Schedule Data, Not Code

by

Micah J Best

BSc (Honours), University of Victoria, 2004
MSc, Simon Fraser University, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

October 2020

© Micah J Best, 2020
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

**Schedule Data, Not Code**

submitted by **Micah J Best** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Computer Science**.

**Examining Committee:**
Alexandra Fedorova, Associate Professor, Electrical and Computer Engineering, UBC  
*Supervisor*

Arvind Gupta, Professor, Computer Science, University of Toronto  
*Supervisory Committee Member*

Ronald Garcia, Associate Professor, Computer Science, UBC  
*University Examiner*

Chen Feng, Assistant Professor, School of Engineering, UBC Okanagan  
*University Examiner*

**Additional Supervisory Committee Members:**
Ivan Beschastnikh, Associate Professor, Computer Science, UBC  
*Supervisory Committee Member*

Sathish Gopalakrishnan, Associate Professor, Electrical and Computer Engineering, UBC  
*Supervisory Committee Member*
Abstract

Parallel programming is hard and programmers still struggle to write code for shared memory multicore architectures that is both free of concurrency errors and efficient. Tools have advanced, but for tasks that are not embarrassingly parallel, or suitable for a limited model such as map/reduce, there is little help. We aim to address some major aspects of this still underserved area.

We construct a model for parallelism, *Data not Code* (DnC), by starting with the observation that a majority of performance and problems in parallel programming are rooted in the manipulation of data, and that a better approach is to *schedule data, not code*. Data items don’t exist in a vacuum but are instead organized into collections, so we focus on concurrent access to these collections from both task and data parallel operations. These concepts are already embraced by many programming models and languages, such as map/reduce, GraphLab and SQL. We seek to bring the excellent principles embodied in these models, such as declarative data-centric syntax and the myriad of optimizations that it enables, to conventional programming languages, like C++, making them available in a larger variety of contexts.

To make this possible, we define new language constructs and augment proven techniques from databases for accessing arbitrary parts of a collection in a familiar and expressive manner. These not only provide the programmer with constructs that are easy to use and reason about, but simultaneously allow us to better extract and analyze programmer intentions to automatically produce code with complex runtime optimizations.

We present Cadmium, a proof of concept DnC language to demonstrate the effectiveness of our model. We implement a variety of programs and show that,
without explicit parallel programming, they scale well on multicore architectures. We show performance competitive with, and often superior to, fine-grained locks, the most widely used method of preventing error-inducing data access in parallel operations.
Lay Summary

Most modern computing devices, from desktops to cellphones, have CPUs (Central Processing Units: the part responsible for the actual computations) with multiple cores. Sparing technical details, this is as if the device has multiple CPUs and can perform truly simultaneous computations allowing them, theoretically, to do more work in less time.

This creates problems for software developers. These independent cores cannot tell what the others are doing and share the same memory. If two or more CPUs attempt to change the same memory at the same time, corruption of data can occur. A CPU can signal the others of their activity, but that can slow down the computation and be difficult to get correct.

We borrow ideas from Databases and other fields to propose a change to how programs are written and develop a better method for organizing these signals for more efficient coordination.
Preface

This work grew out of a collaborative project under Dr. Alexandra Fedorova. I produced several papers on these ideas with Craig Mustard, Shane Mottishaw, Mark Roth and with the assistance of the others listed as coauthors below.

- Micah J Best, Nicholas Vining, Daniel Jacobsen and Alexandra Fedorova, *Collection-focused Parallelism*, *Fifth USENIX Workshop on Hot Topics on Parallelism (HotPar 13)*, 2013

  I did the vast majority of design, implementation and writing for this paper. Nicholas Vining and Daniel Jacobsen contributed suggestions, feedback and aid with some of the code. Alexandra Fedorova supervised this process.

- Mark Roth, Micah J Best, Craig Mustard and Alexandra Fedorova, *Deconstructing the Overhead in Parallel Applications*, *IEEE International Symposium on Workload Characterization*, 2012

  Mark Roth was the primary driver of the design of the work in this paper. Both Craig Mustard and I contributed refinements, wrote code and performed tests. The writing was a collaborative effort. Alexandra Fedorova supervised this process.


  I contributed the original ideas that formed the basis of this paper. Shane Mottishaw, Craig Mustard and Mark Roth provided numerous refinements
and improvements. The coding, testing and writing was a collaborative effort
between the four of us. Andrew Brownsword provided significant advice,
feedback and refinement. Alexandra Fedorova supervised this process.

- Micah J Best, Shane Mottishaw, Craig Mustard, Mark Roth, Parsiad Azimzadeh, Alexandra Fedorova, Andrew Brownsword, Schedule Data Not Code, Third USENIX Workshop on Hot Topics on Parallelism (HotPar 11), 2011

I contributed the original ideas that formed the basis of this paper. Shane
Mottishaw, Craig Mustard, Mark Roth and Parsiad Azimzadeh provided nu-
merous refinements and improvements. The coding, testing and writing was
a collaborative effort between the five of us. Andrew Brownsword pro-
vided significant advice, feedback and refinement to all stages of this project.
Alexandra Fedorova supervised this process.

- Micah J Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova and Andrew Brownsword, Synchronization via Scheduling: Managing Shared State in Video Games, Second USENIX Workshop on Hot Topics on Parallelism (HotPar 10), 2010

This was the workshop paper that formed the basis for the PLDI paper
above. The contributions here are identical to those listed there.


This was a collaborative effort between myself, Ryan Dickie, Andrea
Tagliasacchi, Alex Couture-Beil, Craig Mustard, Shane Mottishaw, Aron
Brown, Zhi Feng Huang, Xiaoyuan Xu and Nasser Ghazali. Each of us
contributed to the design, coding, testing and writing. Andrew Brownsword
provided significant advice, feedback and refinement to all stages of this project. Alexandra Fedorova supervised this process.

Shane Mottishaw extended some of these ideas, primarily the paper published in PLDI, in his 2011 MSc thesis *Synchronization Via Scheduling: Techniques For Efficiently Managing Concurrent Shared Memory Accesses*. The work presented here takes a different direction from that work.

This work is, primarily, an attempt to refine and extend the ideas and techniques from these publications. Chapter 5, in particular, is an extension of this work and cites the above, as appropriate.

I implemented essentially all the code discussed in this document. The overwhelming majority of the code taken from these previous projects has been rewritten.

In Section 7.2.4 I describe a piece of software that I collaborated on during a series of MITACS internships at Gaslamp Games with Nicholas Vining and Daniel Jacobsen. No code from this period was used for this work as it is property of the company, nor were any of the techniques or algorithms in this Section implemented in the software during my employment. I used my experiences to inspire the tools that would have aided me at the time of my employment. No endorsement from Gaslamp Games is implied.
# Table of Contents

Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii

Lay Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . v

Preface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vi

Table of Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ix

List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xiii

List of Source Code Fragments . . . . . . . . . . . . . . . . . . . . . . xv

Acknowledgments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xvi

Dedication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xvii

1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.1 Playing Chess in the Dark . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Beyond Peeling Potatoes . . . . . . . . . . . . . . . . . . . . . . . 3
1.3 A Cautionary Tale . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.4 Why So Serial? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
1.5 An Intentional Solution . . . . . . . . . . . . . . . . . . . . . . . . 10
1.6 What We Did . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
1.7 What We Didn’t Do . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
1.8 Fantastic Contributions and Where to Find Them . . . . . . . . . 14
2 Related Work ................................................................. 16
  2.1 Signatures .............................................................. 16
  2.2 Software Transactional Memory (STM) ......................... 17
  2.3 Computations on Collections ...................................... 19
  2.4 Parallel Query Processing ......................................... 19
  2.5 Parallel Languages .................................................. 20
  2.6 Actor Model ........................................................... 22
  2.7 Deadlock Detection .................................................. 22

3 Expressing Intention ....................................................... 23
  3.1 Cadmium Language Overview ..................................... 23
    3.1.1 Entities .......................................................... 24
    3.1.2 Managers ......................................................... 27
    3.1.3 Collections ..................................................... 28
    3.1.4 Message Sending and Program Flow ....................... 28
    3.1.5 Accumulators .................................................. 30
  3.2 Collection Usage, Collection Contracts and Queries .......... 31
    3.2.1 Collection Contracts ......................................... 31
    3.2.2 Declarative Syntax for Collection Contracts .............. 32
    3.2.3 Memberships and Subcollections ............................ 36
    3.2.4 Views ........................................................... 37
    3.2.5 Delegate Blocks ............................................... 38
    3.2.6 Isolation Semantics .......................................... 38
  3.3 Further Queries ..................................................... 40
    3.3.1 INSERT .......................................................... 40
    3.3.2 DELETE .......................................................... 41
    3.3.3 UPDATE .......................................................... 42
  3.4 Conclusion ............................................................ 43

4 Extracting Intentions: Compiling Cadmium and Static Analysis ... 44
  4.1 Parallel Opportunities .............................................. 44
    4.1.1 Task Parallelism ............................................... 45
    4.1.2 Data Parallelism ............................................... 46
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A subtree substructure query on a tree</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Deadlock detection algorithm</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>Mapping queries to signatures</td>
<td>61</td>
</tr>
<tr>
<td>5.2</td>
<td>Partition numbering for linked lists</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>Partition numbering ((n = 16)) of quadtree (4 children)</td>
<td>67</td>
</tr>
<tr>
<td>6.1</td>
<td>Example Code Analysis Report</td>
<td>73</td>
</tr>
<tr>
<td>7.1</td>
<td>Signature error rates – weakswapper versus brazen</td>
<td>84</td>
</tr>
<tr>
<td>7.2</td>
<td>PageRank on 10,000 vertices Cadmium versus Green-Marl</td>
<td>92</td>
</tr>
<tr>
<td>7.3</td>
<td>PageRank on 10,000 vertices Cadmium versus Green-Marl - Batch Factor Variation</td>
<td>93</td>
</tr>
<tr>
<td>7.4</td>
<td>PageRank (Google web graph) Activity Graph for batch factor 10</td>
<td>94</td>
</tr>
<tr>
<td>7.5</td>
<td>PageRank (Google web graph) Activity Graph for batch factor 2500</td>
<td>94</td>
</tr>
<tr>
<td>7.6</td>
<td>PageRank (Google web graph) Cadmium versus Green-Marl</td>
<td>95</td>
</tr>
<tr>
<td>7.7</td>
<td>Mesh Refinement on 10,000 vertices Cadmium versus Galois</td>
<td>99</td>
</tr>
<tr>
<td>7.8</td>
<td>Mesh Refinement on 10,000 Vertices – Signature Size Variation</td>
<td>101</td>
</tr>
<tr>
<td>7.9</td>
<td>Mesh Refinement on 10,000 Vertices – Batch Size Variation Normalized</td>
<td>102</td>
</tr>
<tr>
<td>7.10</td>
<td>canneal on 100k elements Cadmium versus PARSEC</td>
<td>107</td>
</tr>
<tr>
<td>7.11</td>
<td>canneal on 200k elements Cadmium versus PARSEC</td>
<td>108</td>
</tr>
</tbody>
</table>
# List of Source Code Fragments

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Example Cadmium Entity</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Cadmium contract: Isolate and Release phases</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Deadlock Potential Example</td>
<td>55</td>
</tr>
<tr>
<td>7.1</td>
<td>PageRank expressed in Green-Marl</td>
<td>90</td>
</tr>
<tr>
<td>7.2</td>
<td>PageRank expressed in Cadmium</td>
<td>90</td>
</tr>
<tr>
<td>7.3</td>
<td>Mesh Refinement Fragment 1 in Cadmium</td>
<td>98</td>
</tr>
<tr>
<td>7.4</td>
<td>Mesh Refinement Fragment 2 in Cadmium</td>
<td>99</td>
</tr>
<tr>
<td>7.5</td>
<td>Snippet of Galois C++ Code</td>
<td>104</td>
</tr>
<tr>
<td>7.6</td>
<td>canneal Cadmium Fragment</td>
<td>106</td>
</tr>
<tr>
<td>7.7</td>
<td>Grid Storage Definition for Agents</td>
<td>121</td>
</tr>
<tr>
<td>7.8</td>
<td>simAgentGrid Query</td>
<td>121</td>
</tr>
<tr>
<td>7.9</td>
<td>Job Selection</td>
<td>122</td>
</tr>
<tr>
<td>7.10</td>
<td>Rendering from the Quadtree</td>
<td>124</td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to thank my supervisors, for showing me more patience than I deserved, as my progress deadlocked and crashed repeatedly. I couldn’t have asked for better mentors and better role models.
Dedication

Pour Maude, qui a ajouté toute une nouvelle dimension à mon univers.
Tout ceci n’aurait pas été possible sans toi. Pour notre vie toute en-
Chapter 1

Introduction

Parallel programming is hard.

1.1 Playing Chess in the Dark

Programming in general is hard, requiring intelligence, attention to detail, intuition, experience and a certain amount of creativity. In this way, it’s much like chess. In which case, parallel programing is like playing 3D chess ... in the dark ... with an opponent that occasionally swaps your pieces without warning.

Writing bug free serial code is difficult enough as it is. Consider the case of the implementation of the LZO algorithm [15], which was not only written by a very talented programer, but also ported and rewritten by several other talented programmers. It was used in Linux and even went to Mars. However, there was a bug in the original code and this bug was propagated though each of the derivative versions. It took 20 years before somebody discovered it.

In Star Trek canon, the purely logical and brilliant Spock was known to lose games of 3D chess against the folksy Dr. McCoy with his unpredictable, seemingly illogical, moves. Similarly, even talented programmers can lose against the erratic and invisible moves of parallel execution. The programmer must face a horde of extra problems including data races, deadlocks, and ulcer-inducing heisenbugs that scuttle into dark corners whenever they spot you coming with a debugger.
Furthermore, correctness is necessary, but not sufficient\(^1\). Parallelism is for performance. No algorithm requires parallel execution. The metric of success is \textit{how much faster is this than the serial version?} and in this case our devious opponent still has more tricks. One can easily introduce suboptimal cache utilization patterns or one of many other unfortunate circumstances from a long list of pitfalls.

The author of this work started serious parallel programming working for a video game studio, producing a title for the 3-cored Xbox 360. He worked for weeks to refactor some fairly deep engine code to free up an extra core for his parallel implementation of one of game’s subsystems. With great anticipation, he ran his first tests – only to find it was slower with the additional core. It was then that he learned about pros and cons (mostly the latter) of hyperthreading and, more generally, how deep the rabbit hole goes. It’s still remarkable, and probably a sign of psychological damage, that he didn’t run screaming from the building and that you’re not reading his thesis on \textit{Chaucer, Shakespeare and Other Things That Existed Long Before Computers}.

There are several different recognized ‘flavours’ of parallel execution. From instruction level parallelism, with SIMD operations, to massive ‘webscale’ distributed services. In this work we’re concerned with multicore parallelism – multiple physical processors running on a single machine with a shared address space memory.

We are now, by most accounts, in the second decade of the ‘multicore revolution’. The increase in clock rates effectively stalled in the mid-2000s, but Moore’s Law\(^2\) [66] has continued to hold, giving chip designers more and more transistors to play with. They used this surplus to put the contents of two or more processors onto the same chip. To paraphrase a tired internet meme: \textit{yo dawg, we heard you liked processors, so we put a processor in your processor}. We then had CPUs that could finally do what operating systems had been faking for years: run more than one thing at once\(^3\). Concurrency had become true parallelism.

\(^1\)For completeness, we note there is an interesting line of inquiry into algorithms that permit some degree of incorrectness for the sake of performance. However, this is limited to a small class of problems and so we will assume correctness as a requirement.

\(^2\)We used to joke in the lab that the three most important laws that governed our work were: Moore’s, Amdahl’s and Murphy’s.

\(^3\)Yes, technically multi-processor systems pre-date even personal computers, but this was the first
Today, even cheap smartphones have multiple cores and researchers are discovering that many big data operations are more efficient on multicore architectures than networked clusters [60], so multicore parallelism is applicable to a large percentage of developers.

If we can be allowed a little reflected arrogance, the community of Systems researchers, both academic and corporate, are the ‘arms suppliers’ for the programmers in the trenches\(^4\). The limits of how far programming can go are shaped by the tools that we create. So, in this light: \textit{How are we doing?}. Well, one need only launch Task Manager, Activity Monitor or \texttt{top} to see the answer: \textit{Not as well as we could be}. Most apps on your device will probably never utilize more than one core, even though by now we all probably have two or more\(^5\). Certainly, a few applications will happily gobble all your computational resources, but they tend to be ‘the big ones’ written by companies, like Google and Adobe, with massive engineering resources.

\subsection{1.2 Beyond Peeling Potatoes}

For completeness, we’re going to need to define ‘embarrassingly parallel’ for the following discussion and the rest of this document. In the seminal \textit{Art of Multiprocessor Programming} [39], the authors state:

\begin{quote}
Some computational problems are “embarrassingly parallel”: they can easily be divided into components that can be executed concurrently.
\end{quote}

When explaining our work to laypeople\(^6\) we often use the metaphor of ‘peeling potatoes’ to describe embarrassingly parallel tasks. \textit{If it takes me an hour to peel a bucket of potatoes, then it would take two of us half an hour.} We would proceed to talk about the fact that we could keep adding people until we had more people than potatoes and the biggest headache we have is distributing the potatoes in an entry into consumer level machines.

\(^4\)In a reverse \textit{No True Scotsman}, lets consider every programmer who advances the state of the art an honorary Systems researcher

\(^5\)This is being written on a Apple laptop with 6 cores and at the time of writing no process has used more than 100\% CPU all afternoon.

\(^6\)We were tempted to say muggles, but referencing both Star Trek and Harry Potter in one section might be too geeky even for a Computer Science PhD thesis.
efficient matter. Though that last part can be surprisingly non-trivial, it’s certainly easier to deal with than having your kitchen help occasionally stabbing each other because you forgot to secure a mutex somewhere in your code.

So, what do we see when we look at the support given to the modern programmer? Broadly, the answer falls into three major categories. The first is ways to easily dispatch embarrassingly parallel work, with enumerable variations on the `parallel_for` with packages such as OpenMP [24]. The second category is systems that use a `dataflow` approach, where data is transformed into a series of pipelined stages. This has become increasingly popular with the recent explosion of machine learning. However, this isn’t much different than the first category, only adding some implicit dependency handling between stages. Furthermore, it’s not a general purpose approach. TensorFlow [9] is a technical marvel, but you wouldn’t want to write a word processor with it.

What if you had an algorithm whose updates were not easily described by independent operations on a sequence of data, where two updates may touch the same data at the same time (which we will refer to as a `state conflict`)? What if you had more than one potential operation on the same data at the same time? The proffered solutions make up the third category of our broad taxonomy and it’s a category that’s mostly made up of locks. Sometimes these include very fancy locks: reentrant mutexes, object based monitors and even more exotic constructs. Furthermore, what’s not a lock is generally a method for ‘forking’ and ‘joining’ threads of execution with constructs such as promises, futures, etc. Essentially this sends the unstated message: *If you’re embarrassingly parallel, we have you covered. Otherwise, here’s a bunch of pieces to build your own system.*

At this point, we need another definition: *complex application*, though in this case it’s our own, more informal term. It may be easier to define it in comparison to a simple application. A simple application is one that has a single purpose, generally taking a set of inputs and producing a set of outputs. Many Unix CLI applications are like this, such as `wc` which takes a stream of text and returns the number of words. These can also include applications that stay resident in memory and ‘serve’ independent requests where the requests are processed in a simple matter. A complex application is then one that is not simple and generally has some, but not necessarily all, of the following attributes:
• deals with a collection of data that, for even reasonable performance, must be modified ‘in place’ in memory (rather than immediately transformed to a new representation)

• deals with multiple collections

• contains processes that employ multiple algorithms

• potentially initiates multiple processes per execution on the same input data

• takes user input in an interactive manner

Examples of these include: web browsers, word processor, image editors, video games, etc.

We make this definition digression, at this point, to highlight the class of applications that often contain non-embarrassingly parallel operations and present difficulties in ‘rolling your own’ scheme for safe parallel execution.

1.3 A Cautionary Tale

Let us give an illustrative story and consider a fictitious *Lorem Ipsum LLC* that creates real-time data visualization software. The software receives updates to the data from the Internet and allows the user to add a set of different components and sub-components for visualization (call these *widgets*), any of which may contain user supplied scripts to filter, organize and interpret the data. Initially, the decision is made that since the widgets organization is hierarchical, then they obviously should be stored in tree structure\(^7\).

Initially, Pat is charged with implementing the actual rendering to the screen. Since the number of different items can be larger than fits on one screen, they simply find the node of the tree that is the common parent of every visible item and walk the tree, drawing each to the screen in a post-order transversal.

At the same time Sanjay is writing the network code, receiving partial updates from the network and modifying the data that feeds the widgets.

\(^7\)For those readers familiar with the web stack, this is purposefully a simplified version of the DOM tree.
Early user testing reveals that the more times the display is updated per second, the more responsive it feels and consequentially leads to higher user satisfaction in the target audience. So it is decided that instead of interleaving the display update and the network update, they could be run on different threads to decrease the time between updates.

As Lorem Ipsum is made up of UI/UX coders and data analysis experts, nobody is really an expert on parallel programming. Pat looks up the documentation to fork a second thread, putting their code on one thread and initiating Sanjay’s code on the other. *That really wasn’t that bad*, thinks Pat for the approximately 30 seconds before the application segfaults when the networking code is updating a node that is currently being rendered. With a sigh, Pat digs into the memory of their third year Operating Systems class and reads the documentation on mutexes, wrapping each access to the tree in lock/release pair. After wasting a weekend tracking down a method with an early exit that didn’t release the mutex properly, a more frazzled Pat sits back in satisfaction and watches as the program runs for more than 10 minutes without a crash. The sense of satisfaction is soon replaced with more frazzle when timings reveal that the update is actually running *less* frequently. Both the rendering code and the network code are pretty much always using *some* part of the tree at any given time and so the two sub-systems are effectively interleaved. With the additional overhead of lock acquisition, the total time is longer.

Pat decides that each individual node in the tree needs a lock, and adds these fine-grained locks, locking each node before rendering and releasing it afterwards. Now the update rate actually does increase and the crashes are still absent, but Pat begins to notice strange rendering bugs have appeared with elements going outside of their proscribed boundaries. With a few more late nights, Pat realizes that some widgets have a size based on their child widgets and when data is updated during rendering, the sizes can change between the time that the layout is constructed and the widget is rendered. Pat comes to the conclusion that they need to lock the entire subtree being rendered, doing a walk first to lock each node, then rendering and finally another walk to unlock. Finally, after several days away from the work that utilizes their actual expertise, Pat has an implementation that works. The rendering errors are gone and the application is updating more frequently than it was serially. Perhaps it’s not quite the total victory they hoped for, as the lock acquisition time
is a still added to the total rendering time, but it is an improvement.

However, while Pat is settling back into designing the perfect rounded rectangle for a new widget, Kamiko merges her branch with the first generation of scripting functionality in it. She’s surprised and disheartened to find that it crashes almost immediately post-merge. So after a few frantic teleconferences, she gets ahold of Pat who tells her about their ad-hoc locking scheme and promises to write something about it on the internal wiki, when they have time. Kamiko goes back, spending an unplanned day wrapping her references to the tree in accordance with the scheme. This works fine until beta-user reports come in that the application locks up intermittently. Kamiko and Pat put their heads together and realize that Kamiko’s scripts can access more than one lock at once. In just the right circumstance a script can lock a node just before the rendering locking can get to it. It then requires a node that the rendering has already locked and each will then wait for the other forever. This is the classic deadlock situation and Kamiko and Pat start to get the classic deadlock induced ulcers.

Finally, they agree to adopt an ‘all or nothing’ approach with their locks. Kamiko’s code takes less locks, so they try that first. However, with many different branches it becomes really complicated and so, with deadlines slipping they finally decide to have the tree code back off if it can’t acquire a lock.

In the end they have an application that runs in parallel, without error, but occasionally it stutters a little when the renderer and the scripting system ‘fight’. However, it is faster and they need to ship and see their spouses at some point. They leave it as is and get back to their delayed ‘real’ work, with the nagging feeling that it could have been better.

What can we do to help the Lorem Ipsums of the world with their complex applications? This document is our answer to that question. Unfortunately, we haven’t quite derived a solveMyParallelProblems() function, but we believe we’ve moved them closer than the Ikea-furniture-assembly-nightmare of trying to keep lock discipline across a large, complex application.
1.4 Why So Serial?

Now that we’ve illustrated the problem, we can start to build towards our solution. We start by analyzing what went wrong in our little fable. Each of our three heroes needed to access different subsets of the same tree, where they needed the entire subset to complete their computations. The reader may have noticed how many times we have, already, used terms like set or collection. Collections are fundamental to Computer Science. We would be hamstrung without the ability to generalize a process across an arbitrarily sized collection. A datum is a social beast, it travels in packs.\(^8\)

If we consider the collection as a single entity, such as when Pat tried the global lock and found that many items were needlessly locked, we miss the many cases where disjoint parts of the collection could be access simultaneously. Conversely, we can fail to see the tree for the nodes if we consider the elements as individual items as with the subtle errors produced with fine-grained locks.

Furthermore, collections aren’t, for the most part, bags of values. Collections have structure. When Pat employed fine-grained locks, they weren’t able to exploit the beautifully recursive structure of the data.

Notice as well that the problems we described didn’t come from inside each of the three components (rendering, network and scripting).\(^9\) The problems came from the interaction between disconnected operations. This is why we underscore the pervasiveness of complex applications, where a parallelization strategy that works for one subsystem may not apply to others sharing the same collection. Finally, to bring this to the most abstract level, we note that these problems all stem from specific combinations of data being accessed simultaneously.

At this point, we will need another definitional digression. We have noticed that one of the lesser discussed problems in STEM is that the English language, like the IPv4 address space, simply doesn’t have enough distinct elements for modern purposes and many words have become overloaded.\(^10\) This is definitely the case

---

8 Consider how rarely you hear the singular form, as opposed to the collective data.

9 Though, had the trio not given up they might have discovered deadlocks trying to parallelize their operations – especially the scripting system, which is essentially the same problem only in miniature.

10 The author has actually spent more than one entire evening arguing with his fiancée, a statistician, over the meaning of variable.
with the term *task*. We will use this term throughout this document in its broader sense. A task is the basic unit that is scheduled. It consists of an entry point to the code and zero or more parameters for the invocation. We will assume that it is designed to terminate\(^{11}\) and will run serially, though it may generate more tasks using the same mechanism that was used to generate the initiating task. A program can then be defined as a scheduler and a set of potentially executed tasks. In this way we could view a classically serial program as a single ‘mono-task’ with an ‘empty’ scheduler or a program using the fork/join threaded model as a scheduler, plus one task for the main thread and a task for each thread spawned.

With that terminology established, we can now describe the crux of the problem as a step in real-time scheduling problem. Say we have *p* processors and let \(D = \{d_1, \ldots, d_n\}\) be all the data items in the system. Let \(T = \{t_1, \ldots, t_m\}\) be the tasks that have been submitted to the scheduler for execution at the current time. For every \(t_i\) there is an \(R_i \subseteq D\) which is the data items to be accessed by that task. In order to guarantee a safe execution, we want to derive a subset of tasks \(T' \subseteq T\) where \(|T'| \leq p\) such that for every pair \(t_j, t_k \in T', j \neq k\), \(R_j \cap R_k = \emptyset\). This \(T'\) give us a ‘coschedule’, a group of tasks to be run simultaneously\(^{12}\). Note that it follows that there must exist a \(T'\) that is non-empty.

The primary observation we want to make from this is that the tasks themselves aren’t important for the safe coschedule, except in their transitive relationship to the data. We can essentially reduce this to determining a mutually disjoint subset of \(R_1, \ldots, R_m\). In a perfect world we could take a program and derive a dependency graph of subsets of \(D\) and whenever a processor is free, select a subset that both had its dependencies satisfied and was disjoint in terms of data items with what was currently running. This would execute without state conflicts. This is the luxury that a system like Tensorflow possesses, where a DAG\(^{13}\) is explicitly constructed and most operations don’t modify existing data in favor of producing results as new data. As mentioned previously, the constraints necessary for that model are too limiting for general purpose applications.

\(^{11}\)Though this constraint could be relaxed for event driven ‘daemon’ style applications, but we are not considering such applications.

\(^{12}\)Note this can also be used a verb; we can talk about coscheduling two or more tasks.

\(^{13}\)Directed Acyclic Graph.
So, in developing a general system for safe and efficient parallel execution what we have to work with is a scheduler and a set of potential tasks instead of the an idealized graph of data subsets. The question: from that starting point, how close can we get to that ideal? In other words, how do we schedule data, not code.

1.5 An Intentional Solution

Our quest to schedule data presents two major problems:

- How do we associate a subset of data with a given task?
- Given two sets of data, how do we test for intersection between sets of data in the microseconds we have to make a scheduling decision?

In both cases the fact that data is naturally grouped into structured collections, a property that we will exploit without mercy, does allow us to shrink the problem size drastically. For example, we know that if two collections $A$ and $B$ are disjoint and if tasks $t_1$ and $t_2$ access only $A$ and $B$ respectively, then intersection testing is constant.

To make things more difficult, the exact data used by each task is going to change based on runtime inputs. To compensate for the lack of exact knowledge, we exploit the fact that our program will still run successfully if our intersection testing allows false positives. If we decline to execute one of two tasks that when executed turn out to be disjoint, we have only sacrificed some performance. So, we relax our goal to finding the smallest possible superset of the data touched. Again, we can leverage the collection based partitioning of data: if a task is doing an operation on a collection, it will at most touch the entire collection, but that operation will never include items outside the collection.

While these facts help, and will be integral to our proposed solution, they still don’t give us everything we need. We could use static analysis on each task and try to determine the used data superset. However, this is notoriously difficult. Consider a program written in C that walks a tree. The programmer sits down, thinks about trees and writes a node `struct` and connects them with pointers. By the time the code is submitted to the compiler, all that semantic information is

\[\text{We are assuming that the set of possible tasks is easily harvestable from the code as written.}\]
gone and the compiler only sees a series of pointer operations. For the information we need, we go to the source: the programmer’s intentions. Unfortunately, neural programming interfaces are still nowhere close to production, so we must rely on creating better tools.

The language is the closest point of contact with the programmer in terms of the toolchain. We elect to design a set of language constructs that better capture programmer intention. This is where our observations about collections really come into play. The use of a global variable is obvious in most every language, but telling which parts of a shared collection are accessed is much harder to determine. By making collections first-class citizens of the language and mediating all access to shared data with stateless, declarative queries that reflect the structure of the target collection, we are able to dramatically increase the power of our analysis.

With this approach, Pat could have written a line of code that effectively said I want to use the subtree of my widget tree rooted at node X. With this we would be able to build a runtime mechanism that could differentiate the desired part of the collection from the part that goes unused. Pat must also tell us I’m done with this at some point. We will discuss this in Chapter 3.

This approach has two other virtues. Firstly, because these constructs replace existing code, we are not suggesting burdening the programmer with an extra layer of annotations on top of already complex code. This also makes our lives easier as we don’t need to check that these hypothetical annotations match the code as written because it is the code. When Pat would say I want to use the subtree rooted at X, this is not an annotation surrounding the actual data access. This directive compiles to code that accesses only that subtree. Secondly, as these constructs are stateless and declarative, they give us the what and the when of the data accessed, but are separated from the how. This gives us plenty of latitude in our scheduler to apply scheduling optimizations across the entire program without violating the intent of the programmer.

Processing queries on structured data is, of course, not new. Database systems have been doing it for decades. We note that a lot of applications, especially those written for the web, are written as a symbiosis between program code and database. One could construe this part of our proposal as generalizing databases for multiple collection types and moving them into the program itself, much like the switch to
multicore moved multiple distinct processors into the same physical chip.

Of course, this would all be for naught if we couldn’t exploit this information for efficiency. This means we need a general mechanism for scheduling which is able to encode the parts of the collection used, as dictated by the programmer’s queries, and do the intersection testing as close as possible to instantaneously.

We have derived such a method, using bit strings, which we call signatures, and leveraging hardware supported atomic operations. These signatures act analogously to a lingua franca, a means of common exchange between operations. By deriving a signature for every query at runtime, we reduce the problem of scheduling disjoint data groupings into one of comparing signatures.

These two systems, query formulation and signature comparison act in harmony to allow us to approach the ideal of scheduling data, not code and so we refer to this model as Data not Code (DnC).

1.6 What We Did

A model by itself is not particularly useful. In Systems, especially, new ideas need to be shown to have practical applications or, at least in the case of new research, demonstrated potential.

We have proposed a system that is a symbiosis between different levels of the software process: software construction, program compilation/analysis and runtime execution. These facets are often studied and refined in isolation – holding the other layers ‘fixed’.

In the classic and well respected The Pragmatic Programmer [44] the authors detail an approach they refer to as a tracer bullet for exploratory development of complex software. The idea is roughly orthogonal to the common software approach of developing a single system and making ‘mock-ups’ or ‘stubs’, that have the interface, but not the functionality, of the systems that the subject will interact with. A tracer bullet on the other hand is a process of developing a narrow pathway through the system from invocation to output. For example, a user interface may be developed with only one or two fields, which are checked by another level of code, processed by a third and finally transmitted over the network with a protocol that only has one message and one response. Is is then stored in a database that
has only a table for those values. All other parts of the software that are ‘adjacent’ to this trace are ‘stubbed’. The results of this exercise serve as proof of concept, a road map and opportunity to expose unforsaken difficulties.

In this spirit we produced Cadmium, a prototype language, compiler and runtime library based around the DnC model. We developed a number of applications in this language in order to demonstrate and justify the process from end to end. Certainly we left out the details that weren’t necessary. Our standard library has about six functions, for example.

Our contributions are not just in the algorithms and techniques, but in the experience we relate and the insight we gained building and attempting to use this system. We will discuss our successes and failures in Chapter 7. We certainly did have some failures: optimizations that didn’t work out and language constructs that turned out to be far more unwieldy in use than ‘on paper’. We hope that these will be of benefit to future researchers as a complement to our successes and as fertile ground to grow even better solutions. Even if we never write another line of Cadmium code, we want to use software that can maximally exploit the beautiful hardware that we’ve paid way too much for.\(^{15}\)

### 1.7 What We Didn’t Do

Our primary goal for this project was to demonstrate the viability of the DnC model and we hope to convince you that we have succeeded. As a byproduct, we have produced a number of novel solutions to different problems. However, it was unfeasible to solve all the problems. Technically, there are an infinite number of things that we didn’t do\(^ {16}\), but there have been certain things that we have been asked about multiple occasions during the course of this project’s development. We will list the major ones here to properly temper expectations:

- deterministic execution
- distributed computations
- interacting with the GPU

\(^{15}\)The reader can probably deduce our brand preference, but we won’t name any names.

\(^{16}\)At least it’s probably countably infinite.
Many of these form large chunks of our future work, though sometimes they are quite low on our list.

Furthermore, the primary thrust of our experiment was the synthesis of many different facets that are normally considered separately. Due to the fact that our result is not a theoretic design, but a realized system we had to develop solutions to achieve the results we want. While we most certainly have novel aspects to our work, there is a lot we just had to adapt for our purposes. We don’t claim to have advanced the state of the art in static analysis, for example. In fact our project could have benefited from even more advanced static analysis. However, the way we assemble the pieces is novel and a contribution. The purpose was to prove that our model is worth further inquiry and worthy of efforts towards better static analysis and other techniques.

### 1.8 Fantastic Contributions and Where to Find Them

Our contributions can be organized into three categories:

**Model** We introduce the Data not Code (DnC) model which describes the general mechanism and constructs we are employing.

**Language constructs** We detail a number of new and adapted programming constructs that can be used to realize the DnC model.

**Runtime algorithms** We introduce a number of new runtime algorithms that facilitate the efficient execution of DnC conforming programs.

After we have discussed the various works that either inspired us or tried to solve similar problems (Ch 2), we organize our discussion following the process of translating programmer intentions into correctly executing and performant software. We have grouped these discussions into three phases:

**Express Intentions** by using a collection of programming constructs while avoiding requirements unrelated to expressing program function (Ch 3).

**Extract Intentions** using static analysis and language extensions to derive useful information and enforce the extracted intentions (Ch 4).

**Enforce/Execute Intentions** using information from the previous phases to con-
struct an executable and inform its runtime elements for efficient and safe execution (Ch 5).

Following this, we give an overview of the compiler implementation (Ch 6) and an in-depth exploration of the results in executing our test applications, presenting both statistics and commentary (Ch 7).

Finally, we conclude with a discussion of future work (Ch 8).
Chapter 2

Related Work

No project is born *ex nihilo*, without inspiration from existing works. As Cadmium is an attempt to realize a model across a number of ‘layers’ of systems research, it has a wide range of influences. Similarly, as these are important problems, many others have sought some solutions. Some of these works dovetail with ours through convergent evolution, while others take different routes.

2.1 Signatures

The signature mechanism (§5.3) involves partitioning the collection into $N$ partitions and using atomic operations on a bit string of length $N$ to mediate access. This is, in essence, a generalization of multiple exclusive locks.

The quest for the ideal methods to control access to some computational object has been going on since Computer Science was in its infancy. Contributions have come from luminaries such as Dijkstra [28], Lamport [53] and others [25, 32, 62].

One of the most notable features of our signature scheme is its ability to arbitrarily change the granularity of the locks on a given data collection by changing a single parameter. There have been other mechanisms proposed that have similar virtues, such as DomLock [49] and others [56, 79]. Though we seek the same property, we take a very different approach. Instead of an organization of separate locks, we collapse the entire set into the single bit-string. This gives us the
definite advantage of being able to set multiple ‘locks’ simultaneously\textsuperscript{1}. Our technique does admit ‘false positives’ and puts us on the performance/accuracy tradeoff spectrum. Furthermore, in all our work we vary the granularity on a collection by collection basis. There is no technical reason that multiple collections couldn’t share the same ‘signature space’, perhaps using one signature for the program that is much larger than we would allocate for a single collection. Again, this would be a tradeoff. By allowing greater accuracy in individual operations, we would be introducing the possibility of operations on different collections interfering with each other. It is one of many interesting avenues of inquiry that branch off from our work.

Others have used a similar scheme to our signature mechanism, except in implemented hardware such as Swarm [47] and Notary [84].

\subsection*{2.2 Software Transactional Memory (STM)}

Synchronization via Scheduling (§5.2), the execution model we are proposing, determines the maximum set of data items affected by a particular task and will only co-schedule tasks with disjoint sets.

On the surface this is a fairly novel approach with few antecedents in the literature. However, it has been pointed out to us [63], that our technique can be seen as a variant of Software Transactional Memory (STM) [78]. In transactional memory the programmer denotes a section of code as a ‘transaction’ and during execution if an executing thread makes memory access that conflicts with another thread, one of the threads is ‘rolled back’ and all the changes it has made are undone. These transactional regions are semantically similar to our Collection Contracts (§3.2.1). Furthermore, it can be interpreted that we are aggregating all possible conflict checks into the one at task admission time in a pessimistic manner, trading precision for the elimination of the overhead of recording and potentially rolling back memory values.

In this light, we are part of a family of research, dedicated to finding methods to make STM more efficient and practical. This family includes many proposals

\footnote{Technically this is virtually simultaneously in cases where the modification crosses the boundary defined by the maximum size of an atomic operation.}
for extension and modification of the STM process, which constitutes a rich and varied body of work. Other pertinent highlights include:

- improving transactional composition [34]
- handling dynamically sized collections [40]
- dealing with nested transactions [12, 14] ²
- harmonizing transactions with Object-Oriented Programming [82]
- using bit-string structures similar to our signatures [75]
- constructing programming languages with transactions as a core construct [22]
- attempting to improve performance by varying granularity as discussed above [57] (§2.1)

Our work differs from all of the above in two ways. The first is that our technique is, as mentioned above, more pessimistic than ‘proper’ STM. We deny execution unless there is no chance of a conflict. This does put a greater restriction on potential parallelism, but buys us the ability to actually execute elided without the need for rollback bookkeeping. Secondly, for the most part a transaction in STM is an annotation in a program, leaving it up to the programmer to determine where best to put them. In our system the transactions (the spans of the Collection Contracts) are defined as an integral part of the language and to use the language is to provide the necessary information. Furthermore, the definition of one of our ‘transactional regions’ is significantly semantically richer and inexplicably tied in with the parallel expression of the work. From the programmers description of what they want to do from the collections point of view, we cannot only protect the data, but schedule it better. In this way we schedule data, not code.

²The related concept of nested Collection Contracts required quite a bit of work to address (§4.3)
³This is a vote in favour of peer review. In earlier versions we used more ‘annotation’ style constructs and reviewers balked in no uncertain terms. This forced us to come up with a method to provide the programmer with a ‘win-win’ situation, where the programmer gave us the necessary information while only writing statements that contributed to their actual purpose for the code. Our solution turned out to be useful for so much more than just harvesting these constraints.
2.3 Computations on Collections

To talk about ‘computations on collections’ (searching, deriving new data, modifying values and modifying the structure of a group of related data) is to effectively discuss the entire foundation of Computer Science. Even if we limited ourselves to discussing processing collections in parallel, the discussion could still fill many volumes. If one were to type the name of their favorite data structure\(^4\) into Google Scholar, one would likely find pages of papers detailing parallel computations on that structure. This includes every data structure mentioned in this work, from lists \([30, 35, 80]\) to trees \([58, 64, 70]\). So we will simply reference a number of systems/packages designed for computations on data structures that are close to our work.

Concurrent Collections [21] considers the primacy of collections as we do, but uses a more streaming/dataflow approach and does not target general purpose applications.

Many of the popular systems address only ‘linear’ collections (arrays, lists, tables), but there are several successful projects that address large scale processing of graphs, such as Green-Marl [43] and GraphChi [52].

Galois [51], has goals similar to our own and is built for handling irregular access\(^5\) to collections, including graphs, but supports only data parallelism and uses a rollback mechanism similar to STM systems discussed above.

2.4 Parallel Query Processing

Query processing is the complement to computations on collections. It involves a set of well defined components that can be assembled to describe questions about a collection of data and a set of methods to, as efficiently as possible, produce answers. Of course, wherever efficiency is sought, potential parallelism will be investigated.

In terms of query formulation, there is likely no more successful standard than SQL(initially SEQUEL) [23], descendents of which are in use on many millions of systems today. It is likely that, unless you are reading this by torchlight in some

---

\(^4\)Everybody has a favorite data structure, right?

\(^5\)I.e. not embarrassingly parallel.
post-apocalyptic future, you have probably already done several things today that have caused the initiation of an SQL query on your behalf. While SQL itself has no notion of concurrency, it’s one of the primary methods of mediation between a ‘programmer’ and a database, which have been subject of intense study as regards concurrency [27, 54, 65, 68, 81]. Once again, entire books have been written on the subject.

In relation to our endeavors, the most prominent system for query processing is LINQ [61], which essentially embeds SQL in the C# language. It is one of the cleanest and easiest ways to evaluate your structured program data in parallel. Though it is limited, as in many cases, to embarrassingly parallel operations and provides no protection from race conditions and other parallel pitfalls. While this was not the inspiration for building a language around query processing, it did convince us to use the SQL-like syntax that we employ. Its popularity and very positive reception convinced us that a system such as ours may also be well received.

Database is still almost synonymous with the table-based relational database, but there are many based around different collection structure such as Neo4j [7], which is graph-based. It should be noted that one cannot use SQL to query these ‘alternate’ database organization schemes. Though we base our query constructs on SQL with familiar commands, such as SELECT, we have attempted to make the first steps to generalizing it towards handling arbitrary collection types. This will be discussed in more detail in Section 3.2.

2.5 Parallel Languages

In terms of influence, no work has had a greater impact on our project than the Jade language [72], which annotates code to specify that a span uses only certain variables for ensuring safe parallel execution. Most of the language level details in this work are an answer to a colleague’s question: *What can your system do that you can’t achieve with Jade’s withonly clause?*\(^6\) Our answer to this question revolves primarily around operations on collections, which the reader will notice is a primary foundation of our model. While Jade is focused generally around variables, we focus on subcollections and provide significantly richer methods of

\(^6\)Thank you, Craig Mustard. You ask the best questions.
describing and interacting with them. Our more functional queries can be seen as generalization of the \texttt{withonly} clause, allowing the user to not only say \textit{the following code uses the variable \texttt{foo}}, but \textit{the following code uses only the even indices of this collection}. Furthermore, Jade is built on a C-like language whereas we started with our queries and built the whole Cadmium language around them. This, like so many things in Systems, is a tradeoff. Basing a new language on a robust and well accepted language like C not only increased its chances of acceptance by the community, but also meant that there were a lot of problems that were orthogonal to parallelism that were already solved. In our case, we spent a great deal of time\footnote{Far more than anticipated, a warning to researchers beginning a similar endeavor.} ‘reinventing the wheel’ to make a coherent syntax. On the other hand, we didn’t have to ‘fight’ with language constructs that were designed with only serial execution in mind.

As well, every parallel language exists in relation to Cilk \cite{19}, one of the first and most successful implicitly parallel languages. To drastically over-simplify, Cilk works by potentially dispatching function calls in parallel – essentially allowing the transparent transformation from serial procedural code to parallel code. In contrast, we wanted a system that had no serial ‘backbone’ and only employed classical procedural programming ‘locally’ inside our Entities (Objects/actors, \S\S 3.1.1).

Another strong influence was the robust and effective Erlang language \cite{13} which is not only implicitly parallel, but designed for fault tolerance. The Erlang memory model, where memory is isolated from executing code and changes are instigated by message passing, is very similar to our own. We take it a step further by generalizing messages to semantically rich, internally stateless queries (\S\S 3.2). Similarly, we were influenced by the Barrelish manycore Operating System \cite{77}, which uses messages as a semantic construct, but will take advantage of shared memory when possible to reduce overhead by potentially executing a message initiation in a manner much closer to a classical function call – a technique that we use widely in the code that Cadmium produces.

There has also been a great deal of output in terms of parallel languages from the function programming community, as the side effect-less nature of functional programming makes it a natural fit for safe concurrency. These languages range
from Multilisp [33] in 1980s, through to the increasingly popular Elixir [55], which runs on the Erlang virtual machine, and includes parallel variants of existing functional languages such as Concurrent Haskell [48]. We note the successes coming from this community and let it serve as inspiration – attempting to mediate flexible and performant, but side-effect inducing, procedural code with internally stateless, descriptive queries.

There are a number of languages that are not explicitly parallel, but contain deeply ingrained concurrent elements. These include the Go language [3] with its popular ‘Go Channels’ and Clojure [2], which contains an STM implementation ‘out of the box’. Similarly, as with Concurrent Haskell, many have built parallel versions of existing procedural languages, such as Deterministic Parallel Java [20].

Though we bemoaned the lack of robust support for ‘complex systems’ in our introduction, this is not to say that there has been zero effort in this direction. Recently, the Unity video game engine has incorporated DOTS [8] which works to automatically parallelize the C# code that the engine uses by using code analysis and compiler optimizations. As with our system, it will detect potential conflicts implied by the code, but unlike Cadmium it will not employ a method such as SvS, but simply alert the user and forbid the operation.

2.6 Actor Model

We draw inspiration from the Actor Model [41] where the program is composed of actors, which can have encapsulated state, but only modify other actors by message passing. This model has been applied by others to address different aspects of concurrency, such as task parallelism [45]. We use this as the basis for our Entities (§3.1.1) and message passing (§3.1.4).

2.7 Deadlock Detection

Work has been done utilizing static analysis techniques to detect deadlock [67, 83], including the generalized GoodLock algorithm [10, 37], which broadly resembles the one we derived (§4.3) before discovering it in the literature.
Chapter 3

Expressing Intention

In this Chapter we will detail the first third of DnC principles: allowing the expression of programmer intentions. We start with a brief overview of the Cadmium language to give the rest context. Following that, we discuss the integral query syntax and the semantics of how query results are handled. Together, the initiation details and scope of query handling form a collection contract, which is integral to how DnC-conforming programs are written. These concepts will inform the final part of this chapter, where we discuss isolation semantics.

3.1 Cadmium Language Overview

As we stated previously (Ch 1), our goal is to explore, elucidate and justify the DnC model. This required that we realize the concepts in functional code. The result of this was Cadmium, an implicitly parallel language designed for the purpose.

We had many choices to make to ‘fill in the gaps’ and many of our choices were arbitrary, being orthogonal to the requirements of our model. An exhaustive description would only serve to obscure instead of inform. We will briefly outline the basics of program organization and general syntax, while hopefully avoiding too much superfluous detail. The role of entities and queries is core to illustrating our model, while the fact that all of our integer types are implicitly 64-bit is not.

Cadmium is an implicitly parallel, statically/strongly typed language, very roughly based on the Actor Model and heavily inspired by our experience with many pro-
cedural languages such as C++.

### 3.1.1 Entities

The central unit of organization in Cadmium programs is the entity. Analogous to a class in C++, Java, etc, an entity describes an aggregation of some data and/or procedural logic. As with objects, the entity block is a ‘blueprint’ that can be used to define multiple instances.

```plaintext
// define a type of message as a named tuple structure
message NewValueMessage <number@int shouldAdd@bool>

// define an entity
definition Example {
  // local member, inferred to be of integer type
  primitiveMember: 23
  // local collection, list of booleans
  collectionMember@CDList[bool]

  // define a method that receives an int
  addValue(value@int) {
    memberInt += value
  }

  // receiver for the named message
  NewValueMessage → {
    if $ .shouldAdd {
      // invoke the method with the value in the message
      addValue($ .number)
    }
  }
}
```

**Listing 3.1:** Example Cadmium Entity

A basic entity is shown in Figure 3.1, which contains an example of its primary facets.

For the sake of completeness, for those readers unfamiliar with C-style languages, the // denotes a line comment, where text is ignored by the compiler until
the next newline character (the block comment style /*…*/ is also supported). When we refer to a block, we are indicating the code delimited by the pair of { and }.

Starting at line 7, local members are defined. The colon is used to denote an initially assigned value and since Cadmium makes heavy use of type inference, the type is unnecessary. Inside code bodies, the colon syntax is used to define a new variable, similar to the `let` keyword in other languages, such as Rust and Swift.

The member below shows a case where the type cannot be inferred as the member has an implicit default. When defining a variable the `⟨typename⟩` is used to denote the type. The language supports the expected primitive types: integers, booleans, floats, etc. In this case, the member is another entity type. Unsurprisingly to those who have read our introduction, the language supports a number of collection types, as built-in data types (much more on this below) and collections are a specialized form of entity (also below). In the case of this example, the member is a CDList, which is a list in the same sense as in C++ or Python: a dynamically sized and indexable series of values.

The code starting at line 13 shows a method definition, which should be familiar to any programmer of an object-oriented language. In this case, a return type is omitted – equivalent of a C/C++ ‘void’ method. If a return type was needed it would use the `⟨typename⟩` suffix to indicate this.

Finally, the signature and block beginning at 18 demonstrates a receiver, code designated to, as the name suggests, execute when a given message is received. Cadmium makes heavy use of tuples and messages are one such use. The message `NewValueMessage` is defined on line 2, which uses the standard tuple definition syntax. Tuple definitions can be named (the message keyword is, at present, syn-

---

1 All collections in Cadmium have the CD prefix. This is probably due to the fact that the author was investigating Objective-C (amongst other languages) during development and liked NS prefix. This was, in retrospect, probably unnecessary. Though it does make it easier to easily differentiate user defined types from built in types at a glance.

2 Cadmium takes advantage of the full unicode character set, so → is a valid operator in Cadmium, though for every non-ascii command there is an equivalent combination of ascii characters (→in this case). We realized that one of the great unexpected challenges of the new language designer is the lack of matched delimiters in ascii. We found a lot of use for the more exotic delimiters, such as the French Guillemets to make our code aesthetically pleasing, even if it required us to add macros to our IDE.
tactic sugar to name a tuple type) and have named elements, such as `number` and `shouldAdd` in the example. These names are optional and tuple members can be accessed by their index as well. Cadmium employs a form of structural typing when dealing with tuples, in that any tuples with the same member types can be used interchangeably. Names are only used in cases where disambiguation is required (this is most common in the case of empty messages (<>), which are often used to invoke behavior that doesn’t require parameters).

A reader may question the difference between a method and a receiver. Why have two very similar ways of invoking code with a set of parameters? There are a few reasons for this, but to fully answer this we first need to talk about the Cadmium memory model.

As we have stated above, we want to take advantage of the benefits of shared memory, but rigorously mediate all accesses. All data and code in the program is owned by some entity. An entity has unfettered access to the data it owns, but must request data through the message passing interface (which we generalize to the query interface, discussed below). So the primary difference between a receiver and method is a matter of access scope. An entity can receive a message no matter the relationship between sender and receiver, but an entity can only invoke a method on an entity that it owns. This differentiation provides a distinct semantic separation between local interface and global interface.

Additionally, during design we concocted other processes that were unique to receivers such as establishing a channel, which in our model was an ephemeral collection that was shared between entities. However, as our work progressed we discovered that we could express our final goal of a viable prototype without these channels and to simplify things, we omit them from the discussion.

The syntax for method invocation of a entity’s own methods is shown on line 21, which is the same as normal function call. For owned entities, Cadmium uses the dot syntax

\[
\langle \text{entity} \rangle . \langle \text{method name} \rangle (\langle \text{parameters} \ldots \rangle)
\]

common to most Object-Oriented languages such as Java and Python\(^3\).

The above is the first of several expression schemas we will give in this chapter.

\(^3\)Or to partially mollify the OO language purists: languages with quasi-object-oriented facilities, such as Python.
3.1.2 Managers

Given our restrictions on where data lives that we detailed above the question naturally arises: How do entities ‘find’ each other if arbitrary data isn’t allowed in global scope? The answer is managers. Managers are a specialized type of entity that implicitly implements the singleton pattern [31]. That is to say, when a manager is defined it is automatically instantiated on program start and there will be exactly one instance present throughout execution.

The name was chosen specifically to denote that it is responsible for controlling one aspect of the program. For example, from our introductory story, Pat may have constructed a Rendering manager, Sanjay a Network manager and Kamiko a Scripting manager. We will see below how the managers make up a critical part of program flow.

Another example from the Cadmium Standard Library is console output. Every Cadmium program has a manager, `STDOUT` that ‘manages’ the currently connected console output stream. To display a line of text the programmer would write

```
STDOUT ← "some informative text"
```

using the message sending syntax (§3.1.4) as the `STDOUT` manager receives a message containing a single string for display. The single primitive is interpreted as being equivalent to a tuple of length one.
The basic syntax for a manager is essentially identical to a basic entity, but replaces the `entity` keyword with `manager` in the definition. The `manager` keyword can also be used with an existing entity definition to create a named manager instance of that entity. This second form allows us to create manager collections, which provides a segue to finally discuss collections.

### 3.1.3 Collections

From a certain point of view, Cadmium entities are trivial collections, serving as aggregate collections of their members. Their members can be queried (discussed below) and modified via messages. The key difference is that ‘official’ collections can have dynamically sized data.

In the development of our prototype, it was decided that, to avoid overextending finite development resources, we would only have collections that were defined in the language, though obviously we foresee that a completely realized Cadmium (or its descendents) would have user specified collections. The current set of collections are implemented via a special application of our C++ interface (§6.2). The prototype, as presented, supports multi-dimensional arrays, dynamic lists, sets, graphs, digraphs and trees.

While the actual definition of new collections is outside the scope of what is required for the demonstration of our model, interacting with them is central to our work and we go into detail below(§3.2).

### 3.1.4 Message Sending and Program Flow

Message sending uses the `sendto` operator (←) with the syntax

\[
\langle targets \ldots \rangle \leftarrow \langle message tuple \rangle
\]

where `targets` is a list of entities.

Note that this implies a synchronous or blocking message, meaning that the calling code will be suspended until the delivery is complete. The runtime engine has been built with support for asynchronous messages, but they turned out not to be required to express our proof of concept.

The reader may wonder how our applications execute in parallel without asynchronous messages. There are three facets of the system that we use to achieve
parallelism.

The first is parallel query execution and query delegate blocks, which we will discuss below (§3.2).

The second method involves the use of broadcast messages. A message can be sent to every member of a collection (or view (§3.2.4)). The delivery will be potentially in parallel, depending on the results of the static analysis (Ch 4).

Both the first and second methods are ways of expressing data parallelism — applying some homogeneous logic to a single collection (though the broadcast delivery may ‘cascade’ to items outside the target collection). Task parallelism, heterogeneous logic potentially applied to potentially multiple collection, is expressed either through specifying multiple targets for the message or through the interplay between managers and a special kind of broadcast messages: system messages. System messages are messages that are sent from the scheduler to every manager that has subscribed to them. A manager is considered subscribed to a system message if it has a receiver defined for that message.

The scheduler uses a message provenance tracking system to track any cascade of messages. In brief, when the receiver of a message \( A \) sends a message \( B \), this creates a tracked dependency between \( B \) and \( A \). We do this at runtime to account for the asynchronous messages supported by our scheduler. In this way the scheduler is aware of when all transitive consequences of a message have been executed, which we refer to as the message being complete. This allows us to have another variation on message receivers: completion receipts. A completion receipt will be delivered to an entity with a receiver of the form

\[
\sim\langle \text{message name} \rangle \rightarrow \{ \text{statements} \ldots \}
\]

which is identical to the general receiver syntax with the addition of the leading tilde. The scheduler will send a receipt when a sending of the named message is complete.

We consider the above, like much of this work, to be a core for a fully realized system of flow control for parallel programs. It is sufficient to realize our prototype and justify our proposals, but also serves to be an avenue for further inquiry\(^4\). In the prototype, completion receipts only apply to system messages and managers.

\[^4\text{Also, completion receipts are a partial fulfillment of our life-long desire to define a meaningful ‘comefrom’ to complement the goto statement.}\]
Currently Cadmium defines 3 system messages: Initialize, Execute and CleanUp. These will be delivered in that order, each after the previous system message (and their corresponding completion receipts) is complete.

So the combination of system messages and managers allows us to do away with a single entry-point (‘main’ statement). Each manager that subscribes to the Initialize system message will begin execution, potentially in parallel, upon program initiation.

As a final note, a system message can be reinitialized by sending an instance of that message to the program. Every Cadmium program defines a manager-like global construct with the same name as the program (this construct provides various other ‘program level’ functionality such as retrieving command line arguments). When program representation receives a system message it will cause that message to be rebroadcast when the current broadcast is complete, as we want to retain the property that only one system message is active at once. We find this to be crude, but effective enough to satisfy the needs of our prototype and consider it yet another prime candidate for further exploration.

3.1.5 Accumulators

If there are two concurrent modifications to the same value, we are at risk of a state conflict. As the reader will be aware by this point, the central thrust of this work is to ensure that this circumstance never occurs by not coscheduling two tasks where this would be possible. However, there are a non-trivial number of circumstances where the final result is independent of the order of modifications and the result is never consulted during the process. As a simple example, consider a process that maintains a count by incrementing an integer. Because addition is commutative, no matter the order that these increments are applied the result is the same.

It is not feasible to detect these circumstances by static analysis. It is, again, a property that must come from the programmer’s head. To fully exploit these order independent, write-only operations, we introduce accumulator types, which can be modified in a parallel context. It will not cause the static analysis to consider this a write, but will consider a read to be a serial execution inducing operation.

---

5 We generally implement these accumulators by storing results ‘thread locally’ and combining at the end. Similar techniques exist in the literature [85].
For primitives, we support an accumulator `int` as an integer type that supports addition and subtraction as with our example above. For collections, the built-in support includes accumulator sets and accumulator lists.

### 3.2 Collection Usage, Collection Contracts and Queries

#### 3.2.1 Collection Contracts

As we discussed in the introduction, at the highest level we want the programmer to tell the scheduler: *I want to use this part of the collection* and correspondingly *I am done with that data*. We want to be able to do this without compelling the programmer to add a bunch of extra markup or put the onus on them to figure out the optimal placement of these statements.

Specifics will follow, but these statements will be used to bound a Collection Contract. We then guarantee exclusive access to the subcollection described for the interval between the two statements. Under DnC all data-accesses must be part of a Collection Contract, and thus mediated by the scheduler.

Every model of parallelism defines the demarcation of critical sections of code. In the standard threading model, one uses mutexes and semaphores; monitors use method bodies and Software Transactional Memory [78] uses specific transaction delimiters. Similarly, DnC uses Collection Contracts to mark critical sections and they serve as the primary unit of scheduling.

Collection contracts, unlike other ways of demarcating critical sections, are inherently tied to the data being protected within. For example, STM allows specifying where the transaction begins and ends, but those demarcations do not tell us what data the transaction protects. Using locks has the same limitation. As a result, any static analysis method would struggle extracting programmer intention at compile time. Jade’s `withonly` clause [72] is perhaps the most similar to our collection contracts, but since Jade was an extension to C, statically verifying that the enclosed code respects the declared intentions would be a quite difficult task.

---

6 Except, as described above, for local variables and, in Cadmium, entity member variables (§3.1.1), which are accessed in a traditional way to avoid burdening the programmer at the cost of more static analysis.

7 In some cases, such as map/reduce [26], this is to some degree implicit.
hope to address this shortcoming by using declarative syntax to express intended data accesses.

In the work that preceded and inspired this, Synchronization via Scheduling (SvS) [16], we considered the data being used as essentially a ‘cloud of pointers’ without reference to how they were organized. When we began to look at how to make our algorithms better, we realized that it wasn’t a matter of needing more clever algorithms\(^8\), it was a matter of needing more information. By incorporating types and collections into our model, it made for much better estimates, eliminated runtime bookkeeping and allowed finer grained decomposition of the program and its access patterns.

A Collection Contract has three phases:

**Isolate** Determine what data is needed, i.e. the bounds of subcollections needed for the operation.

**Modify/Derive** Transform isolated data and/or produce new data.

**Release** Conclude the operation (this is often implicit).

### 3.2.2 Declarative Syntax for Collection Contracts

As we outlined above (Ch 2) SQL is perhaps the most widely adopted and time-tested language for declaratively expressing data accesses. The separation of what the programmer wants to do with the data from how they wish to do it, allows database systems to implement a variety of query optimizations underneath the covers. Expecting that parallel programmers will write their code using SQL is a rather bold assumption, but we capitalize on the fact that C# LINQ [5], essentially as subset of SQL, has gained wide adoption as an extension to an imperative programming language, due to its expressiveness.

Cadmium builds on the success of LINQ by adopting a very similar syntax; the difference is that unlike C#, and more similarly to Dryad [46], we capitalize on declarative aspects to deliver transparent parallelism.

The following is an example of the Cadmium query the programmer would write to express the boundaries of a collection contract:

\(^8\)Though we do have these now as well, if we can be allowed a little self-congratulations.
Listing 3.2: Cadmium contract: Isolate and Release phases

The contract is the span between the initiation of a query and when the view is released. Queries comprise the Isolate step of a Collection Contract. The Modify/Derive phase can be expressed almost identically to a standard imperative program. The query results are represented by a view, borrowing terminology from the database community.

SQL, and by extension LINQ, are designed for relational data. Though LINQ queries can be issued on any collection that implements the IEnumerable interface, the examination of this interface reveals that it assumes tabular data and would not effectively support data structures, like graphs. To address this shortcoming we extend the LINQ-like syntax to support the concepts of collection structure and content.

A collection’s content is the data itself. Its structure is the organization of the content and the description of how each data item relates to the others, such as the ordering of a list or the hierarchy of a tree. This second aspect includes data concerned with these relationships, such as next pointers or vertex labels.

To effectively support queries concerning both content and structure, we provide three categories of queries:

```sql
   | SELECT foo FROM bar | → view1
   | release view1
```

Figure 3.1: A subtree substructure query on a tree
Content Queries based on data attributes, such as finding all elements with an \( x \) coordinate above 10;

Structural Queries based on the structure of the collection, such as selecting array elements 3 through 5 or the subtree rooted at node \( v \).

Hybrid Queries a combination of the two, such as finding all elements in a subtree that have a \textit{dirty} flag set to true.

Instead of attempting to create a grammar that incorporates every possible collection, we define \textit{atoms}: parameterizable, collection-specific logic. These atoms can be seen in the \texttt{SELECT} query which has the syntax:

\[
\text{SELECT} \left( \left[ \text{cardinality} \right] \langle \text{bound variables} \rangle \left| \left\{ \langle \text{substructure atoms} \ldots \rangle \right\} \right. \right) \\
{\text{FROM}} \langle \text{source collection} \rangle \left[ \right. \left. \left[ \text{WHERE} \langle \text{filter expression} \rangle \right. \right. \left. \left. \text{VISITBY} \langle \text{visit policy} \rangle \right. \right. \left. \left. \text{ORDERBY} \langle \text{order policy} \rangle \right. \right. \left. \left. \text{AS} \langle \text{output names} \rangle \right] \right)
\]

where the various elements are as follows:

\textit{cardinality} the number of results returned, respecting visit order policy (see below). \texttt{SELECT FIRST foo FROM myTree} yields the first item found by a depth-first search (the default tree visit policy). By default the \texttt{ALL} cardinality is assumed (\texttt{*} in SQL).

\textit{bound variables} the placeholders variables for specifying a node. \texttt{SELECT a FROM arr WHERE a.val == 5} describes all elements with a \texttt{val} of 5. SQL uses similar concept to describe columns, whereas variables refer to collection elements, and the dot notation is used for properties.

\textit{substructure atoms} collection specific parameterized identifiers for indicating a query on a particular type of substructure of the collection. Essentially, while every collection type has at least one basic, indivisible element type (nodes or edges for graphs, elements for list, etc) there are emergent structure types that involve

\footnote{Note that we do not implement common SQL mainstays such as COUNT, AVG, etc. These can be accomplished using other features. Adding them and others would not be difficult and expanding atom functionality is part of our future work.}
multiple of these basic elements and in some way are dictated by the structure of
the collection. Examples are easy to see in recursively defined collections such as
trees and lists where the composite pieces of the structure are also structures of the
same type (i.e. subtrees for trees and sublists for lists, which are also trees and lists
in their own right\textsuperscript{10}). These substructures are important enough that any work with
collections that aims to encompass their general use needs to incorporate them into
their ‘vocabulary’. The substructure atoms are collection specific and are currently
defined by the Cadmium Standard Library.

As an example, the following query, \texttt{\{SELECT range\{n,n+2\} FROM arr\}}, where
\texttt{arr} is an array, allows us to isolate the three contiguous elements. These can
also contain bound variables. Figure 3.1 shows the results of \texttt{\{SELECT FIRST
subtree\{s\} FROM myTree WHERE s.val > 5\}}, making use of the \texttt{WHERE} clause
as discussed below.

\textit{source collection} the collection the query is performed on. This can include sub-
collections(§3.2.3), views (§3.2.4) and pseudo-collections which lazily-evaluated
collections such as \texttt{1..100}, which denotes the integers between 0 and 99, inclusive
(we will see this used in §7.2.3).

\textit{filter expression} a boolean valued, side effect free expression, making reference
to the names defined following the \texttt{SELECT} statement. Used to filter candidates
for the result. We have seen this already in the example \texttt{\{SELECT a FROM arr
WHERE a.val == 5\}} which will produce a set of results of exactly the elements of
\texttt{arr} where the \texttt{val} member field is 5.

\textit{visit policy} identifier for specifying the order the collection elements are evaluated
in, including for example: \texttt{reverse} for arrays and \texttt{depthfirst} for trees and graphs.
As with substructures, these are collection specific and defined in the Cadmium
Standard Library. Each collection defines a default visiting order. For example,
the \texttt{CDList} collection type has \texttt{randomaccess} as the default visiting policy which
means that it will process the elements in no specified order.

\textit{order policy} the ordering for the results, which may differ from the visit order. A
predicate, such as \texttt{a.val < b.val} can be provided. Importantly, the results can

\textsuperscript{10}With, perhaps a little definitional slight-of-hand for the null element: empty tree or empty list.
be marked as unordered, indicating that the ordering is unimportant. The combination of visit policy and order policy are mostly applicable to delegate blocks (§3.2.5) and will determine if parallelizing this query is possible and if so, what methods are applicable.

**output names** gives names to the items of the output if a bound variable has not been used. This is primarily useful for delegate blocks which are to be discussed below (§3.2.5).

A query itself is a special type of a message so it uses the bidirectional message passing syntax\(^{11}\):

\[ \langle \text{target collection} \rangle \leftarrow \langle \text{query} \rangle | \{ \langle \text{view name} \rangle \mid \langle \text{processing code} \rangle \}. \]  

In this case the _FROM_ is omitted in _query_. The version with the _FROM_ clause omits the sendto operator and is syntactic sugar to ensure that queries on constructs such as pseudo-collections have a clearer form.

The output is optional as there are ‘queries’, such as _INSERT_ and _DELETE_ that change the state of the collection without needing to ‘return’ anything to the invoker. They will be discussed below (§3.3).

The first alternative for the output is a view, discussed in §3.2.4 and the second is a delegate block, discussed in §3.2.5.

### 3.2.3 Memberships and Subcollections

Under the DnC model, every entity\(^{12}\), collection or primitive is a global manager (§3.1.2) or is in exactly one collection.

However, there are times when a programmer will wish to demarcate some elements of a collection in a specific way. Cadmium allows the definition of subcollections which are collections that contain references to elements of a particular collection\(^{13}\). A subcollection functions identically to a collection, though it may have different structure\(^{14}\). Collections may be composed, such as the quadtree

---

\(^{11}\)This is currently the only allowable use of the bidirectional message syntax, but we expected that to change very shortly into our future work.

\(^{12}\)Aggregate object, potentially with behaviour attached (§3.1.1)

\(^{13}\)For those wondering, yes this does create additional overhead, but the compiler is being designed to emit only the necessary code when required.

\(^{14}\)Currently only sets and lists are supported as subcollection types in Cadmium, but work proceeds on expanding this.
which is a tree of resizable arrays.

A subcollection must be explicitly marked with its origin collection. If we wanted a list that contained the nodes of a tree, arb, the type would be written \( \text{CDList}[^*\text{arb}] \). When any function-like construct receives a subcollection as a parameter, that parameter must also indicate the origin collection. The consequence of this is that \textit{it is possible to know at compile time which collection is being accessed in any query}. This solves a large number of potential aliasing issues.

This does require the programmer to explicitly create a hierarchy of data in their code. Though in our experience, most complex classes have a single intended use and most programs are built with collections for each of these classes. It should be noted that the \textit{one collection rule} doesn’t prohibit moving items between collections that contain the same type. Even when the aggregate data items are used only singularly, the worst the programmer needs to do is define a manager set and reference them from there. Interestingly, this enforced organization eliminates a few memory errors. Given that the implementation of the collection is correct, it’s impossible to have an ‘orphaned’ allocation, as it must exist in some collection, somewhere.

### 3.2.4 Views

As shown in Listing 3.2, the result of a query is stored in a view. As in most database implementations, the creation of a view does not imply that the data is copied. Views are the key construct used to enforce transparent parallel access to collections using mechanisms described in Ch 4 and Ch 5.

The view is released implicitly when it goes out of scope; the programmer need not worry about ‘dangling views’. Furthermore, a programmer can elect to end the contract explicitly with \textit{release}. As this keyword implies, the termination of a view corresponds to the Release phase of a collection contract.

While a view cannot be stored directly, it can be retained by transforming it into a subcollection by \textit{release} \( \text{arrView} \rightarrow \text{arrSub} \). In which case, \( \text{arrSub} \) would need to be queried in order for the programmer to obtain access to its contents.

This process is often used to make references to a specific element to be retained for future reference. These function as a bit of a hybrid between smart
pointers and optional types. The latter because they may be ‘empty’. We called this individual element subcollection creation single item stored view\textsuperscript{15}. It was common enough that the compiler emits specific support for it, without all the extra baggage of actually being a subcollection.

3.2.5 Delegate Blocks

In many cases, it is desirable to directly process the elements resulting from a query. This is similar to the ‘apply’ operation common to many languages and systems. This is shown in the following example:

```
1 i: 7
2
3 |SELECT range(i, i+2) FROM arr ORDERBY unordered AS v| → {
4   if v == 5 {
5     STDOUT ← "We found a five in range"
6   }
7 }
```

This is one way we can express Data Parallelism in Cadmium. The policy we use for all operations is that if results are ordered, they will be processed in that order so as to give the programmer more control and fewer unpleasant surprises. If unordered, we can take advantage of that and often execute in parallel. Similarly, if the collection is non-linear, such as trees, the delivery can still be performed in parallel, where ‘parents’ are evaluated before children. Note that if the results of a query are non-disjoint, no two overlapping elements will be processed simultaneously.

3.2.6 Isolation Semantics

By default, the view is the subcollection – by which we mean that once the query can be satisfied, the invoking code has direct access to the memory of the elements described. As a consequence, we can directly reap the performance benefits of a shared memory system with no unnecessary copying. This is one of several ways that our model differs from Software Transactional Memory. There are no

\textsuperscript{15}This really needs a snappier name or at least cute short form. SIS View? Solo View? Another line item for our future work.
rollbacks or failed operations in our model\textsuperscript{16}.

From the point of view of the contract invokers, it is as if they are the only entity accessing that collection. No changes can be made to the isolated data from outside the contract until it is complete. Furthermore, no contract that overlaps a contract in its Modify/Derive stage will be satisfied, allowed to proceed, until the first contract is complete. Effectively, changes will not be visible outside the contract until its completion.

This set of policies gives very strong isolation guarantees, and the set of non-nested contracts are serializable. However, the DnC model, and thus Cadmium, does allow for nested and overlapping contracts as in

\texttt{1 | SELECT foo FROM bar \rightarrow view1}

\texttt{2 | SELECT foo FROM baz \rightarrow view2}

\texttt{3 \hspace{1em} release view2}

\texttt{4 \hspace{1em} release view1}

This is two contracts in terms of isolation. The invoker has unfettered access to the results of the \texttt{bar} query upon satisfaction of the first \texttt{SELECT} statement; however, other entities may modify \texttt{baz} until the satisfaction of the second select statement. The system does not guarantee the repeated reads property between sequential contracts, only within them. Thus, a second query to \texttt{baz} after \texttt{release view2}, but before \texttt{release view1} may show changes made under other contracts\textsuperscript{17}. This manner of strong guarantee and nested ‘transactions’ leaves open the potential for deadlock. We detail how we prevent this in Section 4.3.

There is an interesting sticking point between wanting to give strong isolation guarantees, giving the programmer tools to reason about their code in a more serial manner and augmenting collections. For example, consider a task holding a collection contract that is the result of a content query and another task adds new elements to that collection. If the new element fits the criteria for inclusion in the contract, the collection contract is out of sync with the current state of the collection. We decided in the end that our isolation guarantees would not cover this circumstance and collection increasing would not be covered under the 'terms' of

\textsuperscript{16}Non-concurrency related exceptions are part of our future work.

\textsuperscript{17}This is one of the few cases where programmers may need to be aware that they are programming in parallel. It was felt that this compromise between potential performance and surprising the programmer was justified.
a collection contract. That is to say, that if one were to secure a view to all items
\[ \text{WHERE } x.\text{val} > 5 \] and before it was released another operation requested to add a
new element with a val of 7, that request would be granted, but the first operations
view would not be augmented.

We are deeply considering adding a facility to specify the level of permis-
siveness with respect to this behavior, likely enforced by something similar to the
collection reservation mechanism (§5.5).

### 3.3 Further Queries

While *select* is the cornerstone of our query system, it is not the only type of
query message supported in Cadmium. These other queries will be easier to define
now that we have passed the previous discussion.

There is a bit of nomenclature nuance here. By the dictionary definition, a
query is a request for information and not, for example, a request for action. So,
*insert* described below, which adds a new element to the collection is not a request
for information, but falls under the umbrella of what we’ve been calling our query
system. However, as SQL has set the precedent of calling such things queries, we
will continue to do so.

#### 3.3.1 INSERT

As the name implies, *insert* is the mechanism by which items are added into a
collection. The *insert*, query message uses the following syntax:

\[
\text{INSERT}\ (\text{NEW} \ [\text{initiation message}] \ | \text{FROM} \ \langle \text{source variable} \rangle) \\
[ \text{INTO} \ \langle \text{target collection} \rangle ] \\
[ \text{BY} \ \langle \text{insertion policy} \rangle ]
\]

where the various elements are as follows:

*initiation message* the mechanism to create new entities inside the collection, a
rough approximation of the *emplace* series of method calls in the C++ STL.
While not pictured in Figure 3.1, a receiver can have an optional *new* keyword that
marks it as the message that can be delivered on instantiation, making it similar in
function to the C++/Java/etc constructor. So the initiation message must match the
message type specified in one of these *new* receivers.
**source variable** used to insert an already existing element into the collection.

**target collection** the collection to be inserted into.

**insertion policy** analogous to the **VISITBY** policy in **SELECT** (in fact, **BY** is simply an alias for **VISITBY**), describing what part of the collection to insert into, with reference to its structure. For example, if we wish to add an element to the end of a list:  

```
INSERT 451 INTO myList BY append.
```

Note that the CDList also has an append method, so if the list was local, the programmer could write `myList.append(451)`.

Currently, if a local item is inserted into a collection, it is done by copying, so the original item is retained. Advanced future work includes more robust language features and optimizations for moving items between collections.

### 3.3.2 DELETE

If you can put something in, you need to be able to take it out again. **DELETE** is the complement to **INSERT**, and it removes items from a collection (and from the program in general). The query message uses the following syntax:

```
DELETE ([cardinality] ⟨bound variables⟩) { ⟨subcollection/view⟩ } [ FROM ⟨target collection⟩ [ WHERE ⟨filter expression⟩ ] [ VISITBY ⟨visit policy⟩ ] [ ORDERBY ⟨order policy⟩ ] ]
```

In general, this follows the same form as the **SELECT** statement. The only exception to this is the second alternative following the **DELETE** keyword. In this case the programmer can submit a subcollection or view taken from that collection and this operation will remove all elements common to both. We use this, for example, in mesh refinement 7.2.2 when we take a subcollection (a contiguous subgraph) that we built in an earlier step and delete all vertices and edges in the original graph, so we rebuild it with better properties.

Note that, if the target in question is a subcollection, which the reader may recall is essentially a collection of references to another collection, with potentially different structure; then the ‘reference’ is removed from the subcollection, but the original collection remains unchanged. However, deleting from a view will remove the item in the original graph, because the view is the collection. This highlights
one of the distinctions between these two constructs.

As stated above, this deletes the data item entirely, but it would not be much more than trivially difficult for it to ‘return’ the deleted item giving it an ‘extraction’ aspect, as well.

### 3.3.3 UPDATE

There is a very common pattern in software engineering. It has many names, but we like to call it *double buffering*. This is the case where for a given set of data, we build a new version of the data, generally for a ‘next iteration’ with reference to the old. When the new one is complete, the old one is discarded. This types of procedure is a desirable target for parallelization with applications of processes like the Stencil Pattern [59]. For this we provide the `UPDATE` query message, which we use to great effect in the PageRank evaluation (§7.2.1).

The `UPDATE` query message is very similar in structure to `SELECT`, for hopefully obvious reasons, with the following syntax:

```
UPDATE ([cardinality] ⟨bound variables⟩ | { ⟨substructure atoms . . .⟩})
   [ IN ⟨source collection⟩ ]
   [ WHERE ⟨filter expression⟩ ]
   [ VISITBY ⟨visit policy⟩ ]
   [ AS ⟨output names⟩ ]
```

where the elements are defined identically to `SELECT`. `ORDERBY` is omitted, at least for our initial version, as delegate blocks for `UPDATE` could be accomplished with an equivalent `SELECT` query.

The primary difference lies in the generated view, which is a specialized update view. An update view behaves like a standard view, except it has two predefined ‘members’: `current` and `next`. `current` has the contents that currently exist in the collection, and `next` is the space constructed for the derived values. `next` is also given the same structure as `current`. When the update view is released, the `next` values will be ‘committed’ in place of the previous values which will then become visible outside of the contract.
3.4 Conclusion

In this chapter we have given a brief outline of the Cadmium language, highlighting the features most pertinent to demonstrating our model. We constructed these aspects specifically to allow the programmer a wider range of built-in facilities, such as visit policies, that allow for expressive and concise operations on collections. We take every opportunity to channel the programmer into expressing parallelizable code, to sharing their intentions.

However, this is only the first step. After the programmer has been kind enough to share their thoughts with us, we need to extract them appropriately. Our constructs expose numerous opportunities for parallel execution, but not all of them are safe, especially in combination. In the next chapter we will detail the compilation process and the static analysis we use to take advantage of our purpose-built language elements.
Chapter 4

Extracting Intentions: Compiling Cadmium and Static Analysis

In the previous chapter, we described the fundamental aspects of the Cadmium language necessary to embody the DnC model. We focused on capturing more of the programmer’s intentions with rich, composable constructs that describe operations on parts of collections. Importantly, these constructs focused on the what and when of their operations, without compelling the programmer to tell us how (beyond specifying high level constraints such as visit order). However, this was only the start of battle. The programmer is, in a sense, ceding control to us in matters of parallelism and it becomes our responsibility to ensure their programs execute safely and efficiently. We have two opportunities to make the best use of the intentions that have been expressed to us: during compilation time (the subject of this chapter) and during runtime (the subject of the next).

4.1 Parallel Opportunities

We specifically designed the constructs in the previous chapter to allow parallelism whenever possible. These possibilities can be broken down into two distinct categories: data parallelism and task parallelism. As stated previously, the difference between them lies roughly in the logic employed. Data parallelism is applying one process to a set of data, where task parallelism is the simultaneous
execution of different processes that may or may not act on the same data items.

There is a third major form of parallelism (assuming, as we are, limiting our discourse to multicore shared memory architectures): *instruction-level parallelism*. This is where some hardware instruction produces more than one effect simultaneously, such as the processor supported SIMD (Single Instruction Multiple Data). We do not address this directly here, but given our transpiling process (Ch 6) we may still reap the benefits of a more mature compiler that may emit such instructions.

We will briefly describe the possible opportunities we have to emit parallel dispatching code when compiling a Cadmium program.

### 4.1.1 Task Parallelism

**Managers/System Message** As detailed above (§3.1.4), when a system message is received, the corresponding receiver body of the manager is ready to run. As we know the identities of receiving managers at compile time, we simply emit a series of statements to create a task for each recipient body and submit it to the scheduler. This way, each body will be executed in parallel, to the limit of available CPU resources.

**Heterogenous Message Targets** Recall (§3.1.4) that the sendto statement may have multiple recipients. In our current implementation where we only use synchronous messages, we normally transform the message directly into a function call, instead of incurring the overhead of enqueuing the message and spawning a new task for it. In the case of multiple recipients, we transform this into a series of task creation/submission statements.

The potential snag with this is that we don’t want to continue executing the body of the sender until the corresponding receivers have completed. Given that receivers can in turn be senders, we must assure that any cascading effects are taken into account. Recall, that we have mentioned previously the *message provenance system* which tracks the cascading effects of receivers sending further messages. We will discuss the composition and implementation of tasks more in Chapter 5; but for now, we will only need to know the fact that Cadmium runtime tasks can be suspended and resumed arbitrarily. This functionality is not available to the
application programmer, but we can certainly emit code to that effect during compilation. Every invocation of a receiver is associated with a *message provenance object*, which is the message that spawned it. When these objects are created, they contain references to the message provenance of the task that spawned them. In effect this creates a tree of provenance leading back to the original system message/scheduler code that initiated the cascade. Each provenance object contains an atomic counter of its child objects. When a new child is created, it is incremented and when a receiver is finished code is run that signals it to decrease. When the counter is changed to zero, if there is a suspended task associated with it, the initial sender in the case we’re discussing, that task will be resumed immediately.

In the case of multiple receivers, the initiating receiver creates the new tasks, seeded with the message data and invocation code, and suspends its host task into the reserved space in the associated provenance object. It uses architecture-appropriate memory barriers and a check to handle the corner case where all messages have completed before the suspension takes place.

This is nearly identical to the way the scheduler is invoked when a system message is complete; but instead of resuming a task, it initiates code to broadcast the next system message (which may be the broadcast of completion receipts) after checking to see if any message requesting a re-broadcast of the current system message has been received. If there are no further system messages to send, the program is complete and it initiates the shutdown process.

### 4.1.2 Data Parallelism

**Query Execution** A query itself may be executed in parallel. This is especially true for content queries that yield a view (we will discuss delegate blocks immediately below). Currently all Cadmium collections are easily partitionable, for reasons that will become obvious when discussing signatures (§2.1). When the query itself involves examining the collection, or parts thereof, these partitions can be submitted, in batches, to the scheduler, using a multipart directive (§5.1).

Invoking this behavior is contingent on the visit policy of the query\(^1\). If the

\(^1\)It guaranteed that we know these policies at compile time. Visit policies and order policies are not first class citizens in the language and so the programmer cannot, for example, assign a different policy to a variable based on some input and invoke the query with that variable.
visit policy has the *neutral* property, such as the default set visit policy, then this is always possible as no ordering has been imposed. If the visit policy has the *linear* property, such as *forward* and *reverse* policies of arrays and list, then as long as we can determine which partitions fall into the range set by the cardinality specifier we can use parallel dispatch. There is no reason we could not extend this to non-linear data structures, such as trees. For the purposes of this prototype these visit polices were sufficient for demonstration purposes.

If any ordering besides *unordered* is requested we have one of two situations. If the collection is already in that order (*forward*) or the desired ordering is implicitly determinable from the existing order (*reverse*), we then can dispatch the query in parallel, again using batches of partitions, and ‘stitch together’ the results in the requested order. If the ordering is arbitrary, i.e. defined by an expression, we can merge sort in parallel by submitting sequences of batched partitions. As the data needs to be already ordered when it is examined by the caller, ordering doesn’t prevent parallel expression in the way it does for delegate blocks.

**Delegate Block Invocation** Delegate blocks (§3.2.5) provide a method of applying a block of code directly to the results of a query. In our very first experiments writing Cadmium code, we discovered that we would initiate a query, receive a view and immediately want to iterate through it; so we created this functionality to further exploit this emergent pattern\(^2\).

Effectively, we have the same situation as satisfying content queries above, with the restriction that the programmer has told us to do something to each item *as we encounter it*. So, we add the constraint that unless the query has the order policy *unordered*, we will execute the delegate blocks serially; though we will report this to programmer to make sure this is what they intended (§6.1). This is one place where a deeper application of static analysis could be quite useful. If the block doesn’t depend on any side effect from a previous instance of the block, there is no reason to force serial execution of the block. This can be a bit

\(^2\)As we created this, we noticed the growth in the use of lambdas and closures in the general programming world, where a parameterized block of code is first class citizen in the language. These are often used similarly to our delegates, which are not first class citizens. We would like to rectify this, though we see this more in line with how C++ handles generics, by building new instances of templated code to retain the knowledge of the exact query for our static analysis.
subtle. Consider the case when the programmer has a list of items in a specific order and has a delegate block that formats them and displays them on the screen. Technically, no data in the program has been transformed, so we may infer that this is safe to parallelize. However, printing the data in a different order may or may not be wrong. We must give the programmer a choice. After much discussion, we elected to take the position that, unless the data structure is implicitly unordered, as in the set, delegate execution would be serial unless specified otherwise. Though, we do note that from a semantic point of view, we aren’t asking the programmer if they want parallel execution or not. We’re asking them about a property of the process – if they intend an order or not.

**Update Views** The creation or ‘committing’ of an update view can involve duplicating a great amount of data, which can be submitted, as with query execution, as batched partitions for parallel execution.

Many of these opportunities are always safe, such as executing a content query in parallel. However, there are cases where there is uncertainty. These include when there is concurrent access to collections and whether or not to parallelize a delegate block to an unordered query. We can’t know if these will be safe from examining the invocation alone. We’re going to need information from the entire program to make a determination.

### 4.2 Evaluating Shared Accesses

In the previous section we identified all potential sites for emitting parallel dispatch code. However, if a section of code can be executed in parallel and still yield correct results, that does not necessarily mean that it can execute in parallel without state conflicts. In the next chapter we will show our runtime technique (§5.2) for mediating collection access; however, if we blindly augmented every collection access with this protection, we would potentially incur a great deal of unnecessary overhead. If we can tell at compile time that protection is always unneeded we can avoid adding it. To do this, we need to evaluate each opportunity for the various accesses to shared data that are made in the region in question. By comparing the access domain, the set of data potentially accessed in each region, we can make informed decisions about the code we emit. We may be forced to serialize the
execution or apply the SvS runtime protection method (Ch 5). Furthermore, we can identify many cases where we can guarantee that the access domain of a region is disjoint with all others that could be potentially coscheduled, even if that data is accessed by other regions that could never be coscheduled.

4.2.1 Determining Access Domains

The first step in evaluating the safety of the regions of a program is to identify their access domains, which include all possible accesses that may be made. The regions in a Cadmium program are defined as a generalization of collection contracts. In the previous chapter, we defined collection contracts (§3.2.1) as being the span between the start of a query and when the view is released. We expand this definition to include one more case. We have stated before that the query system is built upon the basis of the message passing system. An invocation of a receiver body is considered to be an implicit contract on the entity members used. This is to say that the scheduler will ensure that it is safe for the body to be executed, based on the members it accesses, and the message will not be delivered until that can be guaranteed. The contract is initiated by the message sender, begins upon receipt of the message and ends when the body terminates.

4.2.2 Inferring the Domain of Contracts

Determining any potential data item accessed directly in a particular contract is relatively simple. Determining which statements belong to a particular contract can be done with a linear walk through the AST, starting at the query statement (or at the initiation of the receiver in the case of implicit contracts). With a little attention to release statements that are enclosed in conditional blocks\(^3\), we can determine the set of statements that are part of the contract. From this we simply analyze each expression for variable references and record if they’re a read or a write. At the end we have a list of data items accessed anywhere during the contract. Local and member variables are identifiable by consulting the parsed entity. Every other data item must belong to a global collection and the result of the one collection rule

\(^3\)As a part of enforcing the contract, if there is a release statement in a conditional block; we emit an extra check after every subsequent access to the resulting view that raises an error condition, if the view has been released.
is that we can trace back through any views and subcollections to determine which one.

This accounts for all the direct accesses during a contract, but we still need to track all indirect accesses. In this case, indirect accesses are those that are present in code outside the contract that is invoked by a statement in the contract. There are three, nearly identical, cases where this may happen: function calls, method calls and message sending\textsuperscript{4}. As we have no polymorphism, or other forms of runtime selection of executed code, we can identify the exact body of code that is called. After that, it’s simply a case of repeating the same AST analysis on the identified code body. This is, of course, recursive and breaking cycles is simply a matter of marking ‘seen’ bodies and limiting recursion based on that bookkeeping. As up the recursive stack, we match parameters to the values given at the call site and once the process completes, we have a complete description of the access domain. We retain the information gathered searching for the indirect accesses for reuse, when other contracts invoke those bodies.

This does mean that we assume that the entire source code is available at compile time, unlike, for example, C++, which breaks the code into one or more ‘compilation units’ and builds the object code for that unit and ‘links’ them together at the end. We acknowledge that this wouldn’t scale well to truly large codebases and part of future work includes deriving a scheme to bundle the results of this analysis with the compilation of some well defined subset of the code. Likely this wouldn’t be the completely independent compilation that C++ achieves, as different interrelated ‘units’ would still have to be compiled in reference to each other. At the very least, we could retain a great deal of this information between compilations or as part of reusable libraries.

\textbf{4.2.3 Analyzing Program Flow}

In order to ascertain the potential for concurrent accesses to a particular shared data item, we need to determine the constraints on when in an execution an access could occur. As a very coarse example, if collection $X$ is always accessed once at the very

\textsuperscript{4}This is why we, with regret, elected to only deal with synchronous messages for the purposes of our prototype. This limits the potential of the language, but simplifies things enough that we can complete our proof of concept to motivate research that would include removing this limitation.
start of the program and always accessed immediately before termination, we can
conclude that there is no possibility of concurrent access. Similarly, consider a
program with only two accesses to a collection \( Y \). The first access is in a contract
in receiver \( A \) and the second in a contract in receiver \( B \) and \( B \) is only invoked by
a message sent from receiver \( A \) after the contract. We can conclude that these
accesses will never be simultaneous. However, if we have a collection \( Z \) that is
accessed in contracts in receiver bodies \( C \) and \( D \), where some manager sends a
message invoking both bodies, we cannot guarantee mutual exclusion.

In order to generalize this kind of analysis we are going to examine and name
the various partitions across the time domain of a program execution. Recall that
any action in a Cadmium program begins with a system message to a manager and
that any subsequent system messages and completion receipts will not be sent until
the current message is complete. The consequence of this is that we can divide
any execution of the program into a series of ‘epochs’ that begin with some system
message broadcast. We will call these epochs phases. Each phase can be named
by the system message that initiates it and we refer to the initiating message as the
phase’s phase message. Given that the program can affect which system message
is sent at runtime\(^5\), we cannot predict the ordering of the phases in any particular
execution, but at compile time we can determine the total set of phases possible.

Once the set of phases applicable to a program is determined, we can refine the
problem to discovering potential concurrent accesses during a particular phase. For
this we need to examine every possible sequence of invocations that result from the
receiver bodies in the managers that subscribe to that particular phase message.

We know that, very deliberately, there are no accesses to shared data outside of
a contract. Furthermore, with the analysis above, we have the access domains of
each contract.

To find the potential concurrent accesses to shared collections, we need to un-
derstand the dependency relationships between the contracts that can be invoked in
the phase and thus understand potential conflicts of their access domains.

For this, we construct a directed graph where each node represents a set of
data items accessed. We initially seed the graph with one node each for the access

\(^5\)Currently the programmer can determine if the current message should be rebroadcast, but more
control will be added as we expand the program flow controls.
domains of the receiver bodies of the subscribed managers. We trace the code from invocation to invocation and build the graph as follows, tracking for cycles, so that we never visit a contract more than once per invocation:

- If one contract follows another in a single body with no multiple target messages in between, we combine their access domains into a single node
- If two contracts overlap in a single body, we combine them into one node
- If a single message is sent inside a contract, we ignore it as we already have its effects reflected in the current contract’s access domain
- If inside a contract a message is sent to multiple receivers, we split the current node $v$, duplicating its contents, into a $v_{pre}$ and $v_{post}$ such that any in edges that pointed to $v$ now point to $v_{pre}$ and any out edges that originated from $v$ now originate from $v_{post}$. If $S_1, \ldots, S_n$ are the subgraphs generated from applying these rules to the appropriate receiver bodies of the target entities, we add an edge from $v_{pre}$ to the node of indegree 0 (there should be only one) in each $S_i$. Conversely, for every node of outdegree 0 in each $S_i$, we add an edge from that node to $v_{post}$. At this point $v_{post}$ is now the current node.
- If a message is sent to a single target outside of a contract, we generate a subgraph $S$ that is the result of applying these rules to that receiver body. If $S$ is a single node, we combine it with the current one. Otherwise, we generate the subgraphs $S_1, \ldots, S_n$ as in the ‘in contract’ multi-target message case. If there are no further contracts or message invocations in the current body, we connect the current node to the nodes of indegree 0 in the generated subgraphs. Otherwise, we split and add edges as in the above case.

By memoizing partial results, we can dramatically speed up this process.

Note that the resulting graph will not be connected unless there was only a single manager subscribed to the phase message in question. This is because there is not a mechanism to act as a ‘sink’ to join any of the chains originating at each manager’s receiver body. We have already mentioned that adding that sink is part
of moving this system beyond the prototype phase. That would complicate this analysis somewhat, involving more complex searching for cut points, such as the recursive process that we detail immediately below.

We then examine each global collection that was referenced during this phase. If it is present in the nodes of more than one connected component of the graph where at least one is a write, then we have a potential state conflict. If not, we need to keep analyzing. For each connected component in which that collection is referenced, we trace from the single indegree 0 vertex until we find a node of outdegree greater than 1, call it \( b \). If no such node is found, then no state conflict can be detected from that subgraph. If found, we look for a node \( j \) such that there exists a directed path \( b, \ldots, j \) and \( j \) is a cutpoint of the graph and there does not exist a \( b' \) that is also a cutpoint of the graph and there is a shorter path from \( b \) to \( j \). We consider the subgraph of all vertices on a path from \( b \) to \( j \), not including those two vertices. If no such \( j \) exists, we consider the subgraph that is comprised of all vertices \( v, v \neq b \) such that a path exists from \( b \) to \( v \). In either case, we have a non-connected graph to consider and we repeat the above process recursively until either a state conflict is discovered or we exhaust all possibilities.

By repeating this method\(^6\) for each phase, we finally derive a list of phases with potential state conflicts for each collection in the system.

### 4.2.4 Acting On Results

After completing this analysis, we have considerable information to make decisions about whether or not we can safely exploit the opportunities we identified above (§4.1). We identified two areas we needed information for: global collection access and delegate block parallel dispatch.

We have, from the previous section, a list of phases and potential concurrent accesses for each collection. Surrounding each collection access for collections where this list is non-empty, we check which phase we’re in (the scheduler maintains such information) and if we’re not in one of the indicated phases the access is

\(^6\)The actual implementation of this does not directly embody this exact procedure, as several steps, such as finding paths, are implicit, reuse information from other procedures, or are spread over different passes in the compiler. We have presented it in its theoretic form to avoid going into uninformative details about the construction and transformations of our AST.
allowed to proceed unprotected. In the other cases, we’re going to need to ensure mutual exclusion between any two accesses. However, as we have pointed out, probably to the point of exhaustion, two accesses of a collection may very well be disjoint if they’re not accessing the whole collection. For this we’re going to need something finer grained than a single mutex around the collection. For this we use a technique we pioneered called Synchronization via Scheduling, which is the topic of the entire next chapter.

For unordered delegate blocks, we need to consult the access domain of the associated block. If it contains a write to a member of the entity or a local defined outside the contract, then we emit only serial code (essentially a standard for loop) for this delegate.

If programmers finds this serialization undesirable, they have three options:

1. Re-factor the code, though often this is not possible, or practical.

2. Query the member variable, in which case the normal rules of collection contracts apply\(^7\).

3. Replace the member with an *accumulator type* which is designed to be accessed in parallel and useful for counters and other commutative operations (§3.1.5).

This may prompt a question: *how is the programmer going to know if something has been declared non-parallelizable?* Given that this decision is based on a great deal of analysis, this is a good question. Our answer to this is the code analysis report, detailed below (§6.1).

### 4.3 Deadlock Avoidance

It’s one of those hard facts of life that so often the cause of problems is the solutions to other problems. In order to prevent state conflicts and the potential errors caused by values being changed mid-use, we introduced collection contracts and their strong isolation semantics. In order to make them useful for writing complex systems, we made them flexible and composable. In doing this, we opened the door

\(^7\)Member collections must always be queried.
to another beast from the concurrency abyss: deadlock. To return to our metaphor from the introduction, if we aren’t careful how we restrain the hands of our evil 3d chess opponent, he’ll be unable to move at all. The game will stop, forever, with no winner and no loser.

Deadlock may occur between two or more sets of overlapping contracts executed in parallel. Consider the following Cadmium code:

```cadmium
1 someMessage → { 
2  ... 
3  | SELECT { element: 23 } FROM foo | → view1 
4  | SELECT { element: 17 } FROM bar | → view2 
5  ... 
6  release view2 
7  ... 
8  release view1 
9  ... 
10 } 
11 
12 someOtherMessage → { 
13  ... 
14  | SELECT { element: 17 } FROM bar | → view3 
15  | SELECT { element: 23 } FROM foo | → view4 
16  ... 
17  release view3 
18  ... 
19  release view4 
20  ... 
21 } 
```

**Listing 4.1: Deadlock Potential Example**

This is the setup for the classic deadlock condition if these two receivers are permitted to execute in parallel. If the first receiver is granted the view on element
17 from foo and before it is granted the view to element 23 of bar, the second receiver is granted the view on that element. This creates a cyclic dependency and given strong guarantees of our collection contracts, the program will wait endlessly. This example is contrived, of course, but the indexes requested may be determined at runtime, as opposed to the explicit number in the example. This would still be problematic, even if all requests were to collection foo.

During static analysis, we can test for deadlock potential. The augmented GoodLock algorithm [11] has a number of similarities to our algorithm, though their model has complications that are eliminated in DnC.

Our algorithm uses much of the same information that we gathered while determining the access domain and phase graphs, as described above. We build a model of the program in terms of contracts, treating receiver bodies and method calls as implicit contracts on members accessed. Recall that collection in each query can be resolved unambiguously, even if it involves a subcollection (§3.2.3).

From this, we derive a set of ordered pairs, \((a, b)\), of collections, where an Isolate step on \(b\) is reachable during the modify/derive of \(a\). Using reachability analysis, we create subsets consisting of the pairs that may be run concurrently. We effectively transform the set of pairs into a directed graph and check for cycles which would indicate a potential deadlock. Figure 4.1 shows this process, where the blocks with \(\lvert X \rvert\) show the maximum lifespan of the contract on collection \(X\).

If a cycle is detected, we do one of following:

1. Ensure that satisfying one of the contracts in the cycle is mutually exclusive with the other contracts, by adding additional scheduling logic in its isolate phase.

2. Augment a contract’s isolate phase to reserve(§5.5) the subject of the contained contract, and thus prevent the cyclic dependency. When execution completes the inner contract’s isolate phase (i.e. the inner query is successful), the reservation is released and execution continues.

3. When a receiver body forms a cycle with instances of itself, mark it so the scheduler will serialize all message delivery to that receiver.

8Some cycles are benign due to details about their components. We ignore these cycles.
4.4 Further Static Analysis

There are several circumstances when multiple instances of a block of code may be executed concurrently. In addition to delegate blocks (§3.2.5), receivers may be marked as sinks, acting as consumers in a producer/consumer relationship. The STDOUT string receiver we discussed earlier (§3.1.2), is an example of one such receiver.

In both cases, if an entity member is accessed, or in the case of view delegates a local variable outside the delegate is modified, incorrect results may occur. We analyze each body of code looking for such assignments and accesses. If found, we ensure that the code body executes independent of any other instances, and issues a compilation warning. Similarly, if two or more receiver bodies access the same member variable, and one of those accesses is a write, we ensure mutual exclusion between these bodies and, again, issue a warning.  

In the same vein, we examine the combination of each query on a particular collection and only compile a program with the particular behavior needed to guarantee safety. This includes the reservation mechanism (§5.5), or as mentioned above, we only apply our locking constructs in those phases where a state conflict is possible.

4.5 Programmer Directed Optimizations

In order to assure safety in all cases, we must make conservative assumptions. However, the programmer may have additional insight. Nearly every statement can contain an annotation, which uses the syntax \{ ? \langle key: value, \ldots \rangle \}, where the set of possible keys is defined by the particular statement. This includes marking a receiver as a sink and adjusting the runtime parameters (§5.3). Annotations can be used to silence the static analysis warnings discussed earlier. An annotation is never required for correct execution, where an equivalent of debug mode would turn all annotations off.

---

9Any method call is considered in this analysis.
Chapter 5

Enforcing/Executing Intentions: Scheduling And Runtime Algorithms

In the previous chapters, we showed how we added language features to allow for better translation of programmer intention to code and then interpreted the information we were able to harvest. What remains is runtime behavior: how do we organize and schedule tasks and, more importantly, how do we efficiently mediate between two or more different queries accessing the same collection?

5.1 Scheduling Implementation

Initially, we spawn a number of threads equal to the number of available cores\(^1\). The worker threads contain very little logic and are designed to be modular, executing directives, arbitrary pieces of code produced by the compiler. Each directive describes an operation from processing an array slice to delivering a set of broadcast messages. Whenever possible, such as the case of system messages (§3.1.4), these directives are precomputed.

It is common in our system for a number of related tasks to be created for a single parallel operation. If we had a standard atomic queue (commonly called

\(^1\)Addressing sharing the machine with the other programs is part of our future work.
a work queue) for storage and distribution of directives, and if an operation had 147 tasks associated with it, we would have to perform 147 enqueue and dequeue operations in the execution of that operation.

To reduce the potential of heavy contention during busy periods, our global task distribution structure, called the directive store, does model a queue with relaxed semantics, but has an extra facet to accommodate these related tasks.

Every one of these multipart directives has an atomic counter associated with it as part of the public interface of the directive store. Instead of dequeuing the directive immediately, the idle worker in search of a task decrements the counter. The system is designed such that the value of this counter at the time of decrement, which atomic subtraction yields automatically\(^2\), will give the worker all the information it needs to find which ‘part’ of the directive it is now assigned to execute. As a very simplified example, consider a parallel operation with 100 tasks operating on each element of an array with size 100. If a worker decremented the associated counter from 47, then it will perform the directive on the element at index 46\(^3\). If the counter is decremented to zero the directive is dequeued before the execution begins. Note that, from the point of view of the directive store the difference between multipart directives and ‘single’ directives is a only descriptive. All directives use the counter, but a multipart directive is one where the initial value is greater than 1.

The directive store itself is designed to balance between needing to accommodate an unbounded number of directives and an aversion to memory allocation (which is quite slow in the terms of the time scale the scheduler operates on). The directives are stored in uniformly sized arrays, called cells, which are in turn organized as a linked list.

As with many queues the head and tail are numerical indices, which strictly increase with enqueue and dequeue operations. When the tail reaches the end of a cell, that cell is removed from the linked list and ‘recycled’, either to internal storage in the directive store, or deallocated if that internal storage already contains a predetermined number of cells. When the head reaches the end of a cell, a new cell is produced either from the internal storage or, in extremis, through memory

\(^2\)At least on x86.
\(^3\)Yes, we did have multiple off-by-one errors while building this.
allocation and linked into the end of the cell list. The numbering of cell elements is kept between cells, that is to say that if the tail was 80 at the end of one cell, it will be 81 at the start of the next cell\(^4\).

In essence, we are trying to emulate having one infinitely long array, by moving no longer needed regions ‘in front’ of the head. It would be like having a train that moved the chunk of track behind it to the area closer to the destination so virtually infinite trips could be done with a very small amount of actual track.

In an ideal situation, there are never any more directives at once than can fit into a single cell. This means that only two cells would ever be needed, avoiding memory allocation and as the cells are designed to be large, cell swapping (a relatively low cost operation anyway) is kept to a minimum. In the less than ideal situation where there is sudden burst of directive creation\(^5\), the structure will grow to accommodate this to the limits of physical memory. There may be a bit of a performance ‘hiccup’ during the allocation, but the program will continue operation.

All application profiled in this work use a cell size of 8,192 with one additional cell pre-allocated at scheduler initialization. While we have not done extensive testing to determine if these are the most effective parameters, they easily handled the demands of our applications.

A directive contains the work we expect one worker to perform, all else being equal. For a data parallel operation, we create a number of dispatches equal to the number of cores. If not all threads can participate, the first one finished will pick up the slack by claiming a remaining directive. This reduces the potential for load balancing, but dramatically relieves the pressure on the central queue. We will show that this gets good results in practice (Ch 7).

When a worker thread begins to execute a directive, it does so inside a coroutine\(^6\). If the operation is unable to proceed, the thread yields the coroutine, enqueues it, and moves to the next work. This ‘next work’ is not necessarily another directive. It may be part of the same directive, such as delivering the next broadcast

\(^4\)Given that these indices are 64-bit unsigned integers, overflow is not a ‘real world’ concern

\(^5\)This also applies to the case where for some reason a really old directive can’t be completed and stays on the queue for a great length of time, causing the tail pointer to stay stuck at its location. This case does not occur in our current implementation, but it was considered during design.

\(^6\)We found that boost::coroutine gives the best performance out of the alternatives we tried.
message. These coroutines are pooled to avoid allocation costs and are written to be thread agnostic, so they can be migrated for work stealing and other purposes. We found that, in practice, yielding a coroutine costs about 500-2000 cycles. While this is nonzero, it is small enough to do very fine-grained operations.

While we initially tried to avoid the extra overhead and complexity of a work stealing scheduler, it turned out that a simple algorithm along the lines of the standard ‘steal half’ [38] scheduler made a significant difference. We will discuss the evolution of our work stealing more in the test that inspired its inclusion (§7.2.3).

![Diagram](image)

**Figure 5.1:** Mapping queries to signatures

### 5.2 Synchronization via Scheduling

The primary methodology we employ is a reformulation and rethinking of the concept of Synchronization via Scheduling (SvS) [16]. This was a technique that we developed and published previously. This entire work is an extension and refinement of these previously published ideas and a response to the feedback we received on that work.

In its original form, SvS had the high-level description: *Know the maximum amount of data every task will touch and only co-schedule disjoint tasks*. Using a combination of static analysis and runtime bookkeeping, we derived a system that would follow pointers across the program data. It would only execute tasks in parallel whose pointers set was unreachable from others. As well, it employed a primarily streaming model. However, this could drastically overestimate the maximum impact of a task. We used our equivalent of receiver bodies as critical section demarcation and used primarily data streaming techniques. These patterns are all expressible in Cadmium, along with the more expressive and flexible constructs we detail here.

The genesis of this work was an attempt to correct the shortcomings of the SvS
model and to push it further. The change from considering pointers to the consideration of types and collections made for much better estimates, as programmer intention was considered, and much runtime bookkeeping is eliminated. Introducing Collection Contracts also allowed for much finer grained decomposition. These two factors make DnC much more than an incremental improvement on SvS.

The challenge in scheduling this way involves solving two related problems: determining the set of items in a subcollection, and performing set intersection tests fast enough to be useful on the micro or nanosecond scale. We next introduce a representation that will be critical in solving both problems.

5.3 Signatures

A signature mapping is a partitioning of a potentially infinite, space into $n$ ordered partitions for some fixed $n$, which we call the width. A signature of some subspace of this space is an $n$ length bitstring, where bit $i$ is 1, if partition $i$ of the space has a non-empty intersection with the subspace and 0 otherwise.

For our purposes, the space generally under consideration is the maximal collection space, the collection that is a superset of all other collections of that type. For linked lists, this would be an infinite list, and for general trees this would be the infinite tree, where every node has infinite children. This will be illustrated below (§5.4).

Signatures give us a lossy representation of the subspace, as it doesn’t reflect how many items are in the intersection — only that it is non-empty. The larger $n$ gets, the more accurately it reflects the distribution of items across the subcollection. The next section will illustrate concrete examples and show how we use signatures in a scheduling heuristic.

Scheduling with Signatures

Suppose we are executing a data-parallel query on an array, which processes the array in chunks of 3 consecutive elements using the query `\texttt{SELECT range}(i, i+2)` \texttt{FROM arr}`. For simplicity, we use an array size of 8, signature width of 4\textsuperscript{7}, and

\textsuperscript{7}In practice, we use much larger widths. Our experiments show that for most cases a width of 512-2048 is effective; and by default we use a width of 1024, which may be adjusted during static
a signature mapping that assigns indices \{0, 1\} to partition 0, \{2, 3\} to partition 1, and so on. As arrays have fixed size, the space to partition is finite.

When the query is invoked, the caller constructs a *use signature* based on the query and the attributes of \(\text{arr}\), such as the width it uses. The *range* substructure atom refers to cells 2, 3, and 4 in the array; and so the use signature generated is 0110, as shown in Figure 5.1. At this point, the scheduling code is invoked. The scheduler retrieves \(\text{arr}'s\) *collection signature*, the record of what parts of it are currently under contract. We will assume that \(\text{arr}\) is currently unused, so its collection signature is 0000. The scheduler compares the two and finds that the second and third partitions of \(\text{arr}\) are available, so the query is satisfiable. The collection signature is updated to 0110. No changes to scheduling are necessary, so control is passed back to the invoking code, and the Modify/Derive begins.

Now, say that another worker thread begins with \(i = 1\). This requests elements 1-3, and thus the use signature 1100. Control passes to the scheduler where 1100 is compared to 0110. The scheduler sees that the second partition is already in use, so this query cannot be safely satisfied. The calling code is suspended by yielding and enqueueing its coroutine, and its owning thread can move on to other work. As this occurs, the first contract concludes, releasing the view, and \(\text{arr}'s\) collection signature is updated to 0000. The second contract is revisited, and the intersection test is now successful. The contract can be satisfied, the coroutine is resumed and its Modify/Derive commences.

**Heuristics, False Positives and Performance**

This is an ideal point to talk about the cost of the information lost when a signature is constructed. As mutual exclusion is guaranteed at partition granularity, the intersection test could report that two subcollections overlap, when in reality they don’t: a *false positive*. For example, if the second contract above had specified range\((5, 7)\), its signature would have been 0011, which would have been evaluated as a conflict, even though ranges 2-4 and 5-7 are disjoint.

The cost of a *false negative*, the failure to report a real overlap, is both the potential violation of the terms of the contract, and subsequent exposure to execution

---

63
error. Because of this, signatures are specifically designed to prevent them. The cost of a false positive is that two contracts are serialized unnecessarily. False positives are inevitable, but can be made quite rare with a wide enough signature. The cumulative negative effects of a small number of false positives are minimal, while the computational efficiency of the technique allows for fine grained scheduling without significant overhead.

**Virtues of Signatures**

Signatures have several beneficial properties that make them ideal for our purposes:

**Quick Intersection Testing** The intersection of two signatures can be computed simply by checking if their bitwise and is non-zero. This tends to be a very simple and cache friendly process that is optimized in the Cadmium runtime with SIMD operations.

**Composability** The union of signatures can be computed quickly with a bitwise or. This is especially useful in batched operations, as the atomic operations of manipulating the collection signature are the most expensive part of the process.

Applying a use signature to a collection signature is the equivalent to taking an arbitrary set of locks with one compound operation. Effectively, this allows us to vary lock granularity without changing the program logic. We will demonstrate a comparison with individual fine grained locks in Section 7.2.4.

**Other Signature Attributes**

In addition to width, we consider two other attributes of signatures:

**Padding** During heavy concurrent use of a collection, the atomic operations involved in signature update can generate much cache invalidation traffic. Separating the signature into smaller units, each on their own cache line, reduces a lot of the false sharing from updating bits near each other in memory.

---

8. We do not currently support changing the width of a collection signature during execution; however, the components necessary are present.

9. We have considered ‘striping’ high use collection signatures with low use collection signatures to save memory; but this is dependent on harvesting more information to tell us which collections are high use (Ch 8).
Sparsity Often times we can determine the maximum number of bits a use signature involves at compile time, as in the swap operation in Canneal (§7.2.3) where, at most, two bits are used. In this case, we can simply store and apply by the indices of these bits.

5.4 Deriving Signatures from Queries

In this section we go deeper into how we derive signatures mappings suitable for populating from different queries. There are two classes of methods corresponding to the two aspects of collections: content and structure.

Content Method

The content method is the default approach when others are unavailable. To employ this method, one first selects some numerical attribute of the content, and then simply partition by the residue classes of that value \( \mod n \). One could use an object id, or even memory location\(^{10}\).

Under this mapping, a use signature is derived by taking the union of the signatures of the items specified in the query, or taking advantage of sparse signatures where applicable. It is possible to optimize this by pre-computing signatures for various substructure types while the collection is being populated. This is analogous to adding an index to a database table. For example, if the program queries for the neighborhood of a graph vertex, as in the Page Rank (§7.2.1) or Delaunay Mesh example (§7.2.2), we can compute the signature of each vertices neighborhood as edges are added. Signatures are loosely related to Bloom filters [18], and these precomputed signatures have the same issue of non-decomposability. That is to say, if an edge is added, the current neighborhood signature can be augmented by adding, at most, one bit; but if an edge is removed, a bit cannot be ‘subtracted’ and the entire signature will need to be recomputed for maximum accuracy\(^{11}\).

This scheme is well-suited to unordered sets or hashes, as we simply store the

\(^{10}\)Care must be taken when using memory locations as they are often allocated at regular intervals, and so some partitions may be considerably more represented than others. Discarding a few lower order bits can correct for this.

\(^{11}\)If the signature is not recomputed, it will simply overestimate the neighborhood, which as we’ve noted has no effect on correctness and has only a minor one on performance.
items in a number of buckets equal to the width of the signature (or some factor thereof). This works well for selecting single items, which is desirable as these data structures are often used as key/value stores. Subcollections will have a variety of structures; but when two or more are being accessed concurrently, we need the fastest way to compare them. Given that we have the items from the subcollection query, we can compute the use signature of the type collection without reference to the type collection itself.

**Structural Methods**

One of the chief contributions of this work is the consideration of collection structure in parallelism. For each structure there are a number of substructural queries that are commonly used, such as subtrees, vertex neighborhoods, or array subranges. If we can derive a signature without having to iterate through each element in the substructure, we can drastically improve scheduling performance. While space constraints prevent us from giving a complete list of common data structures and substructures, we present a variety to demonstrate the concept and give more insight into our experimental applications (Ch 7).

**Arrays** The signature mapping is a generalization of that in §5.3, where each partition is a range of size $\frac{s}{n}$, and where $s$ is the size of the array. If $n \not\mid s$, the final partition also contains the remainder. If $n > s$, each element is given its own partition, and any remaining partitions are empty. Range based signatures are computed by determining the partition of the first and last element through simple division, and populating 1s between them, inclusive.

**Resizable Arrays** A resizable array, corresponding to the C++ vector, is treated in a similar manner to fix sized arrays, as long as the size stays constant. If the array grows past the established size, the new elements will be considered to be part of the final partition. Once the array is not currently under contract, it will update the size used for future signatures. Ranges are computed identically to fixed size arrays based on the established size.

**Lists** For linked lists, we maintain a number of sublists not greater than the signa-
Figure 5.2: Partition numbering for linked lists

Figure 5.3: Partition numbering \((n = 16)\) of quadtree (4 children)

ture width. Each sublist corresponds to a partition in the signature mapping, see Figure 5.2. We keep these sublists balanced as follows: when a partition sublist is accessed, we check without locking (as approximate answers are fine) to see if the current partition is significantly larger or smaller than its neighbours. If it is, and the current collection signature allows us access to the sublists, we distribute the nodes. If we are unable to distribute the nodes, correctness of the procedure is maintained and the rebalancing is only postponed.

Trees We first consider trees with a bounded number of children, such as binary trees or quadtrees(§7.2.4). Let the maximum number of children be \(c\), and, for the sake of simplicity, assume that \(c \mid n\). Cases where \(c \nmid n\) are handled similarly to the array partitioning above.
The root is assigned to the first partition. For the child $i$ of the root, with enumeration beginning at 0, we assign it to partition $i \times c$. For the children of child $i$, we assign them to the range of partitions between $i \times c$ and $(i + 1) \times c - 1$ at an interval of $\frac{n}{2^c}$. This pattern, illustrated in Figure 5.3, continues until all children of a node are in single partition, similar to the buckets previously discussed. The signature for a given position in the tree can be determined by a simple tree walk, which almost all queries on the tree perform anyway.

The partitioning ensures that all descendants of a node are in a predictable range; and so subtree signatures can be derived by finding the node, which the vast majority of queries need to do anyway, and filling that range with 1s.

Trees without a bounded number of children are handled similarly, but when considering the enumeration of children we use $i \mod t$ in place of $i$ for some fixed $t$. Without an annotation, we use $t = 16$. The consequence of this is that child 0 and child 16 will use the same range. Once again, this can lead to securing access to more than is requested, but will have no effect on safety and, except in pathological cases, little effect on performance.

### 5.5 Collection Reservation

We discussed the need to secure exclusive access to an entire collection (§4.3). One could apply a use signature of all 1s, but this would involve several more atomic operations than the average. It also would be prone to a considerable number of failures, if the collection was in any kind of demand. As this need is common, we add a mechanism for reserving a collection for exclusive use.

We add another set of bits, by default 32, to the signature as a reservation block. This functions primarily as a counter, similar to a reader/writer lock. Before the use signature is applied, the query invoker increments the block, and decrements it when complete. Reservation is requested by atomically setting the highest order bit to 1, and no new contracts will be allowed until it is removed. The reserver is suspended, and a reference is placed in the collection. When the reservation block, excepting high order bit, becomes zero, the waiting coroutine will be resumed with exclusive access to the collection.

This mechanism has uses beyond deadlock prevention. For content queries, the
entire collection has to be iterated over. We first reserve the entire collection, and then test the members, write the signature for the values described to the collection signature, and, finally, remove the reservation. Once again, we employ a form of two-phase locking. In other cases, where reservation is required, we can exploit some of the properties of structure based signature mapping. If we are iterating across the collection, say for a list, we can reserve the collection, write 1s into every position of the collection signature without atomic operations, and, as we complete processing of each partition, write a 0 to allow other contracts before our operation is complete. If we can execute the original contract in parallel, the same applies, except each worker thread is assigned a range and frees the finished parts, as demonstrated above.

5.6 Signature Hoisting

As we mentioned above (§5.3), one of the virtues of signatures is their composability and we exploit this wherever possible. The primary cost of our signature based system is the application of a use signature to a collection signature, which employs one or more atomic operations. If we can reduce the number of such applications, we can reduce both the total overhead and potential contention. For example, consider 10 tasks that access the same collection, each with their own use signature. At the time of the tasks’ dispatch, we could hypothetically combine the use signatures into one single aggregate use signature. We can then add this signature into the target collection’s signature and, when successful, execute all 10 tasks serially without further need for any synchronization. The aggregation is no more work, in terms of types and number of instructions, than adding each of them into the collection signature. However, since the aggregation needs no protection we have reduced the number of atomic operations by 90%, which is definitely significant, as not only have we avoided a number of potential cache misses that come hand-in-hand with atomic operations, but under heavy contention the signature system can tend to flood the bus with atomic traffic.

This aggregation is not an absolute win, the aggregate signature is often ‘bigger’ (in terms of the number of 1s) and so has a higher chance of collision with other potential use signatures from other tasks, where one of the component use
signatures may have been able to ‘fit’. Furthermore, every time we batch tasks, we
do reduce the scheduling overhead, but we also reduce the granularity of scheduling
and so may have trouble balancing the load between the different cores. However, in our experience, we find that the balance is overwhelming in favor of doing
at least some batching/aggregation.

Given the potential rewards we want to find every opportunity for this type of
optimization. However, the problem is often that the logic that produces the use
signature, generally a query, is ‘inside’ a task and not implicit in the task creation
itself. Our solution for this is a compile time technique we call signature hoist-
ing. When emitting the code for a particular operation we move, or ‘hoist’, the
use signature population code from inside the body of the task and into the point
where batches are constructed and tasks are dispatched, adding aggregation code
as appropriate.

There are limitations to when we can employ signature hoisting as we need
to be able to populate the signatures before any of the bodies in the combined
operator are executed. We can, however, hoist signatures for an operation that
contains a series of code bodies, such as those generated from a delegate block,
and the following criteria are met:

- the ‘outer’ operation can be executed in parallel
- the inner bodies contain a collection contract (generally a query)
- the parameters for the collection contract are fixed at the time of the initiation
  of the inner body (generally based on constants or parameters to the body)

These conditions occur twice in our experiments. Firstly, we specifically en-
gineered our canneal implementation (§7.2.3) to trigger this optimization and we
will discuss its effectiveness in detail in our discussion of that evaluation. Sec-
ondly, it occurred naturally in our video game example (§7.2.4) when the agents
are updated. This updating is implemented by selecting from the Set of all agents
and then sending them an update message. Recall that sending a message creates
an implicit contract for the duration of the receiver body and the ‘parameters’ for

---

13 This balancing act between overhead and granularity in terms of batches of tasks is a major
theme in our evaluations (§7.2).
this contract are the agents themselves. During compilation the signature of the individual agent entity, with respect to the Set, is hoisted as the outermost select is batched.
Chapter 6

Compiler Implementation Details

The Cadmium compiler is written in C++17, and is currently approximately 80 klocs. It uses the excellent ANTLR [71] parser generator and various Boost [76] libraries with few other dependencies. Cadmium began life as a C++ library before we realized the necessity for static analysis, and a form of interface that C++ wasn’t suited for. The compiler acts as a translator between cadmium and C++, the results of which are combined with the Cadmium standard library and runtime, and subsequently fed to clang [1] to produce the final executable.

Once again, we note that the compiler is currently in the prototype stage. There is still a great deal of work to make it ‘production ready’: filling out the standard library; adding more query atoms, in addition to all the other issues we will discuss as future work (Ch 8). It does, however, compile every example in Chapter 7 without needing to resort to ‘behind the scenes tinkering’.

6.1 Code Analysis Reporting

The author has been often frustrated by lack of feedback given by most compilers. While they will happily inform you of errors and are improving every year with some quite useful warnings, they are still frustratingly silent on when they make a decision on your behalf. *Did you actually inline that function like I asked?* It’s telling that Microsoft felt the need to add an extra __forceinline directive. The author recalls an afternoon trying to get gcc’s auto vectorizer to engage with a
With Cadmium we took steps to fill this gap. Every time a program is compiled, the output includes a series of notes detailing the choices made, especially in the case of forced serialization or forced reservation of entire collections. A snippet of which, cleaned up for clarity, is shown in Figure 6.1.

As we moved from developing Cadmium to implementing our test applications, we found this feedback, along with the automatically generated activity graphs that we will see in Chapter 7, to be invaluable even though we, hopefully, understood the system inside out.

We could foresee a time when this was integrated with the Language Server Protocol and informative tool tips or coloured ‘squiggles’ would appear in the IDE to allow the programmer to see how their intention is interpreted in real time.

### 6.2 Integrating with C++

Due to its compilation process, Cadmium can easily be incorporated with existing C/C++. By defining external modules, a collection of user defined state and C functions. These modules can include any C/C++ headers and the corresponding source or library can be included in the project. We use this to load the data in canneal(§7.2.3), as the file system component has not yet been finalized. Only one function from a particular module can be executed at any given time, unless the function is marked with the `threadsafe` annotation. Cadmium objects are automatically reserved when passed to an external function.
Cadmium primitive types are automatically translated into C++ types, and the compiler can generate the header necessary to interface with a Cadmium entity or collection. Effort has been made to make the generated code humanly readable. The contents of an external are exempt from the static analysis, and the programmer is free to ‘shoot themselves in the foot’, but effort has been made to avoid invisible side effects. Currently, there is no support for messages passing from C++, but many of the prerequisites are present.

As we will see in the next chapter, the presence of the C++ interface dramatically extends what we can do with just the prototype Cadmium. Our video game example (§7.2.4) displays graphics, responds to user input and interfaces with media stored in the file system. This enabled us to get away with Cadmium’s paltry standard library and allowed us to demonstrate that if Cadmium were fully realized it could be used alongside existing libraries (using some, but not overly burdensome, care).
Chapter 7

Evaluation

In the previous sections we have detailed our model, algorithms and tools in addition to our prototype implementation of these ideas. In this section we will evaluate the effectiveness of our model. This evaluation will attempt to incorporate the various axes of the project. Obviously, as the end goal of parallelism is to increase performance, we will examine numerical timing data. Additionally, we will explore and discuss the effectiveness of the programming experience implied by our model: what went right and any shortcomings discovered that may be rectified by future work.

A full user-study is beyond the scope of this project and, as we discovered exploring this idea, the tools would need to be markedly more mature in order to effectively measure the ability of programmers not familiar with the core project to grasp the principles of the model and exploit its virtues. However, to gain insight, the model was defined before the following experiments were implemented in order to gain some kind of insight into using this model and helping to define the path towards a software artifact that is appropriate for initial human testing.
7.1 Signatures

7.1.1 Introduction

In the course of our work, we have identified several factors that affect the performance of signature operations. The performance of signatures – especially during adding to a shared signature (called a usesig) – is critical to the performance of our concurrent algorithms, especially when considering microtasks (those processing kernels that take a small enough amount of time to compute that they can be easily overwhelmed by the overhead of scheduling them). So reducing a few cycles, a few cache misses or the amount of bus traffic has a cascading effect on our final result.

With this in mind we divided several potential optimizations for consideration. These fall into three classes:

**Padding** distributing the signature data over more cache lines than necessary so that concurrent signature operations are less likely to be modifying the same cache line

**Sparsity** storing only the values that are present (a bit that is 1, also referred to as set) in the signature in order to avoid taking the time to deal with 0 values that do not affect the outcome of the operation – especially useful for signatures that are known to have a small number of set bits

**Algorithmic** modifications in the method used to test or update the contents of the signature – generally a trade-off such as checking a value non-atomically before performing a more expensive atomic operation

7.1.2 Operation Overview

We focus on four operations that comprise the bulk of our use of signatures:

**Add** combining two or more signatures into a composite signature that has bits set for each bits set in the source signatures. This is a non-atomic operation – generally done before accessing a usesig in order to, for example, combine
the signatures of a batch of data for one bulk access to the shared collection. We preserve the property that adding order is monotonically increasing – that is to say that if bit $b_2$ has a higher position than some bit $b_1$ then $b_2$ must not be modified before $b_1$ – though modifying them simultaneously is permitted.

**Intersection** determining if two signatures have any bits set in common – essential to performing our set intersection heuristics. The intersection is of primary use in the protected operation below.

**Protected Try Add** the main focus of our optimizations – protected try and add is the most important interaction with any usesig. Acting as a concurrent combination of the two operations above: given two signatures, the operation attempts to add the second to the first if they don’t intersect and returns true if the addition was successful and false otherwise. This operation must be what we will call *bitwise-atomic*, where the operation need not be atomic with respect to the entire signature, but only to the set of bits to be modified. For example, one thread can be setting bit 23, while another thread is unsetting bit 172 simultaneously, but for that first thread bit 23 must be 0 immediately prior to the setting. Recall that, for signatures, a false positive is permitted though obviously to be avoided when possible, but as long as an optimization increases false positives also decreases total runtime – then the optimization is acceptable.

**Protected Remove** the complementary operation to Add/Protected Try Add, Remove takes two signatures and unsets the bits in the first that are set in the second. The protected version has the same bitwise-atomic guarantee as Protected Try Add. Note that this operation assumes that the every bit set in the second signature is present in the first – i.e. the second has been previously added, results otherwise are undefined. We we will use this assumption for several operations.

### 7.1.3 Padding

As noted above, padding a signature involves adding extra ‘dead space’ into the signature storage in order to increase the number of cache lines it occupies. We
refer to the portion of the cache line that is used to store bits as a signature’s cell
and the number of bits in a signature’s cell, rather obviously, as its cell size. The
generally employed pattern is that for two signatures of the same size (number of
total bits) that are used for an operation – the one that is shared (i.e. a usesig)
will be padded, while the one that is computed in an elided environment will be
stored non-padded. We have no cases where we have operations on two shared
signatures, so we don’t consider that case. As a consequence of this pattern, all
operations need to support combinations of padded and non-padded versions.

As a matter of nomenclature, we still discuss the cell size of a non-padded
signature as being the largest ‘chunk’ of data that is considered at one time by the
algorithm acting on it. This will be seen below when considering sparse signatures.

7.1.4 Sparsity
For the sparse case we only store the cells that have at least one bit set – with a
maximum cell size of the largest type that can be used by CPU supported atomic
operations (currently 64-bits). Our implementation uses a fixed array that could
hold the number of cells whose sizes adds up to the total maximum size of the
signature. For a sparse signature of size 512 with a cell size of 64 will have 8 slots
in the array. Similarly, an array of indices is stored. For example, if in the 512/64-
bit signature from above, bit 129 is set, then the first cell of the storage will have
the second bit set and the first index value will be 2, corresponding to the third 64-
bit portion of the full signature. Future work includes testing to see if interleaving
the storage and the indices would be more efficient.

Given how they are used, sparse signatures are designed only for the non-
shared side of a Protected Try Add operation and, consequently, do not support
being the target of any protected operation, only the source.

7.1.5 Algorithmic Modifications
We have derived the following optimizations for the internal workings of the signa-
ture operations – identified by short names for ease of discussion (and abbreviations
for the following graphs).

For the purposes of discussion, we will refer to a component as the data type
accessed in any step of the algorithm, which is generally the cell size, if the cell fits into a native machine type (no more than 64-bits), or 64-bits otherwise.

**Hasty (ha)** By default, while doing an intersection test, the various operations logically and the corresponding components from each signature and if the result is non-zero, stops and returns false. In the Hasty variant, the intersection operation forgoes the check after each pair, instead computing the cumulative and of the and of each corresponding pair and then checking for non-zero. The idea is that omission of the comparisons will save more than time than the cost of the potential extra ands.

**WeakSwapper (ws)** C++ exposes two different Compare and Swap operations on its atomic datatypes: strong and weak. The emitted assembly instructions differ by platform/compiler, but in essence the strong version only returns a negative when the comparison part of the operation is actually negative; while the weak version allows spurious failures, in that the operation may return a negative value even if the comparison would allow for a successful swap. By default, signatures use the strong version and this variant uses the weak version, which, incidentally, is much closer in function to the CAS operation supported on our target family of Intel CPUs.

**Brazen (br)** Based on the old phrase ‘Better to seek forgiveness than ask permission’, the Brazen variant uses atomic or to add each component to the target signature directly instead of using the conditional Compare and Swap. The atomic or operation always returns the directly previous state of the modified value and we take advantage of this fact by checking this result to see if the previous value would have intersected with the value we added. If not, we proceed to the next component. In the case where the addition would have failed, we then use atomic xor to remove the bits that we added (not including the ones already there) and return false. The rationale behind this is that atomic or is almost always cheaper to use than the atomic Compare and Swap, as it locks the bus for less time, and the second atomic operation is only used in the case of failure.

**Frugal (fr)** In the frugal variant the algorithm simply checks for a zero component
before reading or modifying memory, as a zero value never affects a test, add or remove operation.

**Cautious (ca)** For the cautious variant we perform a non-atomic intersection test before every Protected Try Add operation. The non-atomic test is significantly cheaper to execute and this would be a massive savings during times when there are lots of failures (a *failure* being any time when Protected Try Add returns false). Of course this is another heuristic. It is possible that the non-atomic nature of the test gets stale values and so the Add would have succeeded or the intersection test yields false when the Add will fail. In the former case we get a false positive and as detailed above this is acceptable if it increases the overall performance. In the latter case the Add is still performed atomically and so the add would fail anyway.

Note that with the exception of *brazen* and *weakswapper*, none of the above are mutually exclusive, and so as we test for performance it behoves us to try the valid combinations of these optimizations, in addition to testing them in isolation.

### 7.1.6 Testing Methodology

We detailed, above, the primary signature operations and have devised a test that uses them, mimicking their actual use in practice. For a given number of cores (as we we bind the worker threads to cores, we use cores and threads interchangeably), we generate a `usesig` (using a size of 1024 for these tests as this is our most common size - though we note that the performance characteristics of other sizes are not dramatically different). For each core we generate a list of signatures (1000 per cores in the results below) and each performs the following:

- do some amount of ‘busy work’ which exercises the cache
- attempt a Protected Try Add on the `usesig` with the next signature in the list
- if failure, we advance the list and repeat
- if success, we do more work, Protected Remove the signature from the `usesig`, advance the list and repeat
We repeat this process a number of times (1000 in the results below) record the number of failures, take the average and record the failure rate. By varying the amount of work, we can induce different levels of contention on the usesig.

In order to separate the time taken for the ‘work’ and the signature operation overhead, which is the quantity we are really interested in, we have a version of the above test that uses fake signatures that are guaranteed to succeed or fail in predetermined manner and can determine this in the smallest amount of time possible (one non-atomic boolean comparison). Once we have the failure rate from the real signature tests, we setup another series (1000 repetitions of 1000 signatures per core) of these fake signatures with their success/failure set to the same ratio. In this way, we can get a decent approximation of the time in the test that is not overhead.

Finally, we have built an unoptimized ‘reference’ signature type by wrapping a C++ std::bitset in a spin lock. This gives us a baseline for performance to compare against – allowing us to detect the case where every implementation is highly suboptimal and picking the best one would just be picking the best of a bad batch. Incidentally, it was also highly useful in debugging the optimized versions and giving us a chance to check their correctness – as bugs in this kind of data structure are sometimes rather hard to detect.

7.1.7 Results

Using the methodology above we test each algorithmic variant, using different padding levels (where 512-bits is no padding) and core counts. We use two types of signature generators: random and hotspot. For random we set a certain percentage of bits. We use 15% in this case, as this gives a lot of potential collisions. Hot spots add a given number of bits (25 in this case) clustered in area (using a normal distribution centered around a random mean in the signature). This gives fewer collisions, but still has a decent number of non-zero bits so there is still work to do. This mimics a large number of cases we see in applying signatures in practice.

We present the timing results adjusted by the ‘fake signature’ time, to isolate only the amount of time used for signature computation. For easier comparison these values are then normalized to the reference signature time, that is the time taken for the spin-lock/std::bitset version. Though for the record a single Protected
Try Add takes, on average, between 0.3 and 2 microseconds on our test machines.

Note that while we describe these as random, we manipulate the random number seed to generate the same set of signatures for every core count – regardless of algorithmic variation or padding.

From these comprehensive results, we can draw the following conclusions:

- The time taken is highly sensitive to the level of contention and the number of cores involved
- From the fact that the relative ‘curve’ for each variant is similar, we can infer that the effects of padding is independent of the algorithmic effects
- As contention rises the benefit of atomic operation optimization over the simple single lock grows
- No padding configuration is superior, but overall the best configuration is those in the middle of the range, with neither 0 padding nor the maximum padding. We omitted the data from testing with smaller cell sizes, but they are almost always worse than the 64-bit case.
• Of all the optimizations, *frugal* has the most dramatic positive effect

• When combined with the above two variants, the *cautious* variant can be combined for sometimes slightly better results. This is not particularly useful

• The second greatest gains are achieved with a different addition method (either *weakswapper* or *brazen*) and the combination of the two is either the clear winner or non-worse than others, depending on the conditions. However, it should be recalled that weak swapping allows spurious failures, so we need to look at the data and the relative failure rates across all tests and other optimizations.

<table>
<thead>
<tr>
<th></th>
<th>Hotspot High</th>
<th>Hotspot Low</th>
<th>Random High</th>
<th>Random Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>weakswapper</td>
<td>24%</td>
<td>8%</td>
<td>72%</td>
<td>42%</td>
</tr>
<tr>
<td>brazen</td>
<td>15%</td>
<td>6%</td>
<td>69%</td>
<td>42%</td>
</tr>
</tbody>
</table>

**Figure 7.1:** Signature error rates – weakswapper versus brazen

From Figure 7.1 we can see that the slight increase in comparison perfor-
mance allows for a non-trivially greater number of false positives and so the brazen may turn out to be the best strategy to use overall.

- We notice that the deviation in time, denoted by the error bars in the graphs, is very consistent for every test not involving 12 cores. As 12 is the total number of cores in the system; this may be due to NUMA effects or interference from the small number of operating system processing tasks that were necessary to keep active while the tests were running

### 7.1.8 Sparse Signature Results

In the above results, all of the signatures were 'full' to not completely overwhelm with details. We now turn our attention to the sparse signature organization.

For this test we use the same testing methodology as above except that instead of hotspot or random, we set 3 random bits in sparse signatures (the usesig is non-sparse as it needs protected operations). This mimics a large number of our practical cases, such as when we went to swap a small number of items in a collection. This happens often and makes specific optimizations for low set bit contents worthwhile.

For the comparison, we average overall optimizations and padding values from the previous tests. We compare against sparse signatures with a cell size of 64 (the maximum) and 8 (the minimum).

We can conclude two things: sparse signature optimization is effective, but using smaller cells is not a useful savings.

### 7.1.9 SIMD Comparsion

We conclude with a mystery. The signatures are optimized with SIMD operations where applicable, using SSE2, as this is the highest level of SIMD support guaranteed to be on our testing machines. In order to test the efficacy of this optimization we ran 1000 trials of the Add and Test Intersection operations with random signatures with SIMD operations enabled and disabled (all signatures are non-padded, as these operations will be used in elided circumstances).

The results are as follows:
We had expected the SIMD version to beat the non-SIMD version soundly. However, this was not the case. This is not only surprising, but does not match the preliminary results from other machines/compilers. Our future work includes investigating this.

### 7.1.10 Conclusion

With this attempt at exhaustive testing, we are confident that we have the keys to minimize the overhead of signature operations as a part of our complete concurrency algorithms. There are, of course, a number of avenues of inquiry remaining, but we suspect we are far along the diminishing returns curve for this topic without fundamentally rethinking the idea of signatures and their corresponding operations.

### 7.2 Comparison with other approaches

Our main focus was to attempt to tame the trials of parallelizing complex applications. To this end, we constructed the complex video game example featured below (§7.2.4). However, as that application resembled the ones that inspired the model
initially, there is the danger of ‘over-fitting’. That is to say, we could have uncon-
sciously, despite best intentions, designed a system that was ideal for producing
that exact application.

We have hand picked 3 examples to compare against, both in terms of raw per-
formance and in programming style. We have selected these examples, all widely
respected by the community, because they represent a variety in terms of program-
ning paradigm, collection use and critical section protection. While none of these
is a ‘complex application’ their smaller scope is ideal for isolating the performance
difference in adding cores to their execution. Furthermore, as Cadmium uses a
novel programming model, porting is non-trivial and the smaller sizes allowed us
to do this kind of comparison within the scope of a project of this size.

7.2.1 PageRank(Green-Marl)

Description

The famous PageRank algorithm [69] was a substantial part of the genesis of
Google’s path to being the world leader in web searching. Essentially, it is based
on the idea that a page’s importance can be heuristically evaluated by the impor-
tance of the pages that point to it. Essentially ‘importance’ is treated as a transitive
property, with each site that points to a page contributing a fraction of their own
importance to the pages they link to.

The algorithm obviously lends itself to being modeled as a directed graph (di-
graph). The algorithm is processed in a number of iterations allowing the ‘value’
of each page to propagate to its immediate neighbors and then to its neighbor’s
neighbors in the next iteration and so on. This proceeds for either a set number of
iterations or until a certain stopping criteria has been reached\(^1\).

While generally computed with adjacency matrices, it can also be computed
using a more dynamic digraph collection, with the value attached to each vertex
being it’s ‘value’ and each directed edge represents a link from source to target.
This avoids the cases where linear algebra based approach has no solution.

\(^1\)For the sake of easier comparison, we have eliminated the early stopping condition and simply
compute a certain number of iterations
Challenges

In terms of computational dependencies, the value of each vertex at iteration \( i \) depends on the values of its neighbors at iteration \( i - 1 \). So the value of a vertex must be retained until all of its neighbors have been updated. This leads, naturally, to having two copies of the graph: one for a completed iteration and one for the next in a well established pattern, sometimes called double-buffering or a Stencil pattern variation [59]. Only in the case where the existing graph strains memory resources would it be worth performing a more fine-grained strategy where, for example, each core/thread works on an expanding subgraph and only keeps the very minimum of multiple vertex values at once\(^2\).

Green-Marl

Green-Marl [42] is DSL\(^3\) designed by researchers at the Oracle corporation specifically for implementing graph algorithms. Aimed at reducing the code overhead and chance for error in such applications, the Green-Marl DSL is, like the Cadmium prototype, compiled to C/C++ code, which uses OpenMP [24] ‘under the hood’. We selected this as a comparison as it compares the ease of using a tool designed for the specific, versus the general purpose tool we are developing.

Implementation

This is an ideal place for us to demonstrate and test the \textit{update} view facility of Cadmium queries (§3.3.3), which implements the double buffer pattern intrinsically.

The performance numbers below are gratifying. The most important number is the length of time it took us to re-implement this in Cadmium: 28 minutes\(^4\).

---

\(^2\)While we did not perform an experiment to justify this statement, we think that the reader will agree that, in general, a simple partitioning schedule with no locks is superior to a complex ordering of vertices possibly with locks for the vertices on the boundaries between the vertices assigned to different cores.

\(^3\)Domain Specific Language

\(^4\)Though of course it then exposed a number of bugs in the compiler which took weeks to fix, which is part of our reason why we felt that a user study would give false data.
Procedure pagerank(G: Graph, e,d: Double, max: Int; pg_rank:
   Node_Prop<Double>)
{
    Double diff;
    Int cnt = 0;
    Double N = G.NumNodes();
    G.pg_rank = 1 / N;
    Do {
      diff = 0.0;
      Foreach (t: G.Nodes) {
        Double val = (1-d) / N + d*
          Sum(w: t.InNbrs) { w.pg_rank / w.OutDegree() };  
        diff += | val - t.pg_rank |;
        t.pg_rank <= val @ t;
       }
      cnt++;
    } While ((diff > e) && (cnt < max));
}

Listing 7.1: PageRank expressed in Green-Marl

program pagerank {
  ARGV: {
    inputFile@string
    iterations@int
    damping@float
  }
}

manager networkGraph@CDDigraph[ float ]( labels: int )

manager Ranker {
  Initialize → {
    data: readCDCN(pagerank.inputFile)
    |INSERT FROM data INTO networkGraph|
    // setup initial values
    initial: 1.0 / toFloat(#networkGraph)
    |SELECT ALL v FROM networkGraph| → {   }
Listing 7.2: PageRank expressed in Cadmium

While there is no established (or likely possible) metric for code simplicity, we invite the reader to compare the Green-Marl implementation (Figure 7.1) to its Cadmium port (Figure 7.2), where the Execute message receiver corresponds to the function body in the Green-Marl code.
In the Cadmium code the Update View, $G$, is created with the query $\text{UPDATE networkGraph}$. At point, the ‘shadow’ copy of the $G$ is instantiated and the reservation mechanism is employed to ensure that access to the graph is exclusive. Note the use of $v.current$ and $v.current$, as $G$ is an Update View of the graph, then automatically $v$ is also a Vertex View of the vertex that’s fed into the query delegate block. The $current$ and $next$ ‘members’ are implicit in the Update View corresponding to the the existing graph and its shadow copy, respectively. The $current$ version is read-only for the duration of the contract.

At the point of $\text{release } G$, the shadow copy and the previous copy are unified, which unavoidably requires a huge amount of copying.

**Performance**

To evaluate the relative performance of the two systems, we use two graphs. The first is a randomly generated graph on 10,000 vertices. The second is a partial graph of the Internet released by Google, as part of programming contest [4], to reflect the origins of the PageRank algorithm. This latter graph has 875,713 vertices and 5,105,039 edges.

The 10,000 vertex graph results are presented in Figure 7.2. We can see that
that the performance of the two systems is roughly equal. Cadmium, as we can see, performs better at lower core counts, able to exploit more parallelism and as core count increase, both converge on approximately the same performance. The less than smooth curve for Cadmium is due to the variability induced by the particular setting of the parameter that determines how work is divided up for execution. This gives us a nice segue to talking about batch factor.

The term batches refers to the partitioning of a set of work where the cardinality is known at the commencement of a parallel operation. Each batch is a ‘unit of work’ for the worker threads. Batch factor, congruently, is the number of partitions that are used for the operation. Essentially, changing the batch factor is the classic tradeoff of scheduling overhead versus scheduling flexibility. As with signature details, batch factor can be adjusted by an annotation to the query in Cadmium code (§4.5). A comparison between different batch factors is shown in Figure 7.3. This experiment is rather sensitive to changes in batch factor due to the fact that the partitioning is on the vertices of the graph, but as can be seen from inspecting the code, the amount of work for each vertex is dependent on the number of its neighbors; so the total amount of work for a batch can vary wildly. In this set

**Figure 7.3:** PageRank on 10,000 vertices Cadmium versus Green-Marl - Batch Factor Variation
Figure 7.4: PageRank (Google web graph) Activity Graph for batch factor 10

Figure 7.5: PageRank (Google web graph) Activity Graph for batch factor 2500
of test data, the graph is generated randomly and the edges are roughly equally distributed, which is not the case for the Google web graph example, which gives us another convenient segue.

Figure 7.6 shows the comparison with Green-Marl on the Google web graph, showing the variation between various batch factors. It is notable that the increased amount of work erases the performance difference between the two systems as the difference are less pronounced at that scale. However, when the batch factor is dramatically increased the performance takes a significant jump.

Observe the difference between the activity graphs in Figure 7.4 and Figure 7.5 showing the differences between batch factor of 10, which is 1 batch per core on test machine, and a batch factor of 2500. Recall that an activity graph shows the core utilization over time, with each vertical column representing one core and each rounded rectangle representing some task being executed at the time span indicated by the y-axis. Both show the tail end of the first iteration and the initiation of the second. Observe that, in the 10 batch factor instance, the cores are completely utilized until batches start to complete, with one longer batch pushing back the iteration completion time noticeably. Note that the single isolated chunk in the between the iterations is initiating message body finalizing and beginning the new
contract. In contrast, the end of the iteration with batch factor 2,500 is barely ragged, but there are continuous interruptions of work to switch batches. Though, we will note that the gaps may appear larger due to the way SVG renders. Closer analysis shows that while the time to schedule a new batch differs based on various conditions it is approximately 1 µs on average. While this gap is miniscule in terms of the total runtime, a high batch factor can make a difference in the aggregate.

Insights

This experiment demonstrates the expressive power of Cadmium, comparing well with a specific purpose DSL and the performance characteristics also compare favorably. The variation due to the batch factor underscore one of our primary focuses for future work: automatically determining the best parameters for the various algorithms (this will be a theme for the remainder of the section) such as the work we contributed to in the literature [73].

7.2.2 Delaunay Mesh Refinement (Galois)

Description

A triangulated mesh is, as the name suggests, a set of vertices and edges embedded in a 2d or 3d space where every closed face is triangle. These objects are used often in several application categories from computer graphics to physics simulations. These meshes can cause problems if they contain so-called ‘skinny triangles’, those with one interior angle significantly smaller than the others. A Delaunay Mesh is a mesh such that for each triangle, no point of the mesh falls within the ‘circumcircle’ of any triangle in the mesh. If one were to take any triangle in the mesh and examine the circle that passes through each of its three points, that circle would not strictly contain any point in the mesh.

One popular way of deriving a Delaunay Mesh from an arbitrary mesh is Ruppert’s Algorithm [74] The full details of which are beyond the scope of this document, but in essence it examines each triangle in the mesh looking for angles less than a given amount. When a ‘bad triangle’ is discovered, it is removed from the graph and replaced with a new point, which is then joined with all the ver-
tices that border this new hole (or cavity) created by the removal to maintain the triangulation property. However, the newly created triangles may also violate the Delaunay property and thus this ‘cavity’ may need to be expanded further before triangulation. This process of cavity growth continues until it replaces all ‘bad’ (non-Delaunay) triangles transitively connected to the first.

**Challenges**

During the execution of the algorithm, the determination, processing, removal and replacement of the triangles in a cavity is a ‘local operation’. That is to say that the rest of the graph is untouched and unreferenced outside the cavity. Given that a graph that is a subject for such processing will generally contain many ‘bad’ triangles there is no reason that their resulting cavities cannot be processed in parallel. In other words, for any two disjoint cavities, there is no dependency implied for their processing. However, the fact that the bounds of any cavity cannot be known before examination means that this algorithm is not ‘embarrassingly parallel’. One cannot simply examine all triangles simultaneously and fix the bad ones, as two triangles may end up being in the same cavity. So while there is no implicit dependency between cavities, there are dependencies between individual triangles that are not known until runtime.

**Galois**

Galois [50] is a project out of the University of Texas designed for ‘irregular parallelization’ (operations on structures that are non-embarrassingly parallel) and is currently maintained by Intelligent Systems. It is a C++ library with a large enough scope to nearly be a DSL in its own right. It uses a symbolic rollback system similar to those in Software Transactional Memory [78] and contains a number of collection types, graphs, lists, etc, all highly optimized for their parallel processing model.

We chose this system as a point of comparison as its goals and scope are one of the closest to Cadmium. As we will discuss below, what is presented to the programmer is very different, being augmentation to C++ instead of a higher-level language approach. Additionally, they do not tackle the problem of heterogenous
algorithmic access\textsuperscript{5}. However, they do address issues of determinism and distributed processing, two aspects that we decided from the beginning were out of scope for the initial version of Cadmium.

**Implementation**

Unlike with PageRank (Section 7.2.1), we did not fully re-implement the Delaunay Mesh Refinement algorithm for this evaluation. For an effective comparison, we took advantage of the Cadmium C++ interface and created a hybrid using Cadmium structures in place of the Galois ones.

In the implementation, a graph structure is used, but the vertices are the triangles of the input graph and edges describe edges shared between triangles. The processing of the graph is essentially divided into two sequential steps:

1. Iterate through the triangles in the graph, find each bad triangle and determine (but not modify) its resulting cavity

2. Iterate through the cavities, remove them and replace with the appropriate re-triangulation

We found that these operations mapped nicely onto different parts of the Cadmium programming model. For the first step we used an accumulator list (§3.1.5) of sub-graphs. Rendered in pseudo-code the operation would be:

```
CDGraph G[triangle]  // a graph that contains triangles
CDAccumulatorList[*G] cavities  // a list of subgraphs of G
...
| SELECT ALL v IN G ORDERBY unordered| → {
  if v is bad {
    create a new subgraph c of G
    for each new triangle t of the cavity determined:
      | INSERT v INTO c|
      | INSERT c INTO cavities|
  }
}
```

**Listing 7.3:** Mesh Refinement Fragment 1 in Cadmium

\textsuperscript{5} Again, the condition where two or more algorithms may be applied to the same structure, perhaps by programmers who are not even aware of one another.
In actual implementation, the contents of the delegate block that processes the
results are replaced with a single function call with a C++ body that contains the
modified Galois code.

The second step, in Cadmium pseudo-code, follows suit:

```plaintext
1 | SELECT ALL cavity IN cavities ORDERBY unordered | → {
2 | cavity | → c // secure the stored view
3 | remove the triangles in c with DELETE
4 | INSERT new vertices and edges INTO c
5 | release c
6 | }
```

Listing 7.4: Mesh Refinement Fragment 2 in Cadmium

Again, the contract between the securing of the stored view (\[cavity\] through
release c) in our implementation is replaced by a call into C++ interface with
modified Galois code. Note that the cavity contains not only the vertices to be
removed by the ‘border’ defined by their neighbours In this case we can insert the
new components into the graph using c as a proxy. In this way we avoid the need
to serialize the graph augmentation due to unbounded structural changes.

Figure 7.7: Mesh Refinement on 10,000 vertices Cadmium versus Galois
Performance

For this comparison we used the 10,000 vertex graph provided by the Galois installation\(^6\). The results, with an optimized signature size and batch factor, are shown in Figure 7.7.

The reader’s first question is probably some variation on *why is the cadmium application so much faster, even using only one core?*. In analysis we concluded that the answer to this was: memory allocation. Galois allocates memory constantly for its roll-back mechanism and is generally tuned for massive parallelism, including distributed computation. As in other places we pre-populate our buffers, pools, free-lists etc. On the other hand, the Cadmium graph implementation uses a thread-local free-list mechanism for allocating new nodes. Given that Cadmium is a prototype and we’re working on good solutions to a certain set of already difficult problems, we elected to leave the problem of parallel memory management to future work and ‘punt’ by making sure that the system had a healthy amount of memory resources devoted to it. If we eliminate the preallocated free-list in the Cadmium version, the overhead and contention of memory management overwhelms the rest of the computation and the system achieves virtually no performance increase with additional cores. In this case, we had to hand tune the pre-allocation size to meet the needs of the system. This becomes another parameter to tune automatically, along with aspects such as signature and batch size, as part of our future work.

As the efficiency of the data structure is mostly, though not completely, orthogonal to the question of parallel performance, we present the rest of our results normalized to the 1-core value of each application for a fairer comparison.

Again we present variations on the batch factor and the signature size achieved by changing the annotations of the collection and queries.

Figure 7.8 shows the effects of varying the signature size and padding. We can see that 512 bits with no padding clearly outpaces the other. The modifications to the graph are relatively long operations compared to those in some other applications, such as the upcoming canneal experiment (Section 7.2.3). Thus the chance of many simulations modifying signatures simultaneously is relatively low,

\(^6\)They also provide a 5 million vertex graph, but we couldn’t execute the Galois application on it without a segfault on four different installations.
so padding is generally a net loss, adding more accesses without contention to alleviate. However, as the cavities can grow to encompass 40 or more triangles (and thus vertices in our representational graph), a use-signature for an operation may have 40 or more bits active and too small of a signature for the collection will cause too much unnecessary synchronization. 512 bits with no padding is revealed to be the ‘sweet spot’ with the best trade-offs between signature computation time, possible atomic collision and partitioning granularity. The fact that is exactly the size of a single cache-line on our test machine is almost certainly no coincidence.

Figure 7.9 shows the variation of batch factor on the final computation. We can see, for the most part, the process is largely unaffected by changes in batch size. Though as the batch factor increases beyond a certain threshold, the aggregate overhead of the batch dispatch mechanism begins to have a detrimental effect on the runtime.

Insights

This was a very instructive example for us. We consider it another positive indication that we were able to express the patterns of another programming model with our existing constructs without having to violate the spirit of our programming
model. The mechanism of stored queries naturally encompassed the Galois subgraph/cavity structure. However, realizing that our system was given a handicap of the memory preallocation, gave us a lot of respect for the engineering work that went into the Galois system. Currently, Cadmium applications use a frankly decadent amount of memory and have a correspondingly long setup time (all applications are measured from the point they begin the main computation and setup, data loading and similar times have been omitted). The costs and contentions of memory management have long been a problem in parallelization that isn’t as talked about as juggling critical sections. We regret not having more of a solution to offer on this front, but we had to resign ourselves to only solving some of the problems.

One of the most interesting aspects was the experience of attempting to code this algorithm ‘from scratch’ before duplicating the Galois solution. Attempting
to put ourselves in the shoes of the uninitiated parallel programmer showed how ‘unnatural’ the final solution was versus the algorithm as originally published. The primary divergence was the partitioning of the algorithm into two distinct halves, as detailed above. It would be very reasonable for a programmer to ask why am I making the cavity just to store it and fix it later? Why shouldn’t I just process it now? The answer is, of course, that this decomposition is what allows for the parallel execution algorithms to be effective. Determining the entire set of modifications previous to their processing allows Galois to derive their schedule and ensures that Cadmium’s static analysis does not automatically serialize the entire operation to avoid potential deadlock.

We were gratified to see that this decomposition was cleanly achievable with existing constructs, but still wondered if there was more that could be offered to the non-expert in this case. We did stop to consider the path that a non-expert would take to achieve this kind of result, in our system or others. This underscored for us the need for detailed and readable compiler feedback (§6.1). Unless we give the programmer the power to affect every aspect of the system with its corresponding complexity, as discussed below, it is incumbent on the compiler to inform the programmer what decisions it has made on the programmers behalf and why. We could easily envision the programmer sitting down, coding a similar algorithm without the partition and seeing the feedback essentially amounting to

```
Query |SELECT...| executed serially, due to unbounded modifications in the statement |DELETE ... |
```

and working to move things around until a similar construction to the partitioned solution is achieved.

Finally, we consider the complexity of producing a solution and trade-offs of control versus complexity. C++ is widely considered to be programming in ‘expert mode’ and though we’re quite a fan of this level of control, we must agree. The Galois solution offers fine grained control over every aspect of the system, but at the cost of requiring that these aspects must be addressed, adding another layer of complexity to the already difficult and sometimes inscrutable task of creating a program in C++. As an example, while compiling this document we took a quick look back on the Galois source and came across the following lines from the class
that fills in cavities after their removal:

```cpp
typedef std::vector<GNode,
galois::PerIterAllocTy::rebind<GNode>::other> NodesTy;
typedef std::vector<EdgeTuple,
galois::PerIterAllocTy::rebind<
    EdgeTuple>::other> EdgesTy;
```

Listing 7.5: Snippet of Galois C++ Code

Again, there is no objective metric for complexity and expressiveness, so we let the reader judge for themselves. We bring this up not to disparage the Galois project, something that we have quite a bit of respect for, but to acknowledge the tradeoff between complexity and control. A Cadmium application offers nowhere near that kind of control — especially over the movement of memory, but this section should show that we’re on our way to achieve our goal of allowing non-experts to create programs that are always safe and often performant.

7.2.3 canneal(PARSEC)

Description

Canneal, as the name suggests, performs simulated annealing. Very briefly, simulated annealing is a classic heuristic optimization technique. Many other techniques in the same class suffer from often arriving at a sub-optimal answer by getting stuck in a ‘local optimum’. Consider the the family of ‘hill climbing’ algorithms that essentially sample a set of near points, choose the the direction of the ‘best’ (defined by some objective function) point, sample that point’s neighborhood, choose the best direction and repeat until a point is found that is superior to all of its neighbors. The problem is that this point may be only the best point in its immediate area and a much better point may exist somewhere distant in the search space. Metaphorically, it may find the top of the hill, but there may be a mountain in the distance. Simulated annealing attempts to solve this problem by taking variable sized jumps away from the sampled points, exploring apparently ‘worse’ areas of the space in order to get away from local optima. The term annealing is
borrowed from metallurgy, where a piece of metal is made stronger by heating and shaping and continuing to shape as the metal cools. Correspondingly, simulated annealing has a concept of temperature, which is gradually reduced as the process proceeds. At high temperature levels the algorithm will make big jumps and increasingly smaller ones as the temperature lessens.

In the case of canneal, simulated annealing is applied to the layout of circuits as wires are required to connect different areas of the chip. Different placements make for different costs and the combinatorial explosion of different possibilities makes a deterministic solution often infeasible.

While this may, at first glance, appear to be a graph problem, canneal boils it down to operations on a list, fixed to the size of the number of elements to be connected. The application randomly considers pairs of elements in the list, computes the change in total cost to swap their locations and, depending on the temperature, swaps them or leaves the structure unchanged.

**Challenges**

The primary challenge in canneal centers around the list collection. The application is constantly modifying the list by swapping its elements. The amount of computation required to determine to swap, or not, is minimal and so there is heavy contention on this primary data structure. Any extra overhead in atomically protecting the list elements adds up quickly in terms of final runtime.

We specifically choose canneal for our evaluations for one simple reason: it’s one of the worst cases for the Cadmium scheduler and SvS. The almost trivial size of the tasks combined with constant contention on the list go against the grain of the general purpose system of signatures designed to elegantly handle arbitrary subsets of larger collections. Furthermore, it is easy to express a solution to this problem without needing to ‘dispatch’ work, except in the coarsest sense. The implementation, described below, that we compare against, dispatches a number of threads that only need to know how many swaps they are to perform and a simple synchronization barrier is used to coordinate the necessary pause between temperature steps. So any overhead problems in the general purpose, work dispatching, batch making Cadmium scheduler is going to show up starkly.
We have touted SvS and its related techniques as being generally competitive with hand-coded locks. In this case, the authors of canneal use a very clever atomic pointer swap. In their implementation, the central list stores pointers to the elements in the graph, to avoid even the use of locks. It’s cases such as this that we use to demonstrate that competitiveness.

PARSEC

The PARSEC benchmark suite [17] is venerable artifact in the community, well known and well regarded as a collection of applications that display a variety of workloads addressed with a variety of parallelization techniques. The canneal application included in PARSEC was written by Daniel Schwartz-Narbonne and Christian Bienia.

Implementation

As with Mesh Refinement (§7.2.2), we leverage the Cadmium C++ interface in order to incorporate as much of the original code as possible as part of our comparison. Reduced to pseudo-code the main loop is very simple:

```
1 | 1 ... canneal.tempSteps| → {
2 | 1 ... canneal.numSwaps ORDERBY unordered| → {
3   index1 = \text{random}( elementListSize )
4   index2 = \text{random}( elementListSize )
5   \text{SELECT} \{ \text{element}( index1 ) \text{element}( index2 ) \} 
6     \text{FROM} elements| → i1, i2
7
8   \text{if} shouldSwap( i1 i2 ) { 
9     temp: i1
10     i1 = i2
11     i2 = temp
12   }
13     \text{release} i1, i2
14   }
15 }
```

Listing 7.6: canneal Cadmium Fragment

Note that the clause \text{element}() is the standard way of indicating that the
substructure requested is a single element at a given index from a list/array. The similarity between the names of element clause and elements collection is completely coincidental\(^7\).

**Performance**

We compared performance on three different network sizes (number of elements) provided with the PARSEC distribution: 100k, 200k and 400k. The essential comparison can be seen in Figure 7.10, Figure 7.11 and Figure 7.12, respectively. There are two immediate observations from these graphs. Firstly, especially at lower core counts (and thus lower contention) Cadmium is nearly identical with the hand optimized PARSEC implementation. Secondly, as the workload increases, the differences become even less.

As with previous evaluations, we compare Cadmium against itself with various permutations of batch size and signature width. Figure 7.13\(^8\) shows the differences

\(^7\)Technically we could have omitted the `element()` altogether and just queried for `{index1, index2}` as the index accessed single element is the default subcollection of a list/array. Similarly, the release at the end of the braced scope is unnecessary as they would be automatically released at that point. We have elected to make our code examples as explicit as possible.

\(^8\)Note that in terms of both batch factor and signature width, the 200k and 400k versions are
Figure 7.11: canneal on 200k elements Cadmium versus PARSEC

Figure 7.12: canneal on 400k elements Cadmium versus PARSEC
in performance characteristics with various configurations of signature width and padding. Interestingly, the application is relatively stable under width variation, which came as a surprise to us. What didn’t come as surprise was that the best results came from the heavily padded (64 bits per cache-line) signature with a relatively low width (256 bits total). This makes sense as each query generates a use-signature of 2 active bits maximum, so the chance of 10 cores colliding often, even with only 256 partitions is low; so a coarser granularity is effective. However, the contention is high, so ‘spreading out’ the signature reduces atomic fighting on a single cache-line.

We see more variation when we look at the effect of batch factor (Figure 7.14). It is of particular interest that the best performance (batch factor of 20) and the worst (batch factor of 10) are both relatively low. This is a case of critical mass. Given that the time it takes to compute one potential swap is generally consistent, very little load balancing needs to be performed. However, a batch factor of 10 (only one batch per core) gives no room for any load balancing. A batch factor of 20 gives a little leeway for some load balancing, but contributes next to no overhead in terms of managing batches. In this case, that little amount of flexibility makes nearly identical to the 100k, so we have omitted them.

Figure 7.13: canneal on 100k elements – Signature Width Variation
Figure 7.14: canneal on 100k elements – Batch Variation

significant difference.

This gives us an excellent opportunity to talk about the scheduler’s work stealing capabilities. We were really loath to add this capability, preferring instead to ‘work deal’ and pre-distribute the work as much as possible. We didn’t want to add yet another protected structure to the runtime engine. Given that we were already making considerable use of the hardware supported atomic operations for both directive management and signature updating, we were afraid of completely flooding the bus with atomic instructions. Consequently, work stealing was the last addition to the core runtime engine and, to our surprise, worked rather well without adding any truly detectable amount to our scheduling overhead. As we detailed above, (§5.1), when the worker finds no directive to work on, it steals in a round robin fashion from the other workers’ deferred queues. We found this to be considerably helpful in the case of canneal. Recall that a task (which in this case is executing a batch of potential swaps) is placed on the deferred queue, if a use-signature protected-add fails. The coroutine yields and its host context is placed on the queue. In the case of canneal with a huge number of queries and thus a huge number of use-signature applications a number of failures is inevitable. As the scheduler prioritizes new work over deferred (a heuristic that we have found
Figure 7.15: canneal on 100k elements – No Work Stealing

Figure 7.16: canneal on 100k elements – With Work Stealing
to be effective in practice) failures tend to build up in the deferred queues. This would be fine except that there weren’t quite enough failures to generally give an even distribution across the cores.

Without work stealing, we would end up with a situation such as the one pictured in Figure 7.15, which shows the end of a temperature step (iteration) of the canneal process\(^9\). Recall that a single batch, if it runs from initiation to completion will form a round rectangle. A shape with a truncated bottom (and dashed line) represents a task/batch that has been suspended and, correspondingly, a task/batch with a truncated top represents a task that has been taken from the deferred queue after its use signature has been successfully applied. We can see that at this stage of processing, each worker has no more new work to perform and is clearing out its deferred queue. The uneven size of these queues causes several of the workers to go idle before all the work of the iteration is complete. Figure 7.16 shows the same experiment with work stealing enabled and it’s clearly shown that the amount of idleness is reduced to nearly zero.

Over the course of this project we attempted many experiments to come up with a robust policy on the priority of new work, reviving tasks from the deferred queue and work stealing (including what order to attempt deferred revival and whether or not to make multiple attempts to revive a task before re-deferral). We found that while we could hand tune a policy that would give a performance boost for a particular application, we found it to be both extremely fragile as minor changes in the application would cause it be degenerate instead of beneficial. Secondly, we couldn’t work out a way to present these options to the programmer in an annotation, as with signature width and batch factor. This manner of scheduling parameter remains a prime point in our future work for both programmer options and automatic parameter tuning, as we’ve mentioned previously.

**Insights**

Overall, we were incredibly pleased with the results of this experiment. Comparing our general, ‘heavy duty’ technique to hand-coded lock-free code nicely validates our assertion of competitiveness.

\(^9\)Here we are showing a batch factor of 320 to make the effect clearer
This is one case where we made a change to the Cadmium programming model for one of our tests. As mentioned above, the PARSEC implementation uses an array of pointers. After some deliberation we added a new primitive to the language: the raw (or opaque) pointer type to facilitate better interactions with the C++ interface. This type is ‘inert’ in the language. There are no operators on it, not even equality. It only serves as a mechanism to receive a value from the C++ interface to later pass back to another part of the C++ interface. We still are not convinced to keep this type in the future, however it is not without precedent. Embedded languages, such as Lua [6], and others that interact heavily with C/C++ have a similar concept. We consider the C++ interface to be not exactly a ‘necessary evil’, but perhaps a ‘critical unpleasantness’. We set out with the belief that in order to make parallel programming more accessible, the models would have to change, which would require a great deal of rewriting of software\textsuperscript{10}. This rewriting would, due to real-world constraints, require an intermediate ‘hybrid’ step, so we consider the Cadmium interoperability not just a happy side effect of our compiling process, but a critical virtue of the system, one that would need to be maintained even once the bootstrapping process is complete and Cadmium is compiled directly.

We also added the multiple object release clause during this process, but that should have been there in the first place and isn’t, at all, a departure from our model.

For the most part we found the expression of the process to fit naturally into our model, with one exception. In order to enable signature hoisting (§5.6), we needed to re-architect the program in an awkward way. With the current static analysis, hoisting is only enabled when all parameters to the query are known at the invocation point of the containing block. In order to trigger this effect, we needed to generate all the random numbers for a temperature step at the start of the step, stored in a list and then dispatched as a parallel query on that list. This induced the compiler to hoist the signatures. This we mark as a failure, given that both inducing this effect was awkward and that our reporting system, which we sung the praises of in dealing with Mesh Refinement (Section 7.2.2), wouldn’t have given the appropriate feedback. We take the lessons learned to inform our

\textsuperscript{10}This was underscored by our experience in rewriting just these applications.
Finally, with the above, we found that signature hoisting didn’t give the expected performance benefits, as shown in Figure 7.17. This came as a surprise to us as in preliminary testing on different hardware\textsuperscript{11} required at least 10 items to a batch, with hoisting, to achieve the quality of results presented above. From this rather negative experience, we take away the obvious lesson that it’s important to not ‘over-fit’ experiments on a particular piece of hardware. More importantly, that robust system would need to be hardware aware to some extent, or at least capable of automatic tuning after the application is installed on a particular machine.

\textsuperscript{11}While we present the results on one specific system, over the course of this project we have experimented on several machines including several ‘pro-grade’ laptops and older ‘server class’ hardware. While we discovered bugs after these experiments that could have affected the results, they were, in general, consistent with the ones presented here.

\textbf{Figure 7.17:} canneal on 100k elements – Batch Variation
7.2.4 Mini-CE (Video Game)

Description

As the concluding, and hopefully climactic, part of our evaluation, we present our ‘complex application’, written from the ground up in Cadmium. As we discussed in the introduction (Section 1), a complex application has no strict definition, but as a rule-of-thumb it can be described as an application that has a number of separate different parts, call them units, subsystems, modules, etc, that work in symbiosis. For example, consider the modern web-browser. The browser is responsible for retrieving content from the Internet, parsing it, laying it out for current display, executing live scripted content, responding to user input, managing cached information, and so on. Conceptually, at the ‘intersection’ of these subsystems, is one or more collections, which are read and/or modified by these systems. In the case of a browser, the central collection is the DOM\textsuperscript{12} tree. This collection describes the elements (or Objects) in the page (or Document) being presented. The renderer reads from it while the scripting system modifies it, etc. Each of the subsystems in a complex application, as can be clearly seen in this example, generally employs a variety of algorithms and access patterns and they have no strict dependencies on each other. This is why we maintain that handles heterogenous algorithmic access to collections is a critical part of facilitates the parallelization of complex applications. Even if the algorithms themselves are all embarrassingly parallel, the potential concurrent access by multiple algorithms can create a whole new kind of complexity. It is support for these applications that we see lacking in the current tools and techniques beyond throw a bunch of locks around the structures and hope that each subsystem has enough work to keep the cores busy while the others are waiting.

We designed Cadmium from the ground up for this class of problems. However this does pose a bit of a challenge in testing it, especially in the prototype stage. A complex application is, unsurprisingly, complex and porting say, an existing web browser, to Cadmium would be an undertaking as large as the project itself. As we have stated, none of the above applications qualifies as complex in our loose

\textsuperscript{12}Document Object Model
Each one of them begins with an input, applies one algorithm and gives an output. So we were compelled to write a skeletal, minimal version of an existing application, a ‘version in miniature’ as opposed to a port.

We chose the video game domain as a target for several reasons.

1. Video games are some of the most complex, performance hungry applications being written today, rivalling small operating systems in the number of facets. Additionally, they are also important ones as the industry has grown to billions of dollars per year and continues to grow.

2. The author has a great deal of experience in the domain, having worked on several titles, including the one chosen for ‘miniaturization’. *I want to make my own games* was the author’s primary motivation to pursue Computer Science and eventually to this document.

3. Games are more fun than web browsers\(^\text{13}\).

As eluded to above, this example is based heavily on an existing game, *Clockwork Empires* by Gaslamp Games (Figure 7.18). The original code is not public and even if it were, as previously mentioned, porting would be a considerable undertaking as the original took several person-years to create. However, the author\(^\text{13}\)

\(^{13}\text{Proof left to the reader.}\)
leveraged his experience to extract some of the most important patterns and implement them from the ground up.

As mentioned in our descriptive explanation of ‘complex system’ the problem areas generally occur at the ‘intersection’ of different subsystems – generally at the point of shared collection access. In this case we chose to focus on the intersection of three systems, AI, Simulation and Rendering, which will be detailed shortly.

The game itself fits into the subgenre of ‘colony builder’, which is in turn related to the genre of ‘city builder’, which the reader may be familiar with from such best-sellers as the venerable SimCity franchise.

In this case, the player takes on the task of running a settlement in a ‘new land’, with only indirect control over the citizens who will inhabit it. They issue commands to build certain types of structures and create tasks (called jobs) for the citizens to perform, though in the tested version there is no human input to facilitate reproducibility.

These citizens do have a certain amount of autonomy, with a certain amount of personality simulated for each one of them, based on a few different attributes assigned to them by the game. To complicate matters, the local area, while rich in the necessary resources, is also plagued by hostile creatures, which for these purposes we will refer to under the umbrella term ‘monsters’.

Our simplified version has the following properties:

- All agents are either citizens or monsters
- The map is generated randomly, with unpassable hills and bodies of water that the agents must circumnavigate in order to reach their various destinations
- The map is covered with resources of various types, distributed randomly
- The citizens have a home base in the center of the map, where they begin and no monster is spawned\textsuperscript{14}
- Each citizen needs a job in order to perform any action
- Jobs come in two varieties: gathering and patrolling

\textsuperscript{14} A term of art describing the process of bringing a new object into the game world.
• Gathering jobs involve a citizen going to a specified area of the map, searching for a specified resource and bringing it back to the home base

• Patrolling jobs involve the citizen going to a specified area and continually moving, looking for monsters

• The job a citizen selects is based on their attributes. A citizen with high ‘courage’ and high ‘health’ will be biased towards patrolling jobs. A citizen with a low ‘ambition’ will be biased away from jobs that are a long distance away from the home base

• Monsters have a job analogue in the form of a simple state machine. If they have had no contact with a citizen they simply wander and if they spot a citizen they will chase them and attempt to injure them if they can physically reach them

• A monster in pursuit has random chance at every ‘step’ of abandoning the chase and returning to wandering

• A citizen, on encountering a monster, will flee to home base if they are gathering or attempt to attack the monster if they are patrolling

• If a citizen or monster is attacked enough that their health attribute drops to zero, they are dead and are removed from the game world

• When an object is removed from the world (resource gathered, agent killed, etc) a new one is randomly spawned after some time has elapsed so the simulation maintains a fairly steady state

• Similarly, when a task is completed a new one will be generated, though not necessarily for the same location or type

The actual game is far more complex, with buildings adding to the map, agents needing downtime or to follow their passions, etc. However, in this way we cover the ‘core loop’ of the game and are able to study the patterns of interactions between data structures and algorithms needed to implement these rules.
While there are many subsystems involved in a game, there is another coarser division often used to discuss these topics, which will be useful here: presentation versus simulation\textsuperscript{15}.

Consider a video game version of tic-tac-toe. The simulation would track which board segments have an ‘X’ or an ‘O’, whose turn it is, be responsible for computing the next move of the AI player and determine when a game end condition is achieved. The presentation (or simply rendering) would keep track of the actual pixel coordinates of the lines that make up the board, the font used for players’ marks and would be responsible for actually producing the pixels.

The timeline of execution is broken up into ‘frames’, a term borrowed from traditional cell animation, marking the point where the presentation updates the contents of the display. A higher ‘frame rate’ (generally measured in Frame Per Second (FPS)) is considered to be more pleasing aesthetically and can make a difference in interactions for games which require reflexes.

The simulation is generally also divided into ‘frames’, to echo the nomenclature from the presentation, where the state of the objects it’s responsible for is ‘updated’ for current conditions (such as in the above, i.e. checking to see if either player has won).

It is not unusual for the presentation and simulation to run at different cadences, such as 60 FPS for the rendering and 10 FPS for the simulation. For simplicity we will consider them to be equal and one ‘frame’ is the time it takes for both presentation and simulation to perform all of their state updating and hardware interaction responsibilities.

\textbf{Challenges}

The primary challenge in expressing this system in parallel is the complete lack of embarrassingly parallel algorithms, especially when considered in context of the other subsystems. An agent, when being updated for the current frame may change its own data, cause another agent’s data to change, affect the list of jobs or change the position of the model that the renderer uses to represent it on screen.

Furthermore, much of the rendering is serial (even if in most modern games it

\textsuperscript{15}There are many different names for each of these in the literature and general discourse. Their definitions are not any more robust than ‘complex system’
is parallelized on the GPU, the preparation of data and transferring often has large serial segments).

We built two versions of the renderer: one using 3d graphics (Figure 7.19) and the OpenGL framework and one that used an older ‘frame buffer’ (Figure 7.20) technique, as the latter can be executed on ‘server class’ hardware, which often does not have the required GPU. The numbers presented are using the more general purpose rendering, as we didn’t find a notable difference between the performance profiles of either of them.

**Implementation**

To organize the various aspects of the system, we used several different collection types. The simulation portion of the Agents were implemented as Cadmium entities: objects with member state and the ability to send and receive messages. The agents were stored in a manager (§3.1.2) Set.

Tracking the physical locations of the agents was an ideal place to use the sub-collection with different structure pattern (§3.2.3), which the reader may recall is where we define a collection to be composed of members from another collection, though this ‘child’ collection has a different structure to the ‘parent’ collection.
In this case, the parent collection is \texttt{agents}, the Set containing Agents, and it is defined:

```c
class AgentsGrid{
  1024 1024
}
```

Listing 7.7: Grid Storage Definition for Agents

...to be a Grid(2d array) of size 1024 by 1024 whose member cells are drawn from our original \texttt{agents} collection. This allows our Agents to easily query to determine the other Agents in a given area. For example:

```c
SELECT { block: <blockUL blockLR> }
FROM simAgentGrid
AS simAgentGridContents
```

Listing 7.8: simAgentGrid Query

This selects a rectangular area of the grid as well-defined substructure (a ‘block’) with the coordinates given demarcating its extents (upper left and lower right),

\footnote{Creating an alias for the type to then use as the type for a manager declaration is not required, but it was a convention to increase readability that we developed as we gained experience writing Cadmium code.}

121
which could be, for example +/-5 of the agents position. This gives the invoking thread of execution a ‘snap shot’ view of that portion of the structure, protected from outside modification, that it can inspect and modify at its leisure for the lifespan of the contract.

It is important to note, from a semantic point of view, the contents of these cells are potentially empty, single item stored views of the agents collection. Having a view on the simAgentGrid does not automatically give the programmer a view on its contents, only the ability to see the cardinality of the stored views (i.e. are they empty or, effectively, ‘null’) and to allow the securing of the query. Though we ended up needing both this behavior and the securing of all the items in the sub collection view immediately (i.e. that a view of the subcollection is a view of the parent collection) and so are deeply considering how these two different approaches should be incorporated as the language matures.

An astute reader will note that this implies that only one agent can occupy a single cell at one time. This is not only true, but a behavior that was desired.

Similarly, we defined another grid to hold resources. Though as resources are interchangeable (one unit of wood is one unit of wood) we didn’t need make a ‘parent’ collection for these entities and simply defined the grid contents be a tuple with type and quantity of the resource, adding an option type for ‘none’.

The jobs for citizen agents could be defined as a tuple. Originally, we had a complex data structure to hold these jobs, but for everything we tried, we ended up with the static analysis needing to reserve the entire collection, a fact that we’ll go into in depth below. In the end we used a simple List. The job selection code looks like the following:

```plaintext
bestJob@Job
bestScore: -1

|availableJobs| → possibleJobs
|possibleJobs AS job| → {
  // if nothing else, we’ll take the first one
  if bestScore < 0 {
    bestJob = job
    bestScore = computeJobScore(job)
  }
}
else {
    currentJobScore: computeJobScore(job)
    if currentJobScore > bestScore {
        bestJob = job
        bestScore = currentJobScore
    }
}
release possibleJobs

Listing 7.9: Job Selection

The rendering was going to be a different challenge. Firstly, this involved
talking to the hardware in a non-trivial way. Up to this point, adding facilities to
the Cadmium standard library for tasks, such as console output and file reading
had sufficed. However, in this case we were compelled to write serious code that
required the C++ interface. We wrote the various versions of renderer using the
popular SDL (Simple Direct Layer) C library. There are already a considerable
number of bindings for using SDL in various languages, but the authors had not
thought to provide Cadmium bindings.

The first, and most important, choice was to consider what was going to ‘live’
in Cadmium and what was going to stay completely on the C++ side. For the
simple 2d renderer (Figure 7.20) we elected to create an interface that allowed
the calling code to specifically refer to the sprites\(^\text{17}\) by number (this was before
we added raw pointers for canneal (Section 7.2.3)). The Cadmium code could
provide screen coordinates and the numerical ‘handle’ to the renderer. When every
image had been placed, it would instruct the renderer to update the screen. This
technique is called double buffering, nearly ubiquitous in the domain where the
next visual frame is constructed in a separate chunk of memory and then in one
step is moved to the screen. Often times this is synced with the refresh of the
monitor (v-sync), but we disabled this for the experiments below, as it created a
great deal of ‘noise’ in the timing that obscured what was actually happening with
the processing. We knew that we didn’t need to worry about the memory objects

\(^\text{17}\)Another term of art. A sprite is a rectangular raster image, with potential transparency, that can
be moved arbitrarily around the screen.
owned by the C++/SDL code. As was detailed earlier a call to an external module will be assumed to conflict with another call to the same module without a flag in its annotation. In this case, the code we wrote was determined to have no chance of conflicts and so the compiler omitted the mutex it would have emitted if not.

This meant that we could represent anything renderable with a simple tuple composed mostly of coordinates and handles, without the need for the heavy duty Entity constructions we needed for Agents. Note that it is common practice to have separate representations of a game object’s simulation presence and its presentation presence. Generally, the simulation representation will in some way ‘own’ the presentation representation and our implementation was no exception.

We chose to model objects in real valued space by storing them in a quadtree [29]. A quadtree is a type of kd-tree used commonly in these applications and is sometimes referred to as a spatial subdivision structure. Essentially, the tree represents finer and finer divisions of a certain space. The root represents the entire ‘space’ in consideration. Every node has zero or four children and each child represents one quadrant of the space represented by its parent. When an object is inserted into the tree, it is assigned to vertex representing the smallest area that strictly contains it. In effect, asking what’s in this area is equivalent to doing a tree walk. This makes it ideal for spatial queries and thus for our purposes.

As with the majority of games, the entire ‘world’ isn’t rendered at one time. Generally, the player is focused on one area. This means that the renderer must select only those objects that are currently visible given the ‘camera’ position, the scope of which is defined by box called viewPort in the main rendering loop:

```
1 | SELECT ALL toRender
2 | FROM renderSpace
3 | WHERE toRender.layer == 0
4 | VISITBY inArea( viewPort ) | → {
5 | ULx: (toRender.position.UL.x - viewOffset.x) *
6 | (1.0/(viewScaleFactor))
7 | ULY: (toRender.position.UL.y - viewOffset.y) *
8 | (1.0/(viewScaleFactor))
9 | LRx: (toRender.position.LR.x - viewOffset.x) *
10 | (1.0/(viewScaleFactor))
```

The tree generally has an externally defined maximum depth.
LRy: (toRender.position.LR.y - viewOffset.y) *
(1.0/(viewScaleFactor))

RenderCanvas::drawImage(
toInt(ULx),
toInt(ULy),
toInt(LRx),
toInt(LRy),
toRender.handle );

| SELECT ALL toRender |
| FROM renderSpace |
| WHERE toRender.layer == 1 |
| VISITBY inArea( viewPort ) | → |
ULx: (toRender.position.UL.x - viewOffset.x) *
(1.0/(viewScaleFactor))''
ULy: (toRender.position.UL.y - viewOffset.y) *
(1.0/(viewScaleFactor))
LRx: (toRender.position.LR.x - viewOffset.x) *
(1.0/(viewScaleFactor))
LRy: (toRender.position.LR.y - viewOffset.y) *
(1.0/(viewScaleFactor))

RenderCanvas::drawImage(
toInt(ULx),
toInt(ULy),
toInt(LRx),
toInt(LRy),
toRender.handle );

RenderCanvas::draw()

Listing 7.10: Rendering from the Quadtree

Note that layer refers to the ‘stacking’ of the images. Layer 0 is background, such as grass, mountains, water, etc and layer 1 is the agents, resources, etc.

In this way, the renderer has a view, and thus uninterrupted access to all the real-spaced representations that it needs to process as a view. The key thing is that the complementary space is left available for modification. An agent will change its rendering representation more often than not during its update. However, most
of them are not being rendered in a given frame and so while the relatively long serial process of pushing pixel data around is happening, the agents who are not being rendered can adjust their spatial positions, giving us a potentially massive reduction in total frame time. This is achieved without the need for an agent to know if it’s being rendered or not. Essentially, SvS ‘sorts’ the agents into being rendered/not being rendered and defers the rendered ones till after the drawing is complete. Most of the code the user requires to achieve this is listed above. This also gives us yet another convenient segue into talking about the performance characteristics.

Performance

Our initial results, with tuned signature width and batch size parameters, are presented in Figure 7.21. The measurement is the average time to complete one frame. We ran 10 repetitions of 3000 frames and discarded the first 1000 of each run to let things ‘warm up’ and achieve a relatively stable state. As a fulfillment of our basic design goals, the single core frame time is under 16 ms, which achieves the current ‘gold standard’ of 60FPS.

We say initial results as the reader may notice that additional cores after the 4th
Figure 7.22: mini-ce – 750 agents – Activity Graph

gives little benefit. This is not due to any deficiencies of the Cadmium scheduler, but large unavoidably serial task that contains the quadtree walking and frame rendering. This can be seen in the activity graph in Figure 7.22, where the renderer task is the large rounded rectangle on the far left (core 0).

This is where video games as an experimental subject demonstrates one of the aspects that makes it a fascinating field for systems research. One can assume that, in the majority of cases, games always want more. If the current performance is better than that required for responsiveness, there is always something more that designers will want to add and players will want to experience: more particles, better anti-aliasing, better collision detection. The list is endless. So, we could increase our workload, simulate more agents, without violating the parameters of the domain.

Our original test simulated 500 agents, 400 citizens and 100 monsters, and we continued to increase the number of agents till we surpassed the threshold of 60FPS for the single core case. This turned out to be 3000 agents total, which gives us the pleasing performance curve in Figure 7.23. Note that while the rendering task does take longer for a greater number of agents, this growth is nowhere near linear. The component of renderer that moves the constructed frame from off-screen buffer to screen is constant, and while there are more agents to render, not all of them will be visible.
Figure 7.23: mini-ce – Population Variation

Figure 7.24: mini-ce – Batch Size Variations
As with all previous experiments, we have evaluated the effects of varying batch size and signature width.

For the batch size, Figure 7.24, we show the variations of the agent update dispatch, as it is, by far, the most sensitive operation in the application. As with previous experiments, we see that a batch factor of 10 (equal to the maximum number of cores) performance suffers, as there is less flexibility in scheduling. Diminishing returns set in around a batch size of 100, where the gains in flexibility are met by the costs in increased overhead.

Unlike previous experiments, we have more than one collection and consequently more than one signature size parameter. Figure 7.25 shows the variation of the signature width and sparsity properties for the quadtree\textsuperscript{19} and Grid collections. We omit the width of the Set, for clarity, as it had the least discernable effect and we omit the List due to the fact that, as we detailed above, static analysis always employed the reservation mechanism, which bypasses the signature mechanism altogether.

\textsuperscript{19}Note the signature width of the quadtree is limited to perfect squares. This is due to a simplification in our implementation that is not implicit in the technique. When we observed the small size of the effect between 256 and 1024 we did not feel that the extra effort to remove this limitations was warranted.
Figure 7.26: mini-ce – SvS versus fine grained locks

In the cases with fewer cores, low width for the quadtree and grid seem to perform the best. As the core count rises, a larger width for the grid performs better, but the quadtree still functions well with a low width. This makes sense, as the quadtree accesses are brief in terms of their ‘critical section’ – simply adding or removing an item from the list attached to a particular node and so speed of ‘locking’ is paramount. On the other hand, accesses to the grids tend to be longer, with the agent evaluating the contents of a block and making choices, so the cost of a false positive will be higher. At maximum core count, both cases benefit from being on an end, albeit opposite ends, of the speed/accuracy tradeoff spectrum.

Finally, as this was a new application, we don’t have a previous application for comparison. Instead, we took the compiler generated C++ code and replaced the signature interaction code with fine grained locks. The results of these efforts are shown in Figure 7.26. As we anticipated, the SvS version outperforms the classic locking version. The difference between the two curves shows the additional overhead of taking the many locks required to secure the required subcollections to satisfy each query. While the SvS version outperforms the locking version at

---

20We also produced a version with coarse ‘global’ locks, but the incremental performance gains for each additional cores was so miniscule that it was not considered to be a viable example.
all times, the gap between the two curves does decrease as the core count rises. This partially reflects the difference in the way the two techniques handle higher contention, but notably includes the effects of false positives that the SvS version can suffer from, while the locking version does not.

**Insights**

At the start of this section, we talked about the dangers of ‘over fitting’ a general solution to a specific example. This was something we tried to avoid in the design of this system. It turns out that we need not have worried so much. Even though the patterns of this software inspired a lot of the design of Cadmium, we still encountered a number of times where the model was awkward or insufficient: a lot of *Oh, yeah. That’s not going to work like I thought it was going to*. At least we could console ourselves on the long nights of restructuring and rethinking that we were being honest about ‘eating our own dogfood’, as the saying goes.

One easily isolated example of this is pathfinding. That is the process an agent initiates to determine the route to get from point $a$ to point $b$, while avoiding impassable obstacles. Pathfinding in mini-ce was done, as is standard practice, with an implementation of the $A^*$ algorithm [36]. We attempted to parallelize this algorithm, but found two things. Firstly, any performant implementation seemed to need to reuse memory buffers and so the language would need the concept of *thread local* storage or similar and as we have stated, we have tried to see how far we could get without directly bringing concepts like ‘threads’ into the language (outside of annotations). Secondly, the algorithm, as written, is definitely sequential as it evaluates the ‘best’ point at every step. This would need to be reconsidered. We realized we would need the complement to an accumulator to hold the priority queue – a ‘source’ to the accumulator’s ‘sink’. This idea excites us a great deal, but due to scope limitations we’re forced to leave it to future work. In the end, we used a serial implementation that the author had already written in C++, showing, yet again, the virtues of the C++ interface. However, because it was serial\textsuperscript{21}, it made for a dramatically unbalanced workload, which only got worse as the number of cores increased, as the serial sections remained the same length. To alleviate this,

\textsuperscript{21}At least for each instantiation of the algorithm, we used the information available from Cadmium Context (§5.1) C++ representation to implement Thread Local Storage
we constructed a heuristic version that found the closest point to goal that could be found by examining no more than an arbitrary number of locations. This worked to tame the extreme variance of the update times, but would not have been necessary if we were able to better express some variation on A*.

We won’t be satisfied with the next iteration of Cadmium unless it can express an algorithm with the following properties:

- a central collection (such as the priority queue in A*) that can safely receive and dispatch new items – as with the channel interface, message passing generalization

- a way to elegantly describe a stopping condition (such as finding the destination in A*) akin to a parallel_while as a complement to the well established parallel_for.

A deeper example was in ways we didn’t anticipate the Entity semantics interacting with the code analysis. The agent update is performed by sending a broadcast message to the set of agents, which essentially gives the receiving Entity a view of its own members. So, if two agents are involved in an interaction that modified both Entities, the code analysis would trigger, correctly, that deadlock was possible. For these more complex interactions we were compelled to perform a more complicated version of the separation inducing restructuring we had to perform for mesh refinement (Section 7.2.2). In that case, we first needed to do a read-only search of the graph to ascertain what we had to change, and a second modification pass to fix everything we found in the first step. In this case, we had to defer these complex interactions till after the main update loop and store the subjects and any parameters. Again, no serial programmer would have ever written code like that.

The code analysis reports are there to guide the programmer away from what isn’t going to work, but the semantics of the language should also guide the programmer towards a good solution. Unlike with pathfinding, we don’t have a clear set of criteria to know when we’ve solved this problem. However, we do notice that this problem, mesh refinement and to a certain extent the list searching problem discussed below all involve the need to do a read-only search, followed by a set of modifications, indicated by the results in the first step.
In general, we found the Entity semantics to be a double-edged sword. They were easy to visualize and reason about, made for well organized code and were very comfortable after nearly two decades of Object-Oriented programming. They were also the cause of a great deal of the awkwardness in implementation. In the end we conclude that Entities are a good start for a natural parallel model, but further work is needed to push this further for a truly effective composition of code and data.

As eluded to several times in this section, we ran into problems allowing concurrent access to the List that contained the jobs. The problem was not the code analysis forcing the inclusion of the reservation mechanism, but the exhaustive search. The citizen needs to see all the potential options before choosing one, and that eventual selection will not be known until the last job is examined. It may have been possible to modify the algorithm so that it uses the two part decomposition detailed above and to check if the job selected still exists, though given that this List is unordered and unindexed, we had no mechanism to retrieve a specific job from the list. The pattern of needing to reserve the entire collection in order to isolate a subset does occur again and again. The other structures in this example all, by complete chance, have an implicit method going directly to part of the collection in question. Obviously, grid coordinates clearly indicate the substructure (and the corresponding SvS partitions). Only slightly more indirectly, the Set, which is unsurprisingly implemented as a hash table, is able to take the subject of the query and immediately reduce the area of concern to a subcollection that at worst contains the data in question and still allows concurrent access. The quadtree is the most subtle in this respect. Given that the tree has a regular structure and that each node corresponds to specific spatial coordinates for any given rectangular area, a subtree can be determined that contains said rectangle. Otherwise, the query would have been forced to do a walk to find the appropriate subtree and so many of the gains of the system would have been lost.

Another potential solution ‘from a different angle’ would have been to use precomputed query signatures. The idea is to take the contents of a WHERE clause and

\footnote{This was exacerbated by the fact that we had neglected to include any concept of ‘this’ (i.e. a language level symbol that represents the current Entity that is executing). Though, this was an oversight and not a hole in the model and will be rectified in the next version.}
as the collection is populated, build the corresponding signature. For example if we had a query on the job list that was \texttt{WHERE job.danger > 10}, every time a job was added to the List that had the appropriate property, it would be added to a signature stored with the collection. When that particular query was initiated, this signature could be employed as a use-signature in place of doing a full collection search. It would be analogous to adding an index to a database table, trading space and some update time for faster, and in our case more concurrently accessible, queries. As a side effect, the composability of signatures transfers directly to composability of clauses. We considered this design from the beginning, as it follows directly from the intersection of queries and signatures. However, there is a considerable number of details to address, mostly to do with maintaining the stored signatures as the collection is modified. This complexity put the technique out of scope for this project, as it would be a good candidate for a project itself.

After reading the preceding paragraphs, one might be led to believe that we consider this experiment to be a failure. This is not the case, by any means. It certainly was a little disheartening to see our shiny new ideas get battered when exposed to some approximation of real world conditions, however, overall this was a success. We were able to, with a few blemishes, express a number of symbiotic patterns in the our candidate model. We point the reader to the elegance of expressing sharing the render space between the renderer and the agents. Finally, the performance curves show numerically that we have definitely achieved the goal of notable parallel performance.

The point of this exercise was to test our model against real world conditions, both in terms of performance and expressability. We had a number of successful results, which validated our direction. We also had a number of failures, which we have spent a lot of text dissecting. This is, however, the point of research. To try new things and learn from them. Given that we have shown that our foundation is sound, in the end each of our failures lead to directions for future work, which gives us one final segue to a discussion of the future directions suggested by this project.
Chapter 8

Conclusion and Future Work

By this point, we hope we’ve made a strong case for the viability of our model. By doing a ‘lateral’ design that explored and exploited the inextricable link between the way software encodes the intentions of the programmer and the minutia of runtime behavior, we tried to show a rough sketch of what an entire ecosystem would look like. We hope that we’ve shown that by considering the problem of parallelism wholistically, even complex systems can be tamed and programmers can use all these cores that the hardware designers have given us, while still focusing on the subjects of their own expertise.

We had lofty goals and no single project is going to solve all the problems in the domain. What this represents is a start. Certainly, we developed some interesting and powerful algorithms and techniques, but we also gained a lot experience trying things in a different way. We hope that our failures are also valuable, leaving threads to weave into something better.

Throughout this work, we have used the phrase future work many times. In video games there is the concept of a ‘vertical slice’, where the studio builds one segment of the game to show off what the final product will look like. This could be one segment of a level or a certain encounter. Scenery is modeled, characters are animated and music is composed. The idea is to present the experience of the finished product, even if only for a few minutes. If one were to move their avatar outside this prepared area it would probably crash or at least be incomprehensible. Generally, this slice is used to show a publisher what it would look like fully re-
alized to secure the funding to make it a reality. Once this funding is granted the long road of fleshing out the rest of the game begins, working many long hours to bring the rest in line with that vision.

Our wish list for future Cadmium (and DnC as a whole) development is longer than an overly-optimistic child’s letter to Santa. We’ve listed many of these throughout this text, but they fall into a few major categories:

**Collections** As we mentioned our prototype only implements parts of the interface described, mostly focusing on fully fleshing out those facets needed for our tests. As well, it will be necessary to flesh out our user provided collections. There are a lot of interesting problems in specifying how a collection responds to various queries. Futhermore, there is still a great deal of untapped potential in different signature schemes. The partitioning of collection space has a noticeable effect on the accuracy of the heuristic which in turn has an effect on performance.

**Queries** We gained a lot by adding a richer set of constructs to describe the data needed for an operation, but this could go so much further. More ways of describing operations, more ways of inserting complex behavior, queries that feed into one another like an embedded map/reduce, and so much more.

**Messages** One key concept we had to abandon to keep the scope of this sane was the idea of message channels, essentially marrying the best of ‘dataflow’ systems with ours. Another was the idea of asynchronous messages. Our runtime supports it, but our static analysis does not. As well, we only scratched the surface of bidirectional messages with the query/view process.

**Program Flow** Our phase breakdown was enough for us to express complex applications like our video game example, but left some ‘negative space’ that’s aching to be filled. Allowing an entity to receive something like a completion receipt when some other operation has completed would dramatically increase the expressiveness of the system. There is a real use for a properly realized ‘comefrom’. Furthermore, there is much more that can be done with system messages and interactions with the scheduler.

**Reporting and Debugging** We were pleasantly surprised at how amazingly helpful just our Code Analysis Reports and autogenerated Activity Graphs were in devel-
oping our application. There really is no such thing as too much insight into what’s going on in a complex application\(^1\).

**Feedback-Directed Optimization** Another avenue that we originally intended to explore, but were forced to abandon to keep this within scope, was the idea that we could use the same ‘hooks’ that we used to generate the activity graphs to generate program traces that could be, in turn, fed back into the compiler to help tune parameters like signature size and batch factor. The fact that our language has a greater semantic richness means that any tool built to optimize it has a leg up in figuring out how pieces relate to each other and our algorithms expose a lot of ‘knobs’ to turn for exploration. The reader may have noticed that in our evaluations (Ch 7) we had a lot of different combinations to try to find the best results. This is a time-consuming process and while not necessary for correctness (or even decent performance) our goal was always to allow the programmer to think about parallelism as little as possible and still get results.

Going further afield, there are even more ambitious directions, such as the things we always get asked about in the introduction (§1.7). There’s also more low level optimizations, such as increased optimization for cache utilization by leveraging our augmented knowledge of the data being scheduled.

So, if you’ll excuse us, we have work to do\(^2\)

\(^1\)Said by someone who uses Compiler Explorer ([https://godbolt.org](https://godbolt.org)) for fun.

\(^2\)Perhaps after taking a break somewhere where nobody says the word ‘parallel’.
Bibliography


doi:10.1007/978-0-387-09766-4_119. URL
https://doi.org/10.1007/978-0-387-09766-4_119. → page 22

http://dx.doi.org/10.1155/2010/521797. → page 19

https://doi.org/10.1145/1133255.1133983. → page 18


https://doi.org/10.1145/362759.362813. → page 16


determination of minimum cost paths. IEEE Transactions on Systems
Science and Cybernetics, 4(2):100–107, 1968. → page 131

[37] K. Havelund. Using runtime analysis to guide model checking of java
programs. In International SPIN Workshop on Model Checking of Software,

[38] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In
Proceedings of the Twenty-First Annual Symposium on Principles of
Distributed Computing, PODC ’02, page 280–289, New York, NY, USA,
→ page 61

0123705916. → page 3

[40] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software
transactional memory for dynamic-sized data structures. In Proceedings of
the Twenty-Second Annual Symposium on Principles of Distributed
Association for Computing Machinery. ISBN 1581137087.
→ page 18

for artificial intelligence. In Proceedings of the 3rd International Joint
Conference on Artificial Intelligence, IJCAI’73, pages 235–245, San
http://dl.acm.org/citation.cfm?id=1624775.1624804. → page 22

ISSN 0362-1340. doi:10.1145/2248487.2151013. URL
https://doi.org/10.1145/2248487.2151013. → page 89

and efficient graph analysis. In Proceedings of the Seventeenth International
Conference on Architectural Support for Programming Languages and
Operating Systems, ASPLOS XVII, pages 349–362, New York, NY, USA,


[63] M. Moir. personal communication. → page 17


