until we reach `table[0][0]`. For diagonal cells, we take a token from each source array and add them to the aligned arrays. If the previous cell was either from the left or above, we add the appropriate token to the aligned array and insert a null token in the other representing a space.

Figure 4.2 shows a sample alignment after the NW algorithm has run. The tokens in the first row are from the solution string and tokens in first column are from the comparison string. The red bordered cells indicate the path followed by the backtrace to generate the optimal alignment. The filled cells indicate the token *types* match. Movement vertically on the alignment path indicates that spaces are inserted in the comparison string. This ensures that when the strings are aligned, they will have an equal number of tokens.

Figure 4.3 shows the contents of the token array after an alignment with the NW algorithm. Null tokens (spaces) are represented by a "-" for the text and 0 for the token type. A close inspection of this sample alignment reveals a common misalignment that occurs because of the construction of the optimal string from bottom to top. The `F()` function call is matched to the second `G()` function call instead of the first one. This issue is addressed in the Post Processing step described next.

```
 1 | class       63          class       63
 2 | Main        67          Main        67
 3 | action      35          action      35
 4 | F           67          F           67
 5 | -           0           output      1
 6 | integer     37          "hello"     68
 7 | a           67          end         62
 8 | =           46          action      35
 9 | 1           65          G           67
10 | output      1           output      1
11 | a           67          "world"     68
12 | end         62          end         62
13 | action      35          action      35
14 | Main        67          Main        67
15 | concurrent  8           concurrent  8
16 | -           0           F           67
17 | -           0           (           58
18 | -           0           )           59
19 | F           67          G           67
20 | (           58          (           58
21 | )           59          )           59
22 | end         62          end         62
23 | end         62          end         62
24 | end         62          end         62
```

Figure 4.3: Sample Alignment.

The source code for my implementation of the Needlemman-Wunsch algorithm is shown in Figure 4.4. The source code for the backtrace on the table returned from the Needlemman-Wunsch `CalcTable()` method is shown in Figure 4.5

40

```
 1  private Token[] solutionTokens;  /* set by constructor */
 2  private Token[] valueTokens;
 3
 4  private class Cell {
 5      int value;
 6      int row;
 7      int col;
 8      Cell prev;
 9      Cell (int row, int col) {
10          this.row = row;
11          this.col = col;
12          this.value = 0;
13          this.prev = null;
14      }
15  }
16
17  public Cell[][] CalcTable() {
18      Cell[][] table = CreateTable();
19      int rows = table.length;
20      int cols = table[0].length;
21      for (int r = 1; r < rows; r++) {
22        for (int c = 1; c < cols; c++) {
23          CalcCell(table, r, c);
24        }
25      }
26      return table;
27  }
28
29  private Cell[][] CreateTable() {
30      int cols = solutionTokens.length + 1;
31      int rows = valueTokens.length + 1;
32      Cell[][] table = new Cell[rows][cols];
33      for (int r = 0; r < rows; r++) {
34          for (int c = 0; c < cols; c++) {
35              Cell entry = new Cell(r,c);
36              if (r == 0 && c == 0) {
37                  entry.value = 0;
38              } else if (r == 0) {
39                  entry.prev = table[0][c-1];
40                  entry.value = entry.prev.value - 2;
41              } else if (c == 0) {
42                  entry.prev = table[r-1][0];
43                  entry.value = entry.prev.value - 2;
44              }
45              table[r][c] = entry;
46          }
47      }
48      return table;
49  }
50
51  private void CalcCell(Cell[][] table, int row, int col) {
52      Cell cell = table[row][col];
53      Cell cellAbove = table[cell.row-1][cell.col];
54      Cell cellLeft = table[cell.row][cell.col-1];
55      Cell cellDiag = table[cell.row-1][cell.col-1];
56      int scoreFromAbove = cellAbove.value - 2;
57      int scoreFromLeft = cellLeft.value - 2;
58      int scoreFromDiag = cellDiag.value;
59      if (solutionTokens[cell.col-1].getType() == valueTokens[cell.row-1].getType()) {
60          scoreFromDiag += 1;
61      } else {
62          scoreFromDiag -= 1;
63      }
64      if (scoreFromAbove >= scoreFromLeft) {
65          if (scoreFromDiag >= scoreFromAbove) {
66              cell.value = scoreFromDiag;
67              cell.prev = cellDiag;
68          } else {
69              cell.value = scoreFromAbove;
70              cell.prev = cellAbove;
71          }
72      } else {
73          if (scoreFromDiag >= scoreFromLeft) {
74              cell.value = scoreFromDiag;
75              cell.prev = cellDiag;
76          } else {
77              cell.value = scoreFromLeft;
78              cell.prev = cellLeft;
79          }
80      }
81  }
```

Figure 4.4: Needleman-Wunsch Source Code - Java.

41

```java
private Token[][] Backtrace(Cell[][] table) {
    List<Token> solutionTokensAligned = new ArrayList<>();
    List<Token> valueTokensAligned = new ArrayList<>();

    Cell current = table[table.length-1][table[0].length-1];
    while(current.prev != null) {
        if (current.row - current.prev.row == 1) {
            valueTokensAligned.add(0, valueTokens[current.row-1]);
        } else {
            valueTokensAligned.add(0, null);
        }
        if (current.col - current.prev.col == 1) {
            solutionTokensAligned.add(0, solutionTokens[current.col-1]);
        } else {
            solutionTokensAligned.add(0, null);
        }
        current = current.prev;
    }

    int n = solutionTokensAligned.size();
    Token[] solution = solutionTokensAligned.toArray(new Token[n]);
    Token[] value = valueTokensAligned.toArray(new Token[n]);
    Token[][] result = new Token[2][n];
    result[0] = solution;
    result[1] = value;
    return result;
}
```

Figure 4.5: Backtrace Source Code - Java.

4.1.3 Step 3: Post Processing

One phase of post processing on the array was done to align tokens by type from the first occurence of the token type in the token array. Because the optimal alignment in the NW algorithm was constructed from a backtrace, tokens are matched first from the end of the string, which caused some odd matchings. The overall score is not affected by the post processing step, but the alignment is improved. Essentially all that is done during the step is to move the token to iterate over the list of aligned tokens and move any matches across any null regions so that the null token is aligned at the latest point.

A graphical illustration of the fix that is applied by this step is shown in Figure 4.6. In the "Before" alignment on the left the highlighted area shows the misalignment from the raw algorithm. On the right, the yellow highlighted area is moved up by swapped row by row with the null tokens in the green section to accomplish the improved alignment.

The alignments can probably be improved further from looking at better brace matching or class/action matching. In a visual inspection of various results there were several examples found that will be explored further. The net result of a manual realignment of the tokens to account for these difference had only a minor effect on the outcomes, however. The total overall score would

42

Before Post Processing

				After Post Processing			
class	63	class	63	class	63	class	63
Main	67	Main	67	Main	67	Main	67
action	35	action	35	action	35	action	35
F	67	F	67	F	67	F	67
-	0	output	1	-	0	output	1
integer	37	hello	68	integer	37	hello	68
a	67	end	62	a	67	end	62
=	46	action	35	=	46	action	35
1	65	G	67	1	65	G	67
output	1	output	1	output	1	output	1
a	67	world	68	a	67	world	68
end	62	end	62	end	62	end	62
action	35	action	35	action	35	action	35
Main	67	Main	67	Main	67	Main	67
concurrent	8	concurrent	8	concurrent	8	concurrent	8
-	0	F	67	F	67	F	67
-	0	(	58	(	58	(	58
-	0	)	59	)	59	)	59
F	67	G	67	-	0	G	67
(	58	(	58	-	0	(	58
)	59	)	59	-	0	)	59
end	62	end	62	end	62	end	62
end	62	end	62	end	62	end	62
end	62	end	62	end	62	end	62

Figure 4.6: Post Processing Example.

not increase from this manual matching and any adjustments overall tended to be minor, so the post processing pass was ignored for now. The biggest impact would likely be on the specific token accuracy mapping score of individual tokens that may not be matched properly. Key words and major uncommon token types tended to naturally match though, so the biggest impact were on tokens like parentheses and end statements which were not deemed as important to examine.

4.1.4 Step 4: Scoring

The overall scoring and individual token map can now be assembled from the individual scores. A token accuracy score array of `int[]` is initialized to 0 in the same size as the solution token array. Each individual token type is then compared to the participant's response and the accuracy score array is set to 1 in that position if the types match. The overall score is calculated as the sum of all the values in the array divided by the total number of correct tokens. Each participant response is processed and the combined map is produced. The results can also be filtered to provide token maps for different demographic gropus if desired.

This approach ignores superfluous or inaccurate tokens in the user response that are matched

with spaces in the solution array. Additional analysis could be done on this extra data to determine if there were patterns in occurrence of unmatched user input, but it was not performed for this scoring algorithm. Token accuracy is measured simply as whether the user correctly placed the right token type in the properly matched position of the solution. Figure 4.7 shows an example alignment. The final column shows the score by token.

```
 1 | class     63          class     63          1
 2 | F         67          F         67          1
 3 | is        20          is        20          1
 4 | Thread    67          Thread    67          1
 5 | action    35          action    35          1
 6 | Run       67          Run       67          1
 7 | output    1           output    1           1
 8 | "hello"   68          "Hello"   68          1
 9 | end       62          end       62          1
10 | end       62          end       62          1
11 | class     63          class     63          1
12 | G         67          G         67          1
13 | is        20          is        20          1
14 | Thread    67          Thread    67          1
15 | action    35          action    35          1
16 | Run       67          Run       67          1
17 | output    1           output    1           1
18 | "world"   68          "World"   68          1
19 | end       62          end       62          1
20 | end       62          end       62          1
21 | class     63          class     63          1
22 | Main      67          Main      67          1
23 | action    35          action    35          1
24 | Main      67          main      67          1
25 | F         67          F         67          1
26 | t1        67          f         67          1
27 | G         67          G         67          1
28 | t2        67          g         67          1
29 | t1        67          f         67          1
30 | :         36          :         36          1
31 | Run       67          Run       67          1
32 | (         58          (         58          1
33 | )         59          )         59          1
34 | t2        67          g         67          1
35 | :         36          :         36          1
36 | Run       67          Run       67          1
37 | (         58          (         58          1
38 | )         59          )         59          1
39 | check     16          -         0
40 | t1        67          -         0
41 | :         36          -         0
42 | Join      67          -         0
43 | (         58          -         0
44 | )         59          -         0
45 | t2        67          -         0
46 | :         36          -         0
47 | Join      67          -         0
48 | (         58          -         0
49 | )         59          -         0
50 | detect    14          -         0
51 | e         67          -         0
52 | end       62          -         0
53 | output    1           output    1           1
54 | "Done"    68          "Done"    68          1
55 | end       62          end       62          1
56 | end       62          end       62          1
```

Figure 4.7: Scored Alignment.

4.2   Final Result

The final result of this processing and analysis is a Token Accuracy Map as shown in Figure 4.8. The tokens are place in position followed by a number in parenthesis that represents the percentage

44

correct reponse for that token by all users combined in that slicing of the group. The automated scoring and tracking enables easy construction and examination of TAMs by demographic. For example we could easily prepare TAMs for a task by year in school or native language.

```
1  class (90.91) Main (100.00)
2    action (93.18) F (100.00)
3      output (90.91) "hello" (75.00)
4    end (90.91)
5
6    action (95.45) G (100.00)
7      output (90.91) "world" (79.55)
8    end (93.18)
9
10   action (90.91) Main (100.00)
11     concurrent (90.91)
12       F (97.73) ( (95.45) ) (97.73)
13       G (100.00) ( (97.73) ) (95.45)
14     end (90.91)
15     output (88.64) "Done" (90.91)
16   end (88.64)
17
18 end (90.91)
```

Figure 4.8: Complete Token Accuracy Map.

Chapter 5

Results

5.1 Study Participants

The participants for the study were all recruited from various classes offered by the Department of Computer Science at the University of Nevada, Las Vegas. The classes ranged from 200, 300 and 400 level undergraduate to 600 and 700 level graduate courses. Table 5.1 shows a breakdown of the participants with valid responses by position in the academic pipeline for each paradigm group.

Grade	Threads	CSP	Total
Freshman	1	1	2
Sophomore	5	5	10
Junior	13	16	29
Senior	17	17	34
Graduate	5	4	9
Post-Graduate	2	1	3
Non-Degree	1	0	1
Total	44	44	88

Table 5.1: Participants by Level in Academic Pipeline.

5.2 Data Recorded

The data from the experiment came directly from the automated testing system described earlier. There were 98 total participants in the database upon conclusion of the data gathering; 88 of these participants were deemed valid. Overall, the results fell into 4 categories, as shown in table 5.2:

Category	Participants	Responses
Full Data	78	234
Partial Data	10	18
Missing Event 8	-	6
Total	88	258

Table 5.2: Participant Breakdown.

1. **Full Data**: Participants in this category had a complete set of responses to all three tasks including a final code submission from pressing the "Submit" button on the Task page of the application. The final code submission from this action results in a special entry in the events table of the database with a code of '8'. There were 78 participants in this category with a total of 234 affirmative responses.

2. **Partial Data**: Participants in this category had full and final submitted responses for at least 1 of the tasks, but incomplete results on some other tasks. There were 10 total participants, 8 of whom completed 2 tasks and 2 of whom completed 1 task for a total of 18 responses.

3. **Missing Event 8**: Participants in this category had responses to all of the tasks, but for some reason, one of the tasks was missing an event '8'. The timestamped 10 second tracking events (coded as '7') in these cases appeared to be final solutions and the code sample for last timestamped '7' event was used. The cause for the missing '8' is undetermined, but I speculate that it was a malfunction when the time allotment expired instead of the user affirmatively hitting the "Submit" button. There were 6 participants in this category (counted in the Partial Data participants previously) where one of their three solutions was a complete looking '7'. Of those 6 participants, 4 were missing '8' events on the first 2 tasks and 2 were missing the final task. In those cases, the participants spent 21 and 16 minutes working on the final task.

4. **Invalid Data**: Participants in this category were disqualified completely for various reasons. Some of the data were clearly junk responses such as an email address `iamnotreal@...`, non-code responses to tasks, and silly answers to clas-

47

sification questions clearly indicating the participant was not intending to take the study. This also includes several situations where the participant gave up during the first task or made no code response at all. It includes two additional cases where the participant must have navigated back and started the tasks over because there were more than one final '8' events separated by longer periods of time. There were a total of 10 invalid responses.

In addition to this final data, the test administration program captured snapshots of the users input text area in 10 second intervals during the time each task was active. In total, there are 17,786 code snapshots in the database which can be analyzed using the token accuracy mapping algorithm. For this thesis, we looked at the 258 final code event submissions only and will examine the 17,528 at a later date.

5.3 Student Course Level

The participants self reported their educational level in the classification portion of the study and this level was matched and confirmed by their enrollment in the particular class which they were recruited through. This classification level gave us classification information as to their relative experience level and the programming skills they had been previously taught. Our detailed knowledge of the curriculum taught by the Department allowed us to classify students according to their relative level of skills based on the classes they were currently enrolled in as well as the prerequisite courses they would have taken previously. The knowledge level of general issues concerning parallel computing differed widely from the sophomore level to graduate students. Table 5.3 describes the parallel material generally taught to students in the standard curriculum.

The number of participants by each course level is shown in Table 5.4. The prerequisite track at UNLV requires that the courses are taken in order, although there are certain overlaps allowed, such as the ability to take CS 326 and CS 370 or CS 460 and CS 472 at the same time. For purposes of classification in this table, the higher course number was used. Five (5) participants did not list a course number on their profile and are listed as "N/A" in the table.

5.4 Overall Scores by Group

The overall scores by group (concurrency paradigm) and task are shown in Table 5.5 along with the number of code submissions (N), the minimum score (Min), the maximum score (Max) and

Number	Course Name	Parallel Material Taught
CS 202	Computer Science II	Second C++ course. No previous exposure to parallel concepts in the curriculum.
CS 218	Intro to Systems Programming	Assembly language programming. One assignment with a threaded program accessing shared memory.
CS 326	Programming Languages	Elements of programming languages, details of concurrent threading implementation in JVM.
CS 370	Operating Systems	OS level task scheduling and deadlocks with one project using `pthreads`.
CS 460	Compiler Construction	Java compiler construction with no specific threading material.
CS 472	Software Engineering	Capstone course to integrate knowlege in team based development, no specific multithreading.
CS 789	Parallel Programming	Graduate course focused on parallel programming primarily concerned with message passing projects.

Table 5.3: Course Knowledge.

Course	Number
CS 202	11
CS 218	11
CS 326	29
CS 370	6
CS 460	0
CS 472	18
CS 789	8
N/A	5
Total	88

Table 5.4: Participants by Course.

the standard deviation of scores (SD). The results are shown graphically in Figure 5.1 with 95% confidence intervals. The CSP students performed better on Task 1 (92.9% to 89.2%), about the same on Task 2 (75.9% to 76.3%) and worse on Task 3 (43.6% to 79.1%). The standard deviations

49

Group	Task	N	Score	Min	Max	SD
CSP	1	44	92.9	30.8	100.0	15.4
CSP	2	44	75.9	3.2	96.0	24.3
CSP	3	44	43.6	3.0	67.2	13.0
All CSP		132	70.8	3.0	100.0	27.3
Thread	1	44	89.2	21.4	100.0	20.2
Thread	2	42	76.3	15.6	99.1	25.8
Thread	3	40	79.1	32.3	96.8	12.0
All Thread		126	81.7	15.6	100.0	20.9

Table 5.5: Score By Group and Task.

were similar for each task with the largest difference on Task 1 (15.4 for CSP and 20.2 for Threads). The graph shows the falloff for Task 3 in the CSP group compared to Task 2, while the Threads group performed about the same.

To analyze the data, I applied a Repeated Measures ANOVA using the statistical package SPSS. The model used the three overall task accuracy scores for within-subjects factors as independent variables, reflecting the sequence given to the participants. Two controlled between-subjects factors were examined together: Paradigm Group and Level in School. Additionally, I examined Gender, and Native Language as random factors.

Since ANOVA's with repeated measures assume sphericity of data, I used Mauchly's Test of Sphericity to check if the assumption was violated. Sphericity assumes that the variances between all combinations of groups are equal. We test the null hypothesis (p < .05) that the variances are equal. In this data, the test for sphericity indicates that the assumption of sphericity has been violated ($\chi^2(2) = 8.718, p = .013$), so we reject the null hypothesis that variances are equal. To correct for this sphericity, I examine the Greenhouse-Geisser score which corrects the degrees of freedom for the F-distribution and provide a valid critical F-Value. SPSS applies this adjustment automatically.

The results for the Within-Subject Effects of the ANOVA with repeated measures shows that Task was a significant factor in overall score F(1.733, 53) = 25.094, p = .000, ($\eta^2_{partial} = .321$). There was a significant interaction between Tasks and Group in the results F(1.733, 53) = 14.730, p = .000, ($\eta^2_{partial} = .217$) indicating we should reject the null hypothesis HO-1 that the groups are equivalent. The interaction between Task and School Level was approaching significance F(10.395,

50
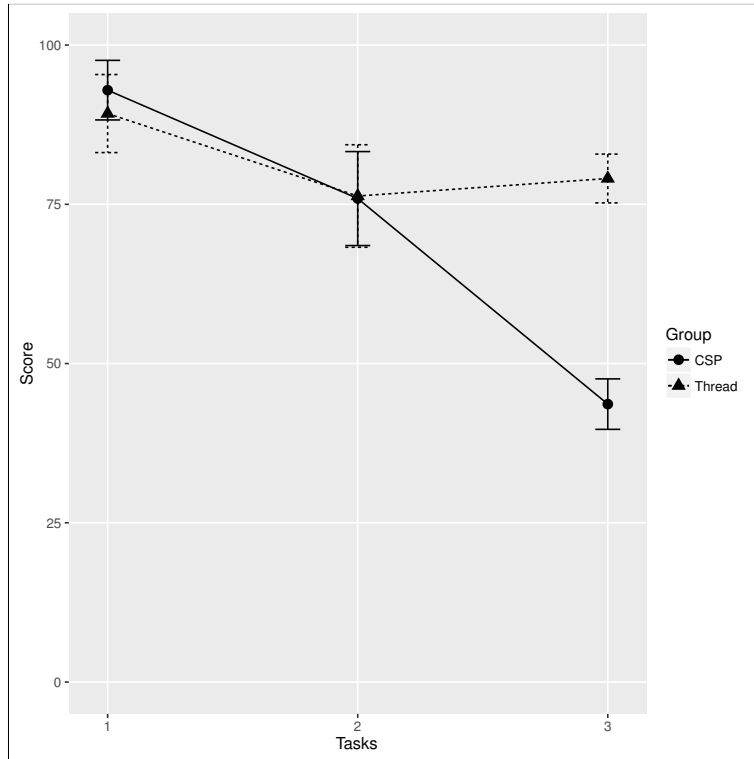
Figure 5.1: Scores by Group.

53) $= 1.846$, p $= .061$, $(\eta^2_{partial} = .173)$ but does not meet our threshold of .050. All other interactions up to 4 ways were checked and no other within-subjects effects were significant.

Figure 5.6 shows the results of the Between-Subjects Effects tests of the Repeated Measures ANOVA. Both Group F$(1, 53) = 10.022$, p $= .003$ $(\eta^2_{partial} = .159)$ and Native Language F$(1, 53) = 5.618$, p $= .021$, $(\eta^2_{partial} = .096)$ were significant, however all other interactions are not statistically significant, although there is not enough power to detect this in our model.

Factor	F	Sig.	df	$\eta^2_{partial}$
Group	10.022	.003	1	.159
SchoolLevel	1.352	.251	6	.133
Gender	.539	.587	2	.020
Language	5.618	.021	1	.096

Table 5.6: Between Subjects Effects.

5.5   Level in School

The significance level for the interaction of Task by Level in School was approaching significance, but it was not a strong enough pattern. The lower number of responses on either end of the ordering is likely a factor in this. The base result is somewhat contradictory to our lab's recent Lambda paper conducted by Uesbeck, [23] but this sample is more constrained in experience levels, so one would expect this interaction to matter less. Table 5.7 and Figure 5.2 depict the results.

	CSP		Threads	
	N	Mean	N	Mean
Non-degree	3	69.24	-	-
Freshman	3	49.69	3	84.04
Sophomore	15	69.42	15	66.13
Junior	39	69.79	44	81.18
Senior	51	71.62	49	85.81
Graduate	15	73.08	12	83.16
Post-graduate	6	79.89	3	91.77

Table 5.7: Score By Academic Level.
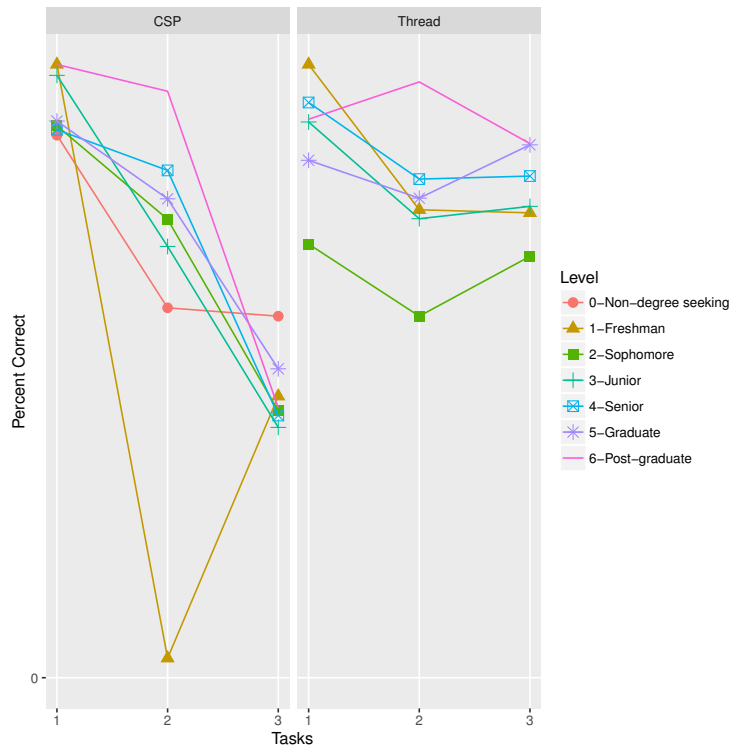
Figure 5.2: Overall Scores by Participant - Task 3.

5.6   Score Graphs by Participant

The graphs in Figures 5.3, 5.4 and 5.5 show the overall scores by participant (sorted from highest to lowest score) for each task. These graphs are intended to give an overall sense for the performance of students in each group, as well as the rate of falloff in terms of the number of students who scored at a certain rate. The results in both the first two tasks show a modest better performance for CSP compared to Threads once students begin to show incorrect tokens in their answers. Task 2 also shows a more gradual falloff in accuracy for both tasks. Task 3 shows a consistent gap between students at each level. These results will be examined more closely in the Discussion chapter where the token accuracy maps are used to identify the specific parts of the CSP answer that were not well understood by participants and were commonly answered incorrectly.
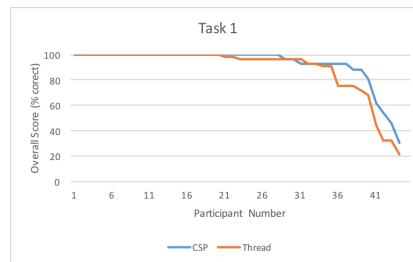

Figure 5.3: Overall Scores by Participant - Task 1.
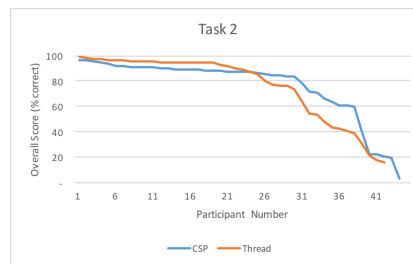
Figure 5.4: Overall Scores by Participant - Task 2.
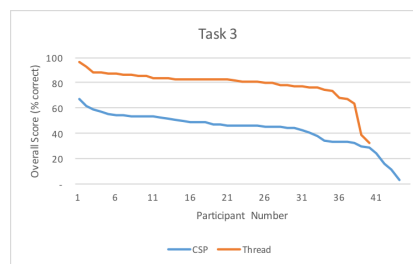
Figure 5.5: Overall Scores by Participant - Task 3.

53

5.7 Final Token Accuracy Maps

Following are the token accuracy maps with scores by token for each of the tasks and groups.

5.7.1 Task 1

```
1  class (90.91) Main (100.00)
2    action (93.18) F (100.00)
3      output (90.91) "hello" (75.00)
4    end (90.91)
5
6    action (95.45) G (100.00)
7      output (90.91) "world" (79.55)
8    end (93.18)
9
10   action (90.91) Main (100.00)
11     concurrent (90.91)
12       F (97.73) ( (95.45) ) (97.73)
13       G (100.00) ( (97.73) ) (95.45)
14     end (90.91)
15     output (88.64) "Done" (90.91)
16   end (88.64)
17
18 end (90.91)
```

Figure 5.6: Token Accuracy Map - CSP Task 1.

```
1  class (93.18) F (100.00) is (88.64) Thread (90.91)
2    action (95.45) Run (97.73)
3      output (95.45) "hello" (72.73)
4    end (86.36)
5
6  end (93.18)
7  class (86.36) G (95.45) is (88.64) Thread (90.91)
8    action (93.18) Run (93.18)
9      output (95.45) "world" (72.73)
10   end (86.36)
11
12 end (93.18)
13 class (86.36) Main (95.45)
14   action (93.18) Main (93.18)
15     F (93.18) t1 (90.91)
16     G (90.91) t2 (90.91)
17     t1 (90.91) : (88.64) Run (93.18) ( (93.18) ) (93.18)
18     t2 (95.45) : (88.64) Run (95.45) ( (90.91) ) (90.91)
19     check (77.27)
20       t1 (81.82) : (79.55) Join (84.09) ( (84.09) ) (84.09)
21       t2 (86.36) : (77.27) Join (84.09) ( (86.36) ) (86.36)
22     detect (77.27) e (79.55)
23     end (79.55)
24     output (95.45) "Done" (100.00)
25   end (95.45)
26
27 end (95.45)
```

Figure 5.7: Token Accuracy Map - Threads Task 1.

5.7.2 Task 2

Task 2 represented the one-way producer-consumer task with two producers and one consumer. In both languages the areas where the participants seems to consisently show the most difficulty was with the parallelism constructs. These can be seen by the score of 63.64% on the `choose` keyword in the CSP task in line 13 in Figure 5.8 and the score of 64.29% on the `synchronized` keyword in the Threads task in line 13 in Figure 5.9. In both of these languages participants also had low scores for the `repeat` loops in lines 4 (38.64%) and 11 (54.44%) in Figure 5.8 and lines 12 (61.90%) and 32 (64.29%) in Figure 5.9.

```
1   class (86.36)
2     Main (100.00) action (88.64) Producer (95.45) ( (88.64) Writer (90.91) < (86.36)
          integer (81.82) > (84.09) c (88.64) ) (40.91)
3       integer (77.27) i (88.64) = (40.91) 0 (43.18)
4       repeat (38.64) while (36.36) true (2.27)
5         c (88.64) : (79.55) Write (84.09) ( (86.36) i (90.91) ) (84.09)
6         i (81.82) = (36.36) i (81.82) + (29.55) 1 (31.82)
7       end (20.45)
8     end (81.82)
9
10    action (84.09) Consumer (93.18) ( (88.64) Reader (88.64) < (84.09) integer (79.55) >
          (77.27) p1 (88.64) , (79.55) Reader (88.64) < (77.27) integer (77.27) > (77.27) p2
          (84.09) ) (79.55)
11      repeat (54.55) while (56.82) true (4.55)
12        integer (65.91) x (79.55) = (70.45) 0 (81.82)
13        choose (63.64)
14          x (84.09) = (75.00) p1 (77.27) : (77.27) Read (88.64) ( (77.27) ) (77.27)
15            output (75.00) "Received Producer 1: " (45.45) + (13.64) x (75.00)
16        or (61.36)
17          x (86.36) = (79.55) p2 (84.09) : (79.55) Read (90.91) ( (81.82) ) (88.64)
18            output (77.27) "Received Producer 2: " (50.00) + (22.73) x (81.82)
19        end (31.82)
20      end (75.00)
21    end (84.09)
22
23    action (84.09) main (86.36)
24      Channel (86.36) < (86.36) integer (81.82) > (84.09) c1 (86.36)
25      Channel (86.36) < (86.36) integer (79.55) > (81.82) c2 (86.36)
26      concurrent (81.82)
27        Producer (88.64) ( (86.36) c1 (84.09) : (79.55) GetWriter (86.36) ( (79.55) )
          (84.09) ) (79.55)
28        Producer (86.36) ( (88.64) c2 (90.91) : (81.82) GetWriter (88.64) ( (84.09) )
          (88.64) ) (81.82)
29        Consumer (88.64) ( (84.09) c1 (88.64) : (79.55) GetReader (88.64) ( (86.36) )
          (81.82) , (79.55) c2 (88.64) : (77.27) GetReader (90.91) ( (75.00) ) (77.27) )
          (79.55)
30      end (65.91)
31    end (86.36)
32
33  end (86.36)
```

Figure 5.8: Token Accuracy Map - CSP Task 2.

```
 1 class (95.24) N (100.00)
 2   integer (92.86) n (97.62) = (92.86) 0 (92.86)
 3 end (83.33)
 4 class (83.33) Producer (90.48) is (78.57) Thread (90.48)
 5   N (85.71) n (83.33) = (88.10) undefined (80.95)
 6   action (76.19) Set (83.33) ( (78.57) N (73.81) n (73.81) ) (78.57)
 7     me (73.81) : (73.81) n (83.33) = (80.95) n (78.57)
 8   end (71.43)
 9
10   action (76.19) Run (78.57)
11     integer (11.90) i (78.57) = (73.81) 0 (71.43)
12     repeat (61.90) while (59.52) true (4.76)
13       synchronized (64.29) ( (80.95) N (80.95) ) (76.19)
14         if (61.90) n (78.57) : (66.67) n (78.57) = (71.43) 0 (71.43)
15           n (76.19) : (69.05) n (80.95) = (66.67) i (64.29)
16           i (73.81) = (71.43) i (78.57) + (83.33) 1 (76.19)
17         end (59.52)
18       end (66.67)
19     end (71.43)
20   end (78.57)
21
22 end (88.10)
23 class (85.71) Consumer (88.10) is (78.57) Thread (85.71)
24   N (80.95) n1 (76.19) = (78.57) undefined (78.57)
25   N (73.81) n2 (78.57) = (78.57) undefined (73.81)
26   action (97.62) Set (97.62) ( (97.62) N (90.48) n1 (90.48) , (83.33) N (73.81) n2
        (88.10) ) (100.00)
27     me (88.10) : (88.10) n1 (92.86) = (90.48) n1 (92.86)
28     me (83.33) : (85.71) n2 (88.10) = (85.71) n2 (83.33)
29   end (83.33)
30
31   action (88.10) Run (88.10)
32     repeat (64.29) while (64.29) true (14.29)
33       synchronized (66.67) ( (88.10) N (71.43) ) (88.10)
34         if (66.67) n1 (83.33) : (69.05) n (80.95) not= (59.52) 0 (76.19)
35           output (71.43) "Received p1: " (35.71) + (19.05) n1 (71.43) : (64.29) n (66.67)
36           n1 (73.81) : (38.10) n (78.57) = (47.62) 0 (76.19)
37         elseif (61.90) n2 (85.71) : (83.33) n (90.48) not= (64.29) 0 (66.67)
38           output (78.57) "Received p2: " (40.48) + (14.29) n2 (83.33) : (78.57) n (85.71)
39           n2 (78.57) : (33.33) n (80.95) = (28.57) 0 (69.05)
40         end (69.05)
41       end (76.19)
42     end (78.57)
43   end (85.71)
44
45 end (69.05)
46 class (85.71) Main (88.10)
47   action (78.57) Main (83.33)
48     N (83.33) n1 (85.71)
49     N (83.33) n2 (83.33)
50     Producer (88.10) p1 (80.95)
51     Producer (80.95) p2 (66.67)
52     Consumer (69.05) c (69.05)
53     p1 (76.19) : (80.95) Set (85.71) ( (83.33) n1 (83.33) ) (69.05)
54     p2 (71.43) : (66.67) Set (73.81) ( (69.05) n2 (66.67) ) (78.57)
55     c (80.95) : (78.57) Set (80.95) ( (78.57) n1 (78.57) , (80.95) n2 (76.19) ) (76.19)
56     p1 (78.57) : (73.81) Run (76.19) ( (78.57) ) (78.57)
57     p2 (80.95) : (76.19) Run (80.95) ( (76.19) ) (78.57)
58     c (85.71) : (80.95) Run (83.33) ( (78.57) ) (80.95)
59     check (76.19)
60       p1 (85.71) : (85.71) Join (88.10) ( (85.71) ) (83.33)
61       p2 (88.10) : (80.95) Join (80.95) ( (80.95) ) (85.71)
62       c (88.10) : (80.95) Join (90.48) ( (83.33) ) (83.33)
63     detect (73.81) e (80.95)
64     end (50.00)
65   end (35.71)
66
67 end (95.24)
```

Figure 5.9: Token Accuracy Map - Threads Task 2.

5.7.3 Task 3

```
1  class (84.09) Main (97.73)
2    action (88.64) P (97.73) ( (90.91) Reader (88.64) < (81.82) integer (86.36) > (84.09)
          in (93.18) , (79.55) Writer (84.09) < (84.09) integer (81.82) > (84.09) out
          (86.36) , (4.55) Writer (6.82) < (2.27) boolean (.00) > (4.55) req (6.82) )
          (90.91)
3      integer (45.45) i (56.82) = (45.45) 0 (43.18)
4      repeat (77.27) while (75.00) i (54.55) < (43.18) 10 (47.73)
5        req (15.91) : (9.09) Write (18.18) ( (9.09) true (36.36) ) (2.27)
6        integer (72.73) myN (86.36) = (84.09) in (81.82) : (72.73) Read (77.27) ( (72.73) )
          (77.27)
7        out (79.55) : (77.27) Write (79.55) ( (81.82) myN (79.55) + (50.00) 1 (68.18) )
          (79.55)
8        i (25.00) = (25.00) i (25.00) + (25.00) 1 (20.45)
9      end (72.73)
10   end (84.09)
11
12   action (81.82) N (84.09) ( (75.00) Reader (79.55) < (79.55) integer (81.82) > (79.55)
          in1 (81.82) , (75.00) Writer (79.55) < (77.27) integer (75.00) > (75.00) out1
          (79.55) , (4.55) Reader (4.55) < (2.27) integer (6.82) > (4.55) in2 (9.09) ,
          (2.27) Writer (2.27) < (2.27) integer (6.82) > (2.27) out2 (6.82) , (.00) Reader
          (.00) < (.00) boolean (.00) > (.00) req1 (.00) , (.00) Reader (4.55) < (.00)
          boolean (.00) > (.00) req2 (4.55) ) (75.00)
13     integer (65.91) n (70.45) = (77.27) 0 (70.45)
14     repeat (68.18) while (70.45) n (52.27) < (36.36) 20 (45.45)
15       boolean (.00) x (40.91) = (6.82) false (29.55)
16       choose (2.27)
17         x (31.82) = (6.82) req1 (6.82) : (68.18) Read (72.73) ( (72.73) ) (11.36)
18           out1 (68.18) : (6.82) Write (9.09) ( (11.36) n (4.55) ) (59.09)
19           n (31.82) = (29.55) in1 (31.82) : (20.45) Read (25.00) ( (25.00) ) (25.00)
20       or (2.27)
21         x (72.73) = (52.27) req2 (54.55) : (47.73) Read (54.55) ( (50.00) ) (47.73)
22           out2 (45.45) : (15.91) Write (25.00) ( (15.91) n (34.09) ) (20.45)
23           n (47.73) = (56.82) in2 (54.55) : (9.09) Read (18.18) ( (11.36) ) (13.64)
24       end (4.55)
25     end (75.00)
26     output (18.18) n (31.82)
27   end (75.00)
28
29   action (77.27) Main (77.27)
30     Channel (84.09) < (72.73) integer (72.73) > (75.00) c1a (75.00)
31     Channel (72.73) < (65.91) integer (65.91) > (68.18) c1b (70.45)
32     Channel (13.64) < (9.09) integer (15.91) > (11.36) c2a (15.91)
33     Channel (4.55) < (4.55) integer (9.09) > (6.82) c2b (15.91)
34     Channel (6.82) < (2.27) boolean (.00) > (2.27) r1 (6.82)
35     Channel (6.82) < (2.27) boolean (.00) > (2.27) r2 (6.82)
36     concurrent (81.82)
37       N (86.36) ( (81.82) c1a (81.82) : (79.55) GetReader (81.82) ( (81.82) ) (84.09) ,
          (77.27) c2b (86.36) : (70.45) GetWriter (79.55) ( (79.55) ) (79.55) , (4.55) r1
          (15.91) : (4.55) GetReader (27.27) ( (18.18) ) (63.64) , (2.27) c2a (79.55) :
          (15.91) GetReader (27.27) ( (79.55) ) (15.91) , (13.64) r2 (77.27) : (75.00)
          GetReader (84.09) ( (77.27) ) (84.09) ) (20.45)
38       P (25.00) ( (20.45) c1a (29.55) : (25.00) GetWriter (31.82) ( (31.82) ) (25.00) ,
          (75.00) c1b (86.36) : (79.55) GetReader (84.09) ( (81.82) ) (84.09) , (9.09) r1
          (11.36) : (4.55) GetWriter (13.64) ( (6.82) ) (79.55) ) (4.55)
39       P (36.36) ( (34.09) c2a (38.64) : (36.36) GetWriter (43.18) ( (38.64) ) (34.09) ,
          (34.09) c2b (52.27) : (43.18) GetReader (50.00) ( (50.00) ) (47.73) , (4.55) r2
          (13.64) : (13.64) GetWriter (27.27) ( (15.91) ) (45.45) ) (4.55)
40     end (65.91)
41   end (86.36)
42
43 end (90.91)
```

Figure 5.10: Token Accuracy Map - CSP Task 3.

```
 1  class (97.50) N (100.00)
 2    integer (92.50) value (100.00) = (95.00) 0 (95.00)
 3  end (60.00)
 4  class (57.50) P (72.50) is (60.00) Thread (67.50)
 5    N (65.00) n (97.50) = (60.00) undefined (60.00)
 6    integer (7.50) i (75.00) = (70.00) 0 (27.50)
 7    action (92.50) Set (95.00) ( (100.00) N (100.00) n (95.00) ) (85.00)
 8      me (87.50) : (95.00) n (100.00) = (95.00) n (97.50)
 9    end (92.50)
10
11    action (95.00) Run (97.50)
12      repeat (40.00) while (40.00) i (92.50) < (37.50) 10 (80.00)
13        synchronized (72.50) ( (77.50) n (92.50) ) (85.00)
14          integer (17.50) temp (90.00) = (60.00) 0 (30.00)
15          temp (92.50) = (62.50) n (90.00) : (80.00) value (87.50)
16          temp (80.00) = (45.00) temp (87.50) + (35.00) 1 (42.50)
17          n (85.00) : (57.50) value (90.00) = (50.00) temp (80.00)
18        end (22.50)
19        i (85.00) = (80.00) i (55.00) + (40.00) 1 (50.00)
20      end (92.50)
21    end (90.00)
22
23  end (90.00)
24  class (87.50) Main (97.50)
25    action (90.00) main (92.50)
26      N (92.50) n (90.00)
27      P (92.50) p1 (92.50)
28      P (90.00) p2 (92.50)
29      p1 (97.50) : (92.50) Set (97.50) ( (97.50) n (92.50) ) (95.00)
30      p2 (100.00) : (95.00) Set (97.50) ( (95.00) n (90.00) ) (95.00)
31      p1 (100.00) : (95.00) Run (97.50) ( (95.00) ) (90.00)
32      p2 (100.00) : (90.00) Run (95.00) ( (95.00) ) (92.50)
33      check (82.50)
34        p1 (97.50) : (95.00) Join (97.50) ( (92.50) ) (95.00)
35        p2 (100.00) : (95.00) Join (97.50) ( (95.00) ) (95.00)
36      detect (82.50)
37      e (97.50)
38      end (60.00)
39      output (32.50) "Final value of N: " (.00) + (2.50) n (52.50) : (27.50) value (35.00)
40    end (90.00)
41
42  end (95.00)
```

Figure 5.11: Token Accuracy Map - Threads Task 3.

Task 3 represented the co-ordinated reader-writer task with two writers and one reader. As in task 2, the primary difficulties were in the parallelism constructs. These can be seen by the score of 2.27% on choose keyword in the CSP task in line 16 in Figure 5.10, although the Threads group scored higher with a score of 72.50% on the synchronized keyword in line 13 in Figure 5.11. The Threads group had more trouble with the repeat loop on line 12 (40.00%) in Figure 5.11 than the CSP group on lines 4 (77.27%) and 14 (68.18%) in Figure 5.10.

Chapter 6

Discussion

In reviewing the evidence from this study, the most interesting result is the wide difference in performance on Task 3 between the groups. The standard deviation of scores between the groups was similar (13.0 for CSP compared to 12.0 for Threads), but the mean was different by 35.5 percentage points (43.6 for CSP compared to 79.1 for Threads). This was interesting not only for the wide gap, but the fact that the other two tasks showed similar performance.

An inspection of the token accuracy maps for Task 3 reveal the particular reason for the differences in scores: participants struggled to understand the synchronous communication pattern in CSP. The evidence for this can be seen in looking at the token accuracy for the `choose` keyword in tasks 2 and 3 and for the number of channels in task 3. In the TAM for all groups, `choose` was matched only 63.64% of the time. In task 3, it was matched only 2.27% of the time. The TAM for task 3 in CSP provides additional evidence of the lack of understanding of the synchronous communication channels by the omission of variable initialization fowr all the required channels. The solution requires a total of 6 channels for the two processes to communicate with the counter. Each process requires 2 channels for two-way communication of data and 1 request channel (to signal that there is data on the communication channel). 84.09% and 72.73% of the respondents created two channels, but only 13.64%, 4.55%, 6.82% and 6.82% created the the third through sixth channels.

The difficulty the participants had with `repeat` constructs in tasks 2 and 3 were not expected. A closer examination of the participants reponses where those tokens were not correct revealed that the entire loop construct was frequently omitted, not just incorrect syntax. This is supported by the TAMs because the `while` token generally had a very similar score to the `repeat` keyword in each of the examples. Initially, I thought that possibly this was an issue with the Quorum syntax

which requires `repeat while` for this type of loop compared to C/C++/Java and other languages that require just `while`, but this is not supported by the TAMs. The difficulty suggests that since this is probably not a syntax issue, it may be related to the participants' overall understanding of the solution to the tasks which clearly require a loop of some kind.

The other notable result of this experiment contradicts our findings in a similarly designed study performed in our lab on the use of lambda expressions compared to iterators as it relates to programmer experience [23]. In this study, we looked at 4 different measures of experience: level in school, course number (with the context of prerequisite courses at UNLV), total years of programming experience and years of Java programming experience. Only one of these measures was even approaching significance in the within-subjects measurement interaction. This finding suggests that learning for parallel programming may be different from traditional programming instruction or is not taught in our curriculum at UNLV. Further exploration of this issue may be warranted.

Chapter 7

Conclusion and Future Work

This study is part of a larger set of experiments on human factors considerations in programming languages. The evidence from this study shows only the paradigm used and the participants native language had an impact on performance on the tasks. None of the measures of experience tracked (academic status, progression in curriculum or self reported years of experience) showed an impact on performance. The group effect was most pronounced on the third task. An analysis of the token accuracy maps of this task indicates that participants in the CSP group consistently omitted statements that indicate they did not naturally understand the mechanics of the synchronous communication in the CSP model.

There are a number of paths to take with future research with the current dataset collected in this randomized controlled trial. The most obvious examination will be to process and score the full 17,786 code snapshots in the event database to try to detect patterns of learning or programming in each task. Interesting information might be found on which tokens are understood and put in place quickly by participants and which were either changed or place later. The paradigms could also be compared and further insight may be gained on differences between them. Additional work could also be performed on the alignment algorithm, such as ensuring proper brace matching in cases where the programmer completed it correctly, or forced alignment of key subsections. Additional automation may be beneficial in the rearrangement of sections of the solution where permissible or in defining alternate solutions. These improvements will not likely have an impact on the overall results of the study, but could fine tune the specific accuracy measurements of individual tokens. Another approach would be to devise a scoring or penalty mechanism for superflous extra tokens in the users answers which do not match with the solution. These extra tokens are ignored in the current scoring algorithm. Finally, no examination of the semantics of a user's answer is made,

such as race conditions or deadlock with this approach.

In further testing of these paradigms, it could be interesting to compare a task involving additional threads acting on a shared object. One could hypothesize that the additional logic and complication of creating a thread-safe locking protocol could be more complex than a CSP implementation and therefore a crossover in results may occur as programs get more complex. Although this study shows no impact from learning or experience among our test subjects, it might be interesting to test groups with and without additional training on the paradigms or concurrency.

The next phase in this research line is to examine approaches to integrate various models for parallelism and vectorization into alternative representations that may be easier or more intuitive for developers. I plan to examine common parallel usage patterns and evaluate alternative approaches for implementing them. The focus will be on research scientists usage patterns requiring high performance computation, not on everyday usage situations. I anticipate starting the study by categorizing key constructs required and performing a number of surveys for keyword and syntax choices in an effort to design a more naturally intuitive system to take advantage of various optimization standards in use by the community.

Appendix A

IRB Documents

Attached are the approval documents from the IRB for this series of experiments on programming languages.

Social/Behavioral IRB – Exempt Review
Deemed Exempt

DATE: October 15, 2014

TO: **Dr. Andreas Stefik,** Computer Science

FROM: Office of Research Integrity – Human Subjects

RE: Notification of IRB Action
                Protocol Title: Scientific Computing Study on Programming Language
                Protocol # 1409-4928

This memorandum is notification that the project referenced above has been reviewed as indicated in Federal regulatory statutes 45CFR46 and deemed exempt under 45 CFR 46.101(b)2.

PLEASE NOTE:
Upon Approval, the research team is responsible for conducting the research as stated in the exempt application reviewed by the ORI – HS and/or the IRB which shall include using the most recently submitted Informed Consent/Assent Forms (Information Sheet) and recruitment materials. The official versions of these forms are indicated by footer which contains the date exempted.

Any changes to the application may cause this project to require a different level of IRB review. Should any changes need to be made, please submit a **Modification Form**. When the above-referenced project has been completed, please submit a **Continuing Review/Progress Completion report** to notify ORI – HS of its closure.

If you have questions or require any assistance, please contact the Office of Research Integrity - Human Subjects at IRB@unlv.edu or call 895-2794.

Office of Research Integrity – Human Subjects
4505 Maryland Parkway • Box 451047 • Las Vegas, Nevada 89154-1047
(702) 895-2794 • FAX: (702) 895-0805

www.manaraa.com

INFORMED CONSENT

Department of Computer Science

TITLE OF STUDY: Empirical Investigation into Programming Language Syntax

INVESTIGATOR(S): Dr. Andreas Stefik

For questions or concerns about the study, you may contact Dr. Andreas Stefik at 702-895-3603**.**

For questions regarding the rights of research subjects, any complaints or comments regarding the manner in which the study is being conducted, contact **the UNLV Office of Research Integrity – Human Subjects at 702-895-2794, toll free at 877-895-2794 or via email at IRB@unlv.edu.**

Purpose of the Study
You are invited to participate in a research study. The purpose of this study is to study programming languages.

Participants
You are being asked to participate in the study because you fit this criteria: over 18 years old with <u>no</u> previous programming experience.

Procedures
If you volunteer to participate in this study, you will be asked to do the following:
1. Read about computer programming languages
2. Practice doing some computer programming
3. Take a survey about your experiences

Benefits of Participation
You may learn about programming languages by participating in this study. Other than possibly contributing to scientific knowledge in the field, there are otherwise no benefits for participation.

Risks of Participation
There are risks involved in all research studies and while this study includes only minimal risks, you may experience some mental distress. Many students that are learning to program a computer find the experience challenging. Computer programming can be very mathematical and might require significant effort. Some students may even find the task of programming itself to be complex and frustrating. Tasks in this study are timed and you may feel uncomfortable with timed tasks. Further, the easiest way for us to gather information about how well our tools are working is to do so electronically. While we will do our best to keep all information confidential, and the information we are collecting is not of a personal nature, no computer security mechanism is perfectly secure.

Deemed exempt by the ORI-HS and/or the UNLV IRB. Protocol 1409-4928
Exempt Date: 10-15-14

Bibliography

[1] *SKA Telescope: Square Kilometre Array*, 2016 (accessed April 29, 2016).

[2] *Quorum Programming Language*, 2016 (accessed April 30, 2016).

[3] BROOKES, S. D., HOARE, C. A. R., AND ROSCOE, A. W. A theory of communicating sequential processes. *J. ACM 31*, 3 (June 1984), 560–599.

[4] CASTOR, F., OLIVEIRA, J. P., AND SANTOS, A. L. Software transactional memory vs. locking in a functional language: a controlled experiment. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11* (2011), ACM, pp. 117–122.

[5] COBLENZ, M., SEACORD, R., MYERS, B., SUNSHINE, J., AND ALDRICH, J. A course-based usability analysis of cilk plus and openmp. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on* (2015), IEEE, pp. 245–249.

[6] HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.

[7] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[8] KAIJANAHO, A.-J. *Evidence-Based Programming Language Design: A Philosophical and Methodological Exploration*. PhD thesis, University of Jyvaskyla, 2015. Information Technology Faculty.

[9] KAPTCHUK, T. J. Intentional ignorance: A history of blind assessment and placebo controls in medicine. *Bulletin of the History of Medicine 72*, 3 (1998), 389–433.

[10] KITCHENHAM, B., PEARL BRERETON, O., BUDGEN, D., TURNER, M., BAILEY, J., AND LINKMAN, S. Systematic literature reviews in software engineering - a systematic literature review. *Inf. Softw. Technol. 51*, 1 (Jan 2009), 7–15.

[11] LEE, E. A. The problem with threads. *Computer 39*, 5 (May 2006), 33–42.

[12] LEWANDOWSKI, G., BOUVIER, D. J., MCCARTNEY, R., SANDERS, K., AND SIMON, B. Commonsense computing (episode 3): concurrency and concert tickets. In *Proceedings of the third international workshop on Computing education research* (2007), ACM, pp. 133–144.

[13] MARX, V. Biology: The big challenges of big data. *Nature 498*, 7453 (2013), 255–260.

[14] MATTMANN, C. A. Computing: A vision for data science. *Nature 493*, 7433 (2013), 473–475.

[15] MCCOOL, M. D., ROBISON, A. D., AND REINDERS, J. *Structured parallel programming: patterns for efficient computation.* Elsevier, 2012.

[16] NANZ, S., TORSHIZI, F., PEDRONI, M., AND MEYER, B. Empirical assessment of languages for teaching concurrency: Methodology and application. In *Software Engineering Education and Training (CSEE&T), 2011 24th IEEE-CS Conference on* (2011), IEEE, pp. 477–481.

[17] NEEDLEMAN, S. B., AND WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology 48*, 3 (1970), 443–453.

[18] PANKRATIUS, V., AND ADL-TABATABAI, A.-R. A study of transactional memory vs. locks in practice. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures* (2011), ACM, pp. 43–52.

[19] PARR, T., AND FISHER, K. Ll(*): The foundation of the antlr parser generator. *SIGPLAN Not. 46*, 6 (June 2011), 425–436.

[20] PARR, T., HARWELL, S., AND FISHER, K. Adaptive ll(*) parsing: The power of dynamic analysis. *SIGPLAN Not. 49*, 10 (Oct. 2014), 579–598.

[21] ROSSBACH, C. J., HOFMANN, O. S., AND WITCHEL, E. Is transactional programming actually easier? *ACM Sigplan Notices 45*, 5 (2010), 47–56.

[22] Stefik, A., and Siebert, S. An empirical investigation into programming language syntax. *Trans. Comput. Educ. 13*, 4 (Nov. 2013), 19:1–19:40.

[23] Uesbeck, P., Stefik, A., Hanenberg, S., Pedersen, J., and Daleiden, P. An empirical study on the impact of c++ lambdas and programmer experience. In *2016 IEEE/ACM 38th IEEE International Conference on* (2016), IEEE.

[24] U.S. Department of Education Institute of Education Sciences. *What Works Clearinghouse Procedures and Standards Handbook*, 2.1 ed. U.S. Department of Education, 2010.

[25] Vogt, W. P. *Quantitative Research Methods for Professionals in Education and Other Fields*, 1st ed. Allyn and Bacon, Columbus, OH, 2006.

[26] Zhang, H., Babar, M. A., and Tell, P. Identifying relevant studies in software engineering. *Information and Software Technology 53*, 6 (2011), 625–637.

68

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Patrick M. Daleiden

Degrees:

Bachelor of Arts (Economics and Arts & Letters Program for Administrators) 1990

University of Notre Dame

Master of Business Administration (Finance and Accounting) 1993

University of Chicago

Professional Certifications:

Chartered Financial Analyst 1993

CFA Institute

Certified Public Accountant 1995

University of Illinois Board of Examiners

Honor Societies:

Beta Gamma Sigma 1994

University of Chicago Chapter

Phi Kappa Phi 2016

University of Nevada, Las Vegas Chapter

Thesis Title: Empirical Study of Concurrent Programming Paradigms

Thesis Examination Committee:

Chairperson, Dr. Andreas Stefik, Ph.D.

Committee Member, Dr. Jan Pederson, Ph.D.

Committee Member, Dr. Ajoy Datta, Ph.D.

Graduate Faculty Representative, Dr. Matthew Bernacki, Ph.D.