TOWARD PRODUCTIVITY IMPROVEMENTS IN PROGRAMMING LANGUAGES THROUGH BEHAVIORAL ANALYTICS

by

Patrick Michael Daleiden

Master of Science - Computer Science
University of Nevada, Las Vegas
2016

Master of Business Administration - Finance and Accounting
University of Chicago, Chicago, IL
1993

Bachelor of Arts - Economics
University of Notre Dame, Notre Dame, IN
1990

A dissertation submitted in partial fulfillment of
the requirements for the

Doctor of Philosophy – Computer Science

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

University of Nevada, Las Vegas
May 2020

UNLV | Howard R. Hughes College of ENGINEERING

The Graduate College

We recommend the dissertation prepared under our supervision by

Patrick Michael Daleiden

entitled

Toward Productivity Improvements in Programming Languages through Behavioral Analytics

be accepted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy – Computer Science

Department of Computer Science

Andreas Stefik, Ph.D., Committee Chair

Laxmi Gewali, Ph.D., Committee Member

John Minor, Ph.D., Committee Member

Angelos Yfantis, Ph.D., Committee Member

Kendall Hartley, Ph.D., Graduate College Representative

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

May 2020

Abstract

Computer science knowledge and skills have become foundational for success in virtually every professional field. As such, productivity in programming and computer science education is of paramount economic and strategic importance for innovation, employment and economic growth. Much of the research around productivity and computer science education has centered around improving notoriously difficult compiler error messages, with a noted surge in new studies in the last decade. In developing an original research plan for this area, this dissertation begins with an examination of the Case for New Instrumentation, drawing inspiration from automated data mining innovations and corporate marketing techniques in behavioral analytics as a model for understanding and prediction of human behavior. This paper then develops and explores techniques for automated measurement of programmer behavior based on token level lexical analysis of computer code. The techniques are applied in two empirical studies on parallel programming tasks with 88 and 91 student participants from the University of Nevada, Las Vegas as well as 108,110 programs from a database code repository. In the first study, through a re-analysis of previously captured data, the token accuracy mapping technique provided direct insight into the root cause for observed performance differences comparing thread-based vs. process-oriented parallel programming paradigms. In the second study comparing two approaches to GPU programming at different levels of abstraction, we found that students who completed programming tasks in the CUDA paradigm (considered a lower level abstraction) performed at least equal to or better than students using the Thrust library (a higher level of abstraction) across four different abstraction tests. The code repository of programs with compiler errors was gathered from an online programming interface on curriculum pages available in the Quorum language (quorumlanguage.com) for Code.org's Hour of Code, Quorum's Common Core-mapped curriculum, activities from Girls Who Code and curriculum for Skynet Junior Scholars for a National Science Foundation funded grant entitled Innovators Developing Accessible Tools for Astronomy (IDATA). A key contribution of this research project is the development of a novel approach to compiler error categorization and hint generation based on token patterns called the Token Signature Technique. Token Signature analysis occurs as a post-processing step after a compilation pass with an ANTLR LL* parser triggers and categorizes an error. In this project, we use this technique to i.) further categorize and measure the root causes of the most common compiler errors in the Quorum database and then ii.) serve as an analysis tool for the development of a rules engine for

enhancing compiler errors and providing live hint suggestions to programmers. The observed error patterns both in the overall error code categories in the Quorum database and in the specific token signatures within each error code category show error concentration patterns similar to other compiler error studies of the Java and Python programming languages, suggesting a potentially high impact of automated error messages and hints based on this technique. The automated nature of token signature analysis also lends itself to future development with sophisticated data mining technologies in the areas of machine learning, search, artificial intelligence, databases and statistics.

Acknowledgements

My first thanks to my family, especially my wife Liliana Bonilla Escobar who joined me from Colombia in the middle of this trek and who made the journey so pleasant with her loving support, tolerance and extra efforts at home (Te amo mi corazón). A very special thanks to my four grown children Brian, Bridget, Valerie and Megan, who challenge me, support me and give me motivation and to my step daughter Antonia who has brought so much love, energy and laughter into my life. Many thanks to my first family, especially my brother Dr. Eric Daleiden and his wife Dr. Shannon Turner Daleiden for their inspiration and consultation as well as to my father, the original Dr. Daleiden, for his support through thick and thin and to all of my Midwestern cousins, uncles and aunts too numerous to mention. A final personal thanks to my good friend and colleague, Dr. Lanh Tran and my newfound brother Chris Preisel.

As to UNLV, I express my gratitude and appreciation to my advisor, Dr. Andreas Stefik, for his counsel, patience and friendship during these past six years in teaching an old dog new tricks. I thank Dr. Jan "Matt" Pedersen for his inspirational teaching and also as a friend and collaborator. Special thanks to Dr. Laxmi Gewali for his wisdom, leadership and mentoring and to the late Dr. Ajoy Datta for his advocacy and confidence in me. I further thank the other members of my two committees, Dr. John Minor, Dr. Angelos Yfantis and Dr. Matt Bernacki and to Dr. Kendall Hartley for his friendship, advice and encouragement. I further acknowledge the other faculty members in the Department of Computer Science for their excellence and care in teaching, as well as my fellow doctoral students and friends including Guymon Hall, Dr. P. Merlin Uesbeck and Dr. Ed Jorgensen. A final thanks to my friends in the Graduate College, especially to Dean Kathryn Korgan for her enthusiasm and advocacy for graduate education.

"*Saints are sinners who kept on going.*" – Robert Louis Stevenson

Patrick Michael Daleiden

University of Nevada, Las Vegas

May 2020

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Digital Revolution [Sch17] which began during the 20th century has made technology ubiquitous in our daily lives. The changes that have occurred in our transition to the Information Age have fundamentally affected the global economy. The U.S. Department of Commerce (USDOC) recently stated that economic growth and competitiveness are increasingly tied to the digital economy as the number of individuals with access to the internet has risen from roughly 45 million in 1995 to over 3 billion in 2014 [Dav17]. In 2016, a USDOC report found that intellectual property (IP) intensive industries support at least 45 million jobs (30% of all U.S. employment) and contribute more than $6 trillion dollars of U.S. gross domestic product (38% of total GDP). Furthermore, workers in IP-intensive industries earn approximately 46% more than workers in non-IP-intensive industries, continuing an upward trend over the last 25 years [U.S17b]. In 2017, a BNY Mellon sponsored report authored by a group of industry experts in computer science education [PZS+17] explained that because computing had profoundly impacted our lives, it is important that students develop competency as generators of digital resources, not just users. Among other things, the report concluded that computer science knowledge and skills have become foundational and that computer science is a central component of innovation, economic growth and employment.

Despite all of this, however, computer science education is not meeting the demands of the economy. In a report of a special commission on Computer Science Education and Information Technology of the Southern Region Education Board, the commission claimed that computational thinking skills and knowledge of computer science are required in nearly all career fields, but lamented that the nation is not on track to meet labor market demand for computing resources [SGA+16]. Code.org affirms the labor shortage with numbers, claiming there are currently 486,686 open computing jobs nationwide, but that last year only 42,969 computer science students graduated into the workforce [cod17]. Part of the reason for this shortage is that only 58% of high schools offer a computer science course and of those that do only 47% teach programming, so only about 1 in 4 offer an opportunity to learn a key element of computer science required in the workplace [Goo17].

There has been movement in recent years to establish national standards for computer science education such as AP Computer Science Principals [The17] and the K-12 framework [k1217]. President Obama attempted to address this issue and recognized the shift toward a digital economy in his 2016 State of the Union in creating the Computer Science for All initiative [Oba17] which proposed $4 billion in funding for states to increase computer science education in K-12 and $135 million in funding for the National Science Foundation.

Even with co-ordinated statewide and federal programs in conjunction with technology industry sponsorships to enhance the quantity of computer science education, however, we are still left with the issue of the quality of education. Computational thinking and computer programming are considered cognitively difficult tasks [WK14] and college level introductory computer science courses often have attrition and failure rates of 20% to 50% [Yue07]. Introductory students are generally confused by basic syntax in computer programming languages [KK03] as well as compiler error messages [DLRT12]. Important skills like debugging are both difficult for novices to learn [MLM+08] and for instructors to teach [MFL+08]. Integrated development environments can assist beginners in learning programming but they can also overwhelm them. [SDM+03, RT05] The problem is compounded by the prevalence of these types of problems across languages [SS13]. Part of the solution to the problem is to understand more about student errors in order to improve instruction [SPBK88] but another part is to improve programming languages themselves.

The problem of programmer productivity also goes beyond just novice instruction. An internal Google study [SSE+14] on over 26 million build errors made by 18,000 of their own profesional software developers found failure rates of 28% to 38% (depending on the language used). Considering that the average developer built their code 7 to 10 times per day according to Google's statistics, if we assume that on average 5 minutes of every hour was spent on building code and one third of those builds failed, about 2.8% of their time was unproductive. At an annual salary of $100,080 (the U.S. Bureau of Labor Statistics average annual salary for software developers and programmers in 2016 [U.S17a]), the lost productivity of these 18,000 developers was in the neighborhood of $50 million that year. At a national level, there were 1,604,570 software developers and programmers in the 2016 survey [U.S17a], which translates to over $4.4 billion in identifiable losses from just one part of the problem. The same productivity problem also exists for scientists and academic researchers, as highlighted by a multi-disciplinary team at Berkeley studying parallel computing [ABC+06]. Lost labor costs only tell part of the story in this area though, because the pace of scientific discovery is affected and the real cost is an opportunity cost. In our increasingly technology-based world, research scientists' productivity is directly tied to their ability to utilize computers efficiently.

This research proposal attempts to address the problem of programmer productivity by examining the core issues of programmer behavior and language design. It is based on the development of a behavioral analytics suite to enable researchers to more fully understand and measure human factors impacts of programming language and library design decisions on software developers in order to achieve the ultimate goal of helping to make programmers more productive. Multidisciplinary research work exploring the psychological impact

of different computer languages on science and problem solving is already underway based on the established hypothesis that different spoken languages can influence thinking in certain directions [Uni17a]. Extending the study of computer programming using a behavior analytics approach seems like a natural fit.

The concept of behavioral analytics is relatively new, but is becoming formalized and is in wide use by corporate interests seeking to maximize profits [IBM13] and for data security and anomaly detection [CA 17]. Cao [Cao08] explained that behavioral data is increasingly examined for pattern analysis and business intelligence because of its predictive capability, citing usage examples in customer relationship management, social computing, fraud detection, event analysis, outlier detection and group decision making. Behavioral analytics is an advancement to traditional transactional data analysis in that it takes advantage of new data points that are now available through digital interaction, such as mouse clicks on a website with timestamps and browsing history in order to gain insight into and influence over consumer intent. These behaviors are analyzed using pattern recognition algorithms, machine learning techniques and artificial intelligence and compared to databases of other consumer behavior with known outcomes. The result of a successful behavioral analytics program is that a business will understand its customers better and can tailor its services to maximize revenue.

Although the metrics will be different, this research plan proposes to borrow this concept and apply it to the study of computer programming. The proposal is based on a series of programming studies and code repository analysis where programmer output and behavior is measured empirically in randomized controlled trials where possible. During these studies, our testing platform captures various forms of data generated while participants complete programming tasks, including code snapshots, compiler error messages, and timestamps. The data will be analyzed collectively using automated machine learning algorithms for pattern recognition. Additionally the code samples will be broken down into token streams and analyzed with automated sequence alignment algorithms borrowed from the field of bioinformatics. Finally, data visualization elements will be developed for visual monitoring and pattern recognition. The overall objective is to identify programming behavior patterns in overall productivity, debugging and program correctness in order to predict areas of difficulty and strength. These patterns can ultimately be used to inform teaching, to improve language design and to increase programmer productivity.

The research plan builds on the testing platform and automated token alignment work previously complete in a study of parallel programming paradigms [Dal16]. In terms of software development for this toolkit, the testing platform itself is being enhanced to capture additional data as well as to provide new functionality for more flexible language studies. The automated token alignment algorithms are being improved to enhance accuracy and consistency and additional algorithms and settings will be explored. Additional behavioral analytics will be applied and developed including machine learning techniques for pattern recognition. A visualization toolkit will be developed for reporting purposes and qualitative assessment for pattern recognition.

The design philosophy of this analytics toolkit is to develop empirical measurement and reporting capabilities for use by the programming language research community. The ultimate goal of this research is to make programming easier, to make programmers better through improved accuracy and program correctness and to improve programmer training and learning. In the future, this research work could provide the foundation for a set of pedagogical tools to assist programmers not only at the novice level but also at advanced levels with more difficult concepts, like parallel programming. Although it is beyond the scope of this project, the intention is to create a research basis that could be built on to provide improved tool support for programmers through a real time token mapping-based pedagogical tool to provide IDE editor hints or debugging suggestions.

Chapter 2

Research Justification

2.1 The Case for New Instrumentation

The empirical study of programming languages is a relatively young, but growing area of computer science research [Kai15]. The empirical studies that have been done so far in this area use basic comparison metrics such as time to completion [USH+16], time to fix [DLRT12], number of errors [BA17], classification of types of errors [RHW10a] or even softer qualitative observations [TCAL13]. These metrics all have value in a researcher's tool kit and they represent a logical place to start, but computer programming is a complex task and more instrumentation is needed for measurement and quantification. Additional information about programmer behavior and intent (both statically and over time) could significantly enhance a researcher's understanding of the human factors impacts of programming language design decisions. A data set of patterns of behavior in errors, error correction, successful solutions, library usage, best practices and other areas could be analyzed to learn much more than we can learn from basic metrics. Additionally, these datasets could be used with machine learning techniques to develop predictive models that could be used to learn about programmer intent.

In addition to providing more detailed information on problem areas than existing study methodologies (which may only measure time to completion, for example, instead of the nature of the problem) the language analytics toolkit could allow us to learn, from another perspective, how people go about programming, problem solving, debugging and even learning programming. For example, we can use these tools to empirically measure how people approach the implementation of a flow control construct and try to correlate those patterns to successful and unsuccessful results. We can use the tools to explore areas where we can speed up programming and provide more contextual help in concepts similar to code completion. We can use token string similarity to match and rank potential constructs from databases to correct bugs or provide library support. We can analyze code to identify possible missing fragments used commonly by others in the same

way Amazon can suggest that if you just bought item A, you might also want item B, C and D because that is what other consumers bought. Amazon is a well known example of using behavioral data to make product recommendations [IBM13]. The freedom from the strict requirements of parsing provide many new avenues to pursue.

2.1.1 The Syntax Problem

The introduction in Becker [Bec16] discusses over forty years of research in syntax and compiler errors and laments the still common themes throughout. The issues consistently span most languages and certainly includes the most popular languages. In 1976, Wexelblast [Wex76] satirically highlighted how malfeasant language designers could make programming as difficult as possible by listing a series of maxims of bad language design which were common in many languages of the time. Although he was highlighting problems in Fortran, PL/I and Algol68 then, many of these issues continue in modern languages today. In the 1980's, Sleeman [SPBK88] summarized and quantified the errors they observed in teaching Pascal at Stanford to introductory students with similar errors. In the 1990s, Schorsch [Sch95] developed corrective tools as a teaching aid to solve the same types of Pascal errors at the United States Air Force Academy. Contemporaneously, Freund and Robers [FR96] at Stanford developed a tool to help with similar ANSI C syntax errors for beginners. In the 2000's, researchers were working on resolving the same novice problems yet again in Java with Flowers [FCJ04] at the United States Miltary Academy (West Point) and Hristova [HMRM03] at Bryn Mawr College. Many more studies confirmed the same problems and in the 2010's we even saw a comparison study by Stefik et al. [SSSS11] of basic syntax errors in Perl to a randomly generated nonsense control language which observed no evidence that Perl was superior. In 2017, studies continue to find and document the same problems and the limited impacts of our approaches as highlighted by Prather et al. [PPM+17]

One approach to solving this problem is to provide programmers with enhanced compiler error messages although the evidence shows mixed results on their effectiveness. Becker [Bec16] showed improvements in 9 out of 15 of the most common errors and Denny [DLRC14] showed no improvements over a control group, although these results suffer from small sample sizes, which impede statistical significance calculations. Petit et al. [PHG17], motivated by the inconsistency of these two results, modified an automated assessment tool they use called Athene to incorporate this research and run their own study over four semesters in a C++ course and found no impact from the enhanced compiler messages. Prather et al. [PPM+17] reviewed the inconsistent results of various studies and identified two possible causes: 1.) that students do not read the enhanced compiler messages and 2.) that the messages are properly designed but students do not understand them in situations of high cognitive load while debugging. In any event, despite focused academic research on the topic and so many attempts to solve the problem, there is still no compiler error enhancement tool in wide use, so the syntax barrier persists.

Enhanced compiler message studies by Becker [Bec16], Denny et al. [DLRC14] and Pettit et al. [PHG17] follow a similar format: i.) researchers categorize common errors and then ii.) develop techniques and approaches for assisting a programmer to fix an error more efficiently by creating some tool for enhancing messages or identifying errors or providing feedback and then iii.) compare student performance before and after the introduction of the tool. This is a valid and useful research approach, but with a language analytics approach we can create the capability to attack the problem closer to the root by developing a research toolkit to assist in language design itself to try to avoid or minimize the syntax problems in the first place. It is time for a fresh approach to these very well documented issues so that researchers can learn more about the underlying causes of what is going on so that a solution can be found.

2.1.2 Illustrative Basic Example

As an illustrative example, consider that the six compiler error studies examined by Becker [Bec16] have provided corroborating evidence that unbalanced parenthesis, curly braces, square brackets and quotation marks are the among the most common syntax errors. Compilers have a difficult time zeroing in on the particular item that is unbalanced because of the nature of parsing. As an aside, integrated development environments often auto-complete the right token when you type the left token either as a convenience or reminder to keep them in balance. Using a token matching approach and observing a programmer's behavior over time with a particular token could shed light on how these errors arise. For example, if a programmer habitually types (or has auto completed) the closing token before filling in the body of the code block, we might be able to determine how likely errors are to arise and then make predictive suggestions at a helpful time. It is possible other syntax or logic errors are correlated with common mistakes also and a token mapping may help predict those.

Consider a JQuery AJAX method in JavaScript with an anonymous callback function that has the basic structure:

```
$.post(url, data, callback);
```

In practice, this type of method can be fleshed out further inside a button callback where keeping track of opening and closing curly braces and parenthesis is more complicated, not to mention the commas separating the parameters and the semicolons at the end of the lines. The code readability may also be affected by optional or non-existent indentation as well. Systematically observing and measuring a programmer's behavior while she is programming could lead to insight in patterns that correlate to different syntax errors and would be difficult or impossible to observe with other measurement and assessment tools.

```
$.("button").click(function(){
$.post(
url,
```

```
{
...
},
function(response) {
...
});
});
```

A missing or mismatched closing token in this case would be very difficult to identify through a standard compiler error, which would likely just get to the end and realize a token was missing somewhere (depending on which token was missing). With a context informed approach with token mapping, additional information would be available and behavior analytics could make a specific recommendation. Additionally, this type of platform could measure the frequency of the specific token errors and identify patterns. For example, it is possible that users are confused by the ordering of the curly brace and parenthesis at the end, but not the curly braces embedded in the data section, or possibly they just omit the closing parenthesis or semicolon. With a compiler alone, the error can not be specifically identified in this more complex case.

2.2 Language Productivity Analytics Platform

The result of this research project will be the development of a suite of tools and methodologies intended to be generally useful to the programming language research community. The platform is intended to provide enhanced information about programmer behavior so that researchers can provide informed and empirically supported recommendations to language designers about language and library design.

We will explore the application of various established pattern matching algorithms, local and global sequence alignment algorithms, string similarity algorithms and ranking algorithms to determine which techniques can allow us to best predict and identify programmer intent. In our algorithm research and analysis phase we will be cognizant of and examine the computational complexity of the pattern matching techniques to evaluate the potential for development of real time tools that can be applied inside an integrated development environment or in error messages to provide assistive information to a programmer for error avoidance, debugging or learning.

2.2.1 Token Analysis and Mapping

A key component of the analytics toolkit will be the further development of the token analysis and mapping methodology originally created by Stefik and Siebert [SS13] to analyze syntax accuracy. I extended and automated the methodology in my Masters Thesis [Dal16] using the Needleman-Wunsch sequence alignment algorithm created to compare DNA sequence similarity. Originally called Token Accuracy Maps, the token mapping technique compared a participant's code submission for a given task to a particular answer on a

	<>	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 2.1: Token Based Sequence Alignment

token by token basis after alignment. The token alignments were scored individually and then aggregated to provide overall accuracy scores as well as token level correctness comparisons. The empirical data in the form of an accuracy score for a participant for each task provided an objective criteria to run statistical analysis. Figure 2.1 provides a visual example of the token sequence alignment using the Needleman-Wunsch algorithm where the correct answer tokens are displayed along the y-axis and the participant response tokens on the x-axis. Figure 2.2 depicts the resulting scored alignment on a token by token level.

In using the token accuracy map approach to measure the results of a study comparing process-oriented parallel programming to a threads-based paradigm, we discovered that although the token accuracy maps can give us useful information about programmer accuracy, they have limitations. A particular limitation of concern is in scoring alternate semantically correct solutions. In our experiment, we compared the participant responses to a particular code solution. The first type of problem can occur with simple syntax differences between a response and the solution, such as variable names or non-material order variation. These can be

```
1  class                63  class                63      1
2  Main                 67  Main                 67      1
3  action               35  action               35      1
4  F                    67  F                    67      1
5  -                    0   output               1       0
6  integer              37  "hello"              68      0
7  a                    67  end                  62      0
8  =                    46  action               35      0
9  1                    65  G                    67      0
10 output               1   output               1       1
11 a                    67  "world"              68      0
12 end                  62  end                  62      1
13 action               35  action               35      1
14 Main                 67  Main                 67      1
15 concurrent           8   concurrent           8       1
16 F                    67  F                    67      1
17 (                    58  (                    58      1
18 )                    59  )                    59      1
19 -                    0   G                    67      0
20 -                    0   (                    58      0
21 -                    0   )                    59      0
22 end                  62  end                  62      1
23 end                  62  end                  62      1
24 end                  62  end                  62      1
25                                               ==
26                                               sum     15
27                                               count   24
28                                               score   62.5%
29
```

Figure 2.2: Optimal Token Alignment of Response to Solution

fairly easily addressed through manual inspection and a comparison to a different valid solution or a manual reordering or renaming of the participant response.

The second type of problem is where the participant solves the problem in the task in a semantically correct, but alternative manner to the solution code. This situation could be handled by manually categorizing responses into groups and generating correct solutions for each approach. An overall accuracy score would still be valid in this case, however the token accuracy comparison would only be valid within the group. In our first experiment, our manual inspection did not reveal this to be an issue in any of the responses, most likely due to the specificity of the instructions and code samples provided, however in a follow on study on enum the participant responses were so different that an overall accuracy score using token mapping was meaningless.

We have found that the token mapping approach to analyzing participant responses in programming language studies has provided additional valuable unintended information on programming languages beyond the intended scope of the studies. For example, in the parallel paradigm experiment referenced above, we found that the repeat loop construct for Quorum had an identifiable and quantifiable error pattern. In an indefinite repeat loop in both paradigm groups, the token "true" used in the looping condition had unusually low accuracy scores in both groups (4.6% and 4.8%). In a manual inspection of the cause, we found that the code samples given as a learning and reference device to the participants (who were all novice Quorum users) did not contain this exact structure, so the participants were left to guess at an intuitive solution for themselves. It suggests a problem with the existing Quorum repeat constructs that warrants further examination for the language designers. This finding has nothing to do with this experiment, but it points to

the usefulness of this approach to provide information that would not be gathered by standard measurement tools like timing data, interviews or error examination. Armed with this experience, we will look to develop capability to do this type of meta analysis across experiments on localized language constructs.

2.2.2 Toolkit Components and Extensions

Through the application of additional algorithms from bio-informatics and natural language processing to the token streams, I will attempt to develop novel ways to measure how people program, what mistakes they make and what patterns they exhibit for fixing mistakes and finding bugs. By matching token patterns in user programs to correct (or incorrect) patterns at micro and macro levels throughout the programming process using snapshots collected in the testing environment, I hope to identify and correlate patterns of behavior to generate predictions. Using behavioral analytics techniques I plan to use token mapping to make educated guesses at programmer intent to provide information beyond what a compiler can determine or what we can empirically measure with existing research tools. In the studies our lab has already conducted, we have discovered that we can learn more about what a programmer may be trying to do by looking at token patterns and accuracy comparisons than just at errors or timings. We will seek now to expand our analysis toolkit with new algorithms and visualization techniques with the goal of developing better instrumentation for programming language research.

I will also extend the basic token mapping methodology to use in analysis over time comparisons. The snapshot system in our testing environment will be used to examine not just static patterns but patterns as they develop over time to enable a type of automated observational study. Looking at *diffs* between periods, we can identify active areas and active tokens. Comparing this activity to compile attempts and errors may yield information about successful and unsuccessful programming behaviors as well as patterns of learning and strategies for problem solving and debugging. An analysis over time approach is similar in concept to and would provide some benefits of a think aloud study protocol, although, it is not a direct substitute because we would not capture subjects explicit reporting of their thought patterns. We could gather sequencing information of behavior at a token level and use our other tools to generate hypothesis about intent. Additionally, the automated nature of our analysis could compare the behavior against a database of others to identify and predict patterns.

Finally we will develop a set of reporting tools, including both statistical evaluation and visualization tools for use by the programming language research community. The statistical evaluation tools will look at quantifiable metrics to examine and measure aspects of programmer behavior. The goal of the tools will be to provide empirical information to identify problem areas in programming language design. The visualization tools will be less empirical and more assistive so that researchers can identify patterns and comprehend scopes through visual representations of programmers' behavior.

2.3 Contribution of a Language Analytics Platform

2.3.1 Limitations of Compiler Error Research

Compiler errors are studied by researchers because they provide valuable information about the types and nature of programming mistakes and represent an area of common problems [KK03]. They can also be quantified and counted and can therefore be used to compare different paradigms and languages. The empirical analysis of compiler errors along with attempts to provide better error information to programmers have inspired numerous researchers to explore various techniques improve programming behavior.

Compiler errors are inherently limited, however, because they can only provide a static code analysis that is generated by some failure to parse the programmer's code. A compiler is constrained to identifying and interpreting what it was expecting. It cannot precisely identify the specific point of certain types of common errors, only the point of failure. Compilers can be built to be tolerant of errors and resume parsing, but errors tend to cascade after even a single error and subsequent errors can bear less and less resemblance to the actual cause. Basic errors can be enhanced to a certain degree, but a compiler has no ability to guess at a programmer's intent, so there is also an inherent limit on this approach to improving programming.

Researchers have generally identified through studies of the most common syntax errors [KK03, DLRC14, DLRT12, Bec16, BA17] that syntax errors disproportionately affect novices both in quantity and time to fix [BA17]. A common approach to providing more help to these novices is to provide enhanced compiler error messages to address shortcomings of basic messages in properly identifying a specific syntax error and assisting novices in finding solutions quickly.

In examining enhanced compiler messaging research, I found several examples of author-cited limitations and threats to validity of parsing and other approaches they have taken in providing enhanced error messages which could be improved with a language analytics platform through a fresh perspective and further information on many of the syntax errors targeted by enhanced compiler messages. This information could both further enhance our understanding of the human factors impact of these messages and allow us to develop even more targeted enhanced messages.

Denny et al. [DLRC14] point out the weakness of a compiler's ability to identify certain types of syntax errors without more information on the context of the error. The authors used a static analysis tool with regular expressions to identify and classify some of these errors in their system to provide enhanced compiler messages. A token mapping approach would be more flexible and accurate in the errors identified, which are limited by the capability of regular expressions and expected responses.

Kery et al. [KLGM16] acknowledge that their AST dataset was limited because of the difficulty of determining which exceptions are possible in a particular `try` block. A token mapping and language analytics approach

could provide more information on the possible exceptions identified by the authors through the mapping of programmer code against a predetermined data set of best practice handling blocks for particular calls.

Brown and Altadmri [BA17] identifies the most commonly occurring error in an analysis of two years of compiler submissions and errors in their Blackbox dataset (1,861,627 occurrences vs 1,034,788 for the second most common error) as: Unbalanced parentheses, curly or square brackets and quotation marks, or using these different symbols interchangeably. This error category includes at least five different issues, including unbalanced parenthesis (missing), unbalanced curly braces (missing), unbalanced square brackets (missing), unbalanced quotation marks (missing), OR using these interchangeably with each other. The context of these errors could be completely different and imply different types of programmer intent as they could stem from different problems. A token mapping approach could determine the specific nature of this error which could be used to provide information on the programmer's intent or ability to correct the mistake.

Brown and Altadmri [BA17] point out that not all mistakes cause a compiler error, like using "=" instead of "==". Brown highlights that the time to fix is higher for these types of errors because programmers do not necessarily realize they even exist. A token mapping approach could identify many of these errors that do not cause a compiler error, especially for high time to fix errors.

Brown and Altadmri [BA17] acknowledge that their automated error detection code can give false positives or false negatives in the case of unparseable code where they can not tell if the error has been fixed. A token mapping approach might increase the accuracy of the automated error detection code used by the authors by more specifically identifying and matching the specific errors. Behavioral analytic data may also provide predictive information to enabled educated guessing of typographical errors a programmer might have made.

2.3.2 Limitations of Language Research Methods

Programming language research aside from compiler error messages also suffers from limitations which could be relaxed or resolved with the language analytics platform. Following is a sampling of additional author-cited limitations accompanied by a brief discussion of how the issue could be overcome.

Denny et al. [DLRTH11b] documented the high prevalence of syntax errors among novices in programming and like Kummerfeld and Kay [KK03], pointed out that until the syntax errors in code are resolved, the programmer can not possibly even receive feedback on any logic areas. So while studying code submissions provides valuable insight into what mistakes novice programmers are making, there are severe limitations in understanding other issues that might be present and important patterns, behaviors or insights are being overlooked. This syntax wall could be lowered with language analytics and token mapping tools not only by assisting with syntax error correction, but in the early identification of non-syntax type errors that could be identified before a compiler can fully parse a programmer's code.

Denny et al. [DLRTH11a] highlights that there are generally multiple solutions to a problem. In their system, students were shown all passing correct answers after they solved the problem from a database developed by student submissions. This multiple solution concept could be enhanced by an automated behavioral and token analytics approach mining a database of correct and alternative solutions to find best matches.

Sleeman et al. [SPBK88] highlights that a major omission with many research studies on programming and programming errors is that they do not determine the nature of the errors associated with various constructs. Their response was to ascertain this information through structured interviews with students who had problems. This structured interview approach could clearly provide a certain type of additional information on a small study, but because of its time-intensive nature, it would not scale to levels such as the Blackbox dataset [BKMU14]. Some of the logic errors that the authors documented, such as semantically constrained reads, declaration order confusion, loop logic and scope errors could be identified by an automated analytics approach through matching token streams to incorrect solutions.

In their work evaluating the time to fix different types of syntax errors, Denny et al. [DLRT12] cite the following threat to the validity of their study results: if a student submission contains multiple errors, they do not know the student's strategy for correcting those errors (i.e. if the student attempts to fix all errors at once or one at a time). They attempt to investigate this by considering a subset of the data with only a single error message. This approach to identify student strategies is a valid good faith attempt to discover intent, however, a time-based behavioral and token analytics approach could identify the differences from one submission to another that would provide an empirical indication of what the programmer actually did, from which a researcher could draw inferences about what the programmers was attempting to fix specifically.

In the static code analyzer developed by Schorsch [Sch95] as teaching aide for novice programming students in Pascal, he mentions an example of a limitation of his pre-compiler approach in discovering certain types of logic errors. In a situation where evaluating whether a procedure parameter should be an OUT parameter or not, his program can not determine whether the parameter value is needed in the calling program because the purpose of the algorithm can not be determined statically. While the problem of identifying a purpose for an algorithm is difficult, it could be improved by a language analytics approach which could match the identifier in the calling code with its future usage in the remaining code and make a reasonable assumption about the intent of the programmer and purpose of the algorithm.

Kummerfeld and Kay [KK03] noted that they observed that students sometimes made "erratic alterations" to their code when they got stuck and tried to experiment to fix syntax errors, which they point out is consistent with other studies which had similar findings of "tinkering." This type of behavior is very difficult to identify empirically because there often is not specific tie to the last observed error. The erratic alteration issue may be manually observable first hand in a small 6 person study like they performed, but on a larger scale an automated language analytics approach could do a much more thorough job of identifying these situations.

2.3.3 Case in Point: Coding Practices

One example of how this may be applied to an identified problem area is exception handling. Kery, et al. [KLGM16] found empirical evidence of what they considered widespread poor programming practices in exception handling through a large scale code repository analysis of over 11.6 million Java `try/catch` exception handling blocks in public code repositories. The study documented the common practice of using minimal or empty handlers and the frequent use of Log, Print and Return statements in `catch` blocks. They also found that programmers tended to locally handle an exception or catch everything with `Exception` instead of propagating it by throwing an exception to the full program scope. The authors call for improved tool support to address the prevalent bad programming practice they documented in exception handling code. Good programming practice recommends that exceptions are sanitized to remove confidential information, like a filepath in an `IOException` or debugging information like a stack trace or information derived from caller inputs. [Ora17] Re-casting, in addition to purging certain information, is an important mechanism to hide implementation details and maintain abstraction.

Although there is support in Eclipse to provide assistance with aspects of this particular problem, it could be extended with token mapping. A token mapping tool could be used to identify bad practice and make suggestions for appropriate handling, re-casting and propagation techniques. It could identify an exception handling block and by examining the contents of the `try` block, provide an optimal mapping to best practice code and use the missing tokens to suggest a code structure for the `catch`. For example, a call to open a file in a `try` could check to see that an `IOEception` is handled in the `catch`. Security could be enhanced by checking the code further for sanitization in that case.

2.3.4 Case In Point: C++17

C++ is an International Standards Organization (ISO) standard which has been and continues to be revised periodically by committee. The next version of C++, C++17 reached the draft stage in March 2017 by unanimous approval with the final standard excepted before the end of 2017. The new standard has dozens of new features, modifications and deprecations. Since C++ is prevalent in systems programming, these changes potentially affect millions of programmers worldwide. Impacts include the cost of implementation of changes in various compilers, training programmers to know and use new features, retroactive adjustment of working code in cases where features changed and other similar costs.

Although the changes are debated publicly by committee, the lack of empirical study of the impact of proposed feature changes is like the United States Congress voting on a bill that has not been scored by the independent Congressional Budget Office. Better information on these language design decisions is necessary and should be a responsibility of the community *before* implementation. Many of the features could be tested simply with token based approaches to examine alternatives. For example, is `for_each_n` the best and most intuitive way to execute a parallel execution of a particular `for` loop or is there a better way?

2.4 Implications for Teaching and Learning

Implementing best teaching practices for computer programming requires knowledge of common errors. Denny et al. [DLRT12] claims "as educators, the more we understand about the nature of these errors and how students respond to them, the more effective our teaching can be." They demonstrate how their experimental results can lead to the conclusion that certain teaching interventions should be made to target specific situations for different types of syntax errors. Sleeman et al. [SPBK88] make the claim that understanding more about student syntax errors should serve an important role in improving programming instruction as well as providing insight into how students learn complex skills generally.

Kummerfeld and Kay [KK03] pointed out that syntax errors are one of the biggest barriers for novices and other researchers have affirmed that view [DLRC14, DLRT12, FCJ04, Bec16, SS13]. This is particularly problematic because fixing syntax errors is the first step in debugging and until a programmer can resolve any syntax problems and compile their code, they can not get any feedback on any logical issues that might be present. Many studies have established the difficulty for novices of writing syntactically correct code by documenting the frequency and types of common syntax errors. [BA17, DLRTH11b, FCJ04, DR10, SPBK88, Bec16] Becker [Bec16] summarizes the evidence to show the consistency of the ten most frequent Java errors across six independent research studies which ranged from 52% to 80% of the total errors and as well as the close matching of the ordering in those studies.

Researchers tend to agree that computer science teaching methods have room for improvement, especially in certain areas. As Hristova et al. [HMRM03] point out, the difficulties in complex languages like Java cause a variety of common errors in novices' code, and despite extensive coverage of these in textbooks and lectures, they continue to persist. Novices have difficulty understanding the intricacies inherent in the design of complex languages and so frequently have a hard time even identifying the true cause of their errors. Researchers have noted a lack of evidence in evaluating enhanced compiler feedback and the need to evaluate them in real classroom situations [DLRC14] as well as little research on how programmers approach syntax error correction. [KK03] Debugging has also identified as a particularly difficult task for novices to learn and for instructors to teach. [MFL+08, MLM+08] Carter and Bank [CB13] claim (based on their review of relevant literature) that debugging skills are not typically explicitly taught by instructors and so students learn through a process of trial and error.. Both Kummerfeld and Kay [KK03] and Flowers et al. [FCJ04] have noted the difficulty for novices of even understanding standard compiler errors in the first place, much less how to correct them, and Murphy [MLM+08] acknowledges that despite considerable study of debugging, there are no established best practices to guide instructors.

The contributions of a token accuracy approach to supplement existing measurement and analysis tools would be expected to help researchers identify causes and effects of these issues as well as provide us ways to empirically measure and quantify the impacts of different teaching and learning strategies. Identifying, categorizing and measuring student programming behavior could prove valuable to educators seeking to apply

different teaching interventions for different students based on patterns they exhibit in their programming. Professional training could be enhanced and even automated with tools designed to identify positive and negative behavior along with corrective suggestions. The first step on this path is developing these strategies as research tools so that eventually they can be developed as real time assistive technology tools.

Chapter 3

Literature Review

3.1 Compiler Errors

Compiler messages are a bigger problem for novices programmers because they don't have as much experience to draw on to recognize and correct the error as a professional, so they only have the compiler error messages as guidance for how to proceed [Bec16]. This syntax and compiler message barrier has been well documented in the literature, along with the time wasting and frustration that follows and many researchers have specifically identified that confusing compiler messages are a major problem for novices. [HMRM03, PPM+17, DR10, NPM08, FCJ04]

Interestingly, when Stanford made the switch from using Pascal to C as the language of instruction for their CS1/CS2 courses in 1991, they found that after the first year, as expected, students were spending a higher portion of their time correcting syntax errors and their frustration level was higher than before. After they studied the matter more closely, however, they made the observation that student frustration was more related to the programming environment than the language. The Pascal compiler they had previously used was better at error detection and debugging. As a result, they developed a programming environment specifically targeted to introductory student use. [FR96] Schorsch had made a similar assessment earlier at the United States Air Force Academy and developed the CAP programming environment for students. [Sch95] Both of these tools, however, recognized that the most important issue was the inadequate, uninformative or misleading error messages of commercial compilers, which are tolerable for experts but not for novices.

Kummerfeld and Kay [KK03] observed that students were not paying close attention to the compiler messages themselves and hypothesized that it may be because they found them incomprehensible. This observation was confirmed by Flowers et al [FCJ04] who found students often confused by messages and sometimes even inferred that the compiler was broken when they received an error message that the compiler could not resolve a symbol for a misspelled identifier.

Kummereld and Kay also note that syntax errors are more significant for beginners than for experienced programmers who can generally spot and correct them quickly. In a study on beginning C language students, they provided a reference guide with a catalog of common syntax errors identified by the compiler. In the guide, the error was explained and highlighted and was presented along with at least one solution. They performed a limited scale qualitative study to evaluate the effectiveness of the guide as well as to understand how syntax error correction might be taught. The evidence they gathered did suggest that providing examples of similar problems and solutions could be helpful for novices in solving syntax problems.

These results were expanded on by Denny et al., [DLRTH11b] who examined the frequency of programming errors among novices in order to understand how big of a barrier syntax is to learning programming. They observed the correctness of code submitted by students on short exercises in an introductory Java course. The goal was to analyze the nature of the submissions in order to improve their teaching practice. They expressed surprise to find that even the students who finished in the top quartile of the class ranking had non-compilable syntax errors in almost half of the code submissions on short exercises where the median lines of code was 8. Furthermore, approximately 70% of the students submitted non-compiling code at least four times in a row. The overall conclusion is that syntax is a major barrier to all novices, even the strongest ones.

Follow on work from Denny et al. [DLRT12] provides evidence not just on the frequency of syntax errors by novices but on the time to solve different types of errors. They confirm their earlier results on the frequency of common errors but also compare the time to resolve the errors for quartile groups of students based on their class ranking as a proxy for skill level. They found that certain types of errors were easier for higher skill students in the top quartile to resolve, but that in general all students spent a similar amount of time solving the most common errors no matter which level they were in. This finding was also noted by Flowers et al. [FCJ04] who noted that even the best students make the same simple errors late into the semester. Denny suggests that the surprising result that the time to resolve the two most common errors did not vary by quartile "indicates that specific teaching support around the causes of these errors may be particularly effective."

In addition to being a significant barrier and time sink for novice programmers (and for teachers who have to explain the same things repeatedly), syntax problems can cause long-term detriment to programmers who can become disheartened by repeated failure and frustration and give up on the problems all together. This "give up" scenario has been observed by numerous researchers, including Kummerfeld and Kay [KK03], Denny et al. [DLRTH11b] and [ref]

Compiler errors for professionals tend to follow a similar pattern to novices in that a large percentage of the errors are concentrated to a small number of different types. In the previously mentioned Google study of 26.6 million build errors with 18,000 developers over nine months [SSE+14] researchers found that for Java code the top five errors accounted for 80% of the total errors reported with the number one most

common error appearing in 43.3% of the errors. Independent of the programming language (Java or C++) 10% of the error types account for 90% of the build failures. The type of errors observed were primarily dependency-related for the professionals instead of syntax-related like novices. They also not surprisingly observed that the number of compiles a professional took to fix was lower than what has been observed with novices in other studies. The median number of build attempts until the errors are resolved was 1 and 75% of the build errors were resolved within at most two builds for all of the 25 most common error kinds for both Java and C++.

3.2 Tools and Approaches for Enhanced Error Messages

One of the major issues with compiler messages aside from the language ised itself is the ambiguity of identifying the actual error in the code since multiple errors are frequently presented with the same error message. This problem is unavoidable from the compiler's perspective because of limitations of compiling. As a result, many researchers have looked at compiler messages generated by various code samples and tried to generate strategies for making these messages more understandable for novices and as mechanisms to improve teaching. Here will will review the techniques taken by these researchers to understand the successes and limitations they have disovered to guide the development of the token mapping technique.

Pre-compiler approaches

The Code Analyzer for Pascal Tool (CAP) [Sch95] was developed by Schorsch as a static code analyzer to perform diagnostic checks on student code to diagnose the most common error messages and coding errors they observed in syntax, logic and style. The CAP program uses a pre-compiler approach that identifies and recovers from all syntax errors in attempting to parse a student's code. CAP forces students to focus by only reporting the first error of a cascading sequence (an approach of limiting error messages to promote student focus also noted as effective by Denny. [DLRC14], Becker [Bec16] and Kolling [KQPR03]. This was accomplished by correcting the problem in the pre-compiler's internal symbol table, which is similar in effect to what a token mapping would achieve on an individual token. The CAP tool is a recursive descent compiler using wide tokens (which include source code row, column and comment information) in order to attempt to ascertain the intent of the programmer. As an aside the CAP and Gauntlet tools were well received by instructors who used them and identified in surveys that the tools significantly reduced their grading times and office hour burden.

The Expresso tool developed by Hristova et al. [HMRM03] is another example of a pre-compiler to identify common Java errors and provide enhanced messaging to avoid cryptic compiler error messages for novice programmers. They assembled their common error list with survey data of faculty and student teaching assistants and categorized the errors by syntax, semantic and logical errors. The implementation of their system was a multiple-pass pre-processor which first cleaned and prepped the students code, tokenizes the

input stream and then detected the mistakes and generated a printout by line number. The specific detection algorithm was not identified but the token matching approach to code analysis is similar to what is proposed in this research plan.

Post-compiler approaches

Denny et al. developed an instructional tool they called CodeWrite [DLRTH11a] for drill and practice of programming and program testing. They did this by creating a recognizer that parsed the programmer's code as well as the raw compiler messages so they could categorize the errors by type. In their sample of 12,369 code submissions, they could correctly classify about 78% of the errors, but the compiler message was was not sufficient to distinguish other types of errors. The key issue was properly identifying more contextual feedback. They responded to this by performing static analysis with regular expressions, which helped them identify an additional 14% of the total errors. This approach has similarities to a token mapping technique but more restricted because of limitations of regular expressions.

The Gauntlet[FCJ04] tool was developed by instructors at the United States Military Academy to respond to patterns of difficulty encountered by freshman in a mandatory Information Technology Course where they learned fundamental programming skills in Java. The instructors observed that student repeatedly made the same mistakes, did not understand error messages and often wasted hours of time on simple errors. Their response was to catalog the top fifty student programming errors and then build a static code analyzer (pre-compiler) to catch and explain the errors in layman's terms. Their tool also finds many common novice semantic errors that are not syntax errors. They describe their system as an expert system which provides the students expertise and advice they would normally receive from their teacher. The system itself does not do anything to narrow in on the type of error, for which a token mapping approach could be useful, but instead provides advice on various possible causes of the error and suggests possible solutions.

Dy and Rodrigo [DR10] present a detector for non-literal java errors, which they define as errors where the compiler-reported error does not match the actual error. The catalog of non-literal errors were assembled by manually inspecting submitted code with compiler errors that was assembled from a plugin to the BlueJ system. They programmed a detector using a post-compiler approach that uses the compiler errors, line number of the error and the code submission and trys to match this against patterns in their catalog of non-literal errors. Their paper did not provide additional information on the precise mechanism they use to perform the matching so the technique is unknown, but seems like some form of search ranking algorithm. The authors identify two major shortcomings of their system, including the inability of the system to determine programmer intent and the limited catalog of non-literal errors which they subjectively determined through their manual inspection of code samples. The approach of matching code samples against patterns is similar conceptually to the token mapping technique.

Becker [Bec16] developed a post-compiler tool called Decaf which analyzes a student's Java source code and presents the standard javac output alongside an enhanced compiler message entitled "Translated" output. The system recognizes 30 common error messages and presents the user with only the first error. He did not provide any detail on the specific technique he used to identify what enhanced message to display to the user based on the error. He presents an example of a "cannot find symbol" error resulting from a `myString.length` call, which provided an interpreted message and suggested correction with `myString.length()`. This message indicates that the tool is customizing responses to the user's code instead of just providing generic enhanced messages, but it is unclear to what extent this is using any unique matching logic or just a code completion-type heuristic. He provided empirical evidence with a control group showing a positive impact of different students using the tool across two semesters.

Rigby and Thompson [RT05] tested an Eclipse IDE plug-in called Gild [SDM+03] with first-year students learning to program Java on a small sample of 10 students with inconclusive results. The plug-in provided many features to customize the presentation of Eclipse including simplifying the user interface and enhancing help and instruction features with a goal of better teaching and learning Java. One of these features was a set of enhanced compiler messages for 51 common novice errors [Uni17b]. There was no description of the mechanism for supplying these enhanced messages and the feature report describes it as links "extra" help messages so these were presumably not customized based on code analysis.

3.2.1 Interpreter

Freund and Robers of the Stanford Computer Science Department took a different approach to the issue of providing enhanced compiler errors to novice students in the development of their Thetis [FR96] programming environment for ANSI C by developing an interpreter instead of a compiler. The logic was that students were better served by trading off the run-time efficiency of compiled code and execution speed for simplicity and debugging features. Although the goal was more than just enhanced compiler messaging, this was listed as the first problem they wanted to address and one of the main differences between Thetis and other IDEs. The approach to providing enhanced messaging was primarily to enhance the verbosity of messages like "type mismatch" to "cannot assign a value of type **string** to a variable of type **char**." They also added syntactic restrictions not present in standard C to prevent common errors, such as an assignment inside a conditional expression instead of a relational operator, such as `if (i = 0)`. The interpreter approach allowed them to perform a substantial amount of run-time checking that was not syntax related. For example, they could identify errors such as: dividing by zero, using an uninitialized variable, dereferencing or freeing an invalid pointer, array out of bounds, out of range value to an enumerated type, exiting a non-void function without returning a value, bad function pointer or passing invalid arguments to a function. A token mapping approach could be used to identify several of these run-time problems prior to execution, but not all of them.

Similar to the CAP and Gauntlet tools, Thetis was popular with instructors because of reduced grading and debugging time and as a live teaching tool in discussion groups and office hours.

3.2.2 Intelligent Tutoring Systems

Johnson and Soloway [JS85] developed a tutoring system at Yale for Pascal that assesses a student's "buggy" code and maps it against a knowledge base of programming plans and strategies in order to generate an instructional path for the student to learn how to fix the problems. The systems consists of two components, a programming expert (which analyzes the program) and a pedagogical expert (which performs the instruction). The primary focus of their paper was on the programming component which relied on building a database of known programmer errors and misconceptions. The key element of this process is forming a hypothesis of the programmer's intention by analyzing the code submission. The authors developed the knowledge base using programming plans, which they posit that experts use extensively in programming. The concept is based on a psychological theory of the programming process, which relies on plan recognition to interpret understanding. An example plan is a RUNNING TOTAL VARIABLE PLAN to generate a result in a stored variable. The plan initializes a variable and carries information on the context in which it should appear, such as a loop. The concept will bear some similarity to the token approach and mapping against a database of possibilities and is a useful guide because of its theory of determining programmer intention.

Truong et al. [TRB04] developed a static code analysis system at Queensland University of Technology initially to automate grading and feedback for students in an introductory Java class motivated by large class sizes and the desire to increase their emphasis on teaching program design methodology which they noted in the literature. Their contribution was a tutoring and grading system that analyzed a student program for quality, generated ideas for alternative solutions and offered hints. The system was designed to be highly configurable and extensible to be used for different exercises. The concept of informative and immediate feedback and correct solutions was designed to address student misconceptions. The primary approach to their system was to examine source code, parse it into an AST tree and then compare that structure against model solutions for given problems. They also place an emphasis on software metrics to measure student performance and evaluate program quality and adherence to good programming practices. They noted that software metrics not usually adopted by the existing similar systems and instead used primarily for marking and plagiarism detection instead of teaching and feedback.

Carter and Blank [CB13] describe an intelligent tutoring system under development which uses Case Based Reasoning (CBR) to represent and reason about errors in a student's code. The CBR consults four database tables with syntax, runtime and logic issues and one representing the overall case base. Cases are identified by main and subtype, difficulty level, id, usage count and suggested steps to a solution and error symptoms. The system intends to provide more descriptive information to supplement the compiler and runtime system

feedback as well as visualizations for reinforcement. A historical tracking system will reason about knowledge gaps for the students based on patterns of previous errors. Carter and Blank [CB14] performed an evaluation of the system on a pretest-practice-posttest methodology in 2014 without a control group on a small sample of 12 college students which did not show a significant difference between the pretest and posttest results (the actual statistical results were not reported in the paper). Carter [Car15] subsequently performed an additional larger evaluation of high school students and did demonstrate a statistically significant improvement in the posttest scores. There were some methodological mistakes in the study design, however, because the participating students self-selected into the study and there was no control group to indicate if the system provided any benefit or if the exercise of practicing itself was the cause of the higher scores. In any event the approach to providing context and suggestive assistance is similar to Denny [DLRTH11a] using a different algorithmic approach.

3.2.3 Recommender Systems

Recommender systems are often used in situations where users either do not have the experience or time to sort through large amounts of data quickly to make a decision. A recommender system gathers and analyzes data and supports user decision making by providing recommendations based on a user profile. They have been applied in various commercial settings, such as Amazon.com's recommender system to help guide consumers on alternative or additional items they might want to purchase based on what they search for or view.

There has been a line of research recently proposing the use of recommender systems as integrated development environment extensions for software engineering. Robillard, Walker and Zimmerman [RWZ10] argue in support of their opinion that recommendation systems for software engineering are "ready to become part of industrial software developers' toolboxes" to help with everything from reusing code, to navigating large code bases and class libraries to writing effective bug reports. Ponzanellii et al. [PBDP$^+$14] describe what they claim is a novel approach to mine StackOverflow comments given a context in an IDE to notify the use of available help. Their recommender system evaluates the relevance of StackOvervflow discussions taking into consideration code aspects (code clones and type matching), conceptual aspects (text similarity)and community aspects (user reputation) to decide if a StackOverflow discussion meets a threshold that generates a user notification. The code context is determined by the package/class/method name, the source code being modified, the types of the API used and the names of API methods. Queries are generated from the context and used to retrieve a list of possible StackOverflow discussions. The results are then ranked by similarity in text, code, API types, API methods and various StackOverflow specific items (like scores and reputations). This type of a recommender system could be used in a token mapping system by matching a users token stream against a library of possible token maps for correct and incorrect solutions. The ranking

could be used to identify a best match and possible understanding of user behavior or corrective action in debugging, syntax correction and even logical or complex use cases.

Hartmann et al. [HMBK10] developed a social recommender system for use in computer science instruction at the University of California, Berkeley and Stanford. The system provided enhanced error feedback by querying a database containing a selection of top errors and providing feedback from other users on how they have corrected similar errors previously. The errors selected for inclusion in the database were created initially by experts instead of user submission logs like most other studies, however the list is dynamic with a list of actual student bugs contributing. One novelty in their system is a voting system by students to bump solutions based on their experience. Their quantitative data indicated that 47% of errors had useful fix suggestions in their database. The main difference between their approach and a StackOverflow approach is that they use code semantics instead of string literals for matching. In particular, their database contains code semantics not just for working code samples but for broken code. It also contains instrumentation for tracking code evolution over time. The database is built up by tracking code changes whenever an error is fixed by capturing a diff report on the code before and after the fix. They also use a progress heuristic for identifying a subset of runtime errors. The matching algorithm uses an approach similar to token mapping system envisioned in this proposal where the code is run through a lexical analyzer and analyzed. They compare strings for similarity using a simpler algorithm than we propose which is the Python `difflib` ratio of matched to unmatched tokens. They suggest that other techniques for similarity detection could be substituted. A primary difference in our approach aside from the social aspect would be an attempt to identify error patterns and a path to correction over time in a model that could be built up with machine learning. Additionally we will examine patterns in blocks of code and not just on a line by line basis.

Debugging Behavior

Several studies have examined patterns and strategies that programmers use to debug programs. The study of debugging behavior and the thought process of programmers during debugging is interesting because it is considered a difficult and somewhat distinct skill from general programming ability. Murphy et al.. [MWB99] explain that there is a need to develop best practices for instructors on how teach debugging because there is not that information typically in text books about it. In an attempt to provide evident of debugging behavior, they performed a qualitative study on 21 CS2 novice java programmers to identify the “good, bad and quirky” debugging strategies they used to fix one of six typical CS1-level programming assignments. They observed that students used tracing, commenting out code, diagnostic print statements, methodical testing, debuggers, online resources and pattern matching with varying degrees of success and good and bad habits.

Think Aloud Studies

The purpose of a think aloud study format in a computer science application is typically to understand strategies and thought processes used by students to solve problems or debug programs. Think aloud studies usually follow strict protocols developed by Ericsson and Simon [ES93] initially for cognitive research studies in psychology to elicit subject verbalization of their thoughts and behaviors while performing cognitive tasks. The protocols are designed to minimize the cognitive effort of their verbalization to allow a subject to focus on the problem being observed as well as to prevent bias. One of the most valuable aspects of information provided by this study format is data about the sequence of events that occur while a subject solves a problem.

Teague et al. [TCAL13] applied this technique to novice programmers at two universities in Australia in an effort to understand the nature of difficulties faced by the programmers in tracing and explaining existing computer code. Their motivation was to understand the nature of reasoning that the students exhibited in order to debunk three common perspectives of why students typically struggle, including program misconceptions, misunderstanding of requirements and poor self expression. They used the think aloud results to provide direct observational evidence refuting these explanations and to generate their own theory that many of the problems were due to the level of reasoning students were using. The result has implications for teaching strategies which can be developed to identify issues and assist students who exhibit them. They point out that the process of a think aloud method can be difficult for a subject to follow, especially a novice suffering from cognitive overload, which suggests that an automated approach using token mapping could improve data quality and make different subjects data more comparable.

Yuen [Yue07] performed a think aloud study on novice CS1 students in an effort to understand how they construct knowledge for complex concepts. His motivation was to understand the level of student knowledge and how they came to acquire it in order to develop instructional aids to combat high dropout and failure rates in the CS1 course at the University of Texas at Austin. His study involved eight undergraduate students from a summer session of CS1 targeted at non-computer science majors who solve three problems while thinking aloud. He states that the goal of cognitive studies is to develop a model of cognitive processes and then map a subject's thought and actions to the model. The study results indicated students exhibited three types of behavior in their knowledge construction ranging from the least preferable "need to code" right away by trial and error, to generalizing the problem from previous knowledge, to the most desirable behavior of designing efficient solutions where a student displayed that they internalized conceptual knowledge.

Whalley and Kasto [WK14] performed a qualitative think aloud study on novice programmers code writing strategies using six students who completed three programming tasks of increasing conceptual difficulty based on knowledge taught in an introductory course they were taking. They documented the subjects learning progression and approaches to learning to obtain insight into the cognitive processes in learning computer programming. They observed similar categories of behavior in learning as other researchers including those

who give up, tinkerers who experiment more randomly and those who solve the problem and move on. An interesting contribution of this paper which would be observable by token mapping is the changing behavior of students during the progression of the tasks as they gained knowledge and experience. They also note the particular difficulty of novice programmers in performing a think aloud study because the programming itself tends to be such a demanding cognitive task.

Prather et al. [PPM+17] performed a think aloud study on 31 students in the middle of a CS1 course where students were observed completing a timed quiz to solve a programming problem in their enhanced compiler message program Athene. The primary goal was to observe the student reactions to compiler messages, especially including the enhanced messages they provided. One of their objectives was to gather evidence on whether students even read the enhanced messages and which messages were most helpful. This think aloud study was an add-on to the quantitative aspect of the study in order to classify trends observed in the quantitative data. Token mapping data, especially over time, is expected to provide similar benefits to improve understanding results of programming language studies and generating theories and explanations for future study.

3.2.4 Productivity of IDEs vs Text Editors

Several researchers have suggested that using Integrated Development Environments to provide assistive tools for programmers could be beneficial to novices in resolving syntax errors. Kummerfeld and Kay [KK03] comment that it "may be suggested that use of Integrated Development Environments would improve syntax error correction through the use of templates and/or intelligent IDE", however, they conclude that most errors that they observed would still be difficult to diagnose/prevent. The Thetis [FR96] tool from Freund was a full programming environment designed specifically for novice students. Gild [SDM+03] was an effort to reduce the complexity of the typical full featured Eclipse IDE for novices so that they can have the benefit of the most important features like debugging tools and enhanced hints.

The analysis of build errors at Google by Seo et al. [SSE+14] provides some interesting evidence regarding the possible impact of IDE usage in programming productivity. In analyzing 26.6 million builds by 18,000 developers over a nine month period, the researchers observed some interesting effects potentially attributable at least partially to IDE usage. They examined build events for two languages: Java and C++ and compared various observable metrics as well as the tool usage by developers in each of these languages. They found that nine out of ten Java developers used an IDE such as Eclipse or IntelliJ IDEA to program while eight out of ten C++ developers just used a text editor such as Emacs or Vim to program. Of the total number of build failures observed, the median percentage per developer was 38.4% for C++ and 28.5% for Java. Although, the most common build failures for both languages were dependency related (52.7% for C++ and 64.7% for Java), the percentage of simpler syntax errors was higher for C++ than Java (detailed results not reported but a graph suggests approximately 12% for C++ and 4% for Java). This result may be attributable to

syntax differences in the languages themselves, however, the authors believed that the built in error checking of the IDE was a contributing factor to the trend.

3.3 Parallelism and High Performance Computing

The problems of learning programming is not limited to novices. Professional programmers and scientists often encounter situations where they return at least partially to a "novice" level when using new libraries, programming paradigms or unfamiliar languages. A common contemporary example of this is with High Performance Computing (HPC) or parallel computing, where complex syntax, libraries and concepts can frustrate even highly experienced programmers who either have limited experience in the area or quickly become out of practice.

The scientific community is acutely aware of the difficulty level of programming high performance applications by non-computer scientists [Wor17] [ABC+06], however there is a general lack of empirical research on the specific causes and possible solutions for this problem [Kai15]. The computer science community and hardware manufacturers have offered numerous paradigms and approaches to solve the highly complex problems presented by concurrency and performance mapping for various hardware, but the solutions are often so complex that they create an obstacle for scientists productivity in applied disciplines.

Concurrency in programming has broad applicability across large-scale commercial enterprises, internet searching, scientific computing, modeling and transaction processing so it is important for the future of computer science education. Teaching complex computer programming concepts, such as concurrency, can be improved if educators have an idea of the specific impacts different techniques have on students. Understanding the precise nature and type of students' mistakes while learning to program for concurrent systems is important in designing teaching practices.

Educators sometimes assume they know the problems their students face, but Brown and Altadmri [BA17] showed that even experts in computer science education often have holes in their understanding of the impact of programming languages or errors. They examined whether 76 educators' opinions of common programming mistakes matched actual observations from their Blackbox [BKMU14] data collection project with over 100 million compilations of students learning to program Java. The results showed that the educators' estimates on frequency and time-to-fix of the 18 most common errors in the database did not exhibit a consensus among the educators themselves nor did the estimates match the actual data. Computer science educators can benefit from this research since there are some popular educational programming products available today, such as Alice [ali17], which use concurrency elements like 'Do Together' and 'For all together' and Scratch [scr17], which allows for parallel execution of threads by launching two stacks. The community can benefit from knowledge about which paradigms are easier for humans to use.

Empirical study of programmer productivity has begun to produce quantifiable measurements of the differences between approaches to programming and learning to program. Some more recent studies are beginning to look at measuring the process of programming itself instead of just compiler errors. Price et al. [PBL+16] looked empirically at programmer productivity using a Frame-based programming editor compared to the original Java editor by comparing time and number of edits as well as overall progress.

3.3.1 Threads vs Processes

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 in contrast to a Process-based approach to parallelism where the fundamental approach 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. Threads tend to be lower level and map to a multicore hardware architecture with processors capable of multiple threads of execution. Process paradigms tend to be higher level with the emphasis on the communication of processes.

We selected these two paradigms for examination primarily because of their very different approaches to solving concurrency problems. The Threads-based paradigm is the most well-known and widely used model and serves as a useful control group because of its prevalence in system software and its use in common programming languages like C, C++, C#, Java and Python. The programming language Go was created at Google by engineers seeking to create a new language which "by its design ... proposes an approach for the construction of system software on multicore machines." The language development was motivated because no new systems language has been developed in over a decade to address the multicore computing landscape that exists today. [gol17b] The engineers built the concurrency primitives of the language on Hoare's Communicating Sequential Processes (CSP) [Hoa78, Hoa85] because it is "one of the most successful models for providing high-level linguistic support for concurrency." [gol17a] Occam-π [pu17] and Erlang [erl17] and JCSP [WAF02, WBM+07] are other languages and libraries inspired by the process-oriented approach of CSP.

3.3.2 Concepts of Parallelism

The problem that parallel programming is "hard" seems to be widely accepted and expressed in many abstracts and book introductions in academic and popular literature, but the statements are generally qualitative and expressed as obvious to the reader. Intuitively this seems true because of the additional considerations and complexities in parallel programming that are not required in sequential programming, but there does not appear to be any evidence in the literature to quantify or support the relative difficulty of parallel programming compared to sequential programming in general.

Lewandowski, et. al. [LBM+07] did examine the intuitive complexity of general parallel concepts involving race conditions and locking by asking non-programmer students to describe in plain English what problems might be encountered in a scenario involving multiple parties attempting to purchase a fixed number of tickets in an online system. He found that 97% of the students could conceptually identify a race condition and that 73% identified at least one possible solution. This seems to indicate that at least the high level concepts of parallelism may be naturally intuitive to people.

Rossbach et al. [RHW10b] conducted an empirical study by examining the results of programming assignments for 237 students and 1,323 programs in an operating systems course to identify and classify the types of synchronization errors that students had, as well as their frequency. In total, the authors identified ten types of errors that occurred and found that the number and types of programming errors were lower for transactions (10% of students) than locks (70% of students) on a similar programming assignment.

Although human factors research in programming languages is still sparse and relatively young as described by Kaijanaho [Kai15], there are some studies examining the area of parallel programming. The main focus area within parallelism has been Software Transactional Memory (STM) where Kaijanaho identified 4 RCTs (randomized controlled trials) [RHW10b, PAT11, COS11, NTPM11] which compared STM to other traditional course and fine-grained locking mechanisms. The non-RCT empirical study conducted by Coblenz [CSM+15] on the use of *reducer* functions (functions to perform a summary operation over all data) in a parallel programming context compared OpenMP to Cilk Plus in a small masters level course. Part of this study provides a further examination of that issue specifically as it relates to offload computation in coprocessor architectures.

3.3.3 GPU Programming

The scientific community is acutely aware of the difficulty level of programming high performance applications by non-computer scientists [Wor17] [ABC+06], however there is a general lack of empirical research on the topic [Kai15]. Various paradigms and approaches to solve the highly complex problems have been offered, but the solutions are often so complex themselves that they create an obstacle for scientific discovery in applied disciplines. Learning GPU programming is important because it provides a relatively low cost approach to massively parallel computing as a core technology in many of the world's fastest and most energy-efficient supercomputer clusters. Storti and Yurtoglu explain that HPC hardware performance measurement has evolved from FLOPS (floating point operations per second) to FLOPS/watt and GPU-based parallelism competes well in terms of FLOPS/watt [SY15]. They make further claims that GPU-based parallel computing can reduce development time by orders of magnitude and that these gains can be achieved at reasonable development and hardware costs. In advocating why developers should learn CUDA, they argue that it is currently the best-supported and most accessible platform for GPU-based parallel computing. By contrast, Thrust is a high level interface to CUDA similar in style to the C++ Standard Template

Library (STL). It is fully interoperable with CUDA and provides a collection of data parallel primitives to allow a high level of abstraction to CUDA. We can imagine some scholars, developers or students, given this difference in abstraction, might assume that this would make it easier to use and thus plausibly improve programmer productivity. However, we are aware of no evidence of such a claim, in either direction, for this kind of programming and evaluate the claim in this work.

The issue of GPU programming complexity specifically has also been widely discussed, not just from the issue of the complexity of classic massively parallel programming issues like data races, locking and thread management, but also specifically related to GPU hardware. GPU programming is important because it provides a relatively low cost approach to massively parallel computing. Bourgoin, et al. [BCL17] identify the primary reasons for the complexity of GP-GPU programming in the two main frameworks, CUDA or OpenCL, an open standard from the Khronos Group [Gro17] as i.) the low-level nature of the programming, ii.) the need to write programs as a combination of two-subprograms and iii.) manually managing devices and memory transfers. They provide an explanation of how statically-typed languages, compilers and libraries can provide higher level abstraction to reduce complexity using the programming language OCaml as well as provide compiler time checking of certain types of errors that are only detected at runtime by the frameworks in order to make debugging easier.

Ueng et al. [ULBH08] claim CUDA is an attempt by NVIDIA to make programming many-core GPUs more accessible to programmers but that there are still many burdens placed upon the programmer to maximize performance. They focus specifically on the complexity of the burden of dealing with the complex memory hierarchy of the GPU in obtaining performance optimization ranging from 2-17x. The paper describes a transformation tool they created to automate to allocation, management and data transfers within the GPU hardware itself.

Thrust, on the other hand is a high level interface to CUDA based on the C++ Standard Template Library (STL). It is fully interoperable with CUDA and provides a collection of data parallel primitives to allow a high level of abstraction to CUDA which in theory makes it easier to use and thus plausibly improves programmer productivity. This study seeks to validate these claims and to quantify any productivity difference between the two approaches to GPU computing. If a measurable benefit can be observed, not only could there be an economic and productivity impact for HPC GPU users, but the specific benefits could potentially be used for programming language design decisions in other parallel computing contexts or to create beneficial abstractions for other parallel computing models. The benefit to scientific discovery of such improvements would likely be wide ranging.

3.3.4 Empirical Literature on Concurrency

Kaijanaho [Kai15] conducted a review of empirical studies up to 2012 that utilize human factors evidence in programming language design and found 156 papers that compared programming language designs with hu-

man factors measures. 35 of the papers used controlled experiments and 22 of these using formal randomized trials, a typical standard [U.S10] for scientific research. Several of these empirical studies relate to concurrency, [RHW10b, PAT11, COS11] primarily on Software Transactional Memory (STM). These studies used college and masters level student participants to identify programmer performance (in development time) and error rates using STM compared to Threads/Locks-based solutions which generally support lower error rates using STM approaches compared to locking. Another empirical study by Coblenz [CSM+15] compared the C/C++ language extensions Cilk Plus to OpenMP using students performance on tasks using *reducers* for concurrency in a graduate course, which provided limited data due to a small study size of 8 students, but suggested usability issues with both approaches. The researchers conclude that new instructional techniques or tools are needed to improve the students' performance on these tasks. Other systematic literature reviews identifying empirical studies in software engineering generally include those performed by Kitchenham et al. [KPBB+09] and Zhang et al. [ZBT11] make no specific mention of papers on concurrency.

3.3.5 Research Standards in Computer Science

In our overall program of research, we are seeking to develop methods, data capture and measurement tools and analytical techniques to contribute to evidence-based theory development in computer science research. These methods are widely used in the medical research community after the work of Austin Bradford Hill [DH50] in developing the Bradford Hill criteria for causality models in randomized clinical trials. The medical community now conducts tens of thousands of RCTs annually and has developed and adopted consolidated standards for reporting trial results (CONSORT) [MSA+01]. The fields of psychology and education have also developed an extensive body of literature on human subjects-based experimental research methods, some of which are applicable to computer science studies. The challenges those fields faced and the advances they have been made in experimental study design provide useful lessons to our field as we seek to develop our own evidence standards for causality, internal and external validity and integrity.

The specific hypothesis we examined in this research paper on GPU programming is based on an intuitive conception (and marketing claim) that a higher level abstraction paradigm for programming a difficult concept will be "easier" to learn and use and will therefore improve a programmer's productivity. Since the use of evidence-based theory development is in its infancy in computer science, we have not found commonly cited existing theories suitable for application to this type of predictive hypothesis testing in this evolving field. Finally, we want to stress that companies make marketing claims about programming productivity regularly, but they are rarely, if ever, evaluated in rigorous ways. As such, we remind the reader that the current authors' goals are to evaluate the claims made about GPU programming objectively, but that we have no vested interest in the outcome of this study.

Chapter 4

Extending Automated Token Accuracy Mapping Through a Re-Analysis

4.1 Introduction

This chapter describes a re-analysis of the data I collected for my Masters project and Thesis called "Empirical Study of Concurrent Programming Paradigms." [Dal16] The revised work was recently accepted for publication by *ACM Transactions on Computing Educaton* under the title "Analysis of a Randomized Controlled Trial of Student Performance in Parallel Programming using a new Measurement Technique." [Dal16] Appendix A contains the Methods section of the paper in press along with experimental materials for reference. In this chapter, we will investigate the following research question:

RQ1: Can the Token Accuracy Map technique provide useful information in identifying the specific problem areas and common types of accuracy errors in parallel programming?

A goal of the chapter is to further develop a supplemental empirical analysis technique to learn whether useful information can be gathered on the specific problem areas student programmers experienced in learning the alternative paradigms. This research question does not lend itself to empirical measurement so we do not offer a testable hypothesis for the secondary investigation, but we address the potential usefulness of the technique in detail in the Discussion section.

In the re-analysis work post-thesis, the entire token accuracy matching algorithm was re-worked and optimized and then manually re-scored as a cross check for accuracy. The revised tool generated more accurate token mapping and led directly to the localized token signature development. The unexpected extensive peer review during the publishing process led to the need to work through and resolve various shortcomings of the original more simplistic approach. Through an extensive, thoughtful and careful re-examination of the limitations of the first version of the token accuracy map, we developed a detailed understanding of the particular areas where token-based analysis would be well-suited. Fortunately, the detailed data for the project was collected through the automated testing system that I created for the thesis, so I was able to go back to the original source data and analyze it again as-new but from a fresh and more sophisticated perspective.

4.1.1 Token Accuracy Maps

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 and Siebert [SS13] and later automated by Daleiden. [Dal16]. The basic concept of the TAM is to parse a code sample into a sequence of tokens using a lexer, then to align the tokens to a correct solution and then to compare the two token arrays to determine the percentage of correctness in the overall response. In addition to an overall accuracy score for the code sample, the technique yields detailed data on the accuracy rate of individual tokens and groups of tokens in the participants' code, which can be analyzed separately in the full Token Accuracy Map. Examining the patterns in this data allows us to draw inferences about both the overall impact of the paradigm and specific elements.

In order to automatically score participant results, we implemented the Needleman-Wunsch [NW70] pairwise sequence alignment algorithm on the token arrays. The Needleman-Wunsch algorithm is a dynamic programming algorithm based on the longest common substring algorithm often used in DNA sequencing. The Needleman-Wunsch algorithm is a global alignment approach that analyzes and aligns every token in the sequence. We used the standard weighting for mismatches (-1) and insertions/deletions (-2) and matches (+1) in order to maximize the similarity of the global alignment. The Needleman-Wunsch algorithm is expensive computationally with a runtime proportional to product of the length of the two token strings being compared, but it is suitable for the number of tokens in our experiment.

The Token Accuracy Mapping approach 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.

4.1.2 Limitations of TAMs

Although TAMs can give us useful information about programmer accuracy, they have limitations. A limitation of concern is in scoring alternate semantically correct solutions. In our experiment, we compare the participant responses to a particular code solution. The first type of problem can occur with simple syntax differences between a response and the solution, such as variable names or non-material order variation. These can be fairly easily addressed through manual inspection and a comparison to a different valid solution or a manual reordering or renaming of the participant response. Before we completed our analysis, we performed this inspection and made these non-material syntax adjustments. Since the code descriptions were fairly specific and the code samples used to learn the paradigms formed a suggestive blueprint, the observed variation among participant responses were minor and only a few required any adjustments to improve the accuracy of the automated algorithm.

The second type of problem with TAMs is where the participant solves the problem in the task in a semantically correct, but alternative manner to the solution code. This situation could be handled by manually categorizing responses into groups and generating correct solutions for each approach. An overall accuracy score would still be valid in this case, however the token accuracy comparison would only be valid within the group. In our observed data, our manual inspection did not reveal this to be an issue in any of the responses, most likely due to the specificity of the instructions and code samples provided.

4.2 Results

4.2.1 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 (first and second year undergraduate students), 300 and 400 (third and fourth year undergraduate students) to 600 and 700 level (graduate students) courses. In the UNLV core curriculum in place at the time, the concept of processes and threads is introduced in an operating systems class typically taken during junior year so students at or above that level would be expected to have some type of conceptual mental model of parallelism. The emphasis of the instruction in these courses is not programming based however so they likely did not have previous knowledge of the implementation of a threaded or process-oriented program. UNLV uses C++ as the instruction language for its Computer Science I and II and Data Structures courses and there is little exposure to threading (or Java-style threading in particular). Participation in the study was voluntary and participants were rewarded with up to 3% extra credit based on what the course instructors decided was appropriate. Table 4.1 shows a breakdown of the participants with valid responses by position in the academic pipeline for each paradigm group and Table 4.2 shows the breakdown by gender for each paradigm group.

Grade	Process	Threads	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 4.1: Participants by Level in School.

Grade	Process	Threads	Total
Female	9	11	20
Male	35	31	66
No response	0	2	2
Total	44	44	88

Table 4.2: Participants by Gender.

4.2.2 Overall Accuracy Scores by Group

The mean accuracy scores by group (concurrency paradigm) and task are shown in Table 4.3 along with the number of code submissions (N), the minimum score (Min), the maximum score (Max) and the standard deviation of scores (SD). The results are shown graphically in Figure 4.1 with 95% confidence intervals. The Process group students had a higher average accuracy score on Task 1 (92.9% to 89.2%), about the same on Task 2 (75.9% to 76.3%) and lower on Task 3 (42.8% to 79.1%). The standard deviations were similar for each task with the largest difference on Task 1 (15.4% for Process and 20.2% for Threads). The graph shows the falloff in accuracy scores for Task 3 in the Process group compared to Task 2, while the Threads group performed about the same.

To analyze the data, we applied a Repeated Measures ANOVA using the statistical package SPSS. The model used the three overall task accuracy scores for within-subjects factors as dependent variables, reflecting the

Group	Task	N	Mean	Min	Max	SD
Process	1	44	92.9	30.8	100.0	15.4
Process	2	44	75.9	3.2	96.0	24.3
Process	3	44	42.8	2.9	65.7	12.8
All Process		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 4.3: Accuracy Score By Group and Task.

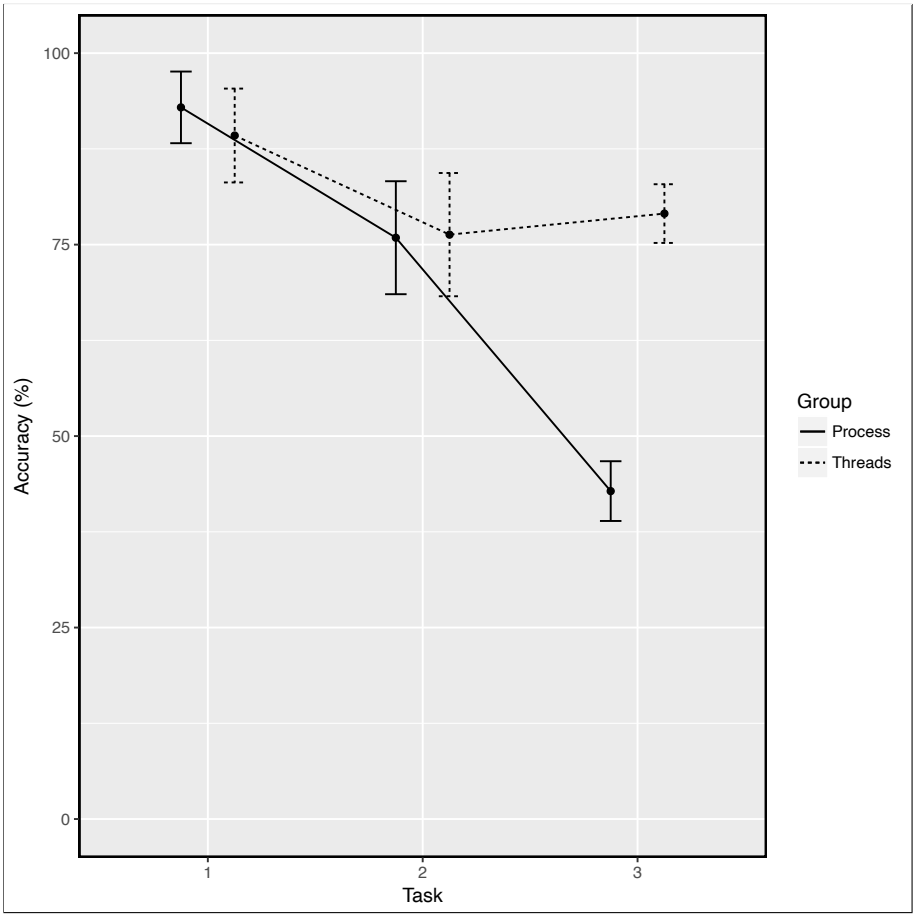


Figure 4.1: Accuracy Scores by Group.

sequence given to the participants. Two controlled between-subjects factors were examined together: Group and School Level. Additionally, we examined other demographic factors including gender, native language, age and self-reported programming experience as random factors and all other interactions up to 4 ways were checked and no other effects were significant.

SPSS automatically adjusted the degrees of freedom using the Greenhouse-Geisser score for the Repeated Measures ANOVA because the sphericity assumption was violated ($\chi^2(2) = 10.460, p = .005$). The results for the Within-Subject Effects of the Repeated Measures ANOVA shows that Task was a significant factor in overall score F(1.756, 53) = 27.356, p < .001, ($\eta_p^2 = .278$). There was a significant interaction between Tasks and Group in the results F(1.756, 53) = 18.279, p < .001, ($\eta_p^2 = .205$) indicating evidence in support of rejecting the null hypothesis, which states that the groups are equivalent. The interaction between Task and School Level was also significant F(10.537, 53) = 2.128, p = .025, ($\eta_p^2 = .152$). Table 4.4 shows the results of the Between-Subjects Effects tests of the Repeated Measures ANOVA. Group F(1, 53) = 5.674, p = .020 ($\eta_p^2 = .074$) was the only statistically significant result.

Factor	F	Sig.	df	η_p^2
Group	5.674	.020	1	.074
School Level	1.011	.425	6	.079

Table 4.4: Between Subjects Effects.

	Process			Threads		
Level In School	**N**	**Mean**	**SD**	**N**	**Mean**	**SD**
Non-degree	3	68.9	16.9	-	-	-
Freshman	3	49.4	48.6	3	84.04	13.8
Sophomore	15	69.1	27.5	15	66.13	30.2
Junior	39	69.5	28.2	44	81.18	23.6
Senior	51	71.4	27.0	49	85.81	14.2
Graduate	15	72.8	28.1	12	83.16	15.3
Post-graduate	6	79.7	29.0	3	91.77	5.1

Table 4.5: Mean Accuracy Score By Academic Level.

4.2.3 Level in School

We classified the participants according to their class year at the university to see if we would observe any impact from a cumulative experience measure. Our participant pool was concentrated around Junior and Senior level students and although measuring learning by level was not a primary research goal, we randomized groups on this independent variable. The significance level for the interaction of Task by Level in School appeared to demonstrate a weak pattern, with a statistically significant result F(10.537, 53) = 2.128, p = .025, ($\eta_p^2 = .152$). The lower number of responses on either end of the ordering may have been a factor in not seeing a more pronounced effect. Table 4.5 shows the Mean Accuracy Score by Level in School where the N value represents the number of responses across all tasks at each level. This N value is the same as Table 4.3 and Table 4.6, where N represents the total number of responses, but different from the N value in Table 4.1 and Table 4.2, where N represents the number of individual participants by demographic.

4.2.4 Time by Group

Although our focus was on accuracy as a dependent variable, we also tracked time to completion by participant. The time score results were similar to the accuracy score results. Figure 4.2 shows a plot of the time to completion by task and group. The time performance was similar for task 1 for the Process group (M=204.8, SD=215.4) compared to the Threads group (M=196.2, SD=117.1) as well as for task 2 for the Process group (M=734.0, SD=329.1) compared to the Threads group (M=754.4,SD=361.7). There was some divergence on task 3 where the Process group took longer on average (M=732.4, SD=357.7) compared to the Threads group (M=579.6, SD=311.0), however a t-test on the time by group for task 3 was not statistically significant, t(81.8)=1.897, p=.061. The mean time to completion by group and task is shown in Table 4.6.

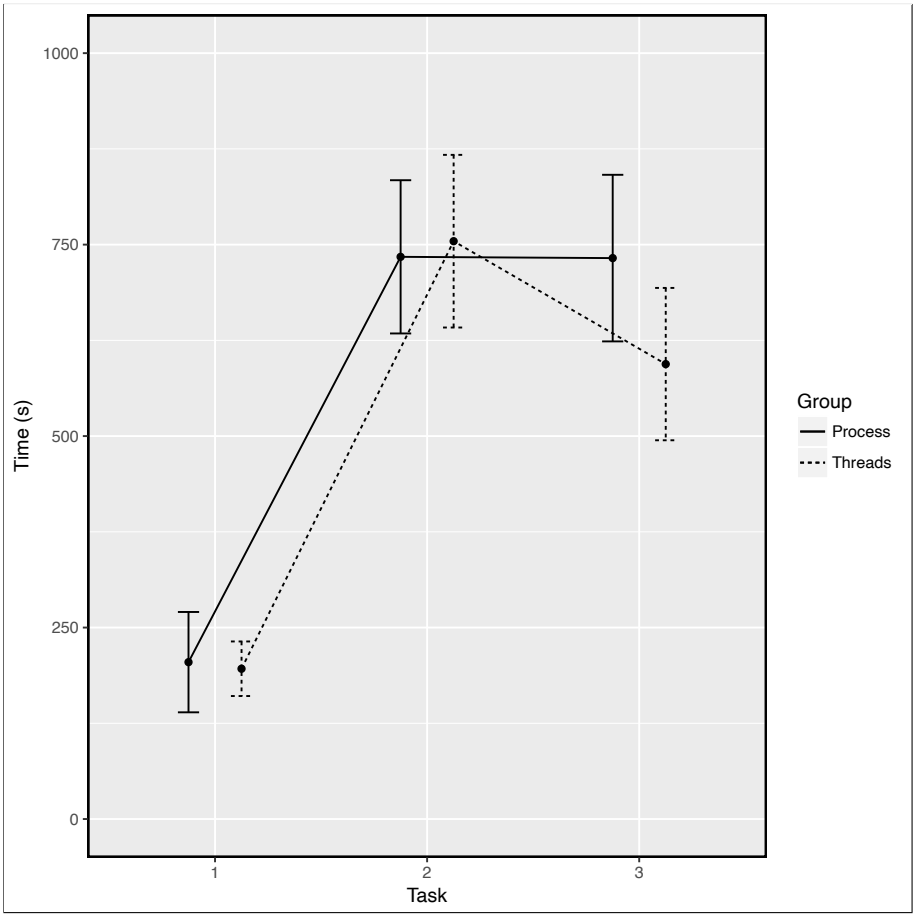


Figure 4.2: Mean Time to Completion by Group.

4.3 Discussion

4.3.1 Research Question 1

One of the limitations of simply using timings, error counting or compiler errors in empirical human trials on programming performance, is that there is not very much information in these measures for the researcher to analyze the cause of the error. Thus researchers sometimes use alternative approaches, like qualitative analysis to attempt to understand why observed effects occurred. We developed the automated Token Accuracy Mapping technique presented here to provide an alternative view to qualitative observational analysis, which despite limitations in its application, can provide a rich data set by which to think through

	Process			**Threads**		
Task	**N**	**Mean**	**SD**	**N**	**Mean**	**SD**
Task 1	44	204.8	215.4	44	196.2	117.1
Task 2	44	734.0	329.1	42	754.4	361.7
Task 3	44	732.1	357.7	41	594.0	311.0

Table 4.6: Mean Time to Completion by Group.

potential theories for observed behavior. We applied the technique to this study to explore the research question (RQ1) of whether the TAM technique can provide useful empirical information for researchers in this area.

In this case, the basic statistical analysis we can perform for the paradigm comparison exercise gives us inconclusive information on whether there are differences between the groups in performing these tasks, but does not yield any information on the causes of programming errors or conceptual misunderstandings in either group. We use the overall accuracy score as a proxy for performance in examining the differences between the groups but the overall scores suggest more similarity than difference, except for Task 3. Drilling down further on the specific TAMs for each task and group, we find that we can learn more information than the overall scores convey.

The Token Accuracy Map is represented as the solution code (by token) followed by a number which represents an accuracy percentage for that token. The accuracy score for a particular student response is calculated by dividing the total number of tokens in the correct solution by the number of correct tokens in the response and then the overall accuracy score for the task is the average of all the student responses for that task and group. The detailed token map percentages are calculated by dividing the total number of responses by the number of correct responses among all participants in that task and group. The detailed map then represents the token accuracy across all participant responses for each token individually and can be useful in identifying what specific words and concepts are troubling for the student programmer by analyzing patterns of groups of tokens. For example, in two excerpts of the full TAMs for Task 2, we can observe that the key parallel construct in the task was only correctly used by about two-thirds of the programmers. In Figure 4.3 we can see that the `choose` token was used correctly 63.6% of the time in the Process group, while in Figure 4.4 we can see that the `synchronized` token was used correctly 64.3% of the time in the Threads group. Although this may appear on the surface to be a simple comparison of alternative syntax, the semantics and function of these concepts are considerably different. In the Process paradigm, the `choose` block contains a list of expressions that may be evaluated concurrently, while in the Threads paradigm, the `synchronized` block is putting a lock on the variable (in this case `N`) so it is protected from other threads. It is interesting that a similar percentage of students in each group properly utilized these different constructs correctly despite the different functions of it.

The primary accuracy problem in Task 3 in the Process group can be identified in the TAM shown in Figure 4.5 for that solution by looking at the low percentage accuracy for the channels required to implement the solution. In the declarations section of the `Main` method, there are 6 channels needed for this paradigm: `Channel<type> identifier`. We can see that the first channel had a very high token accuracy score for the four tokens: `Channel` (91%), < (89%), `integer` (91%) and `c1a` (93%), indicating that most students understood both the need for the channel and the syntax required to declare it. A smaller, but still large percentage of students realized the need for a second channel as indicated by the token scores of 80%, 77%,

```
action84 Consumer93 (89 Reader89 <84 integer80 >77 p189 ,80 Reader89 <77 integer77 >77 p284 )80
 repeat55 while57 true*
  integer66 x80 =71 082
  choose64
   x84 ==75 p177 :77 Read89 (77 )77
    output75 "Received Producer 1:  "46 +14 x75
  or61
   x86 =80 p284 :80 Read91 (82 )89
    output77 "Received Producer 2:  "50 +23 x82
  end32
 end75
end84

* indicates < 5.0% accuracy score
```

Figure 4.3: Task 2 TAM Excerpt (Process).

```
action76 Run79
 integer12 i79 =74 071
 repeat62 while60 true*
  synchronized64 (81 N81 )76
   if62 n79 :67 n79 =71 071
    n76 :69 n81 =67 i64
    i74 =71 i79 +83 176
   end60
  end67
 end71
end79

* indicates < 5.0% accuracy score
```

Figure 4.4: Task 2 TAM Excerpt (Threads).

```
action93 Main95
 Channel91 <89 integer91 >80 c1a93
 Channel80 <77 integer77 >77 c1b80
 Channel7 <* integer9 >* c2a9
 Channel* <* integer* >* c2b*
 Channel* <* boolean* >* r1*
 Channel* <* boolean* >* r2*
 concurrent89
  N71 (68 c1a66 :64 GetWriter66 (64 )66 ,64 c2a64 :59 GetReader66 (66 )66 ,* c1b11 :7 GetWriter11 (9 )*
,* c2b11 :* GetReader9 (* )7 ,* r17 :* GetReader7 (7 )* ,* r27 :* GetReader9 (* )7 )66
  P96 (91 c1a89 :84 GetReader91 (86 )84 ,77 c2a82 :77 GetWriter84 (80 )80 ,* r17 :* GetWriter* (* )*
)91
  P57 (57 c1b57 :52 GetReader52 (52 )52 ,48 c2b50 :46 GetWriter46 (48 )48 ,* r2* :* GetWriter* (* )*
)57
 end89
end91

* indicates < 5.0% accuracy score
```

Figure 4.5: Task 3 TAM Excerpt (Process).

77% and 80%, respectively. Very few students created the other 4 channels needed, however, as indicated by that fact that none of the tokens in any of those declarations has a score over 9%. A consistent effect is noted in the method invocations for the consumer process (`N`) where the number of channels being passed falls from the mid 60% range for the first two channels (`c1a` and `c2a`), down to 9-12% for the second two channels (`c1b` and `c2b`) to near 0% for the last two channels (`r1` and `r2`).

The patterns of these tokens suggest a few different things about what the students may have understood. First, the fact that the declaration lines and the method invocations had consistent scores across the four to five tokens required indicates that they generally understood the syntax and how to declare and use a channel. On the other hand, the falloff in accuracy rates for the other channels tells us that they did not understand some important principles about how a channel functions. Most students declared at least two channels which indicates that they understood that the consumer needed communication channels with both producer objects. Since communication was required both ways, however, the students did not grasp that the channels are uni-directional, so less than 10% declared channels 3 and 4. The completely missing channels 5 and 6 (all under 5%) are used for notifications to the consumer that the producer was ready to receive. We interpret these token patterns to indicate that students did not have a clear understanding of the mechanics of the synchronous communication of the Process model because the missing tokens on the channels show that they did not have the conceptual understanding to recognize the complexity of the required solution. Of course, this could be due to the mode of instruction with only sample reference code in this study design, but in any case suggests that the solution was not intuitively as obvious overall as the threads solution to that specific task. The TAMs clearly point out the nature of the failure though and are suggestive of the concepts that need to be taught.

With this ability to examine patterns of programmer behavior on a token level for differing programming structures in alternative paradigms we find that TAMs are providing useful information suggesting problem areas and possible causation and explanation in this study. This information can be informative both to suggest specific further areas of study and in future study design. Additionally, for the computing science education research community, this knowledge could be used to develop better teaching strategies. These results suggest that we made progress toward RQ2 because the TAMs did provide useful empirical information through an automated technique. Although the TAM analysis required a significant level of non-trivial human interpretation of the students' semantic difficulties, they provided a useful empirical starting point and measurable evidence for comparison for the observed differences.

4.3.2 Additional Analysis

Informal interviews with participants and a graduate course classroom discussion of the problems in Task 3 in the Process group confirmed the data in the TAMs about the specific misunderstanding that the channel communication synchronizes the tasks and blocks execution as a barrier. The result of this misunderstanding

was that the process group participants did not use the request channel required to implement a correct solution with this approach. While we could have used output guards in the process example to possibly make the task easier by needing less channels for communication, we were interested in this experimental design to examine whether novices could understand the wiring of channels required since it is a key complexity for process-oriented programming. It suggests an area of further examination for process-oriented language designers to make constructs that are more easily understood by programmers.

Although this experiment was designed to test the two parallel paradigms, the TAM data also provided some potentially valuable unintended information on the repeat loop construct for Quorum. In both Figure 4.3 (line 2) and Figure 4.4 (line 3), the token "true" used in the indefinite repeat loop had unusually low accuracy scores of 4%, which prompted a manual inspection of the cause. We found that the code samples given as a learning and reference device to the participants (who were all novice Quorum users) did not contain this exact structure, so the participants were left to guess at an intuitive solution for themselves. It suggests a problem with the existing Quorum repeat constructs that warrants further examination for the language designers. This finding has nothing to do with RQ1 in this experiment, but does provide information relating to RQ2. It points to the usefulness of TAMs to provide information that would not be gathered by standard measurement tools like timing data, interviews or error examination.

The lack of established evidence gathering and experimental design standards in the computer science discipline for programming language study makes it difficult to compare our results to previous studies directly, however we believe elements of our study are consistent with the findings of other empirical studies in the area. In the context of other work on novice programming errors by Brown and Altadmri [BA16, BA14, AB15] and on enhanced compiler errors messages by Becker [Bec16], we see similar types of syntax errors in our study, although the languages are different (Java instead of Quorum). Semantic errors that showed a high frequency of occurrence in the blackbox data, like type errors in parameters and return errors in function calls, are different but analogous to errors we observed in methods with channel communication in CSP in particular. The types of syntax errors identified in the syntax study by Denny et al. [DLRT12] are also similar to our results on missing and incorrect tokens. The ancillary data we observed on the repeat loop construct is analogous to error rates observed by Weintrop and Wilensky [WW15, WH17]. Although their study differed from ours in comparing the different modalities of blocks-based compared to text-based languages, we can see some similarity between the correct usage of the tokens "repeat" (61.9% Group 1, 54.6% Group 2) and "while" (59.5% Group 1, 56.8% Group 2) in our study, and the "repeat" loops in the block-based language they observed.

4.4 Limitations

All empirical studies have threats to their validity and ours is no exception. While methodologically, our study conforms to long-studied traditions in the design of randomized controlled trials, understanding the impact of

programming language design on students is difficult. For example, it could be that these paradigms impact different kinds of people in different ways. For example, young students using concepts from concurrency in Alice may very well harbor different impacts than college-level students using a similar approach in another language. Professional programmers also have an ongoing need to learn new languages, libraries and paradigms, and the impacts may vary over time or experience or other factors in those cases. Ultimately, documenting needs, behaviors and performance across these communities under different circumstances could help provide a clearer picture and this study does not have the scope to make broad generalizations.

One potential criticism of the TAM approach is that it only tests syntactical differences compared to a fixed solution and does not reflect semantic differences or alternative answers. While this is a fair criticism theoretically and points to a limitation of the TAM approach for general use, there are ways to mitigate this limitation in study design and post processing of answers. In our case, for example, the ordering or naming in a variable declaration was made irrelevant because we did not specifically check the name of an identifier, simply the token's presence or absence. We are also not making the claim that this syntactic analysis is sufficient as a measurement tool, but rather that its use can provide supplemental data and can yield information on semantic and conceptual understanding through an analysis of measurable patterns. The TAM approach would not be useful in every study design or task analysis and could be particularly ineffective in analyzing complex solutions on an overall basis. The best usage seems to be in narrowly examining specific program structures and in identifying conceptual errors through patterns of missing or erroneous tokens.

A potential limitation of our experimental design was the uneven complexity of the solution to the third task with the six channel requirement of the Process group compared to a single synchronized variable requirement of the Thread group. While this uneven complexity is acknowledged for that task, this was partially the comparison we were intending to test and quantify from the outset. The difference in performance may also be attributable to the conceptual fit of the problem with the paradigm for that task. We can imagine various other example tasks, involving locking for example, where a Thread based group would potentially be at a disadvantage. For this reason, we make no claims in this paper about the overall ease of use of one paradigm versus the other. We tested three common problems taught at the college level knowing there are hundreds or thousands of others to be tested in the future and experimental design is subject to unintended bias.

Further, studies like Rossbach et al.'s [RHW10b] looked at students over long periods of time, whereas ours was conducted in the lab over a few hours. As such, ours is more a measurement of initial ease of use for students than it is long-term measurement of learning or educational outcomes. Both kinds of studies have pros and cons. In studies like Rossbach et al.'s, time measurements are probably better for understanding learning, but they also lack considerably in control, making causality difficult to determine. Studies like ours have the opposite problem. While we can control the setting and variables carefully, our results may not yield the same effects in the field. Ultimately, we think the well-known medical scholar Bradford-Hill's [Mar00]

ideas of 'coherence' make sense. In effect, fully understanding programming language usability will likely require various kinds of methodologies and replication from independent teams to ensure correctness. Our work has revealed trade-offs about the impact of the paradigms, but is not the end of the story.

The choice of language and syntax for a programming language study of this kind could have an unknown impact on the participants, particularly with differences in previous experience with the given language. We chose the Quorum language as a neutral isomorphic option to try to control for previous experience since the participants had little to no experience with it. The size or impact of any novelty effect of this decision is not known or measured in this experiment and could have been different between the two paradigm groups.

4.5 Conclusion

In this study we have found that programming accuracy for student programmers is about the same for thread-based and process-based paradigms when working on simple tasks, but that students had trouble with the process approach if more channels were required, as they scored 35 percentage points lower in the final task. The Token Accuracy Map technique provided evidence that the root cause of the lower accuracy score may have been a difficulty in comprehending the complexity of the channel communication. A teaching strategy tailored to increase understanding of the synchronous communication may improve student understanding of the process paradigm in complex situations. Although this experiment was limited to three basic tasks in parallel computing, it was designed to contribute to the overall body of experimental work on parallelism and programming languages, not to pass an overall judgment on threads vs. process-oriented computing.

While Token Accuracy Maps, as they are described in this paper, have limitations, they proved useful as a tool to gain insight into the overall accuracy of students when working on these tasks, as well as a mechanism to investigate which specific parts of the program were problematic for the students. TAMs might prove useful in future studies to track participant progress through tasks by utilizing time-slice data and to find more information about which parts of programming language syntax are causing problems for programmers.

Chapter 5

GPU Programming Productivity In Different Abstraction Paradigms: A Randomized Controlled Trial Comparing CUDA and Thrust

Coprocessor architectures in High Performance Computing (HPC) are prevalent in today's scientific computing clusters and require specialized knowledge for proper utilization. Various alternative paradigms for parallel and offload computation exist, but little is known about the human factors impacts of using the different paradigms. We conducted a randomized controlled trial to test the hypothesis that students programming in a paradigm using a higher level abstraction approach will be more productive than students using a lower level abstraction paradigm. With computer science student participants from the University of Nevada, Las Vegas with no previous exposure to GPU programming, our study compared NVIDIA CUDA C/C++ as a control group (lower abstraction) and the Thrust library (higher abstraction). The designers of Thrust claim this higher level of abstraction enhances programmer productivity. The trial was conducted on 91 participants and was administered through our computerized testing platform. While the study was narrowly focused on the basic steps of an offloaded computation problem and was not intended to be a comprehensive evaluation of the superiority of one approach or the other, we found evidence that although Thrust is at a higher level of abstraction, the abstractions tended to be confusing to students and in several cases diminished productivity. Specifically, higher level abstractions in Thrust for i.) memory allocation through a C++ Standard Template Library-style vector library call, ii.) memory transfers between the host and GPU coprocessor through an overloaded assignment operator, and iii.) execution of an offloaded routine

through a generic transform library call instead of a CUDA kernel routine all performed either equal to or worse than a lower level abstraction in straight CUDA.

5.1 Introduction

Virtually every High Performance Computing (HPC) cluster architecture in use in the world today is utilizing some type of multiple coprocessor arrangement (either in a Graphics Processing Unit (GPU) card from NVIDIA [Cor17d] or AMD [AMD17] or a straight coprocessor card like the Intel Xeon Phi^{TM} Coprocessor Card [Cor17c]) attached to a host node in order to achieve their highest rated Giga- or Peta-FLOP peak performance [KES+09]. This offload coprocessor architecture requires specialized programming skills because of the need for memory and cache management to place the data so that it is accessible to the processing cores for computation. There are a variety of parallel programming paradigms available depending on the hardware environment and programmer preference, including NVIDIA's CUDA [Cor17e], the open standard OpenCL from the Khronos Group [Gro17], OpenMP [Boa17], Intel's Threading Building Blocks [Cor17b] and Cilk Plus [Cor17a] to name a few. These libraries and models provide generally similar functionality to manage capabilities like offloading computation and thread management, but they do so in different ways with different syntax and compiler instructions.

In order to exercise the low level control and data manipulation often needed for HPC applications, programmers are frequently required to mix and match the paradigms to obtain their desired result. The complexity of this type of programming and the volume of specialized knowledge required, along with the redundancy of choices available, provides a potentially compelling incentive to maximize the relative productivity of programmers. This paper describes a study which compared the higher level abstractions of the Thrust parallel algorithms library (Thrust) [Gab16] to lower level CUDA in a series of tasks required to offload code to a GPU processor. Thrust is an open source high-level interface which the authors claim "greatly enhances programmer productivity while enabling performance portability between GPUs and multicore CPUs [HB15]." To be clear, we are not the authors of these tools and have no vested interest in the outcome of our study. Thus, we evaluate the Thrust designers' claim in a randomized controlled trial using a simple example with new student learners. We use the labels "high" and "low" for the types of abstraction we tested in order to mirror language used in the Thrust website in citing Thrust's design intent. Readers may understandably consider and debate abstraction layers differently than the Thrust team, but we found it reasonable and so use the labels accordingly in this paper.

We investigated four key research questions:

- **RQ1:** Does a high level abstraction for memory allocation on a host/coprocessor device improve programmer performance?
- **RQ2:** Does a high level abstraction for iterating over an array improve programmer performance?

- **RQ3:** Does a high level abstraction (assignment overloading) for memory copy to/from host/coprocessors improve programmer performance?
- **RQ4:** Does a high level abstraction for a kernel routine on a GPU improve programmer performance?

One of the aims of our research program is to contribute to the development of research processes in computing education research (CER) which utilize empirical scientific methods accepted in other scientific disciplines, as described by Malmi, et al. [MSB+14] We also strongly believe CER needs to develop its own discipline-based theories because important issues such as how students understand programming concepts cannot easily be explained by applying general education theories. An important motivation for this study was to provide evidence on the human factors impacts for computing education, especially in teaching advanced concepts to students with previous formalized programming instruction. Our long term aim is to help develop and test CER-specific, data-driven theories with predictive capability which can ultimately be used by programming language designers as well as by educators to develop more effective teaching strategies. As our discipline advances scientifically we seek to contribute to the development of standards, alongside other educational researchers, which can measure the impacts and effects of different design and instructional methodologies. This study is one brick in the wall of an overall approach to computer science research and is not meant to be conclusive or comprehensive on the topic of abstraction or GPU programming paradigms.

5.2 Methods

We used an automated testing program to conduct an RCT on students at the University Nevada, Las Vegas to evaluate programming productivity and learning of a GPU programming task. We compared the time to completion, success rates and other accuracy measures of students who completed 6 programming tasks using either CUDA or Thrust during March and April 2018.

The participants were given 10 minutes to review a set of instructions, which they retained during the study, that provided details on how to complete the tasks using the C++ language and their group paradigm. The instructions included variable types, method syntax, array syntax, looping, and library calls and had common descriptions for how to allocate and deallocate memory on the host CPU and GPU, how to invoke an offloaded method and names of available library calls. Specific details varied only where required for the paradigm being used. The instructions were designed to provide all information necessary to solve the programming tasks, but not give 'cut-and-paste' material which could be used to guess at a correct answer. The instructions were designed to be as consistent as possible in an effort to remove difference in instruction as a factor in the study. The instruction sheets for each group are provided in the Appendix to this paper.

The testing program served as an automated proctor and initiated the timed testing protocol by presenting the tasks at consistent pre-set times in the student's browser. The user display included a testing screen with three areas: a reference area for instructions, an editable area for coding and an output area for feedback.

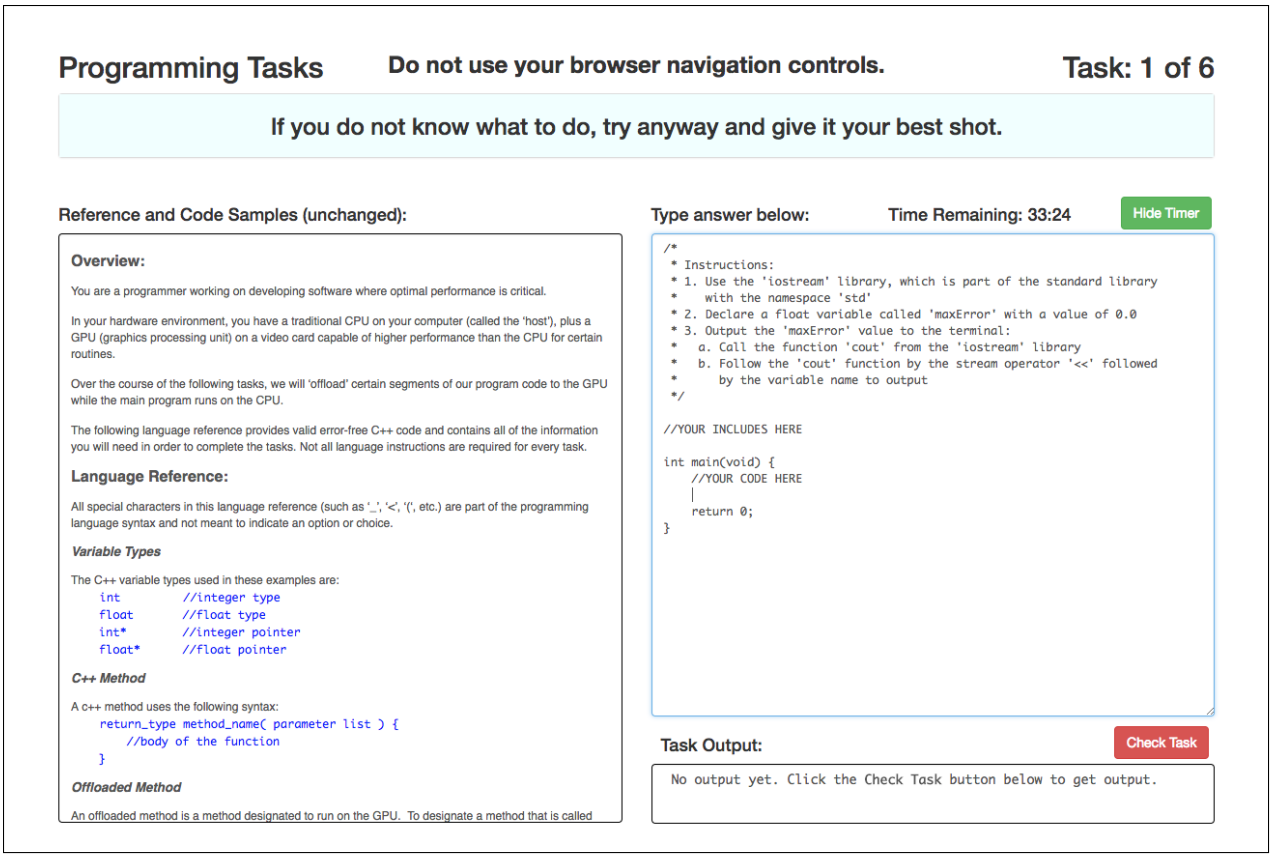


Figure 5.1: Automated Testing Application.

A screenshot of the testing program in a browser is shown in Figure 5.1. In addition to submitted samples, the testing application logs a variety of event-based data, including periodic snapshots, click activity, typing activity, copy and paste activity and focus change activity.

The application had a compile button which sent the code contained in the input area to a remote workstation equipped with a CUDA-enabled GPU with the necessary libraries and tools installed to compile and execute the program in either CUDA or using the Thrust library. The compiler or program output was then returned to the testing application and displayed in the output area and the code submission and output was logged. If the student did not solve the task after seven minutes an additional button became visible with the option to give up and move to the next task. An automated timeout for the task occurred after a pre-set period of time.

5.2.1 Trial Design

The randomized controlled trial used a two factor between subjects design where the programming tasks and instruction were controlled. Participants were randomly placed in one of the two groups and then completed the tasks on the testing platform which consistently administered uniform instruction and time allotments

Level in School	CUDA	Thrust	Total
Sophomore (Second Year)	8	8	16
Junior (Third Year)	20	23	43
Senior (Fourth Year)	15	15	30
Graduate	1	1	2
Total	44	47	91

Table 5.1: Participants by Level in School.

Gender	CUDA	Thrust	Total
Female	5	10	15
Male	39	37	76
Total	44	47	91

Table 5.2: Participants by Gender.

Language	CUDA	Thrust	Total
English	30	33	63
ESL	14	14	28
Total	44	47	91

Table 5.3: Participants by Native Language.

and tracked periodic code snapshots, compiler submissions and time to completion on an automatic basis. The coding tasks and source code for the study were inspired by guides and tutorials at the NVIDIA and Thrust websites [Cor17e, Har17, Gab16].

5.2.2 Recruitment

Subjects were recruited from seven different classes in the Computer Science Department including courses in systems programming, operating systems, programming languages, algorithms and compilers. In every case, the prerequisite courses included Computer Science I and II, which are taught in C/C++. The student participants were all in their second, third or fourth year of study and were given extra credit in the courses as an incentive to participate. Tables 5.1, 5.2 and 5.3 show the breakdown of participant by education level, gender and native language.

5.2.3 Pilot Testing and the Doubling Method

Since the design of empirical testing methods for randomized controlled trials in CER is both new and complex, we adhere to a strict pilot testing framework as a core element of our research methods. We use an iterative approach, which we call the Doubling Method, where we test our experiment design, task instructions, measurement techniques and timing constraints on at least three series of participants in advance of the formalized study. We start with a preliminary testing of at least one participant in each group in

phase one and then make changes in our study design before iterating the process in phases two and three. In each successive phase, we at least double the number of participants in each group before we iterate the process. For this study, the pilot phases consisted of 2, 4 and 10 participants in the Spring of 2017 so that we could fine tune the tasks, instructions and timing.

5.2.4 Study Setting

The participants in the study completed the testing on their own computers in a web browser without any direct observation. The timing and ordering was administered and enforced by the computer and was applied consistently. Students were instructed to turn off cell phones and televisions and complete the study in a quiet environment, however there is no way to know if the testing conditions were similar for all participants or if any students consulted external resources for assistance in completing the tasks.

5.2.5 Intervention

The intervention for this trial was the CUDA or the Thrust programming paradigm group. Each group received instructions within the application based on the language group to which they were assigned. The starting code for each task and group contained the correct code from any previous tasks so that a participant who failed to complete a previous task was not at a disadvantage. Task 6 had custom comments for the CUDA group to explain how to include the `Add` kernel function and how to invoke it in CUDA. These instructions were not necessary in Thrust because of the `thrust::plus<type>` function that can be called with `thrust::transform(..)` to perform the task.

Overall, the tasks flowed sequentially as the user went through the steps needed to perform the offloaded computation. Table 5.4 describes each of the successive tasks in detail as well as the specific abstractions tested in each of the groups. The six steps were to learn to use any required libraries (Task 1: Figure 5.2), allocate memory on both the CPU (Task 2: Figure 5.3, 5.4) and GPU (Task 4: Figure 5.7, 5.8), iterate over the array data (Task 3: Figure 5.5, 5.6), move data between the CPU and GPU in both directions (Task 5: Figure 5.9, 5.10 and execute a method on the GPU (Task 6: Figure 5.11, 5.12). The task instructions in the starting code were identical, however the language intervention had significant differences in syntax, design and complexity which formed the essence of and motivation for this study. The starting code for each task indicated areas for the participant to write their code to complete the task. The header file `checkX_Y.h` interface was provided to check the participant's answer and allow them to move on to the next task. The code used to check the answer was compiled and linked in on the remote GPU workstation and was not provided to the participants since it would provide clues about the correct responses to the tasks.

Detailed Task Description and Abstractions Tested		
Task	**Item**	**Detail**
1	*Description:*	Warm up task consisting of a basic memory declaration and a standard library call.
	Abstraction:	None.
	CUDA:	Use a `#include` and `std::cout`
	Thrust:	Identical solution to CUDA
2 RQ1	*Description:*	Allocate memory for two arrays on the host.
	Abstraction:	Memory allocation on the host.
	CUDA:	Standard c `malloc` declarations with pointers.
	Thrust:	Library call using `thrust::host_vector` similar to C++ STL `vector`
3 RQ2	*Description:*	Populate the host arrays.
	Abstraction:	Library call to replace `for` loop, but possible without.
	CUDA:	Standard c `for` loop assigning value to array with index: `hostX[i] = 1.0`.
	Thrust:	Same as CUDA or use `thrust::fill(..)` with `#include` for Thrust library.
4 RQ1	*Description:*	Allocate memory for two arrays on the GPU.
	Abstraction:	Memory allocation on the GPU.
	CUDA:	Requires pointer declaration and then using `cudaMalloc` to allocate GPU memory.
	Thrust:	Identical to Task 2 solution except call to `thrust::device_vector`
5 RQ3	*Description:*	Copy the contents of the host arrays to the GPU memory.
	Abstraction:	Overloaded assignment operator.
	CUDA:	Requires using `cudaMemcpy` to copy array values with 4 parameters.
	Thrust:	Overloaded operator copies the full array with just pointers: `devX = hostX`
6 RQ3-4	*Description:*	Add the arrays on the GPU and copy the GPU array to the host array.
	Abstraction:	1. Library call to replace a kernel routine and 2. overloaded assignment operator.
	CUDA:	Required writing a kernel routine for adding arrays on the GPU, calling the kernel in CUDA format, synchronizing the device with `cudaDeviceSynchronize()` and then `cudaMemCpy` to copy the array back to the host.
	Thrust:	Library call using `thrust::transform(..)` with 5 parameters to add two vectors on the GPU followed by a memory copy identical to Task 5.

Table 5.4: Task Abstractions.

```
/*
* Instructions:
* 1. Use the 'iostream' library, which is part of the standard
*    library with the namespace 'std'
* 2. Declare a float variable called 'maxError' with a value
*    of 0.0
* 3. Output the 'maxError' value to the terminal:
*   a. Call the function 'cout' from the 'iostream' library
*   b. Follow the 'cout' function by the stream operator '<<'
*      followed by the variable name to output
*/

//YOUR INCLUDES HERE

/* SOLUTION BEGIN
#include<iostream>
** SOLUTION END */

int main(void) {
    //YOUR CODE HERE

/* SOLUTION BEGIN
    float maxError = 0;
    std::cout << maxError;
** SOLUTION END */

    return 0;
}
```

Figure 5.2: Task 1: Identical for Both Groups

```
/*
* Instructions:
*   1. Allocate memory on the host computer for two arrays of
*      type float named hostX and hostY with N elements.
*   2. Put any include statements, if necessary, at the top
*      of your code.
*/

//YOUR INCLUDES HERE

#include "check2_1.h"
int main(void) {
    int N = 1048576;
    //YOUR CODE HERE

/* SOLUTION BEGIN
    float* hostX = (float *) malloc(N * sizeof(float));
    float* hostY = (float *) malloc(N * sizeof(float));
** SOLUTION END */

    check2_1(N, hostX, hostY);
    return 0;
}
```

Figure 5.3: Task 2 : CUDA Group

5.2.6 Randomization

The randomization for group selection was done by the computer using the `rand()` function in php, which utilizes the Mersenne Twister algorithm. The groups were segmented by year in school in order to keep the groups balanced. The testing application maintained a persistent table to track the ordering of participants. The program chooses the order of each pair of participants for each group so that the first language is randomly chosen each time a pair of participants completes the classification survey. Since the test is not administered by a human, there is no interaction with the group ordering and it requires no human intervention. The study was a double blind protocol because the computer server, as the automated proctor, assigned the groups based on a randomization protocol without any intervention by the research team.

```
/*
* Instructions:
*   1. Allocate memory on the host computer for two arrays of
*      type float named hostX and hostY with N elements.
*   2. Put any include statements, if necessary, at the top
*      of your code.
*/

//YOUR INCLUDES HERE

/* SOLUTION BEGIN
#include <thrust/host_vector.h>
** SOLUTION END */

#include "check2_2.h"
int main(void) {
    int N = 1048576;
    //YOUR CODE HERE

/* SOLUTION BEGIN
    thrust::host_vector<float> hostX(N);
    thrust::host_vector<float> hostY(N);
** SOLUTION END */

    check2_2(N, hostX, hostY);
    return 0;
}
```

Figure 5.4: Task 2 : Thrust Group

```
/*
* Instructions:
*   1. Assign a value of 1.0 to each element of the hostX
*      array and 2.0 to each element of the hostY array
*      using a for loop.
*   2. Put any include statements, if necessary, at the top
*      of your code.
*/

//YOUR INCLUDES HERE

#include "check3_1.h"
int main(void) {
    int N = 1048576;
    float* hostX = (float *) malloc(N * sizeof(float));
    float* hostY = (float *) malloc(N * sizeof(float));
    //YOUR CODE HERE

/* SOLUTION BEGIN
    for (int i = 0; i < N; i++) {
        hostX[i] = 1.0;
        hostY[i] = 2.0;
    }
** SOLUTION END */

    check3_1(N, hostX, hostY);
return 0;
}
```

Figure 5.5: Task 3 : CUDA Group

5.3 Results

5.3.1 Baseline Data

A total of 98 students completed at least one task in the experiment. The results of 7 students were excluded either because of incomplete data for all the tasks, a program malfunction or for restarting the experiment. After the invalidation phase, there were 91 participants remaining with 44 in the CUDA group and 47 in

```
/*
* Instructions:
*   1. Assign a value of 1.0 to each element of the hostX
*      array and 2.0 to each element of the hostY array
*      using a for loop.
*   2. Put any include statements, if necessary, at the top
*      of your code.
*/

//YOUR INCLUDES HERE

/* SOLUTION BEGIN
#include <thrust/host_vector.h>
** SOLUTION END */

#include "check3_2.h"
int main(void) {
    int N = 1048576;
    thrust::host_vector<float> hostX(N);
    thrust::host_vector<float> hostY(N);
    //YOUR CODE HERE

/* SOLUTION BEGIN
    for (int i = 0; i < N; i++) {
        hostX[i] = 1.0;
        hostY[i] = 2.0;
    }
** SOLUTION END */

    check3_2(N, hostX, hostY);
    return 0;
}
```

Figure 5.6: Task 3 : Thrust Group

```
/*
*  Instructions:
*   1. Allocate memory on the GPU for two arrays of type float
*      named devX and devY with N elements.
*   2. Put any include statements, if necessary, at the top
*      of your code.
*/

//YOUR INCLUDES HERE

#include "check4_1.h"
int main(void) {
    int N = 1048576;
    //YOUR CODE HERE

/* SOLUTION BEGIN
    float *devX = NULL;
    cudaMalloc((void**) &devX, N * sizeof(float));
    float *devY = NULL;
    cudaMalloc((void**) &devY, N * sizeof(float));
** SOLUTION END */

    check4_1(N, devX, devY);
    return 0;
}
```

Figure 5.7: Task 4 : CUDA Group

the Thrust group.. Table 5.5 shows the mean time to completion for each of the six tasks along with the standard deviation of each task by group.

Table 5.6 shows the number of successful final task results, the number of incorrect compiles prior to a successful result and the average number of incorrect compiles per successful result for both the CUDA and Thrust group. The CUDA group had a lower number of incorrect compile attempts on Tasks 2 (4.1 CUDA vs 6.1 Thrust), 3 (1.1 CUDA vs 2.8 Thrust), 5 (2.2 CUDA vs 2.3 v) and 6 (7.9 CUDA vs 10.0 Thrust) while

```
/*
*  Instructions:
*   1. Allocate memory on the GPU for two arrays of type float
*      named devX and devY with N elements.
*   2. Put any include statements, if necessary, at the top
*      of your code.
*/

//YOUR INCLUDES HERE

/* SOLUTION BEGIN
#include <thrust/device_vector.h>
** SOLUTION END */

#include "check4_2.h"
int main(void) {
    int N = 1048576;
    //YOUR CODE HERE

/* SOLUTION BEGIN
    thrust::device_vector<float> devX(N);
    thrust::device_vector<float> devY(N);
** SOLUTION END */

    check4_2(N, devX, devY);
    return 0;
}
```

Figure 5.8: Task 4 : Thrust Group

```
/*
* Instructions:
*   1. Copy the array data from the host CPU to the GPU
*   2. Put any include statements, if necessary, at the top
*      of your code.
*/

//YOUR INCLUDES HERE

#include "check5_1.h"
int main(void) {
    int N = 1048576;
    float* hostX = (float *) malloc(N * sizeof(float));
    float* hostY = (float *) malloc(N * sizeof(float));
    float *devX = NULL;
    cudaMalloc((void**) &devX, N * sizeof(float));
    float *devY = NULL;
    cudaMalloc((void**) &devY, N * sizeof(float));
    for (int i = 0; i < N; i++) {
        hostX[i] = 1.0;
        hostY[i] = 2.0;
    }
    //YOUR CODE HERE

/* SOLUTION BEGIN
    cudaMemcpy(devX, hostX, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(devY, hostY, N*sizeof(float), cudaMemcpyHostToDevice);
** SOLUTION END */

    check5_1(N, devX, devY);
    return 0;
}
```

Figure 5.9: Task 5 : CUDA Group

the Thrust group had less errors on the identical warm up Task 1 (1.4 CUDA vs 1.0 Thrust) and 4 (2.5 CUDA vs 1.5 Thrust).

```
/*
* Instructions:
*   1. Copy the array data from the host CPU to the GPU
*   2. Put any include statements, if necessary, at the top
*      of your code.
*/

//YOUR INCLUDES HERE

/* SOLUTION BEGIN
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
** SOLUTION END */

#include "check5_2.h"
int main(void) {
    int N = 1048576;
    thrust::host_vector<float> hostX(N);
    thrust::host_vector<float> hostY(N);
    thrust::device_vector<float> devX(N);
    thrust::device_vector<float> devY(N);
    for (int i = 0; i < N; i++) {
        hostX[i] = 1.0;
        hostY[i] = 2.0;
    }
    //YOUR CODE HERE

/* SOLUTION BEGIN
    devX = hostX;
    devY = hostY;
** SOLUTION END */

    check5_2(N, devX, devY);
    return 0;
}
```

Figure 5.10: Task 5 : Thrust Group

	CUDA			Thrust		
Task	**N**	**Mean**	**SD**	**N**	**Mean**	**SD**
1	44	158.3	109.1	47	234.3	234.8
2	44	432.9	335.1	47	735.3	253.9
3	44	224.5	257.8	47	520.1	352.2
4	44	330.5	303.6	47	437.0	334.6
5	44	602.8	330.4	47	489.9	345.3
6	44	1,331.7	364.0	47	1,313.9	362.6
Total	44	513.5	283.3	47	621.7	313.9

Table 5.5: Time to Completion by Group and Task.

5.3.2 Quantitative Analysis

The data was analyzed with a repeated measures ANOVA with a single within subjects factor (the 6 tasks) and two between subjects factors (the paradigm groups and native language). Mauchly's test indicated that the assumption of sphericity had been violated ($\chi^2(2) = .691, p = .005$) therefore degrees of freedom were corrected using Greenhouse-Geisser estimates of sphericity ($\epsilon = 0.87$). The main effect of group, F(1, 87)=8.03, p=.006, ($\eta_p^2 = .037$) was qualified by an interaction between group and task, F(5, 435)=8.94, $p < .001$, ($\eta_p^2 = .056$). We examined other between subjects measures, including level in school, gender and native language, but found no significant effects.

```
/*
* Instructions:
*   1. Run an "add" operation on the GPU by adding the values
*      in devX to devY.
*   2. Put any include statements, if necessary, at the top of
*      your code.
*   3. Copy the contents of the resulting array on the GPU
*      (devY) back to the CPU memory (hostY)
*/

//YOUR INCLUDES HERE


#include "check6_1.h"
/*
* Write a function called 'add' which will be run on the GPU.
* It should have a 'void' return type and three parameters:
* 1 for the number of items in the arrays and 2 for the
* pointers of the arrays to add.
* The addition should be of the form Y = Y + X
*/
//YOUR CODE HERE

/* SOLUTION BEGIN
__global__
void add(int n, float* x, float* y) {
    for (int i = 0; i < n; i++) {
        y[i] = x[i] + y[i];
    }
}
** SOLUTION END */

int main(void) {
    int N = 1048576;
    float* hostX = (float *) malloc(N * sizeof(float));
    float* hostY = (float *) malloc(N * sizeof(float));
    float *devX = NULL;
    cudaMalloc((void**) &devX, N * sizeof(float));
    float *devY = NULL;
    cudaMalloc((void**) &devY, N * sizeof(float));
    for (int i = 0; i < N; i++) {
        hostX[i] = 1.0;
        hostY[i] = 2.0;
    }
    cudaMemcpy(devX, hostX, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(devY, hostY, N*sizeof(float), cudaMemcpyHostToDevice);
    // invoke the 'add' function with 1 block/grid and
    // 1 thread/block
    //YOUR CODE HERE

/* SOLUTION BEGIN
    add<<<1,1>>>(N, devX, devY);
    cudaDeviceSynchronize();
    cudaMemcpy(hostY, devY, N*sizeof(float), cudaMemcpyDeviceToHost);
** SOLUTION END */

    check6_1(N, hostY);
    return 0;
}
```

Figure 5.11: Task 6 : CUDA Group

The mean by group and task shown in the plot in Figure 5.14 as well as Table 5.5 depicts the differences most apparent in tasks 2 and 3. Pairwise t-tests using a Bonferroni correction showed statistically significant differences in means in tasks 2 and 3, but not in any other task. Task 2 observations of the CUDA group (mean=432.9, SD=335.1) compared to the Thrust Group (mean=735.3, SD=253.9) was significant at t(85.3)=-6.05, $p < .001$. Task 3 observations of the CUDA group (mean=224.5, SD=257.8) compared to the Thrust Group (mean=520.1, SD=352.2) was significant at t(72.4)=-5.47, $p < .001$. The effect of task was significant, F(5, 435)=186.60, $p < .001$, ($\eta_p^2 < .556$), however this does not hold much meaning since

```
/*
* Instructions:
*   1. Run an "add" operation on the GPU by adding the values
*      in devX to devY.
*   2. Put any include statements, if necessary, at the top of
*      your code.
*   3. Copy the contents of the resulting array on the GPU
*      (devY) back to the CPU memory (hostY)
*/

//YOUR INCLUDES HERE

/* SOLUTION BEGIN
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/transform.h>
*  SOLUTION END */

#include "check6_2.h"
int main(void) {
    int N = 1048576;
    thrust::host_vector<float> hostX(N);
    thrust::host_vector<float> hostY(N);
    thrust::device_vector<float> devX(N);
    thrust::device_vector<float> devY(N);
    for (int i = 0; i < N; i++) {
        hostX[i] = 1.0;
        hostY[i] = 2.0;
    }
    devX = hostX;
    devY = hostY;
    //YOUR CODE HERE

/* SOLUTION BEGIN
    thrust::transform(devY.begin(), devY.end(), devX.begin(), devY.begin(), thrust::plus<float>());
    hostY = devY;
** SOLUTION END */

    check6_2(N, hostY);
    return 0;
}
```

Figure 5.12: Task 6 : Thrust Group

the tasks were of varying complexity. It does indicate that some tasks were clearly more difficult since the actual amount of code that needed to be written was only a few lines in every case.

A particular task ended in one of the following ways: i) the user successfully completed the task, ii) a predetermined timer expired (15 min for tasks 1-5 and 25 min for task 6) or iii) the participant gave up after at least 7 minutes of trying (in which case the task was assigned the full time). Table 5.7 shows the number of participants in each of the three possible ending conditions for each task by group. The CUDA group showed a higher successful completion percentage compared to Thrust across all tasks (69% compared to 57%), fewer time out events (24% compared to 35%) and fewer give up events (7% compared to 8%). Thrust participants had a more difficult time on Task 2 (36% compared to 73%) an Task 3 (57% compared to 89%)and all participants had a difficult time on Task 6 (20% for CUDA and 26% for Thrust).

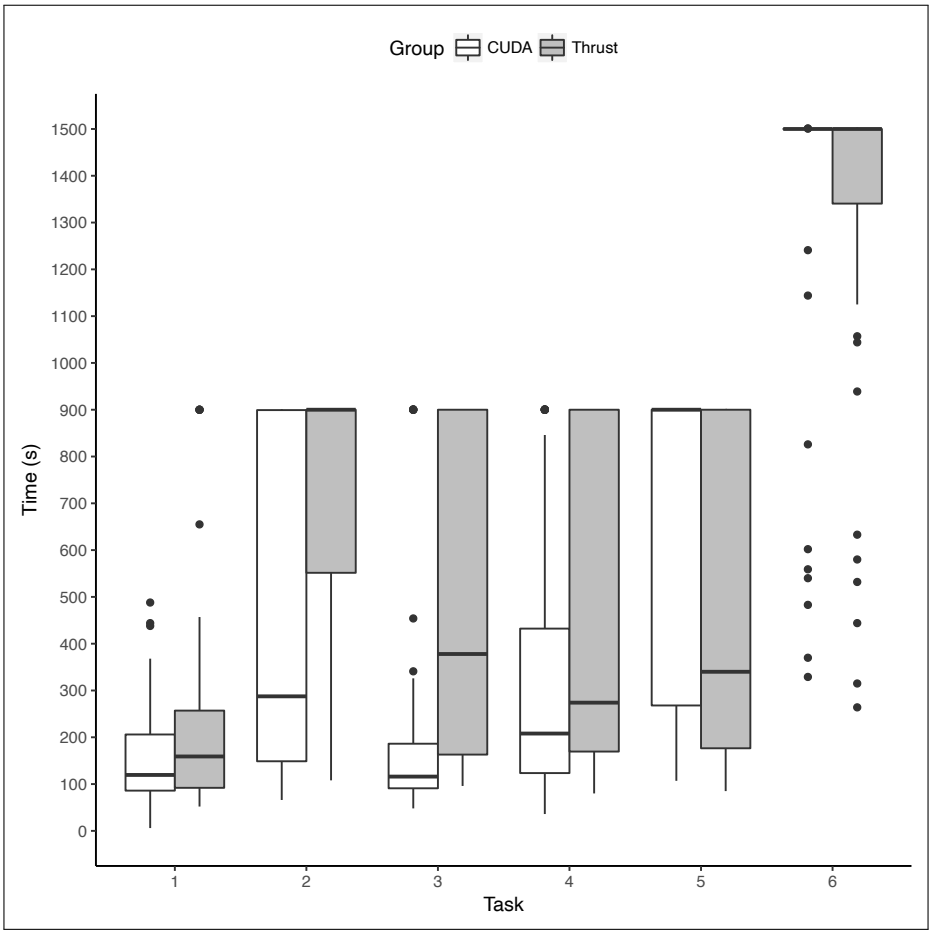


Figure 5.13: Mean Time by Group.

	CUDA			Thrust		
Task	**Success Results**	**Total Errors**	**Avg. Errors**	**Success Results**	**Total Errors**	**Avg. Errors**
1	44	61	1.4	43	45	1.0
2	32	130	4.1	17	103	6.1
3	39	41	1.1	27	75	2.8
4	37	92	2.5	32	48	1.5
5	21	47	2.2	29	68	2.3
6	9	71	7.9	12	120	10.0

Table 5.6: Compiler Errors for Successful Results.

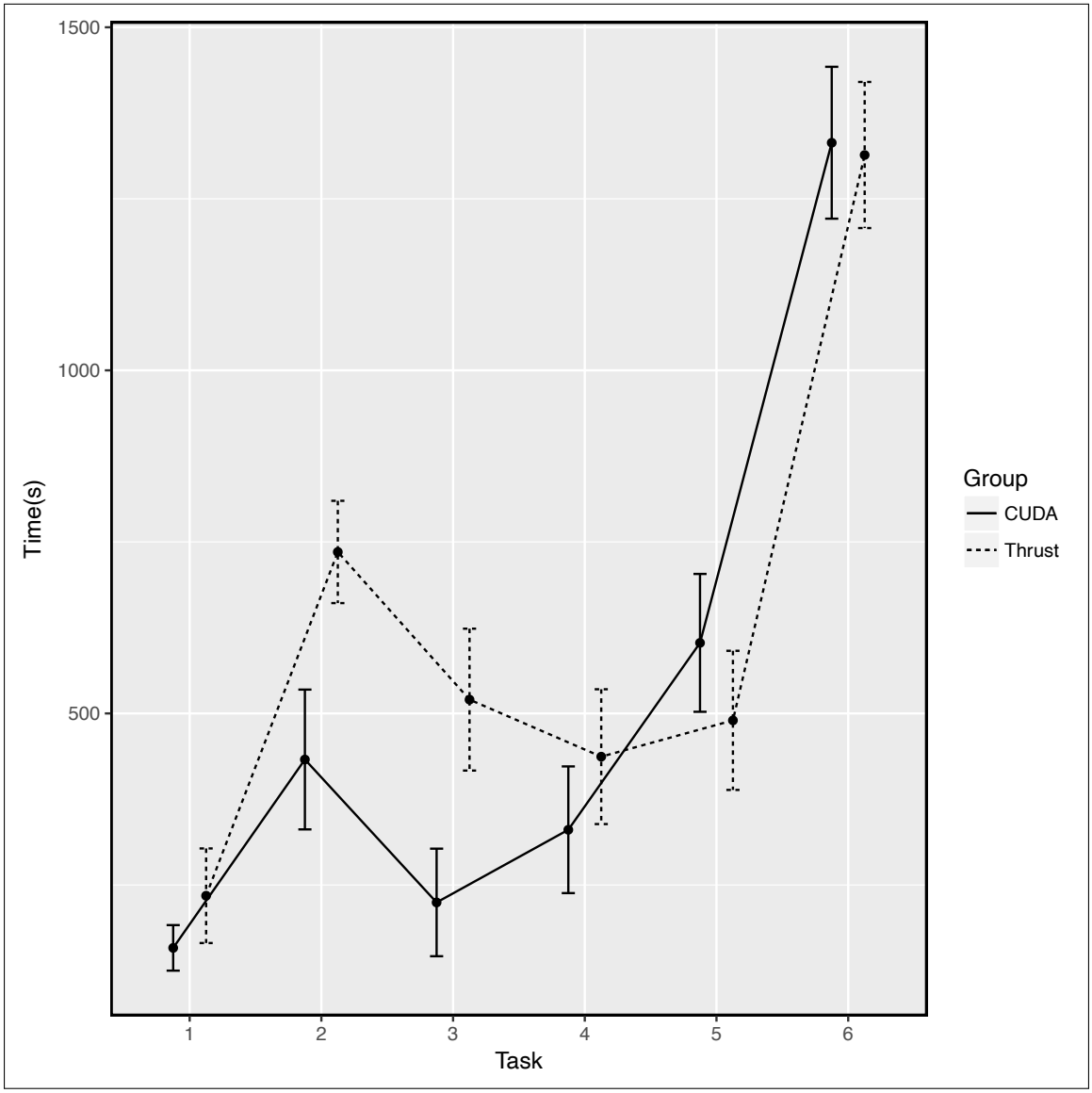


Figure 5.14: Mean by Group and Task.

	COMPLETE		TIME OUT		GIVE UP		SUCCESS RATE	
Task	**CUDA**	**Thrust**	**CUDA**	**Thrust**	**CUDA**	**Thrust**	**CUDA**	**Thrust**
1	44	43	0	2	0	2	100%	91%
2	32	17	6	24	6	6	73%	36%
3	39	27	4	17	1	3	89%	57%
4	37	32	6	10	1	5	84%	68%
5	21	29	19	16	4	2	48%	62%
6	9	12	29	30	6	5	20%	26%
Total	182	160	64	99	18	23	Avg.	Avg.
%	69%	57%	24%	35%	7%	8%	69%	57%

Table 5.7: Number of Participants by Task Result for each Paradigm.

5.4 Discussion

5.4.1 Overall Interpretation

The results of this experimental study present some interesting lessons for future research and we must be cautious to place the findings in a suitable overall context. As a baseline study in an important area we are not seeking to comprehensively and conclusively make a determination that either i.) CUDA or Thrust or ii.) low level or high level abstraction is superior for programmer productivity. Furthermore, as objective researchers, we are not vested in the result. Any such conclusion would require many studies under many different interventions and study designs. This is a single study with a fixed method that sought to control for threats to internal validity in its specific design, but the conditions which we attempted to standardize could have impacted the results in an unintended way. These impacts can only be learned over time with the application of the scientific method to learn and build upon previous studies before conclusions can be drawn and predictive theories developed.

In this study, the observed results require us to reject the core hypothesis that using a higher level abstraction paradigm for GPU programming produces productivity benefits. In each of the successive tasks in our study, the CUDA group performed at least equal to the Thrust group and there is clear empirical evidence of superior performance by the CUDA group in some tasks. We must be very careful not to overstate that finding, however, because it applies to this specific task of CUDA and Thrust for students during their first GPU programming experience with our single tested instruction method.

5.4.2 Research Questions

RQ1: *C++ STL-style library call as an abstraction for memory allocation*

Our first question was tested in both Task 2 (memory allocation on host) and Task 4 (memory allocation on GPU). For Task 2, the CUDA group used a standard `malloc` call in C:

```
float* hostX = (float *) malloc(N * sizeof(float));
```

compared to the Thrust group that used an abstraction virtually identical to the C++ STL-style `vector` call for memory allocation:

```
thrust::host_vector<float> hostX(N);
```

We observed a significantly better student performance on Task 2 for the CUDA group (mean=432.9, SD=335.1, avg. errors=4.1) compared to the Thrust Group (mean=735.3, SD=253.9, avg. errors=6.1) which indicated that the lower level abstraction approach was less confusing to students. This result may have been the result of student familiarity with a standard `malloc` call compared to a C++ STL-style library call, but the differences were significant.

For Task 4,the CUDA group first had to create a pointer in standard C and then use more complicated CUDA-specific syntax to allocate the memory on the GPU, using a likely unfamiliar `void**` type cast:

```
float *devX = NULL;
cudaMalloc((void**) &devX, N * sizeof(float));
```

compared to the Thrust group that used an abstraction identical to the requirement in Task 2 except for the substitution of `host_vector` with `device_vector` in the syntax:

```
thrust::device_vector<float> devX(N);
```

It should also be noted that the Thrust group viewed the correct syntax for this call as scaffolded code in Task 3. Given the complexity and newness of the CUDA solution for Task 4 and the similarity between Task 2 and Task 4 for Thrust, we expected to find that the Thrust group would perform better on Task 4. Although we saw higher relative improvement in the Thrust group from Task 2 to Task 4 (mean reduced by 363.3, compiles reduced by 4.6) compared to the CUDA group (mean reduced by 102.4, average errors reduced by 1.0), both groups improved and the CUDA group still outperformed the Thrust group in overall time to completion. The Thrust group did have less compiler errors on average for this task (1.5) compared to the CUDA group (2.5) however. In both cases of memory abstraction then, even despite higher similarity of solution code for the Thrust group between the two tasks, we observed better performance on the lower

level abstraction of CUDA than on the higher level abstraction of Thrust which contradicted our expectation.

RQ2: *Library call for iterating over an array as an abstraction for a* `for` *loop*

Our second question was tested in Task 3 where the students were asked to fill the arrays they declared in Task 2 with a single `float` value. Since the task was on the host computer, the task could be performed with either a standard `for` loop or in the case of the Thrust group, optionally using the `thrust::fill(..)` library call. Since the `for` loop was highly familiar for the students, we expected to see close similarity in the results between the two groups. Our observed values for Task 3 were significantly in favor of the CUDA group (mean=224.5, SD=257.8, avg. errors=1.1) compared to the Thrust Group (mean=520.1, SD=352.2, avg. errors=2.8). Since the Thrust abstraction so closely parallels the C++ STL-style `vector` call, this was not just a GPU specific knowledge issue, but points to a more general student confusion with the specific higher level abstraction.

RQ3: *Assignment overloading as an abstraction for memory copy between host and GPU*

Our third question was tested in Task 5 (copy array to GPU) and Task 6 (copy array from GPU to host). For Task 5, the CUDA group used CUDA specific syntax to copy the array to the GPU:

```
cudaMemcpy(devX, hostX, N*sizeof(float), cudaMemcpyHostToDevice);
```

compared to the Thrust group which used operator overloading for the assignment operator (=) as an abstraction for a more complex memory copy:

```
devX = hostX;
```

The simplified syntax of the Thrust solution compared to the CUDA solution led us to expect superior performance in the Thrust group for this abstraction. The performance of the CUDA group (mean=602.8, SD=330.4, avg. errors=2.2), although worse than the Thrust group (mean=489.9, SD=345.3, avg. errors=2.3), was not significantly different. Task 6 used constructs very similar to Task 5 for this abstraction, primarily rearranging parameters and variables, and we observed virtually identical results between the groups. Our results suggest that for this research question the assignment operator overloading abstraction

did not provide a benefit to programmer productivity but also did not appear to harm it.

RQ4: *Library call as an abstraction for writing and executing a kernel routine*

Our fourth question was tested in Task 6 which required a much more complex solution than the previous tasks. Since both RQ3 and RQ4 were simultaneously addressed in the solution, the results may also have intermingled. Furthermore, the low success rate on Task 6 (20% for CUDA and 26% for Thrust) compared to previous tasks probably makes our observations less conclusive. Since the overall results were similar for the CUDA group (mean=1,331.7, SD=364.0, avg. errors=7.9) compared to the Thrust group (mean=1,313.9, SD=362.6, avg. errors=10.0), we can only say that there did not appear to be any advantage to the higher level abstraction of Thrust. This was not a result we expected, however, because the single line Thrust library call:

```
thrust::transform(devY.begin(),devY.end(),devX.begin(),devY.begin(),thrust::plus<float>());
```

seemed less complicated than the CUDA requirement to both write a kernel routine with CUDA-specific syntax:

```
__global__
void add(int n, float* x, float* y)
for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
```

as well as the CUDA requirements to both invoke the kernel operator in an unfamiliar format as well as synchronize the host and GPU before copying the result back to the host:

```
add<<<1,1>>>(N, devX, devY);
cudaDeviceSynchronize();
```

5.4.3 General Threats to Validity

We must also interpret these results in the context of a thoughtful examination of how our study design choices impact the study's internal and external validity. The internal validity of a study represents whether

any reasonable conclusions of causality can be drawn between the intentional variation of the independent variable and the observed changes in the dependent variable. The external validity is the extent to which any valid causality can be generalized to other cases. All empirical studies have threats to these forms of validity and any conclusions that are drawn must be measured relative to the factors impacting validity.

This study conforms methodologically to long-studied traditions in the design of randomized controlled trials, although these types of trials are still relatively new in the study of the human factors impacts of programming languages, where study design is inherently difficult and alternative approaches have yet to be explored. In this study specifically, many threats to internal validity were controlled including:

- **History** - A two group design controls for history by comparing a treatment and control groups at the same time with a single measurement.
- **Maturation** - A two group design with the same task set controls for a threat of the groups not changing at the same rate. Our study was also performed in a single sitting and removed external maturation effects.
- **Statistical Regression** - The students were assigned to groups randomly by class year and not based on any pre-test or demographic knowledge so statistical regression was unlikely to have caused any threat to validity.
- **Testing** - A two group design with a single test eliminated testing as a threat to validity.
- **Instrumentation** - Our computerized testing system administered the study consistently and objectively for each group and minimized the threat of instrumentation to validity.
- **Selection** - Randomized group assignment removes the risk of selection threatening validity since there is no bias introduced.
- **Experimental Mortality** - In our study, 7 of 98 trial participants experienced mortality with 44 final participants in the CUDA group and 47 in the Thrust group so there was not a significant threat to validity from mortality.
- **Design Contamination** - Students generally undertook the study independently and everyone did so in a single sitting as required by the computerized system, so there was only limited opportunity for any interaction between the groups during the administration of the tasks. The students were not all physically observed during the study however so it is possible that some contamination occurred without our knowledge.
- **Compensatory Rivalry** - The participants had no knowledge that the study design had multiple groups so were not likely to have tried to compare themselves to another group with any systematic biased so this was not likely a threat to validity.

Regarding this study specifically, the subject matter is not commonly taught in a university curriculum so things like the clarity and method of instruction may play a role in performance. These results may not be generalized to other universities where the curriculum is different (such as those where introductory courses are taught in a language other than C/C++) or to scientists or professionals who may have some related experience with coprocessors. Since this is the first study in this complex area, there are many aspects to measurement, study design and instruction that may affect our conclusions. The study was designed to provide some insight into how students initially responded to the fundamental concept required to use coprocessors (offload computation) and the teaching implications for those basics, however, it does not test alternate teaching methods or the impacts of complex computational algorithms with parallelism considerations.

5.4.4 Instruction Methodology

The complexity of the task of study design cannot be overstated and this study yields some informative examples for future research in computer science. One of the key study design decisions we made was to control the method of instruction between the two groups as much as possible, knowing that it is decidedly *impossible* to provide the exact same instructions to two groups of students to perform a task in two different ways. In attempting to provide a common instruction sheet in the way that we chose, we could have unintentionally biased the results in favor of one group or the other on some or all of the tasks. An obvious area for future study would be to test other fixed instructional methods for the same tasks to see if the observed results are similar. It is probable that our chosen instruction methodology introduced an extraneous or confounding variable to the study that threatens its internal validity, but the extent of its impact cannot be determined from this single study.

Our decision to include a single instruction sheet to both groups for a fixed period of time for their first learning experience of a complex task was almost certainly a sub-optimal instruction method, but we were not attempting to test the effectiveness of the instruction method in this study. That would likely be an interesting area for future exploration, but it is irrelevant to this particular evaluation. In future studies we could imagine other ways to provide instruction to improve overall success rates, possibly including a lecture or video describing the general steps required to complete the task before paradigm specific tasks were introduced. Again, our interest was to study if there were measurable differences in performance between groups given the same instruction and while higher overall success rates would likely improve the overall conclusiveness of the findings, it was not the subject of the study.

We also chose a generic common instruction technique for this study which prevented the participant from copy-pasting code from the instruction sheet to the coding area. This decision was a response from a previous study we conducted where we suspected copy-pasting was a confounding variable and we wanted to control that risk on this study. As a result of this decision, however, students did not have the benefit

of working code samples as reference which may have made them slower overall. Both groups had the same experience however, so there should not have been a bias toward one group or the other. Nonetheless, several students commented in an optional post-study survey that they felt they would have performed better if they had access to this type of resource, suggesting that example-based instructional techniques might improve performance. Study design necessarily includes making these difficult and complex decisions and there are no perfect answers.

5.4.5 Task Difficulty

In our study, high level abstraction performed worse than low level abstraction where our participants were new student learners. It is possible that the effect we observed may have been the result of high level abstraction being more difficult at first and only becoming easier later. One way to test this in the future would be to include professionals compared to students who may have a firmer grasp on high level abstraction. Similar to instruction method, we controlled this variable (programmer experience) but we did not intend to conclusively anoint one paradigm superior to the other.

We also observed increasing difficulty in both groups instead of a learning effect over the course of the tasks. This does provide some formalized evidence in support of the common conception that the colloquial phrase "parallel programming is hard" has at least some objective truth since even the final, but still rudimentary, task of offload computing (adding two vectors) was only fully successfully completed by 21 of the 91 students (23.0%). The study results indicate that our teaching approach in successive task design did not improve performance in later tasks. In fact, the contrary was true and the students had worse performance on the later tasks, indicating that they were not effectively learning the basics of the offloading technique. We found this particularly surprising for Task 5 in the Thrust group which only required the proper `#include` statements and a simple assignment in the form `devX = hostX; devY = hostY;` to copy data from the device memory to host memory. By this point in the task progression we had expected some learning effect to lower task times, especially since the later tasks had the correct code from previous tasks scaffolded in, which provided clues for correct syntax. The final task had the lowest number of successful outcomes (20% for CUDA and 25% for Thrust) and the highest number of give ups and time outs. Although both groups performed equally poorly on this final task, the results may not be comparable given the comparatively low success rate.

Regarding programmer experience, we observed no significant difference in performance on the tasks by the groups based on any experience metric we tracked, including year in school or years of programming experience. This suggests that experience with a new advanced technique may not be as fine-grained at the student level compared to that between students and experienced professionals. Since none of the participants reported any previous CUDA experience, they may have all effectively been starting from the

same knowledge base with regard to learning this new technique. Student comments regarding the difficulty of learning new GPU syntax were consistent with this explanation.

5.4.6 External Validity

One of the surprise aspects of this study data that may be more generalizable is the difficulties experienced by student programmers in using external libraries and dependencies. In an effort to understand why the Thrust group performed worse, we manually inspected the code results and determined that a common issue students had was related to library dependencies and usage. Students frequently improperly used either i) a `#include` statement for the correct Thrust library, ii) the `thrust` namespace, iii) the scope resolution operator `::` for a method or data type, or iv) the `using namespace` statement.

Although not an identical problem, our results are consistent with build and dependency errors documented in professional software developers in industry. In a large scale 2014 study of 26.6 million build errors from 18,000 developers at Google [SSE+14], researchers found that 10% of the error types accounted for 90% of the build failures and that the type of errors observed were most commonly dependency-related. An important difference with our study is that professional developers fixed the errors more quickly than the student programmers, suggesting an impact of experience. In the Google study 75% of the build errors were resolved within at most two builds for all of the 25 most common error kinds for both Java and C++, while in our study, the mean number of failed compiles was 4.8. The cause of the difficulties with issues of `#include` statements could relate to either C/C++ language design itself or in the instructional methods used. It should be informative to educators in their teaching strategies that even students with programming experience in advanced classes may need review or detailed instruction on these concepts since even professional developers continue to err with them.

Another potential criticism of the study with respect to generalizability is that GPU-based parallel programming is intended to be performed by experienced experts on medium and large scale development efforts, not by university students on a trivial function to add two vectors. Although we acknowledge this potential limitation in interpreting the results, we believe this type of research is important to improve and understand both the educational process of students as well as for language design to ensure the successful development of future programmers. Put another way, analyzing results in both professionals and students has value, as these different kinds of demographics may provide different kinds of information to researchers. The results we observed in this study may only apply to student learners using the limited instruction technique described, but it also provides a basis for future research to understand the extent of the limitations.

5.5 Conclusion

This study provides evidence from a randomized controlled trial that computer science students learning GPU programming for the first time performed worse using a higher level abstraction paradigm (Thrust)

compared to a lower level paradigm (CUDA). The results also show that even the most simple version of a fundamental task of parallel computing (offloading a basic vector addition computation to a coprocessor) is challenging for students.

We examined 4 research questions which corresponded to 4 specific different abstractions in GPU programming for i.) memory allocation, ii.) array iteration, iii.) memory copy to/from host/coprocessor and iv.) an offloaded kernel routine. In the 5 tasks where abstractions were tested, we observed that the low-level CUDA abstraction paradigm tested equal to or better than the high-level Thrust abstraction paradigm in every case among student learners. While our results are not a comprehensive or conclusive determination of superiority for either CUDA or Thrust, the fine-grained examination of these specific abstractions provides interesting and potentially useful information for language designers and instructors.

Chapter 6

Analysis of Compiler Errors Using Token Signature Analysis

Decades of study [BDP+19] have shown that compiler error messages are considered difficult to understand, particularly for novices learning to program. The research has also suggested the message feedback is extremely important to programmers, so there have been various approaches suggested to providing enhanced messages. In this paper, we present a novel approach to identifying and enhancing error messages based on a token signature technique. The token signature technique can provide additional contextual information for certain types of errors based on patterns of tokens and missing tokens in a segment of code. We applied this technique to a database containing 108,110 programs written in the Quorum programming language that had at least one compiler error and found error patterns similar to those observed by other researchers. We used the token pattern data to develop more targeted customized messages and a hint engine for the most common errors patterns we observed.

6.1 Introduction

The format of this paper is inspired by the 2019 report from the working group on compiler error research of the Innovation and Technology in Computer Science Education Conference (ITiCSE) [BDP+19]. The ITiCSE paper presents a comprehensive synthesis of the body of knowledge on compiler error research, including certain semi-standardized results reporting, well-defined nomenclature, a framework for analyzing the technical challenges of effective error messages, and guidelines for known approaches to providing effective compiler messages. We adhere to the general format and nomenclature of the ITiCSE paper in the reporting of an analysis of our compiler error database.

In this paper, we will present a summary and analysis of a code repository in which we have been accumulating data for over three years from the Quorum programming language. Quorum is a evidence-based JVM-based programming language designed for simplicity for novice users and motivated by the need for accessibility to the blind and visually impaired community. Quorum is an open-source project with an extensive online curriculum primarily funded for over five years by the National Science Foundation. The compiler is now in its seventh generation with additional libraries and supporting software, like the recently introduced Quorum Studio integrated development environment (IDE) added annually. The code repository has tracked user submitted programs since the introduction of on online IDE and JavaScript cross compiler which allowed Quorum to run through a browser on a website. An analysis of that database is presented here. Additionally we will present and discuss, along the lines of the ITiCSE report, our work to develop a technique to supplement compiler error message and editor hints for auto-completion or auto-correction which we call the Token Signature Technique.

6.1.1 Motivation for Token Signature Technique

The token signature technique we developed is based on our previous work in token accuracy maps. [DSUP20, Dal16, SS13] It is motivated by the need to provide additional information beyond an Adaptive LL* parse [PF11] to detect potential root causes of errors. The essential concept of our token based analysis approach involves running a segment of code through a lexical analyzer ("lexer"), created by the ANTLR program (antlr.org), to generate a list of tokens which have a corresponding number for the type of token. Using a token-by-token matching or alignment algorithm, we are able to make objective comparisons about two differing code samples or look for token patterns similar to historic error patterns.

In our earlier work, we applied the token approach to a full program and then generated an overall accuracy score after applying a string alignment algorithm [NW70]. We also looked at a token specific score to determine the accuracy rate of a particular token. This approach had limitations in a generalized application for various reasons, including the variability of possible correct solutions for comparison as well as the imperfect nature of applying a string alignment algorithm. A detailed analysis of localized token by token responses in certain code segments, however, allowed us to observe patterns of behavior. In the particular study of student participants in the randomized controlled trial on concurrency paradigms [DSUP20] we could see that patterns of consistently missing tokens in a particular concurrency construct revealed that the students lacked a key understanding of the paradigm in order to complete the task successfully. With this insight, we developed the token signature technique as a mechanism to target localized code segments on a systematic basis.

6.2 Definitions

Throughout this chapter and the next we refer to a number of terms which can be similar and so to avoid confusion, we provide the following definitions to provide clarity and specificity to the meaning we intend:

- **Error Code** - The numeric code assigned by the Quorum compiler to categorize different types of compiler errors based on their origination. The numbers vary from 0 up to 46, identified in the list in Table 6.3
- **Lexer** - The software program that performs the lexical analysis and takes a stream of characters from a program and converts it to a set of tokens. We used the same ANLTR lexical generator [PF11] used by Quorum.
- **Lexical Analysis** - The process of breaking a stream of characters into tokens usually performed by a lexer or tokenizer program.
- **Token** - A sequence of characters that matches a pattern recognized by a language. Also in our case a class containing all the information about a token accessible to the parser at the time of a compiler error. A list of Quorum tokens is provided in Figure 6.2.
- **Token Data** - The user entered information for a particular type, for example, an token representing an integer literal has a user entered value for the token.
- **Token Map** - The Token Signature mapped to the Token Symbols and Names.
- **Token Name** - A descriptive name we gave to the token, for example, `INTEGER_KEYWORD`.
- **Token Type** - A numeric code assigned by the Quorum lexer to a particular token, for example, the Token Type associated with an INTEGER_KEYWORD is `37`,
- **Token Signature** - A string of Token Types separated by spaces for each token lexed from a particular segment of code (a line in this paper).
- **Token Symbol** - The string of characters recognized as a token by the language, for example `integer` is the symbol in Quorum to mean an integer keyword.

6.3 Methods

In order to apply the token signature technique to a body of actual compiler errors, we utilized a database of code submissions from the Quorum Programming Language site (quorumlanguage.com). The Quorum site has an online interactive development environment (IDE) shown in Figure 6.1 which allows a user to type code in an editor and submit it for compilation and execution. When code is submitted, our system logs the submission and certain meta data in a private database. The system went live in June 2017 and as of

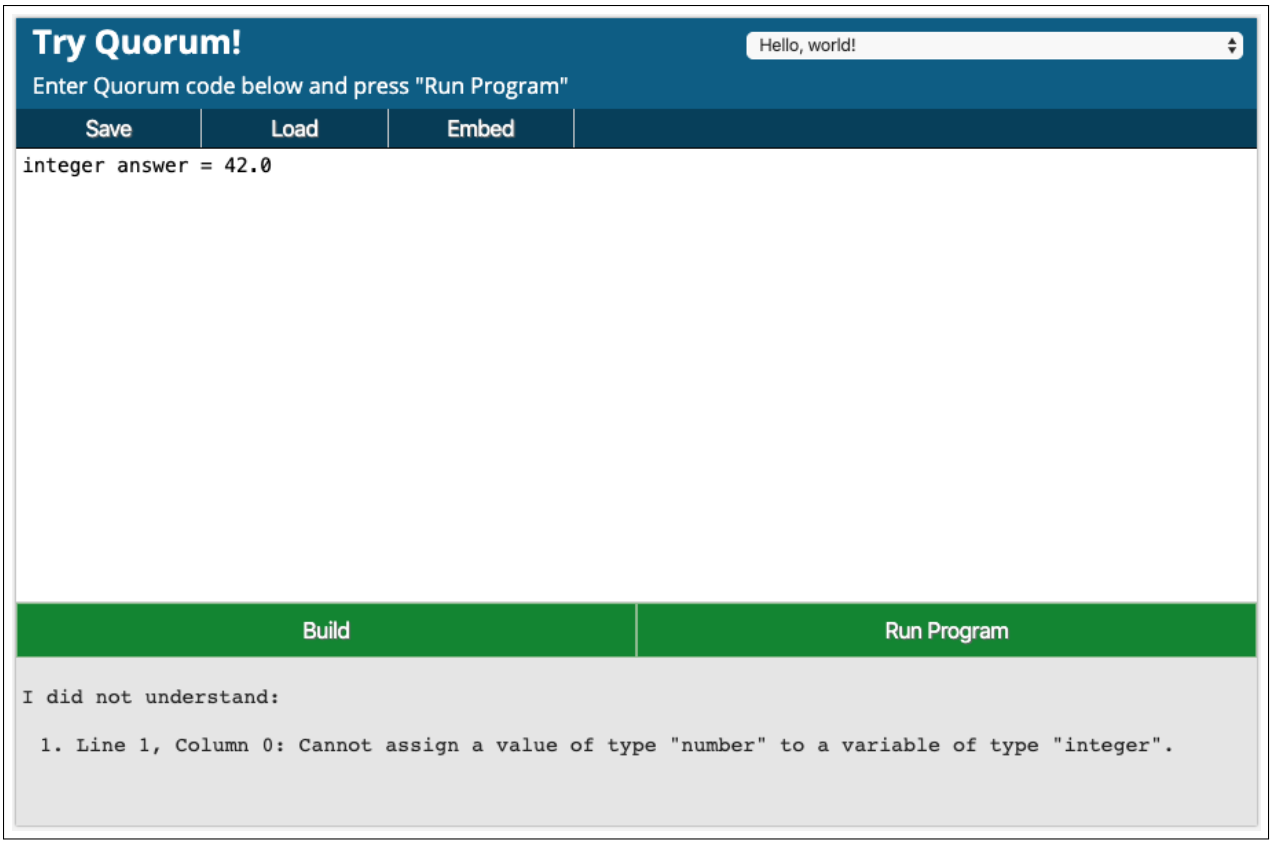


Figure 6.1: Quorum Online Interactive Development Environment (IDE).

Summer 2019, when the data for this paper was pulled, contained 294,631 usable code samples, 108,110 of which contain at least one compiler error.

The Quorum site contains extensive interactive curriculum, tutorials and activities, most of which have IDEs built in to the pages. Since we are able to track which IDE initiated the compiler request, we know which page the user was on when they made their submission. The primary sources of users data, sorted by number of error files (see Table 6.1) are:

Data Source	**Errors**	**Pct.**
Hour of Code Astronomy	47,124	43.6%
Quorum Lessons	23,452	21.7%
Quorum Tutorials	22,835	21.1%
Girls Who Code	8,025	7.4%
Skynet Junior Scholars / IDATA	4,333	4.0%
General	2,341	2.2%
TOTAL	108,110	100.0%

Table 6.1: Sources of Files In Quorum Database

1. **Hour of Code Astronomy Activity** - A 20 segment activity designed for first time programmers for Code.org's annual Hour of Code activity.
   (`https://quorumlanguage.com/hourofcode/astro1.html`)
2. **Quorum Lessons** - Activity labs designed for teachers to use in the classroom, accessible to blind and visually impaired students.
   (`https://quorumlanguage.com/learn.html`)
3. **Quorum Tutorials** - A multi-track curriculum designed for teachers to use in the classroom, accessible to blind and visually impaired students.
   (`https://quorumlanguage.com/reference.html`)
4. **Girls Who Code** IDEs placed in activities.
   (`https://girlswhocode.com`)
5. **Skynet Junior Scholars** IDEs placed in activities.
   (`https://skynetjuniorscholars.org`)
6. **General** (Home page and Other)
   (`https://quorumlanguage.com)`

6.3.1 Compiler Output Modification

In order to analyze a data set with over 108,110 compilation errors we had to automate certain aspects of the process. The first step was to modify the error handling routine within the Quorum compiler to prove a verbose output option with more detailed information about the errors it finds. The detailed error output comes in JavaScript Object Notation (JSON) [jso20] format, which can be written to an external file. Sample JSON output from a compilation is shown in Figure 6.2. The JSON output file provides detailed information in a parseable format about the exact error, the error messages and the locality of the error, all of which the compiler had available to it at the time the error was detected. After we had this capability, we wrote a processing program which compiled all of the 294,631 source files and wrote JSON output files for each of the 108,110 files that had compiler errors. We created a post processing program that parsed the JSON output files to create, group and sort a master list of errors by type for the whole collection of error files. The JSON errors used for this dissertation were based on Version 7.0 of the Quorum Compiler. We stored this information in a database along with i.) the actual line of code which caused the first error in the file, ii.) the token signature which we calculated from the error line, iii.) the error type, iv.) the error description and v.) the error message displayed to the user. We deliberately chose to focus on the first error for this analysis and for the hint engine, which the literature [BMT+18, BM84] indicates is often better for the novice programmer in order to reduce confusion.

```
{
  "Compiler": {
    "Name": "Quorum",
    "Version": "Quorum 7.0"
  },
  "Intro": "This program did not compile. I have compiled a list of errors for you
below:",
  "Errors": [
    {
      "Display": "/home/daleiden/compile/code_files_errors/115.quorum, Line 1, Column
0: I noticed that the variable a has the declared type of integer, but the right hand
side of the statement was blank. For example, you might try integer a = 0",
      "Type": 45,
      "Type Display": "For an assignment of a primitive with a declared type, a value
is required",
      "Message": "I noticed that the variable a has the declared type of integer, but
the right hand side of the statement was blank. For example, you might try integer a =
0",
      "Line": 1,
      "Column": 0,
      "Key": "/home/daleiden/compile/code_files_errors/115.quorum"
    },
    {
      "Display": "/home/daleiden/compile/code_files_errors/115.quorum, Line 2, Column
0: Cannot assign a value of type 'number' to a variable of type 'integer'.",
      "Type": 12,
      "Type Display": "Invalid operator",
      "Message": "Cannot assign a value of type 'number' to a variable of type
'integer'.",
      "Line": 2,
      "Column": 0,
      "Key": "/home/daleiden/compile/code_files_errors/115.quorum"
    }
  ]
}
```


Figure 6.2: Sample JSON Output from Quorum Compiler.

6.3.2 Token Signature Generation

The token signature is a set of numbers corresponding to tokens in a particular line of code based on the token type's unique assigned number. In order to generate a token signature for the line of code we take a file with a code sample and a line number and run it through a tokenizer program which extracts the line of code and runs it through a lexer which was generated by the ANTLR program [PF11]. The lexer then breaks down the line of code into a series of tokens. The token types are looked up in a table and assigned a number to represent their type. The tokenizer program then outputs a list of tokens including the token number, name, symbol and user input. The token signature is constructed using the list of token numbers separated by a single space. Figure 6.3 shows an example of a line containing an error which was run through the tokenizer.

For example, this line of code written in the Quorum programming language:

```
integer answer = 42
```

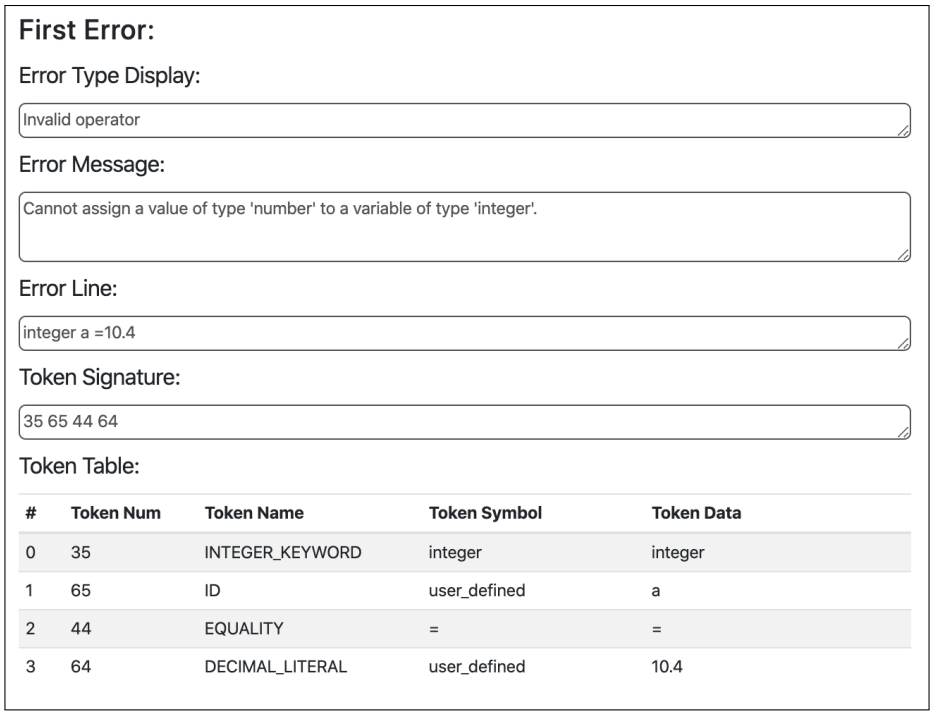


Figure 6.3: Example Token Signature.

has four tokens: i.) an integer keyword ("`integer`"), ii.) a variable name or identifier ("`answer`"), iii.) an assignment operator ("`=`") and iv.) an integer literal ("`42`"). Looking up each of these token types in the table of tokens for Quorum (shown in Table 6.2), we get the token numbers `35` for the integer keyword, `65` for the identifier, `44` for the equal sign and `63` for the integer literal. The token signature for this line of code is therefore `‘‘35 65 44 63’’`, depicted as follows:

```
      35          65   44          63
integer_keyword   id    =   integer_literal
```

6.3.3 Signature Comparison

Using this token signature, we can now make comparisons of lines with errors to lines with correct syntax or observed patterns of errors to suggest hints from a database or rules engine. Using the example shown in Figure 6.3, we can see that there is a mismatch between the declared type of the variable ("`integer`") and the actual type of the literal ("`number`"). The error message generated by the compiler in this particular case is: ("`Cannot assign a value of type ’number’ to a variable of type ‘integer’`"). Our intent is to provide additional supplemental hints or suggestions based on observed patterns of common mistakes.

We can see that the token signature:

```
      35          65   44          64
integer_keyword   id    =   decimal_literal
```

Token Type	Token Name	Token Symbol
1	OUTPUT	output
2	ON	on
3	CREATE	create
4	CONSTANT	constant
5	ELSE_IF	elseif
6	ME	me
7	UNTIL	until
8	PUBLIC	public
9	PRIVATE	private
10	ALERT	alert
11	DETECT	detect
12	ALWAYS	always
13	CHECK	check
14	PARENT	parent
15	BLUEPRINT	blueprint
16	NATIVE	system
17	INHERITS	is
18	CAST	cast
19	INPUT	input
20	SAY	say
21	NOW	now
22	WHILE	while
23	PACKAGE_NAME	package
24	TIMES	times
25	REPEAT	repeat
26	ELSE	else
27	RETURNS	returns
28	RETURN	return
29	AND	and
30	OR	or
31	NULL	undefined
32	STATIC	static
33	ACTION	action
34	COLON	:
35	INTEGER_KEYWORD	integer
36	NUMBER_KEYWORD	number
37	TEXT_KEYWORD	text
38	BOOLEAN_KEYWORD	boolean
39	USE	use
40	NOT	not
41	NOTEQUALS	not=
42	PERIOD	.
43	COMMA	,
44	EQUALITY	=
45	GREATER	>
46	GREATER_EQUAL	>=
47	LESS	<
48	LESS_EQUAL	<=
49	PLUS	+
50	MINUS	-
51	MULTIPLY	*
52	DIVIDE	/
53	MODULO	mod
54	LEFT_SQR_BRACE	[
55	RIGHT_SQR_BRACE	]
56	LEFT_PAREN	(
57	RIGHT_PAREN	)
58	DOUBLE_QUOTE	”
59	IF	if
60	END	end
61	CLASS	class
62	BOOLEAN_LITERAL	user_defined
63	INTEGER_LITERAL	user_defined
64	DECIMAL_LITERAL	user_defined
65	ID	user_defined
66	STRING	user_defined
67	NEWLINE	
68	WS	
69	COMMENTS	

Table 6.2: Quorum Language Tokens Types and Numbers.

is incorrect, but we don't know if the programmer meant:

a.)

```
      35         65   44         63
integer_keyword  id    =   integer_literal
```

or b.)

```
      36         65   44         64
number_keyword   id    =   decimal_literal
```

or something else entirely. Since we know definitively that the first and fourth tokens need to be consistent in any line consisting of `keyword id = literal`, we can create a rule to suggest two correct lines of code based on 3 of the 4 tokens matching a known correct pattern. Using the compiler error manager's knowledge of the users token symbols, we could generate a friendly error and suggestion such as:

```
On line 1 of your program, you attempted to assign a value (10.4) with a ‘‘number’’
type to a variable (a) with an ‘‘integer’’ type, which is not allowed.
You might have been trying to do one of the following:
number a = 10.4
integer a = 10
```

Of course, our rule-based suggestions are not guaranteed to match the user's intent, but they could guide the programmer to a quicker resolution or they could be used in an auto-correcting mechanism in an interactive code editing environment. Relating to the ITiCSE working group paper again [BDP+19], this type of messaging targets the deliver of feedback at the time and place of the error in the context where learning seems most likely to occur.

6.3.4 Analytics Dashboard

In order to work more easily with the database system and the assorted computation and presentation of the token signatures, we built a display dashboard to view, sort, filter and navigate the data for qualitative inspection. The dashboard is shown in Figure 6.4. The dashboard provides detailed information captured at the time the compile event was submitted, such as the user name (if the user is logged in on the Quorum site), the name of the IDE on the site, the web page where the user request originated, the version of the compiler used, the timestamp of the event and the error type and number of compiler errors. The Dataset section provides filtering mechanisms and the number of items in the filtered set. Additionally the users code and the compiler error message they were given are on the left along with the verbose error message we generate in JSON in the recompilation. The First Error section displays the error message, error line and the calculated token signature and table.

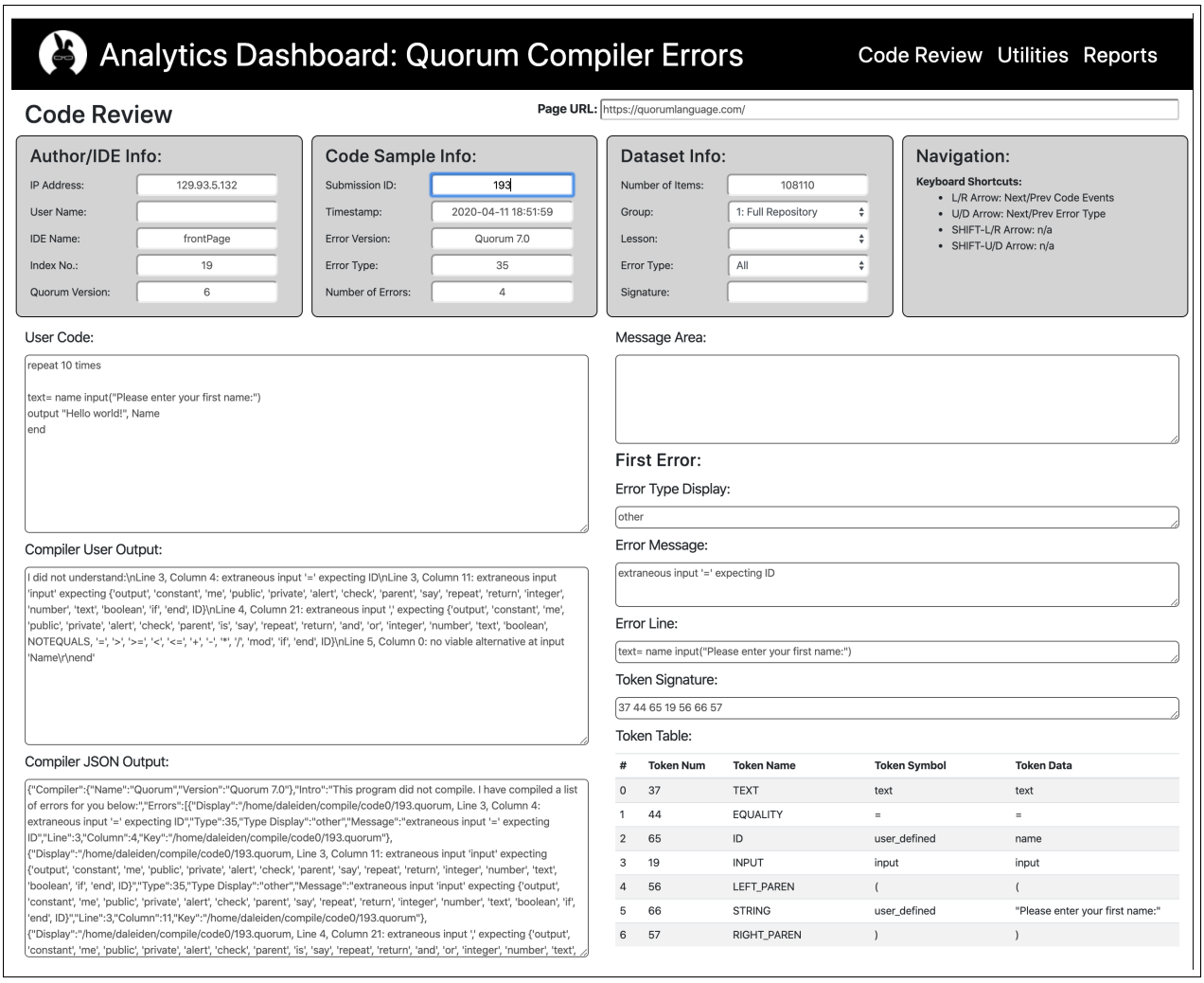


Figure 6.4: Analytics Dashboard.

6.4 Results

As Becker et al. [BMT+18] observed, it is common for studies on compiler errors to include a presentation of the "most common" errors based on the logical assumption that helping students with the most common errors is the most productive. We adhere to the common standard here with a presentation of error frequencies in the Quorum programming language database. The results of the first phase of processing our database, which consisted of sorting and counting the errors by compiler error code, are shown in Table 6.3 and Figure 6.5. As a percentage of the total errors, the top single error, `PARSER_NO_ALTERNATIVE` accounted for 31.7% of all errors. The top 5, 10 and 15 error types combined comprised 76.7%, 97.7% and 99.9% of all errors.

The number of errors by compiler error code for All Errors, not just the First Error, displayed a similar graphical pattern as shown in Table 6.4 and Figure 6.6, but the ranking of the error codes shows a different

Error Code	Error Code Description	N	%	Cum. %
43	PARSER_NO_VIABLE_ALTERNATIVE	34,316	31.7%	31.7%
35	OTHER	20,111	18.6%	50.3%
0	MISSING_VARIABLE	12,348	11.4%	61.8%
11	MISSING_USE	8,218	7.6%	69.4%
41	INPUT_MISMATCH	7,882	7.3%	76.7%
42	LEXER_NO_VIABLE_ALTERNATIVE	7,730	7.2%	83.8%
14	DUPLICATE	6,167	5.7%	89.5%
3	MISSING_METHOD	4,003	3.7%	93.2%
12	INVALID_OPERATOR	3,326	3.1%	96.3%
5	INCOMPATIBLE_TYPES	1,520	1.4%	97.7%
37	VARIABLE_INFERENCE	735	0.7%	98.4%
7	MISSING_CLASS	658	0.6%	99.0%
45	NO_RIGHT_HAND_SIDE_ON_NORMAL_ASSIGNMENT	520	0.5%	99.5%
4	MISSING_MAIN	326	0.3%	99.8%
2	MISSING_RETURN	102	0.1%	99.9%
24	IF_INVALID_EXPRSSION	61	0.1%	99.9%
33	REPEAT_NON_BOOLEAN	24	0.0%	99.9%
32	REPEAT_TIMES_NON_INTEGER	17	0.0%	100.0%
13	UNREACHABLE	15	0.0%	100.0%
25	MISMATCHED_TEMPLATES	8	0.0%	100.0%
26	INSTANTIATE_ABSTRACT	8	0.0%	100.0%
44	PRIMITIVE_INVALID_ACTION_CALL	7	0.0%	100.0%
46	ACCESS_ERROR	6	0.0%	100.0%
34	CONSTANT_REASSIGNMENT	2	0.0%	100.0%
31	METHOD_DUPLICATE	0	0.0%	100.0%
10	MISSING_PARENT	0	0.0%	100.0%
	TOTAL	108,110	100%	

Table 6.3: Errors By Compiler Error Code - First Error Only

Error Code	Error Code Description	N	%	Cum. %
42	LEXER_NO_VIABLE_ALTERNATIVE	175,899	39.7%	39.7%
35	OTHER	65,883	14.9%	54.5%
43	PARSER_NO_VIABLE_ALTERNATIVE	60,347	13.6%	68.2%
14	DUPLICATE	53,628	12.1%	80.3%
0	MISSING_VARIABLE	20,185	4.6%	84.8%
41	INPUT_MISMATCH	17,405	3.9%	88.7%
5	INCOMPATIBLE_TYPES	14,449	3.3%	92.0%
11	MISSING_USE	11,365	2.6%	94.6%
12	INVALID_OPERATOR	8,005	1.8%	96.4%
3	MISSING_METHOD	7,094	1.6%	98.0%
37	VARIABLE_INFERENCE	3,055	0.7%	98.7%
7	MISSING_CLASS	2,642	0.6%	99.3%
24	IF_INVALID_EXPRSSION	1,680	0.4%	99.6%
45	NO_RIGHT_HAND_SIDE_ON_NORMAL_ASSIGNMENT	690	0.2%	99.8%
4	MISSING_MAIN	496	0.1%	99.9%
2	MISSING_RETURN	197	0.0%	99.9%
33	REPEAT_NON_BOOLEAN	71	0.0%	100.0%
25	MISMATCHED_TEMPLATES	45	0.0%	100.0%
13	UNREACHABLE	41	0.0%	100.0%
31	METHOD_DUPLICATE	29	0.0%	100.0%
32	REPEAT_TIMES_NON_INTEGER	21	0.0%	100.0%
46	ACCESS_ERROR	19	0.0%	100.0%
44	PRIMITIVE_INVALID_ACTION_CALL	9	0.0%	100.0%
26	INSTANTIATE_ABSTRACT	8	0.0%	100.0%
34	CONSTANT_REASSIGNMENT	2	0.0%	100.0%
10	MISSING_PARENT	1	0.0%	100.0%
	TOTAL	443,226	100%	

Table 6.4: Errors By Compiler Error Code - All Errors

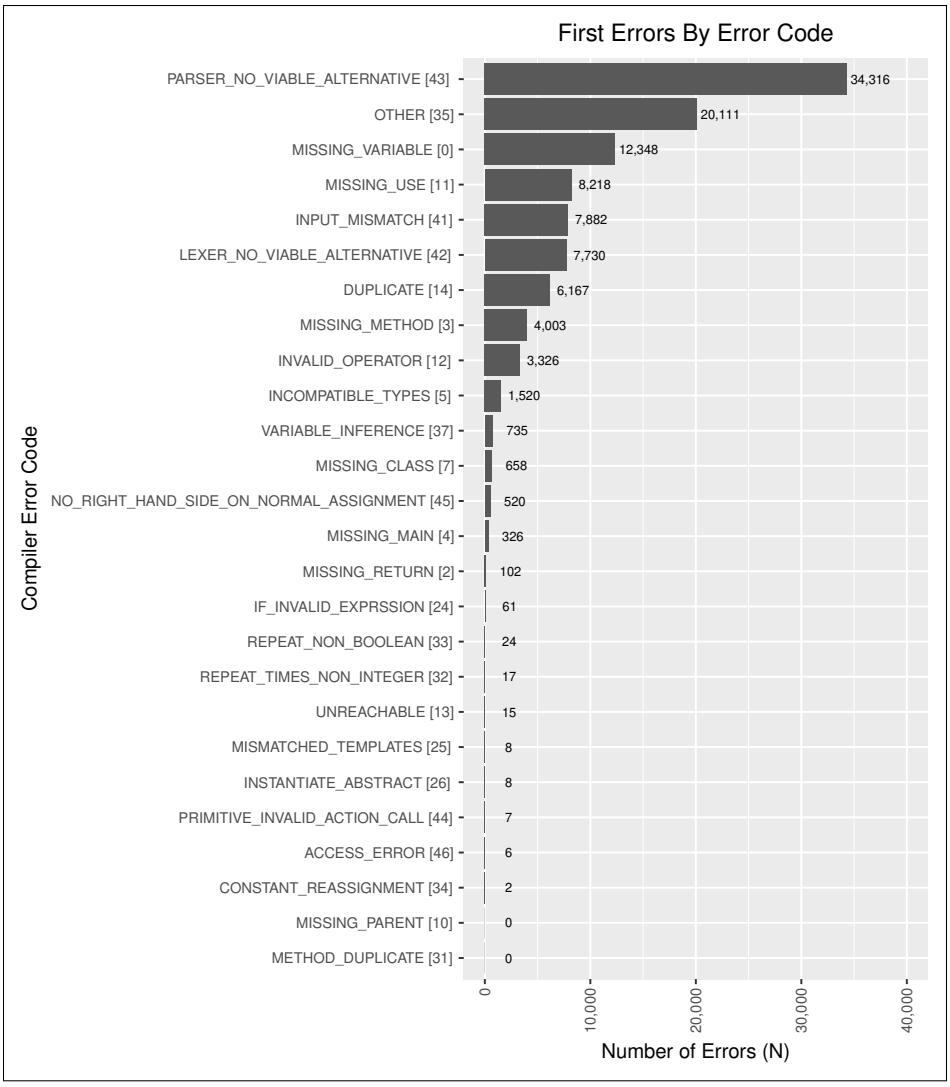


Figure 6.5: Total Errors by Compiler Error Code - First Error.

ordering for each group. (Figure 6.7). The top ten errors accounted for 98.0% of all errors in this dataset and the top error code accounted for 39.7% of the total errors. Although the top ten error codes were the same ten error codes in both the First Error and All Errors groups, the ranked ordering of the error codes were different, as shown in Table 6.5. The biggest movement up the ranking from the First Error data set to the All Errors data set was `LEXER_NO_VIABLE_ALTERNATIVE`, which increased by 32.6% of errors. The biggest movements down in ranking was `PARSER_NO_VIABLE_ALTERNATIVE` which decreased by 18.1% of errors.

6.4.1 Token Signature Frequencies

We grouped and counted the signatures and then sorted them in descending order. The top 20 most common signatures overall comprised 28.8% of the total errors. The rank, error code, token signature, token map,

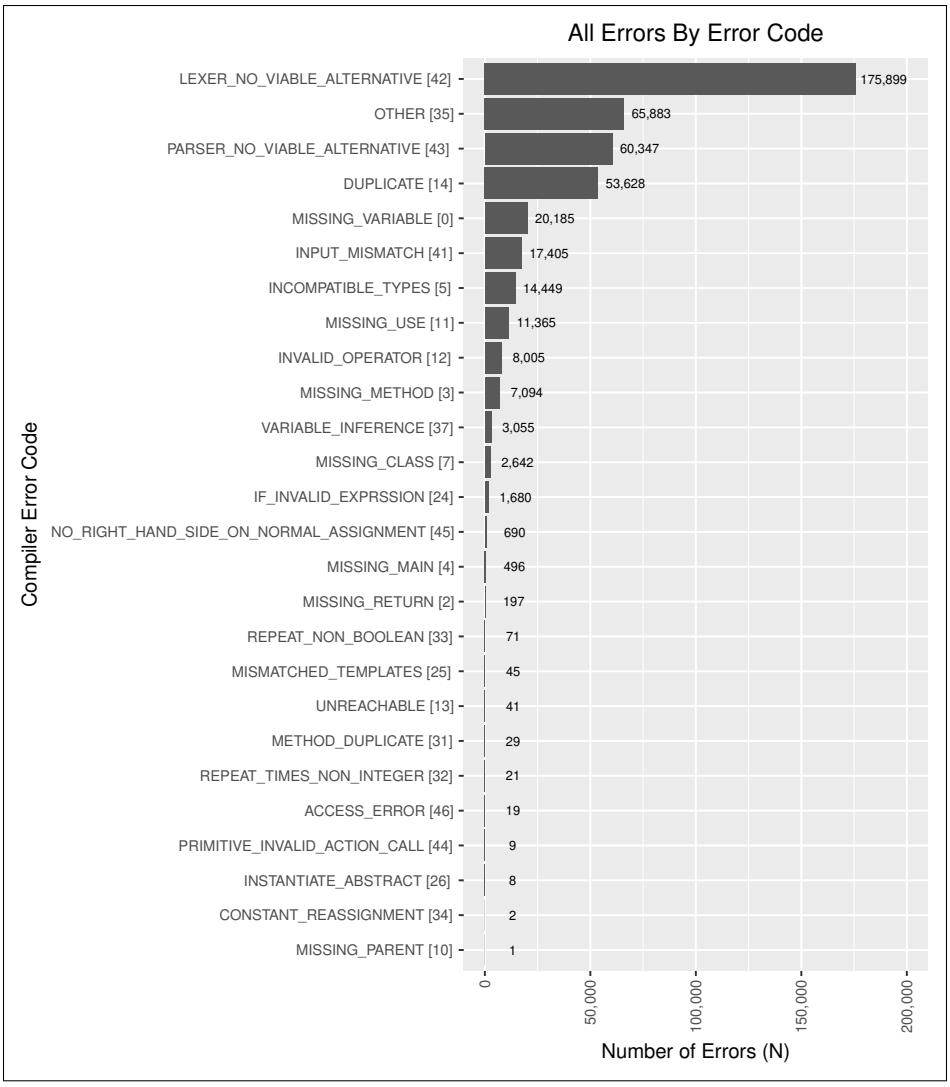


Figure 6.6: Total Errors by Compiler Error Code - All Errors.

frequency (N) and percentage of all errors are depicted in Table 6.6 for the 25 most common signatures causing compiler errors.

We also calculated the frequency of the top ten errors for each of the top six error types as listed in Figure 6.3, which are presented in Figure 6.10. The top 5 errors for these top 5 error types (a total of 25 token signatures) represent 25.3% of the total 108,110 errors in the database. The chart demonstrates that the error concentration of the top errors in each category is high.

6.5 Discussion

Overall the error frequency data we observed in the Quorum data repository analysis displayed remarkable similarity to published results for the top ten errors from six other studies cited by Becker et al. [Bec16],

Error Code	Error Code Description	Rank of First Error	Rank of All Errors	Rank Change	Pct. Change
43	PARSER_NO_VIABLE_ALTERNATIVE	1	3	-2	-18.1%
35	OTHER	2	2	0	-3.7%
0	MISSING_VARIABLE	3	5	-2	-6.9%
11	MISSING_USE	4	8	-4	-5.0%
41	INPUT_MISMATCH	5	6	-1	-3.4%
42	LEXER_NO_VIABLE_ALTERNATIVE	6	1	5	32.5%
14	DUPLICATE	7	4	3	6.4%
3	MISSING_METHOD	8	10	-2	-2.1%
12	INVALID_OPERATOR	9	9	0	-1.3%
5	INCOMPATIBLE_TYPES	10	7	3	1.9%
37	VARIABLE_INFERENCE	11	11	0	0.0%
7	MISSING_CLASS	12	12	0	0.0%
45	NO_RIGHT_HAND_SIDE_ON_NORMAL_ASSIGNMENT	13	14	-1	-0.3%
4	MISSING_MAIN	14	15	-1	-0.2%
2	MISSING_RETURN	15	16	-1	0.0%
24	IF_INVALID_EXPRSSION	16	13	3	0.3%
33	REPEAT_NON_BOOLEAN	17	17	0	0.0%
32	REPEAT_TIMES_NON_INTEGER	18	21	-3	0.0%
13	UNREACHABLE	19	19	0	0.0%
25	MISMATCHED_TEMPLATES	20	18	2	0.0%

Table 6.5: Comparison of Frequency by Error Codes - First Error vs. All Errors

Rank	Error Code	Token Signature	Token Map	N	%
1	0	1 65	output ID	5,796	5.4%
2	43	65	ID	5,287	4.9%
3	11	65 65	ID ID	5,122	4.7%
4	43	NULL	NULL	4,305	4.0%
5	43	65 66	ID STRING	2,920	2.7%
6	35	66	STRING	2,459	2.3%
7	12	65 65 44 63	ID ID = INTEGER_LITERAL	2,192	2.0%
8	43	39 65 42 65 42 65	use ID . ID . ID	2,145	2.0%
9	43	60	end	1,998	1.8%
10	43	35 65 44 63	integer ID = INTEGER_LITERAL	1,785	1.7%
11	3	65 56 57	ID ( )	1,627	1.5%
12	0	59 65	if ID	1,336	1.2%
13	41	1 66	output STRING	1,238	1.1%
14	0	35 65 44 65 52 65	integer ID = ID / ID	1,152	1.1%
15	14	37 65 44 66	text ID = STRING	822	0.8%
16	43	1	output	792	0.7%
17	14	36 65 44 18 56 36 43 66 57	number ID = cast (number , STRING)	785	0.7%
18	41	65 65 65	ID ID ID	736	0.7%
19	12	65 44 65 49 64	ID = ID + DECIMAL_LITERAL	625	0.6%
20	43	1 65 65	output ID ID	599	0.6%
21	41	33 65	action ID	588	0.5%
22	0	36 65 44 18 56 36 43 65 57	number ID = cast (number , ID)	584	0.5%
23	0	65 34 65 56 66 57	ID : ID (STRING)	571	0.5%
24	43	63	INTEGER_LITERAL	539	0.5%
25	11	65 65 44 66	ID ID = STRING	503	0.5%

Table 6.6: Top 25 Most Common Token Signatures Causing Errors Overall.

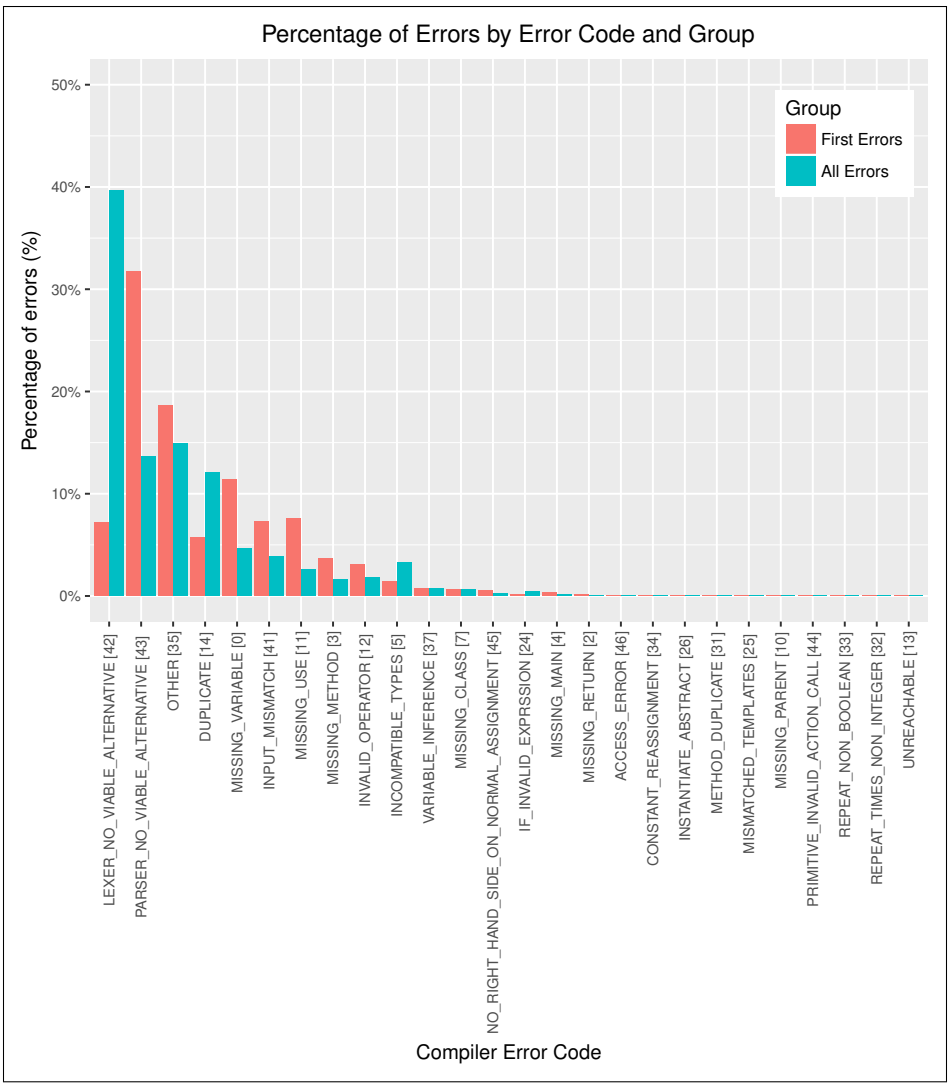


Figure 6.7: Error Frequency By Group - First Error vs. All Errors

	Error 43		Error 35		Error 0		Error 11		Error 41		Error 42	
Rank	**N**	**%**	**N**	**%**	**N**	**%**	**N**	**%**	**N**	**%**	**N**	**%**
1	4,854	14.1%	1,296	6.4%	2,877	23.3%	2,328	28.3%	511	6.5%	1,726	22.3%
2	2,891	8.4%	647	3.2%	1,287	10.4%	1,837	22.4%	476	6.0%	944	12.2%
3	1,669	4.8%	485	2.4%	1,039	8.4%	750	9.1%	450	5.7%	347	4.5%
4	762	2.2%	459	2.3%	417	3.4%	260	3.2%	401	5.1%	324	4.2%
5	519	1.5%	361	1.8%	366	3.0%	244	3.0%	333	4.2%	270	3.5%
6	463	1.3%	331	1.6%	350	2.8%	201	2.4%	298	3.8%	165	2.1%
7	462	1.3%	324	1.6%	289	2.3%	183	2.2%	112	1.4%	164	2.1%
8	440	1.3%	294	1.5%	232	1.9%	180	2.2%	111	1.4%	126	1.6%
9	372	1.1%	279	1.4%	192	1.6%	154	1.9%	73	0.9%	119	1.5%
10	299	0.9%	248	1.2%	181	1.5%	112	1.4%	70	0.9%	108	1.4%

Table 6.7: Frequency Chart of Top 10 Token Signatures of Top 6 Error Codes.

including [Bec15, BKMU14, JCC05, TRJ11, DR10, Jad05]. It is particularly interesting to us that this similarity exists with our data since the other studies were all based on the Java programming language and ours was based on Quorum.

We also note that our observations of the differences between the first error dataset and all errors dataset were also consistent with data from Becker et al. [BMT+18]. In their analysis of 21.5 million error messages in the Blackbox data, they found 28.5% of the total errors were first error messages, which is substantially similar to the 24.5% we observed in our much smaller sample. Regarding the specific major changes between the groups, we were not surprised that the top error type for all errors was a `LEXER_NO_VIABLE_ALTERNATIVE` error given the nature of cascading errors where this type of error commonly occur after a first error in a cascade.

Although we cannot directly compare the frequencies of token signatures to the frequency of compiler error codes, we found it interesting that the logarithmic decline in frequencies generally resembled the distributions of the compiler error codes. The fact that the concentration of errors is consistent lends some validity to the notion that the underlying causes of the errors, reflecting the novice programmers general understanding level, is consistent across various measurement techniques and types of errors.

6.5.1 Zipf's Law

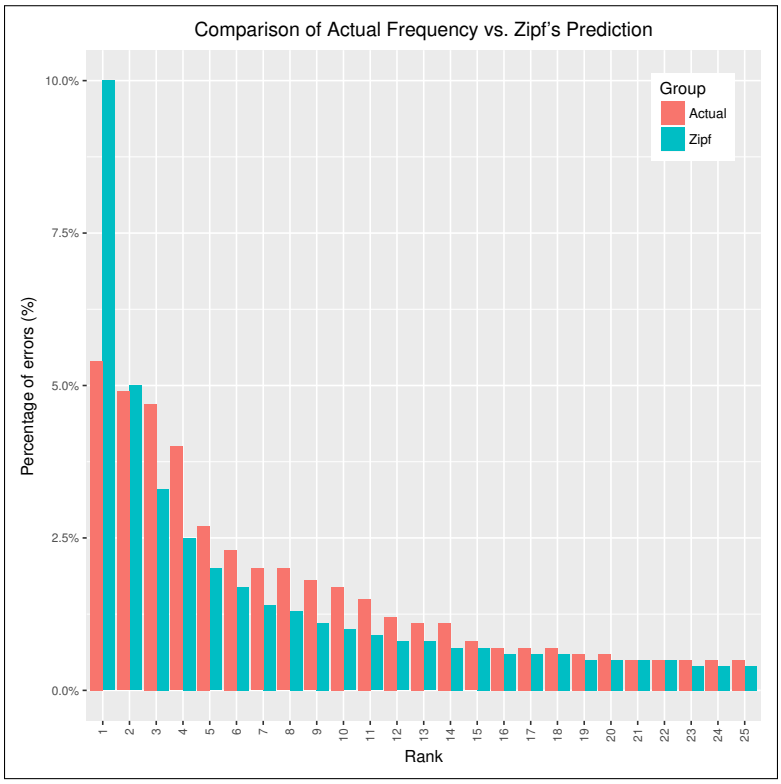


Figure 6.8: Frequency of Actual Token Signatures vs. Zipf's Law Prediction

The frequency distributions we observe in the various error code and token signature lists appeared visually to follow the predicted frequency for Zipf's Law (also called Pareto distributions or Power laws) [New05]. These distribution essentially predict that the probability of encountering the rth most common word is given roughly by $P(r) = 0.1/r$ using the calculation at Wolfram Mathworld [Wei20]. Zipf's prediction is interesting because it was originally applied to linguistic study and word frequency and recently to Java and Python computer programming languages as well. [Pri15] We applied the test to the frequency data for the top 25 token signatures overall and can see a close similarity to the Zipf Law prediction as shown in Figure 6.8

6.5.2 Exponential Decay Model

To quantify the rate of fall off in the frequency of errors at a given rank, I fit an exponential decay model to the data to satisfy the formula:

$$E(y) = \alpha e^{\beta x} + \theta$$

The non-linear regression model indicated values of $\alpha = 0.0629$ (t=29.95, $p < 0.001$), $\beta = -0.1677$ (t=-13.22, $p < 0.001$) and $\theta = 0.0036$ (t=3.573, $p < 0.01$) with a residual standard error of 0.0021 with 22 degrees of freedom. The fitted curve is shown in Figure 6.9.

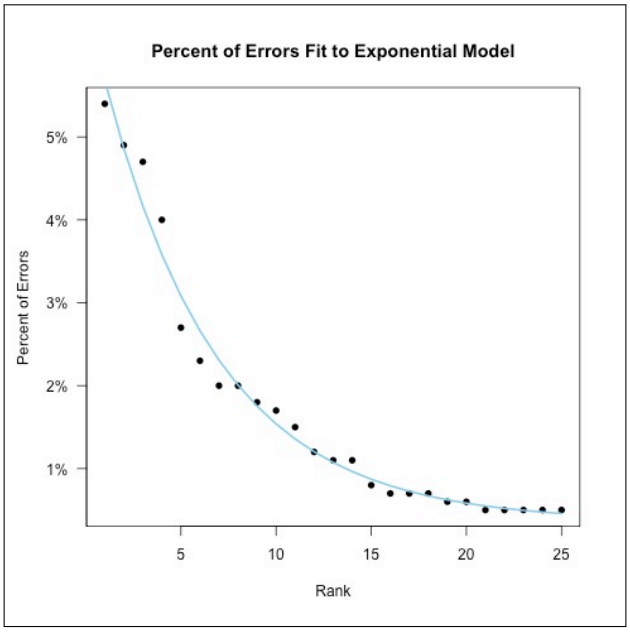


Figure 6.9: Exponential Decay Model

6.5.3 Frequency Distributions of Top Error Codes

The frequency distributions of the top token signatures for the six largest compiler error codes shown in Table 6.7 is inspired by a comparison done by Becker [Bec15] of the top 10 Java errors from six different studies

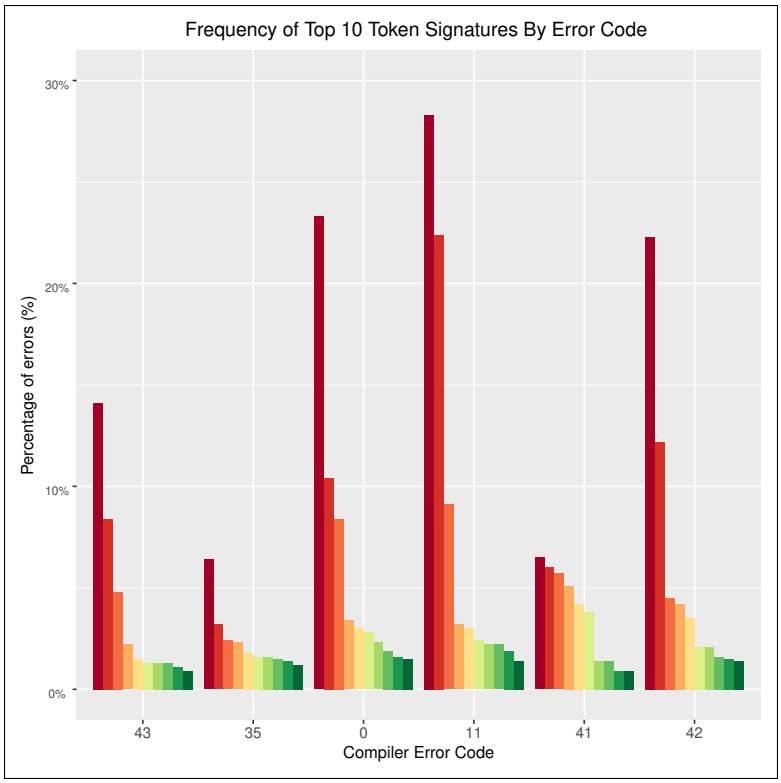


Figure 6.10: Frequency Graph of Top 10 Token Signatures of Top 6 Error Codes

spanning several years, which the ITiCSE working group noted had "very strong similarities" [BDP+19] between the distributions. Our Quorum data for the frequency distributions of token signatures is both i.) similar to Becker's observation of Java errors in different studies and ii.) similar across different compiler error codes within Quorum. The implication suggested by the ITiCSE group is that this similar pattern "(beyond an interesting and possibly useful way to compare languages) is an measuring what languages give more distinctive programming error messages." One of the goals of the token signature feedback system is to identify the patterns within the most concentrated errors to provide more granular distinctions of the root causes of those errors as the group suggests.

Chapter 7

Rules

In this chapter, we will: i.) describe the technical implementation and methodology behind the rules engine, ii.) give examples of rules based on commonly occurring token signature patterns we observed in the Quorum database and iii.) evaluate the Token Signature Technique as a compiler error enhancement technique. The purpose is to give the reader a sense of the types of errors that occurred most commonly and to show how the technique could be used in specific cases to identify and categorize root patterns and suggest corrections or hints in those situations.

7.1 Rules Engine Methodology

Token Signatures are not like compiler errors, they are single strings embedded within a user's mistake. We briefly describe here how these strings, generated from the locality garnered by Adaptive LL * [PF11] are generated. The technical implementation of the rules engine for the Token Signature Technique occurs at a low level in the architectural model presented in the ITiCSE paper [BDP+19]. After the lexical analysis occurs in the first compiler pass, the token information is retained into the next phase as the compiler begins to parse the tokens into a grammar it understands. If the compiler fails to find a valid parse tree, it immediately triggers an error manager and passes the error manager all the information that the parser has about the state of the parse as well as the complete lexical analysis. It is at this point in the first error in the second compiler pass that the Token Signature Technique resides. As described in Section 6.3.2, the compiler error manager can generate a token signature for analysis with just the information from the lexical stream and the locality information from where the parser found a problem. This token signature, the locality and error type information and the full token stream from the lexer is passed to the rules engine for analysis.

7.1.1 Solution for the "String" Example

By way of example in another context, the ITiCSE paper presents a common situation, at least based on the Quorum data, as a motivating example giving a line of Java code containing an error:

```
public static void main(string[] args) {
```

with an error message stating "`cannot find symbol`" with the point of error identified at the letter `s` in the word `string` on the line we extracted above. We agree that the compiler generated message in this case does not adequately reflect the user error.

The approach that the Token Signature method would take would be to break down the tokens in the line using token numbers (which we will assign sequentially for illustration) to generate a Token Signature:

User	Token Name	Token Num
public	PUBLIC	8
static	STATIC	32
void	VOID	70*
main	ID	65
(	LPAREN	56
string	ID	65
[	LEFT_SQR_BRACE	54
]	RIGHT_SQR_BRACE	55
args	ID	66
)	RPAREN	57

** Note that the `void` keyword does not exist in Quorum*

The token signature would be "`8 32 70 65 56 65 54 55 66 57`" and the compiler would pass the locality to the rules engine so it would know that the error occurred at token number 6 which is an ID. The rules engine would recognize that the signature matches a correct signature but that ID at token number 6 has a problem. The problem could be one of several things, including an undeclared ID, misspelled ID or a case error to a known keyword or library type. A capitalization and spell check would be among the first tests since this type of error is very common among novices. Knowing that capitalizing the first letter makes the line valid, the error message could suggest the exact solution in this case. We note that this type of error represents a significant portion of the third highest overall error and a spell/capitalization check is the first rule.

7.2 Exposition of Errors and Rules

Having completed this analysis, we can focus the efforts of the rules and hints construction on the top errors in each category to try to generate better messages and potential solutions. Remembering that the key

priority with this technique is not to try to definitively resolve the source of every possible problem, but rather to provide potentially useful feedback to a novice programmer at the point of error with the ultimate goal of improving learning and productivity.

7.2.1 Signature 1: "1 65"

Token Signature	1 65
Token Map	output ID
N Overall	5,796
Rank Overall	1
% of Overall	5.4%
Error Code	0
Error Description	MISSING_VARIABLE
N in Error Code	2,877
Rank In Error Code	1
% of Error Code	23.3%
Error Code	42
Error Description	LEXER_NO_VIABLE_ALTERNATIVE
N in Error Code	1,726
Rank In Error Code	1
% of Error Code	22.3%
Error Code	43
Error Description	PARSER_NO_VIABLE_ALTERNATIVE
N in Error Code	463
Rank In Error Code	7
% of Error Code	1.4%

Observations:

The most common occurrences of this error are when the user either i.) has not previously declared the ID or ii.) misspelled the ID they intended to use or iii.) had a capitalization error on the ID.

Another common cause of this error is when the ID is intended to be a STRING_LITERAL, but the programmer forgot to enclose the word in quotation marks. Since Quorum does not recognize a single quotation mark as a special token, it was common for users to attempt to form a string using them. A simple program like `output 'hello world'` would give a token signature of `1 65` in Quorum and cause this error.

Rules / Suggestions:

- Check the token stream in previous line to see if a spell check or capitalization check could suggest an ID.

- Suggest to the user that they may need to declare and assign a value to the ID before attempting to output a value if there is no identifiable ID previously.
- Suggest that the ID may have been intended to be enclosed in quotation marks if there is no identifiable ID previously.
- Suggest that the ID needs to be enclosed in *double* quotation marks instead of *single* quotation marks if the ID token contains single quotation marks.

7.2.2 Signature 2: "65"

Token Signature	65
Token Map	ID
N Overall	5,287
Rank Overall	2
% of Overall	4.9%
Error Code	43
Error Description	PARSER_NO_VIABLE_ALTERNATIVE
N in Error Code	4,854
Rank In Error Code	1
% of Error Code	14.7%
Error Code	11
Error Description	MISSING_USE
N in Error Code	183
Rank In Error Code	7
% of Error Code	2.2%
Error Code	42
Error Description	LEXER_NO_VIABLE_ALTERNATIVE
N in Error Code	164
Rank In Error Code	7
% of Error Code	2.1%

Observations:

The word `output` is commonly on a line by itself or combined with an identifier, for example `outputa` or `outputseconds`.

Another common mistake is to incorrectly call a method of a library, for example `PlayUntilDone` without using the correct Quorum syntax `ID:PlayUntilDone`.

Rules / Suggestions:

- Check if the ID is `output` (or a common misspelling of it) and then give them programmer a hint with other declared variables in the scope or give an example of the format `output STRING_LITERAL`.
- Check if the ID *contains* the word `output` and if so, cut the token into two words as a hint.
- Check if the ID was a previously declared variable and suggest `output` + the ID.
- Check for a `use` statement earlier in the token stream and then search if the ID matches (or is a close misspelling of) a method of the library class, then hint for the correct usage.

7.2.3 Signature 3: "65 65"

Token Signature	65 65
Token Map	ID ID
N Overall	5,122
Rank Overall	3
% of Overall	4.7%
Error Code	11
Error Description	MISSING_USE
N in Error Code	2,328
Rank In Error Code	1
% of Error Code	28.3%
Error Code	14
Error Description	DUPLICATE
N in Error Code	1,697
Rank In Error Code	1
% of Error Code	27.5%
Error Code	42
Error Description	LEXER_NO_VIABLE_ALTERNATIVE
N in Error Code	944
Rank In Error Code	2
% of Error Code	12.2%

Observations:

These signatures contain many of the `output` errors already described, especially with mis-spellings and capitalization errors. For example, the line: `Output ID` will generate an error code 14 (DUPLICATE) message ("Variable ID is already defined") for a previous and correctly declared ID because the compiler interprets `Output` as another type and the line as an attempt to declare a duplicate instance of the non-existent `Output` class instead of a likely intent by the user to use token string `1 65 : output ID`.

The same code with the same signature can also generate different error codes and messages depending on previous lines of code. For example `Output ID` will generate an error code 11 (MISSING_USE) message ("I could not locate a type named ID. Did you forget a use statement?") if ID has not been declared or a 14 (DUPLICATE) message ("Variable ID is already defined") if the ID has been previously declared. Either way though, token signature analysis can help get to the root cause of the actual error in both of these cases (incorrect capitalization), when neither other message actually correctly identifies the problem.

A related but separate common situation is the attempted declaration of a Library call, for example `Drawable bunny` with similar capitalization or mis-spellings of the first ID.

Rules / Suggestions:

- For errors of type 14 (DUPLICATE) check for a misspelling
- Check if the first ID is a Library method that has a `use` statement already declared.
- Check for a missing `use` statement for a known

7.2.4 Signature 4: NULL

Token Signature	NULL
Token Map	NULL
N Overall	4,305
Rank Overall	4
% of Overall	4.0%
Error Code	43
Error Description	PARSER_NO_VIABLE_ALTERNATIVE
N in Error Code	2,457
Rank In Error Code	3
% of Error Code	7.5%
Error Code	35
Error Description	OTHER
N in Error Code	1,259
Rank In Error Code	2
% of Error Code	6.3%
Error Code	41
Error Description	INPUT_MISMATCH
N in Error Code	511
Rank In Error Code	1
% of Error Code	6.5%

Observations:

An NULL line registering an error was most commonly caused in this dataset by a programmer typing an incomplete line of code followed by blank line which the parser did not know how to interpret. Over 1,400 of these cases are 2 line programs with a blank second line.

A missing `end` statement with a blank last line is the primary cause of the 511 error code 41 errors. In these cases it is difficult to determine where the `end` belongs if the program has nested statement.

Rules / Suggestions:

- Pull the previous, non-empty line to determine a possible error or missing next token causing the incomplete parse.
- For missing `end` statements, a hint suggesting specific lines of code that might be missing ends, such as lines containing the tokens: `if`, `repeat`, or `action`.

7.2.5 Signature 5: "65 66"

Token Signature	65 66
Token Map	ID STRING
N Overall	2,920
Rank Overall	5
% of Overall	2.7%
Error Code	43
Error Description	PARSER_NO_VIABLE_ALTERNATIVE
N in Error Code	2,891
Rank In Error Code	2
% of Error Code	8.8%
Error Code	42
Error Description	LEXER_NO_VIABLE_ALTERNATIVE
N in Error Code	15
Rank In Error Code	52
% of Error Code	0.2%
Error Code	35
Error Description	OTHER
N in Error Code	9
Rank In Error Code	314
% of Error Code	0.0%

Observations:

1,741 (59.6%) of these signatures were the results of mis-spellings or incorrect capitalization of the keyword `output` and another 449 (15.4%) incorrectly categorized the word `Say`, so almost three quarters of these errors could be fixed with a capitalization / spell check rule on the first token. Other common errors included programmers using the word `print` (2.5%) or `input` (2.8%) with a string.

Rules / Suggestions:

- Capitalization and spell check of the first token.

7.2.6 Signature 6: "66"

Token Signature	66
Token Map	STRING
N Overall	2,459
Rank Overall	6
% of Overall	2.3%
Error Code	43
Error Description	PARSER_NO_VIABLE_ALTERNATIVE
N in Error Code	1,669
Rank In Error Code	4
% of Error Code	5.1%
Error Code	41
Error Description	INPUT_MISMATCH
N in Error Code	450
Rank In Error Code	3
% of Error Code	5.7%
Error Code	35
Error Description	OTHER
N in Error Code	324
Rank In Error Code	8
% of Error Code	1.6%

Observations:

A manual review of samples indicates that in many instances, the programmer appears to have intended the string to be output by the computer, likely as an `output` or a `say` statement, but it is difficult to know for certain. Occasionally program control flow words like `else` and `end` where put in quotation marks for no apparent reason. At times, the string appears to have been intended as a code comment, but that is based on a qualitative assessment only.

Rules / Suggestions:

- Provide hints for `output STRING` and `say STRING`
- Remove quotation marks from known keywords if the STRING is a single word (such as `else` and `end`).
- Suggest that the STRING might be a comment.

7.2.7 Signature 7: "65 65 44 63"

Token Signature	65 65 44 63
Token Map	ID ID = INTEGER_LITERAL
N Overall	2,192
Rank Overall	7
% of Overall	2.0%
Error Code	11
Error Description	MISSING_USE
N in Error Code	1,837
Rank In Error Code	2
% of Error Code	22.4%
Error Code	12
Error Description	INVALID_OPERATOR
N in Error Code	186
Rank In Error Code	4
% of Error Code	5.9%
Error Code	14
Error Description	DUPLICATE
N in Error Code	146
Rank In Error Code	7
% of Error Code	2.4%

Observations:

These errors are almost always an attempted assignment of integer to an ID, where the first token presents as an ID instead of as a 35 INTEGER_KEYWORD. In the sample, the first token was some variation of `Int...` or `int` 1,894 times (86.4%). Other examples included the word `decimal` instead of `number` or some attempt to make an assignment like `set x = 1`.

Rules / Suggestions:

- Capitalization and spell check of the first token.

- Suggest a fix for assignment in common languages (`int x = 1;` or `set x = 1`).
- Suggest a fix for other language type declarations like `float`, `double` or `decimal` to `number`.

7.2.8 Signature 8: "39 65 42 65 42 65"

Token Signature	39 65 42 65 42 65
Token Map	use ID . ID . ID
N Overall	2,145
Rank Overall	8
% of Overall	2.0%
Error Code	11
Error Description	MISSING_USE
N in Error Code	750
Rank In Error Code	39.1
% of Error Code	%
Error Code	35
Error Description	OTHER
N in Error Code	647
Rank In Error Code	3
% of Error Code	3.2%
Error Code	41
Error Description	INPUT_MISMATCH
N in Error Code	476
Rank In Error Code	2
% of Error Code	6.0%

Observations:

The error code 11 (MISSING_USE) errors generally occur when there is a spelling error in one of the IDs in the library chain. For example, `use Libraries.System.Files` with any of the 3 words spelled wrong, not in the correct order or a non existent library will trigger this error

The error code 35 (OTHER) errors generally occur when the user types a `use` statement somewhere other than at the beginning of the program. A large portion of the samples in this case contain scaffolded code of the Quorum basic game engine, but the programmer added the `use` statement in the wrong place.

The error code 41 (INPUT_MISMATCH) errors generally occurs when the compiler is expecting and EOF, but finds a `use` statement below the end of the `Main method`. This occurred in the Quorum dataset with some frequency because of the scaffolded code of the Quorum basic game engine. See Figure 7.1 for an example of the misplaced code in lines 14-17.

```
1  use Libraries.Game.Game
2  //INSERT YOUR "USE" STATEMENTS HERE
3
4  class Main is Game
5      action Main
6          StartGame()
7      end
8      action CreateGame
9      //INSERT YOUR CODE HERE
10
11
12     end
13 end
14 use Libraries.Sound.Audio
15 Audio.clickSound
16 clickSound:Load("media/astro/click.wav")
17 clickSound:Play()
```

Figure 7.1: Example Misplaced Code in Game Engine Scaffolding

Rules / Suggestions:

- Check chain of IDs in the library calls for spelling errors and correct or likely matches to the Quorum Library.
- If a `use` statement is detected with a code 41, suggest that the use statement needs to be at the top.
- Identify if the scaffolded Quorum Game engine code is unmodified from the prior token patterns and if so, suggest to the user that the code they typed after the game engine code is in the wrong place. A `use` statement could be identified for the beginning insertion point (after line 2 in Figure 7.1)and the other code in one of the game methods (such as after line 9 in Figure 7.1).

7.2.9 Signature 9: "60"

Token Signature	60
Token Map	end
N Overall	1,998
Rank Overall	9
% of Overall	1.8%
Error Code	35
Error Description	OTHER
N in Error Code	1,296
Rank In Error Code	1
% of Error Code	6.4%
Error Code	41
Error Description	INPUT_MISMATCH
N in Error Code	401
Rank In Error Code	4
% of Error Code	5.1%
Error Code	43
Error Description	PARSER_NO_VIABLE_ALTERNATIVE
N in Error Code	263
Rank In Error Code	13
% of Error Code	0.8%

Observations:

The `end` statement is challenging for any type of compiler enhancement, because it is difficult the users intent is often ambiguous and a token signature line of `60 : end` is the way it should always occur in Quorum on a line by itself. The only real option for rules for this error are to look at control flow constructs that may be open, like `if`, `repeat`, or `action`. Specifically, the hint engine could suggest the last layer of nesting to check if that is intended behavior. Particularly with novices, there are lots of examples where they forget to close their previous code block as opposed to intentionally nesting. Nonetheless, any hints here would have to be carefully phrased as hints because it is very difficult to provide code correction suggestions.

In the case of error code 43 (PARSER_NO_VIABLE_ALTERNATIVE) the error generally occurs when there is an incomplete or incorrect line of code in the previous line.

Rules / Suggestions:

- For missing `end` statements, a hint suggesting specific lines of code that might be missing ends, such as lines containing the tokens: `if`, `repeat`, or `action`.

- For error code 43, revert to the previous line of code and re-run the token signature analysis.

7.2.10 Signature 10: "35 65 44 63"

Token Signature	35 65 44 63
Token Map	integer ID = INTEGER_LITERAL
N Overall	1,785
Rank Overall	10
% of Overall	1.7%
Error Code	14
Error Description	DUPLICATE
N in Error Code	1,213
Rank In Error Code	2
% of Error Code	19.7%
Error Code	42
Error Description	LEXER_NO_VIABLE_ALTERNATIVE
N in Error Code	347
Rank In Error Code	3
% of Error Code	4.5%
Error Code	43
Error Description	PARSER_NO_VIABLE_ALTERNATIVE
N in Error Code	182
Rank In Error Code	19
% of Error Code	0.6%

Observations:

The base signature token here is correct, which is the declaration and assignment of an INTEGER_LITERAL to an integer ID. The most common error code is 14 (DUPLICATE) and appears to be and accurate reflection of the root cause. A casual observation seems to suggest that novices may think that using the same name is OK if they use a different type since that situation occurs regularly in the top error code.

Rules / Suggestions:

- Use the lexical information available at the time of the error to point out the line number and type of the previous declaration. For example; `I noticed you are trying to declare an integer variable called ID, but you previously declared a number variable on line X with the same name.`
- Teaching explanation for not being able to reuse IDs on different types.

7.3 General Observations

Having a suggestion list available of previously declared IDs during programming as a form of code completion or hint suggestion or to use for spell checking) would have a big impact. This is implemented by Microsoft with their IntelliSense technology for code completion, parameter info and lists. [Mic20]

An extremely common cause of various kinds of errors is the incorrect capitalization of the first letter of a keyword. One of the first checks of any hint or error engine should be simple testing if an unrecognized token would be recognizable and parseable if it was lower cased. This check should be followed by a spell checker for known mis-spellings of keywords and standard library classes. Of the 108,110 error files, 3,461 (3.2%) of the error lines across various token signatures started with the token `Output` which was classified as an ID instead of the keyword `output` because of a simple capitalization error, similar to the example highlighted by the ITiCSE working group [BDP+19].

There are numerous teaching opportunities which could be exploited through this form of program analysis. We have a specialized example with our data set because the vast majority of our code samples come from specific targeted curriculum pages on the Quorum website. It is easy to observe patterns with a naked eye qualitative examination with the aid of the knowledge of the task assigned from a given page and the foreknowledge of the correct answer. Future research endeavors along this line by specialized educational researchers could enable the creation of a customized learning capability.

7.4 Threats to Validity

In this section we will use the five point general technical challenges to effective error messages outlined in the ITiSCSE paper [BDP+19] as a framework for an examination of the threats to validity of the Token Signature approach. We will generally address the strengths and weakness of the approach in this context using the nomenclature and definitions in the paper.

1. **The Completeness Problem** - It is impossible to ever reach 100% error detectability at the compiler level if for no other reason than the universal understanding that it is impossible to definitively know the users intent. This approach is not geared around detecting all possible errors at compile time and then providing messages for every eventuality, but instead focuses on efficiently correcting errors already identified by the compiler using a heuristic approach as a positive step on the continuum towards "better messages".

2. **The Locality Problem** - My first version of a rules engine is decidedly localized in its nature and the general shortcomings identified in the ITiCSE paper regarding locality will be equally challenging with this approach. That being said, there are some cases and error types where token based rules could provide suggestion-based advice on the type of error that a programmer should look in order to shed light on the nature of the problem.

3. **The Mapping Problem** - The mapping problem is a limited constraint for this approach, especially with the Quorum language compiler, since the rules engine works on the actual token stream at face value, not any type of mapped or pre-transformed code.

4. **The Engineering Problem** - The impact of an engineering challenge is reduced but not eliminated in this approach for reasons stated in the Completeness response. The enhancement occurs after an error is detected in second pass of parsing, not in higher architectural levels of type checking, code generation or interpreting. This approach seeks to provide solutions and hints to correct problems, not to perform additional engineering work during complicated phases of compilation. Notably, the rules engine could easily be threaded and offloaded upon the occurrence of the first error while the compile continued its work in the later stages of compilation. The threads would need to synchronize before final error reporting, but the results of the rules engine will not influence or hold up continued compilation.

5. **The Liveness Problem** - One of the key design features of this approach was the goal of providing real-time or "live" hints in the context of a red squiggly line or highlight during coding inside the IDE. The Quorum compiler and Quorum Studio IDE have a co-ordinated hint capability built-in which we can utilize. The processing load for this from an engineering perspective is similar to the general engineering problem because the computation and rules engine can be threaded to not interfere with anything else that is going on. With the focus on the first hint, the processing requirement is further limited to a single error until the user corrects it and encounters another.

7.4.1 Technical Progress Requirement

As a final discussion point, we agree with the conclusion of the ITiCSE working group that a necessary requirement for progress to be achieved in compiler error enhancement, is the utilization of the latest developments in technologies such as machine learning and artificial intelligence and in Human Computer Interaction to improve the overall readability of error messages and suggestions. The token signature approach was designed with exactly the type of heuristic pattern matching machine learning might provide within the scope of an extensible rules engine. Since the computation is fully automated and "mechanical" from an engineering standpoint, a machine learning or optimization algorithm can reasonably be expected to improve the accuracy of the rules engine over time based on observed data. For example, a system like this can observe which suggestions are actually selected by users given certain token patterns, and in the next round offer better hint ranking similar to a search engine. With more extensive access to user level data, the hint ranking and suggestions could be tailored at a user level based on observed experience.

7.5 Considerations for Language Design for the Quorum Language

One of the foundations of the Quorum language is that it is evidence-based, meaning that the language design choices should be supported by empirical studies where ever they are available to improve the language (in terms of ease of use). The repository analysis in Chapters 6 and 7 have implications for language design based on the actual usage of human beings on the curriculum offered on the website. A few areas for consideration that stood out to me that could be data-mined through the Analytics Dashboard or database for further investigations follow:

- **Single Quotation Marks** - Not only is it difficult as an experienced programmer to work with only one set of quotation marks (in working with the JSON library for example), but there were numerous errors and attempts by novices to use single quotation marks in various places. 6,887 error files had a single quotation mark somewhere in the code body. The problem likely stems from the fact that most other languages use single and double quotation marks to identify strings and that fact that Quorum does not is unusual and causes confusion and errors.

- **Input** - The keyword `input` is involved with 6,572 errors (6.1% of overall)in our sample. There were so many misuses around how to use it including i.) the need to cast it to a non-text type, ii.) the format requirements of a string inside parenthesis, iii.) the necessity to use the text type in declaring the text variable even though input always returns text, iv.) the inability to return a numeric type directly and v.) confusion with other common ways that languages accept user input at the console.

- **Repeat** - The `repeat` flow control construct in Quorum didn't generate as many errors as input, but there were still 1,731 where it was involved. The errors should be mostly teachable with token signature style hints, but the design itself may need to be looked at because the patterns in the errors were common, such as `repeat until x > 50 times`, `repeat until x is 50`, python-style `repeat 5,8 times`, `repeat while a` (a is not boolean), `repeat Add() 5 times` (or Subtract, etc.), `repeat add 5`, `repeat * 5` or declaring a variable on the repeat line like a C-style for loop `repeat while integer a < 5)`.

- **Cast** - The keyword `cast` is involved with 5,427 (5.0% of overall) errors in the database. Common mistakes included: i.) attempting to assign the variable being cast back on to itself with a different type `integer a = cast(integer, a)` (where a is declared previously as a different type), ii.) using to cast the variable to its same type but assign the result to a new type, iii.) `integer a = cast(text, ''hi'')`, or iv.) forgetting an apostrophe between the parameters of the cast method. The compiler errors are generally fairly accurate for these, but there are a lot of common errors that may suggest examination.

7.6 Conclusion

The goal of this analytical exercise was to develop an automated approach that can be used to provide better feedback to programmers for the most common errors they make when they are learning to program. Through an analysis of the database of 108,110 errors, the token signature approach was useful in identifying root causes of errors across different types of compiler errors. In particular, we were able to identify patterns in the most vague ANTLR error categories (Error Codes 43-PARSER—_NO_ALTERNATIVE, 35-OTHER, 41-INPUT_MISMATCH, AND 42-LEXER_NO_ALTERNATIVE) which could be used to generate suggesting rules to improve feedback. The automated technique is well suited for the application of other sophisticated technologies in the areas of search, spell checking, grammar checking, and machine learning for pattern matching.

Chapter 8

Conclusion

8.1 Summary

The two randomized controlled trials conducted for this dissertation provided hundreds of thousands of code samples across dozens of tasks, while the online Quorum database captured over 294,631 code samples. This rich repository of data required automated analytic tools to process. The token-based approach coupled with an analytic dashboard provided a platform to begin to understand and identify patterns of behaviors among the student subjects. This dissertation will conclude by i.) reviewing the key findings of these studies, ii.) examining the software development effort to reach it and finally iii.) discuss where this project can go from here.

8.1.1 Concurrency Paradigms using TAMs

This re-analysis of the data from a Randomized Controlled Trial on two popular concurrency programming paradigms using improved token accuracy mapping provided new information on the root causes of why students had difficulty with a particular task. We also explored that while Token Accuracy Maps have limitations, they proved useful as a tool to gain insight into the overall accuracy of students when working on these tasks, as well as a mechanism to investigate which specific parts of the program were problematic for the students. TAMs might prove useful in future studies to track participant progress through tasks by utilizing time-slice data and to find more information about which parts of programming language syntax are causing problems for programmers.

8.1.2 GPU Study on Abstraction

This study provides evidence from a randomized controlled trial that computer science students learning GPU programming for the first time performed worse using a higher level abstraction paradigm (Thrust) compared

to a lower level paradigm (CUDA). The results also show that even the most simple version of a fundamental task of parallel computing (offloading a basic vector addition computation to a coprocessor) is challenging for students. We examined 4 research questions which corresponded to 4 specific different abstractions in GPU programming for i.) memory allocation, ii.) array iteration, iii.) memory copy to/from host/coprocessor and iv.) an offloaded kernel routine. In the 5 tasks where abstractions were tested, we observed that the low-level CUDA abstraction paradigm tested equal to or better than the high-level Thrust abstraction paradigm in every case among student learners. While our results are not a comprehensive or conclusive determination of superiority for either CUDA or Thrust, the fine-grained examination of these specific abstractions provides interesting and potentially useful information for language designers and instructors.

8.1.3 Token Signature Technique on Quorum Repository

The analysis of 108,110 compiler errors from a novice programming database using the Token Signature Technique indicated concentrated error frequency results consistent with other large scale studies of compiler errors of student learners using Java. A detailed examination of the token signature maps for each general error category also demonstrated consistent and concentrated error frequency results. Empowered by this demonstrated error concentration for our novel analysis technique, we analyzed and explored the leading underlying error categories in search of patterns and root causes in order to identify improved error messaging and hints through a rules-based approach.

8.2 Software Development

The following is a summary of the key pieces of software that I worked to collect and capture the data for this project and to develop the automated analysis.

8.2.1 Automation of Token Accuracy Mapping Technique

The concurrency paradigm study we ran described in Chapter 4 describes the token accuracy mapping technique in a high level of detail. There were a few very complicated development challenges with that system though, including i.) the implementation of the Needleman-Wunsch string alignment algorithm and the subsequent optimizations and adjustments for software token alignments, ii.) the automated tokenizer in both Java and Quorum languages using the ANTLR parser generator, and iii.) the automated processing tool to compare code samples, generate token accuracy maps, sorting and scoring techniques and reporting. I did not invent the idea of Token Accuracy Mapping (Dr. Stefik did), but the automation of the process to allow it to be run on the thousands of code samples we collected would not have been possible without the software.

8.2.2 Web-based Human Subjects Study Testing System

I built the first iteration of this system for the concurrency study after running into problems administering and proctoring a multi-site replication study in concert with Software Carpentry, which we never managed to collect sufficient data on. I made the decision to create this so that we could more easily get more participants to do our studies at convenient times, rather than expecting them to come into labs to undertake studies when they could be observed. The effort was amazingly successful as we have now used the system for over half a dozen studies with hundreds of participants, which would not likely have happened with proctored and scheduled testing times. I'd like to thank P. Merlin Uesbeck for his help in developing additional features and using it for his studies too.

8.2.3 Curriculum Development and Implementation for Hour of Code, Tutorials and Lessons

The curriculum design effort was not software development per se, but it has been a massive effort over the years as we have continually provided new, free curriculum for novice programmers available on the Quorum site, all accessible to the blind and visually impaired community. All of this curriculum required software implementation, however, because it only exists in an online form. Once we built and implemented the online IDE to enable people to build and run their Quorum programs easily in a browser, the effort mushroomed and Quorum's impact has grown as a result.

The development effort for the Astronomy-themed Hour of Code was an original part of the IDATA grant proposal and I designed and built the entire 20 part activity, with a notable exception of the online 3D graphics engine used in the finale thanks to William Allee and Dr. Stefik. It was our second Hour of Code activity, so we had the knowledge from first experience to shape our design, but the usage from the activity was pleasantly surprising. Referring to Table 6.1, the Astronomy Hour of Code is responsible for 43.4% of the content in the Quorum repository, along with another 42.9% for the Quorum Lessons and Tutorials which I had an extensive hand in reviewing and rewriting over the years.

8.2.4 Skynet Telescope Quorum Implementation for IDATA

The software development effort required to implement the IDATA curriculum required us to build a complete system to enable a blind or visually impaired student to send commands to a robotic telescope network, which can take astronomical photos from telescopes around the world from a browser. Fortunately, the team at University of North Carolina already built the robotic telescope network (`https://skynet.unc.edu/`) with an API, but we had to build the entire network system for Quorum from scratch. In order to support the curriculum for IDATA using this network, I had to build a number of libraries and plugin's for Quorum including i.) two plugins for Quorum to enable http communication through Java for desktop computers and JavaScript for browsers to enable communication with Skynet, ii.) a full Quorum Network library modeled

on the Python Requests library, iii.) a Quorum JSON library to read and write JSON data, iv.) a Quorum Matrix library for astronomical computation and v.) a Quorum Random Number generator. It took 18 months to develop but the effort was a success.

8.2.5 Analytics Dashboard

Although our automated testing system generates hundreds of thousands of events, I have found that manually inspecting the results yields insights that help shape automated systems. I can't look at all the results, but the dashboards I've built help me to see things in context and to be able to quickly shift between tasks, users and groups to see patterns. The Token Signature method came to me while I was navigating around user errors in the dashboard that I had filtered in a certain way and I started to see patterns. I knew we wanted to apply the TAM technique on a localized basis within segments of code, but it was because of the dashboard that I saw how to do it. This is the type of behavioral analytics that I set out to try to identify so that we can improve computer science education and programmer productivity.

8.3 Future Software Development

There are a few natural ways to continue development around the Token Signature concept. The idea is at its very early stages but it could be applied very broadly over time in a number of directions.

- **Enhanced Error Message System** - The first obvious area is to build the enhanced error message system into the Quorum compiler in the Compiler Error Manager. The hooks for the system are already there and when the Compiler Error Manager is triggered, it has access to everything it needs to compute a Token Signature and put out enhanced messages.
- **Implement Rules Engine** - The next obvious area in conjunction with the first is to build an extensible rules engine that will enable additional rules to be built and implemented into the error reporting and hint generation system. The idea would be that once the Compiler Error Manager has an error and a Token Signature, it invokes the rules engine for a hint or enhanced message to see if anything is applicable.
- **Tie into Quorum Studio IDE** - The final piece of the "delivery" system for the enhanced messaging or code suggestions system would be to tie into the new Quorum Studio IDE where certain rules could generate auto-completion or auto-correction options, hints and messages for live, on the fly, assistance while programming. Its the ultimate way to provide the kind of immediate feedback to student learners that Becker et al. [BDP+19] refer to.
- **Advanced Rules Development** - After the full implementation of the "delivery" system, there are untold ways that rules could potentially be developed through additional rules, advanced pattern

recognition, machine learning, spell checking, grammar-style checking or other ways of heuristically attempting to divining the programmer's intent or common error patterns.

8.4 Future Research

There are some obvious studies that could be done after the enhanced error message and hint delivery system is built in order to evaluate its effectiveness. I'd first look to design studies similar to other comparison studies that have been conducted using a two group design, with one using standard error messages and the other using enhanced messages. I also think it would be interesting to conduct a study over the course of a semester of instruction in early stage programming classes to measure improvements in student performance with and without the enhancements. Implementing the Token Signature analysis in other languages on other compiler error repositories would also be very interesting. In particular, the Blackbox data [BKMU14] set is an excellent candidate because of both its size and the extent to which it has already been examined by other researchers.

Further research in the area of teaching and learning computer programming would also be an interesting area to explore. Taking inspiration from the computing education practitioners research wish list assembled by Denny et al. [DBC+19], using this token signature technique to study the highly rated questions "What fundamental programming concepts are the most challenging for students?" and "'How and when is it best to give students feedback on their code to improve learning?" could potentially prove very interesting with an extensible and malleable rules and hint delivery system.

Appendix A

Methods and Materials For RCT on Parallel Programming from Chapter 4

This appendix contains materials for a paper that describes a randomized controlled trial designed to empirically examine the human factors impacts of two differing concurrency paradigms by measuring student performance on three common parallel programming problems using one of the two paradigms. In addition to providing comparative empirical data of the students' performance on their first introduction to the paradigms, we sought to further develop an empirical measurement technique designed to measure token accuracy. Our long term purpose is to contribute to improving both computer science education and the future design of programming languages and approaches. Since concurrency is an increasingly important area and generally regarded as being hard to learn, a better understanding of precisely what areas are stumbling blocks may help the programming language community improve their designs and find creative approaches around problem areas. The primary contribution of this paper is the empirical data from the study showing performance differences between programming paradigms. A secondary contribution is that this paper documents and further develops the use of the Token Accuracy Map (TAM) technique as a tool to provide more detailed data about the exact nature of the problem areas so that teaching techniques can be refined.

A.1 Methods

To evaluate our research questions on student performance and the usefulness of Token Accuracy Mapping, we conducted a randomized controlled trial with a repeated measures design. We compared the students' performance on three successive programming tasks using one of two paradigms. Students were randomly assigned to either the threads or process-oriented programming group by the testing application using Stratified Randomization in a manner recommended by the CONSORT 2010 Statement for transparent reporting

of trials [Gro09]. The pre-task demographic survey determined their self-reported level of education (year in school) and then randomly assigned them to one of the two groups, using an algorithm to keep the groups balanced at each level by randomly selecting the paradigm within each educational group. This randomization process was designed to ensure a roughly equal number of participants at each grade level in each group. Once assigned to a group, the testing application presented the same tasks to both groups, along with reference code samples in the same paradigm as their task. The timing and instructions were held constant in both groups so that we could directly observe the differences between the paradigms and grade levels.

We tracked two dependent variables in this experiment. The first was a straightforward time on task, measured in seconds. Since the participant submission was not compiled or run during the experiment, the time measurement period was from the time the task was presented until the student submitted their answer. The second dependent variable is the accuracy score for a participant on a task as measured by a token accuracy mapping algorithm [SS13, Dal16]. 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 Process), 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 a university in the western United States. The tasks were numbered from 1 to 3 and were the same for each group. We also tracked other demographic information that was not used for classification, including age, gender, native language, and self-reported programming and job experience. Although self-reporting of programming experience is imperfect, it is a reliable approach as documented by Siegmund et al. [SKL+14]

We made the study design decision to control for teaching method variations between groups by standardizing the training materials provided to each group. In order to teach students the concepts and syntax required to complete the tasks in this experiment, the participants were provided with correct code samples in the paradigm for their group. These code samples, along with a description of what the code did, provided as standardized a learning experience as possible for our purpose. Each group received the same number of code samples and we did not provide any customized instruction for either group. We attempted to give code samples that would illustrate the techniques required to solve the three problems we tested, but there could have been bias unintentionally introduced on particular tasks for either group. It was beyond the scope of this experiment to test for variations in teaching and although our method was likely not the optimal teaching method, it was designed to be as consistent between groups as possible.

A.1.1 Materials

In an attempt to accurately and specifically measure the impact of the programming paradigm instead of comparative language syntax, we translated both paradigms into a neutral language, Quorum. Quorum was designed to be a straightforward, evidence-based language with minimal syntax for natural speaking through

Task Description
Using the code sample given to you, write a program that starts two concurrent things, named **F** and **G**. The thing **F**, should show on the screen the word 'hello'. The thing **G** should show on the screen the word 'world'. After both **F** and **G** have finished show the word 'Done'. The whole program should start in a thing named **Main**.

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

```
hello
world
Done
```

or

```
world
hello
Done
```

Figure A.1: Task 1 Description.

screen reading programs for accessibility [quo18]. We used a Java-style syntax for threads as a model to create a hypothetical Quorum implementation. Similarly, we used an `occam`-style syntax for the process-based paradigm. Hypothetical Quorum code samples were then created to be as similar as possible to each other. Minor syntactic differences were required to implement the different programming paradigms (for example, the inclusion of the keyword `concurrent` in the Process group and `synchronized` in the Threads group), but otherwise the Quorum portion of the language was the same in both groups.

The code samples for both groups are included in the Appendix. The samples were given to the participants and were available when they completed the tasks. The task descriptions and instructions were also identical for each group for each task.

Task 1: Two Concurrent Objects

The description for Task 1, shown in Figure A.1, asks the participant to write a program to launch two concurrent objects which each print a statement. This task is similar to the code sample provided and it was intended to be a warm-up task to give the participant practice in setting up the code to execute the two tasks concurrently in the paradigm. The use of the non-technical term "things" in all of the Task Descriptions was deliberate for methodological reasons, specifically, to allow the Task Descriptions to be identical between the two groups and therefore not bias or inadvertently provide an advantage to either group. Since a "thing" could be either a process method or a thread depending on the group, we used a generic non-technical term instead. To the extent that the non-technical term may have adversely impacted student performance in the study, the effect should have been the same for each group since it was a neutral term. The solutions to this task are shown in Figure A.2 and Figure A.3.

```
class Main
  action F
    output "hello"
  end

  action G
    output "world"
  end

  action Main
    concurrent
      F()
      G()
    end
    output "Done"
  end
end
```

Figure A.2: Task 1 Solution (Process).

```
class F is Thread
  action Run
    output "hello"
  end
end

class G is Thread
  action Run
    output "world"
  end
end

class Main
  action Main
    F t1
    G t2
    t1:Run()
    t2:Run()
    check
      t1:Join()
      t2:Join()
    detect e
    end
    output "Done"
  end
end
```

Figure A.3: Task 1 Solution (Threads).

Task Description
Using the code sample given to you, write a program that has two things named **Producer**, one thing named **Consumer**, and one thing named **Main**.

The producer generates integers in ascending order, starting at zero, forever. The consumer reads values from the producers forever, showing the values on the screen with the words 'Received producer [1 or 2]: ' and then the value. The consumer may not skip any values generated by either producer. The thing main starts the producers and the consumer.

Task Sample Output
The final stuff shown on the screen will be two sets of numbers increasing forever. The consumer could consume a value from either producer at any time. For example:

```
Received producer 2:  1
Received producer 1:  1
Received producer 1:  2
Received producer 2:  2
Received producer 2:  3
Received producer 2:  4
...
```

Figure A.4: Task 2 Description.

Task 2: Producer-Consumer

The task description shown in Figure A.4 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 Process 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. The solutions to this task are shown in Figure A.5 and Figure A.6.

Task 3: Readers-Writers

The third task is asking the participants to implement a variation of the Readers-Writers problem with 3 threads where multiple concurrent objects/processes access and write to a shared memory location. The sample provides the structure required to implement the task described in Figure A.7, but does so with only two threads. The solutions to this task are shown in Figure A.8 and Figure A.9.

A.1.2 Procedure

In order to administer this experiment, we created a web-based testing application to run in a standard browser. It administers the experiment and measures and tracks timing of events and snapshots of the

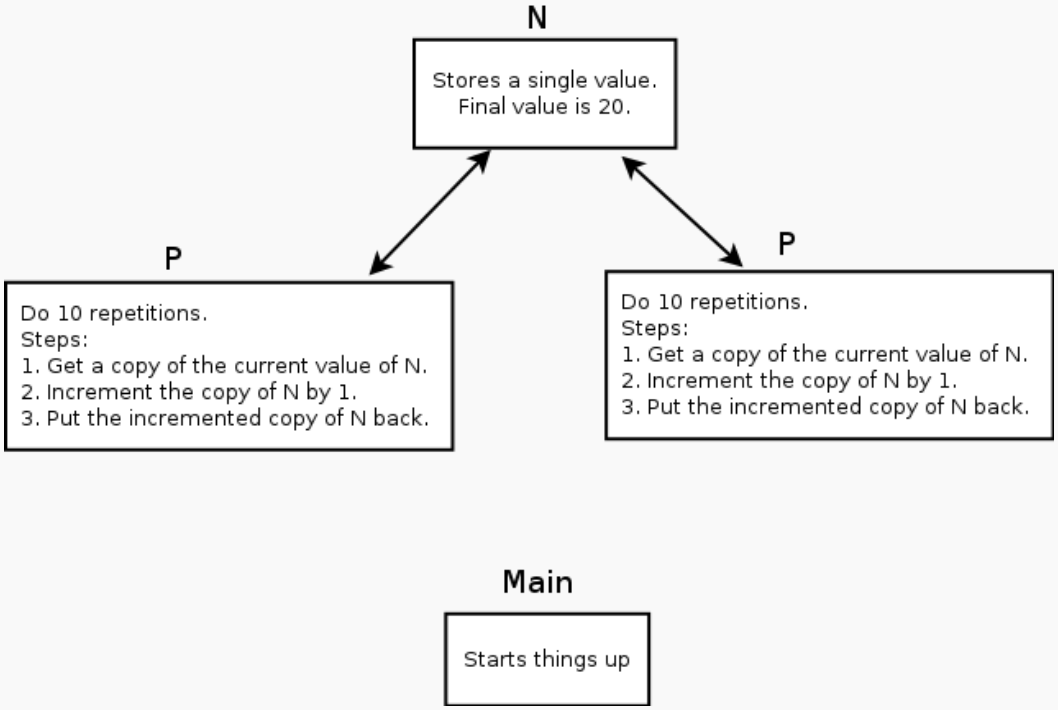
```
class Main
  action Producer(Writer<integer> c)
    integer i = 0
    repeat while true
      c:Write(i)
      i = i + 1
    end
  end

  action Consumer(Reader<integer> p1, Reader<integer> p2)
    repeat while true
      integer x = 0
      choose
        x = p1:Read()
          output "Received Producer 1: " + x
      or
        x = p2:Read()
          output "Received Producer 2: " + x
      end
    end
  end

  action main
    Channel<integer> c1
    Channel<integer> c2
    concurrent
      Producer(c1:GetWriter())
      Producer(c2:GetWriter())
      Consumer(c1:GetReader(), c2:GetReader())
    end
  end
end
```

Figure A.5: Task 2 Solution (Process)

```
class N
  integer n = 0
end

class Producer is Thread
  N n = undefined
  action Set(N n)
    me:n = n
  end
  action Run
    integer i = 0
    repeat while true
      synchronized(N)
        if n:n = 0
          n:n = i
          i = i + 1
        end
      end
    end
  end
end

class Consumer is Thread
  N n1 = undefined
  N n2 = undefined
  action Set (N n1, N n2)
    me:n1 = n1
    me:n2 = n2
  end
  action Run
    repeat while true
      synchronized(N)
        if n1:n not= 0
          output "Received p1: " + n1:n
          n1:n = 0
        elseif n2:n not= 0
          output "Received p2: " + n2:n
          n2:n = 0
        end
      end
    end
  end
end

class Main
  action Main
    N n1
    N n2
    Producer p1
    Producer p2
    Consumer c
    p1:Set(n1)
    p2:Set(n2)
    c:Set(n1, n2)
    p1:Run()
    p2:Run()
    c:Run()
    check
      p1:Join()
      p2:Join()
      c:Join()
    detect e
    end
  end
end
```

Figure A.6: Task 2 Solution (Threads)

Task Description
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 A.7: Task 3 Description.

```
class Main
  action P(Reader<integer> in, Writer<integer> out, Writer<boolean> req)
    integer i=0
    repeat while i<10
      req:Write(true)
      integer myN = in:Read()
      out:Write(myN + 1)
      i = i + 1
    end
  end

  action N(Writer<integer> out1, Reader<integer> in1, Writer<integer>
   out2, Reader<integer> in2, Reader<boolean> req1, Reader<boolean> req2
   )
    integer n=0
    repeat while n < 20
      boolean x = false
      choose
        x = req1:Read()
          out1:Write(n)
          n = in1:Read()
      or
        x = req2:Read()
          out2:Write(n)
          n = in2:Read()
      end
    end
    output n
  end

  action Main
    Channel<integer> c1a
    Channel<integer> c1b
    Channel<integer> c2a
    Channel<integer> c2b
    Channel<boolean> r1
    Channel<boolean> r2
    concurrent
      N(c1a:GetWriter(), c2a:GetReader(), c1b:GetWriter(), c2b:GetReader
   (), r1:GetReader(), r2:GetReader())
      P(c1a:GetReader(), c2a:GetWriter(), r1:GetWriter())
      P(c1b:GetReader(), c2b:GetWriter(), r2:GetWriter())
    end
  end
end
```

Figure A.8: Task 3 Solution (Process).

```
class N
  integer value = 0
end

class P is Thread
  N n = undefined
  integer i = 0
  action Set(N n)
    me:n = n
  end
  action Run
    repeat while i < 10
      synchronized(n)
        integer temp = 0
        temp = n:value
        temp = temp + 1
        n:value = temp
      end
      i = i + 1
    end
  end
end

class Main
  action main
    N n
    P p1
    P p2
    p1:Set(n)
    p2:Set(n)
    p1:Run()
    p2:Run()
    check
      p1:Join()
      p2:Join()
    detect e
    end
    output "Final value of N: " + n:value
  end
end
```

Figure A.9: Task 3 Solution (Threads).

code entry area while the participants are programming. Participants did not have any knowledge of what language was being used, consistent with the design intention of having the study be as language syntax neutral as possible.

A.2 Code Samples

A.2.1 Process Group

Code Sample 1 - Process Group

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

```
class Main
  action F
    output 1
    output 2
  end

  action G
    output 3
    output 4
  end

  action Main
    concurrent
      F()
      G()
    end
    output "Done"
  end
end
```

The code will display any of the following statements:

Case1	Case2	Case3	Case4	Case5	Case6
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

Code Sample 2 - Process Group

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

```
class Main
  action F(Writer<integer> c, integer x)
    c:Write(x)
    c:Write(x)
  end

  action G(Reader<integer> c1, Reader<integer> c2)
    integer i = 0
    repeat while i < 4
      integer x = 0
      choose
        x = c1:Read()
          output x
      or
        x = c2:Read()
          output x
        end
        i = i + 1
      end
  end

  action Main
    Channel<integer> c1
    Channel<integer> c2
    concurrent
      F(c1:GetWriter(), 1)
      F(c2:GetWriter(), 2)
      G(c1:GetReader(), c2:GetReader())
    end
    output "Done"
  end
end
```

The code will display any of the following statements:

Case1	Case2	Case3	Case4	Case5	Case6
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

Code Sample 3 - Process Group

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

```
class Main
  action A(Reader<integer> request, Writer<integer> reply)
    repeat while true
      integer x = request:Read()
      reply:Write(x * 2)
    end
  end

  action B(Writer<integer> request, Reader<integer> reply)
    integer i = 0
    repeat while true
      request:Write(i)
      integer y = reply:Read()
      output y
      i = i + 1
    end
  end

  action Main
    Channel<integer> c1
    Channel<integer> c2
    concurrent
      A(c2:GetReader(), c1:GetWriter())
      B(c2:GetWriter(), c1:GetReader())
    end
  end
end
```

The code will display a list of even numbers starting at 0 forever

```
Output
    0
    2
    4
    6
    ...
```

A.2.2 Threads Group

Code Sample 1 - Threads Group

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

```
class F is Thread
  action Run
    output 1
    output 2
  end
end

class G is Thread
  action Run
    output 3
    output 4
  end
end

class Main
  action main
    F f
    G g
    f:Run()
    g:Run()
    check
      f:Join()
      g:Join()
    detect e
    end
    output "Done"
  end
end
```

The code will display any of the following statements:

Case1	Case2	Case3	Case4	Case5	Case6
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

Code Sample 2 - Threads Group

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

```
class N
  integer value = 0
end

class F is Thread
  N n = undefined
  integer x = 0
  action Set(N n, integer x)
    me:n = n
    me:x = x
  end

  action Run()
    i = 0
    repeat while i < 2
      synchronized(N)
        if n:value = 0
          n:value = x
          i = i + 1
        end
      end
    end
  end
end

class G is Thread
  N n1 = undefined
  N n2 = undefined
  action Set (N n1, N n2)
    me:n1 = n1
    me:n2 = n2
  end

  action Run()
    i = 0
    repeat while i < 4
      synchronized(N)
        if n1:value not= 0
          output n1:value
          n1:value = 0
          i = i + 1
        elseif n2:value not= 0
          output n2:value
          n2:value = 0
          i = i + 1
        end
      end
    end
  end
end

public class Main
  action Main
    F f1
    F f2
    G g
    N n1
    N n2

    f1:Set(n1, 1)
    f2:Set(n2, 2)
```

```
    g:Set(n1, n2)
    f1:Run()
    f2:Run()
    g:Run()
    check
      f1:Join()
      f2:Join()
      g:Join()
    detect e
    end
    output "Done"
  end
end
```

The code will display any of the following statements:

Case1	Case2	Case3	Case4	Case5	Case6
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

Code Sample 3 - Threads Group

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

```
class N
  integer value = 0
  boolean newItem = false
end

class A is Thread
  N n = undefined

  action Set(N n)
    me:n = n
  end

  action Run()
    repeat while true
      synchronized(N)
        if n:newItem = true
          n:value = n:value * 2
          n:newItem = false
        end
      end
    end
  end
end

class B is Thread
  N n = undefined

  action Set(N n)
    me:n = n
  end

  action Run()
    integer i = 1
    repeat while true
      synchronized(N)
        if n:newItem = false
          integer y = n:value
          output y
          n:value = i
          i = i + 1
          n:newItem = true
        end
      end
    end
  end
end

class Main
  action Main()
    N n
    A a
    B b
    a:Set(n)
    b:Set(n)
    a:Run()
    b:Run()
    check
      a:Join()
      b:Join()
    detect e
    end
  end
end
```

The code will display a list of even numbers starting at 0 forever

```
Output
  0
  2
  4
  6
  ...
```

Appendix B

Materials For RCT on GPU Programming from Chapter 5

B.1 Group 1 - CUDA

Group 1 - Reference Sheet given to the CUDA group. The following PDF file shows the instructions with formatting as presented to user through the electronic testing system.

(Remainder of page intentionally left blank.)

Group 1 – Reference Sheet

Overview:

You are programming in an environment where you have a host computer with an attached co-processor capable of higher performance than the host. Since you are working on an application where performance is a key design specification, it is advantageous for you to run certain methods on the co-processor to maximize performance.

The following language reference provides valid error-free C++ code and contains all of the information you will need in order to complete the tasks. Not all language instructions are required for every task.

Language Reference:

Variable Types

The C++ variable types used in these examples are (they are referred to as '*variabletype'* in these instructions):

```
int         //integer type
float       //float type
int*        //integer pointer
float*      //float pointer
```

Kernel Method

A kernel method contains the code for a method that is run on the co-processor. To specify a kernel to run on the co-processor on your system use the following format (this example has a method named '*kernelName'* that returns type void and has 3 parameters: an 'int' and two pointers of type 'int'.):

```
__global__
void kernelName (int n, int *a, int *b) {
        //body of method code goes here
}
```

Memory Allocation

Host:

To allocate memory on the host for the vector, use the following format. This code allocates the memory space on the host for a vector named 'myVector' which contains 'num' items of type '*variabletype'*:

variabletype *myVector = (*variabletype* *) malloc(num*sizeof(*variabletype*));

Co-processor:

In order for the kernel to run, the data must be present in the memory of the co-processor, which can be done using the following format (this code allocates the memory space on the co-processor for an integer vector named 'myVector' which contains 'num' items of type '*variabletype'*):

variabletype *myVector = NULL;
cudaMalloc((void **) &myVector, num*sizeof(*variabletype*));

Memory Deallocation

Host:

To free memory on the host used by a vector, use the following format:

delete [] *variableName*;

Co-processor:

To free memory on the co-processor used by a vector, use the following format:

Group 1 – Reference Sheet

cudaFree(*variableName*);

Iterating loop

An iterating loop traverses all the elements in the vector one at a time. For this system, use an iterating loop in the following format:

```
for (int i = 0; i < n; i++) {
        // body of loop code goes here
}
```

Copy Data To/From Host/Co-processor

To move data from the host to the co-processor, use the following format:

cudaMemcpy(fromVectorName, toVectorName, num*sizeof(*variabletype*), cudaMemcpyHostToDevice);

To move data from the co-processor to the host, use the following format:

cudaMemcpy(fromVectorName, toVectorName, num*sizeof(*variabletype*), cudaMemcpyDeviceToHost);

Launch a kernel on the device

To launch the kernel method on the co-processor use the following format (the values 1,1 inside the <> are configuration settings whose meanings are not relevant to this study, but are necessary as written):

kernelName<<<1,1>>>(parameters);

When a kernel is launched on the co-processor, the host program will continue. In order to pause execution on the host until the kernel on the co-processor finishes use the following format:

cudaSynchronize();

B.2 Group 2 - Thrust

Group 2 - Reference Sheet given to the Thrust group. The following PDF file shows the instructions with formatting as presented to user through the electronic testing system.

(Remainder of page intentionally left blank.)

Group 2 – Reference Sheet

Overview:

You are programming in an environment where you have a host computer with an attached co-processor capable of higher performance than the host. Since you are working on an application where performance is a key design specification, it is advantageous for you to run certain methods on the co-processor to maximize performance.

The following language reference provides valid error-free C++ code and contains all of the information you will need in order to complete the tasks. Not all language instructions are required for every task.

Language Reference:

Variable Types

The C++ variable types used in these examples are (they are referred to as '*variabletype'* in these instructions):

```
int         //integer type
float       //float type
int*        //integer pointer
float*      //float pointer
```

Kernel Method

A kernel method contains the code for a method that is run on the co-processor. To specify a kernel to run on the co-processor on your system use the following format (this example has a method named '*kernelName'* that returns type void and has 3 parameters: an 'int' and two pointers of type 'int'.):

```
__global__
void kernelName (int n, int *a, int *b) {
        //body of method code goes here
}
```

Memory Allocation

Host:

To allocate memory on the host for the vector, use the following format. This code allocates the memory space on the host for a vector named 'myVector' which contains 'num' items of type '*variabletype'*:

```
thrust::host_vector<<variabletype>> myVector(num);
```

Co-processor:

In order for the kernel to run, the data must be present in the memory of the co-processor, which can be done using the following format (this code allocates the memory space on the co-processor for an integer vector named 'myVector' which contains 'num' items of type '*variabletype'*):

```
thrust::device_vector<<variabletype>> myVector(num);
```

Memory Deallocation

Host:

Memory used by vectors deallocate automatically when the vector goes out of scope so it is unnecessary to do it manually unless the memory is needed immediately. To deallocate a vector immediately, use the following format:

```
myVector.clear();
myVector.shrink_to_fit();
```

Co-processor:

Group 2 – Reference Sheet

Same as host dellocation.

Iterating loop

An iterating loop traverses all the elements in the vector one at a time. For this system, use an iterating loop in the following format:

```
//This code fills a vector named myVector with a constant value:
thrust::fill(myVector.begin(), myVector.end(), 3);


//This code transforms every element in an vector X by adding each element in vector Y to each element in vector Z.
All the vectors are of type  'variableType'
thrust::transform(X.begin(), X.end(), Y.begin(), Z.begin(), thrust::plus<variableType>());
```

Copy Data To/From Host/Co-processor

Assuming myHostVector is a thrust::host_vector and myDevVector is a thrust::device_vector.

To move data from the host to the co-processor, use the following format:

```
myDevVector = myHostVector;
```

To move data from the co-processor to the host, use the following format:

```
myHostVector = myDevVector;
```

Execute the add operation on the co-processor

The add operation to add Y = Y + X requires an iterating loop using thrust::transform

To launch a custom kernel use the following format:

Kernelname <<<number of blocks, number of threads>>>(kernel parameters)

Get a C++ point from a vector:

*variableType *variableName* = thrust::raw_pointer_cast(&*vectorName*[0]);

Appendix C

Token Signatures of Common Errors

C.1 Description

This section contains tables of token signatures and error counts for the 10 most common token signatures based on the number of actual compiler errors for each one.

C.2 Error Code 43: PARSER_NO_VIABLE_ALTERNATIVE

Rank	Error Code	Token Signature	Token Map	N	%
1	43	65	ID	4,854	14.7%
2	43	65 66	ID STRING	2,891	8.8%
3	43			2,457	7.5%
4	43	66	STRING	1,669	5.1%
5	43	1	output	762	2.3%
6	43	63	INTEGER_LITERAL	519	1.6%
7	43	1 65	output ID	463	1.4%
8	43	1 66	output STRING	462	1.4%
9	43	65 65 65	ID ID ID	440	1.3%
10	43	1 58 65	output ID	372	1.1%
11	43	OMITTED	OMITTED FOR LENGTH	299	0.9%
12	43	1 65 65	output ID ID	273	0.8%
13	43	60	end	263	0.8%
14	43	1 44 65	output = ID	262	0.8%
15	43	39 42 65 42 65 42 65	use . ID . ID . ID	231	0.7%
16	43	39 65 42 65 42 65	use ID . ID . ID	217	0.7%
17	43	1 44 66	output = STRING	201	0.6%
18	43	37 65 44 19 66	text ID = input STRING	200	0.6%
19	43	35 65 44 63	integer ID = INTEGER_LITERAL	182	0.6%
20	43	20 66	say STRING	180	0.5%
21	43	56 65 49 65 57 51 56 65 50 65 57 49 56 65 52 65 57	(ID + ID) * (ID - ID) + (ID / ID)	171	0.5%
22	43	37 65 44 66	text ID = STRING	154	0.5%
23	43	19 66	input STRING	148	0.4%
24	43	63 65	INTEGER_LITERAL ID	144	0.4%
25	43	65 49 65	ID + ID	132	0.4%
26	43	1 66 65	output STRING ID	117	0.4%
27	43	65 63 43 65 63 34 65 65 65 65 19	ID INTEGER_LITERAL , ID INTEGER_LITERAL : ID ID ID ID input	114	0.3%
28	43	65 63	ID INTEGER_LITERAL	112	0.3%
29	43	56 63 49 63 57 51 56 63 50 63 57 49 56 63 52 63 57	(INTEGER_LITERAL + INTEGER_LITERAL) * (INTEGER_LITERAL - INTEGER_LITERAL) + (INTEGER_LITERAL / INTEGER_LITERAL)	112	0.3%
30	43	65 65 65 65 65	ID ID ID ID ID	106	0.3%
31	43	65 66 49 65	ID STRING + ID	89	0.3%
32	43	20 58 65 65	say ID ID	88	0.3%
33	43	65 34 65	ID : ID	85	0.3%
34	43	65 63 43 65 63	ID INTEGER_LITERAL , ID INTEGER_LITERAL	85	0.3%
35	43	1 37 65	output text ID	81	0.2%
36	43	20 58 65	say ID	79	0.2%
37	43	65 45 65	ID >ID	77	0.2%
38	43	65 63 24	ID INTEGER_LITERAL times	77	0.2%
39	43	37 65 44 66 49 37 65 44 66	text ID = STRING + text ID = STRING	74	0.2%
40	43	65 56 57	ID ( )	70	0.2%

C.3 Error Code 35: OTHER

Rank	Error Code	Token Signature	Token Map	N	%
1	35	60	end	1,296	6.4%
2	35			1,259	6.3%
3	35	39 65 42 65 42 65	use ID . ID . ID	647	3.2%
4	35	37 65 66	text ID STRING	485	2.4%
5	35	35 63 65 44 63	integer INTEGER_LITERAL ID = INTEGER_LITERAL	459	2.3%
6	35	1 66	output STRING	361	1.8%
7	35	26	else	331	1.6%
8	35	66	STRING	324	1.6%
9	35	33 65	action ID	294	1.5%
10	35	1 65 66	output ID STRING	279	1.4%
11	35	65 65 42 65 42 65	ID ID . ID . ID	248	1.2%
12	35	56 65 49 65 57 51 56 65 50 65 57 49 56 65 52 65 57	(ID + ID) * (ID - ID) + (ID / ID)	203	1.0%
13	35	1 65	output ID	203	1.0%
14	35	5 65 66	elseif ID STRING	199	1.0%
15	35	37 44 19 56 66 57	text = input (STRING)	197	1.0%
16	35	1 65 63	output ID INTEGER_LITERAL	140	0.7%
17	35	35 44 63	integer = INTEGER_LITERAL	129	0.6%
18	35	61 65 17 65	class ID is ID	121	0.6%
19	35	37 36 44 19 56 66 57	text number = input (STRING)	116	0.6%
20	35	66 49 66	STRING + STRING	110	0.5%
21	35	37 65 44 66 66	text ID = STRING STRING	102	0.5%
22	35	65 44 65 44 63	ID = ID = INTEGER_LITERAL	97	0.5%
23	35	65 42 65 42 65	ID . ID . ID	93	0.5%
24	35	65 65 49 65	ID ID + ID	92	0.5%
25	35	65 65 34 65 56 63 43 63 57	ID ID : ID (INTEGER_LITERAL , INTEGER_LITERAL)	92	0.5%
26	35	65 34 65 56 66 57	ID : ID (STRING)	91	0.5%
27	35	37 65 19 56 66 57	text ID input (STRING)	89	0.4%
28	35	65 42 65	ID . ID	89	0.4%
29	35	37 65 63 44 66	text ID INTEGER_LITERAL = STRING	89	0.4%
30	35	37 65 49 19 56 66 57	text ID + input (STRING)	89	0.4%
31	35	39 65 42 65 42 65 42 65	use ID . ID . ID . ID	88	0.4%
32	35	38 44 65 45 65	boolean = ID >ID	87	0.4%
33	35	59 65 63 47 63	if ID INTEGER_LITERAL <INTEGER_LITERAL	87	0.4%
34	35	38 65 44 62 40 44 62	boolean ID = BOOLEAN_LITERAL not = BOOLEAN_LITERAL	85	0.4%
35	35	65 65 66	ID ID STRING	82	0.4%
36	35	37 65 56 66 57	text ID (STRING)	81	0.4%
37	35	65 42 63 65 44 64	ID . INTEGER_LITERAL ID = DECIMAL_LITERAL	80	0.4%
38	35	37 65 50 63 44 19 56 66 57	text ID - INTEGER_LITERAL = input (STRING)	77	0.4%
39	35	37 44 66	text = STRING	74	0.4%
40	35	37 65 44 66	text ID = STRING	73	0.4%

C.4 Error Code 0: MISSING_VARIABLE

Rank	Error Code	Token Signature	Token Map	N	%
1	0	1 65	output ID	2,877	23.3%
2	0	59 65	if ID	1,287	10.4%
3	0	35 65 44 65 52 65	integer ID = ID / ID	1,039	8.4%
4	0	36 65 44 18 56 36 43 65 57	number ID = cast (number , ID)	417	3.4%
5	0	65 44 65	ID = ID	366	3.0%
6	0	65 34 65 56 66 57	ID : ID (STRING)	350	2.8%
7	0	35 65 44 18 56 35 43 65 57	integer ID = cast (integer , ID)	286	2.3%
8	0	1 66 49 65	output STRING + ID	232	1.9%
9	0	36 65 44 65 53 65	number ID = ID mod ID	192	1.6%
10	0	59 65 44 66	if ID = STRING	181	1.5%
11	0	38 65 44 65	boolean ID = ID	140	1.1%
12	0	36 65 44 65 52 65	number ID = ID / ID	137	1.1%
13	0	1 65 49 65	output ID + ID	136	1.1%
14	0	37 65 44 65	text ID = ID	134	1.1%
15	0	20 65	say ID	124	1.0%
16	0	35 65 44 65	integer ID = ID	112	0.9%
17	0	37 65 44 66 49 65 49 66 49 65 49 66 49 65 49 66	text ID = STRING + ID + STRING + ID + STRING + ID + STRING	111	0.9%
18	0	1 65 51 63 49 65	output ID * INTEGER_LITERAL + ID	110	0.9%
19	0	1 66 49 65 49 66	output STRING + ID + STRING	109	0.9%
20	0	65 44 65 29 65	ID = ID and ID	101	0.8%
21	0	36 65 44 65	number ID = ID	95	0.8%
22	0	65 34 65 56 65 57	ID : ID (ID)	94	0.8%
23	0	65 34 65 56 57	ID : ID ( )	94	0.8%
24	0	59 65 44 63	if ID = INTEGER_LITERAL	83	0.7%
25	0	25 7 65 45 63	repeat until ID >INTEGER_LITERAL	69	0.6%
26	0	35 65 44 65 51 63	integer ID = ID * INTEGER_LITERAL	68	0.6%
27	0	36 65 44 65 49 65 49 65 52 65	number ID = ID + ID + ID / ID	65	0.5%
28	0	59 65 48 63	if ID <= INTEGER_LITERAL	65	0.5%
29	0	65 34 65 56 63 43 63 57	ID : ID (INTEGER_LITERAL , INTEGER_LITERAL)	64	0.5%
30	0	59 65 45 65	if ID >ID	63	0.5%
31	0	38 65 44 65 44 65	boolean ID = ID = ID	61	0.5%
32	0	1 65 44 66	output ID = STRING	61	0.5%
33	0	1 65 49 65 49 65 49 65 49 65	output ID + ID + ID + ID + ID	60	0.5%
34	0	28 65	return ID	60	0.5%
35	0	65 56 65 57	ID (ID)	59	0.5%
36	0	59 65 44 65	if ID = ID	55	0.4%
37	0	65 44 65 49 56 65 52 63 57	ID = ID + (ID / INTEGER_LITERAL)	55	0.4%
38	0	35 65 44 65 34 65 56 63 43 63 57	integer ID = ID : ID (INTEGER_LITERAL , INTEGER_LITERAL)	50	0.4%
39	0	36 65 44 65 51 65	number ID = ID * ID	49	0.4%
40	0	25 7 65 46 63	repeat until ID >= INTEGER_LITERAL	46	0.4%

C.5 Error Code 11: MISSING_USE

Rank	Error Code	Token Signature	Token Map	N	%
1	11	65 65	ID ID	2,328	28.3%
2	11	65 65 44 63	ID ID = INTEGER_LITERAL	1,837	22.4%
3	11	39 65 42 65 42 65	use ID . ID . ID	750	9.1%
4	11	65 65 65 65	ID ID ID ID	260	3.2%
5	11	65 65 44 66	ID ID = STRING	244	3.0%
6	11	65 65 44 63 49 63	ID ID = INTEGER_LITERAL + INTEGER_LITERAL	201	2.4%
7	11	65	ID	183	2.2%
8	11	65 65 44 19 56 66 57	ID ID = input (STRING)	180	2.2%
9	11	65 65 44 18 56 65 43 65 57	ID ID = cast (ID , ID)	154	1.9%
10	11	65 65 44 62	ID ID = BOOLEAN_LITERAL	112	1.4%
11	11	65 65 44 18 56 35 43 19 56 66 57 57	ID ID = cast (integer , input (STRING))	87	1.1%
12	11	35 65 44 18 56 65 43 65 57	integer ID = cast (ID , ID)	83	1.0%
13	11	65 65 44 65 45 65	ID ID = ID >ID	81	1.0%
14	11	65 65 65 65 65 65	ID ID ID ID ID ID	71	0.9%
15	11	36 65 44 18 56 65 43 65 57	number ID = cast (ID , ID)	60	0.7%
16	11	65 65 44 65 51 63	ID ID = ID * INTEGER_LITERAL	57	0.7%
17	11	1 65 65 65	output ID ID ID	50	0.6%
18	11	65 65 44 64	ID ID = DECIMAL_LITERAL	49	0.6%
19	11	65 65 44 18 56 35 43 65 57	ID ID = cast (integer , ID)	43	0.5%
20	11	61 65 17 65	class ID is ID	43	0.5%
21	11	65 65 44 63 50 63	ID ID = INTEGER_LITERAL - INTEGER_LITERAL	42	0.5%
22	11	65 65 44 65	ID ID = ID	35	0.4%
23	11	35 65 44 18 56 65 43 19 56 66 57 57	integer ID = cast (ID , input (STRING))	34	0.4%
24	11	37 65 44 65 65 65	text ID = ID ID ID	34	0.4%
25	11	65 65 44 63 52 63	ID ID = INTEGER_LITERAL / INTEGER_LITERAL	32	0.4%
26	11	65 65 44 56 66 57	ID ID = (STRING)	32	0.4%
27	11	39 65 42 65 42 65 42 65	use ID . ID . ID . ID	28	0.3%
28	11	65 65 44 65 49 65	ID ID = ID + ID	24	0.3%
29	11	65 65 65 65 65 65 65 65	ID ID ID ID ID ID ID ID	24	0.3%
30	11	26 65 65 44 63	else ID ID = INTEGER_LITERAL	22	0.3%
31	11	33 65 56 35 65 43 65 65 57	action ID (integer ID , ID ID)	21	0.3%
32	11	65 65 44 63 48 63	ID ID = INTEGER_LITERAL <= INTEGER_LITERAL	21	0.3%
33	11	65 65 44 62 30 62	ID ID = BOOLEAN_LITERAL or BOOLEAN_LITERAL	21	0.3%
34	11	65 65 65	ID ID ID	21	0.3%
35	11	65 44 65 65 65	ID = ID ID ID	21	0.3%
36	11	38 65 65 65 44 62	boolean ID ID ID = BOOLEAN_LITERAL	21	0.3%
37	11	65 65 44 18 56 36 43 65 57	ID ID = cast (number , ID)	20	0.2%
38	11	65 65 44 63 51 63	ID ID = INTEGER_LITERAL * INTEGER_LITERAL	19	0.2%
39	11	65 65 44 62 29 62	ID ID = BOOLEAN_LITERAL and BOOLEAN_LITERAL	19	0.2%
40	11	65 65 44 65 52 65	ID ID = ID / ID	18	0.2%

C.6 Error Code 41: INPUT_MISMATCH

Rank	Error Code	Token Signature	Token Map	N	%
1	41			511	6.5%
2	41	39 65 42 65 42 65	use ID . ID . ID	476	6.0%
3	41	66	STRING	450	5.7%
4	41	60	end	401	5.1%
5	41	1 65	output ID	333	4.2%
6	41	1 66	output STRING	298	3.8%
7	41	59 65 44 66	if ID = STRING	112	1.4%
8	41	37 65 44 19 56 65 6 65 65 42 42 42 58 57	text ID = input (ID me ID ID . . .)	111	1.4%
9	41	35 65 44 63 34 63 34 63	integer ID = INTEGER_LITERAL : INTEGER_LITERAL : INTEGER_LITERAL	73	0.9%
10	41	37 65 44 66	text ID = STRING	70	0.9%
11	41	19 66	input STRING	67	0.9%
12	41	59 65 45 65	if ID >ID	67	0.9%
13	41	37 66	text STRING	65	0.8%
14	41	65 65	ID ID	63	0.8%
15	41	61 65	class ID	63	0.8%
16	41	1 66 49 65 49 66	output STRING + ID + STRING	62	0.8%
17	41	36 65 44 18 56 36 43 65 57	number ID = cast (number , ID)	62	0.8%
18	41	25 56 65 52 65 49 56 65 50 63 57 24	repeat (ID / ID + (ID - INTEGER_LITERAL) times	58	0.7%
19	41	19 56 66 57	input (STRING)	52	0.7%
20	41	1 66 49 65	output STRING + ID	51	0.6%
21	41	37 65 44 19 56 66 57	text ID = input (STRING)	46	0.6%
22	41	65 65 65 65 65 65 65	ID ID ID ID ID ID ID	46	0.6%
23	41	5 65 44 66	elseif ID = STRING	46	0.6%
24	41	33 65	action ID	45	0.6%
25	41	35 65 44 18 56 35 43 19 56 65 65 65 34 58 57 57	integer ID = cast (integer , input (ID ID ID :))	43	0.5%
26	41	25 65 65 63	repeat ID ID INTEGER_LITERAL	43	0.5%
27	41	35 65 44 65 51 63	integer ID = ID * INTEGER_LITERAL	40	0.5%
28	41	1 66 49 65 49 66 42 58	output STRING + ID + STRING .	40	0.5%
29	41	63 43 63 51 63	INTEGER_LITERAL , INTEGER_LITERAL * INTEGER_LITERAL	39	0.5%
30	41	37 65 44 19 56 65 65 65 65 65 65 58 57	text ID = input (ID ID ID ID ID ID)	34	0.4%
31	41	37 65 44 19 56 66 57 56 66 57	text ID = input (STRING) (STRING)	33	0.4%
32	41	52 51 33 65 65 65	/ * action ID ID ID	33	0.4%
33	41	51 52 33 65 65 65	* / action ID ID ID	33	0.4%
34	41	35 65 44 63 43 63 43 63	integer ID = INTEGER_LITERAL , INTEGER_LITERAL , INTEGER_LITERAL	32	0.4%
35	41	1 66 65 42 58	output STRING ID .	32	0.4%

C.7 Error Code 42: LEXER_NO_VIABLE_ALTERNATIVE

Rank	Error Code	Token Signature	Token Map	N	%
1	42	1 65	output ID	1,726	22.3%
2	42	65 65	ID ID	944	12.2%
3	42	35 65 44 63	integer ID = INTEGER_LITERAL	347	4.5%
4	42	1 65 65	output ID ID	324	4.2%
5	42	65 65 65	ID ID ID	270	3.5%
6	42	20 65 65	say ID ID	165	2.1%
7	42	65	ID	164	2.1%
8	42	20 65	say ID	126	1.6%
9	42	37 65 44 19 56 65 65 65 65 65 65 65 65 65 57	text ID = input (ID ID ID ID ID ID ID ID ID)	119	1.5%
10	42	1 66	output STRING	108	1.4%
11	42	OMITTED	OMITTED FOR LENGTH	99	1.3%
12	42	37 65 44 65 65	text ID = ID ID	81	1.0%
13	42			77	1.0%
14	42	36 65 44 65 65 51 65	number ID = ID ID * ID	75	1.0%
15	42	37 65 44 65 43 65	text ID = ID , ID	75	1.0%
16	42	37 65 44 19 56 66 57	text ID = input (STRING)	69	0.9%
17	42	65 65 65 65 65 65 65	ID ID ID ID ID ID ID	65	0.8%
18	42	65 56 63 43 63 43 63 43 63 57	ID (INTEGER_LITERAL , INTEGER_LITERAL , INTEGER_LITERAL , INTEGER_LITERAL)	63	0.8%
19	42	OMITTED	OMITTED FOR LENGTH	61	0.8%
20	42	65 65 65 65	ID ID ID ID	60	0.8%
21	42	65 34 65 56 65 52 65 42 65 57	ID : ID (ID / ID . ID)	51	0.7%
22	42	37 65 44 65 49 65	text ID = ID + ID	47	0.6%
23	42	1 65 65 65	output ID ID ID	45	0.6%
24	42	37 65 44 64	text ID = DECIMAL_LITERAL	45	0.6%
25	42	37 65 44 34	text ID = :	42	0.5%
26	42	60	end	38	0.5%
27	42	1 65 65 65 65 65 65 17 66	output ID ID ID ID ID ID is STRING	37	0.5%
28	42	65 65 65 65 65	ID ID ID ID ID	33	0.4%
29	42	1 65 65 65 65 65 65 17 66 65 65 42	output ID ID ID ID ID ID is STRING ID ID .	33	0.4%
30	42	65 56 65 57	ID (ID)	30	0.4%
31	42	65 56 57	ID ( )	26	0.3%
32	42	39 65 42 65 42 65	use ID . ID . ID	25	0.3%
33	42	1	output	23	0.3%
34	42	1 66 65 58	output STRING ID	23	0.3%
35	42	37 65 44 65 65 65	text ID = ID ID ID	22	0.3%
36	42	37 65 44 19 56 65 65 65 65 65 65 65 39 58 57	text ID = input (ID ID ID ID ID ID use)	22	0.3%
37	42	20 65 43 65 43 65 43 65	say ID , ID , ID , ID	21	0.3%
38	42	65 65 44 29	ID ID = and	21	0.3%
39	42	65 56 63 43 63 43 63 57	ID (INTEGER_LITERAL , INTEGER_LITERAL , INTEGER_LITERAL)	19	0.2%

C.8 Error Code 14: DUPLICATE

Rank	Error Code	Token Signature	Token Map	N	%
1	14	65 65	ID ID	1,697	27.5%
2	14	35 65 44 63	integer ID = INTEGER_LITERAL	1,213	19.7%
3	14	36 65 44 18 56 36 43 66 57	number ID = cast (number , STRING)	748	12.1%
4	14	37 65 44 66	text ID = STRING	509	8.3%
5	14	37 65 44 66 49 63	text ID = STRING + INTEGER_LITERAL	196	3.2%
6	14	37 65 44 19 56 66 57	text ID = input (STRING)	175	2.8%
7	14	65 65 44 63	ID ID = INTEGER_LITERAL	146	2.4%
8	14	35 65 44 18 56 35 43 65 57	integer ID = cast (integer , ID)	119	1.9%
9	14	38 65 44 62	boolean ID = BOOLEAN_LITERAL	110	1.8%
10	14	35 65 44 18 56 35 43 19 56 66 57 57	integer ID = cast (integer , input (STRING))	98	1.6%
11	14	36 65 44 18 56 36 43 65 57	number ID = cast (number , ID)	69	1.1%
12	14	35 65 44 65 51 63	integer ID = ID * INTEGER_LITERAL	66	1.1%
13	14	65 65 44 66	ID ID = STRING	49	0.8%
14	14	1 65 65 65	output ID ID ID	48	0.8%
15	14	35 65	integer ID	43	0.7%
16	14	59 63 65 65 1 56 66 57	if INTEGER_LITERAL ID ID output (STRING)	43	0.7%
17	14	36 65 44 18 56 36 43 19 56 66 57 57	number ID = cast (number , input (STRING))	29	0.5%
18	14	33 65	action ID	28	0.5%
19	14	36 65	number ID	27	0.4%
20	14	36 65 44 65 51 63	number ID = ID * INTEGER_LITERAL	27	0.4%
21	14	36 65 65 44 63	number ID ID = INTEGER_LITERAL	27	0.4%
22	14	1 65 65 65 65 65	output ID ID ID ID ID	25	0.4%
23	14	36 65 44 56 65 49 65 52 63 49 65 52 63 57 51 63	number ID = (ID + ID / INTEGER_LITERAL + ID / INTEGER_LITERAL) * INTEGER_LITERAL	24	0.4%
24	14	36 65 44 64	number ID = DECIMAL_LITERAL	24	0.4%
25	14	65	ID	24	0.4%
26	14	35 65 44 66	integer ID = STRING	20	0.3%
27	14	36 65 44 65 34 65 56 57	number ID = ID : ID ( )	16	0.3%
28	14	37 65 65 44 66 49 65 49 66 49 65 49 66 49 65 49 66	text ID ID = STRING + ID + STRING + ID + STRING + ID + STRING	15	0.2%
29	14	65 65 65 65	ID ID ID ID	15	0.2%
30	14	36 65 44 63	number ID = INTEGER_LITERAL	14	0.2%
31	14	65 65 44 63 65 65	ID ID = INTEGER_LITERAL ID ID	14	0.2%
32	14	65 44 63	ID = INTEGER_LITERAL	13	0.2%
33	14	35 65 44 56 66 57	integer ID = (STRING)	10	0.2%
34	14	36 65 44 65 50 65	number ID = ID - ID	10	0.2%
35	14	35 65 44 65 49 65	integer ID = ID + ID	8	0.1%
36	14	37 65 44 64	text ID = DECIMAL_LITERAL	8	0.1%
37	14	33 65 56 36 65 57	action ID (number ID)	8	0.1%
38	14	36 65 44 65 52 65	number ID = ID / ID	7	0.1%
39	14	65 34 65 56 63 43 63 57 65 65	ID : ID (INTEGER_LITERAL , INTEGER_LITERAL) ID ID	7	0.1%

C.9 Error Code 3: MISSING_METHOD

Rank	Error Code	Token Signature	Token Map	N	%
1	3	65 56 57	ID ( )	1,504	37.6%
2	3	37 65 44 65 56 66 57	text ID = ID (STRING)	241	6.0%
3	3	65 34 65 56 66 57	ID : ID (STRING)	106	2.6%
4	3	65 56 63 57	ID (INTEGER_LITERAL)	97	2.4%
5	3	1 65 34 65 56 57	output ID : ID ( )	94	2.3%
6	3	65 56 63 43 63 57	ID (INTEGER_LITERAL , INTEGER_LITERAL)	92	2.3%
7	3	36 65 44 65 56 63 50 65 57	number ID = ID (INTEGER_LITERAL - ID)	89	2.2%
8	3	65 56 66 57	ID (STRING)	87	2.2%
9	3	65 34 65 56 57	ID : ID ( )	75	1.9%
10	3	35 65 44 65 56 57	integer ID = ID ( )	70	1.7%
11	3	1 65 34 65 56 63 43 63 57	output ID : ID (INTEGER_LITERAL , INTEGER_LITERAL)	68	1.7%
12	3	65 34 65 56 64 57	ID : ID (DECIMAL_LITERAL)	63	1.6%
13	3	35 65 44 65 56 63 43 63 57	integer ID = ID (INTEGER_LITERAL , INTEGER_LITERAL)	62	1.5%
14	3	65 56 65 57	ID (ID)	60	1.5%
15	3	65 34 65 56 63 57	ID : ID (INTEGER_LITERAL)	56	1.4%
16	3	36 65 44 65 34 65 56 63 43 63 57	number ID = ID : ID (INTEGER_LITERAL , INTEGER_LITERAL)	53	1.3%
17	3	65 34 65 56 66 43 66 43 66 43 66 43 66 43 66 43 66 57	ID : ID (STRING , STRING , STRING , STRING , STRING , STRING , STRING)	52	1.3%
18	3	65 34 65 56 63 43 63 57	ID : ID (INTEGER_LITERAL , INTEGER_LITERAL)	41	1.0%
19	3	1 65 49 56 66 57 49 65 56 66 57 49 65	output ID + (STRING) + ID (STRING) + ID	41	1.0%
20	3	65 34 65 56 65 57	ID : ID (ID)	36	0.9%
21	3	1 65	output ID	35	0.9%
22	3	1 65 56 66 57	output ID (STRING)	34	0.8%
23	3	36 65 44 65 34 65 56 57	number ID = ID : ID ( )	32	0.8%
24	3	1 65 34 65 56 63 57	output ID : ID (INTEGER_LITERAL)	31	0.8%
25	3	35 65 44 18 56 35 43 65 56 66 57 57	integer ID = cast (integer , ID (STRING))	31	0.8%
26	3	35 65 44 65 34 65 56 63 43 63 57	integer ID = ID : ID (INTEGER_LITERAL , INTEGER_LITERAL)	30	0.7%
27	3	65 44 65 34 65 56 63 57	ID = ID : ID (INTEGER_LITERAL)	27	0.7%
28	3	65 56 63 43 63 43 63 43 63 57	ID (INTEGER_LITERAL , INTEGER_LITERAL , INTEGER_LITERAL , INTEGER_LITERAL)	26	0.6%
29	3	65 34 65 56 63 43 65 57	ID : ID (INTEGER_LITERAL , ID)	22	0.5%
30	3	1 65 34 65 56 57 49 66 49 65 34 65 56 57 49 66 49 65 34 65 56 57	output ID : ID ( ) + STRING + ID : ID ( ) + STRING + ID : ID ( )	22	0.5%
31	3	65 34 65 56 63 43 63 43 63 57	ID : ID (INTEGER_LITERAL , INTEGER_LITERAL , INTEGER_LITERAL)	21	0.5%
32	3	65 56 63 43 63 43 63 57	ID (INTEGER_LITERAL , INTEGER_LITERAL , INTEGER_LITERAL)	21	0.5%
33	3	1 65 56 63 57	output ID (INTEGER_LITERAL)	20	0.5%

C.10 Error Code 12: INVALID_OPERATOR

Rank	Error Code	Token Signature	Token Map	N	%
1	12	65 44 65 49 64	ID = ID + DECIMAL_LITERAL	611	18.4%
2	12	35 65 44 64	integer ID = DECIMAL_LITERAL	325	9.8%
3	12	65 65 44 66	ID ID = STRING	195	5.9%
4	12	65 65 44 63	ID ID = INTEGER_LITERAL	186	5.6%
5	12	35 65 44 66	integer ID = STRING	164	4.9%
6	12	35 65 44 19 56 66 57	integer ID = input (STRING)	137	4.1%
7	12	37 65 44 64	text ID = DECIMAL_LITERAL	117	3.5%
8	12	35 65 44 65 52 65	integer ID = ID / ID	100	3.0%
9	12	35 65 44 65 56 63 43 63 57	integer ID = ID (INTEGER_LITERAL , INTEGER_LITERAL)	65	2.0%
10	12	35 65 44 63 44 18 56 35 43 65 57	integer ID = INTEGER_LITERAL = cast (integer , ID)	62	1.9%
11	12	38 65 44 63	boolean ID = INTEGER_LITERAL	59	1.8%
12	12	65 65 44 19 56 66 57	ID ID = input (STRING)	58	1.7%
13	12	65 65 44 62	ID ID = BOOLEAN_LITERAL	51	1.5%
14	12	37 65 44 63	text ID = INTEGER_LITERAL	49	1.5%
15	12	35 65 44 56 66 57	integer ID = (STRING)	47	1.4%
16	12	35 65 44 65 49 65	integer ID = ID + ID	42	1.3%
17	12	65 65 44 18 56 35 43 65 57	ID ID = cast (integer , ID)	41	1.2%
18	12	35 65 44 18 56 36 43 19 56 66 57 57	integer ID = cast (number , input (STRING))	37	1.1%
19	12	36 65 44 19 56 66 57	number ID = input (STRING)	36	1.1%
20	12	35 65 44 65 49 65 49 65	integer ID = ID + ID + ID	33	1.0%
21	12	35 65 44 63 52 64	integer ID = INTEGER_LITERAL / DECIMAL_LITERAL	23	0.7%
22	12	65 65 44 65 34 65 56 63 43 63 57	ID ID = ID : ID (INTEGER_LITERAL , INTEGER_LITERAL)	22	0.7%
23	12	35 65 44 18 56 36 43 65 57	integer ID = cast (number , ID)	21	0.6%
24	12	35 65 44 63 44 63	integer ID = INTEGER_LITERAL = INTEGER_LITERAL	21	0.6%
25	12	35 65 44 66 49 66	integer ID = STRING + STRING	20	0.6%
26	12	65 44 63	ID = INTEGER_LITERAL	20	0.6%
27	12	65 44 65	ID = ID	19	0.6%
28	12	37 65 44 18 56 35 43 65 57	text ID = cast (integer , ID)	19	0.6%
29	12	35 65 44 65 56 57	integer ID = ID ( )	16	0.5%
30	12	35 65 44 65 49 65 49 65 49 65	integer ID = ID + ID + ID + ID	15	0.5%
31	12	65 65 44 65 45 65	ID ID = ID >ID	15	0.5%
32	12	38 65 44 65	boolean ID = ID	15	0.5%
33	12	35 65 44 65	integer ID = ID	15	0.5%
34	12	38 65 44 19 56 66 57	boolean ID = input (STRING)	15	0.5%
35	12	65 44 65 44 56 65 52 63 57	ID = ID = (ID / INTEGER_LITERAL)	15	0.5%
36	12	37 65 44 66 44 66	text ID = STRING = STRING	14	0.4%
37	12	36 65 44 65 49 65 49 66	number ID = ID + ID + STRING	14	0.4%
38	12	65 44 65 34 65 56 63 57	ID = ID : ID (INTEGER_LITERAL)	13	0.4%

C.11 Error Code 5: INCOMPATIBLE TYPES

Rank	Error Code	Token Signature	Token Map	N	%
1	5	59 65 44 63 47 63	if ID = INTEGER_LITERAL <INTEGER_LITERAL	140	9.2%
2	5	38 65 44 65 45 65	boolean ID = ID >ID	111	7.3%
3	5	1 65	output ID	91	6.0%
4	5	37 65 44 66 50 66	text ID = STRING - STRING	61	4.0%
5	5	59 65 44 63	if ID = INTEGER_LITERAL	50	3.3%
6	5	59 65 47 63	if ID <INTEGER_LITERAL	42	2.8%
7	5	59 65 48 63	if ID <= INTEGER_LITERAL	42	2.8%
8	5	65 44 62 49 62 29 62 49 62	ID = BOOLEAN_LITERAL + BOOLEAN_LITERAL and BOOLEAN_LITERAL + BOOLEAN_LITERAL	34	2.2%
9	5	36 65 44 65 52 65	number ID = ID / ID	31	2.0%
10	5	59 65 44 66	if ID = STRING	28	1.8%
11	5	1 65 34 65 56 63 57	output ID : ID (INTEGER_LITERAL)	27	1.8%
12	5	59 65 44 66 30 66	if ID = STRING or STRING	24	1.6%
13	5	1 66 44 65	output STRING = ID	24	1.6%
14	5	64 47 65 47 64	DECIMAL_LITERAL <ID <DECIMAL_LITERAL	24	1.6%
15	5	1 65 51 65	output ID * ID	23	1.5%
16	5	59 65 45 63	if ID >INTEGER_LITERAL	23	1.5%
17	5	5 65 48 63	elseif ID <= INTEGER_LITERAL	22	1.4%
18	5	1 65 44 66	output ID = STRING	20	1.3%
19	5	59 65 47 65	if ID <ID	20	1.3%
20	5	38 65 44 65 47 65	boolean ID = ID <ID	18	1.2%
21	5	5 65 44 63	elseif ID = INTEGER_LITERAL	18	1.2%
22	5	59 65 45 65	if ID >ID	17	1.1%
23	5	65 44 65 49 65	ID = ID + ID	16	1.1%
24	5	1 65 34 65 56 57 49 66 44 65 34 65 56 57 44 66 49	output ID : ID ( ) + STRING = ID : ID ( ) = STRING +	16	1.1%
25	5	1 65 51 63	output ID * INTEGER_LITERAL	15	1.0%
26	5	36 65 44 65 51 63	number ID = ID * INTEGER_LITERAL	15	1.0%
27	5	38 65 44 63 29 63	boolean ID = INTEGER_LITERAL and INTEGER_LITERAL	13	0.9%
28	5	37 65 44 66 29 66	text ID = STRING and STRING	13	0.9%
29	5	38 65 44 62 49 62	boolean ID = BOOLEAN_LITERAL + BOOLEAN_LITERAL	12	0.8%
30	5	36 65 44 65 51 65	number ID = ID * ID	12	0.8%
31	5	35 65 44 65 44 63 49 65 44 65	integer ID = ID = INTEGER_LITERAL + ID = ID	12	0.8%
32	5	59 65 45 63 30 65 47 63	if ID >INTEGER_LITERAL or ID <INTEGER_LITERAL	11	0.7%
33	5	28 65	return ID	10	0.7%
34	5	1 65 50 65	output ID - ID	9	0.6%
35	5	35 65 44 66 50 66	integer ID = STRING - STRING	9	0.6%
36	5	1 65 29 65	output ID and ID	9	0.6%
37	5	28 65 44 65	return ID = ID	9	0.6%
38	5	65 44 66 50 66	ID = STRING - STRING	8	0.5%
39	5	35 65 44 65 45 65	integer ID = ID >ID	8	0.5%

Bibliography

[AB15] Amjad Altadmri and Neil C.C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 522–527, New York, NY, USA, 2015. ACM.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ali17] alice.org. Alice. http://alice.org, 2017.

[AMD17] Inc. Advanced Micro Devices. *High Performance Computing*, 2017 (accessed 27-April-2017).

[BA14] Neil C.C. Brown and Amjad Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 43–50, New York, NY, USA, 2014. ACM.

[BA16] Neil CC Brown and Amjad Altadmri. Novice java programming mistakes: Large-scale data vs. educator beliefs. *ACM Transactions on Computing Education (TOCE)*, 2016.

[BA17] Neil CC Brown and Amjad Altadmri. Novice java programming mistakes: large-scale data vs. educator beliefs. *ACM Transactions on Computing Education (TOCE)*, 17(2):7, 2017.

[BCL17] Mathias Bourgoin, Emmanuel Chailloux, and Jean-Luc Lamotte. High level data structures for gpgpu programming in a statically typed language. *International Journal of Parallel Programming*, 45(2):242–261, 2017.

[BDP+19] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, pages 177–210. ACM New York, NY, USA, 2019.

[Bec15] Brett A Becker. An exploration of the effects of enhanced compiler error messages for computer programming novices. *none*, 2015.

[Bec16] Brett A. Becker. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 126–131. ACM, 2016.

[BKMU14] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 223–228. ACM, 2014.

[BM84] Benedict du Boulay and Ian Matthew. Fatal error in pass zero: How not to confuse novices. *Behaviour & Information Technology*, 3(2):109–118, 1984.

[BMT+18] Brett A Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney, and Kate Sanders. Fix the first, ignore the rest: Dealing with multiple compiler error messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 634–639, 2018.

[Boa17] OpenMP Architecture Review Board. *OpenMP*, 2017 (accessed 24-April-2017).

[CA 17] CA Technologies. Enterprise data security: The basics of user behavior analytics. https://www.ca.com/content/dam/ca/us/files/white-paper/enterprise-data-security-the-basics-of-user-behavior-analytics.pdf, 2017.

[Cao08] Longbing Cao. Behavior informatics and analytics: Let behavior talk. In *Data Mining Workshops, 2008. ICDMW'08. IEEE International Conference on*, pages 87–96. IEEE, 2008.

[Car15] Elizabeth Carter. Its debug: practical results. *Journal of Computing Sciences in Colleges*, 30(3):9–15, 2015.

[CB13] Elizabeth Carter and Glenn D Blank. A tutoring system for debugging: status report. *Journal of Computing Sciences in Colleges*, 28(3):46–52, 2013.

[CB14] Elizabeth Carter and Glenn D Blank. Debugging tutor: preliminary evaluation. *Journal of Computing Sciences in Colleges*, 29(3):58–64, 2014.

[cod17] code.org. Promote computer science. https://code.org/promote, 2017 (accessed 27 November 2017).

[Cor17a] Intel Corporation. *Intel Cilk Plus*, 2017 (accessed 24-April-2017).

[Cor17b] Intel Corporation. *Intel Threading Building Blocks*, 2017 (accessed 24-April-2017).

[Cor17c] Intel Corporation. *Intel Xeon Phi Coprocessors*, 2017 (accessed 27-April-2017).

[Cor17d] NVIDIA Corporation. *GPU vs CPU? What is GPU Computing*, 2017 (accessed 27-April-2017).

[Cor17e] NVIDIA Corporation. *CUDA Toolkit Documentation - v8.0.61*, 2017 (updated March 20, 2017).

[COS11] Fernando Castor, Joao Paulo Oliveira, and Andre LM Santos. 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*, pages 117–122, 2011.

[CSM+15] Michael Coblenz, Robert Seacord, Brad Myers, Joshua Sunshine, and Jonathan Aldrich. A course-based usability analysis of cilk plus and openmp. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*, pages 245–249. IEEE, 2015.

[Dal16] Patrick Daleiden. Empirical study of concurrency programming paradigms. Master's thesis, University of Nevada, Las Vegas, Las Vegas, NV, 2016.

[Dav17] Alan B Davidson. The commerce departments digital economy agenda. https://www.commerce.gov/news/blog/2015/11/commerce-departments-digital-economy-agenda, 2015 (accessed 27 November 2017).

[DBC+19] Paul Denny, Brett A Becker, Michelle Craig, Greg Wilson, and Piotr Banaszkiewicz. Research this! questions that computing educators most want computing education researchers to an-

swer. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 259–267, 2019.

[DH50] Richard Doll and A Bradford Hill. Smoking and carcinoma of the lung. *British medical journal*, 2(4682):739, 1950.

[DLRC14] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 273–278. ACM, 2014.

[DLRT12] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 75–80. ACM, 2012.

[DLRTH11a] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Codewrite: supporting student-driven practice of java. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 471–476. ACM, 2011.

[DLRTH11b] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 208–212. ACM, 2011.

[DR10] Thomas Dy and Ma Mercedes Rodrigo. A detector for non-literal java errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 118–122. ACM, 2010.

[DSUP20] P. Daleiden, A. Stefik, P. M. Uesbeck, and J. Pedersen. Analysis of a randomized controlled trial of student performance in parallel programming using a new measurement technique. *ACM Transactions on Computing Education (TOCE)*, In Press, 2020.

[erl17] erlang.org. Erlang programming language. https://www.erlang.org/, 2017.

[ES93] Karl Anders Ericsson and Herbert Alexander Simon. *Protocol analysis*. MIT press Cambridge, MA, 1993.

[FCJ04] Thomas Flowers, Curtis A Carver, and James Jackson. Empowering students and building confidence in novice programmers through gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T3H–10. IEEE, 2004.

[FR96] Stephen N Freund and Eric S Roberts. Thetis: An ansi c programming environment designed for introductory use. In *SIGCSE*, volume 96, pages 300–304, 1996.

[Gab16] Evghenii Gaburov. *Quick Start Guide*, 2016 (posted 16-March-2016).

[gol17a] golang.org. The go programming languages: Faq: Design. https://golang.org/doc/faq#csp, 2017.

[gol17b] golang.org. The go programming languages: Faq: Origins. https://golang.org/doc/faq#origins, 2017.

[Goo17] Google and Gallup. Searching for computer science: Access and barriers in u.s. k-12 education., 2015 (accessed 27 November 2017).

[Gro09] The CONSORT Group. Consort 2010 statement: updated guidelines for reporting parallel group randomised trials. section 8b. randomisation: type. http://www.consort-statement.org/checklists/view/32–consort-2010/87-randomisation-type, 9 Dec 2009.

[Gro17] Khronos Group. *OpenCL: The open standard for parallel programming of heterogeneous systems*, 2017 (accessed 24-April-2017).

[Har17] Mark Harris. *An Even Easier Introduction to CUDA*, 2017 (posted 25-January-2017).

[HB15] Jared Hoberock and Nathan Bell. Thrust–parallel algorithms library. *Available: thrust. github. io*, 2015.

[HMBK10] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1019–1028. ACM, 2010.

[HMRM03] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, volume 35, pages 153–156. ACM, 2003.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[IBM13] IBM Corporation. Oh behave! how behavioral analytics fuels more personalized marketing. http://hosteddocs.ittoolbox.com/ohbehave.pdf, 2013.

[Jad05] Matthew C Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40, 2005.

[JCC05] James Jackson, Michael Cobb, and Curtis Carver. Identifying top java errors for novice programmers. In *Proceedings frontiers in education 35th annual conference*, pages T4C–T4C. IEEE, 2005.

[JS85] W Lewis Johnson and Elliot Soloway. Proust: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, 3:267–275, 1985.

[jso20] json.org. Introducing json, 2020.

[k1217] k12cs.org. K–12 computer science framework. http://www.k12cs.org, 2016 (accessed 27 November 2017).

[Kai15] Antti-Juhani Kaijanaho. *Evidence-Based Programming Language Design: A Philosophical and Methodological Exploration*. PhD thesis, University of Jyvaskyla, 2015. Information Technology Faculty.

[KES+09] Volodymyr V Kindratenko, Jeremy J Enos, Guochun Shi, Michael T Showerman, Galen W Arnold, John E Stone, James C Phillips, and Wen-mei Hwu. Gpu clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.

[KK03] Sarah K Kummerfeld and Judy Kay. The neglected battle fields of syntax errors. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20*, pages 105–111. Australian Computer Society, Inc., 2003.

[KLGM16] Mary Beth Kery, Claire Le Goues, and Brad A Myers. Examining programmer practices for locally handling exceptions. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 484–487. IEEE, 2016.

[KPBB+09] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering - a systematic literature review. *Inf. Softw. Technol.*, 51(1):7–15, Jan 2009.

[KQPR03] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[LBM+07] Gary Lewandowski, Dennis J Bouvier, Robert McCartney, Kate Sanders, and Beth Simon. Commonsense computing (episode 3): concurrency and concert tickets. In *Proceedings of the third international workshop on Computing education research*, pages 133–144. ACM, 2007.

[Mar00] Harry M Marks. *The progress of experiment: science and therapeutic reform in the United States, 1900-1990*. Cambridge University Press, 2000.

[MFL+08] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, 2008.

[Mic20] Microsoft. Intellisense, visual studio code, https://code.visualstudio.com/docs/editor/intellisense, 2020.

[MLM+08] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *ACM SIGCSE Bulletin*, volume 40, pages 163–167. ACM, 2008.

[MSA+01] David Moher, Kenneth F Schulz, Douglas G Altman, Consort Group, et al. The consort statement: revised recommendations for improving the quality of reports of parallel-group randomised trials, 2001.

[MSB+14] Lauri Malmi, Judy Sheard, Roman Bednarik, Juha Helminen, Päivi Kinnunen, Ari Korhonen, Niko Myller, Juha Sorva, Ahmad Taherkhani, et al. Theoretical underpinnings of computing education research: what is the evidence? In *Proceedings of the tenth annual conference on International computing education research*, pages 27–34. ACM, 2014.

[MWB99] Gail C. Murphy, Robert J. Walker, and ELA Banlassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on software engineering*, 25(4):438–455, 1999.

[New05] Mark EJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[NPM08] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: What can help novices? In *ACM SIGCSE Bulletin*, volume 40, pages 168–172. ACM, 2008.

[NTPM11] Sebastian Nanz, Faraz Torshizi, Michela Pedroni, and Bertrand Meyer. Empirical assessment of languages for teaching concurrency: Methodology and application. In *Software Engineering Education and Training (CSEE&T), 2011 24th IEEE-CS Conference on*, pages 477–481. IEEE, 2011.

[NW70] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[Oba17] Obama White House. Fact sheet: President obama announces computer science for all initiative. https://obamawhitehouse.archives.gov/the-press-office/2016/01/30/fact-sheet-president-obama-announces-computer-science-all-initiative-0, 2016 (accessed 31 October 2017).

[Ora17] Oracle Corporation. Secure coding guidelines for java se. http://www.oracle.com/technetwork/java/seccodeguide-139067.html, 2017 (accessed 28 September 2017).

[PAT11] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 43–52. ACM, 2011.

[PBDP+14] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. ACM, 2014.

[PBL+16] Thomas W Price, Neil CC Brown, Dragan Lipovac, Tiffany Barnes, and Michael Kölling. Evaluation of a frame-based programming editor. In *ICER*, pages 33–42, 2016.

[PF11] Terence Parr and Kathleen Fisher. Ll(*): The foundation of the antlr parser generator. *SIGPLAN Not.*, 46(6):425–436, June 2011.

[PHG17] Raymond S Pettit, John Homer, and Roger Gee. Do enhanced compiler error messages help students?: Results inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 465–470. ACM, 2017.

[PPM+17] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 74–82. ACM, 2017.

[Pri15] David Pritchard. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 1–8, 2015.

[pu17] pop users.org. occam-pi in a nutshell. http://pop-users.org/occam-pi/, 2017.

[PZS+17] Alan Peterfreund, Jennifer Dounay Zinth, Jim Stanton, Katie A. Hendrickson, Lynn Goldsmith, Pat Yongpradit, Rebecca Zarch, Sarah Dunton, and W. Richards Adrion. State of the states landscape report: State-level policies supporting equitable k–12 computer science education. https://www.ecs.org/state-of-the-states-landscape-report-state-level-policies-supporting-equitable-k-12-computer-science-education/, 2017 (accessed 27 November 2017).

[quo18] quorumlanguage.com. Quorum: Programming languages and learning. https://quorumlanguage.com/evidence.html, 2018.

[RHW10a] Christopher J Rossbach, Owen S Hofmann, and Emmett Witchel. Is transactional programming actually easier? *ACM Sigplan Notices*, 45(5):47–56, 2010.

[RHW10b] Christopher J Rossbach, Owen S Hofmann, and Emmett Witchel. Is transactional programming actually easier? *ACM Sigplan Notices*, 45(5):47–56, 2010.

[RT05] Peter C Rigby and Suzanne Thompson. Study of novice programmers using eclipse and gild. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 105–109. ACM, 2005.

[RWZ10] Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE software*, 27(4):80–86, 2010.

[Sch95] Tom Schorsch. Cap: an automated self-assessment tool to check pascal programs for syntax, logic and style errors. In *ACM SIGCSE Bulletin*, volume 27, pages 168–172. ACM, 1995.

[Sch17] Steven E. Schoenherr. The digital revolution. https://web.archive.org/web/20081007132355/http://history.sanc 2014 (accessed 27 November 2017).

[scr17] scratch.mit.edu. Scratch. https://scratch.mit.edu, 2017.

[SDM+03] Margaret-Anne Storey, Daniela Damian, Jeff Michaud, Del Myers, Marcellus Mindel, Daniel German, Mary Sanseverino, and Elizabeth Hargreaves. Improving the usability of eclipse for novice programmers. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 35–39. ACM, 2003.

[SGA+16] Jim Stanton, Lynn Goldsmith, Richards Adrion, Sarah Dunton, Katie A. Hendrickson, Alan Peterfreund, Pat Yongpradit, Rebecca Zarch, and Jennifer Dounay Zinth. Bridging the computer science education gap: Five actions states can take, November 2016.

[SKL+14] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[SPBK88] D Sleeman, Ralph T Putnam, Juliet Baxter, and Laiani Kuspa. An introductory pascal class: A case study of students' errors. *Teaching and Learning Computer Programming: Multiple Research Perspectives. RE Mayer. Hillsdale, NJ, Lawrence Erlbaum Asociates*, pages 237–257, 1988.

[SS13] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.

[SSE+14] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, pages 724–734. ACM, 2014.

[SSSS11] Andreas Stefik, Susanna Siebert, Melissa Stefik, and Kim Slattery. An empirical comparison of the accuracy rates of novices using the quorum, perl, and randomo programming languages. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 3–8. ACM, 2011.

[SY15] Duane Storti and Mete Yurtoglu. *CUDA for Engineers: An Introduction to High-performance Parallel Computing*. Addison-Wesley Professional, 2015.

[TCAL13] Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136*, pages 87–95. Australian Computer Society, Inc., 2013.

[The17] The College Board. Ap computer science principles. https://apstudent.collegeboard.org/apcourse/ap-computer-science-principles, 2017 (accessed 27 November 2017).

[TRB04] Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education*, volume 30, pages 317–325. Australian Computer Society, Inc., 2004.

[TRJ11] Emily S Tabanao, Ma Mercedes T Rodrigo, and Matthew C Jadud. Predicting at-risk novice java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research*, pages 85–92, 2011.

[ULBH08] S.-Z. Ueng, M. Lathara, S.S. Baghsorkhi, and W.-M.W. Hwu. Cuda-lite: Reducing gpu programming complexity. *Lecture Notes in Computer Science (including subseries Lecture Notes*

in Artificial Intelligence and Lecture Notes in Bioinformatics), 5335 LNCS:1–15, 2008. cited By 54.

[Uni17a] University of Chicago. Computer programming languages can impact science and thought. https://news.uchicago.edu/article/2017/09/06/computer-programming-languages-can-impact-science-and-thought, 2017 (accessed 28 September 2017).

[Uni17b] University of Victoria. Gild feature and technical report. http://gild.cs.uvic.ca/docs/summary/gild_feature_and_technical_report.pdf, 2007 (accessed 19 October 2017).

[U.S10] U.S. Department of Education Institute of Education Sciences. *What Works Clearinghouse Procedures and Standards Handbook*. U.S. Department of Education, 2.1 edition, 2010.

[U.S17a] U.S. Bureau of Labor Statistics. May 2016 national occupational employment and wage estimates united states. https://www.bls.gov/oes/current/oes_nat.htm#15-0000, 2016 (accessed 27 November 2017).

[U.S17b] U.S. Department of Commerce. Intellectual property and the u.s. economy: 2016 update, 2016 (accessed 27 November 2017).

[USH+16] P.M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden. An empirical study on the impact of c++ lambdas and programmer experience. In *2016 IEEE/ACM 38th IEEE International Conference on*. IEEE, 2016.

[WAF02] Peter H Welch, Jo R Aldous, and Jon Foster. Csp networking for java (jcsp. net). In *International Conference on Computational Science*, pages 695–708. Springer, 2002.

[WBM+07] Peter H Welch, Neil CC Brown, James Moores, Kevin Chalmers, and Bernhard HC Sputh. Integrating and extending jcsp. *IOS Press, US*, 2007.

[Wei20] Eric W Weisstein. Zipf distribution, https://mathworld.wolfram.com/zipfdistribution.htm, mathworld - a wolfram web resource, 2020.

[Wex76] Richard L Wexelblat. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. In *Proceedings of the 2nd international conference on Software engineering*, pages 331–336. IEEE Computer Society Press, 1976.

[WH17] David Weintrop and Nathan Holbert. From blocks to text and back: Programming patterns in a dual-modality environment. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 633–638, New York, NY, USA, 2017. ACM.

[WK14] Jacqueline Whalley and Nadia Kasto. A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 279–284. ACM, 2014.

[Wor17] Scientific Computing World. *Programming difficulty is killing engineers' productivity*, 2006 (accessed 2-May-2017).

[WW15] David Weintrop and Uri Wilensky. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *ICER*, volume 15, pages 101–110, 2015.

[Yue07] Timothy T Yuen. Novices' knowledge construction of difficult concepts in cs1. *ACM SIGCSE Bulletin*, 39(4):49–53, 2007.

[ZBT11] He Zhang, Muhammad Ali Babar, and Paolo Tell. Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637, 2011.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Patrick Michael Daleiden

pmdaleiden@uchicago.edu

DEGREES::

Master of Science in Computer Science 2016

University of Nevada, Las Vegas, Las Vegas, NV

Master of Business Administration (Finance and Accounting) 1993

University of Chicago, Chicago, IL

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

University of Notre Dame, Notre Dame, IN

PROFESSIONAL CERTIFICATIONS:

Chartered Financial Analyst 1993

CFA Institute

Certified Public Accountant 1995

University of Illinois Board of Examiners

Certificate in Commercial Real Estate 2019

Cornell University, SC Johnson College of Business

FELLOWSHIPS AND SCHOLARSHIPS:

UNLV Foundation Board of Trustees Fellowship 2018-19, 2019-20

Summer Doctoral Research Fellowship 2017

Patricia Sastaunik Scholarship 2016-17, 2017-18, 2019-20

Summer Session Scholarship 2016, 2018

Wolzinger Family Research Scholarship 2015-16, 2017-18

Gilman & Bartlett Engineering Scholarship 2015-16, 2016-17

UNLV Access Grant 2015-16, 2016-17, 2017-18

RESEARCH AND TEACHING:

Graduate Research Certificate 2016

UNLV Graduate College

Doctoral Graduate Research Assistant 2014-15, 2015-16

UNLV Software Engineering and Media Lab, Dr. Andreas Stefik

Graduate Research Assistant (Innovators Developing Accessible Tools in Astronomy) 2016-17, 2017-18

UNLV Software Engineering and Media Lab, Dr. Andreas Stefik

Temporary Faculty - Research 2018

UNLV Software Engineering and Media Lab, Dr. Andreas Stefik

Prospects in Theoretical Physics: Computational Plasma Astrophysics Conference 2016

Institute for Advanced Study, Princeton, NJ

Research Assistant under Letter of Appointment 2016

UNLV Physics and Astronomy Department, Dr. Daniel Proga and Dr. Andreas Stefik

Teaching Assistant - Computer Science II (CS 202) 2016

Dr. Laxmi Gewali, Professor

Teaching Assistant - Software Engineering I and II (CS 472, CS473) 2015-16, 2016-17, 2017-18

Dr. Andreas Stefik, Professor

SERVICE

UNLV Graduate Rebel Ambassador 2015-16, 2016-17, 2017-18, 2018-19

UNLV Graduate and Professional Student Association (GPSA), Student Body Vice President 2016-17

UNLV GPSA, Department Representative, Computer Science 2014-15, 2015-16

UNLV GPSA, Sponsorship Committee, Chairman 2016-17

UNLV GPSA, Sponsorship Committee, Member 2014-15, 2015-16

HONOR SOCIETIES:

Beta Gamma Sigma 1994

University of Chicago Chapter

Phi Kappa Phi 2016

University of Nevada, Las Vegas Chapter

PUBLICATIONS:

Daleiden, P., Stefik, A., and Uesbeck, P.M. "*GPU Programming Productivity In Different Abstraction Paradigms: A randomized controlled trial comparing CUDA and Thrust.*" Manuscript submitted for publication.

Daleiden, P., Uesbeck, P.M. , Stefik, A., and Pedersen, J.. "*Analysis of a Randomized Controlled Trial of Student Performance in Parallel Programming using a New Measurement Technique*" ACM Transactions on Computing Education (TOCE). In press.

Rafalski, T, Uesbeck, P.M., Panks-Meloney, C., Daleiden, P., Allee, W., Mcnamara, A., and Stefik, A. "*A Randomized Controlled Trial on the Wild Wild West of Scientific Computing with Student Learners.*" Proceedings of the 2019 ACM Conference on International Computing Education Research, pp. 239-247. ACM, 2019.

Uesbeck, P. M., Stefik, A., Hanenberg, S., Pedersen, J., and Daleiden, P. "*An Empirical Study on the Impact of C++ Lambdas and Programmer Experience*" Software Engineering (ICSE), 2016 IEEE/ACM 38th IEEE International Conference on. Vol. 1. IEEE, 2016.

Masters Thesis Title: Empirical Study of Concurrent Programming Paradigms

Masters Thesis Examination Committee:

Chairperson, Dr. Andreas Stefik, Ph.D.

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

Committee Member, Dr. Jan "Matt" Pedersen, Ph.D.

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