RICE UNIVERSITY

Cooperative Execution of Parallel Tasks with Synchronization Constraints

by

Shams Mahmood Imam

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Vivek Sarkar

Vivek Sarkar, Chair
Professor of Computer Science
E.D. Butcher Chair in Engineering

John Mellor-Crummey
Professor of Computer Science

Swarat Chaudhuri
Associate Professor of Computer Science

Lin Zhong
Associate Professor of Electrical and
Computer Engineering

Houston, Texas
May, 2015

ABSTRACT

Cooperative Execution of Parallel Tasks with Synchronization Constraints

by

Shams Mahmood Imam

This thesis addresses the effective execution of parallel applications on emerging multicore and manycore systems in the presence of modern synchronization and coordination constraints. Synchronization and coordination can contribute significant productivity and performance overheads to the development and execution of parallel programs. Higher-level programming models, such as the Task Parallel Model and Actor Model, provide abstractions that can be used to simplify writing parallel programs, in contrast to lower-level programming models that directly expose locks, threads and processes. However, these higher-level models often lack efficient support for general synchronization patterns that are necessary for a wide range of applications. Many modern synchronization and coordination constructs in parallel programs can incur significant performance overheads on current runtime systems, or significant productivity overheads when the programmer is forced to complicate their code to mitigate these performance overheads. An alternative to the aforementioned approaches is to have the programmer and the runtime system cooperate to reduce the parallel overhead and to execute the available parallelism efficiently in the presence of synchronization constraints.

With a cooperative approach, an executing entity yields control to other entities at well-defined points during its execution. This thesis shows that the use of

cooperative techniques is critical to performance and scalability of certain parallel programming models, especially in the presence of modern synchronization and coordination constraints such as asynchronous tasks, `future`s, `phaser`s, data-driven tasks, and actors. In particular, we focus on cooperative extensions and runtimes for the `async-finish` Task Parallel Model and the Actor Model (AM) in this thesis. We use a rich task-parallel programming model called Habanero-Java library (HJlib) as an implementation vehicle for our cooperative extensions to runtimes and programming models.

We then present our framework for cooperative scheduling of parallel tasks with general synchronization patterns. We believe that our cooperative runtime can support any task-parallel synchronization construct. Next, we introduce the Eureka Programming Model (EuPM) to simplify the expression and management of speculative parallel tasks, which are especially important for parallel search and optimization applications. The EuPM relies on the use of cooperative termination techniques to save computation time by avoiding redundant exploration of a solution space. Finally, we describe our extension to the AM called *selectors*. Selectors have multiple mailboxes, and each mailbox is guarded i.e. it can be cooperatively enabled or disabled to affect the order in which an actor processes messages. Our work shows that each of these cooperative techniques simplify programmability and deliver significant performance improvements by reducing the overhead in modern parallel programming models.

To my mother, Shamsun

To my father, Mahmood

To my sister, Asma

To my niece, Alayna

To my wife, Mahar

This work is a sign of my love to you!

Acknowledgments

All praises go to the Almighty Allah for giving me the opportunity to pursue every aspect of my PhD. With all humility, I acknowledge the immense grace and blessings which have been bestowed on me.

Foremost, I would like to thank my advisor and mentor Vivek Sarkar for his suggestions and guidance through the process of finishing my PhD. I am eternally in his debt for his support through my difficult times, both academic and personal. Thank you for gently directing me back when I would stray from the course and most importantly, for believing in and having patience with me. I am grateful to him for the freedom I was allowed to pursue and explore new directions and applications of my work. Without his guidance, I would not be where I am today.

I am fortunate to have worked with an excellent thesis committee: Vivek Sarkar, John Mellor-Crummey, Swarat Chaudhuri and Lin Zhong. Thank you all for agreeing to be a part of my thesis committee. I would also like to thank you all for your time, feedback and suggestions for numerous valuable improvements in my thesis. I am lucky to have taken John's course on Parallel Computing, which laid the foundations for my introduction to mainstream parallel programming. I am also thankful to Swarat and Lin for feedback on my work on Actors and their constant encouragement.

I would like to express my gratitude to Robert (Corky) S. Cartwright Jr. His course on principles of programming languages introduced me to the world of functional programming and continuations — a key component of this thesis. His discussions with me on related topics, refreshing enthusiasm and encouragement helped me a lot through my journey. I would like to express my gratitude to my professors Abul L. Haque, Partha Pratim Dey, Arshad Momen, Mohammad Kabir Hossian at

my undergraduate university — North South University. They carry in common a passion for their students, and a desire for their students to succeed. Without their inspiration and reassurance, I could not have even considered graduate school. You set me on the path I am on now and I have never looked back.

In addition to going to more places than I could have hoped for, I have had the privilege of meeting many brilliant people. I've benefited greatly from advice and interactions with extraordinary collaborators at LLNL and Oracle Labs. I thank my mentors Tom Epperly, Adrian Prantl, Christian Wimmer, David Leibs and Peter B. Kessler for their feedback and advice on my research during my Summer internships. I've had the privilege to work with and build my ideas through interaction with many other amazing researchers at Rice. In particular, I would like to thank the members of the Habanero Extreme Scale Software Research Project.

I would like to thank my friends for all the support and encouragement. There are too many of you to mention but I would especially like to thank Adnan, Ali, Alina, Ankush, Bahauddin, Deepak, Dragos, Fouad, Hamim, Kamal, Karthik, Kumud, Max, Mihika, Milind, Mouree, Niketan, Prasanth, Pulok, Raghavan, Raju, Rishi, Ronnie, Rubel, Sagnak, Sanjay, Shahriar, Shashi, Siam, Sriraj, Suguman, Xu, and Zubair. Thanks for always being up for a good laugh over the years. Many others have helped shape my views, research and took the time to comment on sections of this dissertation during its evolution. There is no way I can put down on paper how much you all mean to me, I thank you all.

I thank my family for their many sacrifices that made this work possible. My parents deserve all the love in the world for their love. I thank them for their patience, encouragement, and generosity. Without love and support from my parents and my sister, this would not even have begun. This PhD is a testament to your faith in me, I hope I have made you proud. I share the delight and satisfaction of completing this work with you all.

Contents

Abstract ii

Acknowledgments v

List of Illustrations xi

1 Introduction 1

1.1 Thesis Statement . . . 3

1.2 Contributions . . . 4

1.3 Outline . . . 5

2 Habanero-Java Library: a Java 8 Framework for Multi-core Programming 7

2.1 Contributions . . . 8

2.1.1 Outline . . . 9

2.2 Motivation . . . 10

2.3 Background: Parallel Constructs . . . 12

2.3.1 The Async/Finish Constructs . . . 12

2.3.2 Loop Parallelism . . . 14

2.3.3 Coordination constructs . . . 15

2.4 Implementation . . . 21

2.4.1 `EventDrivenControl` API . . . 23

2.4.2 EventCount Synchronization Construct . . . 25

2.4.3 Handling Blocking Operations . . . 26

2.5 Abstract Execution Metrics . . . 29

2.5.1 Computing Critical Path Length . . . 31
2.6 Deadlock Detection . . . 35
2.7 Summary . . . 39

3 Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns 41

3.1 Contributions . . . 43
3.1.1 Outline . . . 44
3.2 Motivating Examples . . . 45
3.3 Cooperative Runtime for Task Scheduling . . . 49
3.3.1 One-shot Delimited Continuations . . . 50
3.3.2 Event-Driven Controls in the Cooperative Runtime . . . 52
3.3.3 The Cooperative Runtime . . . 53
3.4 Support for Synchronization Constructs . . . 58
3.4.1 Fork-Join Synchronization . . . 58
3.4.2 Producer-Consumer Synchronization . . . 59
3.4.3 Collective Barrier Synchronization . . . 61
3.4.4 `phaser` Synchronization . . . 62
3.4.5 Single Blocks . . . 63
3.4.6 Weak Isolation . . . 63
3.5 Implementation . . . 64
3.5.1 Delimited Continuations - Kilim Weaver . . . 65
3.5.2 Habanero-Java Language Implementation . . . 66
3.5.3 Habanero-Java Library Implementation . . . 68
3.6 Experimental Results . . . 71
3.6.1 Fork/Join Benchmarks . . . 72
3.6.2 `future` Benchmarks . . . 74
3.6.3 `phaser` Benchmarks . . . 76

3.7 Conclusions 78

4 The Eureka Programming Model for Speculative Task Parallelism 80

4.1 Contributions 82
4.2 Motivating Example: Parallel Search of 2D Array 83
4.3 Task Termination Strategies 84
4.3.1 Delimited Continuation-based Cooperative Termination 87
4.4 Programming with `Eurekas` 89
4.4.1 `Eureka` Construct and API 89
4.4.2 Eureka Programming Model (EuPM) 91
4.4.3 Redundant Computation Guarantee 94
4.5 Parallel Patterns and Eureka Variants 95
4.5.1 Parallel Search 95
4.5.2 Count Eureka 96
4.5.3 N-Version Eureka 97
4.5.4 Optimization Eureka 97
4.5.5 Soft Deadlines 98
4.5.6 Convergence Iterations 99
4.6 Reusability and Composability of Eureka Components 101
4.6.1 Composability by Component Composition 102
4.6.2 Reusability by leveraging Functional Decomposition 103
4.7 Implementation 105
4.8 Experimental Results 108
4.8.1 Execution Times Comparison 110
4.8.2 Productivity Metrics Comparison 112
4.9 Summary 113

5 Selectors: Actors with Multiple Guarded Mailboxes 114

5.1 Contributions . . . 116
5.2 Background and Motivation . . . 117
5.2.1 The Actor Model . . . 117
5.3 Selectors: Actors with Guarded Mailboxes . . . 120
5.3.1 Implementation . . . 123
5.4 Synchronous Request-Response Pattern . . . 126
5.5 Join Patterns in Streaming Applications . . . 129
5.6 Supporting Priorities in Message Processing . . . 133
5.7 Reader-Writer Concurrency . . . 134
5.8 Producer-Consumer Pattern . . . 137
5.9 Experimental Results . . . 139
5.9.1 Performance Evaluation . . . 140
5.9.2 Productivity Results . . . 151
5.10 Summary . . . 152

6 Related Work 153

6.1 Habanero-Java Framework . . . 153
6.2 Cooperative Scheduling in the presence of Synchronization Patterns . . . 155
6.3 Speculative Task Parallelism . . . 159
6.4 Actors extensions for synchronization patterns . . . 161

7 Future Work & Conclusions 164

7.1 Future Work . . . 164
7.2 Conclusions . . . 166

Bibliography 168

Illustrations

2.1 HJlib version of a simple `async-finish` program 14
2.2 Syntax and semantics of `forall` and `forasync` 15
2.3 HJlib example with `isolated` statements 16
2.4 HJlib Fib using `futures` 17
2.5 HJlib Fib using DDFs 19
2.6 Diagrammatic explanation of the HJlib runtime 22
2.7 Simplified representation of an `EventDrivenControl` 24
2.8 Implementation of the `eventcount` synchronization construct 27
2.9 Algorithm for EDC `suspend` method 28
2.10 Abstract metrics for a program that uses `futures` 32
2.11 Abstract metrics for a program that uses `phasers` 33
2.12 Abstract metrics for a program that uses `isolated` 35
2.13 HJlib deadlock example with `asyncAwait` tasks 37
2.14 HJlib deadlock example using `phasers` with explicit `signal` and `wait` 37
2.15 HJlib program that deadlocks when using eventcounts 38

3.1 `async-finish` example that starves on a thread-blocking runtime . . 46
3.2 Event-driven version of Figure 3.1 to avoid thread-blocking operations 47
3.3 Fibonacci program written using event-driven style 48
3.4 Example use of one-shot delimited continuations 51
3.5 `futures` implemented using the EDC API 53
3.6 Diagrammatic representation of the cooperative runtime 54
3.7 State of the call stack when the cooperative runtime executes a task . 56

3.8 Implementation of synchronization variables in the cooperative runtime 60
3.9 Implementation of barriers in the cooperative runtime 62
3.10 Architecture of the HJ language cooperative compiler 67
3.11 Architecture of the HJlib cooperative runtime and compiler 69
3.12 Cooperative runtime results for `async-finish` benchmarks 73
3.13 Cooperative runtime results for `future` benchmarks 76
3.14 Cooperative runtime results for `phaser` benchmarks 78

4.1 `async-finish` parallel search on a 2D Matrix 83
4.2 Parallel search with manual cooperative termination 86
4.3 Life-cycle of `Eureka` 90
4.4 Parallel search using the Eureka model 93
4.5 Parallel search using `CountEureka` 96
4.6 Parallel search using `MinimaEureka` 98
4.7 Example of parallel search with soft deadlines 100
4.8 Example of an iterative method using the Eureka model 101
4.9 Example of a parallel search on two elements using the Eureka model 103
4.10 Function decomposition involving nested eureka computations 105
4.11 Features supported by a Eureka task parallel runtime 106
4.12 Execution of Eureka tasks in a work-stealing environment 107
4.13 Configurations of the eureka benchmarks 109
4.14 Execution time metrics for eureka benchmarks 110
4.15 Productivity metrics for eureka benchmarks 112

5.1 Decomposition of an actor 118
5.2 Decomposition of a selector 121
5.3 Life cycle of a selector 122
5.4 Use of concurrent linked-list in mailbox 124
5.5 Scala actors solution to the Request-Response Pattern 127

5.6 Akka actors solution to the Request-Response Pattern 128
5.7 Selectors solution to the Request-Response Pattern 129
5.8 Actor network simulating a join pattern 130
5.9 Actors solution to the join pattern 131
5.10 Selectors solution to the join pattern 132
5.11 Selectors solution to the Reader-Writer Concurrency problem 137
5.12 Selectors solution to the Bounded-Buffer pattern 139
5.13 Selectors results for the Fork-Join Bank benchmark 141
5.14 Selectors results for the Chameneos benchmark 142
5.15 Selectors results for the LogisticMap benchmark 143
5.16 Selectors results for the Bank Transaction benchmark 144
5.17 Selectors results for the Filter Bank benchmark 145
5.18 Selectors results for the AdderJoin microbenchmark 146
5.19 Selectors results for the NQueens benchmark 147
5.20 Selectors results for the A* Search benchmark 148
5.21 Selectors results for the Dictionary benchmark 149
5.22 Selectors results for the Sorted List benchmark 150
5.23 Selectors results for the Bounded Buffer benchmark 150
5.24 Productivity metrics for selector/actor benchmarks 151

Chapter 1

Introduction

The only thing that will redeem mankind is cooperation.

Bertrand Russell

Multicore processors are now ubiquitous in server, desktop, and laptop hardware [1]. They are also making their way into smaller devices, such as smartphones and tablets. With the advent of the multicore era, it is clear that future growth in application performance will primarily come from increased parallelism. Parallelism, is hence, the future of computing. In embarrassingly parallel applications, parallel tasks rarely or never communicate. Other parallel applications have tasks that need to synchronize or communicate with each other more often [2]. This thesis addresses the effective execution of parallel applications on emerging multicore and manycore systems in the presence of modern synchronization and coordination constraints.

Designing and implementing correct and efficient parallel and concurrent programs is a notoriously challenging task. Parallel programming models ideally enable programmers to express parallel algorithms using abstractions that hide all but the relevant information. Thus, these models aid in increasing programmer productivity by reducing complexity and offer a scalable solution for the future where core counts are expected to increase. A dominant programming model for multicore processors is the Task Parallel Model (TPM), as exemplified by programming models such as Cilk [3], TBB [4], OpenMP 3.0 [5], UPC++ [6], Java's ForkJoinPool [7], Chapel [8], X10 [9], Habanero-C [10], and Habanero Java (HJ) [11]. All instances of the TPM

include primitives for task creation and termination, but they vary in the extent to which they include other modern constructs such as `futures` [12], `phasers` [13], and data-driven tasks [14]. The Actor Model (AM) [15, 16, 17] of concurrency has also recently gained popularity, mainly due to the success achieved by its flagship language - Erlang [18]. With the success of Erlang in production settings, the AM has catapulted into the mainstream and there has been a proliferation of the development of Actor frameworks in modern languages such as C/C++ (CAF[19], Act++ [20]), Smalltalk (Actalk [21]), Python (Stackless Python [22], Stage [23]), Ruby (Stage [24]), .NET (Microsoft's Asynchronous Agents Library [25], Retlang [26]). The TPM and AM offer promising approaches for developing parallel and concurrent systems with high productivity.

A major complexity that arises during the design and execution of parallel programs is the synchronization and coordination of parallel entities. Implementing synchronization and coordinating parallel entities in parallel programs is complex. The complexity is because incorrect implementations can lead to catastrophic conditions such as deadlocks or worse, result in reduced efficiency of the parallel application. While the basic TPM and AM can be used for writing parallel programs, efficient support for more general synchronization patterns that are necessary for a wide range of applications is often lacking. For example, barriers and `futures` are two typical synchronization patterns advocated by many industry multicore programming models that go beyond the fork-join model. Similarly, understanding and managing synchronization and coordination in an asynchronous AM is also hard; the AM's property of no shared state and asynchronous communication can make implementing coordination protocols harder than using established constructs such as locks [27]. The resulting AM code often intertwines both algorithmic logic and synchronization constraints without any separation of concerns.

Despite any productivity promises, a parallel programming model must be realizable in an efficient and scalable fashion for it to be useful to programmers. Poor

abstractions or implementations can introduce avoidable overheads that will likely limit the performance and scalability of parallel programs. Current implementations for the TPM provide efficient support for fork-join parallelism but are unable to support such more general synchronization patterns efficiently. Solutions for coordination protocols to support synchronization constraints may require actors to buffer messages and resend the messages to itself until the message is processed [28]. The net result of supporting synchronization and coordination constraints is that there is increased computation performed either in the runtime implementation or the user-level code trying implement such protocols. An alternative to the approaches mentioned above is to use a cooperative approach.

With a cooperative approach, an executing entity explicitly yields control to other entities at well-defined points during its execution. For example, cooperatively context switching among tasks in the same process can be implemented more efficiently than relying on the operating system to perform a relatively heavyweight thread-level context switch. This is, in part, because operating systems provide service guarantees such as isolation and fair resource allocation that need not be supported among tasks within the same process. We believe that a cooperation between the programming system and the runtime system is necessary to execute the available parallelism efficiently in the presence of general synchronization constraints. Cooperative techniques introduced in programming models can reduce parallel overheads in user-level code. The primary goal is to ensure that such an approach does not sacrifice the readability and maintainability of the code, while still delivering useful parallel performance.

1.1 Thesis Statement

Many modern synchronization and coordination constructs in parallel programs can incur significant performance overheads on current runtime systems, or significant productivity overheads when the programmer is forced to complicate their code to mitigate these performance overheads. Our thesis is that the use of cooperative schedul-

ing techniques can help address the performance and productivity challenges of using modern synchronization and coordination constructs. These techniques can be implemented using extensions to programming models, compilers, and runtime systems, depending on the desired trade-offs among performance, productivity, and portability.

1.2 Contributions

This dissertation makes the following contributions in defense of our thesis statement:

1. A pure library implementation of a rich task-parallel programming model called Habanero-Java library (HJlib). HJlib uses lambda expressions as parameters to syntactically simplify the code of programs, and can run on any standard Java 8 JVM without any other dependencies. The library supports a multitude of parallel and concurrent programming constructs and is used as a framework for both teaching and research. Expert library developers can extend HJlib to add custom parallel constructs. We have used HJlib as an implementation vehicle for the runtimes and programming models mentioned below.

2. A framework for cooperative scheduling of parallel tasks with general synchronization patterns. Our solution is founded on the use of *one-shot delimited continuations* and *single-assignment event-driven controls* to schedule tasks cooperatively in the presence of different synchronization patterns. We believe that any task-parallel synchronization construct can be supported by our cooperative runtime. A key challenge addressed by our runtime is that the resolution of a synchronization can, in general, trigger the enablement of multiple suspended tasks, a scenario that does not occur in traditional fork-join operations such as cooperative scheduling in Cilk [3, 29] programs.

3. The introduction of a new Eureka Programming Model (EuPM) to simplify the expression and management of speculative parallel tasks, which are especially important for parallel search and optimization applications. A pattern common

to such algorithms to solve these problems is a *eureka* event, a point in the program at which a task announces that a result has been found. Such an event can save computation time by avoiding further exploration of a solution space or by causing the successful termination of the entire computation. We have implemented a manifestation of the EuPM as a standard Java API, to support cooperative termination of avoidable tasks at well-defined program points. We also identify various patterns that are well-suited for the `Eureka` construct, but hard to implement using current parallel programming models. These include computations that exhibit patterns such as search, optimization, convergence, and soft real-time deadlines.

4. The introduction of a new extension to the asynchronous message passing actor model called *selectors*. This extension builds on our past work on integrating task parallelism with the actor model [30]. Selectors have multiple mailboxes, and each mailbox is guarded i.e. it can be cooperatively enabled or disabled to affect the order in which an actor processes messages. Selectors reduce the complexity of implementing many general synchronization and coordination patterns using actors. These patterns include *a*) synchronous request-reply; *b*) join patterns in streaming applications; *c*) supporting priorities in message processing; *d*) variants of reader-writer concurrency; and *e*) producer-consumer with a bounded buffer.

1.3 Outline

This thesis is organized as follows:

- Chapter 2 introduces the newly-created Habanero-Java library, a pure library implementation of the pedagogic Habanero-Java language [11].

- Chapter 3 discusses how we address the problem of scheduling tasks in a Task

Parallel Model with general synchronization patterns using a cooperative runtime.

- Chapter 4 describes the Eureka Programming Model that simplifies the expression of speculative parallel tasks and is especially well-suited for parallel search and optimization applications.
- Chapter 5 introduces our extension to the actor model called *selectors*, selectors have multiple cooperatively guarded mailboxes that simplify writing of synchronization and coordination patterns using actors.
- Chapter 6 discusses related work.
- Chapter 7 wraps up by summarizing the thesis and areas for future work.

Chapter 2

Habanero-Java Library: a Java 8 Framework for Multicore Programming

High achievement always takes place in the framework of high expectation.

Charles Kettering

In this chapter, we introduce Habanero-Java library (HJlib), a pure library implementation of the pedagogic Habanero-Java (HJ) language [11]. The library places a particular emphasis on the usability and safety of parallel constructs. For example, no HJlib program using ***async***, ***finish***, ***isolated***, and ***phaser*** constructs can create a logical deadlock cycle. In addition, the ***future*** and *data-driven task* variants of the `async` construct facilitate a functional approach to parallel programming. Event-driven programming is also supported via the *actor* programming construct. Finally, any data race free HJlib program written with `async`, `finish`, and `phaser` constructs is also guaranteed to be deterministic. Since 2014, HJlib is being used in teaching a sophomore-level course titled "Fundamentals of Parallel Programming" at Rice University. In addition to being an implementation of a pedagogic programming model, HJlib is actively being used in multiple research projects at Rice (including projects to exploit multicore parallelism in Java-based Hadoop "big data" applications [31, 32]) and also by external independent collaborators.

Among existing programming languages, Java provides a robust (though relatively low-level) support for multithreading and concurrency. The introduction of the Fork/Join Framework [7] in Java 7 gave Java developers the ability to start leverag-

ing parallelism at an application, rather than system-programming, level. We build on the capabilities offered by the Fork/Join Framework to provide a higher level of abstraction and a wider range of parallel programming constructs in HJlib. HJlib is built using lambda expressions and can run on any Java 8 JVM. Older JVMs can be targeted by relying on external bytecode transformations tools (e.g. retrolambda [33]) for compatibility. HJlib adds to the Java ecosystem a powerful and portable task parallel programming model that can be used to parallelize both regular and irregular applications.

HJlib supports an explicit parallel programming model and promotes an execution model for multicore processors based on three orthogonal dimensions for portable parallelism: *a*) lightweight dynamic task creation and termination; *b*) collective and point-to-point synchronization; and *c*) mutual exclusion and isolation. The HJlib runtime is responsible for orchestrating the creation, execution, and termination of tasks, and builds on a work-stealing scheduler (`java.util.concurrent.ForkJoinPool`) in the standard JDK. A separation of concerns enables programmers to focus on the design of their application, while the HJlib runtime delivers performance and scalability with respect to the available number of cores as well as the nature of parallelism that an application exhibits.

2.1 Contributions

Our work on Habanero-Java library (HJlib) makes the following contributions [34]:

- Habanero-Java library (HJlib): a pure library implementation of a pedagogic task-parallel programming model. HJlib is built using lambda expressions and can run on any Java 8 JVM without any other dependencies. The library supports a multitude of parallel programming constructs that include support for task, data, and pipeline parallelism; peer-to-peer synchronization; weak isolation; etc.

- The `EventDrivenControl` API which can be used to implement custom synchronization constructs while executing asynchronous tasks managed by the HJlib runtime. Existing HJlib synchronization constructs such as `future`s, data-driven futures, and `phaser`s, are all built using this API. Expert library developers can use this API to extend HJlib to add custom parallel constructs, if needed, while reusing our runtime.

- A new framework for Abstract Execution Metrics (AEM) which can be used to reason about the performance of parallel programs at the algorithmic level. Compared to previous work that supported only `async` and `finish` computations [11], our implementation supports AEM for the full gamut of synchronization constructs in HJlib. An API is available for users to add metrics support for custom synchronization constructs. The metrics can be particularly useful when debugging performance problems, and when comparing different parallel algorithms for the same problem.

- We introduce a new deadlock detector that detects deadlocks in HJlib programs with diagnostic information. The diagnostic information includes source code locations that help users in debugging their parallel programs. Our detector also supports custom synchronization constructs written using the `EventDrivenControl` API.

2.1.1 Outline

The chapter is organized as follows: in Section 2.2 we motivate the design of HJlib as a pedagogical Java library for exploiting multicore parallelism. Section 2.3 summarizes the parallel constructs supported in HJlib that enable writing of programs that utilize task, data, and pipeline parallel programming patterns. We describe our implementation and the `EventDrivenControl` API in Section 2.4: this API allows users to write their custom parallel constructs while reusing our runtime. Section 2.5

introduces the AEM framework supported by HJlib which can be used to reason about the performance of parallel programs at the algorithmic level. In Section 2.6, we describe the new deadlock detector available in HJlib. We summarize the chapter and outline possible directions for future work in Section 2.7.

2.2 Motivation

Programs typically exhibit varying degrees of task, data, and pipeline parallelism [35]. Current mainstream programming models are based on concurrent system programming primitives, and provide limited support for expressing parallelism at the application level. Programmers, hence, need higher level parallel programming models to reduce the burden of reasoning about and writing parallel programs. In addition, the programming model must carefully distinguish between parallelism (using deterministic task decomposition to utilize multiple computational resources effectively) and concurrency (correctly coordinating non-deterministic interactions among multiple tasks). At Rice, we have developed a pedagogic parallel programming model to address this issue as part of the Habanero Extreme Scale Software Research Project [36].

The previous implementation of the programming model was the Habanero-Java (HJ) language [11]. HJ includes a powerful set of task-parallel programming constructs that can be added as simple extensions to standard Java programs to take advantage of current and future multicore architectures. There are many practical advantages and disadvantages of choosing a language or a library approach. A key advantage of a language-based approach is that the intent of the programmer is easier to express and understand, both by other programmers and by program analysis tools [37]. However, a language-based approach requires the standardization of new language constructs for mainstream adoption.

The implementation of HJ requires language extensions to Java with special compiler support. The current HJ implementation is based on a subset of Java 5 (which includes generics, but not enums and foreach loops), which is a source of confusion

among students and collaborators. Although HJ generates Java bytecode, it is a language with its own type system separate from Java. For example, when a programmer instantiates a new `Integer` object, the implementation package is `hj.lang` rather than `java.lang`, which was a major source of confusion for external collaborators. It was also a major source of overhead in migrating existing Java code to HJ or using standard Java libraries and APIs in a newly written HJ program.

The use of library APIs to express all aspects of task parallelism has the drawback of allowing programmers to make lexical or scoping errors that could readily be caught by a compiler with a language-based approach. However, a key advantage of a library-based approach is that it can integrate with existing code more easily. It does not introduce any new language rules nor does it require modifying a compiler. In addition, users can write the code in an IDE of their choice and use their preferred debuggers.

Being an implementation of a pedagogic programming model, HJlib is also an attractive tool for educators. There are numerous educational resources available from the COMP 322 course offered at Rice University [38]. In addition to lecture notes and videos, HJlib has extensive documentation [39] and examples available to help users get started. Java is amongst the most popular languages taught in introductory programming courses [40, 41] and this bodes well for an increased use of HJlib as other institutions start introducing parallel programming earlier in their curricula. HJlib offers two main advantages in its use as an introductory parallel programming language for first or second year undergraduate students. First, students already know the Java language and are familiar with the Java compiler and runtime toolchain. They can build on this knowledge as HJlib programs are essentially Java programs and conform to the Java syntax and language rules. Second, since HJlib is a high-level parallel programming model, it makes it easier to focus on general parallel programming concepts, algorithms and patterns without being distracted by low-level details, such as threads and locks.

The ultimate goal of parallel computation is to increase performance by reducing computation time. Extracting performance from parallel solutions requires reasoning about the asymptotic complexity of the parallel algorithm. In HJlib, we use the notion of AEM to enable users to reason about the performance of their parallel algorithms. The metrics involve computing the total work done and the critical path length of the dynamic computation graph. These metrics can be used to reason about the ideal abstract execution time of the computation, assuming an infinite number of processors. While the HJ language supported AEM for `async-finish` computations, HJlib supports the metrics for a much wider variety of parallel constructs. The main advantage of using abstract execution times is that these performance metrics are reproducible; they will be the same regardless of which physical machine executes the HJlib program. HJlib also provides a deadlock detector to enable users to debug their programs while using the various synchronization constructs (if they venture beyond the deadlock-free subset of HJlib).

2.3 Background: Parallel Constructs

HJlib integrates a wide range of parallel programming constructs (e.g., `async`, `forall`, `isolated`, `future`s, data-driven tasks, barriers, `phaser`s, actors) in a single programming model that enables unique combinations of these constructs (e.g., nested combinations of task and actor parallelism). The orthogonal classes of parallel constructs allow programmers with a basic knowledge of Java to get started quickly with expressing a wide range of parallel programming patterns. HJlib is capable of expressing many different forms of parallel patterns including data parallelism, pipeline parallelism, stream parallelism, loop parallelism, and divide-and-conquer parallelism.

2.3.1 The Async/Finish Constructs

The Async/Finish Model (AFM) is a structured variant of the task parallel Fork/Join Model. It is well-suited to exploit task parallelism in divide-and-conquer style and

loop-style programs. In the AFM, a task can *fork* a group of child tasks. These child tasks can recursively fork even more tasks. All these descendant tasks can potentially run in parallel with each other. Tasks are created at *fork* points and HJlib provides the `async` method to create a task. The statement `async(()->` ⟨*stmt*⟩`)` causes the parent task to create a new child task to execute ⟨*stmt*⟩ (logically) in parallel with the parent task. An inner `async` is allowed to read and operate on a variable declared in an outer scope. The runtime is responsible for the scheduling of tasks created by `async`s.

Further, a parent/ancestor task can selectively *join* on a subset of child/descendant tasks to wait for their completion. This join is the primary form of synchronization between tasks in `async-finish` style programs. The `finish` method in HJlib is used to represent a join operation. The task executing `finish(()->` ⟨*stmt*⟩`)` has to wait for all transitively spawned child tasks inside ⟨*stmt*⟩ to terminate before it can proceed past the `finish` construct. All computations execute inside a global finish scope for the main program: the computation is allowed to terminate when all tasks nested inside the global finish terminate. This rule ensures that each executing task has a unique *Immediately Enclosing Finish* (IEF) [9, 13, 42].

Figure 2.1 shows a sample program that uses `async` and `finish` constructs to preserve task dependences while exploiting available parallelism. Note that until all forked tasks (Task A, Task B, Task B1, and Task B2) reach the join point, Task C cannot be executed. The scopes of `async` and `finish` can span method boundaries that simplify parallelizing sequential programs. `async`s are inserted to wrap statements that can be executed in parallel and then these `async`s are wrapped inside `finish` blocks to ensure the parallel version produces the same result as the sequential version.

`async-finish` style computations are known to be deadlock free [9]. In the absence of data races, these programs are deterministic [42]. The scopes of `async` and `finish` can span method boundaries. As a result, parallelizing sequential programs using

```
1 class AsyncFinishPrimer {
2   public static void main(String[] args) {
3     finish(() -> {
4       println("Task 0"); // Task-0
5       finish(() -> {
6         async(() -> { // Task-A
7           println("Task A");
8         });
9         async(() -> { // Task-B
10          println("Task B");
11          async(() -> { // Task-B1 created by ↩
                Task-B
12            println("Task B1")
13          });
14          async(() -> { // Task-B2 created by ↩
                Task-B
15            println("Task B2");
16          });
17        });
18      }); // Wait for tasks A, B, B1, B2 to ↩
            finish
19      println("Task C") // Task-C
20    });
21  } }
```

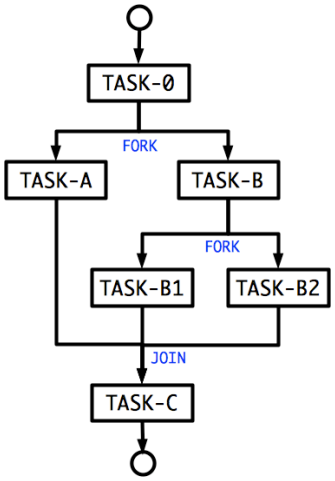

Figure 2.1 : HJlib version of a simple Fork/Join program using `async` and `finish` constructs for synchronization.

`async-finish` is fairly easy. `async`s are inserted to wrap statements which can be executed in parallel. Then these `async`s are wrapped inside `finish` blocks to ensure the parallel version produces the same result as the sequential version. This process of inserting `finish` blocks can also be automated [43].

2.3.2 Loop Parallelism

Loop parallelism is especially important when it comes to handling large sets of data in parallel. HJlib supports a variant of for loops over rectangular regions using the `forall` and `forasync` methods to spawn parallel tasks for each iteration inside the loop. HJlib also provides an implicit conversion for `Iterable`s to remove the restriction of rectangular regions and to simplify the use of `forall` and `forasync` methods on Java collections. Figure 2.2 displays the syntax and semantics of one-dimensional versions of `forall` and `forasync`. An implicit `finish` is included for all iterations of the `forall`, while there is no implicit `finish` in `forasync`. Another

```
1 forall(start, end, (i) -> f(i)) ==
2   finish(() -> {
3     for (int i = start; i <= end; i++) {
4       async(() -> f(i));
5     }
6   });
7 forasync(start, end, (i) -> f(i)) ==
8   for (int i = start; i <= end; i++) {
9     async(() -> f(i));
10  }
```

Figure 2.2 : Syntax and semantics of one-dimensional versions of `forall` and `forasync`.

typical way to take advantage of loop-level parallelism is to partition into chunks the data to be processed and create one computational task to process each chunk of data. HJlib supports the `forallChunked` and `forasyncChunked` methods to support this style of computation.

2.3.3 Coordination constructs

There are often dependences among parallel tasks. In such scenarios, coordination between tasks is required to determine when dependent tasks can be executed. HJlib augments the AFM with a handful of coordination constructs: `isolated`, `future`s, data-driven futures, `phaser`s, and actors. In addition, as we describe in Section 2.4, HJlib provides an API for end-users to be able to create their custom synchronization constructs.

Synchronized access using `isolated`

A concern common in most shared memory models is the issue of data races and the need to synchronize the accesses to shared resources/variables between tasks. HJlib provides an `isolated(() ->` $\langle stmt \rangle$`)` construct to support weak isolation, i.e. atomicity is guaranteed only with respect to other statements also executing inside `isolated` scopes. No guarantees are provided for interactions with non-`isolated` statements; accesses to shared variables by parallel tasks outside `isolated` blocks

```
1 public class IsolatedPrimer {
2   public static void main(String[] args) {
3     finish(() -> {
4       final int[] counter = new int[4];
5       finish(() -> {
6         forall(1, 399, (i) -> {
7           isolated(() -> {
8             int n = i % 4;
9             counter[n] = counter[n] + 1;
10          });
11        });
12        // the statement below would introduce a
13        // data race as outside an isolated scope
14        /* counter[0] = counter[0] + 1; */
15      });
16      for (int i = 0; i < 4; i++) {
17        assert("data race found", counter[i]==100);
18      }
19    });
20  }
21 }
```

Figure 2.3 : HJlib `isolated` statements at work. Each `isolated` block executes sequentially and there are no data races.

may participate in data races. In HJlib, we do not allow other parallel constructs to be nested inside `isolated` blocks and the runtime reports an error in such scenarios.

The example in Figure 2.3 demonstrates the use of global isolation that causes all `isolated` statements to be serialized. This serialization can be a serious performance bottleneck in applications with moderate contention [11]. HJlib adds support for finer-grained support to `isolated` blocks, called object-based isolation, of the form:

```
isolated(variable11, ..., variableN, () -> { stmt; })
```

which provides better performance. The last argument to the `isolated()` construct is a lambda expression that represents the statement(s) which execute under the weak isolation guarantee. The object-based isolation version serializes only conflicting `isolated` blocks, the conflicts are determined by the intersection of the representative set of locks for the objects $variable11, ..., variableN$.

```
1 public class FibFuturePrimer {
2   public int fib(int n) {
3     if (n < 2) {
4       return n;
5     } else {
6       HjFuture<Integer> x = future(() -> fib(n-1));
7       HjFuture<Integer> y = future(() -> fib(n-2));
8       return x.get() + y.get();
9   } }
10 }
```

Figure 2.4 : HJlib Fib using `future`s. A relatively large value of n will cause the program to run out of memory due to excessive creation of threads in the current HJlib runtime.

`futures`

A `future` represents the result of an asynchronous computation and extends HJlib `async` to support return values [12]. The statement `HjFuture<T> f = future<T>(() -> ⟨expr⟩)` creates a new child task to evaluate *expr* that is ready to execute immediately. In this case, `f` contains a ***future*** handle to the newly created task and the operation `f.get()` can be performed to obtain the result of the `future` task. If the `future` task has not completed as yet, the task performing the `f.get()` operation blocks until the `future` task completes and the result of *expr* becomes available. One advantage of using `future`s is that there can never be a data race on accesses to a `future`'s return value. In addition, if we store all `future`s in immutable variables, it ensures that we cannot create a deadlock cycle with `future` tasks.

While `future`s are simple to use (as seen in Figure 2.4), their injudicious use hurts the performance and scalability of HJlib programs when used with a thread-blocking runtime. This is because calls to the `get()` on the `future` object currently blocks the worker thread. In order to maintain parallelism, the runtime responds by creating more worker threads. Threads are heavyweight resources and the management of their life cycle is expensive and this hurts the program's performance. In addition to consuming resources such as memory, each thread requires two execution call stacks, which can be large [44]. Creating too many threads in one JVM can cause the system

to run out of memory or thrash due to excessive memory consumption. In Chapter 3, we show that such performance and scalability issues can be easily avoided by the use of a cooperative runtime. The cooperative runtime surrenders the thread (instead of blocking the thread) by suspending the task and reschedules the task when the producer populates a value inside the `future`.

Data-Driven Futures (DDFs)

DDFs are an extension to `future`s to support the dataflow model [14]. DDFs support a single assignment property in which each DDF must have at most one producer. Any `async` can register on a DDF as a consumer causing the execution of the `async` to be delayed until a value becomes available in the DDF. The exact syntax for an `async` waiting on a DDF is as follows: `asyncAwait(` $ddf1, ..., ddfN$`, () ->` $\langle stmt \rangle$`)`. An `async` waiting on a set of DDFs can only begin executing after a `put()` has been invoked on all the DDFs. Since tasks are scheduled for execution only after each DDF in the waiting set has been resolved, a call to the `get()` on the DDF will never cause the thread to block. Accesses to values passed inside DDFs that are members of the waiting set are guaranteed to be data race free and deterministic. However, invoking the `get()` on a DDF not present in the waiting set of an `async` is considered program error and has nondeterministic results. Note that no such restriction exists on the use of `get()` on `future` objects. Figure 2.5 represents a DDF version of the Fibonacci program from Figure 2.4. Using DDFs requires the programmer to change the natural flow of the program and to think in terms of callbacks.

`phasers`

The `phaser` construct [13] unifies collective and point-to-point synchronization for phased computations. Each task has the option of registering with a `phaser` in *signal-only/wait-only mode* for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. The latest release of Java includes `phaser` synchronizer ob-

```
1 public class FibDdfPrimer {
2   public int fib(int n, HjDDF<Integer> v) {
3     if (n < 2) {
4       v.put(n);
5     } else {
6       HjDDF<Integer> x = newDataDrivenFuture();
7       HjDDF<Integer> y = newDataDrivenFuture();
8       async(() -> fib(n-1, x));
9       async(() -> fib(n-2, y));
10      asyncAwait(x, y, v.put(x.get() + y.get()));
11  } }
12 }
```

Figure 2.5 : HJlib Fib using DDFs. Each call to `fib()` produces an `async` task that waits on values to be produced by its children before it computes the local result and stores it in the `v` (result) DDF. This version is more scalable compared to the `futures` version in Figure 2.4. It requires the programmer to change the natural flow of the program to think in terms of continuations and the DDFs.

jects, which are derived in part [45] from the `phaser` construct in the Habanero runtime. In general, a task may be registered on multiple `phaser`s, and a `phaser` may have multiple tasks registered on it. `phaser`s ensure deadlock freedom when programmers use only the `next` statements in their programs. In programs where tasks are involved with multiple point-to-point coordinations, explicit use of `doWait()` and `doSignal()` on multiple `phaser`s might be required. In such scenarios, some effort is required on the part of the programmer to reason carefully about the sequence of such calls to ensure correctness and deadlock freedom.

Finish Accumulators

Finish accumulators [46] support predefined parallel reductions for dynamic task parallelism. Finish accumulators are designed to be integrated into the `async` and `finish` constructs so as to guarantee determinism for accumulation and to avoid any possible race conditions in referring to intermediate results. Parallel tasks asynchronously transmit their data to finish accumulators with `put` operations and retrieve the results by `get` operations. A finish accumulator, `ac`, is accessible to sub-tasks if and only if `ac` is associated with a `finish` statement and the sub-tasks are created within the

`finish` scope. To ensure an absence of races, `get` operations by sub-tasks return the value at the beginning of the associated `finish` scope and are not affected by put operations within the same finish scope.

Finish accumulators are closely related to Cilk++ `reducer` objects [47]. Like Cilk++ `reducer`s, finish accumulators aggressively combine values from individual tasks in such a way that the operator need not be commutative to produce the same result as a serial execution. Cilk++ `reducer`s allow programmers to access intermediate results of `reducer`, which gives more flexibility to expert users but also increases the possibility of errors due to nondeterminism. On the other hand, finish accumulators specify an *end-of-finish* synchronization point where the reduction is to be completed (i.e. passively combines values), and prevents programmers from accessing incomplete intermediate results [46].

Actors

An actor is defined as an object that has the capability to process incoming messages produced by other actors [17]. Typically, the actor has a mailbox to store its incoming messages. An actor also maintains local state that is initialized at creation. Henceforth, only the actor is allowed to update its local state using data (usually immutable) from the messages it receives and from the intermediate results it computes while processing the message. The actor is restricted to process at most one message at a time. There is no restriction on the order in which the actor decides to process incoming messages. As an actor processes a message, it is allowed to change its behavior affecting how it processes the subsequent messages. From the AFM perspective, actors in HJlib are treated as long running `async`s and hence can nest any of the Async/Finish compliant constructs in their message-processing body [30]. This simplifies termination detection and enables exposing parallelism inside the actor while processing messages.

The COMP 322 course is also an excellent opportunity to get a sense of how

non-experts in parallel programming perceive HJlib. Overall, the use of HJlib in COMP 322 garnered positive feedback from students due to *a*) the lack of differences in rules from the Java language; *b*) the ability to use an IDE of their choice while writing programs; and *c*) the ability to debug their programs using standard Java IDEs. Encouraged by this, we are now planning to use HJlib for the COMP 322 equivalent of a massively open online course to be offered by Rice.

2.4 Implementation

HJlib is implemented as a pure Java library with no dependencies outside the standard JDK. The implementation relies heavily on the use of functional interfaces and lambda expressions introduced in Java 8. Lambdas allow the syntax of HJlib programs to be close to that of programs written in the HJ language. The entire library has been implemented from scratch to support all the parallel constructs described in Section 2.3. The implementation also publishes events for the life-cycle of individual tasks and allows custom listeners to register on these events. Supporting custom listeners enables the implementation of additional features such as abstract execution metrics (Section 2.5) and deadlock detection (Section 2.6). In addition, the library provides a core API (Section 2.4.1) to enable users to implement custom synchronization constructs (SyncCons). The implementation techniques described below will be useful for language designers and implementers (who may choose to build `async-finish` style task-parallel implementations on other serial languages or target architectures). It will also aid application and library programmers who may build their frameworks on top of HJlib.

An HJlib application execution can be modeled as a directed acyclic graph, where nodes represent computational tasks and edges define the data dependences among them. The HJlib runtime is responsible for orchestrating the creation, execution, and termination of tasks. The management of actual threads and related thread pools is done by the runtime and is transparent to the tasks in the program. The HJlib

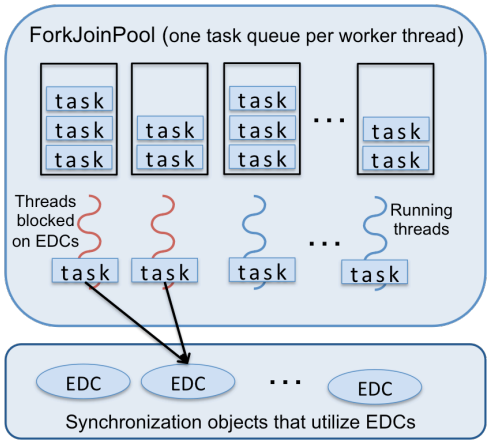

Figure 2.6 : The HJlib runtime includes worker threads and ready task queues like most other task parallel runtimes. In addition, there are `EventDrivenControl`s which maintain a list of blocked tasks to implement higher-level synchronization constructs.

runtime schedules tasks whose dependences have been satisfied over worker threads.

HJlib uses the `ExecutorService` and Fork/Join framework [7] to manage the parallel execution of tasks using work-stealing schedulers. Our runtime uses the *help-first* policy [48] for task scheduling. Under this policy, spawning a child task enqueues it in the task queue and allows the parent task to continue execution past the spawn operation. The child task can then be executed by any of the worker threads.

The Fork/Join framework is designed to make divide-and-conquer algorithms easy to parallelize. HJlib provides additional synchronization constructs (SyncCon) (such as `future`s, `phaser`s, actors, etc.) that allow parallelizing a larger class of algorithms and applications. These SyncCons are implemented using the `EventDrivenControl` data structure described in Section 2.4.1. Synchronization constraints can prevent a currently executing task from making further progress as it *waits* to synchronize with other ready but not executing task(s). Our runtime implements such waits by blocking the worker thread, but first it smartly attempts to make further progress in the overall computation (Section 2.4.3).

The HJlib runtime is also responsible for the management of data structures to resolve blocking conditions in the presence of arbitrary dependences or synchronization

constraints. Figure 2.6 provides a diagrammatic explanation of the HJlib runtime. The runtime places tasks into queues; the pool of worker threads continuously attempts to execute tasks dequeued from these queues. In addition, execution of tasks may result in more tasks being spawned and enqueued into the queues. Worker threads may get blocked due to synchronization constraints on the tasks. When the synchronization constraint is resolved, the task is unblocked, and the worker thread can resume execution. An application starts with a single *main* task in the work queue which promptly gets executed by one of the worker threads. The application terminates when: *a*) the work queues are empty, and *b*) all synchronization constraints in the program have been satisfied (i.e. there are no deadlocks).

2.4.1 `EventDrivenControl` API

Event-Driven Controls (EDCs) are an extension to Data-Driven Controls (DDCs) which were earlier presented in [30] and used to support event-driven actors in a task-parallel runtime. A DDC lazily binds a value and a closure called the execution body (EB), both the value and the EB follow the dynamic single-assignment property ensuring data-race freedom. When the value becomes available, the EB is executed using the provided value. We generalize DDCs to EDCs [49, 34] to allow multiple EBs to be attached to the EDC as callbacks. We treat the availability of a value in the EDC as an *event* and use the event to trigger the execution of EBs. Due to the single-assignment property, the registered EBs are executed at most once. We also allow multiple values to be added into the EDC as long as the values are *logically equivalent*, this does not violate the dynamic single assignment property, and it does not trigger re-executions of the EBs. Attempting to add unequal values into the EDC is reported as a runtime error. Figure 2.7 shows a simplified implementation of an EDC excluding concurrency concerns. The EB of the EDC executes either asynchronously or synchronously. For example, in a task-parallel runtime the EB could store bookkeeping data and act as a synchronous callback into the runtime.

```
1 class EventDrivenControl {
2   ValueType value = ...;
3   List<ExecBody> ebList = ...;
4   /** triggers callback execution **/
5   void setValue(ValueType theValue) {
6     if (!valueAvailable()) {
7       value = theValue;
8       // execute the callbacks/EBs
9       ebList.each().scheduleWith(value);
10    } } else {
11      // check for error
12  } }
13  /** enables callback registration **/
14  void addExecutionBody(ExecBody theBody) {
15    if (valueAvailable()) {
16      // value available, execute immediately
17      theBody.scheduleWith(value);
18    } else {
19      // need to wait for the value
20      ebList.add(theBody);
21 } } }
```

Figure 2.7 : Simplified representation of an EDC not displaying synchronizations or validations. Both the value and the execution body can be lazily attached. The execution body uses the value of the EDC in the `scheduleWith()` method; it can trigger the execution of asynchronous or synchronous code.

The EB could trigger possible asynchronous actions, such as scheduling and execution of a task, by interacting with the runtime.

To allow library/language developers to create custom SyncCons, we expose EDCs as an API in our runtime. The `EventDrivenControl` API contains the following operations:

- The static `newEDC()` factory method is used to instantiate a new EDC. EDCs are initialized without a resolved value and with an empty EB list. An EDC can be used like a regular object, e.g. stored as a field, passed around as a parameter, invoked as a receiver for methods, etc.

- The static `suspend(anEdcInstance)` method signals the possible creation of a suspension point. If the EDC passed as an argument has not been resolved, the current task (and the underlying worker thread) is **blocked** until the EDC is resolved.

- The `setValue(someValue)` method resolves the EDC, i.e. it binds a value with the EDC and triggers the execution of any EBs registered with the EDC. Tasks (and hence worker threads) blocked on calls to `suspend` on this EDC instance will get unblocked as the EDC is resolved.

- The `getValue()` method retrieves the value associated with the EDC. It is only safe to call this method if the value in the EDC has already been resolved. If the execution proceeds past a call to `suspend()`, it is guaranteed that a value is available in the EDC.

- The `isValueAvailable()` method can be used to check whether the value in the EDC has been resolved.

As we describe in Section 3.3.2, the same API with minor semantic differences in the `suspend()` and `setValue()` methods is used by the cooperative runtime for HJlib.

The API is used in our runtime implementation to support synchronization constructs such as `finish`, `future`s, `phaser`s, etc. The implementation details for each of these constructs are described in Section 3.4. The key idea is to translate the SyncCons into producer-consumer constraints on EDCs and to suspend the consumer of an EDC when waiting on the producer to resolve the EDC. We claim that any task-parallel SyncCon can be translated in such a manner and hence be supported by the HJlib runtime. As a result, library developers can use this API and implement their custom SyncCons and synchronization patterns. In Section 2.4.2, we show how this API can be used to implement a new SyncCon called EventCount in HJlib.

2.4.2 EventCount Synchronization Construct

An *`eventcount`* is an object that keeps a count of the number of events in a particular class that have occurred so far in the execution of the system [50]. The primitive operations on an `eventcount` may be concurrent. Events are the executions of three primitive operations:

- `advance`: It is used to signal the occurrence of an event. The effect of this operation is to increase the integer value of the `eventcount` by 1.

- `await(v)`: It suspends the calling task until the value of the `eventcount` is at least `v`. The await primitive may not return immediately once the $\mathtt{v}^{th}$ advance on the `eventcount` is executed; the only guarantee is that at least `v` advances have been performed by the time `await(v)` returns.

- `read`: The value returned by `read()` counts all of the `advance` operations that precede the execution. It may or may not count those in progress during the `read`.

Figure 2.8 highlights a simple implementation of the `eventcount` synchronization construct using EDCs. A `Map` is used to track the representative EDC for each event value.* The `advance` operation increments the counter atomically and then resolves the EDC corresponding to that event count (line 15). The `await` operation looks up the EDC entry corresponding to the event count `v` and then calls the `suspend` method on the EDC (line 21). If the EDC has already been resolved, the `suspend` method will return immediately. Otherwise, the `suspend` method will block until an appropriate call to `advance` causes the count to reach `v` and the EDC is resolved. The `read` operation is the simplest to implement; it looks up the value of the counter and returns it. Note that the `await` operation is blocking (since it invokes `suspend`), our runtime will allow a task executing the `await` operation to benefit from the optimization described in Section 2.4.3.

2.4.3 Handling Blocking Operations

In the presence of synchronization points from constructs such as `future`s, barriers, and `phaser`s, our implementation reverts to thread-blocking scheduling of tasks. The

*A more sophisticated implementation would minimize the entries stored in the `Map` to avoid memory leaks.

```
1 public final class EventCount {

3   Map<Long, EventDrivenControl> eventMap = new ConcurrentHashMap<>();
4   AtomicLong eventCounter = new AtomicLong(0);

6   public EventCount() {
7     EventDrivenControl edc = newEDC();
8     eventMap.put(0L, edc);
9     edc.setValue(Boolean.TRUE);
10  }
11  public void advance() {
12    long v = eventCounter.incrementAndGet();
13    eventMap.putIfAbsent(v, newEDC());
14    EventDrivenControl edc = eventMap.get(v);
15    edc.setValue(Boolean.TRUE);
16  }
17  public void await(final long v) {
18    eventMap.putIfAbsent(v, newEDC());
19    EventDrivenControl edc = eventMap.get(v);
20    // rely on the EDC inside the map to control when the task is unblocked
21    suspend(edc);
22  }
23  public long read() {
24    return eventCounter.get();
25  }
26 }

28 public class EventCountPrimer {
29   public static void main(String[] args) {
30     finish(() -> {
31       final EventCount ec = new EventCount();
32       forasync(1, 10, (i) -> {
33         println("Task-" + i);
34         ec.advance();
35       });
36       ec.await(5);
37       println("At least 5 iterations completed");
38     });
39   }
40 }
```

Figure 2.8 : Implementation of the `eventcount` synchronization construct using `EventDrivenControl`s. The example program in `EventCountPrimer` shows the use of the `advance` and `await` operations. The `await` at line 36 waits for *any* 5 iterations from the `forasync` loop to complete.

scheduler then spawns additional worker threads to compensate for blocked worker threads. But this adds to overhead in the runtime as each thread needs its own system resources. Our implementation minimizes the overheads of blocking by attempting to make progress in the overall computation before blocking the worker thread. When a thread detects it is going to block (in the `EventDrivenControl.suspend()` method

```
1 public static void suspend(EventDrivenControl edc) {
2   // execute tasks, return early if EDC is resolved
3   while (!edc.isValueAvailable() &&
4       !nbTasks.isEmpty()) {
5     final Task task = nbTasks.pop();
6     task.run();
7   }
8   // check once again, attempt to avoid blocking
9   if (edc.valueAvailable()) {
10    return;
11  }
12  // Use latch and block until EDC is resolved
13  final CountDownLatch cdl = new CountDownLatch(1);
14  edc.addExecutionBody((v) -> cdl.countDown());
15  // arrange for spawning additional worker thread
16  // before blocking the worker thread
17  handleAwait(cdl);
18 }
```

Figure 2.9 : Algorithm for EDC `suspend` method. The runtime tries to execute non-blocking tasks before blocking the worker thread.

call), it searches for 'non-blocking' tasks and executes them before blocking. This requires HJlib to distinguish between tasks that contain blocking operations and those that do not, we enforce that by having users implement different functional interfaces for the bodies of their tasks. This distinction is a useful optimization since the user does not need to provide an explicit continuation task when a blocking condition is discovered. The non-blocking tasks are duplicated in a separate concurrent queue (so they can be found efficiently) during task creation and removed from the queue when they begin execution. Often executing these non-blocking tasks can cause the EDC to get resolved, and the `suspend()` method can return without blocking the worker thread. This optimization is safe as it disallows a deadlock scenario where a blocking point is reached which can only be resolved by executing a task earlier in the call stack. The runtime ensures that non-blocking tasks do not inadvertently call blocking operations and reports a runtime error if this occurs.

The algorithm for the `suspend()` method is displayed in Figure 2.9. The runtime tries to execute non-blocking tasks before blocking the worker thread (lines 4-7). If while executing the non-blocking tasks, the EDC gets resolved, the `suspend` method returns (line 10) without blocking the thread. If all the available non-blocking tasks

get executed and the EDC has still not resolved, the runtime goes ahead to block the worker thread until the EDC is resolved (line 17). Our implementation uses a `CountDownLatch` to actually block the thread by making a call to the `await` method. A `CountDownLatch` is initialized with a given count and the `await` methods block until the current count reaches zero due to invocations of the `countDown` method. Once the current count reaches zero, all waiting threads are released and any subsequent invocations of `await` return immediately.

2.5 Abstract Execution Metrics

In order to effectively parallelize a program, we need to know which parts of a program take the most computation time. We also need to know the computation time required by different fragments of a program. Using actual computation time may not be repeatable as it includes system times such as delays due to cache misses and thread context switches. As an alternate we can measure the performance by measuring the number of abstract operations performed. These operations can be weighted combinations of floating-point operations, comparison operations, stencil operations, or any other data structure operations. We call such metrics for performance measurement abstract execution metrics (AEM). The AEM can be particularly useful when debugging performance problems, and when comparing alternate implementations of an application (e.g. sequential vs. parallel implementation). For example, students in the COMP 322 course first reason about the performance of their parallel programs (e.g. array sum, quicksort, etc.) by analyzing the AEM before looking at real execution times for performance later in the course. The main advantage of using abstract execution times is that the performance metrics will be the same regardless of which physical machine the HJlib program is executed on. The main disadvantage is that the abstraction may not be representative of actual performance on a given machine.

In addition to tracking the total number of abstract operations performed by

a computation, the HJlib runtime also computes the critical path length (CPL) of the computation. The runtime dynamically forms the directed-acyclic graph that represents the computation while scheduling tasks and the critical path is the longest necessary path through this graph when respecting dependences between tasks. The critical path is useful since it determines the shortest time (in terms of operations) possible to complete the entire computation.

The HJlib runtime provides an API for the programmer to register and request AEM. The programmer can insert a call of the form, `doWork(N)`, anywhere in a task to indicate execution of `N` application-specific abstract operations. Multiple calls to `doWork()` are permitted within the same task, they have the effect of adding to the abstract execution time of that task. The HJlib runtime maintains metrics for each task and coordinates the logic for aggregating the metrics among tasks for the various synchronization constructs (i.e. `future`s, `phaser`s, `isolated`, actors, etc.). Like the EDC API, an AEM API allows users to implement metrics support for their custom synchronization constructs. Using the AEM API, the metrics for the overall program can be found by querying for the AEM after execution of the program.

The `-Dhj.abstractMetrics=true` option is used when executing an HJlib program to enable AEM. The metrics for a given task captures the total number of operations executed (WORK) and the critical path length (CPL) of the call graph generated by the program execution for that specific task. The ratio, WORK/CPL, is also printed as a measure of the maximum possible speedup for the program from its dynamic computation graph. This ratio is useful for programmers tuning their application for parallel speedup. It can be viewed as the maximum performance improvement factor due to parallelism that can be obtained if we ideally had an unbounded number of processors. The AEM for a task can be obtained and printed using the following code snippet:

```
final HjMetrics metrics = abstractMetrics();
AbstractMetricsManager.dumpStatistics(metrics);
```

2.5.1 Computing Critical Path Length

Non-blocking (`async`) task The CPL of a non-blocking task is trivial to compute. It equals the total work (represented by calls to `doWork()`) done in the task. Blocking tasks contain other synchronization constructs; we describe their CPL computation in the following paragraphs.

Finish scope Naively, one would assume the CPL of a `finish` scope to be the maximum of the CPLs of the tasks spawned inside the `finish` scope. However, one must take care of the sequential part of the computation inside the `finish` block and the dependence restrictions among the spawned tasks. The *representative* CPL of each task inside a `finish` scope consists of the sum of three fragments:

- Start CPL: This is the CPL inside the `finish` scope before the task starts executing. It is the maximum of two values: the CPL of the fragment executed inside the finish scope before this task is spawned and the CPL of dependences that delay the task from starting execution.
- Task CPL: The actual CPL of the task's body when it is executed.
- End-Continuation CPL: This is the CPL of any computation inside the `finish` scope that is executed after the task ends execution.

The CPL of a `finish` scope is the maximum of the CPL of the sequential part and the maximum of the representative CPLs of all tasks spawned inside the `finish` scope.

`future` and Data-Driven Future Computing the CPL of a `future` task follows the same rules as computing the CPL of an asynchronous task. It equals the sum of the Start CPL and the Task CPL; we call this the *completion* CPL. The more interesting case is while computing the CPL at the point where the `get()` operation is invoked. Whenever a task invokes a `get()` operation on a `future`, the CPL of the task needs to be updated to the maximum of its current CPL and the completion

```
1 public class FutureMetrics {
2   public static void main(String[] args) {
3     HjSystemProperty.abstractMetrics.set(true);
4     finish(() -> {
5       finish(() -> {
6         HjFuture<Integer> f1 = future(() -> {
7           doWork(2);
8           return 10;
9         });
10        async(() -> {
11          doWork(1);
12          f1.get();
13          doWork(3);
14        });
15      });
16      HjMetrics actualMetrics = abstractMetrics();
17      // Metrics: WORK=6, CPL=5
18    });
19  }
20 }
```

Figure 2.10 : Abstract metrics for a program that uses `futures`.

CPL of the `future`'s producer task. In contrast, computing the CPL while using data-driven futures is much simpler since the start of the task is delayed due to input dependences of the `asyncAwait` clause. So, the CPL of the `get` operation is already accounted for in the start CPL of the task.

Figure 2.10 displays an example HJlib program that report metrics while using `future`. The `future` created in line 6 does two units of work before resolving the value of the `future` (by returning 10). The `async` created at line 10 does one unit of work before trying to retrieve the value of the `future`. At this point, the CPL for the task is updated to two (maximum between CPL of the task and completion CPL of the `future`). As the task completes execution, it does three more units of work and the CPL of the task is updated to five. When the `finish` scope completes, it updates its CPL to the maximum of all the tasks and sets the CPL to five. The total work done in the program is the aggregate of all the arguments passed to `doWork()` and is six in this example. As a result, we can obtain a maximum parallel speedup of 1.20 from this program.

```
1 public class PhaserMetrics {
2   public static void main(String[] args) {
3     HjSystemProperty.abstractMetrics.set(true);
4     finish(() -> {
5       finish(() -> {
6         HjPhaser ph = newPhaser(SIG_WAIT);
7         asyncPhased(ph.inMode(SIG_WAIT), () -> {
8           doWork(1);
9           next();
10          doWork(5);
11          next();
12          doWork(3);
13        });
14        asyncPhased(ph.inMode(SIG_WAIT), () -> {
15          doWork(3);
16          next();
17          doWork(1);
18          next();
19          doWork(5);
20        });
21      });
22      HjMetrics actualMetrics = abstractMetrics();
23      // Metrics: WORK=18, CPL=13
24    });
25  }
26 }
```

Figure 2.11 : Abstract metrics for a program that uses `phasers`.

`phasers` Tasks that use `phasers` are part of multi-phased computations involving other tasks. In such computations, the CPL needs to be updated at each phase for all the participating tasks. `phasers` produce two main events, `signal` and `wait`. At each of these events, a local metric object associated with the `phaser` needs to be updated for its CPL (and WORK). When a task enters the `signal` state on a `phaser` (by calling the `doSignal()` method), the task needs to update the CPL of the metric associated with the phase. The updated CPL is the maximum of the task's current CPL and the existing CPL of the `phaser`. When a task enters the `wait` state on a `phaser` (by calling the `doWait()` or `next()` methods), the task needs to update its own CPL with the maximum of its current CPL and the CPL value of the `phaser` at that phase.

Figure 2.11 displays an HJlib program that reports abstract metrics for `phaser`-based programs. The two `asyncs` at line 7 and line 14 are registered in `SIG_WAIT` mode and hence treat the `next()` operation as a barrier. As a result, the computation has

three phases delineated by the two `next()` operations. The total work done in the application is 18. However, the CPL is computed as the CPL of each phase. The `async` at line 14 provides the CPL for the first phase (5 units) and third phase (3 units). The `async` at line 7 provides the CPL for the second phase (5 units). Consequently, the CPL of the program is the sum of the three CPLs, i.e. 13 units .

Actors Like `phaser`s do for each phase of phased-computations, each actor maintains its own metric object to store the CPL and work. The two main events associated with actors are sending of messages and processing of messages; these events affect the CPL of an actor-based computation. When a message is sent to an actor, the message is packaged with a metric object wrapping the CPL of the sender. As a message is picked to be processed from its mailbox, the actor updates its CPL to the maximum of the current CPL and the CPL of the message sender. Then as this message is processed by the actor, the CPL is updated like the CPL for any other task. When the actor finally exits, it notifies its enclosing `finish` scope of its total work and CPL. This ends up updating the CPL of the `finish` scope as mentioned earlier.

Isolated Since HJlib maintains multiple locks to support both global and object-based isolation, a metric object is maintained for each lock. When the runtime uses a lock in some `isolated` code fragment, it serializes the computation for other code fragments that will use the same lock. As each lock is successfully acquired, the CPL of the task's metric is updated to the maximum value using the lock's CPL. After the `isolated` block completes execution and before releasing the lock, the CPL from lock's metric is updated to the maximum of the current value and the task's CPL.

Figure 2.12 displays an HJlib program that reports AEM for programs using `isolated` statements. Lines 6 and 13 create two `async`s that each do 7 units of work. In the absence of the `isolated` statements, the CPL of the program would be 7 units. However, the presence of the `isolated` statements causes serialization of

```
1 public class IsolatedMetrics {
2   public static void main(String[] args) {
3     HjSystemProperty.abstractMetrics.set(true);
4     finish(() -> {
5       finish(() -> {
6         async(() -> {
7           doWork(2);
8           isolated(() -> {
9             doWork(3);
10          });
11          doWork(2);
12        });
13        async(() -> {
14          doWork(2);
15          isolated(() -> {
16            doWork(3);
17          });
18          doWork(2);
19        });
20      });
21      HjMetrics actualMetrics = abstractMetrics();
22      // Metrics: WORK=14, CPL=10
23    });
24  }
25 }
```

Figure 2.12 : Abstract metrics for a program that uses `isolated`.

the two fragments that each do 3 units of work. As a result, the CPL of the overall computation goes up to 10 units.

2.6 Deadlock Detection

A deadlock occurs when one or more tasks come into conflict over some resource, in such a way that no further execution is possible for at least one of the tasks. Deadlock prevention is the name given to schemes that guarantee that deadlocks can never happen because of the way the synchronization constructs are structured. No HJlib program using `async`, `finish`, `isolated`, and `phaser`[†] constructs can create a logical deadlock cycle. However, programmers can use some of the other constructs (DDFs, actors, eventcounts, custom synchronization constructs built using EDCs, etc.) to write programs that deadlock. For example, programs written using actors

[†]Using only the `next()` operation

might forget to terminate the actor or programs using DDFs may have faulty logic that never resolves the DDF causing awaiting `async` tasks to be blocked forever.

The HJlib runtime follows the deadlock detection scheme where deadlocks are allowed to occur. The runtime continuously examines the computation to detect that a deadlock has occurred. In our current implementation, the runtime does not attempt to correct the deadlock. Instead, the runtime reports a diagnostic error message with tasks participating in the deadlock and terminates the program. When configured to do so, the runtime also includes the exact line number of the source code where each task is blocked in its computation. Since the blocked tasks are available, they can be queried for their stack trace just before blocking to obtain the user code that caused the blocking operation. The diagnostic information provided to the user can simplify debugging of the program.

Our basic algorithm for detecting deadlocks relies on tracking the number of ready tasks in the work queue and the number of blocked or pending tasks. Pending tasks are identified by tasks that have been created, but not scheduled for execution. Blocked tasks are discovered by their use of the `suspend()` method on EDCs. Since all our synchronization constructs are built using EDCs, all constructs are supported by the deadlock detector. Whenever a task blocks, it is added to a blocked tasks collection, and when it gets unblocked, it is removed from the collection. If the work queue becomes empty and there are no more actively executing tasks on the worker threads, there is no more computational progress being made by the program. In such a state, if the blocked tasks collection is non-empty, then we have a deadlock. The runtime immediately discovers this state (i.e. it does not use a polling strategy at intermittent intervals) and reports it. The drawback of this approach is that the runtime needs to wait for all ready tasks to complete executing before reporting the deadlock.

Figure 2.13 displays an example of a deadlock when writing HJlib programs with DDFs. The programs deadlocks as tasks at line 8 and line 12 are never executed as their dependences are never satisfied. The program instantiates two DDFs, but

```
1 public class DdfDeadlockExample {
2   public static void main(String[] args) {
3     HjSystemProperty.trackDeadlocks.set(true);
4     finish(() -> {
5       HjDataDrivenFuture<Integer> right = newDDF();
6       HjDataDrivenFuture<Integer> left = newDDF();
7       // waits on left before resolving right
8       asyncAwait(left, () -> {
9         right.put(1);
10      });
11      // waits on right before resolving left
12      asyncAwait(right, () -> {
13        left.put(2);
14      });
15  });
16  }
17 }
```

Figure 2.13 : A simple HJlib program that deadlocks because the `asyncAwait` tasks at line 8 and line 12 are never executed. As a result, the `finish` at line 4 cannot complete.

```
1 public class PhaserDeadlockExample {
2   public static void main(String[] args) {
3     HjSystemProperty.trackDeadlocks.set(true);
4     finish(() -> {
5       HjPhaser ph1 = newPhaser(SIG_WAIT);
6       HjPhaser ph2 = newPhaser(SIG_WAIT);
7       asyncPhased(ph1.in(WAIT), ph2.in(SIG), () -> {
8         ph1.doWait(); {
9         ph2.signal(); {
10      });
11      asyncPhased(ph1.in(SIG), ph2.in(WAIT), () -> {
12        ph2.doWait(); {
13        ph1.signal(); {
14      });
15    });
16  }
17 }
```

Figure 2.14 : HJlib program that deadlocks when using `phaser`s with explicit `signal` and `wait` operations.

there is no execution flow that resolves these DDFs at lines 8 and 12. Since the tasks do not execute, the `finish` at line 4 cannot complete the join operation, and no computational progress will occur. The deadlock detector will identify this scenario as soon as line 15 executes and report that *three* tasks are involved in a deadlock. These tasks are the main task running the `finish` block and the two `asyncAwait` tasks.

```
1 public class EventCountDeadlockExample {
2   public static void main(String[] args) {
3     HjSystemProperty.trackDeadlocks.set(true);
4     finish(() -> {
5       HjEventCount eventCount = newEventCount();
6       forasync(1, 10, (i) -> {
7         eventCount.advance();
8       });
9       async(() -> {
10        eventCount.await(20);
11      });
12    });
13  }
14 }
```

Figure 2.15 : HJlib program that deadlocks when using eventcounts.

No HJlib program using `phaser`s and only `next()` operations can create a logical deadlock cycle. However, this guarantee is lost when users write programs that call the `signal` and `wait` operations explicitly. Figure 2.14 displays such a program that deadlocks due to incorrect use of the `signal` and `wait` operations. Line 7 creates an `async` task that is registered in wait mode on the first `phaser` and in signal mode on the second `phaser`. Similarly, line 11 creates an `async` task that is registered in signal mode on the first `phaser` and in wait mode on the second `phaser`. Both tasks perform the wait operation in the current phase as the first line of their computation (lines 8 and 12). Since the corresponding `phaser`s have pending signals for the current phase (lines 9 and 13), both the tasks block indefinitely and the program deadlocks. The `doWait()` operations use EDCs under the covers that allow the deadlock detector to identify the blocked tasks.

Figure 2.15 displays a program that deadlocks while using custom SyncCons implemented by a user, e.g. the `eventcount` construct described in Section 2.4.2. Since the `await` operation is implemented using EDCs, the deadlock detector can track when the task using eventcounts get blocked. Line 6 creates ten tasks that advance the value of `eventCount` to ten. However, `async` task created at line 9 is waiting for the `eventCount` to reach a value of twenty (at line 10). As a result, this task will block indefinitely, and the deadlock detector can report the deadlock.

2.7 Summary

In this chapter, we introduced Habanero-Java library (HJlib), which is a pure library implementation of our pedagogic parallel programming model. HJlib is built using lambda expressions and can run on any Java 8 JVM. While programmers focus on the design of their application, the HJlib runtime delivers performance and scalability with respect to the available number of cores as well as the nature of parallelism that an application exhibits. HJlib offers the `EventDrivenControl` API, which can be used to implement custom synchronization constructs while executing asynchronous tasks managed by the HJlib runtime. Finally, the HJlib runtime feedback capabilities such as the AEM and the deadlock detector help the programmer to get feedback on theoretical performance as well as the presence of potential bugs in their program.

The usability of HJlib has been assessed in various occasions when porting benchmarks from Java, C, or Fortran to HJ. HJlib has also been used in the introductory parallel programming class for second-year undergraduate students at Rice University (COMP 322). It has allowed students to build on their previous knowledge of Java and focus on the fundamentals of parallel programming and algorithm design instead of being distracted by the low-level intricacies of using a Java API like the `java.util.concurrent` library. By relying on simple orthogonal parallel constructs with important safety properties, HJlib allows programmers to harness the power of multicore architecture. It enables programmers with a basic knowledge of Java to get started with parallel programming concepts by writing or refactoring applications.

Additionally, HJlib is actively being used in multiple research projects at Rice and also by external independent collaborators. HJlib is an attractive tool for researchers exploring parallel runtimes and programming models. In the next chapter (Chapter 3), we will discuss our work on building a cooperative runtime for HJlib, which allows blocking semantics in synchronization constructs to be implemented without blocking the underlying worker thread. In Chapter 4, we describe the runtime and programming model we built to support eureka-style computations. An essential

ingredient of our implementation is the augmented support for task termination in HJlib. Chapter 5 introduces our work on selectors that are an extension of the actor model implementation in HJlib.

Chapter 3

Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns

Scheduling me is not easy, as most people know, because once I start, I don't stop (yield).

Donna Karan

With the advent of the multicore era, it is clear that future improvements in application performance will primarily come from increased parallelism in software. Worker threads are commonly used to support execution of shared-memory parallel programs as opposed to processes. This use of threads yields a significant performance advantage, typically an order of magnitude better, relative to traditional processes [51]. The performance advantages come from saving overheads in context switching and communication. Context switching between threads in the same process is typically faster than context switching between processes because there is less state to change. In particular, this means avoiding the expensive operation of switching the process' memory address space. In addition, communication between processes is quite difficult and resource-intensive due to the security constraints enforced on the isolated address space. All threads running within a process share the same address space. This sharing allows threads to read from and write to the same data structures, and it also facilitates communication between threads.

A dominant parallel programming model for multicore processors utilizing threads is the Task Parallel Model (TPM). TPM implementations include Cilk [3], TBB [4], OpenMP 3.0 [5], Java's ForkJoinPool [7], Chapel [8], X10 [9], Habanero-C [10], and

Habanero Java (HJ) [11]. Current implementations for the TPM provide efficient support for fork-join parallelism, but are unable to efficiently support more general synchronization patterns that are important for a wide range of applications. In the presence of patterns such as `futures` [12], barriers, and `phasers` [13], current TPM implementations revert to blocking worker threads on context switches for all synchronization operations other than task termination. Barriers and `futures` are two common synchronization patterns advocated by many industry multicore programming models that go beyond the fork-join model. But, there is as yet no demonstration of an effective solution to schedule programs with `futures` and barriers in a scalable fashion when the number of blocked tasks exceeds the number of worker threads.

One of the key issues in dealing with synchronization constraints is what to do when a task's progress is hampered due to dependences. Most runtimes handle such operations by blocking the underlying worker thread and creating additional threads to execute other ready tasks and maintain application-level parallelism. If these additional threads are not created, an application can run out of threads to serve as executors for parallel tasks, even when there are available idle processors. On the other hand, creating additional threads leads to a scenario where there are more active threads than processors. The operating system usually employs some form of multithreading to context-switch threads and ensures fairness in each thread makes progress [51]. These preemptive switches can add to overheads while executing applications. We believe using a cooperative approach to handling context-switching of tasks due to synchronization constraints is a better approach.

In this work, we address the problem of efficient cooperative scheduling of parallel tasks with general synchronization patterns. Our solution is founded on the use of *one-shot delimited continuations* (OSDCs) and *single-assignment event-driven controls* (EDCs) to schedule tasks cooperatively in the presence of different synchronization patterns. The OSDC and EDC primitives can be used to support a wide range of *synchronization constructs* (SyncCons) including those where a task/event

may trigger the enablement of multiple suspended tasks. This general case is not supported by work-stealing schedulers for Cilk and other fork-join models for task parallelism. While efficient continuation-based scheduling is well established for fork-join parallelism in well-structured *tree-like* computations in projects such as Cilk and Manticore [52], we are unaware of any past work that supports more general (and a wide variety of) synchronization patterns in a scalable manner with support for large numbers of suspended tasks. To the best of our knowledge, our work is the first to support synchronization patterns that represent arbitrary computation graphs through the use of one-shot continuations.

Our cooperative approach of using OSDCs and EDCs demonstrates superior performance compared to schedulers that spawn additional worker threads to compensate for blocked worker threads (as well as approaches that leave worker threads blocked without spawning new worker threads). Transparent use of OSDCs allows us to leverage the benefits of event-driven programming while the user code remains in standard thread-based structure, thereby avoiding the need to write fragmented difficult to understand event-driven programs where logical units are broken down into multiple callbacks [53].

3.1 Contributions

The contributions of our work on cooperative scheduling are as follows [49, 54]:

- Use of OSDCs and EDCs to create a new generic cooperative runtime for task-parallel programs. We believe that any task-parallel SyncCon can be supported by this cooperative runtime. To the best of our knowledge, this is the first effort to systematically use OSDCs to support a task-parallel runtime. A key challenge we address in our runtime is that the resolution of a synchronization can, in general, trigger the enablement of multiple suspended tasks, a scenario that does not occur in traditional fork-join operations.

- We describe recipes for implementing different SyncCons using the API exposed by our cooperative runtime. These (and other) SyncCons are all treated uniformly by the runtime and can all be used together without issues in the same program.

- Implementations of our cooperative runtime for the HJ language and HJ library, both of which support a large variety of SyncCons. The cooperative runtime achieves portability by transparently supporting unmodified user code (written for older non-cooperative HJ runtimes). In addition, our implementation conforms to all the constraints imposed by a standard Java Virtual Machine.

- Empirical evaluation of the performance of our cooperative runtime relative to a runtime that uses thread-blocking operations. Our experiments on various benchmarks show that the cooperative runtime can achieve over 10× speed-up over a runtime that uses thread-blocking operations to implement SyncCons such as *futures* and *phasers*.

3.1.1 Outline

This chapter is organized as follows: Section 3.2 uses simple example programs to illustrate the performance issues with scheduling programs that use blocking SyncCons and the productivity issues with event-driven programming. Section 3.3 describes our cooperative runtime for scheduling task-parallel programs. We describe recipes for implementing different SyncCons using the API exposed by our cooperative runtime in Section 3.4. For the interested reader, 3.5 contains details on our implementation, including the use of an extended version of the open source Kilim bytecode weaver [55] to support OSDCs. Section 3.6 contains empirical evaluation of the performance of our cooperative runtime relative to a runtime that uses thread-blocking operations. Finally, Section 3.7 summarizes this chapter.

3.2 Motivating Examples

In popular task parallel runtimes such as those for HJ, X10, and Chapel, the runtime is usually able to handle synchronization points associated with fork and join operations without blocking the worker. However, other potential synchronization points (such as resolution of future results, point-to-point synchronization points, lock-based implementation of atomic regions) are blocking operations in the runtime. This may result in the worker threads being blocked, effectively resulting in fewer parallel threads that are executing. The runtime can compensate by creating additional worker threads, but this adds to overhead in the runtime as each thread needs its own system resources. In addition, context switching overhead is incurred when the blocked threads become unblocked.

On Intel processors it takes about 1100 ns per thread context switch (without cache effects) [56]. In contrast, object allocation, method call, and setting fields takes around 30 ns, 5 ns and 1 ns respectively [57] on the Common Language Runtime(CLR) - the virtual machine component of Microsoft's .NET framework. While the CLR manages the execution of .NET programs, timings for similar operations on the JVM should be similar since C# and Java programs have similar performance [58]. Our continuation creation scheme includes one object allocation, setting fields per live variable to be saved, and returning from method calls. Using continuations should cost less than 50 ns per method in the call chain. Besides, the compensation strategy of creating additional threads is contradictory to the goal of the TPM which relies on using comparatively few heavyweight threads to run many lightweight tasks. Finally, these current solutions do not scale as increasing the number of worker threads can eventually cause the runtime to crash due to exhaustion of memory or other system resources.

In addition to the problems mentioned above, presence of synchronization constraints can also lead to starvation situations when all available worker threads become blocked. In such scenarios, the program behavior can change when the parallel

```
1 public class CyclicProducers {
2   public static void main(final String[] args) {
3     // number of tasks to create
4     final int numTasks = 64;
5     finish {
6       final ItemHolder itemHolder = new ItemHolder(numTasks);
7       for (int i = 0; i < numTasks; i++) {
8         final int myId = i;
9         async {
10          // first produce an item
11          final int myProducedItem = produceItem(...);
12          itemHolder.put(myId, myProducedItem);
13          ...
14          // now consume item produced by neighbor
15          final int neighId = (myId + 1) % numTasks;
16          // wait until neighbor produces item
17          final Object itemToConsume = itemHolder.get(neighId);
18          consumeItem(myId, itemToConsume);
19 } } } } }
```

Figure 3.1 : An example that can lead to starvation when a thread-blocking runtime runs this program with too few worker threads.

program is run with different numbers of worker threads with the starvation scenario not occurring when enough worker threads are provided to compensate for the number of tasks involved in the synchronization constraint.

Consider an example program, in Figure 3.1, which spawns a number of tasks that form a *ring*. Each task is involved in a two-stage computation: in the first stage the task produces a value (in lines 11-12) and in the second stage the task consumes the value produced by its immediate right neighbor in the ring (in line 17-18). The synchronization constraint of having to wait for the neighbor to produce the item handled in the `get()` method of the `ItemHolder` data structure is not shown in the example, but one can imagine it being implemented by traditional locks in any of the languages mentioned above (HJ, Cilk, X10, etc.). Locks in these languages have blocking implementations and cause the tasks to block worker threads. Consider this program, which spawns 64 tasks, being run with 64 worker threads. In such a scenario, each of the spawned tasks would potentially be assigned to individual worker threads, and each task will have an opportunity to run and produce a value. As a result, all of the blocking calls to `get()` would eventually be satisfied and the

```
1 public class CyclicProducers {
2   public static void main(final String[] args) {
3     // number of tasks to create
4     final int numTasks = 8;
5     finish {
6       final promise<int>[] items = new promise<>[numTasks]...;
7       for (int i = 0; i < numTasks; i++) {
8         final int myId = i;
9         async {
10          // first produce an item
11          final int myProducedItem = produceItem(...);
12          items[myId].put(myProducedItem);
13          ...
14          // now consume item produced by neighbor
15          final int neighId = (myId + 1) % numTasks;
16          // trigger callback when neighbor produces item
17          asyncAwait(items[neighId]) {
18            final Object itemToConsume = items[neighId].nbGet();
19            consumeItem(myId, itemToConsume);
20 } } } } } }
```

Figure 3.2 : An event-driven version of Figure 3.1 where callbacks are used to avoid thread-blocking operations. A `promise` can be viewed as a container with a full/empty state that obeys a dynamic single-assignment rule. The `nbGet()` methods represents a non-blocking `get()` operation. The `nbGet()` can only be performed inside an `asyncAwait` block on any promise registered in its await clause (e.g. `items[neighId]` on line 17).

computation would complete. Instead, consider the program being run on a runtime with 32 worker threads. If the task scheduler schedules alternate tasks (i.e. tasks with id 0, 2, 4, ..., 62), each of them will produce their value and block in the call to `get()` since their neighbor has not been scheduled to run and never produces the value that these tasks want to consume. Since all available worker threads become blocked, no computational progress can be made, and we have a starvation! Instead, in a cooperative runtime no starvation would occur in this program irrespective of the number of worker threads used.

To avoid blocking, programmers can choose to write their code in an event-driven style with callback registrations. Figure 3.2 shows an event-driven version of Figure 3.1 where callbacks are used to avoid thread-blocking operations. In this version, the possible blocking calls to `get()` are replaced by a callback registration (at line 17-19) on the rest of the computation to run when the value from the neighbor is

```
1 public class FibCallback {
2   public static void fib (int n, promise<int> f) {
3   if ( n < 2) { f.put(n); return; }
4     promise<int> x = newPromise<>();
5     promise<int> y = newPromise<>();
6     async { fib(n-1, x); };
7     async { fib(n-2, y); };
8     asyncAwait(x, y) { f.put(x.nbGet() + y.nbGet()); }
9   }
10  public static void main (String[] args) {
11    int n = Integer.parseInt(args[0]);
12    promise<int> res = newPromise<>();
13    async { fib(n, res); };
14    asyncAwait(res) {
15      println(res.nbGet());
16 } } }
```

Figure 3.3 : Version of the Fibonacci numbers program that uses event-driven style with callbacks and asynchronous tasks. Asynchronous tasks are created with `async`; asynchronous callbacks are registered on promises using `asyncAwait`. The `fib()` method needs an extra parameter to store the promise and allow callback registrations. Calling `fib` does not return a result directly; rather an additional callback needs to be registered on line 14 to receive and display the result.

eventually produced. This version requires additional support from the language or runtime to allow callback registrations on the underlying primitive (e.g. `promise`) being used to implement the data structure. Though this version of the program is cumbersome to write, it will never display starvation irrespective of the task scheduler used. There is no starvation since there are no blocking operations, and worker threads can always be used to make computation progress.

As another example, Figure 3.3 shows the classic (and inefficient) parallel version of the Fibonacci function written in an event-driven style. This style of programming makes writing and maintaining code somewhat onerous and error-prone. A key difficulty is that the logical unit of work is broken across callbacks and methods are passed extra parameters to help registering on the callbacks. There is no direct return of a value from the callee to the caller. These make the code harder to read and maintain, especially as the method size grows and multiple parameters need to be passed along the call chain.

In contrast, Figure 2.4 shows an example program to compute Fibonacci numbers using `future`s to asynchronously compute values of the subproblems. This example is aligned with thread-based code where no extra parameters are required to register callbacks, and the function calls return values directly. The code for this version follows a more standard program structure and is easier to read and maintain compared to Figure 3.3. The `get()` operations are potential synchronization points where the task may suspend itself if the value of the `future` has not already been resolved. In many current runtimes, these potential synchronization points could result in thread blocking operations. In our runtime, we handle the synchronization points cooperatively using OSDCs without blocking the worker thread. This allows us to leverage the benefits of event-driven programming while the user code remains in standard thread-based structure (i.e. the user writes programs similar to Figure 3.1 and Figure 2.4). As we see in Section 3.6.2, the non-blocking version of Fibonacci with `future`s clearly outperforms a blocking version by a factor that exceeds 100×. Similar performance gains can also be achieved by cooperatively scheduling tasks with other SyncCons, such as `phaser`s (see Section 3.6.3).

3.3 Cooperative Runtime for Task Scheduling

The general TPM allows programmers to represent their computations as directed acyclic graphs with dependences between inter-dependent tasks. As a result, there has been plenty of work done in developing structured synchronization constructs (SyncCons) on the TPM by the community. These constructs include the well-structured `async-finish` variant of fork-join style tasks; point-to-point synchronization with `future`s; localized and group synchronization using `phaser`s; and weak atomicity in critical sections [3, 12, 13, 11]. Such constructs introduce new challenges for the runtime while scheduling and executing tasks.

Synchronization constraints can prevent a currently executing task from making further progress as it *waits* to synchronize with other ready but not executing task(s).

Many task parallel runtimes implement such waits by either busy-waiting until the constraint is resolved or by blocking the worker thread. An alternative approach is to use cooperative scheduling of tasks where an executing task, via runtime support, decides to suspend itself actively and yield control back to the runtime. The runtime can then perform bookkeeping on the suspended task and use the worker thread to execute other ready tasks. When the synchronization constraint that caused the task to suspend is resolved, the suspended task is resumed and scheduled for execution. This approach allows the runtime to continue making progress in the computation and to steadily exploit available parallelism during application execution without spawning additional threads. This nonblocking approach enables us to provide proper time guarantees since each worker is actively making some progress towards the computation. Supporting the time bound guarantee comes at a cost of space bound. Additional space is required since many tasks may be *in flight* (either suspended, ready or executing) with all the additional space for temporary local variables in the heap.

In the rest of the section, we present some background on one-shot OSDCs and EDCs. Then we describe the API we expose in our cooperative runtime to allow language/library developers implement a variety of SyncCons. Finally, we explain how OSDCs and EDCs are used to build a cooperative task-parallel runtime.

3.3.1 One-shot Delimited Continuations

Felleisen introduced Delimited Continuations (DeConts) in 1988 [59] where he referred to them as *prompts*. Other variants for DeConts include the `shift/reset` mechanism introduced by Danvy and Filinski [60]. Many mainstream languages offer support for various forms of DeConts, such as Boost coroutines in C++ [61], Kilim framework in Java [55], `shift/reset` in Scala [62], Ruby fibers [63], etc.

Continuations represent the rest of a computation from any given point. They refer to the ability to *capture* the state of a computation at that point, the computation can later continue execution from that point by *resuming* the continuation. In contrast,

```
1 class DeContPrimer extends DelimitedContinuation {
2   Boundary Override public void run() {
3     foo();
4   }
5   public void foo() {
6     println("foo: A");
7     bar(1);
8   }
9   public void bar(int x) {
10    int b = x + 1;
11    DelimCont.suspend("bar-" + x);
12    baz(b);
13  }
14  public void baz(int b) {
15    println("baz: B " + b);
16  }
17  public static void main (String[] args) {
18    DelimCont c = new DelimContPrimer();
19    do {
20      c.resume();
21      println(" cause = " + c.cause());
22    } while(!c.completed());
23 } }
```

Figure 3.4 : Example use of one-shot delimited continuations. `run()` is demarcated as the boundary method and it is invoked by calls to the resume() method. Suspending the delimited continuation involves in saving the call stack and local variables state, before returning to `resume()`.

DeConts represent the rest of the computation from a well-defined outer boundary, i.e. a subcomputation. This boundary allows DeConts to return to their caller allowing the program to proceed at the call site. DeConts are hence a good choice when a limited part of the computation needs to be saved/restored [64]. In general, a continuation can be resumed multiple times from the same captured state; however one-shot continuations refer to continuations that are resumed at most once. This guarantee makes them cheaper to implement because they do not require making additional copies of the state.

Figure 3.4 demonstrates an example use of OSDCs where the `run()` method at line 2 defines the boundary of the OSDC. After creation of the OSDC at line 18, we call the `resume()` method to start executing the body of the continuation from the boundary method, i.e. the `run()`. During the computation the OSDC can be suspended by calling `suspend()` and providing an intermediate cause for the suspension, e.g. the

string `"bar-1"` at line 11. As the OSDC is suspending, the state of local variables and program counters in the call chain are saved and control is returned to the caller of the boundary function, i.e. `resume()` at line 20. Subsequent calls to `resume()` on the OSDC causes the computation to resume from the last suspension point, e.g. the call to `resume()` at line 20 in the second iteration of the loop causes the control to return to line 12 which follows the `suspend()` call. The state of a OSDC can be queried to check whether it is currently suspended (e.g. `completed()` and if so, intermediate return values can be retrieved (e.g. by calling `cause()` at line 21). When the OSDC() eventually returns without suspending, the OSDC executes to completion (`completed()` returns `true`) and has no cause (i.e. `null`). This makes the condition in the do-while loop to become false and exit the loop. Any subsequent calls to `resume()` on a completed OSDC will result in an error. Running the program from Figure 3.4 produces the following output:

```
1 foo: A
2  cause = bar-1
3 baz: B 2
4  cause = null
```

3.3.2 Event-Driven Controls in the Cooperative Runtime

`EventDrivenControl`s (EDCs) have already been introduced in Section 2.4.1. The main differences in the API, from the one presented in Section 2.4.1, are as follows:

- The static `suspend(anEdcInstance)` method signals the possible creation of a suspension point. If the EDC passed as an argument has not been resolved, the current task is suspended, and the runtime handles the bookkeeping to register an EB to resume the task when the EDC is resolved.

- The `setValue(someValue)` method resolves the EDC, i.e. it binds a value with the EDC and triggers the execution of any EB registered with the EDC. Suspended tasks registered with the EDC will be resumed and scheduled for execution by the runtime.

```
1 class Future<T> {
2   EventDrivenControl<T> edc = EventDrivenControl.newEDC<T>();
3   public void put(T item) {
4     edc.setValue(item); // resumes consumer(s)
5   }
6   public T get() {
7     // suspend consumer task till value produced
8     EventDrivenControl.suspend(edc);
9     // return value after it is resolved
10    return edc.getValue();
11 } }
```

Figure 3.5 : `futures` implemented using the EDC API provided by the cooperative runtime. All consumer tasks suspend until the item is produced. Once the item is available, multiple suspended consumers are resumed by the runtime.

Library/language developers can create their custom SyncCons in our runtime using the EDC API. The OSDCs created to manage the bookkeeping are not exposed to the developer. This hiding is especially desirable since continuations are notorious for being hard to use and to understand by developers (as opposed to compilers and runtime systems). With these operations in place, language/library developers can implement their custom SyncCons and synchronization patterns. The same API is used in our implementation of Habanero-Java to support the constructs such as the end of `finish`, `futures`, `phasers`, etc. For example, Figure 3.5 shows how simple it is to implement `futures` using the exposed API. A single EDC is used to suspend all consumers who try to read the value of the `future` before it has been resolved. When the value of the `future` is available, the EDC is resolved with a call to `setValue()` and any suspended consumer tasks are resumed by the runtime.

3.3.3 The Cooperative Runtime

In our cooperative runtime, when a potential synchronization point is discovered dynamically, thread blocking operations are avoided by suspending the currently executing task and cooperatively scheduling other ready tasks from the work queue. When the EDC is resolved, the suspended task (and its continuation) is put back into the work queue to be eventually resumed by a worker thread. Task suspensions

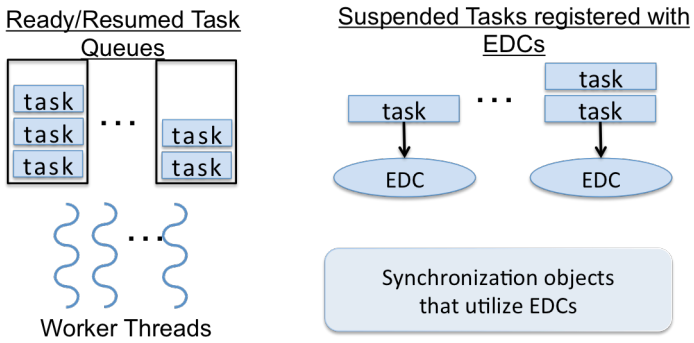

Figure 3.6 : The cooperative runtime includes worker threads and ready task queues like most other task parallel runtimes. In addition, there are EDCs that maintain a list of suspended tasks to implement higher-level synchronization constructs. Resolving an EDC moves a suspended task into the ready queue.

are implemented by using standard OSDCs, and this guarantees that the runtime never spawns more worker threads than it was initially started with. The trade-off is that the compiler and the runtime now need to support the overhead of creating the OSDCs and handling the management of the EDCs in addition to the management of threads and tasks.

A pictorial summary of our runtime is provided in Figure 3.6. The runtime cooperatively schedules tasks using OSDCs and EDCs in the presence of arbitrary dependences or synchronization constraints. The runtime places tasks into queues while the pool of worker threads continuously attempt to execute tasks dequeued from these queues. Execution of tasks may result in more tasks being spawned and enqueued into the queues. An application starts with a single *main* task in the work queue which promptly gets executed by one of the worker threads. The application terminates when *a*) the work queues are empty, and *b*) all synchronization constraints in the program have been satisfied (i.e. no deadlocks).

The work-first policy can be more efficient than help-first for recursive divide-and-conquer parallelism when steals are infrequent [48], the work-first policy cannot be used to support general synchronization constraints. Instead, our runtime uses a

help-first policy [48] while scheduling tasks. Under this policy, spawning a child task enqueues it is in the task queue and allows the parent task to continue execution past the spawn operation. The child task hence has a stack of its own and can be executed by any of the worker threads. The independent stack allows us to treat the task as a subcomputation and to have a well-defined outer boundary while forming the OSDC. In contrast, using a *work-first* policy [48] does not provide an independent call-stack for a spawned task and requires maintaining fragmented call-stacks to allow helper threads to resume computations. This precludes the use of OSDCs in a work-first policy (though the work-first policy can be more efficient than help-first for recursive divide-and-conquer parallelism when steals are infrequent, the work-first policy cannot be used to support general SyncCon). In addition, constructs such as `phaser`s are not amenable to work-first scheduling since these constructs do not satisfy the "serial elision" property.

With the help-first policy in effect, we wrap the stack of each task around an OSDC which defines an `execute()` method as the continuation boundary. When a worker thread executes a task, it resumes the computation of the OSDC which in turn invokes the `execute()` method, as shown in Figure 3.7. At synchronization points where a task is not allowed to make progress semantically, an OSDC is captured, and only the state until the `execute()` method needs to be saved. On returning from a call to `execute()`, the runtime verifies the cause for the return and performs bookkeeping if the task was suspended. The worker thread then goes ahead and tries to dequeue other scheduled tasks to execute and continue making progress towards the overall computation.

In our runtime, the static `suspend` method of the API (Section 3.3.2) restricts the cause of OSDC suspensions to instances of EDCs. After returning to the runtime thread's `worker.executeTask()` method, the runtime checks whether an EDC was returned as a cause (i.e. the task was suspended) and registers an EB with that EDC. There is no limit to the number of tasks that can be registered to an EDC (in the form

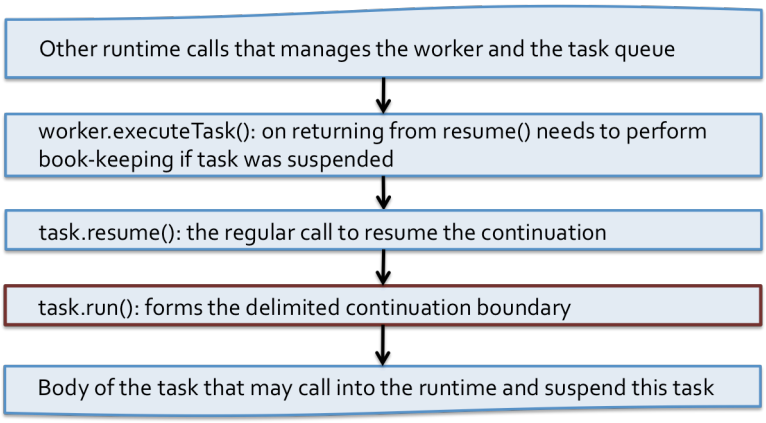

Figure 3.7 : Representation of the runtime call stack when a task is being executed by a worker thread. The `worker.executeTask()` method is responsible for managing the OSDCs that may be suspended while executing the body of a task.

of an EB). When the EDC is resolved, the EBs are executed, and the suspended tasks are rescheduled. Note that this approach does not need to use polling to keep track of when suspended tasks can be resumed. After being scheduled, the queued task is picked up by a worker thread and execution is resumed from the previous suspension point. When the execution of the task completes normally, without suspending, the runtime performs any cleanup operations associated with the task and looks for more work from the queue.

The remaining pieces in the runtime are the steps to undertake where synchronization points: *a*) capture continuations, *b*) create EDCs, and *c*) resolve EDCs. The use of OSDCs and EDCs are abstracted by the implementer of the SyncCons and transparent to an end user of these constructs. We discuss how various SyncCons can be developed in our runtime in Section 3.4.

Work-Stealing Scheduler

The exact policy to retrieve tasks from work queues is unspecified in our cooperative runtime. Recent work has shown that work-stealing policies work very well on multicore architectures. A scheduler using a work-stealing policy maintains a queue of

pending tasks per worker thread. When a worker completes a task, it pops a pending task from its owned queue. If the queue is empty, it attempts to steal a task from another worker's queue. Our runtime uses the help-first policy and maintains an independent stack for each task, the OSDC created is thread independent and can be run by any thread. Hence, any worker thread may execute a task, and we can use both the work-stealing or work-sharing scheduling policies in our runtime.

Serializability of Computations

Serializability of a group of parallel or concurrent statements refers to the ability to provide a serial ordering of the statements. In our runtime, since a single worker thread can execute the entire computation, that schedule provides a serializable order for the statements. The caveat is that the granularity of the statement blocks is around suspension points rather than user-written tasks. These new statement blocks can be used to form structures to represent the program dependence graph of the computation and reason about parallel portions and simplify, for example, data race analysis. With additional support from a scheduler, the statement blocks from the dependence graph can be scheduled in a deterministic order if so desired or can be used to generate different schedules. Both these capabilities can be very useful for debugging programs.

Preemptive Scheduling

Our runtime API can easily support preemptive scheduling by simulating timed preemption. This requires some additional support from a preprocessor which instruments the tasks and inserts artificial suspension points (similar to "yield points" used by garbage collectors) to track and suspend long running tasks when they exhaust their allocated computation time. Our runtime can also support the building of Engines [65], an abstraction for timed preemption where a task runs for a specified amount of time before suspending to allow other tasks to run. Engines can help in

fairness by varying the granularity of time units a task is allowed to run before it suspends itself. Engines also enable us to support speculative parallelization, e.g. in the form of Cilk's [3] abort statement, more efficiently. This allows executing tasks to be terminated prior to task completion if it is discovered that results from those subcomputations are unnecessary. Previous work by Feeley on *balanced polling* [66] and in the Jikes VM *yieldpoints* [67] provide a scheme to automatically insert such suspension points or engine consumption checks while minimizing overhead.

3.4 Support for Synchronization Constructs

Synchronization constructs (SyncCons) are used to coordinate the parallel execution of tasks. In this section, we describe how various SyncCons can be supported by our cooperative runtime. The key idea is to translate the coordination constraints into producer-consumer constraints on EDCs and to use OSDCs to suspend consumers when waiting on item(s) from producer(s). We claim that any task-parallel SyncCon can be translated in such a manner and hence be supported by our runtime. The constructs we present include: *a*) termination detection of child tasks, *b*) producer-consumer synchronization, *c*) collective barrier synchronization, *d*) single blocks executed by only one task in a group, and *e*) weak isolation while accessing a shared resource. While constructs a) through d) are typically used for deterministic parallelism, construct e) can be used to support nondeterminism as well.

3.4.1 Fork-Join Synchronization

In structured fork-join parallelism, a *parent* task can spawn one or more *child* tasks that can logically run in parallel with the parent task. The parent task can then wait, by joining, until all of its transitively spawned children complete execution. An EDC, which wraps a counter*, is created for each parent task. The counter is

*Distributed counters can be used for increased scalability.

atomically incremented each time a child task is forked and atomically decremented as each child task completes execution, either normally or abnormally. When the count reaches zero, the value of the EDC is resolved. The join operation serves as a possible suspension point in our runtime and uses the EDC as its cause for suspending if it is invoked before the count reaches zero. If the count is zero when the join operation is called, execution of the parent task continues without the need for suspension. This model can be easily extended to also support nested fork-join parallelism.

3.4.2 Producer-Consumer Synchronization

In producer-consumer patterns, producer tasks are responsible for resolving the values inside EDCs while consumer tasks suspend until the value inside an EDC has been resolved. A typical case is the single-producer multiple-consumer case, also known as *futures* [12]. A `future` represents an immutable value, an EDC in our runtime, which will become available at a later point by a producer task. When the producer task completes execution, it resolves the value inside the EDC thus resuming any previously suspended consumers. Consumers who read the value of the `future` after it has already been resolved can continue execution without being suspended. The single-producer single-consumer case can be supported by further wrapping an EDC and ensuring that only one consumer can read the value of the EDC, read requests from other consumers report an error.

The general producer-consumer problem with a mutable buffer location can also be modeled using our API. An example of such a construct is the synchronization variable construct available in Chapel [8]. In effect, the buffer location is either empty or full, and producers/consumers need to wait when the location is full/empty, respectively. This can be modeled in our runtime by maintaining a doubly-linked list of a pair of EDCs and a pointer to the active pair. The first element in the pair represents whether a producer has produced the item making the location full, while the second

```
1 class EdcPair<T> {
2   EventDrivenControl<T> p = EventDrivenControl.newEDC<>();
3   EventDrivenControl<Boolean> c = EventDrivenControl.newEDC<>();
4 }
5 class SynchronizationVariable<T> {
6   Node<EdcPair<T>> pNode; // producer chain
7   Node<EdcPair<T>> cNode; // consumer chain
8   Node<EdcPair<T>> nextNode(Node<EdcPair<T>> n) {
9     if (n.nextNode == null)
10      n.nextNode = new Node<>(n, ...);
11    return n.nextNode;
12  }
13  public SynchronizationVariable() {
14    Node<...> item = new Node<>(null, ...);
15    item.c.setValue(true);
16    cNode = pNode = nextNode(item);
17  }
18  public void write(T item) { /*suspendable method*/
19    Node<EdcPair<T>> n;
20    isolated { n = pNode; pNode = nextNode(n); }
21    EventDrivenControl.suspend(n.prevNode.c);
22    n.p.setValue(item);
23  }
24  public T read() { /*suspendable method*/
25    Node<EdcPair<T>> n;
26    isolated { n = cNode; cNode = nextNode(n); }
27    EventDrivenControl.suspend(n.p);
28    n.c.setValue(true);
29    return n.p.getValue();
30 } }
```

Figure 3.8 : Synchronization variables implemented using operations provided in the cooperative runtime. Producers suspend until the previous item is consumed; consumers suspend until the current item is produced.

element represents whether a consumer has consumed the item making the location empty. A producer suspends until the previous consumer-EDC has been resolved while a consumer suspends until the producer-EDC in the currently active pair has been resolved. Separate producer and consumer pointers are maintained, and they are advanced to the next node in the list when write and read operations are invoked, respectively.

An example implementation of the synchronization variables recipe using the cooperative API is provided in Figure 3.8. There are two pointers being maintained to track the progresses made by producers and consumers. A pair of EDCs is maintained to ensure there is the strict alternation of writes and reads by producers and consumers, respectively while accessing the synchronization variable. If a producer

arrives before the previous value has been read by a consumer it is suspended and vice versa.

3.4.3 Collective Barrier Synchronization

A barrier synchronization provides a means to ensure tasks in a group have all arrived at a particular point before advancing. This synchronization is especially useful in phased computations by ensuring each task in the group of tasks has completed one phase before starting the next phase of the computation. It is possible that the group of tasks involved in the barrier remain static or change dynamically over time, either form of barriers can be supported by our API/runtime. Implementing barriers in a runtime that uses thread-blocking operations is not scalable if the number of tasks registered on the barrier exceeds the number of available worker threads. The blocking can lead to starvation or deadlocks if the runtime is not allowed to create additional worker threads to allow all tasks to reach the barrier and release the blocked threads. In the case where the runtime can compensate by creating additional worker threads, scalability and efficiency are affected due to the overhead of having to manage additional worker threads. In our cooperative runtime, there are no thread blocking operations. The tasks can suspend themselves if they arrive too early at a barrier allowing the worker threads to execute other ready tasks and reach the barrier point. Eventually, all tasks will arrive at the barrier, and the suspended tasks will be resumed.

To support barriers with dynamic task registration (the static task version is a special case), we maintain a count of registered tasks, a count of arrived tasks, and an EDC for each phase in the barrier computation as shown in Figure 3.9. When a task dynamically registers on the barrier, it registers on the *next* phase and increments the count of registered tasks for that phase. However, a task always deregisters in the current phase of the barrier and increments the arrived task count. As each task arrives at the barrier, it increments the count for arrived tasks in the current phase

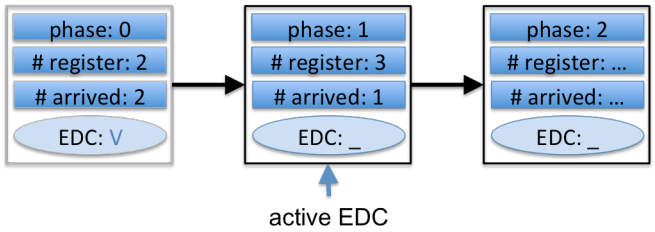

Figure 3.9 : The barrier represents each phase with two counters to keep track of registered and arrived tasks and an EDC, which is used to track *early* arrivers. As each phase completes, the EDC in the current phase is resolved resuming suspended tasks and the active phase pointer is moved to the next item in the linked list.

and the count for registered tasks in the next phase. Additionally, if the task is not the last to arrive at the barrier point it suspends itself using the EDC for the current phase as the cause. The last task to arrive at the current phase of the barrier resolves the EDC of the current phase, advances the phase of the barrier and continues without suspending. Resolving the EDC resumes all the tasks suspended on the barrier and the tasks now participate in the next phase of the computation when executed.

3.4.4 `phaser` Synchronization

An extension to barrier synchronization is provided by `phasers` [13]. They unify collective and point-to-point synchronization for phased computations. Unlike traditional barriers where tasks register in *signal-and-wait* mode, tasks can also be registered on a `phaser` in *signal-only* or *wait-only* modes. Tasks registered on a wait mode (wait-only or signal-and-wait) need to wait for all tasks registered on a signal mode to arrive at the barrier point. The implementation for barriers (Section 3.4.3) needs to be extended by allowing only signalers to increment the counts of their local phase. Since signaler tasks can be in different phases, care is required to ensure that the correct counters are incremented. Tasks registered in signal-only mode never suspend and continue to make progress. Tasks registered in wait mode need to suspend themselves and wait for the EDC for a given phase to be resolved when all signaler

tasks for a given phase arrive at the barrier point. As the EDC for the *oldest* phase is resolved by the last signaler task, it also advances the current phase for use by the waiter tasks.

3.4.5 Single Blocks

The OpenMP `single` construct specifies that a statement block is executed by only one task among a group of registered tasks [5]. The *wait* version requires that all registered tasks wait until some task has executed the `single` block. This is similar to supporting barriers with a single phase. All tasks, except the last task, that arrive at the `single` suspend themselves. The last task that resolves the EDC executes the statement block before resolving the EDC and causing the tasks registered on the `single` to be resumed. The *nowait* version does not require to suspend tasks, it requires some bookkeeping to ensure that exactly one task to arrive at the `single` executes the statement block.

`phaser`s also support a variant of `single` blocks when tasks are registered using the *signal-wait-single* mode. The semantics defines that the single block is executed only after all the signalers and waiters have arrived at the `single` block. Both the signalers and the waiters need to ensure they proceed only after all signalers have arrived and at least one task has executed the `single` block. Supporting such blocks in our runtime requires the use of two EDCs for each phase of the `phaser`. The first EDC keeps track of whether all signalers have arrived while the second EDC is used to track whether the statements inside the `single` block has been executed by some task. Thus, tasks can be suspended twice while executing a single block.

3.4.6 Weak Isolation

Habanero-Java provides the `isolated` SyncCon, which can be used to implement critical sections and coordinate the mutation of shared data. The weak isolation guarantee states that the statements inside the critical sections will be executed mu-

tually exclusively with respect to other demarcated critical sections (DCS). In general, weak isolation enforces a serializability bottleneck as only one critical section may be executed by the runtime in the absence of a more sophisticated analysis. Often this serializability is implemented using locks where worker threads block while waiting to attain the lock. Use of locks can limit performance in scenarios where there is moderate or high contention for the lock by the interfering DCS. In the cooperative runtime, blocking of threads while using locks is avoided by maintaining a dynamic linked-list of EDCs. Each task executing the DCS registers itself to an EDC in the list and suspends itself if it does not link to a resolved EDC. The first EDC in the list already resolved by default to allow the first requester of the lock to make progress in its DCS without suspending. Any task linked to a resolved EDC gets to execute its DCS and resolves the next EDC in the list.

3.5 Implementation

We have implemented our cooperative runtime in the both the Habanero-Java language [11] and the Habanero-Java library. Both implementations support all Habanero programming model constructs (Section 2.3) without any changes to the syntax of the input programs. We describe the HJ language implementation in Section 3.5.2 and the HJlib implementation in Section 3.5.3. Our implementations conform to the constraints imposed by a standard Java Virtual Machine (JVM). In particular, standard JVMs do not provide support for continuations or for storing away and restoring the stack. Drago et al. [64] extend a custom VM to support continuations. However, the delimited and one-shot continuations they support are not thread-independent, i.e. the continuations cannot be safely resumed on any thread. Their implementation reports an error if a thread resumes a continuation captured by another thread. Instead, as we will see in Section 3.5.1, we choose a solution where OSDCs are thread-independent.

3.5.1 Delimited Continuations - Kilim Weaver

As mentioned earlier, standard JVMs do not provide native support for continuations. The alternative to support OSDCs on a standard JVM is to transform the bytecode in class files. We use a slightly modified version of the open source bytecode weaver provided by the Kilim framework [55] to support OSDCs. Our extension to Kilim allows for OSDCs to be captured and resumed on custom threads (e.g. provided by the HJ runtime) rather than on threads managed by the Kilim runtime.

The Kilim weaver works by transforming code of methods which can suspend. It recognizes such methods by the presence of a `SuspendableException` exception in the method signature. It is important to note that no actual exceptions are thrown or caught to minimize the overhead of capturing and resuming continuations. Instead, the transformation performed is similar to a continuation passing style transform, except that the weaver transforms only methods that can suspend.

While transforming suspendable methods, the weaver injects code in all suspendable methods to provide each method with an extra argument of type *fiber*. The fiber is a data structure used to save the state of the stack including local variables while capturing a continuation [68]. While resuming the continuation, the call stack is restored using the control flow information stored in the fiber. However, we do use the weaver's support for the handling of `try-finally` blocks. The weaver also includes other optimizations such as lazily restoring the continuation frames in the call chain, custom state objects to store local variables, and avoiding store of dead local variables.

One of the limitations of using Kilim is that calling suspendable methods within constructors is disallowed. This not a major restriction as the user code can be refactored into factory methods that pass fully initialized arguments without the need for suspension to constructors. The weaver also does not handle the capturing continuations running inside `synchronized` blocks. This limitation does not concern us because `synchronized` blocks are neither used in our runtime nor are they en-

couraged in HJ or HJlib programs. In fact, the HJ compiler will report an error if `synchronized` blocks are present in HJ code. Finally, suspending and resuming deep call stacks in Kilim-based OSDC is expensive and can hurt performance.

3.5.2 Habanero-Java Language Implementation

In HJ, language constructs are supported by a compiler and a runtime system. The compiler generates standard Java class files that run on any standard JVM. Figure 3.10 highlights the different stages in the HJ compiler. The frontend parses the HJ program and constructs AST nodes to represent the different HJ constructs. The frontend also performs semantic checks for the constructs. The backend then gradually applies a series of transformations, all HJ constructs are later expanded into standard Java operations with calls to the HJ runtime library inserted as needed. For instance, an async is transformed into an instance of an anonymous inner class (closure) that can be passed to the HJ task scheduler. Further details on the general compiler architecture are available at [11]. The runtime is responsible for managing the creation, execution, and termination of tasks using a variety of task scheduling policies. The decision of which policy to use must be made at compile time since the generated bytecode is tailored to the appropriate scheduler.

Our implementation for the cooperative scheduling policy adds two additional stages in the backend. These stages translate calls to *suspendable* methods (Section 3.5.2) and perform bytecode transformation (Section 3.5.1), as shown in Figure 3.10. The cooperative runtime system is responsible for scheduling the tasks on a (user configurable) fixed number of worker threads. In our implementation, we extend the compiler to generate code that detects suspension points and generates code for the tasks. We rely on the help-first policy to have independent stack frames for tasks. As a result, child tasks do not share the call stack with their parents, and this simplifies our use of OSDCs. These continuations are resumed exactly once which allows us to have an efficient implementation compared to general continuations.

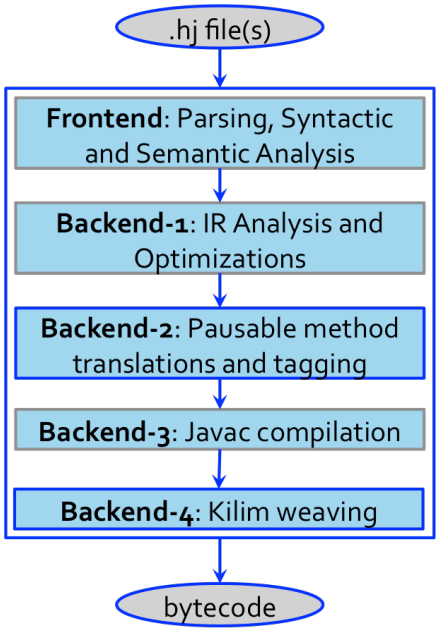

Figure 3.10 : Architecture of the HJ language cooperative compiler.

Tagging Suspendable Methods

The Kilim weaver relies on the presence of the `SuspendableException` exception in the list of thrown exceptions of a method declaration to identify which methods to transform. In HJ, users do not annotate their methods with such an exception in their code. The HJ compiler is responsible for tagging possible suspendable methods and creating the continuations are created transparently. Hence, the HJ compiler needs to insert the `SuspendableException` exception into the declarations of user-written methods that may suspend. The HJ compiler first translates the user-written code, which uses various synchronization constructs, into calls to suspendable methods provided by the cooperative runtime. After this transformation, the code in Java class files is no longer valid as there are method bodies in user-written code which throw the checked `SuspendableException` exception without declaring it in the throws clause of the method.

Next, the compiler detects calls to suspendable methods in the user code and incrementally annotates all the user methods with the `SuspendableException` ex-

ception. The algorithm for tagging suspendable methods does a reachability analysis for throwing the `SuspendableException` exception up the call chain and is as follows:

- Overridden methods of classes provided by the runtime or standard HJ/Java API do not change their suspendable behavior. They preserve their defined signature to avoid the weaver having to transform possibly a large number of API classes.
- Methods are suspendable if they invoke other suspendable methods. This tagging can recursively cause the signatures in user-written parent classes to change and become suspendable.
- Similarly, interface methods in user-written classes are suspendable if any of the implementing classes have method bodies invoking other suspendable methods.
- Constructors and undetermined static methods are not suspendable.
- All remaining methods are conservatively tagged as suspendable.

This algorithm terminates when it has successfully tagged (either as suspendable or as not suspendable) all user-written methods or if an error occurs. If during the tagging process there is a conflict, the HJ compiler fails and returns the appropriate error message. An example of such a conflict is the implementation of a non-suspendable interface method from the HJ API in the user-written code contains a call to a suspendable method. Once the annotation process completes without error, the weaver can be run to perform the transformation required to support continuations. After successful compilation, the generated bytecode can be run on any standard JVM by including the HJ runtime in the classpath.

3.5.3 Habanero-Java Library Implementation

Using the library-based approach allows us to rely on any standard Java compiler to perform parsing and generate the bytecode. Hence, the runtime and user programs

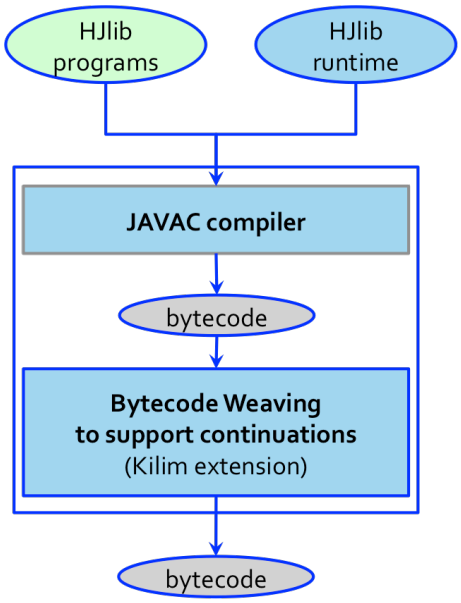

Figure 3.11 : Architecture of the HJlib cooperative runtime and compiler.

can be written in standard Java syntax following the Java language rules. Figure 3.11 highlights the different stages an HJlib program goes through during the compilation process. Implementation in the Habanero-Java Library consists of two parts: the runtime and the bytecode weaver during compilation.

The HJlib cooperative runtime is implemented purely in Java and extends the thread-blocking runtime presented in Section 2.4. The runtime has been reimplemented using the Kilim extension (Section 3.5.1) to support delimited continuations (DeCont). Each task maintains a one-to-one mapping with a DeCont instance and ensures the user-code of an asynchronous task executes inside a DeCont. The only locations that suspend the DeCont are calls to the static `EventDrivenControl.suspend()` methods. Since all SyncCons are implemented using EDCs, the cause of OSDC suspensions are restricted to instances of EDCs. When the DeCont is suspended, control returns to the runtime on a worker thread by unrolling the call stack.

The runtime then performs cleanup to release any resources held by the task but does not perform the bookkeeping to mark the task as terminated. The runtime then

queries the state of the DeCont to find the EDC that caused the suspension and registers a callback on the EDC. The worker thread then retrieves another available task from the ready queue, performing a steal if necessary, and starts executing that task. The callback registered on the EDC adds the task back into the ready queue when the EDC gets resolved. Once added to the work queue, some worker thread will retrieve the task and begin executing it. At that point, the DeCont inside the task is resumed, and control jumps to the point immediately past where the code was previously suspended. The execution of the code is thus guaranteed to make forward progress once resumed. Eventually, the DeCont executes to completion (without suspending) and control returns to the runtime with a completely executed DeCont (i.e. with a `null` cause). At this point, the runtime performs the bookkeeping to mark the task as terminated and the worker thread attempts to execute another task from the ready queue.

The HJlib API already includes a marker `SuspendableException` to mark method calls that can suspend. Our extended Kilim weaver ensures that this exception is never explicitly caught in the user code and is always explicitly rethrown in suspendable method signatures. The weaver allows the DeCont implementation as the only `class` that explicitly catches this exception. The presence of this exception in method signatures obviates the need for tagging suspendable methods stage required in the HJ language implementation (Section 3.5.2). During compilation of the user code, the `javac` compiler is run to generate bytecode that is then passed to our weaver. The Kilim weaver then transforms the input bytecode using the process described in Section 3.5.1. If any violations are discovered (e.g. using suspendable methods in constructors), the weaver stops the compilation process and reports the errors and fails. Otherwise, the weaver completes successfully, and the generated bytecode can be run on any standard JVM by including the HJlib runtime in the classpath. One optimization the compiler automatically performs is to reduce the overhead by avoiding bytecode transforms of tasks that are statically known to be nonblocking as

they do not throw the marker exception.

3.6 Experimental Results

The performance comparison result for the HJ language cooperative runtime have been presented in [49]. In this section, we focus on the performance results of our cooperative runtime in HJlib. The benchmarks were run on individual nodes in an IBM POWER7 compute cluster. Each node contains 256GB of RAM and four eight-core IBM POWER7 processors running at 3.8GHz each. Each core has a 32 KB L1 cache and a 256 KB L2 cache. The software stack includes IBM Java SDK 1.8.0 (build pxp6480 – 20150129_02) and HJlib version 0.1.5. The JVM configuration flags used were (`-XX:-UseGCOverheadLimit -Xmx65536m -XX:+UseParallelGC -XX:+UseParallelOldGC`). Each benchmark was configured to run using 32 worker threads and run for thirty iterations in six separate JVM invocations. The arithmetic mean of the best fifty execution times (from the hundred and eighty iterations) are reported. The execution times were computed using the `System.nanoTime()` method for the best resolution. Using the best execution time allows us to minimize the effects of JVM warm up, just-in-time compilation, and garbage collection.

We implemented our cooperative runtime in HJlib supporting all its available constructs without requiring any changes to the legacy HJlib programs. Hence, users are unaware of the use of EDCs and one-shot OSDeConts by the runtime. All benchmarks in our experimental evaluations use the same algorithm in their implementation. We focus on benchmarks for the different SyncCons to compare the performance of our implementation of the cooperative runtime with *a*) the `ForkJoinPool` and helper classes from the `java.util.concurrent` package in Java like `AtomicInteger`, `CyclicPhaser`, `CountDownLatch`, etc., and *b*) the existing work-stealing thread-blocking runtime available in HJlib. Both the variants we compare against have *blocking* implementations for most of the SyncCons.

Both these blocking runtimes have been shown to deliver performance competitive

with other runtimes (e.g., OpenMP, X10), for general SyncCons. All implementations were configured to use the same work-stealing scheduler (`ForkJoinPool` from the standard JDK) as previous experience has shown the scheduling policy to be more effective than a work-sharing policy. All the benchmarks were run using **thirty-two** worker threads as the starting seed. In the blocking runtimes, additional threads are created around blocking suspension points while the cooperative runtime never creates more worker threads. Management of and context-switching between threads adds overheads during the execution of the benchmarks in the thread-blocking runtimes compared to the relatively lightweight context-switching of tasks in the cooperative runtime.

3.6.1 Fork/Join Benchmarks

The benchmarks used are recursive divide-and-conquer style applications which suit the `async-finish` style of programming. Parallel tasks are spawned recursively until a certain task granularity is reached after which the computation is performed sequentially. The benchmarks used are as follows. The first benchmark is a recursive BitonicSort implementation inspired from [69] where the *split* and *merge* operations are performed in parallel. Bitonic sort runs in O($Nlog^2N$) time, but parallel implementations of the sort can exhibit dramatic speed-ups. The second benchmark is the FFT microbenchmark was ported to HJlib from a Cilk version published by Alex Iliev [70]. This program uses multiple `async` and `finish` constructs over various methods calls. The copious use of `finish` blocks means there are ample end-of-finish suspension points. The third benchmark is Integrate benchmark [71] obtained from the X10 benchmarks source repository and ported to HJlib. The fourth benchmark is a Matrix Multiplication algorithm which uses a four-way recursive split to multiply sub-matrices. The fifth benchmark uses a recursive MergeSort algorithm where the splits are parallelized but the merge operation is sequential. The final benchmark is the Strassen (BOTS benchmark [72]) benchmark.

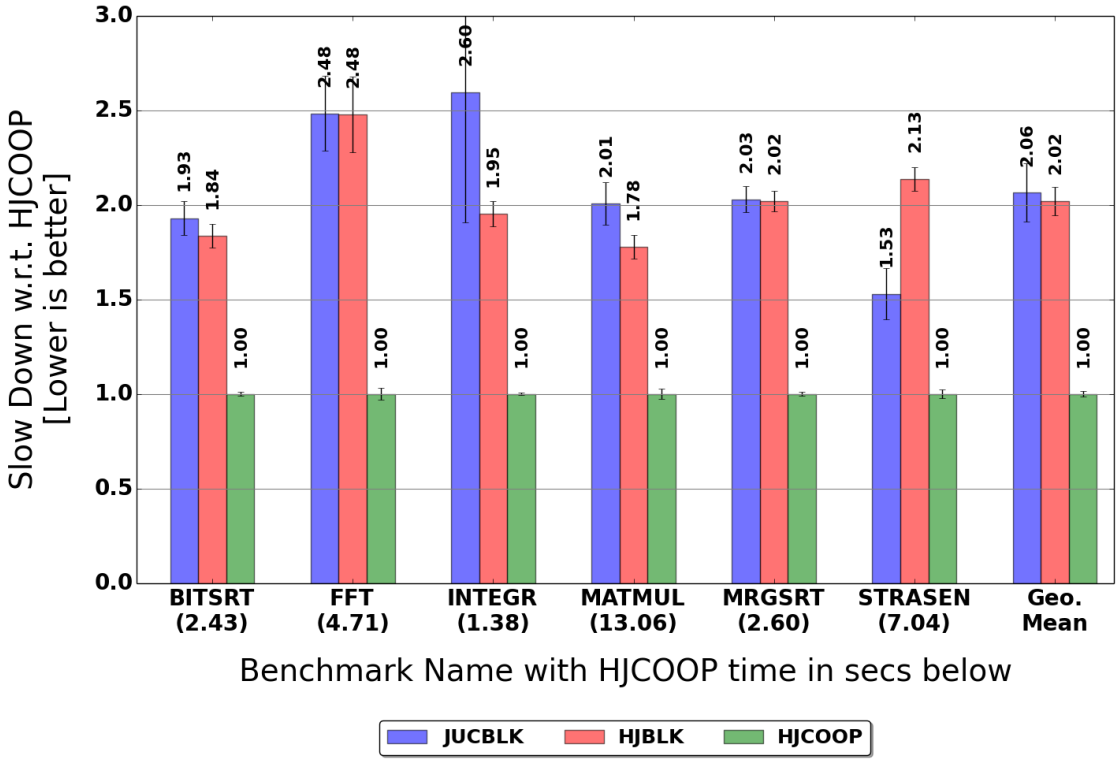

Figure 3.12 : Results for `async-finish` benchmarks. Bitonic Sort (BITSRT) on an input of size 40 million and sequential cutoff of 2 million. Fast Fourier Transform (FFT) on input array of length 2^{24} with a sequential cutoff of 2^{18}. Integrate (INTEGR) on the range $(0, 10,000)$ and sequential cutoff after a recursive depth of 10. Matrix Multiplication (MATMUL) on a matrix of size $5,120 \times 5,120$ and block size of 512×512. Merge Sort (MRGSRT) on an input of size 160 million and sequential cutoff of 1 million. Strassen (STRASEN) on a matrix of size $4,096 \times 4,096$ and sequential cutoff of 5.

Figure 3.12 shows the result of fork-join benchmarks using the `async-finish` style constructs. All these benchmarks have been parallelized by inserting `async-finish` statements into the serial versions of the benchmarks. In each of these benchmarks, the cooperative version (HJCOOP) performs significantly better than the thread-blocking versions (HJBLK and JUCBLK). On average, the cooperative runtime displays over $2\times$ speed-up over both HJBLK and JUCBLK. This speed-up is despite the cooperative runtime encountering the overhead of wrapping tasks in a OSDC

and checking whether the task suspended every time it is executed. The HJBLK and JUCBLK runtimes implement their `finish` constructs using atomic integers and latches which have thread-blocking semantics. The negative effects of blocking threads are shadowed to some extent since the HJBLK and JUCBLK runtimes compensate for blocked threads by spawning additional worker threads. On larger inputs (not shown in the chart), this strategy of allocating additional worker threads causes the HJBLK and JUCBLK runtimes to throw `OutOfMemoryError`s. The cooperative runtime, however, can avoid such overheads and complete the execution on a fixed number of threads.

3.6.2 `future` Benchmarks

We use six benchmarks to compare the performance of the `future` implementation of the different runtimes. Most of the benchmarks used are also recursive divide-and-conquer style applications where the parallel tasks return values via `future` handles. Dependent tasks resolve the value on the `future` handles before they can make computational progress. The first benchmark is the Fibonacci benchmark implemented similar to Figure 2.4 with support for a sequential cutoff. Fibonacci spawns two `future`s inside each task and attempts to retrieve values from both `future`s immediately afterwards. This allows us to measure the overheads of the `future` SyncCon in the different runtimes. The second benchmark is the Knapsack benchmark ported to HJlib from a Cilk version [73]. NQueens is the third benchmark where the goal is to find K solutions to placing N queens on a chessboard such that no queen can attack any other. The solution is computed by repeatedly placing a new queen in a non-attacking position on the board until N queens have been placed on the board. Each task spawns N tasks for each of these sub-problems and then perform a sum reduction over these tasks to compute the number of valid solutions. The Quicksort benchmark is a parallelized implementation of the quicksort algorithm and is written in a functional style with immutable values. The fifth benchmark is the Series bench-

mark from the Java Grade Forum Multi-threaded Benchmarks suite [74]. The final benchmark is a Smith-Waterman computation which uses a dynamic programming approach to compute the optimal alignment of two input strings. The computation of each element relies on three other elements: top, left, and top-left and is usually implemented as two nested loops. The well-defined data dependences in the loop makes it an ideal benchmark to use for `future`s and a `forall` loop (incrementing index in inner loop before incrementing index in outer loop order).

Figure 3.13 summarizes the results of `future` benchmarks. Two versions of the Fibonacci program were mentioned in Figure 3.3 and Figure 2.4. With the cooperative runtime we achieve performance close to the program written in event-driven style while still using the easier to read thread-based style. The cooperative version using `future`s comfortably outperforms the blocking versions. In fact, both JUCBLK and HJBLK run out of memory for larger values of `n` as the runtime attempts to create extra threads to compensate for the blocked threads while the cooperative version completes without creating any additional threads. The Knapsack benchmark is similar to Fibonacci in that two `future`s inside each task and attempts to retrieve values from both `future`s immediately afterwards. A distinctive feature of this benchmark is the support for early task terminations when tasks outrun the globally best-known value, this leads to an irregular computation. In the Smith-Waterman benchmark, due to the comparative lack of delay while trying to resolve a `future` after its creation many blocking operations are performed in the blocking runtime. This degrades performance, as seen in Figure 3.13, and the cooperative version outperforms the blocking version by a factor of 3.7×. On average, over all the benchmarks with the specified inputs, the HJCOOP runtime performs over 1.9 × better than the JUCBLK runtime and obout 2.4 × better than the HJBLK runtime. As with the `async-finish` benchmarks, these numbers would be larger for larger inputs where the JUCBLK and HJBLK runtimes would suffer additional context-switching overheads and extra memory pressure due to creation of large numbers of threads.

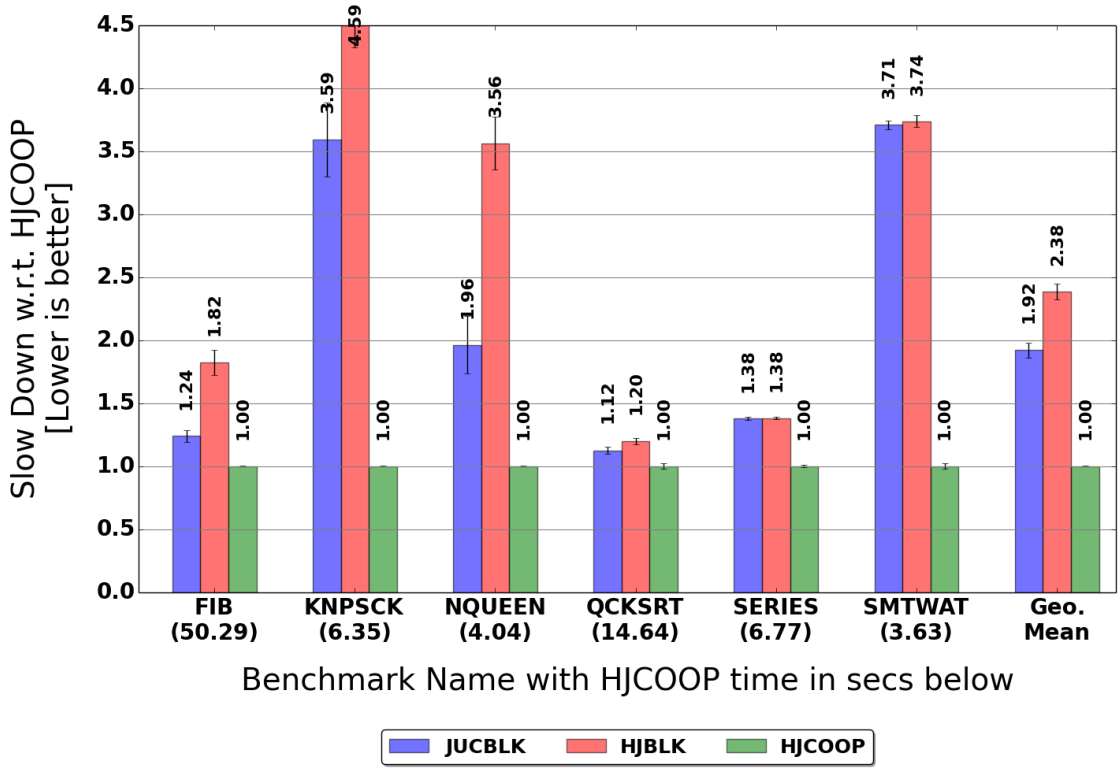

Figure 3.13 : Results for `future` benchmarks. Fibonacci (FIB) microbenchmark for the $100,014^{th}$ term with a cutoff to compute fib(100, 000) and lower terms sequentially. The Cilk Knapsack (KNPSCK) benchmark on 51 randomly generated items with a sack capacity of 2, 500 and sequential cutoff after 10 items have been processed in parallel. N-Queens (NQUEEN) benchmark on board of size 14×14 with sequential cutoff after recursive depth of 4. Quicksort (QCKSRT) on an input of size 160 million and sequential cutoff of 1 million. JGF Series (SERIES) which computes 16 million terms in 10 million steps. Smith Waterman (SMTWAT) on strings of length 1, 850 and 1, 010.

3.6.3 `phaser` Benchmarks

The HJCOOP runtime uses a non-blocking implementation of `phaser`s built using EDCs and OSDC. The HJBLK runtime also uses EDCs in its phaser implementation but relies on thread-blcoking operations. The JUCBLK runtime uses the `CyclicPhaser` class from the `java.util.concurrent` package in standard Java. We use six data-parallel benchmarks with phased computations to compare the perfor-

mance of the `phaser` implementation of the three runtimes. The Breadth-First Search using Bellman Ford algorithm (BELFRD), Dijkstra Routing (DIJKRT), and Vertex Coloring (VERCOL) benchmarks come from the IMSuite benchmark suite [75] for various graph algorithms written in task parallel languages. Each of these benchmarks take a configurable parameter to introduce a user-specified workload in each asynchronous task. The fourth benchmark is the KMeans benchmark [76] obtained from the X10 benchmarks source repository and ported to HJlib. The final tow benchmarks are the the Moldyn (MOLDYN) and Successive Over-Relaxation (SOR) benchmarks from the Java Grade Forum Multi-threaded Benchmarks suite [74].

Figure 3.14 summarizes the results of `phaser` benchmarks. Since our hardware had thirty-two cores, we registered more more than thirty-two tasks on the `phaser`s in each of the benchmarks to stress test the runtimes. This ensures that most of the registered tasks encounter a *forced* suspension point at the *next* operation as only a maximum of thirty-two tasks can be running at any given time. In the blocking runtime, each such suspension point causes the worker thread to block and additional threads are created to run the other tasks. As such, the runtime has to deal with the overhead of these additional thread context switches. In contrast, the cooperative runtime avoids such thread context switches and relies on the continuations to perform the relatively lightweight task context switches. The cooperative performs a comparatively cheaper task context-switch. Java's implementation of `CyclicPhaser` and the `phaser` implementation in the HJlib blocking runtime version perform similarly on most benchmarks (except KMEANS and SOR). On the various `phaser` benchmarks, the HJ cooperative version can be over 30× faster than the thread-blocking runtimes. On average, we can achieve more than 3× speed-up when compared to the JUCBLK and HJBLK runtimes. These speed-ups are even greater in some of the benchmarks when larger number of tasks are registered on the `phaser`s.

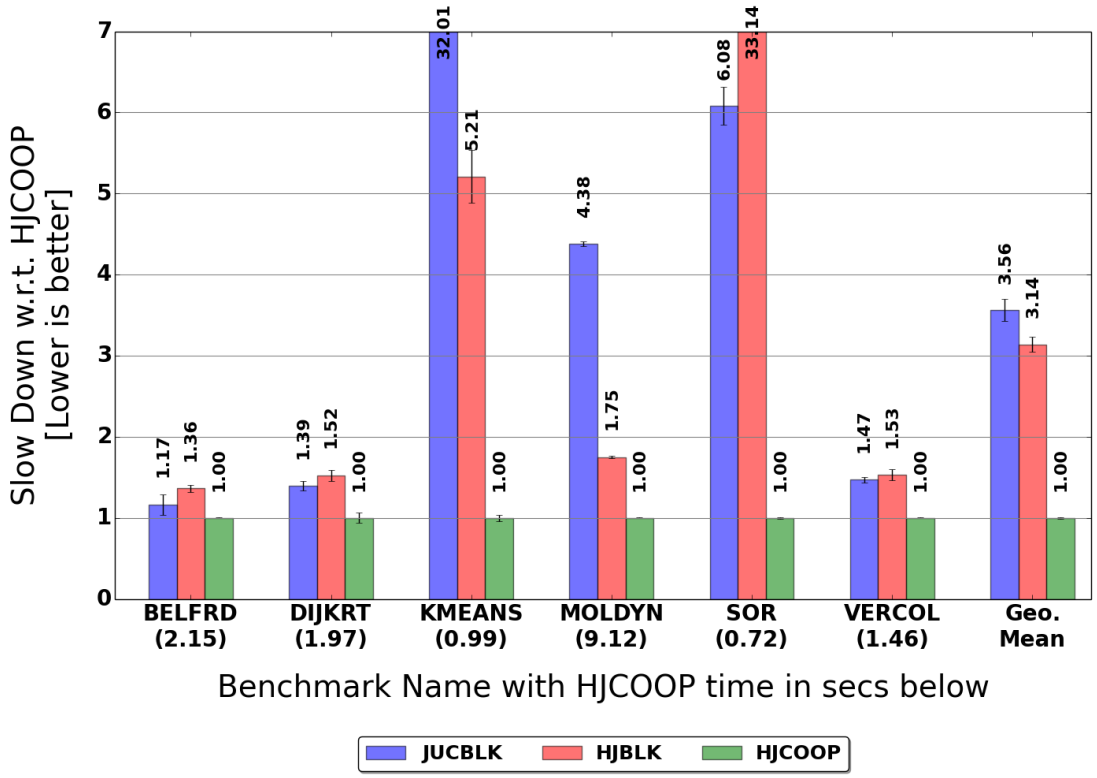

Figure 3.14 : `phaser` benchmark results. BFS Bellman Ford (BELFRD) with an input graph of size 512 nodes and artificial load values of 75 million. Dijkstra Routing (DIJKRT) with an input graph of size 512 nodes and artificial load values of 20 million. K-Means (KMEANS) benchmark from X10 with 300,000 points running 300 iterations with 60 tasks. JGF Moldyn (MOLDYN) 50 tasks registered on the `phaser` running 3,000 rounds. JGF Successive Over-Relaxation (SOR) benchmark with an input array size of 1,750. Vertex Coloring (VERCOL) with an input graph of 512 nodes and artificial load of 30 million.

3.7 Conclusions

In this chapter, we address the problem of scheduling parallel tasks with general synchronization patterns using a cooperative runtime for scalability and performance. Our solution is founded on a novel use of one-shot delimited continuations and event-driven controls. We describe recipes for implementing various SyncCons using our cooperative API and provide an implementation of our cooperative runtime for the Habanero-Java language. Experimental results for our implementation for Habanero-

Java, on various `future` and `phaser` benchmarks, show that the cooperative runtime delivers significant improvements in performance and memory utilization relative to a thread-blocking runtime system while using the same underlying work-stealing task scheduler.

Chapter 4

The Eureka Programming Model for Speculative Task Parallelism

Eureka! — I have found it!

Archimedes

A wide range of problems, such as combinatorial optimization, constraint satisfaction, image matching, genetic sequence similarity, iterative optimization methods, can be reduced to tree or graph search problems [77, 78, 79]. A pattern common to such algorithms to solve these problems is a *eureka* event, a point in the program which announces that a result has been found. Such an event curtails computation time by avoiding further exploration of a solution space or by causing the successful termination of the entire computation. For example, in optimization problems, a eureka event declares (and updates) the currently best-known result and can prune the computation by causing the termination of specific tasks that cannot provide a better result. On the other hand, in satisfiability problems, the first eureka event can trigger the termination of the entire computation as a proof of existence has been found.

With the advent of the multicore era, future growth in application performance will primarily come from increased parallelism. While many efforts have focused on programming models that expose increased amounts of deterministic parallelism, we believe that it is also important to explore new programming model directions for speculative parallelism. Eureka-Style Computations (EuSCs) include search and optimization problems that could benefit greatly from speculative parallelism. However,

writing parallel programs is a non-trivial, bug-prone, and complex endeavor in general, and the addition of speculation to current models can further add to the complexity. There is, hence, a need for programming models that support simple specification of parallel EuSC algorithms, combined with efficient parallel implementations.

One of the challenges in EuSCs is the efficient termination of multiple active tasks once a solution is published. Current termination techniques include the use of *a*) terminating processes and threads [80], *b*) exceptions for control flow as used in Microsoft's Task Parallel Library [81] and Java thread interrupts in blocking calls [82], *c*) function-scoped cancellation points in Cilk `abort` [29] and OpenMP 4.0 [83], and *d*) manual cooperative termination tokens as in Intel Thread Building Blocks [84]. As explained in Section 4.3, all these solutions have their limitations and are inadequate for supporting parallel EuSCs with high productivity and performance.

In this work, we introduce the Eureka Programming Model (EuPM), an explicitly task parallel programming model that simplifies the expression of parallel EuSCs. The EuPM builds on a structured `async-finish` task parallel programming model. It works by exploiting parallelism opportunities in computations that are divided into several *speculative tasks*; these tasks are called *speculative* because their results may or may not be needed. Section 4.3 motivates our approach to terminating speculative tasks once a result is published. Our solution uses a simplified cooperative termination technique that only requires a single method call at each eureka point - a point in a program that announces that a result has been found. The EuPM is described in Section 4.4. It promotes out-of-order executions and the constructs in our EuPM are expressive enough to encode many parallel programming patterns common to EuSCs (Section 4.5). These different patterns can also be safely combined or nested, thereby enabling both composability and reusability (Section 4.6).

We have implemented the EuPM as a Java-based task parallel cooperative runtime that runs on a standard Java Virtual Machine, and it is summarized in Section 4.7. Our approach could be implemented for parallel C++ programs as well.

The burden of performing code transformations to ensure that all redundant tasks are terminated cleanly at well-defined program points is assumed by the compiler and runtime. We evaluate the performance of search and optimization benchmarks, when using standard task-based solutions, hand-coded cooperative speculative task-based solutions, and solutions based on our EuPM. Experimental results (Section 4.8) show that the EuPM solutions can deliver significant performance and productivity improvements over standard task-based solutions. The EuPM abstraction achieves acceptable overheads with performance that is comparable to that of hand-coded speculative task-based solutions, while the EuPM solutions are simpler to write.

4.1 Contributions

Our work on the EuPM makes the following contributions [85]:

- Introduction of the Eureka Programming Model (EuPM) to simplify the expression and management of speculative parallel tasks, which are especially important for parallel search and optimization applications.
- A manifestation of the EuPM as a standard Java API, with compiler support for the cooperative termination of avoidable tasks at well-defined program points.
- Identification of various patterns that are well-suited for the eureka construct, but hard to implement using current parallel programming models. These patterns include search, optimization, convergence, and soft real-time deadlines.
- An implementation of the EuPM in a cooperative runtime for task parallelism that uses delimited continuations.
- An empirical evaluation of the productivity and performance benefits of the EuPM implementation on various EuSC benchmarks.

```
1
2 class ParallelSearch {
3  def atomicRefFactory() {
4   return new AtomicRef([-1, -1])
5  }
6  def doWork(matrix, goal) {
7   val token = atomicRefFactory()
8   finish
9    for rowIndices in matrix.chunks()
10    async
11     for (r in rowIndices)
12      procRow(matrix(r), r, goal, token)
13   // return either [-1, -1] or
14   // valid index [i, j] matching goal
15   return token.get()
16  }
17  def procRow(array, r, goal, token) {
18   for (c in array.length())
19    if goal.match(array(c)) // eureka!!!
20     token.set([r, c])
21     return
22 } }
```

Figure 4.1 : `async-finish` parallel search on a 2D Matrix. (Listings use pseudo-code syntax.)

4.2 Motivating Example: Parallel Search of 2D Array

We will consider a well-understood parallel programming example as a motivating example where speculative computation may be used for parallelization. We build on this example in the rest of the chapter to discuss various concepts and explain different EuSC variants. The example performs the parallel search of a 2D array to find the index of a particular item if it exists. The eureka event occurs when the item is found. The parallel version may expand (or generate) more states than a serial version. We are ready to tolerate such redundancy in the hope of a faster execution time in finding the result.

Figure 4.1 displays an implementation of such a parallel search using the `async` and `finish` constructs, a structured variant of the task parallel Fork/Join Model (Section 2.3.1). Using the `atomicRefFactory()` method, the program initializes a `token` container to invalid indices at line 7. Tasks are spawned at line 10 and enclosed in the finish scope declared at line 8. The finish scope ensures that all spawned tasks

complete before the result is returned at line 15. To overcome the overheads of tasking, a common strategy while working with arrays is to chunk the data. Each `async` processes a chunk of data using the `for` loop at line 11. The eureka event occurs at line 20 when the search successfully finds a match to the goal and updates the `token` atomic variable. Since we are interested in any result, we use `set` instead of a `compare-and-swap` operation. After the eureka event, the `procRow` method promptly returns.

Figure 4.1 highlights a few inefficiencies, also common to other EuSCs, while using `async-finish` style parallelism. Firstly, there is the need to pass the task-specific `token` variable to each relevant method in the call chain to allow triggering the eureka event. Secondly, returning early from `procRow` doesn't terminate the task as the task can continue to process other iterations of the `for` loop at line 11. Next, there are other tasks executing concurrently which need to be terminated. If these tasks are long running, there can be a potentially large wait time before the finish scope ends, and the result is returned. Finally, as per `async-finish` semantics, all tasks (including those sitting in the work queue) will be executed even after the result is known. We discuss existing solutions to these problems in terminating tasks and the drawbacks of such solutions in Section 4.3.

4.3 Task Termination Strategies

It is well-known that speculative computation can yield performance improvements over conventional approaches to parallel computing [86, 87]. The speculative tasks can be started eagerly before they are known to be required, for example, by spawning parallel tasks to search disjoint fragments of a data structure. Once a solution is found, other attempts at solving the problem may be avoided (in optimization) or terminated (in search). Eagerly terminating such tasks improves performance by minimizing the amount of unnecessary computations. One of the challenges in EuSCs is the efficient coordination of the termination of several related tasks. Supporting

termination requires ensuring that a task can stop gracefully and leave the system in a state that is known to be valid.

One approach is terminating processes and threads [80]. This is not a scalable solution as the runtime then needs to spawn additional processes or worker threads to maintain the parallelism in the application. When worker threads are terminated repeatedly, the overheads of resource initialization cause the performance of the application to degrade. In addition, terminating a task abruptly might cause a computation to be interrupted asynchronously which can cause havoc in the programmer's understanding of the code's behavior. As in [88], we believe that it should not be possible to terminate a task in any execution state, but only at places where certain program invariants hold such that the execution may be interrupted safely. As a result, a preemptive approach is not desirable, and a cooperative approach where the task actively decides when to terminate is preferred.

One cooperative approach is the use of exceptions for control flow [81, 82]. Using exceptions allows the task to terminate with the runtime providing the exception handler to process that exception. It has the benefit that only specific program points defined by the programmer need to be edited to insert the `throw` clause; the bodies of callees need not be modified*. Use of exceptions to terminate tasks fail when users provide custom handlers that inadvertently catch the exception being thrown. This prevents the exception from reaching the handler provided by the runtime and thus interferes with the termination logic. A compiler can rewrite the exception handlers to immediately rethrow these special exceptions and prevent user code from interfering with termination logic [89]. However, this policy fails to work in the presence of *inaccessible* functions (whose source code is not directly available for modification). In addition, native support for throw code is comparatively inefficient even in the absence of filling the stack trace. The frequent use of exception handlers for control flow program execution logic is expensive and should be avoided.

*Callee signatures may need to be modified to include the exception, however.

```
1
2 class ParallelCooperativeSearch {
3  def atomicRefTokenFactory() {
4   return new AtomicRefToken([-1, -1])
5  }
6  def doWork(matrix, goal) {
7   val token = atomicRefTokenFactory()
8   finish
9    for rowIndices in matrix.chunks()
10   async
11    for r in rowIndices
12    procRow(matrix(r), r, goal, token)
13    // cooperative termination check
14    if token.isResolved()
15     return
16  return token.get()
17 }
18 def procRow(array, r, goal, token) {
19  for c in array.length()
20   // cooperative termination check
21   if token.isResolved()
22    return
23   if goal.match(array(c))
24    token.set([r, c])
25    return
26 } }
```

Figure 4.2 : Parallel search with manual cooperative termination.

Another cooperative approach is function-scoped cancellation points [29, 83]. For example, possible locations for cancellation points in Figure 4.1 would be at lines 10 to 12 which include the scope at which the `async` was declared. These work better as the compiler rewrites the code to support task termination; however, the limitation is that cancellation is not possible when the code is executing in a nested function call.

Another approach is manual cooperative termination via cancellation tokens or interrupt checking [84, 82]. Within long-running tasks, manually inserted periodic termination checks allow the task to determine if further work is avoidable (i.e. the task can be terminated). The granularity of checks controls the trade-off between the responsiveness of termination of tasks and the overhead of such check calls. Figure 4.2 displays the program from Figure 4.1 with support for manual cooperative termination. This approach is cumbersome to write as the programmer needs to manually transform all methods to support this style with an additional `token` parameter. It

also requires `if` checks and early return statements (lines 14-15 and 21-22). Inserting such checks in the source code is awkward and impossible in the case of calls to inaccessible functions. If the computation includes inaccessible functions in the call stack, we need to wait for the body of each such function to complete before termination can be effected.

4.3.1 Delimited Continuation-based Cooperative Termination

Delimited Continuations (DeConts) were introduced by Felleisen in 1988 [59] where he referred to them as *prompts* (Section 3.3.1). DeConts work even in the presence of inaccessible functions in the call stack. When a computation is suspended, a DeCont causes control to return to the caller of the boundary function irrespective of the functions (including inaccessible ones) that are in the call path. DeConts are hence a good choice when a limited part of the computation needs to be saved/restored [64]. DeConts are notorious for being hard to use and to understand by developers (as opposed to compilers and runtime systems). Hence, in a system that uses DeConts, an approach that does not expose a developer to DeConts is desirable.

Our approach to termination is cooperative and relies on the transparent use of DeConts. DeConts are required to let the control return to the runtime by safely unrolling the task's call stack but not the runtime worker's call stack, after which the runtime can perform cleanup as if the task terminated or returned normally. The termination approach involves coordination between the code that requests termination (task that resolves a eureka) and the task that responds to termination (other executing tasks that have become redundant). Our approach guarantees that early termination of tasks can only occur at at well-defined program points: one of the check points or at points that resolve a eureka. If a task needs to be terminated, the call stack is unwound and control returned to the runtime via the use of DeConts. Such tasks are not resumed, so, unlike general DeConts, the state of the computation at that point need not be saved. Introducing a method call for the check introduces

some overhead, so tasks can decide if they are *terminable* and how responsive they want to be to interruption. We expect a sweet spot, where a balance is reached between the overheads incurred by checks and the gains from early task termination. The optimal frequency of checking is application-specific and is determined by the developer. While the programmer does not need to hard-code exceptions and `if` checks on cancellation, our approach attempts to merge the benefits of using:
(a) **Exceptions**: There is no need for repeated `if` checks every time a method returns from a terminable method in the call chain. Only specific program points need to be edited to insert a method call. No exceptions are used for control flow, and the approach is unaffected by the presence of exception handlers.
(b) **Cancellation tokens**: It allows the programmer to determine the frequency of checks to terminate a task and determine responsiveness. However, our approach does not require changing the signature to add an additional parameter, the cancellation token is discovered implicitly by the task (further described in Section 4.4). Also, the body of inaccessible functions do not need to complete before termination of a terminable task can be effected.

The main limitation of our cooperative approach is that the programmer has to determine the frequency of the check calls and manually insert the termination check calls. These termination checks can also be inserted automatically by a compiler. Previous work by Feeley on *balanced polling* [66] and in the Jikes VM *yieldpoints* [67] provide a scheme to automatically insert these calls while minimizing overhead. Another limitation of our approach is that termination should not be triggered in a critical section that is implemented with programmer-defined locks. Acquiring a lock but failing to ensure that it is released can cause termination of the overall computation to be arbitrarily delayed; this issue plagues all the cooperative schemes discussed above. The issue can be circumvented by the use of managed resource control techniques that allow the runtime to track the lock(s) obtained by the user code while executing critical sections. These locks obtained by a task can then be released by the

runtime when requested to do so. Such techniques include the use of custom virtual machines or language constructs to execute the critical section. For example, the Habanero programming model [34] includes such a language construct, called `isolated`, which would work with terminated tasks.

4.4 Programming with `Eurekas`

Parallel programming models ideally enable programmers to express parallel algorithms using abstractions that hide all but the relevant information to reduce complexity and to increase programmer productivity. Our goal is to define the Eureka Programming Model (EuPM) so that it can be used to write parallel programs for EuSCs more productively than current parallel programming models. In this section, we first introduce the `Eureka` construct that is used by speculative tasks to trigger eureka events. Next, we describe how parallelism is expressed via speculative tasks in the EuPM. We also explain how the termination of a single task, as well as a group of tasks, is supported in the EuPM.

4.4.1 `Eureka` Construct and API

A `Eureka` is a new construct that provides support for speculative parallelism in an `async-finish` setting. Once a `Eureka` construct has been *resolved* by reaching a stable value, it enables detection of a group of speculative tasks that can be terminated. It abstracts away implementation details, facilitates encapsulating any mutable state, and provides an API to allow tasks to concurrently notify eureka events as well as to query the state of the eureka object. Encapsulation simplifies data race avoidance while attending to concurrent eureka events triggered by speculative tasks.

As seen in Figure 4.3, a `Eureka` object has a well-defined life cycle; it can only transition between the states in response to eureka events. During its life cycle, a `Eureka` is in one of the following states:

(a) **new**: an instance of the `Eureka` has been created and initialized; however, it has

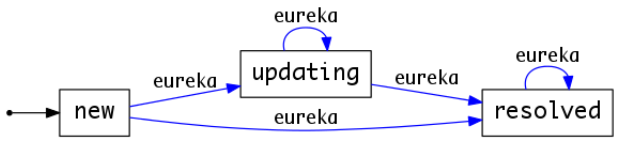

Figure 4.3 : Life-cycle of `Eureka`. The states and transitions will become clearer when we introduce the different `Eureka` patterns in Section 4.5.

not yet received any eureka events.

(b) **updating**: the `Eureka` has received at least one eureka event, and its internal state has not reached a stable value (e.g. computing minimum during optimization).

(c) **resolved**: the `Eureka` has reached a stable value; any subsequent eureka events may be ignored. Once a `Eureka` enters the resolved state, all speculative tasks that can trigger eureka events to update this `Eureka` become terminable and can be terminated.

We have developed an interface which supports basic behavior needed by tasks in EuSCs. This `Eureka` interface can be used to support a wide variety of patterns in EuSCs including search, optimization, and convergence. User's can also use this interface to build their custom implementations for `Eureka`s. The operations that can be performed on a `Eureka`, `eu`, are defined by the following interface:

(a) `offer(auxiliaryData)`: Notifies `eu` that a eureka event has been triggered; additional information used to mutate the internal state of `eu` is available in `auxiliaryData`. This operation enables `eu` to transition to different states in its lifecycle (as shown in Figure 4.3). Whether or not the event resolves `eu`, it can cause the task invoking this operation to terminate at a well-defined program point.

(b) `check(auxiliaryData)`: This operation allows a speculative task to check whether it has become terminable as, e.g., `eu` has been resolved. If the task has become terminable, a call to `check` will cause the task to be terminated. By accepting an argument, `check` enables the caller to pass additional values that can be used to determine whether to terminate a task.

(c) `isResolved()`: Allows a speculative task to query whether `eu` has been resolved.

This method returns a boolean value and never causes a task to be terminated.
(d) `get`(): If `eu` has been resolved, it returns the resolved value. Otherwise, a transitory value of `eu` is returned. One is guaranteed to receive the resolved value if this operation is invoked outside the `finish` scope on which `eu` was registered (e.g. as we will see in the explanation of Figure 4.4 line-13).

4.4.2 Eureka Programming Model (EuPM)

The EuPM is an extension of the task-parallel `async-finish` model where speculative tasks are created using the `async` keyword. Through the hierarchical nature of `async` and `finish` blocks, we advocate a structured approach to parallel programming of EuSCs. The task executing a `finish` has to wait for all transitively spawned child tasks created inside the finish scope to terminate before it can proceed. A `finish` block can register on a `Eureka`, `eu`, with the following pseudocode syntax (the library API includes `eu` as a parameter to the `finish` API): `finish`(eu) $\langle stmt \rangle$. The `finish` construct simplifies the identification of the group of tasks that participate in a eureka-style synchronization on a particular `Eureka` instance.

All tasks having the same immediately enclosing finish (IEF) belong to the same group and inherit the registration on the `Eureka` instance, `eu`, from the finish scope. Finish scopes with different `Eureka` instance registrations can be nested allowing composability of different speculative computations. Similarly, multiple `finish` blocks can register on the same `Eureka` instance, `eu`, to represent that different speculative sub-computations are linked. When one of the speculative tasks resolves `eu` it makes other tasks from the same or different groups also registered on `eu` to become redundant and terminable. If none of the tasks trigger a eureka event that resolves the registered `eu`, the computation completes normally when all tasks inside each `finish` scope complete.

The EuPM specific operations that a task, `T`, can perform on a `Eureka`, `eu`, are

defined as follows:

(a) **new**: Task `T` can create a new instance of the `Eureka` construct, `eu`, and obtain a handle to it. The reference `eu` can now be used to register on new `finish` scopes. The creator task can pass the reference of `eu` to other tasks.

(b) **registration**: `eu` can be explicitly registered on a `finish` scope. Note that the task that created `eu` cannot register on `eu`. A newly spawned task, `T`, implicitly registers on `eu` only if the IEF of `T` was explicitly registered on `eu`. Currently, we do not provide a mechanism for an `async` task to explicitly register on `eu`.

(c) **offer**: This method retrieves the `Eureka` instance, `eu`, that the task is registered on and invokes the `eu.offer`() method to notify `eu` of a eureka event. This simplifies the computation body of `T` where method calls do not need to add an extra parameter to pass `eu` down the call chain. Invoking this method can cause the task to terminate (depending on the implementation of `eu`).

(d) **check**: A task indirectly performs a check on `eu` by invoking the static `check` method. Like the previous operation, it retrieves `eu` to make the call `eu.check`(). Invoking this method can cause the task to terminate (depending on the terminating logic of `eu`), so programmers need to ensure that side-effects introduced by `T` are in a consistent state at the program point where this method was invoked.

With the EuPM, a programmer can focus on writing operational code explicitly specifying potentially parallel operations, leaving the underlying details of parallel execution and termination detection to the runtime system. The EuPM places no requirements on the use of a shared-memory infrastructure. Like the `async-finish` model, the EuPM presented is also applicable to a distributed environment. One of the key features of a system that supports EuSCs is the efficiency with which the eureka events are triggered. The EuPM provides the abstraction, static `offer` method, that simplifies how eureka events are triggered in tasks. Invalid calls to `check/offer` from a task not executing in a EuSC (i.e. `finish` scope not registered on a `Eureka`) results in a runtime error.

```
1
2 class ParallelEurekaSearch {
3  def eurekaFactory() {
4   return new SearchEureka([-1, -1])
5  }
6  def doWork(matrix, goal) {
7   val eu = eurekaFactory()
8   finish (eu) // eureka registration
9    for rowIndices in matrix.chunks()
10    async
11     for r in rowIndices
12      procRow(matrix(r), r, goal)
13   return eu.get()
14  }
15  def procRow(array, r, goal) {
16   for c in array.length()
17    check([r, c]) // coop. term. check
18    if goal.match(array(c))
19     offer([r, c]) // trigger eureka event
20 } }
```

Figure 4.4 : Parallel search using the Eureka model.

Another feature is the efficiency with which the state of the program can be updated after the result has been found. The EuPM needs to provide a means to easily detect tasks that need to be terminated and a mechanism to guarantee effective termination of terminable tasks. Once a `Eureka` instance, `eu`, moves to the resolved state, all incomplete tasks belonging to a finish scope registered on `eu` become terminable. Any ready tasks belonging to the *resolved* finish scope can be terminated by removing them from the work queue – each task is assumed to contain an implicit `check` call at the start of the task execution. Redundant executing tasks are terminated at program points where the `check` method is invoked. This allows tasks to terminate cooperatively in a programmer controlled manner and, more importantly, simplifies reasoning about the correctness of the speculatively parallel program.

Figure 4.4 displays the parallel search program of Figure 4.1 using `async` and `finish` constructs in the EuPM. We create the `SearchEureka` instance, `eu`, inside the factory method `eurekaFactory`. This instance, `eu`, is registered by the `finish` scope defined on line 8. Hence, all `async` tasks launched at line 10 are automatically registered on `eu` and belong to the same group. The tasks trigger the eureka event

by invoking the `offer` method at line 19. There is no need for an explicit `return` statement in `procRow`, as `offer` on a `SearchEureka` causes the task to terminate. To enable cooperative termination, there are also calls to `check` (line 17) to check the state of the registered eureka implicitly. When `eu` has been resolved, `check` will cause the terminable tasks to terminate. Eventually, all tasks inside the `finish` block at line 8 will complete execution or be terminated, and the computation will proceed to line 13 and the result will be returned. Note that, like Figure 4.2, the final answer in this example is nondeterministic, but there are no data races involved. It should be noted that this program (20 lines) required fewer lines of code than the equivalent program in Figure 4.2 (26 lines). In addition, the code in Figure 4.2 is more complicated and error-prone than the code in Figure 4.4. As we will see in Section 4.5, we will build on this example to explore various EuSC patterns with minimal change in the kernel code (lines 6 to 20 of Figure 4.4).

4.4.3 Redundant Computation Guarantee

The EuPM provides a bound on the avoidable work done by speculative tasks once a `Eureka`, `eu`, has been resolved. Let there be P processing units (worker threads) in use by the EuPM runtime. Let a total of T speculative tasks be (dynamically) spawned by computations of `finish` scopes registered on `eu`. Let each task perform a total work of W. The total work done by this computation in the absence of eureka support is $T \times W$. Assume $C - 1$ tasks have executed to completion and the C^{th} task triggers the eureka event that resolves `eu`. The avoidable work done by this computation in the absence of termination support or tasks which are non-terminable is $(T - C) \times W$. If each task chooses to be terminable, the frequency of checks is programmer controlled and let us assume the tasks to have `check` calls every N work units. When `eu` is resolved, there are at most $A = min(P - 1, T - C)$ tasks executing on the other $P - 1$ processing units that are performing avoidable computation. The EuPM runtime ensures any avoidable tasks in the work queue will

not be executed. In the worst case, the total avoidable work done after the eureka event will be $A \times W$. However, each of these will perform an additional N units of work before their next termination check. In the EuPM, the total avoidable work done after the eureka event will be $A \times N$. We can also do an average-case analysis as follows. Let n_i be the average number of termination checks performed by the avoidable tasks before they are terminated, then the average total work done by the computation is $(C \times W) + (n_i \times A \times N)$.

4.5 Parallel Patterns and Eureka Variants

In this section, we describe frequently occurring patterns that arise in EuSCs and how they can be solved using the EuPM. These patterns include computations that produce both deterministic and non-deterministic results.

4.5.1 Parallel Search

Search is a well-known pattern in EuSCs. It is a non-deterministic computation in the sense that if the goal is present at multiple locations in the data structure being searched, any of those locations is an acceptable result. Searching disjoint partitions of a data structure can be done in parallel though it may considerably increase the amount of work that the algorithm performs. Such parallelism is speculative since more than one partition may contain a solution. Once the result is discovered, all parallel searching entities should ideally be terminated as quickly as possible to minimize doing redundant work. With respect to the EuPM, this means that the first eureka event triggered by a task will resolve the `Eureka` instance, `eu`, registered by the task. Hence, a `SearchEureka` construct is designed to be resolved by the first eureka event it processes, and it promptly terminates the task that triggered the event. Any subsequent calls to `check` or `offer` by other tasks registered on `eu` result in those tasks being terminated.

The same concept can be used for termination detection in the `finish` statement

```
1
2 class AsyncFinishEurekaCount {
3   def eurekaFactory() {
4     val K = 4
5     return new CountEureka(K)
6   }
7   // same code as  Figure 4.4
8 }
```

Figure 4.5 : Example of counted parallel search using the Eureka model.

with regards to exception semantics. If any `async` throws an exception, then it can resolve an implicit `SearchEureka` registered by the `finish` scope. All other `async`s belonging to the same IEF can then be terminated at their next `check`/`offer` checkpoint. The IEF can then rethrow the exception thrown by the `async`. This offers an alternate strategy to the *MultiException* scheme [9] where a collection of all exceptions thrown by all `async`'s in the IEF is rethrown.

4.5.2 Count Eureka

Another variant of a parallel search is where we wish to know the first K results that match a query. This pattern is inspired by the `ParallelTry` command in Mathematica 7 [90]. In this pattern, we wish to terminate the computation when at least K of the asynchronous computations have completed successfully. Any evaluations still underway after K results have been received are avoidable and should be terminated. Like the `SearchEureka` pattern, the `CountEureka` pattern produces non-deterministic results as the results received are dependent on the scheduling of parallel tasks and the arrival of concurrent eureka events.

Figure 4.5 displays the use of the `CountEureka` construct to enable supporting this pattern in our EuPM. The program from Figure 4.4 can be modified to use the `CountEureka` construct by changing the factory method. A `CountEureka` is initialized with a count K and is resolved after exactly K eureka events have been triggered. Once resolved, any calls to `offer` and `check` cause the calling task to be terminated. A call to `CountEureka.get()` returns a list of values of maximum length K instead of

a single value. If none of the tasks triggered a eureka, then an empty list is returned. In general, a `SearchEureka` can be viewed as a `CountEureka` with a count of 1.

4.5.3 N-Version Eureka

N-Version Programming [91] uses software redundancy to achieve fault-tolerance. In N-Version Programming, there are multiple functionally equivalent implementations of the same specification. These implementations can run be independently in parallel to compute results; the results are notified using eureka events. Using a decision algorithm, such as when any N (≥ 2) agree on their results, the eureka is resolved. The agreed upon value is accepted as the result matching the specification, and other computations are terminated. This pattern also produces non-deterministic results as the final result is dependent on the scheduling of parallel tasks and the arrival of concurrent eureka events from the independent implementations.

4.5.4 Optimization Eureka

Many problems from artificial intelligence can be defined as combinatorial optimization problems. For example, Branch and Bound (BnB) is a widely used tool for solving large-scale NP-hard combinatorial optimization problems [92]. A BnB algorithm searches the complete space of solutions for a given problem for the best solution. Subproblems are derived from the originally given problem through the addition of new constraints. An objective function computes the lower/upper bounds for each subproblem. The upper bound is the worst value for the potential optimal solution; the lower bound is the best value. The entire tree maintains a global upper bound (GUB): this is the best upper bound of all nodes. Nodes with a lower bound higher than the GUB are eliminated from the tree because branching these sub-problems will not lead to the optimal solution. In many practical cases, the amount of pruning that occurs in this type of BnB algorithm can be very significant.

In parallel implementations, pruning the branches of the search tree may lead

```
1
2 class AsyncFinishEurekaMinima {
3   def eurekaFactory() {
4     // comparator to compare indices
5     val comparator = (a, b) -> {
6       ((a.x - b.x) == 0) ? (a.y - b.y) : (a.x - b.x)
7     }
8     val initValue = (INFINITY, INFINITY)
9     return new MinimaEureka(initValue, comparator)
10  }
11  // same code as  Figure 4.4
12 }
```

Figure 4.6 : Example of parallel search finding the lowest index of the goal node in the array if it exists.

to terminating existing computations. The structure of the BnB search requires the ability to terminate individual subtrees of the search tree [93]. A BnB version of our array search example is where we are interested in finding the lowest index of the goal item if it exists in the array. We can achieve this by modifying the factory method in Figure 4.4 to return a `MinimaEureka` instance. Figure 4.6 displays such a BnB version of our array search example where we are interested in finding the lowest index of the goal item if it exists in the array. In our EuPM, the GUB is available in the `MinimaEureka` instance, `eu`, that a speculative task is registered on and can be retrieved by a call to `eu.get()`. Calls to `check` and `offer` pass the current known upper bound or solution, respectively, as the argument. If the argument in the `offer` call is lower than the GUB, the GUB is updated in the `MinimaEureka` instance, otherwise the current task is terminated. Similarly, calls to `check` terminate a task if the argument is larger than the currently known GUB in `eu`.

4.5.5 Soft Deadlines

For soft real-time systems [94] the goal is to meet a certain subset of deadlines to optimize some application-specific criteria. If a soft real-time task takes longer than the allotted time since its creation to complete, then it needs to be terminated with its latest results. Another similar notion is that of engines that abstract the notion of

timed preemption [65]. An engine is run by giving it a quantity of abstract time units that measure computation. If the engine completes its computation before running out of units, it returns the result of its computation and the quantity of remaining units. If it runs out of units, the computation is terminated. Unlike Haynes' original notion of engines, nesting of engines is allowed in our model thus allowing time units to be divided among parallel sub-tasks if required.

In our soft deadline version of the array search example, the deadlines could be overall execution time (soft real-time) or number of comparison operations performed (abstract time units). The eureka version of these programs helps the system by releasing the resources of the tasks which have already missed their deadlines and allocating more resources to the other tasks which can still potentially meet their deadlines. We support both kinds of `Eureka`s in the form of `TimerEureka` and `EngineEureka`, respectively. In our array search example of Figure 4.7, the deadlines could be overall execution time (soft real-time) or number of comparison operations performed (abstract time units). A `TimerEureka`, `eu` is resolved when either the first eureka event is triggered or the computation runs out of time since the creation of `eu`. An `EngineEureka`, `eu` is resolved when either the first eureka event is triggered or the computation runs out of time units (measured by the sum total of the arguments to `check`). These `Eureka` instances can trigger the termination of a group of tasks without an explicit `offer` from a task. Note that since tasks only get cancelled when they invoke `check`, tasks can run for much longer than the allotted time unless the user is careful with the calls to `check`. This is consistent with our philosophy of allowing the programmer to determine responsiveness (Section 4.3).

4.5.6 Convergence Iterations

Iterative methods [95] refer to a wide range of techniques that use successive approximations to obtain more accurate solutions to a set of equations at each step. Examples of iterative methods include the Jacobi method, Gauss-Seidel method, and

```
1
2 class AsyncFinishEurekaSoftRealTime {
3   def eurekaFactory() {
4     // terminate finish scope if running after 4 seconds
5     val time = 4.seconds
6     return new TimerEureka(time)
7   }
8   // same code as  Figure 4.4
9 }
10 class AsyncFinishEurekaEngine {
11   def eurekaFactory() {
12     val units = 400 // allow 400 units of fuel in check calls
13     return new EngineEureka(units)
14   }
15   // same code as  Figure 4.4  except:
16   def processRow(rowData, rowIdx, goal) {
17     ...
18     check(1) // 1 unit consumed per comparison
19     ...
20   }
21 }
```

Figure 4.7 : Example of parallel search with soft deadlines.

the Successive over-relaxation method. An iterative method is called *convergent* if the corresponding sequence converges for given initial approximations. Speculatively parallelizing an iterative algorithm results in creating tasks for computations of *future* iterations.

Figure 4.8 displays an example which computes r using the equation: $x_{i+1} = f(x_i)$, $y_{i+1} = g(y_i)$, $r_i = h(x_i, y_i)$ and converges when successive values of r become *close*. The parallel version launches `maxIters` iterations ahead of time (line 23) and parallelizes the computation to respect the dependences (using the `await` clause). The `await(T1, ..., TN)` clause in a task causes it to be suspended from execution until the execution of tasks `T1, ..., TN` has completed. The `await` clause in line 36 ensures that values of r are offered to `eu` in the expected order. Once convergence is reached (i.e. when true is returned by the call to `predicate` inside `eu` with the current and new value) `eu` is resolved and all tasks spawned by the computation need to be terminated. Note that this includes tasks that may be transitively suspended on `await` clauses as each await task is assumed to contain an implicit `check` call at the start of task execution (Section 4.4.2). Calls to `check` in the `async`s and possibly

```
1
2 class ParallelEurekaConvergence {
3   def eurekaFactory(initVal, tolerance) {
4     val predicate = (a, b) -> {
5       Math.abs(decimalDiff(a, b)) <= tolerance
6     }
7     return new ConvergenceEureka(initVal, predicate)
8   }
9   def doWork() {
10    val maxIters = 100
11    val initVal = INFINITY
12    val tolerance = 1e-4
13    val eu = eurekaFactory(initVal, tolerance)
14    finish (eu) {
15      // arrays to store task handles
16      val xs = newArray(maxIters + 1)
17      val ys = newArray(maxIters + 1)
18      val rs = newArray(maxIters + 1)
19      xs[0] = async { return xInit }
20      ys[0] = async { return yInit }
21      rs[0] = async { return initVal }

23      for (i in 1 to maxIters) {
24        xs[i] = async {
25          await(xs[i - 1]) // dependence
26          check()
27          return f(xs[i - 1])
28        }
29        ys[i] = async {
30          await(ys[i - 1]) // dependence
31          check()
32          return g(ys[i - 1])
33        }
34        rs[i] = async {
35          check()
36          await(xs[i], ys[i], rs[i - 1])
37          val iterRes = h(xs[i], ys[i])
38          // converge if rs[i] and rs[i-1] are close
39          offer(iterRes)
40          return iterRes
41    } } }
42    return eu.get()
43 } }
```

Figure 4.8 : Example of an iterative method using the Eureka model.

inside methods `f`, `g`, and `h` ensure executing tasks can be terminated early.

4.6 Reusability and Composability of Eureka Components

Abstractions and productivity are among the most important requirements for programming models. Further, it is important for the abstractions to be as orthogonal as possible, so as to aid composability and reusability. Constructing reusable compo-

nents aids in programmer productivity by simplifying building of larger systems from relatively simpler parts. While current approaches to support EuSCs lack composability and reusability in general, we show in this section that all the different styles of eureka computations presented in Section 4.5 can be safely combined or nested thereby enabling general composability and reusability.

4.6.1 Composability by Component Composition

Component composition involves the systematic combining of independent components to form useful components. Such incremental aggregation of existing components yields further components. This method is scalable as code replication is avoided while implementing new functionality. In the EuPM, independent `Eurekas` can be combined to form new `Eurekas`. The constituent `Eurekas` are used as encapsulated black-box components and are accessed solely through their exposed interfaces (the `check` and `offer` operations).

We support basic logical conjunctive and disjunctive binary composition semantics for `Eurekas`. Calls to `offer` and `check` need to be passed a pair of values, one for each component `Eureka`. The conjunctive composition of `Eurekas` is considered resolved only when both the constituent `Eurekas` are resolved. A disjunctive composition of `Eurekas` is considered resolved when either of the constituent `Eurekas` is resolved. Since the state of a `Eureka` instance evolves monotonically (once resolved a `Eureka` always remains resolved), the binary composed `Eureka` also preserves monotonicity, i.e., resolution of a `Eureka` is a stable property.

Figure 4.9 shows an example with a conjunctive eureka. We extend the example from Figure 4.4 to search for two target items in parallel and report a success only when both items are found. The example creates two search `Eurekas` and then creates a conjunctive eureka, `eu`, at line 10. This `eu` is used to register on the finish scope and launch the parallel search tasks on individual rows. Each call to `check` and `offer` now passes a pair of values (lines 24 and 31). The internal implementation delegates the

```
1
2 class ParallelEurekaDoubleSearch {
3   def eurekaFactory() {
4     val initValue = [-1, -1]
5     return new SearchEureka(initValue)
6   }
7   def doWork(matrix, goal1, goal2) {
8     val eu1 = eurekaFactory()
9     val eu2 = eurekaFactory()
10    val eu = eurekaComposition(AND, eu1, eu2)
11    finish (eu) // eureka registration
12      for rowIndices in matrix.chunks()
13        async
14          for r in rowIndices
15            procRow(matrix(r), r, goal1, goal2)
16    // eu.get() returns pair of values
17    return eu.get()
18  }
19  def procRow(array, r, g1, g2) {
20    for c in array.length()
21      // pair of values for eu1 and eu2
22      val checkArg = [[r, c], [r, c]]
23      // cooperative termination check
24      check(checkArg)
25      val loopElem = array(c)
26      val res1 = g1.match(loopElem) ? [r, c] : null
27      val res2 = g2.match(loopElem) ? [r, c] : null
28      // pair of values for eu1 and eu2
29      val foundIdx = [res1, res2]
30      // possible eureka event
31      offer(foundIdx)
32  } }
```

Figure 4.9 : Example of a parallel search on two elements of an 2-D array using the Eureka model.

individual values from the pairs to the component `Eureka`s. These individual values may end up resolving only one of the component `Eureka`s. The overall conjunctive eureka, `eu`, is resolved when both component `Eureka`s are resolved, possibly from different calls to `offer` in different tasks. Once `eu` has been resolved, calls to `check` (line 24) will result in the task being terminated.

4.6.2 Reusability by leveraging Functional Decomposition

Function decomposition refers to the process of splitting a computation into multiple fragments. These fragments are invoked as helper functions and their results combined to produce the overall result. Alternatively, it can also be viewed as the functional

composition of existing functions so that the results of these function calls are used to evaluate a larger computation. Reuse is achieved by building functions on top of existing functions. Such uses of functions for individual computation fragments simplify maintainability and avoid code duplication.

In the EuPM, we enable opportunities for functional decomposition by allowing the nesting of EuSCs with `finish` and `async` statements. This nesting can be arbitrarily deep and allows exposing nested fork-join parallelism opportunities in distinct EuSCs. Nesting of `finish` blocks registered on `Eureka` instances is allowed and enables composability of different speculative computations. Each `finish` scope registered on the same `Eureka` instance forms a single group. When different EuSCs are nested, calls to `offer` are delegated to the registered `Eureka` on the IEF of the currently executing task. However, calls to `check` are recursively delegated to registered `Eureka` instances up the nested `finish` scope hierarchy. This allows the innermost EuSC to continue to work as before, but tasks may be aborted if `Eureka`s up the hierarchy have been resolved by other parallel tasks. Thus, nesting EuSCs causes a tree hierarchy of `Eureka`s to become linked whereby resolving a `Eureka` up the hierarchy causes computations lower in the tree to be terminated.

This nesting mechanism is explained in Figure 4.10 which shows an example to do a search in a multidimensional array. The solution reuses the `doWork` function from Figure 4.4 and is thus performing functional composition. Nested EuSCs are also created at line 15 where an individual `Eureka` instance is created for every row (line 14) in the leading dimension of the array in the recursive call. When the `Eureka`, `eu`, is resolved for a dimension `N1`, it causes all nested eureka computations processing dimension `N2` (where `N2` $<$ `N1`) to be treated as resolved. As a result, subsequent calls to `check` (at line 18) in the redundant tasks of the nested computations will cause the tasks to terminate. After returning from a call to a nested eureka computation, the result is either returned immediately (for the base case at line 12) or processed further (line 19). Further processing involves updating the result index with the value

```
1
2 class ParallelEurekaArraySearch {
3   def eurekaFactory(dim) {
4     val initValue = array(dim).fill(-1)
5     return new SearchEureka(initValue)
6   }
7   def doWork(matrix, dim, goal) {
8     if (dim < 2)
9       throw IllegalArgumentException("Invalid dimension: " + dim)
10    else if dim == 2
11      // reuse by call to existing eureka computation Figure 4.4
12      return ParallelEurekaSearch.doWork(matrix, goal)
13    else
14      val eu = eurekaFactory(dim)
15      finish (eu) // nested eureka registration from recursive calls
16        for i in matrix.length()
17          async
18            check(array(dim-1).fill(-1).insert(i)) // termination check
19            val resIndices = doWork(matrix(i), dim-1, goal)
20            if isIndexNonNegative(resIndices)
21              offer(copy(resIndices).insert(i))
22      return eu.get()
23  } }
```

Figure 4.10 : Example of parallel search on a multidimensional array using function decomposition and nested eureka computations in the Eureka model.

for the current dimension before offering the result via `offer` (line 21).

4.7 Implementation

Despite any productivity promises, a parallel programming model must be implementable in an efficient and scalable fashion for it to be accepted by programmers. Our implementation of the EuPM is an extension of a Java-based task parallel runtime [34] that supports cooperative scheduling of `async-finish` style computations, though our ideas can also be implemented in other languages including C/C++. For example, Dolan et al. have implemented support for one-shot continuations in C++ using a new LLVM compiler primitive [96]. Figure 4.11 highlights the features and responsibilities of the task-parallel Eureka runtime in our model. These include implementation of efficient eureka variants; management and scheduling of the speculative tasks; classification of tasks into eureka groups; termination of redundant tasks; and synchronization and coordination of tasks inside `finish` blocks. The challenges in the

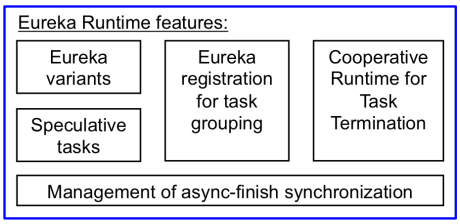

Figure 4.11 : Features supported by a Eureka task parallel runtime.

implementation of the EuPM involve effective load balancing of tasks, terminating tasks in a group efficiently, and supporting `Eureka`s in a scalable manner.

Our implementation supports all the `Eureka` variants described in Section 4.5 based on the API defined in Section 4.4.1. The key challenge is to support the synchronization by keeping each thread busy without any blocking. We rely on Java's support for atomic variables to implement the `Eureka`s and detect when a `Eureka` has been resolved. In particular, we rely heavily on the use of *compare-and-swap* operations, on `AtomicBoolean` and `AtomicLong` instances. This ensures thread safety and avoids data races in the concurrent calls to `check` and `offer` on the `Eureka`.

Our runtime uses the help-first policy [48] and maintains an independent stack for each task. Hence, any worker thread may execute a task, and we can use either work-stealing or work-sharing scheduling policies in our runtime. Since subproblems are generated and consumed dynamically, we rely on the load balancing algorithm provided by Java's `ForkJoinPool`. The `ForkJoinPool` is an optimized thread pool, which employs a work-stealing mechanism to efficiently distribute submitted tasks among its pool threads. Figure 4.12 displays how the runtime uses work-stealing threads to schedule tasks. Each task also maintains a reference to its IEF and inherits the `Eureka` registration from its IEF. This `Eureka` instance is stored in the IEF and used as the recipient while delegating calls to `check` and `offer`. Thus the tasks registered with the same `Eureka` instance are implicitly grouped together and become terminable when the `Eureka` is resolved. The tasks are eagerly terminated when there is a call to `check` or `offer`. Tasks executing inside a `finish` scope not registered with

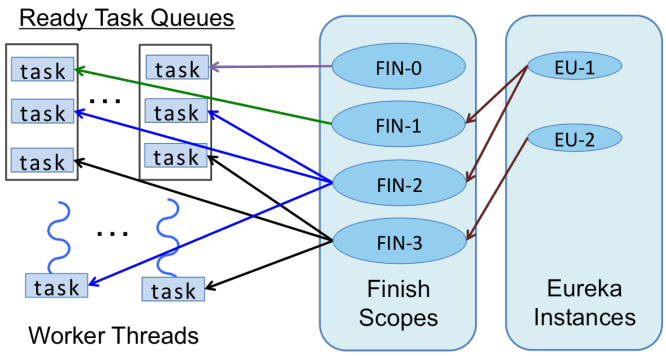

Figure 4.12 : Execution of parallel Eureka tasks in a work-stealing environment.

a `Eureka` (e.g. `FIN-0`) execute as regular `async-finish` style tasks without support for early termination.

As mentioned in Section 4.3, we rely on a cooperative task termination technique. When a task needs to be terminated at one of the check points, the call stack is unwound and control returned to the runtime via the use of Delimited Continuations (DeConts) [59]. Our implementation conforms to the constraints imposed by a standard Java Virtual Machine (JVM). In particular, standard JVMs do not provide support for DeConts or for storing and restoring the stack that we need to support cooperative termination. We have built a cooperative runtime that schedules tasks in the presence of end-of-finish synchronization constraints without blocking underlying worker threads using the strategy described in [49]. The DeConts created are thread independent and can be resumed on any worker thread. This strategy is known to provide a more scalable solution than other schemes that use thread-blocking operations [49].

We use an extended version of the open source bytecode weaver provided by the Kilim framework [55] to support DeConts. The Kilim bytecode weaver works by transforming the code of methods which can terminate. It recognizes such methods by the presence of a `SuspendableException` exception in the method signature. It is important to note that no actual exceptions are thrown or caught which minimizes the overhead of capturing and resuming continuations. Instead, the transformation

performed is similar to a continuation passing style transformation, except that only methods that can suspend are transformed. The weaver also allocates custom state objects to store local variables to enable restoring the computation. We extended Kilim to enable execution of parallel tasks run by worker threads in the `ForkJoinPool`. Then we added support for terminated tasks. Such tasks are never resumed, so unlike general DeConts the state of the computation at that point need not be stored. This avoids additional memory allocation and garbage collection overheads while suspending the DeCont.

Next, we extended the classical `async-finish` constructs with support for the proposed `finish` and `eureka` constructs used in EuSCs. We provide support for EuPM `finish` constructs that register on a `Eureka`, `eu`, and each time a task is spawned from the finish scope, the task is also registered on `eu`. Nested finish scopes can register on different `Eureka` instances as each finish scope maintains its own `Eureka`. This enables composability of different speculative computations. Static helper methods, such as `check` and `offer`, then retrieve the `Eureka`, `eu`, registered with the currently executing task from its IEF before delegating the call on `eu`. Whenever a eureka is resolved, the finish scope, `f`, is notified, and all tasks whose IEF is `f` and that are in the work queue are terminated immediately. In-flight executing tasks belonging to `f` are terminated at `check` or `offer` points as termination is cooperative. Terminating executing tasks is done by suspending the current DeCont and flagging it as terminated, so that the runtime can perform cleanup operations and schedule other tasks for execution. Once all of these tasks have been successfully terminated, the end-of-finish point for `f` is resumed.

4.8 Experimental Results

The benchmarks were run on individual nodes in an IBM POWER7 compute cluster. Each node contains 256GB of RAM and four eight-core IBM POWER7 processors running at 3.8GHz each. The software stack includes IBM Java SDK Ver-

Benchmark Name	Acronym	Source (Eureka Pattern)	Description
Single Linear Search	SLS	Authors (Section 4.5.1)	Search for a single item in a 2D array.
Unbalanced Tree Search	UTS	UTS [97], (Section 4.5.1)	Search for a goal node in UTS trees which represent characteristics of various parallel unbalanced search applications.
Sudoku	SUD	Authors (Section 4.5.1)	Solving a Sudoku puzzle by exploring a game tree.
NQueens first K solutions	NQK	Authors (Section 4.5.2)	Find first K solutions to placing N queens on a chessboard such that no queen can attack any other.
UT Shortest Path	UTP	Authors (Section 4.5.4)	Trees from the UTS benchmark, adds edge weights to find shortest path to any goal node.
BOTS Floorplan	FLP	BOTS [72], (Section 4.5.4)	Compute the optimal floorplan distribution of a number of cells using branch and bound technique.
Traveling Salesman Problem	TSP	R. Wiener [98], (Section 4.5.4)	Solved using a branch and bound algorithm.
Jacobi 2D	J2D	Authors (Section 4.5.6 style)	Stencil computation with iterative convergence.
K-Means Clustering	KMC	Authors (Section 4.5.6 style)	An iterative refinement technique which converges to a local optimum.
Double Linear Search	DLS	Authors (Section 4.6.1)	Search for two items in a 2D array.
Composite Search	CS	Authors (Section 4.6.2)	Search for a single item in a multi-dimensional array.

Figure 4.13 : Configurations of the benchmarks: SLS on a $1,000 \times 2,500,000$ array, with the result located at index (350, 875000). UTS using the UTS T1 configuration, a geometric tree with a branching factor of 4 and a maximum height of 12; graph of size 164 million. SUD on a 25×25 board with 196 unsolved entries. NQK computes first 250 thousand solutions on board size of 15×15. UTP tree using the UTS T3 configuration, a binomial tree with a maximum height of 11; 700 goal nodes; and graph of size 13 million nodes. FLP on a 64×64 grid with 14 cells. TSP on 24 cities. KMC with 3 million points partitioned into 15 clusters. J2D using an array of size $5,000 \times 5,000$ with a block size of $1,000$. DLS on a $1,000 \times 2,500,000$ array, with the results located at index (100, 250000) and (350, 875000). CS on a $20 \times 20 \times 60 \times 15,000$ array, with the result located at index (8, 8, 24, 6000).

sion 7 and our implementation of eureka on top of the HJlib cooperative runtime version 0.1.5. The JVM configuration flags used were (`-XX:-UseGCOverheadLimit -Xmx65536m -XX:+UseParallelGC -XX:+UseParallelOldGC`). Each benchmark was configured to run using 32 worker threads and run for thirty iterations in six separate JVM invocations. The arithmetic mean of the best fifty execution times (from the hundred and eighty iterations) are reported. The execution times were computed using the `System.nanoTime()` method for the best resolution. Using the best execution time allows us to minimize the effects of JVM warm up, just-in-time compilation, and garbage collection.

Speculative Execution Benchmarks: The benchmarks are described in Figure 4.13. The benchmarks include some of our motivating examples, search bench-

Name	Execution Time (in seconds)					Ratio of Exec. Time				Abstract Operations ($\times 10^3$)				
	AF	FS	EX	CT	EU	AF:EU	FS:EU	EX:EU	CT:EU	AF	FS	EX	CT	EU
SLS	58.37	17.70	16.61	16.71	16.85	3.46	1.05	0.99	0.99	2,476	845	798	800	806
UTS	15.89	8.81	5.94	5.81	5.76	2.76	1.53	1.03	1.01	1,571	512	444	446	437
SUD	DNC	5.52	5.53	5.48	5.72		0.96	0.97	0.96		146	148	142	152
NQK	24.90	3.33	3.86	3.20	3.96	6.28	0.84	0.97	0.81	1,711	216	210	212	216
UTP	2.95	2.73	2.58	2.37	2.48	1.19	1.10	1.04	0.96	233	233	189	189	189
FLP	38.35	30.25	7.83	7.94	8.04	4.83	3.79	0.98	0.98	688	523	232	233	231
TSP	DNC	1.51	1.18	1.19	1.11		1.35	1.06	1.07		857	839	839	754
KMC	15.22	12.26	12.32	12.56	12.44	1.22	0.99	0.99	1.01	1,125	916	916	916	917
J2D	16.35	13.01	13.21	13.04	13.10	1.25	0.99	1.01	1.00	1,125	902	902	903	903
DLS-AND	169.67	50.94	47.67	48.15	47.87	3.54	1.06	1.00	1.01	500	172	164	162	163
DLS-OR	169.00	4.65	0.53	0.53	0.54	315.17	8.66	0.99	0.99	490	15	2	2	2
CS	7.33	7.36	7.44	7.55	2.86	2.56	2.57	2.60	2.64	360	360	360	360	135
						>4.15	1.53	1.08	1.06	>1,027	474	434	433	409
						Geometric Mean				Arithmetic Mean				

Figure 4.14 : Benchmark execution time metrics, DNC means Did Not Complete inside 30 minutes. Except SUD, all the benchmarks had a coefficient of variation (CoV, ratio of the standard deviation to the arithmetic mean) less than 2 percent for the execution time of each variant. For SUD the CoV was about 10 percent for each variant.

marks, game puzzles, greedy algorithms, branch and bound algorithms, and a stencil computation. We present empirical evaluation of our implementation of the EuPM (`EU`) relative to variants that: (a) provide no support for early termination of `async-finish` tasks (`AF`); (b) use function-scoped cancellation points for termination of speculative `async-finish` tasks[†] (`FS`); (c) use exceptions for termination of speculative `async-finish` tasks[‡] (`EX`); and (d) use `if` checks and `return` statements via cancellation tokens speculative `async-finish` tasks (`CT`). The results for execution time and productivity metrics are described below.

4.8.1 Execution Times Comparison

We compare the performance of the different `Eureka` patterns in the benchmarks. The results are summarized in Figure 4.14. The comparison with the `AF` versions shows that most of these benchmarks benefit from speculation. In fact, in some

[†] `if` checks and `return` happen only at the body of the `async`, not inside nested function calls.

[‡] Our implementation minimizes overheads as it does not terminate worker threads, and it does not fill the stack trace of the abort exceptions.

of the benchmarks (e.g. SUD, TSP) the non-speculative variant does not complete execution. In other benchmarks, e.g. SLS, UTS, NQK, FLP, the non-speculative versions perform higher numbers of abstract operations (e.g. comparisons, arithmetic operations, nodes visited, etc.) which reflects in larger execution time compared to the speculative variants.

In general, the benchmarks SLS – J2D use a single eureka pattern and the `EX`, `CT`, and `EU` variants perform similarly. `EU` performs much better than the `FS` variant since the `EU` variant, like `EX` and `CT`, can trigger task cancellation even inside nested function calls. Overall, the `EU` variants compete favorably with the other speculative variants (`EX` and `CT`). On most benchmarks, the `EU`, `EX` and `CT` variants perform within 5% of each other, both in terms of execution time and the number of abstract operations. This shows that our `EU` abstractions and different `Eureka` patterns do not add significant overhead in their implementations. Note that our implementation uses delimited continuations without modifying the VM; the performance of our implementation would be greatly improved by using native support for DeConts in the VM. Work by Stadler et al. [99] to provide such native support in a Java VM reported over two orders magnitude speedup on micro-benchmarks compared to a bytecode transformation approach. Additionally, we decided to exclude benchmarks that further highlight the limitations of the other approaches (e.g. inaccessible functions, user exception handlers) to allow a fairer comparison between all the approaches. These benchmarks would show our `EU` approach in "an even more" positive light.

The DLS benchmark uses binary composition of `EU Eureka`s and performs similarly to `EX` and `CT` confirming that no significant overhead is introduced by the composition. The CS benchmark is interesting as the hierarchical nature of the computation allows the `EU` variant to terminate other tasks searching on different leading indices. The `CT`, `EX`, and `FS` variants cannot implement such hierarchical information easily and end up doing redundant computation even after the goal element has been found. This causes them to perform as much work (in terms of abstract operations)

Name	Lines of Code					Cyclomatic Complexity					Development Effort ($\times 10^3$)				
	AF	FS	EX	CT	EU	AF	FS	EX	CT	EU	AF	FS	EX	CT	EU
SLS	72	75	79	78	69	1.66	1.77	1.88	1.88	1.55	11.22	12.16	14.84	12.97	10.80
UTS	76	84	88	87	81	1.50	1.70	1.80	1.80	1.60	7.63	11.66	12.74	12.15	9.16
SUD	86	94	98	97	92	1.60	1.80	1.90	1.90	1.70	15.62	21.55	23.49	22.61	18.99
NQK	81	87	94	93	85	1.60	1.70	1.80	1.80	1.60	16.12	18.90	22.91	20.72	17.48
UTP	85	101	105	104	91	1.70	1.81	1.90	1.90	1.54	11.63	19.84	24.30	21.34	13.69
FLP	115	115	116	115	108	1.91	2.00	2.08	2.08	2.00	62.89	64.42	65.47	65.97	70.86
TSP	89	101	105	104	99	1.60	1.80	1.90	1.90	1.54	22.28	33.79	35.83	35.28	27.80
KMC	115	127	128	127	135	1.38	1.69	1.69	1.69	1.46	46.57	51.24	55.98	51.24	56.85
J2D	146	152	156	155	148	1.64	1.78	1.85	1.85	1.53	111.70	116.83	127.58	119.18	104.36
DLS	80	85	89	88	79	1.88	2.22	2.33	2.33	1.66	16.80	19.46	22.42	19.99	17.40
CS	108	131	139	138	107	1.61	1.70	1.82	1.82	1.61	39.37	73.43	87.35	84.67	43.73
A. Mean	96	105	109	108	99	1.64	1.82	1.90	1.90	1.62	32.89	40.30	44.81	42.37	35.56
%age of EU	-3.75	5.30	9.41	8.41		1.63	12.25	17.76	17.76		-7.49	13.33	26.02	19.17	

Figure 4.15 : Productivity metrics for benchmark kernels. LoC was computed using `cloc` command while CC and DE were computed using the CodePro Analytix Eclipse plugin. LoC for common support code are not reported in the table, the arithmetic mean for support code LoC is 240.

as the non-speculative version and causing larger execution times than the `EU` version.

4.8.2 Productivity Metrics Comparison

The most commonly used software productivity metric is program size or lines of code (LoC) to compare programs that use the same language and coding standards. There are other quantitative evaluation techniques for productivity apart from measuring LoC. McCabe introduced the Cyclomatic Complexity (CC) metric [100] based on the control flow structure of programs. CC represents the complexity of the algorithm, and poorly designed solutions have high CC values. Halstead's metrics [101] are also well-known measure of software complexity. The Development Effort (DE) metric uses values computed from a bunch of other Halstead metrics. It represents the effort required to convert an algorithm to an actual code in a specific programming language.

We report values for LoC, CC, and DE for all our benchmark kernels in Figure 4.15. We compare the metrics for the `AF` variants with four variants (`FS`, `EX`, `CT` and `EU`) which implement different task cancellation strategies. Overall, the `EU` versions require less LoC, CC, and DE being at least 5%, 12%, and 13% better, respectively, than any

of the other speculative variants. More importantly, the percentage improvements are even larger when compared to the closest performing speculative variants – `EX` and `CT`. In addition, as explained in Section 4.3, the `EU` versions do not suffer from any of the drawbacks compared to the other speculative methods. On average, the `EU` solutions for the kernels are only slightly larger than the `AF` variants requiring three extra LoC, some extra DE (7.5%), while the CC is actually smaller. This shows that, for the benchmark kernels, minimal effort was required to transform the `AF` versions into speculative versions using our proposed model. In particular, the comparatively low value of DE for the `EU` variant also reflects positively upon the usability of the `Eureka` API. In summary, the `EU` solutions are more productive to implement than the `FS`, `EX`, and `CT` variants in terms of all three productivity metrics.

4.9 Summary

We introduced the Eureka parallel programming model (EuPM) that simplifies the expression of speculative parallel tasks in search and optimization algorithms. We have demonstrated the power of the EuPM as a mechanism for codifying various parallel eureka patterns. The constructs we propose simplify writing EuSCs and improve programmer productivity. Our implementation shows that the EuPM can also be implemented efficiently, especially with the need to terminate tasks cooperatively. Our performance results on benchmarks show that our implementation performs close to manual hand-coded versions while being shorter and less complex to write. We believe that our implementation techniques of the EuPM can easily be ported to other languages as well.

Chapter 5

Selectors: Actors with Multiple Guarded Mailboxes

When you drop your guard..., the acting process compensates.

Ben Kingsley

The Actor Model (AM) [15, 16, 17] for nondeterministic concurrency has recently gained repute, mainly due to the success achieved by its flagship language - Erlang [18]. With the success of Erlang in production settings, the AM has catapulted into the mainstream and there has been a proliferation of the development of Actor frameworks in popular sequential languages like C/C++ (CAF[19], Act++ [20]), Smalltalk (Actalk [21]), Python (Stackless Python [22], Stage [23]), Ruby (Stage [24]), .NET (Microsoft's Asynchronous Agents Library [25], Retlang [26]). The AM offers a promising approach for developing reliable concurrent systems with general nondeterministic behavior (as opposed to Eureka computations for which the nondeterminism is more limited). Actors provide a programming model that gives stronger guarantees about concurrent code such as data race freedom and location transparency when compared to traditional shared-memory-based abstractions such as fork-join tasks. The key idea is to encapsulate mutable state and use asynchronous messaging to coordinate activities among actors. This asynchronicity implies that the sender does not wait for a message to be received (processed) upon sending it; it immediately continues execution after issuing the send.

Despite being a model of concurrent computation, the AM is not a silver bullet -

not all concurrent programming problems are best solved by the AM. Synchronization mechanisms are needed in concurrent programming to control the order of message processing so as to preserve the integrity of objects [102]. However, for developers who are new to the AM, understanding and managing such synchronization and coordination in an asynchronous model might be harder than in the task-parallel model. Coordination patterns involving multiple actors are particularly difficult. The property of no shared state and asynchronous communication can make implementing coordination protocols harder than using established constructs like locks [27]. Solutions for such protocols to support synchronization constraints may require the actor to buffer messages and resend the messages to itself until the message is processed [28]. The resulting code is a petri dish of code that intertwines both algorithmic logic and synchronization constraints, without any separation of concerns. In addition, recycling of messages in the mailbox is expensive due to additional overhead from message sending and maintenance of internal buffers. There is, hence, a need for an abstraction to support synchronization and coordination mechanisms that integrate well with the AM.

In this work, we describe our extension to the AM to address this issue, which builds on our past work on integrating actors and task parallelism [30]. Our extension, called *selectors*, allows actors to have multiple mailboxes. We believe that this is a powerful extension, analogous to that of adding condition variables [103] to semaphores. While messages can be sent to any specific mailbox of the selector, each of these mailboxes are guarded. The guard refers to the ability to manage cooperatively which mailboxes are searched for processing the next message of the selector in response to the selector processing its current message. Selectors continue to encapsulate their local state and process one message at a time. Thus, the benefits of modularity from the AM are still preserved, and the data locality properties of the AM continue to hold. With selectors, the significant change to using regular actors is that along with the message, the `send` operation receives an additional argument,

the recipient mailbox name.

The selectors model allows us to simplify writing of standard synchronization / coordination patterns using actors such as *a*) synchronous request-reply [104], *b*) join patterns in streaming applications [105], *c*) supporting priorities in message processing [106], *d*) variants of reader-writer concurrency [107], and *e*) producer-consumer with bounded buffer [108]. Each of these patterns are further described in Section 5.4, Section 5.5, Section 5.6, Section 5.7, and Section 5.8, respectively in the following format. We first summarize the difficulties with an actor-based solution. Next, we present selector-based solutions to each of these patterns.

In Section 5.9, we provide performance and productivity results for the selector-based and actor-based solutions. To allow comparison with other performant JVM-based actor libraries, we developed a Scala API for our implementation of selectors and implemented all our benchmarks in Scala. Our experimental results for the benchmarks show selector-based solutions deliver significant improvements in performance compared to actor-based solutions without compromising productivity. Our productivity measurements are limited to comparison of the Java code of Habanero actors and selectors solutions as we could not find tools to measure Cyclomatic Complexity and Development Effort for Scala code. Our selectors implementation is in many ways an extension of Habanero actors, only comparing the Habanero solutions seems reasonable. In addition, since all the actor-based solutions use the same algorithm and have similar code, using Habanero actors as the representative actor-based library does not affect the validity of our productivity results.

5.1 Contributions

Our work on selectors makes the following contributions [109]:

- Introduce our extension to the actor model called selectors. We also describe our library-based implementation of selectors and evaluate performance against other modern actor frameworks that run on the Java Virtual Machine.

- We use selectors to express synchronization in an asynchronous and nondeterministic actor background, and to achieve it in an expressive, reusable and efficient manner. We present solutions to each of these above-mentioned synchronization / coordination patterns using selectors.

- We developed `Savina`, a standardized benchmark suite that represents various use-cases in actor-oriented programs and allows users to do an apples-to-apples comparison between different actor libraries [110]. The benchmarks in `Savina` are diverse, realistic and represent compute-intensive applications with various synchronization and coordination patterns. The benchmarks range from popular micro-benchmarks to classical concurrency problems to applications that demonstrate various styles of parallelism.

- We evaluate the performance of our library implementation of selectors on these benchmarks that exhibit such synchronization / coordination patterns. We compare our implementation against actor-based solutions using Scala, Akka, Jetlang, Scalaz, Functional-Java and Habanero actor libraries. Our experimental results for the benchmarks show that using selector-based solutions simplify programmability and deliver significant performance improvements compared to other actor-based solutions.

5.2 Background and Motivation

5.2.1 The Actor Model

Carl Hewitt et al. first defined the Actor Model (AM) in 1973 while researching on Artificial Intelligent (AI) agents [15]. Further work by Henry Baker [16], Gul Agha [17], and others added to the theoretical development of the AM. Actors provide a programming model that gives stronger guarantees about concurrent code when compared with the traditional shared-memory-based abstraction such as data race freedom and location transparency. The key idea is to encapsulate mutable

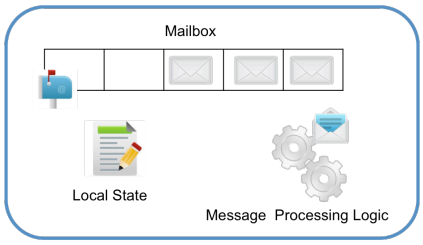

Figure 5.1 : Decomposition of an actor: a single mailbox, local state, and message processing logic.

state and use asynchronous messaging to coordinate activities among actors. This asynchronicity implies that the sender does not wait for the message to be received (processed) and it immediately continues its execution after sending the message.

An actor is the central entity in the AM, it has the capability to process incoming messages. As shown in Figure 5.1, every actor has its own *mailbox* for storing incoming messages, implemented as asynchronous, race-free, non-blocking queues. Other actors act as producers for messages that go into the mailbox. An actor also maintains local state which is initialized during creation. After creation, the actor is only allowed to update its local state using data from the messages it receives and from the intermediate results it computes while processing the message. The actor is restricted to process at most one message at a time making the actor's own code implicitly thread-safe. The actor's state can thus be modified safely from within its body without any other extra (synchronization or locking) effort.

There is no restriction on the order in which the actor decides to process incoming messages, thereby leading to non-determinism in actor systems. As an actor processes a message, it is allowed to change its state and behavior affecting how it processes the subsequent messages. While processing a message, an actor performs actions requested by the messages and sends messages to other actors, including themselves, to perform complex sequences of operations. In a program that employs multiple actors, each actor processes messages on a separate thread, allowing for parallel execution up to the number of active actors.

Limitations

The AM is not a silver bullet - not all concurrent programming problems are best solved by the AM. We address some common programming patterns here which are cumbersome to implement using the AM to motivate our extension to the AM. In Section 5.3, we introduce our extension called Selectors and describe how our extension of allowing actors to have multiple guarded mailboxes can be used to elegantly solve **each** of these patterns.

All messages between actors are asynchronous, meaning that the sender continues processing before any reply is received. Coupled with the asynchronous messaging philosophy is a lack of guarantee provided on which message is processed next by an actor. Typically an actor processes the next available message from its mailbox. An actor can enforce a criterion on the next message processed by the actor by recycling messages in the mailbox (i.e. by resending the message to itself). However, this strategy is expensive due to an increase in time in searching for and finding the next message to process from the mailbox.

Another strategy for enforcing the criterion mentioned above is using pattern matching [111], an abstraction commonly used in functional programming languages. The construct is like a `switch-case` statement but with the additional capability of describing the structure of multiple objects to match on in the `case` statement. For example, Scala actors [112] use the pattern matching strategy. For each message from the mailbox, the actor runtime attempts to match the pattern expressions from the actor's message processing body. If the current message reports no match, the process of finding a match repeats on another message from the mailbox until the runtime finds a matching pattern. Eventually, such a pattern is found for a message, and that message is processed by executing the block of code corresponding to the pattern. Processing a message can mutate the internal state of the actor causing the dynamic state of the pattern expressions also to change. Hence, the matching process has to be repeated for previously unsuccessfully matched messages from the

mailbox as the result of the pattern expression evaluations can have changed. Like the previous strategy of recycling messages, the pattern matching strategy is also expensive to implement due to repeated pattern expression evaluations on a message that may be necessary.

5.3 Selectors: Actors with Guarded Mailboxes

We have extended the basic AM and call our extension *Selectors*. Similar to an actor, we define a selector as an object that has the capability to process incoming messages. Selectors continue to encapsulate their local state and process one message at a time (as shown in Figure 5.2). Thus, the benefits of modularity from the AM are still preserved, and the data locality properties of the AM continue to hold. Selectors differ from actors in two main ways:

- Selectors have multiple mailboxes to receive messages, senders specify which mailbox a message is targeted to. The sender can be any entity: an actor or a selector. As with actors, sending messages to any of the mailboxes is a non-blocking operation. Messages can be concurrently added to different mailboxes without synchronization and thereby reducing contention.

- Each mailbox maintains an internal boolean guard that is used to *enable* or *disable* the mailbox while the selector is processing messages. The guard does not affect which mailbox can receive messages; it only affects which mailboxes are inspected to select the next message to be processed. Attempting to enable an active mailbox and to disable an inactive mailbox are considered to be no-ops, not errors. It is an error if all the mailboxes become disabled while no message is actively being processed*.

*Alternatively, we can treat this case like an `exit` operation (after which new messages can accumulate but will never be processed).

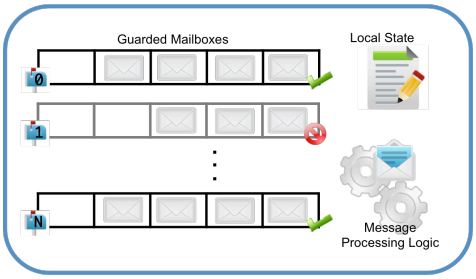

Figure 5.2 : Decomposition of a selector: guarded mailboxes, local state, and message processing logic.

We observe that a standard actor can just be viewed as a selector with a single mailbox.

Our extension to actors is inspired by condition variables [103] where a thread checks whether a condition is true before continuing its execution. The thread then waits for an event that changes the state of the condition variable and enables the waiting thread to continue execution. Similarly, selectors check conditions (guards) on their mailboxes and only process messages from *active* mailboxes. The selector waits for particular mailboxes to become active (state change) before considering messages from the activated mailboxes for processing. Logically, every condition variable is associated with one (or more) boolean condition expressions. Similarly, every mailbox is guarded with one (or more) boolean condition expressions that control when it is in an enabled or a disabled state.

Life Cycle of a Selector The life cycle of a selector is similar to that of an actor. As shown in Figure 5.3, a selector is in one of the following three states:

- *new*: An instance of the selector (including its mailboxes) has been created; however, the selector is not yet ready to process messages from its mailboxes. Other entities can send messages to the selector at any of its mailboxes. Initially, all the mailboxes of the selector are enabled.

- *started*: A selector moves to this state from the *new* state when it has been

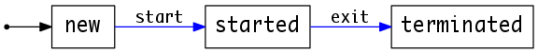

Figure 5.3 : Life cycle of a selector.

started using the `start` operation. It can now receive asynchronous messages and process messages from any active mailbox one at a time. During processing of a message, the selector can enable or disable any of its mailboxes. After a message is processed, another message can be selected from any active mailbox to be processed next. While processing a message, the selector should continually receive any messages sent to it without blocking the sender. There is no restriction on the order in which the selector decides to process incoming messages from active mailboxes, thereby leading to non-determinism.

- *terminated*: The selector moves to this state from the *started* state when it has been terminated and will not process any messages in its mailboxes or new messages sent to it. A selector signals termination by using the `exit` operation on itself while processing a message. Any messages sent to a terminated selector will be ignored. A terminated selector may not be restarted.

Sending messages to a Selector All messages sent to selectors are asynchronous, meaning that the sender continues processing before any reply is received. At its simplest, the `send` operation receives two arguments: the target mailbox name and the actual message to send. Selectors offer flexibility in determining the target mailbox of a message; it may be decided in two ways:

- *By the sender*: The sender can directly specify the target mailbox as an argument in the `send` operation. The sender may have received the mailbox name in a message or the name could be globally known.

- *By the recipient selector*: The recipient can abstract away the decision making

of which target mailbox to use from the sender. In such scenarios, the sender just invokes the `send` operation with the message (just like sending messages to actors). The selector can then introduce logic to introspect the message and its internal state to decide the recipient target mailbox.

A selector can choose to use a combination of both schemes where recipient mailboxes are decided by the sender for some messages while the recipient selects the target mailboxes for the remaining messages.

Declarative Guards Selectors can be further enhanced by introducing explicitly declared predicated guard expressions on mailboxes (making selectors more functional in nature). These guard expressions are *registered* for each mailbox after the selector instance has been created and may be updated later. The selector runtime ensures the mailboxes are enabled or disabled at the end of processing each message based on the result of evaluating the predicate. This avoids the use of imperative `enable` and `disable` operations on mailboxes. This style of using declarative guards also separates the message processing logic from the logic to enable or disable mailboxes. The trade-off is that the style introduces additional overhead of possibly redundant computations of the guard expressions for each mailbox. Section 5.8 presents an example use case where declarative guards can be used to simplify selector-based solutions.

5.3.1 Implementation

Actors, being a model of computation, can be used to emulate selectors. However, doing so would require hand-coded implementations of mailboxes as part of the actor's local state, and would require messages to be recycled or resent. On the other hand, actors can be considered a special case of selectors with just one mailbox that is always enabled. Hence, any actor-based solution to a pattern is also a valid selector-based solution. We chose to implement selectors using the lower-level constructs

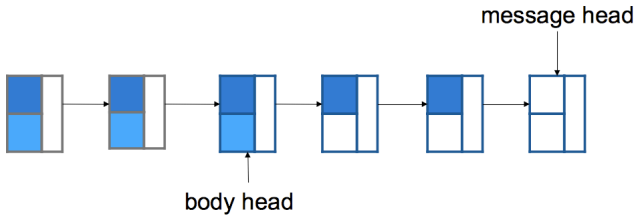

Figure 5.4 : The message head determines where the next message is stored in the mailbox. The body head determines which message will be processed next from this mailbox. The nodes on the left of the body head have already been processed and will be garbage collected.

described in our previous work on actors [30] and we can use the `finish` construct for detecting termination of selectors. Our implementation of selectors is an extension to the actors implementation in the Habanero-Java library [34] and we rely on Java's `ForkJoinPool` to manage the worker threads. Since we wish to compare against other actor frameworks implemented in Scala, we further expose the parallel constructs as a Scala library called Habanero-Scala [113].

The selector maintains a list of mailboxes; each mailbox represented by a concurrent linked-list (Figure 5.4). Each node in the list is a pair of values consisting of the actual message and a state field. The state field tracks whether this message is pending, is being actively processed or has been processed. As messages are sent to the selector's mailboxes, a chain of these pairs are built. As each worker picks up a message to process, the state field is updated accordingly to avoid duplicate processing of the message by other concurrent worker threads. Each node in the list is ready for garbage collection once the message inside the node is processed. The size of the mailbox is limited by the amount of available memory.

Once the selector has been started (via the call to `start()`), it registers a callback to schedule a task as soon as a message becomes available in an active mailbox. A worker thread eventually executes this task, the task then tries to process as many available messages from a particular selector by traversing messages in active mailboxes until there are no more messages to process. Once such a state is reached the

worker thread then goes ahead and tries to execute other ready tasks (selectors with pending messages) from its work queue. Before the worker thread is let go, if the selector has not terminated, a callback is registered that triggers an asynchronous task whenever a new message is sent to an active mailbox of the selector. In this scheme if a selector has no work to do, it does not consume any worker thread cycles as there are no thread-blocking operations. Hence, this solution scales well as worker threads are never blocked and are busy doing useful work in the processing of messages. Worker threads become idle only when there is no pending work (tasks in the work queue).

Number of Mailboxes In our model of selectors, the number of mailboxes used can be determined dynamically (based on the distinct mailbox names used in the `send` operations). However in our implementation, the number of mailboxes to use by a given selector is configured during initialization. Each mailbox in a selector is identified by a unique id, a non-negative integer starting at zero. A boolean array is also maintained to track the value of the guard condition whether a given mailbox is active or inactive. This boolean array simplifies the implementation of enabling or disabling mailboxes, making it a constant time operation. Using invalid mailbox ids in `send` operations results in a runtime error.

Message Selection Policy In our current implementation, messages are selected using a Multilevel Queue (MQ) scheduling algorithm [114, 115]. We exploit the fact that messages are permanently assigned to one of the mailboxes. The mailboxes are given priorities using the ordering id of the enumeration values. The MQ scheduling technique ensures that at any given time, the worker thread picks a message from the lowest numbered active mailbox of all those messages that are currently ready to be processed. A message from a lower-priority mailbox is scheduled for execution only if all higher-priority active mailboxes are empty or disabled. One drawback of the MQ policy is that messages in higher numbered mailboxes are vulnerable to

starvation and could wait an indefinite amount of time before being processed. An alternative solution, that can help address starvation issues, would be to randomly select messages from an active mailbox to process next. We observe that there is a possibility of a message remaining indefinitely in a mailbox's queue in pure actor models that do not require FIFO processing of messages.

Since our benchmarks (Section 5.9) are unaffected by starvation issues, we leave options for other message selection policies as possible future work. In particular, we wish to explore a Least-recently used Round-robin (LR) scheduling algorithm [114, 116]. In LR, the runtime will remember the least-recently used mailbox, *LRUM*, which provided a message to be processed. While searching for the next message to process, the runtime will start the search from the LRUM and return a message if that mailbox is enabled and contains an unprocessed message. Otherwise, the runtime will search for a message by cycling through the remaining mailboxes starting at the mailbox after the current LRUM. After finding a message from an active mailbox, the LRUM will be updated and the message will be scheduled for processing. Thus, LR will use round-robin scheduling with a dynamic time slice which is infinite until the mailbox becomes empty or disabled at which point the time slice becomes zero.

5.4 Synchronous Request-Response Pattern

The synchronous request-response pattern [104, 117] occurs when a requestor sends a request message to a replier system which receives and processes the request, ultimately returning a message in response. In the AM, the notion of synchronous request-response occurs when an actor sends another actor a message and stalls further processing of messages until it receives a reply to its message. As the AM is asynchronous, this pattern requires two asynchronous messages, a request and a response. While being conceptually elegant, this pattern is hard to implement efficiently, because the requestor actor's single mailbox must handle both the reply message and new messages sent to it from other actors. An option is to use pattern

```
1
2 class ReqRespActor extends ScalaActor {
3   def act() {
4     loop {
5       react {
6         case m: SomeMessage =>
7           // a case where we want a response
8           val req = new SomeRequest(m)
9           anotherActor.send(req)
10          receive {
11            case someReply: SomeReply =>
12              ...
13          }
14        case ... => // logic for other messages
15      } } } }
```

Figure 5.5 : Using Scala actors to solve the Request-Response Pattern using thread-blocking `receive`.

matching on the set of pending messages to implement the receive operation, but this can be expensive to implement due to the increase in time while searching for the next message to process from the mailbox.

The synchronous message passing style available in Scala actors (using `receive` at line 10 in Figure 5.5) provides programmers with a convenient way of doing messaging round-trips [112]. When the actor receives a message that is not matched, it will stay in the mailbox of the actor and is retried when a new `receive` block is entered. Using `receive` makes the actor heavyweight, since `receive` blocks the underlying thread while the actor is suspended waiting for a reply message. Another practical option to support this feature, to avoid complications in the processing of existing messages, is implemented using some notion of blocking explicitly and usually limits scalability. For example, the *ask* pattern (line 8 of Figure 5.6) uses a thread blocking `await` on the `future`'s value (line 9). The responding actor completes the `future` when it sends a response after processing the message thereby unblocking the previously mentioned thread.

It is cumbersome to support synchronous reply in a non-blocking manner since it requires preventing the actor from processing other messages in its mailbox. A non-blocking solution involves *stashing* the messages an actor receives until it finds

```
1
2 class ReqRespActor extends AkkaActor {
3   def receive = {
4     case m: SomeMessage =>
5       // a case where we want a response
6       val req = new SomeRequest(this, m)
7       implicit val timeout = Timeout(600 seconds)
8       val aFuture = ask(anotherActor, req)
9       val someReply = Await.result(aFuture, Inf)
10      ...
11    case ... => // logic for other messages
12  } }
```

Figure 5.6 : Using Akka actors to solve the Request-Response Pattern. The *ask* pattern uses a thread blocking await on the `future`'s value until the responding actor completes the `future` by sending a reply.

the reply message. Once the reply message has been found and processed, the stashed messages need to be *unstashed* into the actor's mailbox to resume processing of these messages. While a user can code this pattern manually, Akka provides the `become` and `become` operations and `Stash` trait to enable this pattern [118]. The issue with this solution is in the overhead introduced to maintain the stash of messages when the actor is in a *reply-blocked* state and the adding of messages back into the mailbox when the actor exits the reply-blocked state. There is additional overhead if the unstashed messages need to be prepended to the head of the mailbox.

Request-Response with Selectors With selectors, we can define two mailboxes one to receive *regular* messages and another one to receives *synchronous response* messages. Figure 5.7 has an example code snippet showing how selectors can be used with regular actors to handle the request-response pattern. The selector has two mailboxes: `REGULAR` used to receive normal messages and `REPLY` used to receive synchronous responses. Whenever a selector is expecting a synchronous response, it disables the regular mailbox (line 10) thereby ensuring that the next message it processes will be from its reply mailbox. When the selector disables the regular mailbox, we say the selector is in a *reply-blocked* state and is awaiting a response from the responding entity. When the message is processed by the responder, it

```
1
2 class ReqRespSelector extends Selector {
3   def process(theMsg: AnyRef) {
4     theMsg match {
5       case m: SomeMessage =>
6         // a case where we want a response
7         val req = new SomeRequest(this, m)
8         anotherActor.send(req)
9         // move to reply-blocked state
10        disable(REGULAR)
11      case someReply: SomeReply =>
12        // process the reply (from REPLY mailbox)
13        ...
14        // resume processing regular messages
15        enable(REGULAR)
16    } } }
17 class ResponseActor extends Actor {
18   def process(theMsg: AnyRef) {
19     theMsg match {
20       case m: SomeRequest =>
21         val reply = compute(m.data)
22         // send to response mailbox
23         sender().send(REPLY, reply)
24       ...
25    } } }
```

Figure 5.7 : Using Selectors to solve the Request-Response Pattern without blocking. In this example the responding entity is an actor and is sending the reply message to the `REPLY` mailbox (line 23).

replies by sending a message to the reply mailbox of the selector. The selector can then process this response message and enable the regular mailbox to move out of the reply-blocked state (line 15) and continue processing other messages sent to it.

5.5 Join Patterns in Streaming Applications

Since actors can be used to pipeline messages, they are a good fit for certain streaming applications. The pipeline parallelism can be exploited by connecting actors in a data flow chain to form producer-consumer pairs and ensuring FIFO order in processing of messages. In such a pipeline network, producers asynchronously propagate data to consumers as data becomes available. However, the join pattern [105, 119] where messages from two or more data streams are combined together into a single message is cumbersome to mimic using actors. The join is also commonly used in the *split-*

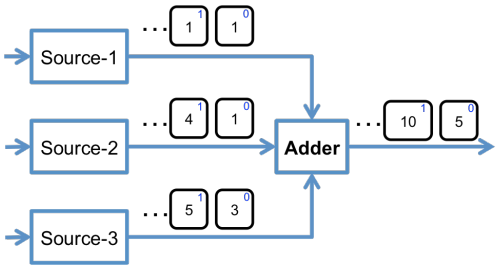

Figure 5.8 : Actor network simulating a join pattern. `Source-1`, `Source-2`, and `Source-3` are producers for data streams. The `Adder` actor aggregates corresponding data items from each of the three sources and sums them up.

join pattern where a parallel set of streams, which diverge from a common splitter and converge to a common joiner [120]. Traditional joins are blocking as they need to match inputs from each of the sources and wait until all corresponding inputs become available. An example of an aggregator is shown in Figure 5.8 where the adder actor is adding streams of corresponding values from three sources.

The lack of guarantee on which message is processed next, for example by taking the sender of the message into account, makes implementing this pattern troublesome. Complexity is also introduced due to the need for an additional dictionary to keep track of all items *in-flight* from the various sequence numbers. Supporting the join pattern also requires tagging messages with the source and sequence number information. This information is then used to aggregate the messages from the various sources in a dictionary indexed by the sequence number from each of the different sources. When items from all its sources for the oldest (lowest) sequence number is received, the aggregator actor can then reduce the items into a single value and forward it to the consumer in the network. The actor also needs to remove entries from the dictionary for sequences which have been aggregated to avoid memory leaks. Figure 5.9 represents such an implementation of the `AdderJoin` actor of Figure 5.8.

Akka provides support for the aggregator pattern [121]. The implementation allows `match` patterns to be dynamically added to and removed from an actor from inside the message handling logic. However, the implementation does not match the

```
1
2 // process join items using actor
3 class AdderJoinActor(...) extends Actor {
4   val items=Map[Int,Array[Int]]()
5   var srcMatched=0
6   def process(theMsg: AnyRef) {
7     theMsg match {
8       case im: ItemMessage =>
9         if (!items.containsKey(im.seqNum)) {
10          // initialize to store current seq items
11          items.put(im.seqNum, Array[Int](numSrcs))
12        }
13        val seqItems = items.get(im.seqNum)
14        seqItems(im.sourceId) = im.intValue()
15        srcMatched += 1
16        if (srcMatched == numSrcs) {
17          val joinResult = computeJoin(seqItems)
18          nextInChain.send(joinResult)
19          // reset locals
20          srcMatched = 0
21          // avoid memory leak
22          items.remove(im.seqNum)
23        } } } }
```

Figure 5.9 : Using actors to solve the Join Pattern problem of Figure 5.8. The aggregator actor version maintains a dictionary to store items from each sequence, hence the messages need to be tagged by the sequence number (`seqNum`).

sender of the message which is required to support the join pattern. As such, enforcing the requirement that one message is received from each data stream source requires additional logic.

Join Pattern with Selectors Supporting the join pattern requires tagging the messages with source and sequence numbers in an actor-based solution. The requirement for sequence numbers goes away if messages from a sender are processed in the order in which they were sent as the aggregator can implicitly build the sequence numbers for each sender. However, requiring that corresponding items from each source are paired up still requires additional logic such as maintaining a dictionary of received items for each sequence. Figure 5.10 presents two selector-based solutions for the example problem introduced in Figure 5.8. The solutions require that the senders send their messages in the correct mailboxes, this can be enforced in one of two ways: *a*) wrapping the send logic in the selector to forward messages from sources to

```
1
2 // process items in any order
3 class AdderAnyOrder(...) extends Selector {
4   var items = Array[Int](numSrcs)
5   var srcMatched = 0
6   def process(theMsg: AnyRef) {
7     theMsg match {
8       case im: ItemMessage =>
9         items(im.sourceId) = im.intValue()
10        // disable the current mailbox
11        disable(im.sourceId)
12        srcMatched += 1
13        if (srcMatched == numSrcs) {
14          val joinResult = computeJoin(items)
15          nextInChain.send(joinResult)
16          // reset locals
17          items = Array[Int](numSrcs); srcMatched = 0
18          // enable all mailboxes for next seq
19          enableAll()
20      } } } }

22 // process items in round-robin order
23 class AdderRoundRobinOrder(...) extends Selector{
24   var items = Array[Int](numSrcs)
25   var srcMatched = 0
26   // expect item from first source
27   disableAllExcept(0)
28   def process(theMsg: AnyRef) {
29     theMsg match {
30       case im: ItemMessage =>
31         items(im.sourceId) = im.intValue()
32         // disable the current mailbox
33         disable(im.sourceId)
34         srcMatched += 1
35         if (srcMatched == numSrcs) {
36           val joinResult = computeJoin(items)
37           nextInChain.send(joinResult)
38           // reset locals
39           items = Array[Int](numSrcs); srcMatched = 0
40         }
41         //enable round-robin mailbox for next seq
42         enable(srcMatched)
43     } } }
```

Figure 5.10 : Using Selectors to solve the Join Pattern problem of Figure 5.8. The aggregator selector versions (`AdderSelectorAny` and `AdderSelectorRoundRobin`) maintain one mailbox for each source. For simplicity we assume sources are identified by consecutive integers starting at 0.

specific mailboxes, or *b*) configuring the sources with specific mailbox names during initialization so that source entities send only to specific mailboxes.

In the first solution, the `AdderAnyOrder` selector accepts an item (message) from any of the pending sources in the current sequence and does not enforce any order

in the receipt of items from sources. As an item is received from a source, the corresponding mailbox is disabled (line 11) to disallow processing items from that source which do not belong to the current sequence. Hence, the set of active mailboxes shrinks and represents the set of sources which do not have a representative item in the current sequence. When items from all sources have been received for the current sequence (line 13), the result is computed (line 14) and forwarded to the next entity in the network (line 15) and all mailboxes are enabled (line 19) to start processing the next sequence.

In the second solution, the `AdderRoundRobinOrder` selector initially disables all the mailboxes except the first mailbox (lines 27) in anticipation of processing an item from the first source. As each item is received, the current mailbox is disabled (line 33) and the mailbox of the next source in round-robin order is enabled (line 42). When one message has been received from each of the sources in the current sequence (line 35), the result is computed (line 36) and forwarded to the next entity in the network (line 37). The currently active mailbox then is the mailbox corresponding to the first source so that processing of items from the next sequence begins.

5.6 Supporting Priorities in Message Processing

In the priority pattern [106], messages with a higher priority are processed before those with a lower priority even if they were sent earlier. This pattern is useful in applications that offer different quality of service guarantees to individual clients where messages from *important* clients are given higher priorities. The priority pattern is also useful while implementing recursive data structure traversal algorithms where nodes deeper in the recursion tree are expected to produce results with higher probability. When solving divide-and-conquer style algorithms using *master-worker* pattern with actors, the master actor divides the work among the worker actors for load balancing and the worker actors report newly generated subproblems back to the master. The master should schedule larger subproblems for processing with higher

priority as these subproblems should give rise to more work for the workers and improve the available parallelism. Normally actors do not support priorities while processing messages. Mimicking priorities in an actor based solution is quite cumbersome as messages sent to actors are stored in a single mailbox. One actor-based solution is to use a priority queue to store messages in the mailbox and require the user to provide some means of comparing the messages sent to the actor. However, this approach adds to overhead in the concurrent mailbox implementation.

Message Priorities in Selectors With selectors, we can support priorities for message processing non-intrusively without changing the message processing body (i.e. the body of the `process()` method). The solution relies on the fact that selectors have multiple mailboxes, each mailbox is used to store messages of a given priority. This scheme requires that messages be categorized by priority. The categorization enables determining the target mailbox for the message in the `send` operation and can be enforced either by the sender or by the recipient selector. The mailboxes are then sorted by priority and the selector configured to always process an available message from the highest priority message. For example, consider a selector with `MAILBOX-0`, `MAILBOX-1`, `MAILBOX-2` and so on. At the end of each message processing cycle, the selector checks the mailboxes in their priority order for available messages to process. This ensures the highest priority message is processed before any lower priority ones. The solution allows supporting as many priority levels as deemed necessary by the problem in hand. Note that we do not rely on enabling or disabling particular mailboxes to support priorities.

5.7 Reader-Writer Concurrency

In the ReadersWriters problem [122, 107], there are multiple entities accessing a resource, some reading and some writing. In addition, there is the natural constraint that no entity may access the resource for reading or writing while another process is

in the act of writing to it. There are two additional variants:

- The first readers-writers problem: the constraint is added that no reader request shall be kept waiting if the resource is currently opened for reading.
- The second readers-writers problem: the constraint is added that no writer request, once added to the message queue, shall be kept waiting longer than absolutely necessary.

Messages do not have dynamic priorities in actors; this makes it hard to support the variants where messages are processed out of *natural* order. In addition, since message processing is serialized in actors, implementation of truly concurrent data structures with actors is non-trivial. As an example, consider a group of actors of which one is a trivial dictionary actor (example from [123]) that responds to read and write requests. The other actors are clients of the dictionary: one actor does updates (writer), while the others only consult the dictionary (readers). The implementation of the dictionary with actors is easy because the programmer does not need to be concerned with data races: reads and writes to the dictionary are ensured to be executed in mutual exclusion. However, when the number of readers increases the resulting application performs badly precisely because of the benefits of serial execution (lack of data races) of requests to the dictionary actor: there are no means to process read-only messages in parallel and thus the dictionary actor becomes the bottleneck.

Reader-Writer Concurrency with Selectors We extend previous work on intra-actor parallelism [30] on actors to support asynchronous tasks with selectors. The `async` keyword is used to create a parallel task inside a selector's message processing body to process read requests in parallel. Figure 5.11 shows the template for a reader-writer selector. The `DataDrivenControl (DDC)` construct [124] is used to notify when all current in-flight read tasks have completed (inside the `decrementReaderAndSetDdc()` method). We maintain a **single** mailbox for both the write messages and the read

messages (i.e. `WRITE` and `READ` refer to the same mailbox). Once a read message is processed, it bumps a counter that tracks the *in-flight* readers (in line 13) and spawns an asynchronous task to process the read message (in line 17). Once this task completes the counter is decremented to notify the selector that there is one less active reader task (in line 20). If there are no in-flight readers we can process a write request immediately (line 26). Otherwise, when a write message comes in, we disable all mailboxes thus preventing the selector from processing any subsequent messages (line 28). Note that disabling all the mailboxes is not an error as the write message is still being processed. We wait for the *in-flight* readers to complete and register a callback to trigger the asynchronous processing of the write message (line 30). Once the write message processing is completed, we can enable all the mailboxes (in line 33) and the selector continues processing subsequent messages.

The scheme presented solves the readers-writers problem where the messages are processed in the order they appear in the mailboxes. This scheme can further be fine-tuned to solve both the first readers-writers problem and the second readers-writers problem using the support available for priorities in message processing Section 5.6. No change is required to the `process()` method from Figure 5.11. For the two variants we need to maintain **two** mailboxes, one for write messages and one for read messages (i.e. `WRITE` and `READ` now refer to different mailboxes). Each write message then ends up disabling all the mailboxes before starting and enabling all the mailboxes after completing the processing of the write message. We only need to add support for priorities to these mailboxes. For the first readers-writers problem, the read mailbox gets higher priority than the write mailbox ensuring all available read messages are processed whenever they appear in the mailbox. For the second readers-writers problem, the write mailbox gets higher priority than the read mailbox ensuring writes are processed as soon as they appear in the mailbox (after *in-flight* readers have completed). As a result we can now support three variants of the reader-writers problem: *a*) arrival order (AO); *b*) first readers-writers (RF); and *c*) second

```
1
2 class ReadWriteSelector extends Selector {
3   def sendWrite(theMsg: AnyRef) {
4     send(WRITE, new WriteMessage(theMsg));
5   }
6   def sendRead(theMsg: AnyRef) {
7     send(READ, new ReadMessage(theMsg));
8   }
9   def process(theMsg: AnyRef) {
10    theMsg match {
11      case ReadMessage(messageBody) =>
12        // another reader tasks is becoming active
13        int activeReaders = incrementReader()
14        if (activeReaders == 1) {
15          readersDdcRef.set(new DDC())
16        }
17        async { // process read asynchronously
18          processRead(messageBody)
19          // notify that reader is no longer active
20          decrementReaderAndSetDdc()
21        } }
22      case WriteMessage(messageBody) =>
23        val readerDdc = readersDdcRef.get()
24        if (readerDdc.valueAvailable()) {
25          // no readers active, safe to process
26          processWrite(messageBody)
27        } else {
28          disable(READ, WRITE) // disable mailboxes
29          // wait for in-flight readers to complete
30          readerDdc.addResumable(() => {
31            processWrite(messageBody)
32            // resume processing read/write messages
33            enable(READ, WRITE)
34          })
35        } } }
```

Figure 5.11 : Using Selectors to solve the Reader-Writer Concurrency problem. For simplicity concurrency constraints elided in the code snippet.

readers-writers (WF).

5.8 Producer-Consumer Pattern

The producer-consumer with a bounded-buffer is a classic example of a multi-process synchronization problem [103, 108]. In this problem, producers push work into the buffer as it is produced while consumers pull work from the buffer when they are ready. In an actor-based solution to this problem, producers, consumers and the buffer are modeled as actors. The buffer actor acts like a manager and needs to keep track of at least the following scenarios: *a*) whether the data buffer is full or empty; *b*) when

consumers request work from an empty buffer, the consumers are put in a queue until work is available; *c*) when producers are ready to produce data and the buffer is full, the producers are put in a queue until space is available in the data buffer; *d*) notify producers to produce more work when space becomes available in the data buffer. The buffer actor needs to maintain additional queues for the available consumers and producers since there is no way to disable the processing of particular messages and this convolutes the logic of the buffer actor. A simpler scheme where messages from producers or consumers were avoided from being processed when the data buffer is full or empty, respectively, would greatly simplify the buffer actor implementation. Using the pattern matching option remains expensive to implement due to increase in time while searching for the next message to process from the mailbox.

Producer-Consumer with Selectors In a selector-based solution to this problem, we can maintain a selector for the buffer while the producers and consumers continue to be modeled as actors. The buffer selector maintains two mailboxes, one to receive messages from producers and another to receive messages from consumers. Then we can disable processing messages from consumers when the buffer becomes empty, and disable processing messages from producers when the buffer becomes full. Additionally, the buffer selector can provide declarative guards to `enable` or `disable` the mailboxes thus easing programming effort. The `guard` operation is used to register a boolean guard condition on a given mailbox. After each message is processed, the guard for each mailbox is evaluated and the mailbox is `enable`d or `disable`d depending on whether the guard evaluated to `true` or `false`, respectively.

Figure 5.12 presents such a solution that uses declarative guards to isolate the message processing logic from the logic to enable or disable mailboxes. Whenever data is available in the buffer, the mailbox for messages from consumers is enabled (line 8) to process consume request messages. Whenever a data item is consumed by a consumer, the buffer is guaranteed space and the mailbox for messages from

```
1
2 class BufferSelector extends DeclarativeSelector {
3  def registerGuards() {
4   // disable producer msgs if buffer might overflow
5   guard(MBX_PRODUCER,
6    (theMsg) => dataBuffer.size() < thresholdSize)
7   // disable consumer msgs when buffer empty
8   guard(MBX_CONSUMER,
9    (theMsg) => !dataBuffer.isEmpty())
10 }
11 def doProcess(theMsg: AnyRef) {
12   theMsg match {
13     case dm: ProducerMsg =>
14       // store the data in the buffer
15       dataBuffer.add(dm)
16       // request producer to produce next data
17       dm.producer.send(ProduceDataMsg.ONLY)
18     case cm: ConsumerMsg =>
19       // send data item to consumer
20       cm.consumer.send(dataBuffer.poll())
21       tryExit()
22     case em: ProdExitMsg =>
23       numTerminatedProducers += 1
24       tryExit()
25   } } }
```

Figure 5.12 : Using Selectors to solve solve the Producer-Consumer with Bounded-Buffer Pattern. The Buffer selector maintains two mailboxes, one to receive messages from producers and another to receive messages from consumers. The use of declarative guards separates the `enable` and `disable` logic of mailboxes into the guard registration method, `registerGuards`.

producers is enabled to process messages from producers (line 5) to fill the populate the buffer with data. This scheme avoids having to maintain separate collections of available producers and consumers in a purely actor-based solution and simplifies the logic of the buffer selector.

5.9 Experimental Results

In this section, we focus on the performance (Section 5.9.1) and productivity (Section 5.9.2) results of our implementation of selectors. To allow performance comparison with other JVM-based actor libraries, we developed a Scala API for our implementation of selectors and implemented all our benchmarks in Scala. Our productivity measurements are limited to comparison of the Java code of Habanero actors

and selectors solutions as we could not find tools to measure Cyclomatic Complexity and Development Effort for Scala code.

5.9.1 Performance Evaluation

The actor libraries used for comparison with our implementation, Habanero Selector (HS), all run on the JVM. The libraries are: Scala actors (SC) [112], Akka (AK) [118], Functional Java (FJ) [125], Jetlang (JL) [126], Scalaz (SZ) [127], and Habanero Actors (HA) [34, 113, 30]. SC provides event-based actors that allow multiple actors to run on a thread. AK is a framework for building event-driven applications on the JVM and has support for highly performant lightweight actors. SC has been deprecated since version 2.10 and replaced by AK in the standard distribution of Scala since version 2.11.0. FJ is an open source Java library for applying functional programming concepts (including concurrency abstractions) and is intended for use in production applications. JL provides a low-level messaging API in Java that can be used to build actors with the responsibility of ensuring the single message processing rule delegated to the user. SZ has an actor implementation that has a simple API and minimizes latency and maximizes throughput of message passing. HA API is inspired by SC event-based actors. However, it does not use exceptions to support control flow and uses a push-based linked-list implementation using Java's `AtomicReference` for its mailbox. HS is our implementation of selectors using the mailbox implementation from HA. All actor implementations of each benchmark use the same algorithm and mainly involved renaming the parent class of the actors to switch from one implementation to the other. All implementations use the pattern matching construct (instead of `if-then-else` and `instanceof` checks) to represent the message processing body (MPB) and hence share the same overheads for the MPB. Similarly, all actor solutions use the same data structures for the user-written code of the benchmarks. We did this to ensure a fair comparison of the internals of the different libraries.

The benchmarks were run on a 12-core (two hex-cores) 2.8 GHz Intel Westmere

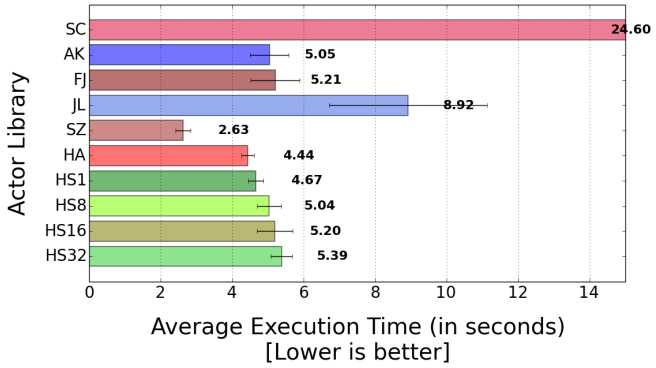

Figure 5.13 : The `Fork-Join` benchmark using 60 actors, each actor was sent 400,000 messages in its mailbox. For the selector versions, the messages were sent round robin to each of the mailboxes.

SMP node with 48 GB of RAM per node (4 GB per core), running Red Hat Linux (RHEL 6.2). Each core has a 32 kB L1 cache and a 256 kB L2 cache. The software stack includes a Java Hotspot JDK 1.8.0, Habanero-Java library 0.1.3 (contains both actors and selectors), Scala 2.11.0, Akka 2.3.2, FunctionJava 4.1, Jetlang 0.2.12, and Scalaz 7.1.0-M6. Each benchmark was configured to run using thirteen worker threads (the main thread gets blocked after initialization waiting for the computation to complete). We used the same JVM configuration flags (`-Xmx16384m -XX:-UseGCOverheadLimit -XX:+UseParallelGC -XX:+UseParallelOldGC`) and each benchmark was run for twenty iterations in six separate JVM invocations. The arithmetic mean of the best fifty execution times (from the hundred and twenty iterations) are reported to minimize effects of JIT and GC overheads from the reported results. In the bar charts, the error bars represent one standard deviation of the fifty execution times.

Message Throughput (ForkJoin) Microbenchmark

The results of the actor variant of the Java Grande Forum `Fork-Join` benchmark [74] are shown in Figure 5.13. This microbenchmark measures messaging throughput, the rate at which messages are processed by the implementation. Each actor does a

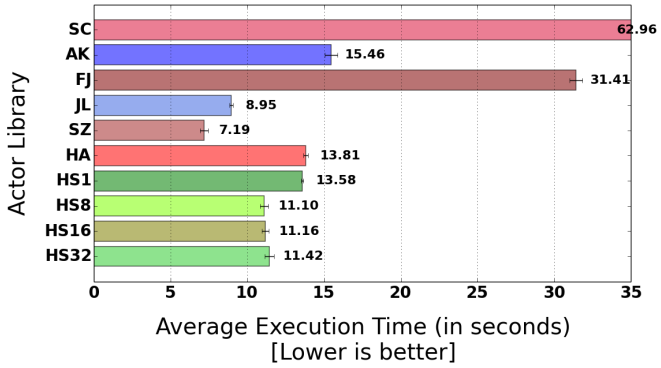

Figure 5.14 : The Chameneos benchmark was run with 500 chameneos (actors) constantly *arriving* at a mall (another actor). There are 8,000,000 meetings between chameneos orchestrated at the mall.

minimal amount of work processing one message, and the actor processes a total of N messages before it terminates. Among the actor implementations, SZ, AK, FJ, and HA versions have competitive performance while JL and SC are slightly slower. HS shares the same implementation scheme as HA, but differs in the way messages are searched from the mailboxes and performs slower than HA. In effect, this slowdown reveals the overhead of message lookup due to the introduction of multiple mailboxes and the checking of guards compared to the Habanero actors implementation. Comparing the single mailbox version (HS1) against HA reveals this overhead to be at about 5%. As the number of mailboxes increases, this overhead also increases which is why the HS versions with 8, 16 and 32 (HS8, HS16 and HS32 respectively) mailboxes perform worse than the 1 mailbox version.

Mailbox Contention (Chameneos) Microbenchmark

The `Chameneos` microbenchmark, shown in Figure 5.14, measures the effects of contention on shared resources while processing messages. We obtained the original SC implementation from the public Scala SVN repository [128]. The benchmark involves all *Chameneos* constantly sending messages to a mall actor that coordinates which two Chameneos get to meet. Adding messages into the mall actor's mailbox serves

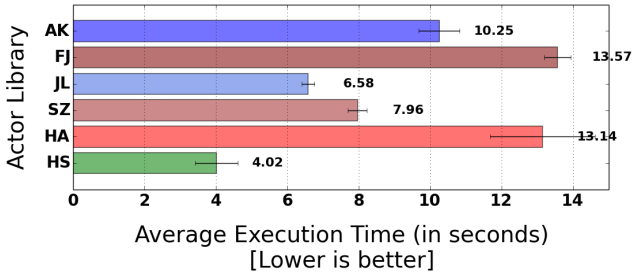

Figure 5.15 : Results of the LogisticMap benchmark using 150 term actors/selectors and 150 ratio actors. Each term actor is responsible for computing 150000 terms. The AK version, which uses blocking ask pattern, runs in over 46 seconds. SC version is not shown on the graph. The SC version, which uses blocking `receive`, runs in over 300 seconds. The SC version, which uses (non-blocking) manual stashing, runs in over 100 seconds.

as a contention point, and stress tests the concurrent mailbox implementation. The SC version pays the penalty of generating exceptions to support control flow in its `react` construct. On the other hand, the AK version is competitive with HA, JL, and SZ. There are no guards applied to the mailboxes in the HS* versions. The messages in the mall selector are processed from the first mailbox before searching the second mailbox for messages and so on, this is not necessarily the order in which the messages were received by the selector. The HS version with one mailbox (HS1) has similar performance as HA as they are effectively similar implementations. The HS* versions with more than one mailbox benefit from having multiple mailboxes as it reduces the synchronization contention while Chameneos actors concurrently send messages to the mall selector. In general, increasing the number of mailboxes improves performance of this microbenchmark. Eventually, the overheads of searching messages should start dominating and having a higher number of mailboxes should start giving poorer performance (HS32 performs slightly poorer than HS16).

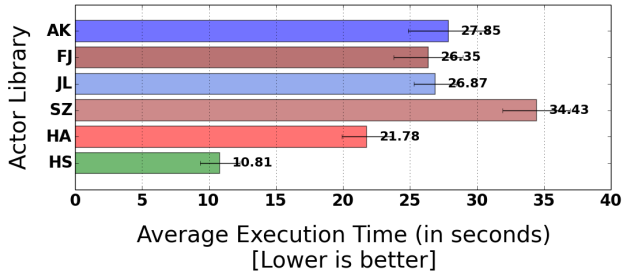

Figure 5.16 : Results of the Bank Transaction benchmark using 1,000 bank accounts and 10 million transactions. The SC version, which uses thread-blocking `receive`, runs in over 200 seconds and is not displayed in the graph. The SC version, which uses manual stashing of messages runs in over 100 seconds.

Synchronous Reply Benchmarks

We created two synthetic benchmarks to measure the performance of selectors against other actor implementations for the synchronous request-response pattern. The first benchmark computes the Logistic Map [129, 130] using a recurrence relation $x_{n+1} = rx_n(1 - x_n)$. In the benchmark there are three classes of actors: a manager actor, a set of term actors, and a set of ratio actors. The ratio actors encapsulate the ratio r and know how to compute the next term given the current term x_n. The term actors require a synchronous reply from the ratio actor before they update their value of x and, only then, process the next message from the master to compute the next term in the series. In the selector-based solution, the term actors are instead selectors and the ratio actors always send messages to the workers in their reply mailboxes. Figure 5.15 displays the results of this benchmark while comparing the selector-based solution against non-blocking actor-based solutions. The HS solutions is at least 1.6× faster than any of the actor solutions. Note that our solution for the AK version uses a custom extension that allows individual unstashing of messages, by default the Akka library only allows `unstashAll` which introduces a lot more overhead.

The second benchmark performs bank transactions between accounts by updating the balances atomically (i.e. the account does not process other messages while

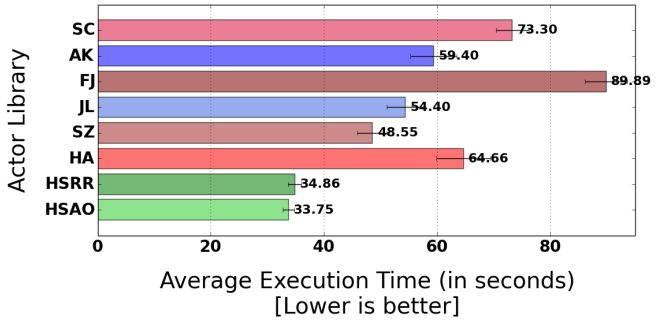

Figure 5.17 : Results of the Filter Bank benchmark results configured to use 8-way join branches. The input used 300, 000 data items and 131, 072 columns.

a transaction is in flight). The source account first decrements its balance and then needs to synchronously wait for the recipient actor to complete participating in the transaction by incrementing its balance with the same amount. During a single transaction, the source account needs to wait for the recipient actor to complete any active transactions it is participating in. The performance results for the benchmark are displayed in Figure 5.16. SC, AK, JL, SZ and HA are all based on the same principle of stashing and unstashing individual messages in the mailbox. The HS solution uses the selector-based technique described in Figure 5.7 that relies on enabling and disabling mailboxes. In addition, the `REPLY` mailbox is looked up first to allow messages participating in an *active* transaction to be processed with a preference (see Section 5.6). This allows the HS version to be over 2× faster than the actor versions.

Split-Join Benchmark (Filter Bank)

We use the Filter Bank benchmark ported from StreamIt [120] to quantify the performance of the join pattern. Filter Bank is used to perform multi-rate signal processing and consists of multiple pipeline branches. On each branch the pipeline involves multiple stages including multiple delay stages, multiple FIR filter stages, and sampling. Since Filter Bank represents a streaming pipeline, it can be implemented using actors or selectors. The Branches stage involves a split-join to combine the results of individ-

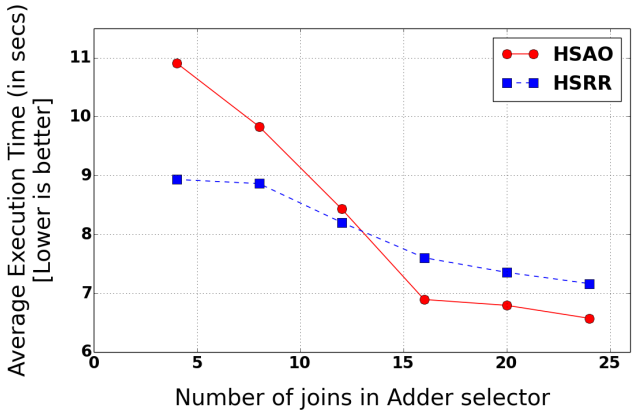

Figure 5.18 : Results of the AdderJoin microbenchmark, configured to add 24 million numbers.

ual Bank stages. This join can be efficiently implemented using selectors as there is no need to maintain a dictionary to track each sequence arriving from the different banks. Figure 5.17 compares the performance of two selector implementations (HS-AnyOrder and HS-RoundRobin described in Figure 5.10) of the Filter Bank benchmark against actor implementations. The selector versions are at least 28% faster than the actor implementations in AK, SC, FJ, JL, SZ, and HA as overheads from maintaining the dictionary do not exist. The HS-AnyOrder (HSAO) version with selectors is slightly faster than the HS-RoundRobin (HSRR) version as expected with a lack of order guarantee. On benchmarks where there are more joining edges (Filter Bank has only 8), the unordered versions are expected to perform even better.

AdderJoin Microbenchmark

We implemented the Adder-Join microbenchmark (similar to Figure 5.8) where a single entity randomly populates messages into the different mailboxes of the `Adder` selector. This enables us to stress test the two implementations (HSRR and HSAO). The results are displayed in Figure 5.18. We see that the HSRR variant performs better than the HSAO variant for 4-way and 8-way split-joins when using 24 million numbers to add. This is because enforcing the round-robin order simplifies finding the

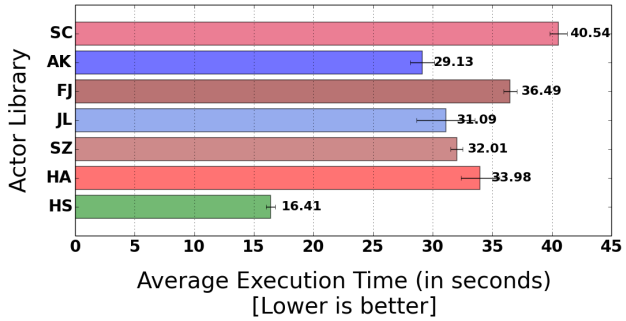

Figure 5.19 : Messages with Priorities: **NQueens - Find 1.5 million solutions**. Board size 15. Sequential threshold 5. 24 workers.

next message to process and helps avoid redundant search on other mailboxes. The performance is close between the two variants for 12-way splits. The HSAO versions starts performing clearly better from 36-way splits as the round-robin evaluation order enforces an order which constrains the throughput. HSAO version benefits from processing messages for a given sequence out of order and hence earlier compared to the round-robin version.

Message Priority Benchmarks

We use variants of the NQueens and A* benchmarks to measure the performance of the message priority pattern. In the NQueens benchmark (Figure 5.19), the goal is to find the first 1.5 million solutions on a board of size 15×15. The divide-and-conquer style is used with master-worker style actors/selectors, the workers report solutions to the master. The master requests the workers to terminate as soon as the 1.5 million solutions are found. Subproblems deeper in the recursion tree are processed with higher priority by the workers. These priorities help in guiding the execution to processing the request which is likely to yield a solution earlier compared to random processing of messages. The AK solution that uses a priority queue comes closest to the HS solution, but the overhead from maintaining the concurrent priority queue limits performance. All the other actor solutions use regular mailboxes and

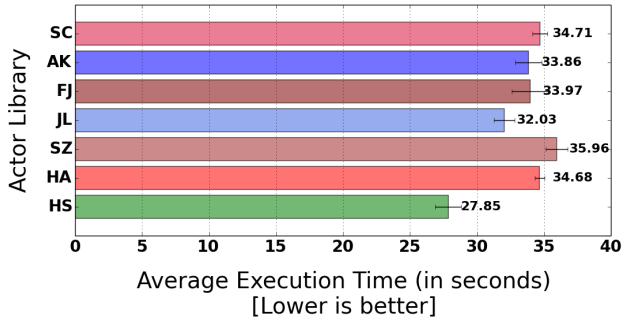

Figure 5.20 : Messages with Priorities: **A* Search**. Grid size 150 × 150 × 150 with 24 workers.

do not support message priorities, they rely on the message order provided by their respective implementations. The HS solution is at least 1.7× faster than any of the actor-based solutions. Some of the performance gain also comes from being able to process the worker termination signal earlier and avoiding performing redundant work once the required number of solutions have been found.

In the A* benchmark (Figure 5.20), a randomly generated 3-D grid is searched for a path to a target node from an initial source node. The heuristic of euclidean distance to the goal node is used as the priority to promote processing of nodes closer to the goal node to find the solution. The JL solution performs best among the actor-based solutions (faster than the AK priority mailbox version). The HS solution is still about 13% faster than JL and over 17% faster than the other actor-based solutions. The main benefit in this case comes from attempting to process nodes closer to the target node and hence increasing the probability of finding the solution quicker.

Reader-Writer Benchmarks

We use a concurrent dictionary (CD) (Figure 5.21) and a concurrent sorted linked-list (CSLL) (Figure 5.22) data structure benchmarks to measure the performance of selectors versus actor-based solutions for reader-writer concurrency. We mix read (lookup by item for CD and lookup by element in CSLL) and write (key-value pair

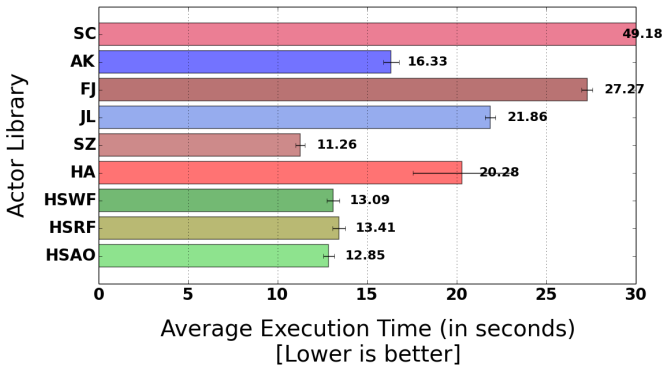

Figure 5.21 : Reader/Writer Concurrency: **Dictionary**. 24 worker actors with a total 400K requests per worker. 10 percent of the requests are write requests.

`put` for CD and element addition for CSLL) requests into the actor/selector representing the concurrent data structure. Pure actor solutions exhibit no concurrency, whereas the HS and HA solutions exhibit intra-actor concurrency. HS solutions allow concurrency from read requests based on AO and also support the RF and WF policies. The intra-selector parallelism in HS* solutions in general allow them to execute much faster than the actor-based solutions. The read and write operations in the CD benchmarks take $O(1)$ time while in the CSLL benchmark they take $O(N)$ time, this affects which of the RF and AO variants perform better. Note that creating and scheduling additional tasks for intra-actor and intra-selector parallelism involves some overhead. When read operations involve comparatively fewer operations and offer a fine granularity of work, we cannot entirely overcome the tasking overhead via computation. As a result, in the CD benchmark which does $O(1)$ work in read operations, the SZ version which has low mailbox contention for the dictionary actor (as seen in Section 5.9.1) can perform competitively with the HS* versions.

Producer Consumer with Bounded Buffer Benchmark

In the actor-based solution to the producer-consumer with bounded-buffer, the buffer actor is like the mall actor in the Chameneos benchmark (Section 5.9.1), so this bench-

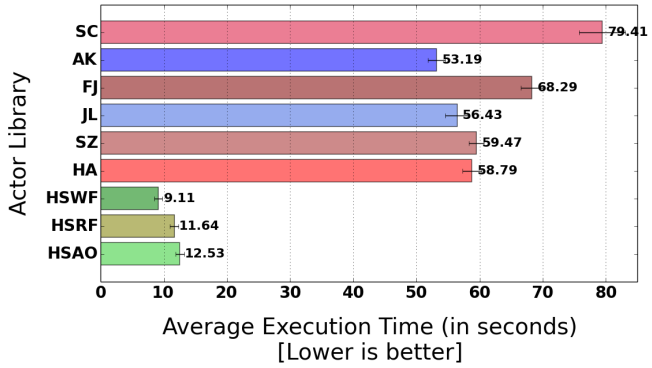

Figure 5.22 : Reader/Writer Concurrency: **Sorted List**. 24 worker actors with a total 15K requests per worker. 10 percent of the requests are write requests.

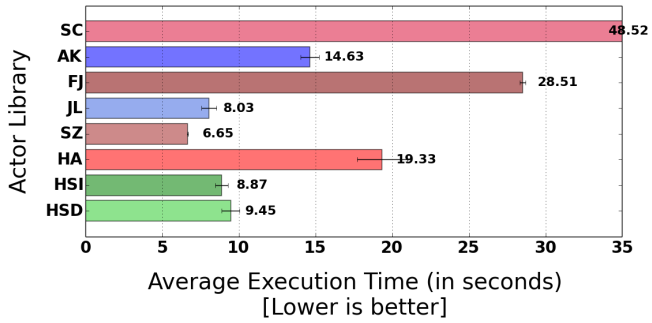

Figure 5.23 : Results of the Bounded Buffer benchmark on bounded buffer size of 6000. There were 5000 producer actors each producing up to 1000 messages. There were 2000 consumer actors.

mark also includes measurement of the mailbox contention overhead. The actor-based solutions have the additional overhead from maintaining additional data structures (adding and removing elements in collections) for the available producers and consumers in the buffer actor. In the selector-based solution, we avoid the need for additional collections by relying on enabling or disabling mailboxes. There are two selector versions, one which uses the declarative guard style (HSD) and another which uses an imperative style (HSI) with manual `enable` and `disable` calls. As seen in Figure 5.23, the HS* versions now performs similar to the JL and SZ versions (despite having overheads in mailbox contention). Importantly, the HS* performs much better

Name	Lines of Code		Cyc. Complexity		Dev. Effort ($\times 10^3$)	
	HA	HS	HA	HS	HA	HS
Logistic Map	165	150	2.23	1.92	110.81	89.42
Bank Transaction	139	129	2.16	1.83	90.88	76.10
Filterbank	351	335	2.00	1.88	352.38	329.92
K-N-Queens	157	165	2.20	2.18	88.59	103.79
A-Star Search	137	141	2.14	2.14	60.74	72.29
Dictionary	132	146	1.61	1.57	59.56	62.16
Sorted List	138	153	1.76	1.71	72.25	75.07
Bounded Buffer	185	179	2.11	1.70	120.39	95.77
`A. Mean`	175	175	2.03	1.87	119.44	112.19
%age of HS	0		8.56		6.47	

Figure 5.24 : Productivity metrics for selector/actor benchmarks. LoC was computed using `cloc` command while CC and DE were computed using the CodePro Analytix Eclipse plugin. LoC for common support code are not reported in the table, the arithmetic mean for support code LoC is 128.

than HA (in the Chameneos benchmark they had similar performance). The HSD version performs about 6.5% slower than the HSI version, this measures the cost of the declarative abstraction where all mailboxes need to be enabled or disabled after each message is processed. As in the Chameneos benchmark (Section 5.9.1), FJ and SC perform much slower compared to the other actor implementations.

5.9.2 Productivity Results

We limit our productivity measurements to a comparison of the Java-based solutions of Habanero actors and selectors. The limitation is because we could not find tools to measure Cyclomatic Complexity and Development Effort for Scala-based code. Our selectors implementation is in many ways an extension of Habanero actors, only comparing the Habanero solutions thus seems reasonable. In addition, since all the actor-based solutions use the same algorithm and have a similar code structure, using Habanero actors as the representative actor-based library does not affect the validity of our productivity results.

We report values for LoC, CC, and DE for our benchmarks in Figure 5.24. On av-

erage, the selector-based solutions require the same LoC as the actor-based solutions. The CC and DE are slightly lower, particularly for the synchronous reply, split-join and bounded buffer benchmarks. The message priority benchmarks take more effort to implement as the programmer explicitly specifies the priorities on the messages. Overall, the table shows that the selector-based solutions require about the same (or lower) amount of effort to develop compared to actor-based solutions. Coupled with results from Section 5.9.1, this shows that selectors offer a more performant alternative to actors for the specified synchronization patterns than actors.

5.10 Summary

In this chapter, we have introduced our extension to the AM called selectors. Selectors have multiple mailboxes and each mailbox is guarded i.e. it can be enabled or disabled to affect the order in which messages are processed. As opposed to using actors, selectors allow us to simplify writing of multiple synchronization and coordination patterns including: *a*) synchronous request-reply; *b*) join patterns in streaming applications; *c*) supporting priorities in message processing; *d*) variants of reader-writer concurrency; and *e*) producer-consumer with bounded buffer. We provided descriptions for solutions to some of these patterns using selectors and also included our results for benchmarks exhibiting these patterns. Our results confirm that selector-based solutions for benchmarks exhibiting these patterns are simpler and execute much faster than actor-based solutions.

Availability

Public distributions of the selectors implementation in Habanero-Java library, including documentation and code examples, are available for download at `http://wiki.rice.edu/confluence/display/PARPROG/HJ+Library`. The source code Scala versions of the selectors benchmarks are available as part of the Savina Benchmark Suite [110] on github.

Chapter 6

Related Work

If I have seen further than others, it is by standing upon the shoulders of giants.

Isaac Newton

6.1 Habanero-Java Framework

Modern languages and libraries provide lightweight dynamic task parallel execution models for improved programmer productivity. There are a handful of library extensions that provides parallel APIs, such as Intel Threading Building Blocks (TBB) [4], Java Concurrency Utilities (JUC) [131] and the Microsoft Task Parallel Library (TPL) [81]. TBB, JUC, and TPL expose constructs to create asynchronous tasks and `future`s. JUC also provides support for constructs similar to `phaser`s. The focus of these libraries is an industrial quality product and they are not necessarily amenable for pedagogic interests. For example, they support far fewer parallel constructs; do not include support for AEM or deadlock detection; and have no clear description of how to implement newer synchronization constructs on them.

The Streams API [132] in Java 8 greatly improves the processing of elements from collections and other sources and aids in writing data parallel programs. The Streams API is largely present to make the Collections Framework parallel-friendly. The Streams API also offers reduction operations such as reduce, collect, summation, maximum, and count. HJlib focuses on task parallelism, but such data parallel constructs can easily be implemented. In fact, such data parallel constructs have been

added as an extension to HJlib while implementing the PBBS benchmarks [133].

The GPars framework [134] offers Java developers intuitive and safe ways to handle Java or Groovy tasks concurrently. GPars has a larger range of parallel constructs than HJlib, it supports dataflow concurrency, actor programming model, communicating sequential processes, fork/join paralleism, etc. It is not always safe for different parallel constructs to be integrated while writing programs with GPars. GPars relies on external libraries to offer support for many of the parallel constructs. HJlib, on the other hand, presents a unified framework where all the parallel constructs can be safely composed. In addition, HJlib has no third-party dependencies and all the parallel constructs are implemented using the standard JDK and the `EventDrivenControl` API.

There are also new parallel languages such as IBM's X10 [135, 9], Cray's Chapel [8], and Oracle's Fortress [136]. These languages provide constructs to support fork-join style computations, but suffer from the drawback that they are new languages and require their own compiler and toolchain. Of these languages, Chapel has been used most successfully as a teaching language for parallel programming [137]. One of Chapel's original design goals was ease of use, which is exploited to quickly introduce the language in the context of an algorithms course to demonstrate parallelism in divide and conquer algorithms. In addition there are languages that are extensions to existing languages, such as the Cilk [3] and OpenMP 4.0 [83] extensions to C. Compared to Cilk's spawn-sync computations (and OpenMP parallel for loops) which must be fully-strict, HJlib's `async-finish` computations are terminally-strict but need not be fully-strict. HJlib also has a number of other constructs that are not all supported by these languages in a unified manner e.g., `future`s, `phaser`s, `isolated`, and actors.

6.2 Cooperative Scheduling in the presence of Synchronization Patterns

The general idea of using event-based programming in thread-based code has also been explored by others in the past. In Tasks [53], explicit method annotations provide yield points. These annotations are used to translate the code into event-based style using a form of continuation passing style (CPS) translation. Unlike what its name might suggest, Tasks has nothing to do with task parallelism, instead it is a programming model for writing event-driven programs. Both our approaches share the same philosophy of translating the user's thread-based code into event-based code. Our implementation requires no explicit method annotations, uses OSDCs, and runs safely on a parallel scheduler (i.e. the operations are thread-safe).

Use of continuations for task parallelism was popularized by Cilk [3], an extension to C that provides an abstraction of threads in explicit CPS. Cilk programs are preprocessed to C and then linked with a runtime library. Cilk supports the `spawn` and `sync` constructs, which are similar to `async` and `finish` constructs from HJ. Our approach uses OSDCs to achieve the same goal as Cilk where there are no thread blocking operations in the generated code. We support additional SyncCons where a task may trigger the enablement of multiple suspended tasks (as in `future`s, barriers and `phaser`s) in contrast to Cilk where only one continuation is enabled by an event (the termination of the last child/descendant task in a join scope). Since Cilk relies on serial elision to be equivalent to a sequential program, such programs are not supported in Cilk as there may be no equivalent sequential program which use these SyncCons. Having nonblocking operations allows us to provide proper time guarantees, since some progress is continually made towards the computation. In Cilk, such time guarantees are lost when locks, which are typically blocking, are used. However, supporting the time bound guarantee comes at a cost of space bound with all the additional space for temporary local variables in the heap.

The Intel Threading Building Blocks (TBB) [4] task scheduler is inspired by the

early Cilk work-stealing scheduler. TBB deals with possible blocking operations by running other tasks on the same stack, effectively stitching the call stack of the new tasks on top of the blocked task's stack. TBB also allows the parent tasks to specify another "continuation" task that will continue its work when such blocking scenarios arise. This minimizes the load on the scheduler and the uncontrolled overflow of the stack. However, this places the burden on the programmer to detect and schedule tasks to avoid blocking. In our approach, the user does not have to deal with the blocking constructs manually, the runtime implicitly handles the creation of continuations and the scheduler picks the next tasks to execute. Also, since each task has its own stack, we do not have to worry about the stack overflowing due to stitching of frames from multiple tasks. Overall, we go a step further than Cilk and TBB by showing how additional SyncCons such as `future`s, `phaser`s and `isolated` blocks can be supported in a nonblocking manner.

Qthreads [138] is a lightweight threading library for C/C++ applications that also uses call stack stitching, it allows spawning and controlling tasks with small (4k) stacks. Our runtime is based on OSDCs and poses no limits on the stack size of tasks created by the runtime, the stack size for worker threads in our implementation is limited by the JVM thread stack size (default around 1M for 64-bit JVMs) and the limits for OSDCs is defined by the size of the heap. The qthreads API provides access to full/empty-bit (FEB) semantics (producer-consumer pattern with mutable buffer) and the threads need to be able to interact with the FEB for synchronization. In our runtime, tasks synchronize among themselves using the EDC primitive which is based on the observer pattern.

Dolan et al. present a language-neutral primitive, called `SWAPSTACK`, for the LLVM compiler [96] for efficient context switching. Like our Kilim-based implementation, they rely on the use of one-shot continuations for an effective implementation and can resume the continuation on any thread, not just the one that created it. The `SWAPSTACK` primitive mimics the implementation of coroutines where a target for the

context switch is always provided explicitly. In their paper, they show how they can support fibers (tasks) which yield and are immediately returned to a work queue (i.e. the fibers are not suspended for any length of time). In the presence of synchronization constraints, separate suspendable queues are needed to maintain suspended fibers. A construct similar to EDCs can be utilized to track and manage the state of individual fibers. The main challenge in implementing EDCs using the SWAPSTACK primitive is in supporting the return to the boundary function while suspending as EDCs do not explicitly preserve the target continuation of each context switch. The runtime can maintain a separate context for each worker thread and store this context in a threadlocal storage. Each task also executes as a separate context and is swapped to the worker thread context when the EDC suspends. The worker thread executes its scheduling algorithm to pick the next ready task (closure) to execute. When the EDC resolves, a closure wraps the task context and places it into a work queue. Execution of this closure by a worker thread swaps the current context to the context of the task and resumes execution of the task past the suspension point.

Li et al. present an alternative approach to implement concurrency in Glasgow Haskell Compiler (GHC) [139]. The runtime offers continuations as a mechanism from which concurrency can be built and also supports preemptive concurrency of very lightweight threads. The one-shot continuations are implemented by providing support in the Haskell VM, which maintains the execution context at all times and makes it very cheap to create continuations. In their implementation of GHC, the list of suspended continuations is periodically polled by the scheduler to see if the cause for blocking has been resolved. We differ in that we use OSDCs and avoid any polling while deciding to resume suspended tasks by allowing EDCs to add resumed tasks into the scheduler's work queue. Also, we run inside the JVM where we cannot create continuations directly and have to rely on CPS-like transforms to support OSDCs.

Fluet et al. [52] use full continuations to support fine-grained parallelism in their Manticore project. Manticore, like the GHC, is based on a functional language. It

relies on a tree of `future`s that allows stacking continuations and a comparatively limited set of synchronization patterns (mainly `future`s). In contrast, our abstractions support a wide variety of synchronization patterns (e.g. `future`s, `phaser`s, atomic) and arbitrary computation DAGs where continuations may be placed in the work queue without restrictions.

The use of continuations in task parallel programs has also been proposed by the C++ implementation of X10 [140]. They support continuations by relying on the compiler and speculatively allocating the stack as the program executes. The work-stealing scheduler in their implementation supports the work-first policy inspired from Cilk. Their implementation supports distributed `async-finish` programs along with conditional atomic blocks but does not support clocks (a precursor to HJ `phaser`s). In our approach we rely on the help-first policy to have independent stack frames for tasks to enable use of OSDCs and can use either a work-sharing or work-stealing scheduler. Our cooperative runtime is general enough to support a wide variety of SyncCons as we prove in our implementation.

Continuations are also used in the Continuators construct for an implicitly parallel implementation of Scheme [141]. There continuations are used to invoke the body of a function application (without blocking the interpreter) after the arguments have been evaluated in parallel. We employ delimited continuations with the same goal of avoiding thread blocking operations, additionally, our work provides an API to implement SyncCons which subsumes the parallel argument evaluation case. Also, our implementation dynamically discovers suspension points and minimizes overhead by avoiding continuation creations when EDCs have already been resolved. In contrast, the parallel Scheme implementation relies on creations of continuations to manage all control flow.

6.3 Speculative Task Parallelism

Kolesnichenko et al. provide a comprehensive classification and evaluation of task termination techniques [88]. C# natively supports interruptive cancellation by throwing exceptions, and since the release of TPL also cooperative techniques [81]. Python supports interruptive cancellation of non-started tasks via executors and terminative cancellation of already started ones [80]. Java supports interruptive cancellations natively [82]. Pthreads library supports both termination and interruption of threads [142].

Burton [86] and Osborne [87] have both worked with speculative computation before. Burton proposes a deterministic feature that has simple semantics, i.e. produces the same result as a sequential computation. Osborne uses numerical priorities to order computations [87], in his work task priorities propagate among dependent (sponsored) tasks. The eureka scope of tasks is determined when they are stated ahead of time in `OR` clauses or as branches of a conditional. Computation termination is via the cancellation token approach where a programmer manually checks termination in each function. Compared to our model, Burton and Osborne style speculative execution support only the parallel search eureka pattern.

Prabhu et al. [143] propose two language constructs to declaratively express value speculation opportunities. Their approach relies on speculating the value of a computation and executing possible future computations that consume this value in parallel with the producer of the value. Our approach does not rely on value speculation and does not need to deal with the rollback of side-effects from *mispredicted* consumer tasks. Instead, we use speculative tasks in the EuPM to support a multitude of EuSCs.

Leaving the system in an inconsistent state is one of the problems with preemptive termination approaches. MVM [144] and J-SEAL2 [89] solve this problem by introducing isolation containers to segregate the data operated upon by tasks. Tasks cannot directly share objects, and the only way for tasks to communicate is

to use standard, copying communication mechanisms. Containers communicate via synchronous `receive` operations to pass notifications. Termination is effected on isolation containers by other containers, termination kills all worker threads assigned to individual containers. Our approach minimizes overheads as it avoids copying data, killing threads, and communicating via synchronous operations. In addition, creating containers is an expensive operation whereas, in our approach, creating multiple eureka sub-computations is cheap as it is akin to creating a task.

Cilk allows speculative work to be terminated through the use of Cilk's `abort` statement [29] inside function-scoped inlets. Cilk does not provide guarantees of when child tasks will be terminated, in fact, child tasks can be spawned even after the execution of an `abort` statement. However, the main difference is that in the EuPM only a subset of tasks can be terminated in contrast to terminating all child tasks via Cilk inlets. Perez and Malecha show several methods for implementing `abort` as a library in the cilk++ system [93] by mechanically translating programs into continuation-passing style. Like our approach, spawned computations periodically poll to determine if they should terminate. While this transformation is simple, the problem with it is that it is not modular because it changes the signatures of functions that use the `abort` mechanism. This breaks the possibility for separate compilation without explicit annotations specifying which functions should be compiled to work with inlets and `abort`.

Ada offers a statement, `abort`, which allows a task to make abnormal any other visible task [145]. The `abort` statement will stop execution of the named task by the time it reaches a synchronization point, e.g. `delay` statement that suspends the execution of a task for some units of time. Unfortunately, the use of `delay` statements (even those with delays of 0.0) can be expensive operations, as each delay statement forces the runtime system to perform a context switch. In our approach, a task cannot directly cancel another task, it influences cancellation by triggering eureka events. Also, our `check` construct does not force a task to context switch, making it

much cheaper to implement.

Both MPI and OpenMP support task grouping and cancellation. MPI provides termination support via the `MPI_Abort` function that terminates an MPI execution environment [146]. This function call makes a best attempt effort to terminate all tasks in the group of the communicator. OpenMP 4.0 API [83], released in July 2013, supports features to terminate parallel OpenMP execution cleanly. Tasks can be grouped to support deep task synchronization, and task groups can be terminated to reflect completion of cooperative tasking activities such as search. Threads check at user-defined cancellation points if cancellation has been requested. The cancellation points must be lexically nested in the type of construct specified in the clause; i.e. we cannot `cancel` from inside a method call. Our approach poses no such limitation on where a task can request cancellation and where the user-defined cancellation points can be placed in the program.

Tahan et al. [147] also propose a cancellation policy for OpenMP similar to Ada's abort where a task can cause the cancellation of a group of tasks (possibly not belonging to the same group as the currently executing task). In our approach a task can request cancellation of other tasks belonging to the same group. Like our approach, however, their approach also causes child tasks to inherit the cancellation properties from the parent task. Unlike our approach, certain tasks can be *protected* from being cancelled even though they belong to a cancelled group, and the task cancellation scheme is based on interrupts and exceptions. We chose to avoid such protected tasks to avoid any confusion and to keep the EuPM clean and simple.

6.4 Actors extensions for synchronization patterns

Tomlinson and Singh propose an extension to the AM to control the order in which messages are processed so as to preserve the integrity of objects [102]. They introduce the concept of enabled-sets that define the messages that may be executed in the new state. This idea is similar to the implementation of Scala actors by Haller [112]

where the pattern matching clause on messages can be modified dynamically. This is expensive to implement due to increased time in searching for the next message to process from the mailbox. In contrast, our approach allows enabling or disabling one of possibly many mailboxes, to control which messages are processed next. It is also efficient to implement as active mailboxes with messages can be found quickly.

Akka provides support for the aggregator pattern [121], however, their implementation does not match the sender of the message which is required to support the join pattern. Ensuring matching of senders requires extra tagging of messages as in a pure actor solution. On the other hand, with selectors, we can elegantly enforce the requirement that one message is received from each data stream source participating in the join.

Sulzmann et al. [148], Plociniczak et al. [149] and Haller et al [150] have proposed an extension to actors with `receive` clauses containing multiple message patterns based on Join calculus (not necessarily the same as the split-join pattern). This allows expression of join patterns by specifying an exhaustive list of messages that participate in the join, requiring that messages contain their source actor information, and including guards to restrict pattern matching. This offers a high-level way of synchronizing messages for processing at the cost of increased processing time (compared to our selector extension) in resolving a match in the `receive` clause for a message. To support other general Join calculus patterns, our extension may require the user to explicitly manage the state from the partial matches as each message from the active mailboxes are consumed.

The SALSA actor-oriented language [151] has supported two-level priority message sends since version 0.6.2. In SALSA, regular messages are placed at the end of the mailbox queue. Priority messages are placed at the front of the actor's mailbox, instead of the end. Nystrom presents a solution for supporting priority processing of messages in Erlang using nested non-blocking `receive` operations [152]. The nesting code structure convolves the pattern-matching code for supporting messages with

multiple levels of priorities. In their technique, since priorities are based on pattern-matching of individual messages, their solution is inefficient if mailboxes contain many pending messages. As stated in [152], the performance penalty of their approach keeps increasing as they support additional priority levels. Sulzmann et al. [148] also support priorities in their model with similar syntax. They concede that enforcing priorities manually via `otherwise` and nested `receive` statements leads to clumsy code. In contrast to these approaches, our extension prioritizes messages by the mailbox in which they reside and high-priority messages are selected using constant-time operations. The message processing logic in our solution is unaffected by the priority scheme.

Scholliers et al. have proposed the notion of Parallel Actor Monitors (PAM) [123] to extend the AM with support for intra-actor parallelization. PAM can easily solve the symmetric reader-writer problem where messages are processed in the arrival order they appear in the mailbox, but it does not support priorities of messages. De Koster et al. propose the TANK model [153] where actors expose part of their state as a shared read-only resource for other tanks (containers for related actors). The model relies on an approach similar to transactional memory that ensures an actor will always observe a consistent version of a read-only state, even in the face of concurrent updates of the actor that owns that state. In particular, it enables reader-writer concurrency where any number of reader methods can be executing in parallel with at most one writer method thus solving the first reader-writer variant. Our support for priorities enables us to solve the three variants to the reader-writer problem.

Chapter 7

Future Work & Conclusions

A story really isn't truly a story until it reaches its climax and conclusion.

Ted Naifeh

7.1 Future Work

We have used HJlib as an implementation vehicle for the runtimes and programming models described in this dissertation. Being an implementation of a pedagogic programming model, HJlib is an attractive tool for both educators and researchers. HJlib is actively being used in multiple research projects at Rice and also by external independent collaborators. HJlib parallel constructs are being used in the implementation of the emulator for Fresh Breeze multiprocessor architecture to speedup the emulation process itself [154]. There are also plans to include HJ's data race detector [155] into HJlib and in exploiting multicore parallelism in Hadoop applications [31].

Extending our cooperative runtime to support preemptive scheduling using the notion of Engines [65] to ensure fairness in the scheduling of tasks is an interesting direction to merge the benefits of both preemptive and priority scheduling. Such a scheme also requires support for runtime generated priorities while scheduling tasks. Exploiting the dynamic dependence graph around suspension points to detect data races and to help in debugging is also an interesting area of future research. We have made some progress in this direction with the support for an intuitive graphical view of computation graphs (CGs) for HJlib programs [156].

An interesting area of future work is exploring a distributed implementation of the EuPM. Extending support for further eureka patterns and providing dynamic task priorities in EuSCs is also a promising direction. We have done some preliminary work in developing a decentralized work-stealing scheduler that dynamically schedules fixed-priority tasks in a non-preemptive manner [157]. Assigning priorities to tasks can be a method to influence the execution ordering in the scheduling of tasks [158]. In fact, benchmarks such as branch-and-bound and single-source shortest path show that prioritization of tasks can reduce the total amount of work required compared to standard work-stealing execution order [159].

Future work also entails exploring additional computational patterns that can be expressed easily using selectors. For example, nondeterministic finite automata can be expressed easily using selectors where each input symbol is sent to a different mailbox. This should facilitate implementing multiple message patterns based on Join calculus using selectors. Another direction for future work is the automatic transformation of actor-based code to selector-based code by a compiler. This will allow us to extract performance in legacy actor-based applications. The effectiveness of the process will be greatly increased by identifying further inefficient synchronization patterns in actor-based code.

We believe the HJlib cooperative runtime is a versatile framework capable of supporting task-based parallel applications and constructs. For example, HJlib has been used to implement object-based isolation that is guaranteed to be deadlock-free, while still retaining the rollback benefits of transactions [160]. The approach differentiates between read and write accesses in its concurrency control mechanisms. The plethora of research done using the core HJlib runtime is a testament to HJlib's flexibility. We look forward to even more interesting work being done, both on the core runtime and in developing new programming constructs on HJlib.

7.2 Conclusions

With the advent of the multicore era, it is clear that future growth in application performance will primarily come from increased parallelism. Certain parallel applications have tasks that need to synchronize or communicate with each other more often [2]. The dissertation builds on the observation that current parallel runtime lack efficient support for more general synchronization patterns that are necessary for such applications is often lacking. This dissertation addressed the effective execution of parallel applications multicore and manycore systems in the presence of modern synchronization and coordination constraints.

Many modern synchronization and coordination constructs in parallel programs can incur significant performance overheads on current runtime systems, or significant productivity overheads when the programmer is forced to complicate their code to mitigate these performance overheads. The thesis claimed that cooperation between the programmer and the runtime system can help address the performance and productivity challenges of using modern synchronization and coordination constructs. The challenge with such an approach is to ensure that it does not sacrifice the readability and maintainability of the code while still delivering useful parallel performance. Our contribution is in demonstrating such goals can be achieved by cooperative approaches in runtime implementations and programming model extensions.

In this dissertation, we first described the implementation framework we used in Chapter 2. Habanero-Java library (HJlib) is a highly extensible library implementation of a task-parallel programming model with many synchronization and coordination constructs. HJlib is also actively being used in multiple research projects at Rice and also by external independent collaborators. Chapter 3 describes a cooperative runtime implementation for such constructs using a unified `EventDrivenControl` API based on a transparent use of delimited continuations. The cooperative runtime supported unmodified HJlib user code and regularly attained over 2 × speed-up on

our benchmarks.

Chapter 4 and Chapter 5 presented two programming model extensions that use cooperative approaches. In Chapter 4, we introduced our extension to support different variants of speculative tasks called the Eureka programming model. Our extension relies on cooperative termination of tasks, and we showed that our approach delivered similar performance to hand-coded variants with much higher productivity. In Chapter 5, we introduced our extension to the event-driven Actor model to support a variety of commonly used synchronization and coordination patterns. Our extension called selectors, relied on the use of cooperatively guarded mailboxes. This approach showed that the selector-based solutions enable achieved better performance than the actor-based solutions without sacrificing productivity.

The framework described in the dissertation also paves the way for experimenting with cooperative approaches in other synchronizations and coordination scenarios such as object-based isolation and supporting task priorities. By demonstrating the above approaches, this dissertation makes concrete contributions towards promoting the use of cooperative techniques for the execution of parallel tasks with synchronization constraints. We foresee such cooperative approaches becoming the norm in mainstream parallel programming languages and runtime.

> *Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience.*
>
> Roger Bacon

Bibliography

[1] J. Ponge, "Fork and Join: Java Can Excel at Painless Parallel Programming Too!." http://www.oracle.com/technetwork/articles/java/fork-join-422606.html, 2011.

[2] P. Vicat-Blanc, B. Goglin, R. Guillier, and S. Soudan, *Computing Networks: From Cluster to Cloud Computing*. John Wiley & Sons, 2013.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, (New York, NY, USA), pp. 207–216, ACM, 1995.

[4] J. Reinders, *Intel Threading Building Blocks*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., first ed., 2007.

[5] "OpenMP Application Program Interface, Version 3.0." http://www.openmp.org/mp-documents/spec30.pdf, May 2008.

[6] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS Extension for C++," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1105–1114, May 2014.

[7] D. Lea, "A Java Fork/Join Framework," in *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, (New York, NY, USA), pp. 36–43, ACM, 2000.

[8] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and

the Chapel Language," *International Journal of High Performance Computing Applications*, vol. 21, pp. 291–312, August 2007.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-Oriented Approach to Non-uniform Cluster Computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '05, (New York, NY, USA), pp. 519–538, ACM, October 2005.

[10] Y. Yan, S. Chatterjee, Z. Budimlić, and V. Sarkar, "Integrating MPI with Asynchronous Task Parallelism," in *Recent Advances in the Message Passing Interface*, vol. 6960 of *Lecture Notes in Computer Science*, pp. 333–336, Berlin, Heidelberg: Springer-Verlag, 2011.

[11] V. Cavé, J. Zhao, Y. Guo, and V. Sarkar, "Habanero-Java: the New Adventures of Old X10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, (New York, NY, USA), pp. 51–61, ACM, 2011.

[12] R. H. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 501–538, October 1985.

[13] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-Point Synchronization," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 277–288, ACM, 2008.

[14] S. Taşırlar and V. Sarkar, "Data-Driven Tasks and their Implementation," in *Proceedings of the International Conference on Parallel Processing (ICPP) 2011*, ICPP '11, (Washington, DC, USA), pp. 652–661, IEEE, September 2011.

[15] C. Hewitt, P. Bishop, and R. Steiger, "Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (N. J. Nilsson, ed.), IJCAI 1973, pp. 235–245, William Kaufmann, August 1973.

[16] Hewitt, Carl and Baker, Henry G., "Actors and Continuous Functionals," tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, February 1978.

[17] G. Agha, *Actors: a model of concurrent computation in distributed systems.* Cambridge, MA, USA: MIT Press, 1986.

[18] R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG (2nd Ed.).* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[19] D. Charousset, R. Hiesgen, and T. C. Schmidt, "CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, (New York, NY, USA), pp. 15–28, ACM, October 2014.

[20] D. Kafura, "ACT++: Building a Concurrent C++ with Actors," *Object Oriented Program*, vol. 3, pp. 25–37, April 1990.

[21] J.-P. Briot, "Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment," in *Proceedings of the 3rd European Conference on Object-Oriented Programming*, ECOOP '89, pp. 109–129, Cambridge University Press, 1989.

[22] C. Tismer, "Continuations and Stackless Python," in *Proceedings of the 8th International Python Conference*, January 2000.

[23] J. Ayres and S. Eisenbach, "Stage: Python with Actors," in *Proceedings of the 2nd International Workshop on Multicore Software Engineering*, IWMSE '09, (Washington, DC, USA), pp. 25–32, IEEE, May 2009.

[24] J. Sillito, "Stage: Exploring Erlang Style Concurrency in Ruby," in *Proceedings of the 1st International Workshop on Multicore Software Engineering*, IWMSE '08, (New York, USA), pp. 33–40, ACM, 2008.

[25] Microsoft Corporation, "Asynchronous Agents Library." http://msdn.microsoft.com/en-us/library/dd492627.aspx, 2013.

[26] Rettig, Mike, "retlang: Message based concurrency in .NET." http://code.google.com/p/retlang/, 2010.

[27] S. Tasharofi, P. Dinges, and R. E. Johnson, "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?," in *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP '13, (Berlin, Heidelberg), pp. 302–326, Springer-Verlag, 2013.

[28] I. A. Mason and C. L. Talcott, "Actor Languages. Their Syntax, Semantics, Translation, and Equivalence," *Theoretical Computer Science*, vol. 220, pp. 409–467, June 1999.

[29] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proceedings of the 19th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '98, pp. 212–223, May 1998.

[30] S. Imam and V. Sarkar, "Integrating Task Parallelism with Actors," in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 753–772, ACM, 2012.

[31] Zhang, Yunming and Cox, Alan and Sarkar, Vivek, "HJ-Hadoop: An Optimized MapReduce Runtime for Multi-core Systems," in *5th USENIX Workshop on Hot Topics in Parallelism (HotPar '13)*, June 2013. Accepted as poster with accompanying paper.

[32] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, (Washington, DC, USA), pp. 1918–1927, IEEE, 2013.

[33] Luontola, Esko, "Retrolambda: Use Lambdas on Java 7." https://github.com/orfjackal/retrolambda, 2013.

[34] S. Imam and V. Sarkar, "Habanero-Java Library: a Java 8 Framework for Multicore Programming," in *Proceedings of the 11th International Conference on the Principles and Practice of Programming on the Java Platform*, PPPJ '14, pp. 75–86, New York, USA: ACM, 2014.

[35] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 151–162, October 2006.

[36] "Habanero Extreme Scale Software Research Project." https://wiki.rice.edu/confluence/display/HABANERO/Habanero+Extreme+Scale+Software+Research+Project, 2014.

[37] V. Cavé, Z. Budimlić, and V. Sarkar, "Comparing the Usability of Library vs. Language Approaches to Task Parallelism," in *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, (New York, NY, USA), pp. 9:1–9:6, ACM, 2010.

[38] "COMP 322: Fundamentals of Parallel Programming." https://wiki.rice.edu/confluence/display/PARPROG/COMP322, 2014.

[39] "Habanero-Java Library Javadoc." http://www.cs.rice.edu/~vs3/hjlib/doc/, 2014.

[40] R. Mason, G. Cooper, and M. de Raadt, "Trends in Introductory Programming Courses in Australian Universities - Languages, Environments and Pedagogy," in *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*, vol. 123 of *ACE '12*, (Darlinghurst, Australia, Australia), pp. 33–42, Australian Computer Society, Inc., 2012.

[41] S. Davies, J. A. Polack-Wahl, and K. Anewalt, "A Snapshot of Current Practices in Teaching the Introductory Programming Sequence," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (T. J. Cortina, E. L. Walker, L. A. S. King, and D. R. Musicant, eds.), SIGCSE '11, pp. 625–630, ACM, 2011.

[42] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient Data Race Detection for Async-Finish Parallelism," in *Proceedings of the First international conference on Runtime verification*, RV'10, (Berlin, Heidelberg), pp. 368–383, Springer-Verlag, 2010.

[43] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar, "Test-Driven Repair of Data Races in Structured Parallel Programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 15–25, ACM, June 2014.

[44] B. Goetz, "Thread pools and work queues." http://www.ibm.com/developerworks/library/j-jtp0730/index.html, July 2002.

[45] Miller, Alex, "Set your Java 7 Phasers to stun." http://tech.puredanger.com/2008/07/08/java7-phasers/, 2008.

[46] J. Shirako, V. Cavé, J. Zhao, and V. Sarkar, "Finish Accumulators: An Efficient Reduction Construct for Dynamic Task Parallelism," in *Languages and Compilers for Parallel Computing* (H. Kasahara and K. Kimura, eds.), vol. 7760 of *Lecture Notes in Computer Science*, pp. 264–265, Springer Berlin Heidelberg, 2013.

[47] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and Other Cilk++ Hyperobjects," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, (New York, NY, USA), pp. 79–90, ACM, 2009.

[48] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism," in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, (Washington, DC, USA), pp. 1–12, IEEE, 2009.

[49] S. Imam and V. Sarkar, "Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns," in *Proceedings of the 28th European Conference on Object-Oriented Programming*, ECOOP '14, pp. 618–643, Springer Berlin Heidelberg, 2014.

[50] D. P. Reed and R. K. Kanodia, "Synchronization with Eventcounts and Sequencers," *Commun. ACM*, vol. 22, pp. 115–123, February 1979.

[51] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, (New York, NY, USA), pp. 95–109, ACM, 1991.

[52] M. Fluet, M. Rainey, J. Reppy, and A. Shaw, "Implicitly Threaded Parallelism in Manticore," *Journal of Functional Programming*, vol. 20, pp. 537–576, November 2010.

[53] J. Fischer, R. Majumdar, and T. Millstein, "Tasks: Language Support for Event-driven Programming," in *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '07, (New York, NY, USA), pp. 134–143, ACM, 2007.

[54] S. Imam and V. Sarkar, "A Case for Cooperative Scheduling in X10's Managed Runtime," in *The 2014 X10 Workshop (X10'14)*, June 2014.

[55] S. Srinivasan and A. Mycroft, "Kilim: Isolation-Typed Actors for Java," in *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, (Berlin, Heidelberg), pp. 104–128, Springer-Verlag, 2008.

[56] B. Sigoure, "How long does it take to make a context switch." http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html.

[57] J. Gray, "Writing Faster Managed Code: Know What Things Cost." http://msdn.microsoft.com/en-us/library/ms973852.aspx.

[58] Onur Gumus, "C# versus C++ versus Java performance comparison." http://reverseblade.blogspot.com/2009/02/c-versus-c-versus-java-performance.html, 2009. [Online; accessed 10-April-2014].

[59] M. Felleisen, "The Theory and Practice of First-Class Prompts," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, (New York, NY, USA), pp. 180–190, ACM, 1988.

[60] O. Danvy and A. Filinski, "Abstracting Control," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, (New York,

NY, USA), pp. 151–160, ACM, 1990.

[61] Oliver Kowalke, "Introduction (Boost Coroutines)." http://www.boost.org/doc/libs/1_53_0/libs/coroutine/doc/html/coroutine/intro.html, 2009.

[62] T. Rompf, I. Maier, and M. Odersky, "Implementing First-class Polymorphic Delimited Continuations by a Type-directed Selective CPS-transform," in *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, (New York, NY, USA), pp. 317–328, ACM, 2009.

[63] Ilya Grigorik, "Untangling Evented Code with Ruby Fibers." https://www.igvita.com/2010/03/22/untangling-evented-code-with-ruby-fibers/, 2010.

[64] I. Drago, A. Cunei, and J. Vitek, "Continuations in the Java Virtual Machine," in *International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS'2007, 2007.

[65] C. T. Haynes and D. P. Friedman, "Engines Build Process Abstractions," in *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, LFP '84, (New York, NY, USA), pp. 18–24, ACM, 1984.

[66] M. Feeley, "Polling Efficiently on Stock Hardware," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pp. 179–187, ACM, 1993.

[67] The Jikes RVM Project, "Threading and Yieldpoints." http://jikesrvm.org/Threading+and+Yieldpoints, 2007.

[68] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative Task Management Without Manual Stack Management," in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical*

Conference, ATEC '02, (Berkeley, CA, USA), pp. 289–302, USENIX Association, 2002.

[69] J. Mellor-Crummey, "Lecture 28: Bitonic Sort." https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s12-lec28-slides-JMC.pdf?version=1&modificationDate=1333163955158.

[70] A. Iliev, "A Simple Parallel FFT Implementation Using Cilk." https://github.com/ailiev/fft-cilk.

[71] I. Corporation, "Integrate." https://svn.code.sourceforge.net/p/x10/code/benchmarks/trunk/microbenchmarks/Integrate/.

[72] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, (Washington, DC, USA), pp. 124–131, IEEE, 2009.

[73] M. Frigo, "knapsack.cilk." http://courses.cs.tau.ac.il/368-4064/cilk-5.3.1/examples/knapsack.cilk.

[74] EPCC, "The Java Grande Forum Multi-threaded Benchmarks." http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/s1contents.html, 2001.

[75] S. Gupta and V. K. Nandivada, "IMSuite: A Benchmark Suite for Simulating Distributed Algorithms," *Journal of Parallel and Distributed Computing*, vol. 75, no. 0, pp. 1–19, 2015.

[76] I. Corporation, "KMeansX10RT.java." https://svn.code.sourceforge.net/p/x10/code/benchmarks/trunk/microbenchmarks/KMeans/src-java/.

[77] W.-M. Lin, W. Xie, and B. Yang, "Performance Analysis for Parallel Solutions to Generic Search Problems," in *Proceedings of the 1997 ACM Symposium on Applied Computing*, SAC '97, (New York, NY, USA), pp. 422–430, ACM, 1997.

[78] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, October 1990.

[79] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *National Academy of Sciences of the United States of America*, vol. 85, pp. 2444–2448, April 1988.

[80] Python Software Foundation, "concurrent.futures Launching parallel tasks." https://docs.python.org/3/library/concurrent.futures.html, August 2014.

[81] D. Leijen, W. Schulte, and S. Burckhardt, "The Design of a Task Parallel Library," in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, (New York, NY, USA), pp. 227–242, ACM, 2009.

[82] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[83] "OpenMP API, Version 4.0." http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013.

[84] A. Marochko, "Exception Handling and Cancellation in TBB - Part II." https://software.intel.com/en-us/blogs/2008/05/29/exception-handling-and-cancellation-in-tbb-part-ii-basic-use-cases, May 2008.

[85] S. Imam and V. Sarkar, "The Eureka Programming Model for Speculative Task Parallelism," in *Proceedings of the 29th European Conference on Object-Oriented Programming*, ECOOP '15, pp. 421–444, 2015.

[86] F. W. Burton, "Speculative Computation, Parallelism, and Functional Programming.," *IEEE Transactions Computers*, vol. 34, no. 12, pp. 1190–1193, 1985.

[87] R. B. Osborne, "Speculative Computation in Multilisp," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, (New York, NY, USA), pp. 198–208, ACM, 1990.

[88] A. Kolesnichenko, S. Nanz, and B. Meyer, "How to Cancel a Task," in *Proceedings of the 2013 International Conference on Multicore Software Engineering, Performance, and Tools*, MUSEPAT'13, pp. 61–72, Springer, 2013.

[89] W. Binder, "Design and Implementation of the J-SEAL2 Mobile Agent Kernel," in *Proceedings of the IEEE 2001 Symposium on Applications and the Internet*, SAINT-01, pp. 35–42, 2001.

[90] Wolfram, "Solve Optimization Problems with Speculative Parallelism." http://www.wolfram.com/products/mathematica/newin7/content/BuiltInParallelComputing/SolveOptimizationProblemsWithSpeculativeParallelism.html, November 2008.

[91] L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Highlights from Twenty-Five Years., The Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS-25, pp. 113–119, IEEE, June 1995.

[92] J. Clausen, "Branch and Bound Algorithms - Principles and Examples," *Parallel Computing in Optimization*, pp. 239–267, 1997.

[93] R. Perez and G. Malecha, "Speculative Parallelism in Cilk++," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2012.

[94] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo, *Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science)*. Springer US, 2005.

[95] A. Greenbaum, *Iterative Methods for Solving Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[96] S. Dolan, S. Muralidharan, and D. Gregg, "Compiler Support for Lightweight Context Switching," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, pp. 36:1–36:25, January 2013.

[97] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "UTS: An Unbalanced Tree Search Benchmark," in *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pp. 235–250, Springer-Verlag, 2007.

[98] R. Wiener, "Branch and Bound Implementations for the Traveling Salesperson Problem," *Journal of Object Technology*, vol. 2, no. 2, 2003.

[99] L. Stadler, C. Wimmer, T. Würthinger, H. Mössenböck, and J. Rose, "Lazy Continuations for Java Virtual Machines," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, (New York, NY, USA), pp. 143–152, 2009.

[100] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, July 1976.

[101] M. H. Halstead, *Elements of Software Science*. New York, NY, USA: Elsevier Science Inc., 1977.

[102] C. Tomlinson and V. Singh, "Inheritance and Synchronization with Enabled-Sets," in *Proceedings of the 4th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '89, (NY, USA), pp. 103–112, ACM, 1989.

[103] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, vol. 17, pp. 549–557, October 1974.

[104] G. Hohpe and B. Woolf, "Enterprise Integration Patterns - Request-Reply." http://www.eaipatterns.com/RequestReply.html, 2003. [Online; accessed 3-April-2014].

[105] Apache Software Foundation, "Apache Camel: Aggregator." https://camel.apache.org/aggregator2.html, 2004.

[106] Microsoft Developer Network, "Priority Queue Pattern." http://msdn.microsoft.com/en-us/library/dn589794.aspx, 2014. [Online; accessed 3-April-2014].

[107] Wikipedia, The Free Encyclopedia, "ReadersWriters Problem." http://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem, 2014. [Online; accessed 3-April-2014].

[108] Wikipedia, The Free Encyclopedia, "ProducerConsumer Problem." http://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem, 2014. [Online; accessed 3-April-2014].

[109] S. Imam and V. Sarkar, "Selectors: Actors with Multiple Guarded Mailboxes," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, (New York, NY, USA), pp. 1–14, ACM, October 2014.

[110] S. Imam and V. Sarkar, "Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, (New York, NY, USA), pp. 67–80, ACM, October 2014.

[111] P. Haller, "On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective," in *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! 2012, (New York, NY, USA), pp. 1–6, ACM, 2012.

[112] P. Haller and M. Odersky, "Scala Actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2–3, pp. 202–220, 2009. Distributed Computing Techniques.

[113] Imam, Shams and Sarkar, Vivek, "Habanero-Scala: Async-Finish Programming in Scala," in *Scala Days 2012*, April 2012.

[114] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Wiley Publishing, 8th ed., 2008.

[115] Wikipedia, The Free Encyclopedia, "Multilevel queue." http://en.wikipedia.org/wiki/Multilevel_queue, 2014. [Online; accessed 28-September-2014].

[116] Wikipedia, The Free Encyclopedia, "Round-robin scheduling." http://en.wikipedia.org/wiki/Round-robin_scheduling, 2014. [Online; accessed 28-September-2014].

[117] Oracle, "Understanding Interaction Patterns." http://docs.oracle.com/cd/E17904_01/doc.1111/e17363/chapter05.htm, 2011.

[118] Typesafe Inc., "Actors - Akka Documentation." http://doc.akka.io/docs/akka/2.3.2/scala/actors.html, 2014.

[119] Sybase Inc., "Complex Event Processing: Ten Design Patterns," white paper, Sybase - An SAP Company, April 2001.

[120] W. Thies and S. Amarasinghe, "An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, USA), pp. 365–376, ACM, 2010.

[121] Typesafe Inc., "Aggregator Pattern - Akka Documentation." http://doc.akka.io/docs/akka/snapshot/contrib/aggregator.html, 2014.

[122] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent Control with "Readers" and "Writers"," *Communications of the ACM*, vol. 14, pp. 667–668, October 1971.

[123] C. Scholliers, ı. Tanter, and W. De Meuter, "Parallel Actor Monitors: Disentangling Task-level Parallelism from Data Partitioning in the Actor Model," *Science of Computer Programming*, pp. 52–64, February 2014.

[124] Imam, Shams, "DataDrivenControl." http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/runtime/forkjoin/DataDrivenControl.html, 2014.

[125] "functionaljava: A Library for Functional Programming in Java." https://code.google.com/p/functionaljava/, 2010.

[126] Rettig, Mike, "jetlang: Message based concurrency for Java." http://code.google.com/p/jetlang/, 2014.

[127] L. Hupel and typelevel.org, "scalaz: Functional programming for Scala." http://typelevel.org/projects/scalaz/, 2010.

[128] Haller, Philipp, "chameneos-redux.scala — FishEye: browsing scala-svn." https://codereview.scala-lang.org/fisheye/browse/scala-svn/scala/branches/translucent/docs/examples/actors/chameneos-redux.scala?hb=true, 2011.

[129] R. M. May, "Simple mathematical models with very complicated dynamics," *Nature*, vol. 261, pp. 459–467, June 1976.

[130] Wikipedia, The Free Encyclopedia, "Logistic map." http://en.wikipedia.org/wiki/Logistic_map, 2014.

[131] "Java Concurrency Utilities." http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/, 2014.

[132] Goetz, Brian, "State of the Lambda: Libraries Edition." http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html, 2013.

[133] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief Announcement: The Problem Based Benchmark Suite," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, (New York, NY, USA), pp. 68–70, ACM, 2012.

[134] The GPars team, "The GPars Project - Reference Documentation." http://www.gpars.org/guide/, 2014.

[135] K. Ebcioglu, V. Saraswat, and V. Sarkar, "X10: An Experimental Language for High Productivity Programming of Scalable Systems," in *In the Second Workshop on Productivity and Performance in High-End Computing*, PPHEC-05, February 2005.

[136] E. Allan, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, G. L. Ryu, Sukyoung Steele Jr., and S. Tobin-Hochstadt, "The Fortress Language Specifica-

tion Version 1.0." http://research.sun.com/projects/plrg/fortress.pdf, March 2008.

[137] "Chapel Educator Resources." http://chapel.cray.com/education.html, 2014.

[138] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, 2008.

[139] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach, "Lightweight Concurrency Primitives for GHC," in *Proceedings of the ACM SIGPLAN Haskell Workshop*, Haskell '07, (New York, NY, USA), pp. 107–118, ACM, 2007.

[140] O. Tardieu, H. Wang, and H. Lin, "A Work-Stealing Scheduler for X10s Task Parallelism with Suspension," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 267–276, ACM, 2012.

[141] C. Herzeel and P. Costanza, "Dynamic Parallelization of Recursive Code Part I: Managing Control Flow Interactions with the Continuator," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, (New York, NY, USA), pp. 377–396, ACM, 2010.

[142] Bradford Nichols and Dick Buttlar and Jacqueline Proulx Farrell, *Pthreads Programming: Chapter 4 - Managing Pthreads*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[143] P. Prabhu, G. Ramalingam, and K. Vaswani, "Safe Programmable Speculative Parallelism," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 50–61, 2010.

[144] G. Czajkowski and L. Daynés, "Multitasking Without Comprimise: A Virtual Machine Evolution," in *Proceedings of the 16th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '01, pp. 125–138, 2001.

[145] J. Goldenberg and G. Levine, "Ada's Abort Statement: License to Kill," *Ada Letters*, vol. IX, pp. 97–103, September 1989.

[146] The Open MPI Project, "MPI_Abort." https://www.open-mpi.org/doc/v1.8/man3/MPI_Abort.3.php, 2014.

[147] O. Tahan, M. Brorsson, and M. Shawky, "Introducing Task Cancellation to OpenMP," in *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, (Berlin, Heidelberg), pp. 73–87, Springer-Verlag, June 2012.

[148] M. Sulzmann, E. S. L. Lam, and P. V. Weert, "Actors with Multi-headed Message Receive Patterns," in *Proceedings of the 10th International Conference on Coordination Models and Languages*, vol. 5052 of *COORDINATION'08*, pp. 315–330, Springer, 2008.

[149] H. Plociniczak and S. Eisenbach, "JErlang: Erlang with Joins," in *Proceedings of the 12th International Conference on Coordination Models and Languages* (D. Clarke and G. Agha, eds.), vol. 6116 of *COORDINATION'10*, pp. 61–75, Berlin, Heidelberg: Springer-Verlag, 2010.

[150] P. Haller and T. Van Cutsem, "Implementing Joins Using Extensible Pattern Matching," in *Proceedings of the 10th International Conference on Coordination Models and Languages*, COORDINATION'08, (Berlin, Heidelberg), pp. 135–152, Springer-Verlag, 2008.

[151] C. Varela and G. Agha, "Programming Dynamically Reconfigurable Open Systems with SALSA," *ACM SIGPLAN Notices*, vol. 36, pp. 20–34, December

2001.

[152] J. H. Nystrom, "Priority Messaging Made Easy," in *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*, ERLANG '07, (NY, USA), pp. 65–72, ACM, 2007.

[153] J. De Koster, S. Marr, T. D'Hondt, and T. Van Cutsem, "Tanks: Multiple Reader, Single Writer Actors," in *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2013, (NY, USA), pp. 61–68, ACM, 2013.

[154] K. Bhandari, "Evaluating the programmability and scalability of memory hierarchies with read-only data blocks," Master's thesis, Rice University, Houston, Texas, April 2015.

[155] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and Precise Dynamic Data Race Detection for Structured Parallelism," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 531–542, ACM, 2012.

[156] P. Elmers, H. Li, S. Imam, and V. Sarkar, "HJ-Viz: A New Tool for Visualizing, Debugging and Optimizing Parallel Programs," in *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '14, (New York, NY, USA), pp. 53–54, ACM, October 2014.

[157] S. Imam and V. Sarkar, "Load Balancing Prioritized Tasks via Work-Stealing," in *Proceedings of the 21st International European Conference on Parallel and Distributed Computing*, Euro-Par'15, Berlin, Heidelberg: Springer-Verlag, August 2015.

[158] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas, "Data Structures for Task-based Priority Scheduling," in *Proceedings of the 19th ACM SIG-*

PLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, pp. 379–380, 2014.

[159] M. Wimmer, D. Cederman, J. L. Träff, and P. Tsigas, "Work-stealing with Configurable Scheduling Strategies," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 315–316, ACM, 2013.

[160] S. Imam, J. Zhao, and V. Sarkar, "A Composable Deadlock-free Approach to Object-based Isolation," in *Proceedings of the 21st International European Conference on Parallel and Distributed Computing*, Euro-Par'15, Berlin, Heidelberg: Springer-Verlag, August 2015.