Efficient and Scalable Parallel Functional Programming Through Disentanglement

Sam Westrick

CMU-CS-22-141

August 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Umut A. Acar, Chair
Guy E. Blelloch
Jan Hoffmann
Matthew Fluet (RIT)
Alex Aiken (Stanford)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2022 Sam Westrick

This research was sponsored by the National Science Foundation under award numbers CCF-1314590, CCF-1408940, CCF-1629444, CCF-1901381, CCF-2028921, CCF-2107241, CCF-2115104, and CCF-2119352. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: parallel programming, functional programming, parallel algorithms, automatic memory management, garbage collection, disentanglement, hierarchical memory management, race conditions

For my parents

Abstract

Researchers have argued for decades that functional programming simplifies parallel programming, in particular by helping programmers avoid difficult concurrency bugs arising from destructive in-place updates. However, parallel functional programs have historically underperformed in comparison to parallel programs written in lower-level languages. The difficulty is that functional languages have high demand for memory, and this demand only grows with parallelism, causing traditional parallel memory management techniques to buckle under the increased pressure.

In this thesis, we identify a memory property called *disentanglement* and develop automatic memory management techniques which exploit disentanglement for improved efficiency and scalability. Disentanglement has broad applicability: (1) it can be guaranteed by construction in functional programs; (2) it is implied by *determinacy-race-freedom*, a classic correctness condition; and (3) it allows for general reads and writes to memory as long as concurrent threads do not acquire references to each other's allocated data. Additionally, entanglement (i.e., violations of disentanglement) can be detected dynamically during program execution with nearly zero overhead in practice. To exploit disentanglement for improved efficiency, we partition memory into a tree of heaps, mirroring the dynamic nesting of parallel tasks, which allows for allocations and garbage collections to proceed independently and in parallel. We develop multiple garbage collection algorithms in this setting.

There are two significant implementation efforts presented in this thesis. The first is the MPL ("maple") compiler for Parallel ML, which extends the Standard ML functional programming language with support for (nested) fork-join parallelism. In MPL, we implement all of our memory management techniques based on disentanglement. The second is the Parallel ML Benchmark Suite, which provides implementations of sophisticated parallel algorithms for a variety of problem domains, ported from state-of-the-art C/C++ benchmark suites. All of these benchmarks are disentangled, which further evidences the wide applicability of our approach. In multiple empirical evaluations, we show that MPL outperforms modern implementations of both functional and imperative languages. Additionally, we show that MPL is competitive with low-level, memory-unsafe languages such as C++, in terms of both space and time. These results demonstrate that, through disentanglement, parallel functional programming can be efficient and scalable.

Acknowledgments

In the final year of my undergraduate education, Umut Acar asked me about my plans after graduation. My honest answer was: "I don't know." In that moment, I believe he realized that I was in need of a little guidance. Umut soon convinced me to apply for the Ph.D. program, and later became my advisor. Without a doubt, I would never have completed a Ph.D. without Umut's encouragement and support. Throughout the past six years, Umut and I have developed a rapport which blurs the lines between work and fun; one minute we're cracking jokes, the next we're lost in the weeds of a complicated proof. I look forward to every meeting, and could not have asked for a more enjoyable Ph.D. experience. Umut, thank you.

I'd like to thank the members of my thesis committee: Guy Blelloch, Matthew Fluet, Jan Hoffmann, and Alex Aiken. Guy: thank you for being like a second advisor to me; you have always been an advocate for my work, and I relish the opportunity to soak up a bit of new knowledge at each of our meetings. Matthew: thank you for constantly being available throughout this journey, and for supporting not only the theoretical aspects of my work, but also the practical aspects, by getting your hands dirty helping me maintain a complex open-source project. Your ability to quickly navigate even the most subtle technical details is inspiring. Jan and Alex: the time and thought you have put into providing feedback for my thesis has been invaluable, not to mention your support throughout the thesis process. Thank you so much.

I have had so many mentors throughout my life that have helped guide me to where I am today. Thank you Kim Ferrell, Paul Bakeman, Barry Flowe, Amy Birdsong, Erin Freeman, Rich Serpa, Jake Nielsen, Matt Koon, Dave Tetlow, LeRoy Orie, Cameron Ralston, Bryan Hooten, Stephen Neely, Lance LaDuke, Craig Knox, and Doug Brown, for sharing the joy of music with me. Thank you Tom Harley, Carla Keyes, Quincy Worthington, Emile Harley, Esta Jarrett, and Doug Brown, for showing me the value of community. Thank you Bob Harper, Umut Acar, Guy Blelloch, Phil Gibbons, Margaret Reid-Miller, Danny Sleator, Anupam Gupta, Mor Harchol-Balter, Karl Crary, Frank Pfenning, Steve Brookes, and Rob Simmons, for giving me the technical skills I needed to succeed and showing me how to navigate technical topics with ease, both in teaching and in life. And thank you Daniel Bartels, for inspiring me to pursue science in higher education, and for showing me that the process of science can be endlessly rewarding.

I am fortunate to have so many friends, both old and new, who every day make life a joy. Dane Orie, Michael Goolsby, Zack Verham, Marc Breidenbaugh, Lance Brown, Ovander Bosher, Doug Tibbett, Bryn Davis, James Thompson, Jarrell Raper, Daniel Lehman, and so many more friends from back home; Alek Kirchmann, Alan Chang, Christian Kasilag, Felicia Alfieri, Greg Campbell, Roy Koganti, Roberto Jaime, Ian Rosado, DeOnte Means, Austin Cheng, Craig Barretto, Lanya Tseng, Shannon Lee, Shannon Horgan, Sarah Johnson, Sarah Horton, Elena Feldman, Brian Fischer, Xavi Villalta, Dylan Quintana, Won-Seong Kim, Kevin Louie, Connor Hayes, Nicole Huang, Chris Addiego, and so many more friends from un-

dergrad: thank you all for the countless games, adventures, parties, and memories that will last a lifetime. A shoutout to Alek Kirchmann in particular, who endured me as a roommate for too many years. You’re the best, man. Thanks for the Pyrex.

As a former aspiring musician, I had an unusual entry into the world of computer science. When I switched into computer science in my junior undergraduate year, at first, I felt that I did not belong. Soon enough, however, working with other teaching assistants, I found a home. Thank you Naman Bharadwaj, Bill Duff, Isaac Lim, Chris Powell, Will Crichton, Terence An, and all of the other 15-210 TAs: you helped me find my place. Thank you Shannon Lee, Roy Koganti, and Sam Eisenhandler for tolerating me in group projects and helping me through that transition. Thank you Sam Eisenhandler, for helping me bridge the gap between the two schools; I’ll always remember those all-nighters we pulled to make our way through 15-411. And thank you to all of the 15-122 and 15-210 students throughout the years, from whom I’ve learned so much.

During my graduate studies at Carnegie Mellon, I’ve met so many wonderful people and made so many friends. Thank you Emily Black, Klas Leino, Pedro Paredes, Ryan Kavanagh, Aymeric Fromherz, Sydney Gibson, and Goran Zuzic, for all of the backyard barbeques, drinks, pool parties, hikes, and beach trips. Thank you Jatin Arora, Rohan Yadav, Stefan Muller, Ram Raghunathan, and Mike Rainey, for all the hours we’ve spent together at the whiteboard. Thank you Laxman Dhulipala, for sharing your endless enthusiasm about research and life. Thank you to my office mates Ellis Hershkowitz, Roie Levin, and of course the honorary Greg Kehne, for creating and sharing a space filled with ideas and wit. Thank you Ellis Hershkowitz, Roie Levin, Alex Wang, Anson Kahng, Mark Gillespie, Arjun Teh, Kevin Pratt, Siva Somayyajula, Jalani Williams, Daniel Anderson, and David Kahn, for all the games, hot pots, pot lucks, chanko, and barbeques. (David, I promise I *will* make you a refrigerator cake one day.) Thank you Arjun and Aria for all of your climbing tips; one day, I hope to be at least half as good as you two are!

There are so many friends who have made my experience at CMU a joy. I wish I could name everyone; of course, I cannot, but I will try. Thank you Julian Shun, Yan Gu, Yihan Sun, Laxman Dhulipala, Charles McGuffey, Naama Ben-David, Daniel Anderson, Magdalen Dobson, Hao Wei, Jatin Arora, Ram Raghunathan, Stefan Muller, Ziv Scully, Ben Berg, and everyone else who has been a part of ParallelRG, for helping build a research community of shared interest and diverse experience. Thank you Carlo Angiuli, Evan Cavallo, Daniel Gratzer, Jon Sterling, Favonia, and Joe Tassarotti, for helping me appreciate the depth and subleties of PL research. And a big hearty thank you to Stephanie Balzer, Anna Gommerstadt, Nika Haghtalab, David Witmer, Noam Brown, David Wajc, Angela Jiang, Nic Resch, Chris Fallin, Adrien Guatto, Yue Niu, Yue Yao, Yifan Qiao, Larry Wang, Rose Bohrer, Sol Boucher, Christopher Canel, Priya Donti, Abhiram Kothapalli, Jonathan Laurent, Klaas Pruiksma, Yong Kiam Tan, Jessie Grosen, Leslie Rice, Bailey Flanigan, Paul Gölz, Marina DiMarco, Nirav Atre, Ainesh Bakshi, Pallavi Koppol, Josh Williams, Giulio Zhou, Vinni Bhatia, Matteo Bonvini, Holly Bossart, James Carzon, Meg Ellingwood, Anni Hong, Addison Hu, Victoria Lin, Terrance Liu, Tudor

Manole, Mikaela Meyer, Kayla Scharfstein, Kyle Schindl, Konrad Urban, Galen Vincent, Julia Walchessen, Catherine Wang, Ian Waudby-Smith, Neil Xu, Cindy Wang, Elaine Fath, and so many more. I appreciate every single one of you, and I hope that we will stay in touch.

Thank you to all of the folks at Five Points Bakery, for getting to know me as I rapidly depleted your supply of chocolate croissants. I will never attempt to calculate how many dollars I've spent on these treats; instead, I will sit happily with the knowledge that I had the opportunity to support the best bakery in Pittsburgh. See you tomorrow, probably, for another croissant.

To my family: thank you Linda, Jane, Hector, Ted, Ashley, Nana, Papa, and my many aunts, uncles, and cousins, for your love and support. I am so grateful that our family has stayed close over the years. Many of you saw my thesis defense over Zoom; thank you for taking time out of your lives to come support me on such a momentous occasion!

Thank you Maya, for your support, your thoughtfulness, your encouragement, your laugh, and the smile you put on my face every day. It was a stroke of luck to find each other at the height of the Covid pandemic, and I am so glad we did. I can't imagine these past two years without you.

Finally, I thank my parents, who instilled in me an endless curiosity—an insatiable desire to learn more. This is a quality which I cherish, and which defines who I am. From my parents, I learned the value of going further, of asking the next question, of not being satisfied until I could pass on my knowledge and expertise to others. They taught me how to *love the process*, a skill which I rely on every day. I dedicate this thesis to my parents, as a small way of saying thank you, for raising me, and for shaping me into the person I am today. I could not not have done even a fraction of this work without their support. Mom, Dad: your love means the world to me. Thank you.

Contents

1 Introduction **1**

2 Parallel Functional Programming with Parallel ML **7**

2.1 Purely Functional Algorithms . . . 8

2.1.1 Parallel Reduction . . . 8

2.1.2 Maximum Contiguous Subsequence Sum . . . 8

2.1.3 Sparse Matrix-Vector Multiplication . . . 9

2.1.4 Tokenization . . . 10

2.2 Parallel Array Operations . . . 11

2.2.1 Tabulate . . . 12

2.2.2 Scan (Parallel Prefix Sums) . . . 12

2.2.3 Filter . . . 13

2.2.4 Flatten . . . 15

2.2.5 Discussion: Fusion with Function Composition . . . 16

2.3 Non-deterministic Parallel Algorithms . . . 17

2.3.1 Parallel BFS . . . 17

2.3.2 Parallel Deduplication by Concurrent Hashing . . . 19

3 Disentanglement **23**

3.1 Language and Graph Semantics . . . 23

3.1.1 Syntax . . . 24

3.1.2 Computation Graphs and Actions . . . 24

3.1.3 Open Computation Graphs . . . 25

3.1.4 Operational Semantics . . . 26

3.2 Example: Transposing Points in 2D . . . 28

3.3 Definition of Disentanglement . . . 30

3.4 Disentanglement and Race-Freedom . . . 31

3.4.1 Proof: Race-Freedom Preserves Disentanglement . . . 35

3.5 Disentanglement Beyond Race-Freedom . . . 41

4 Disentangled Memory Management **45**

4.1 Preliminaries: Heaps and Heap Objects . . . 46

4.2 Heap Hierarchy . . . 47

4.2.1 Pointer Directions . . . 47

4.2.2 Guarantees of Disentanglement 48
4.2.3 Relationship to Computation Graphs 48
4.3 Subtree Collection 49
4.3.1 Tracing Phase 49
4.3.2 Optional Promotion Phase 50
4.3.3 Example 50
4.3.4 Correctness 52
4.3.5 Independence of Subtree Collections 52
4.4 Scheduling and Local Garbage Collection (LGC) 52
4.5 Concurrent Garbage Collection (CGC) 54
4.5.1 Primary Heaps and CGC-heaps 54
4.5.2 CGC Chaining 55
4.5.3 Pointer Directions, Revisited 56
4.5.4 CGC Snapshotting and Tracing 57
4.5.5 CGC Scheduling 58
4.6 Collection Policy 58

5 Entanglement Detection **59**
5.1 Overview 60
5.2 Entanglement and Determinacy Races 60
5.3 Language and Graph Semantics 62
5.3.1 Parallelism, Task Trees, and Computation Graphs (Dags) 65
5.3.2 Entanglement Detection 66
5.3.3 Example Revisited 67
5.4 Soundness and Completeness 67
5.4.1 Completeness Proof 69
5.4.2 Soundness Proof 69
5.5 Entanglement Detection Cost Analysis 72
5.5.1 Utilizing Heap Chunks to Optimize Space 72
5.6 Entanglement Candidates 73
5.6.1 Marking and Unmarking Candidates 73
5.6.2 Cost Analysis of Tracking Candidates 74
5.6.3 Candidate Arrays 75
5.6.4 Asymptotically Fewer Graph Queries 75
5.6.5 Candidates in the Detection Semantics 75

6 The MPL Compiler for Parallel ML **79**
6.1 Scheduler 79
6.1.1 Thread and Heap Maintenance 80
6.1.2 Scheduler Jobs and Synchronization 83
6.1.3 Implementing the `par` Function 85
6.2 Block Allocator 85
6.3 Heaps and Heap Objects 87
6.4 Parallel Initialization of Sequence Objects 89

6.5 Heap Queries . . . 90
6.5.1 Memory Reclamation for Union-Find Nodes . . . 90
6.5.2 Allocation for Heap Records and Union-Find Nodes . . . 91
6.6 Remembered Sets and Write Barriers . . . 91
6.7 Garbage Collection . . . 91
6.7.1 LGC . . . 92
6.7.2 CGC . . . 93
6.7.3 Amortization Policy for LGC and CGC . . . 93
6.8 Entanglement Detection Implementation . . . 95
6.8.1 Vertex Identifiers and SP-order maintenance . . . 95
6.8.2 Read and Write Barriers for Detection . . . 95
6.8.3 Memory Management for Detection . . . 96
6.8.4 Chunk Pinning: Handling the Possibility of Entanglement . . . 97

7 The Parallel ML Benchmark Suite **101**
7.1 Graph Algorithms . . . 101
7.2 Computational Geometry . . . 102
7.3 Images and Audio . . . 103
7.4 Text Processing . . . 104
7.5 Numerical Algorithms . . . 104
7.6 Other Algorithms . . . 105

8 Evaluation **107**
8.1 Overview . . . 107
8.2 Methodology and Experimental Setup . . . 109
8.3 Overheads and Scalability . . . 110
8.4 Comparison with Multicore OCaml . . . 115
8.5 Comparison with Java and Go . . . 117
8.6 Comparison with C++ . . . 120
8.7 Evaluation of Entanglement Detection . . . 122
8.7.1 With and Without Entanglement Detection . . . 123
8.7.2 Improvement Due To Entanglement Candidates . . . 124
8.7.3 Entangled Tests . . . 126

9 Related Work **127**
9.1 Parallel Memory Management . . . 127
9.2 Race Detection . . . 129
9.3 Parallel Programming Languages . . . 130

10 Concluding Remarks **133**
10.1 Discussion . . . 133
10.2 Conclusion . . . 136

Bibliography **137**

List of Figures

2.1 Sequential left-fold . . . 8
2.2 Divide-and-conquer parallel reduction in Parallel ML. . . . 9
2.3 Maximum contiguous subsequence sum . . . 9
2.4 Sparse matrix-vector multiplication . . . 10
2.5 Tokenization . . . 11
2.6 Sequential and parallel for-loops. . . . 12
2.7 Parallel array tabulation . . . 12
2.8 Three phases of scan . . . 13
2.9 Parallel scan (prefix sums) . . . 14
2.10 Parallel filter . . . 15
2.11 Parallel flatten . . . 16
2.12 BFS in Parallel ML. This algorithm is non-deterministic. . . . 18
2.13 Alternative implementation of `tryVisit` for parallel BFS, based on priority updates. The resulting parents computed for each vertex are deterministic. . . . 19
2.14 Parallel deduplication by concurrent hashing . . . 20

3.1 Syntax . . . 24
3.2 A series-parallel computation graph is either the empty graph $\bullet$, a single action α, a sequential composition $g_1 \oplus g_2$, or a parallel composition $g_1 \otimes g_2$. The dashed lines are control dependencies, implicitly pointing down. . . . 25
3.3 Language dynamics (main computation steps). . . . 27
3.4 Language dynamics (additional administrative rules). . . . 28
3.5 The function `transpose` transposes each element in array P of length n. . . . 29
3.6 Computation graph for `transpose` on an input array of length 4. . . . 29
3.7 Auxiliary Definitions: expression roots, and graph allocations and writes. . . . 31
3.8 Definition of disentanglement. Variable A denotes the set of known allocations. 32
3.9 Definition of determinacy-race-freedom. Variable F denotes a “forbidden” set of locations (that are allocated or updated by a concurrent task). . . . 33
3.10 Strengthening of disentanglement with the guarantees of simultaneous determinacy-race-freedom. . . . 34

4.1 Forks and joins. Active tasks are black circles, and suspended tasks are white circles. Each task has a heap, drawn as a gray rectangle. . . . 47
4.2 A disentangled heap hierarchy. Up, down, and internal pointers (solid) are permitted. Cross-pointers (dotted) are disallowed. . . . 47

4.3 Before and after an example subtree collection of the bottom-most three heaps, with and without the optional promotion phase. The large rectangles are heaps, the squares are objects, and the diamonds are root objects. Highlighted groups **A** and **B** are kept live due to down-pointers. The group **C** is garbage and is reclaimed. 51
4.4 Example heap assignments for local collections, with two processors (**P1** and **P2**) and two corresponding active tasks. 54
4.5 Spawning a new CGC-task: processor **P1** pushes heap **A** onto a CGC-chain and continues with a fresh (primary) heap. 55
4.6 Two cases handling CGC-chains at joins. The rectangles are heaps, and the ovals are chains containing one or more heaps. Primary heaps are shaded. . . . 56

5.1 Example disentangled program. 61
5.2 Syntax 62
5.3 Execution with entanglement detection (main computation steps). 63
5.4 Execution with entanglement detection (administrative rules). 64
5.5 Functions *fork* and *join* on computation graphs. 65
5.6 Example dag and entanglement checks for the disentangled program in Figure 5.1. 65
5.7 Single-step disentanglement invariant, consisting of memory property for all immutable locations, and disentanglement property for all program "roots". . . 68
5.8 A partial dag on the left and its corresponding task tree on the right. Each node of the tree corresponds to contracted sub-dags, shown delimited by boxes. In the tree, disentangled pointers may point up, down, or internally to a node. An example entangled pointer is shown in dotted blue. 76

6.1 Auxiliary functions used by the scheduler. The modules `Thread` and `Heaps` are implemented in the run-time system and linked as foreign functions. 80
6.2 Example threads and heaps. Each thread has a list of associated heaps at various depths, corresponding to a path of heaps in the heap hierarchy. 81
6.3 Simplified interface of the work-stealing scheduler. 82
6.4 Example usage of `sync_vars`, as provided by MPL's scheduler. The call to `leftSynchronize` blocks until the corresponding call to `rightSynchronize` completes. 82
6.5 Implementation of thread synchronization in MPL, using atomic fetch-and-add operations and switching between first-class threads. 82
6.6 Simplified presentation of the implementation of `par` in MPL. 84
6.7 Example of race between local GC and the entanglement check. Thread A first acquires a pointer to y. Meanwhile, B forwards y to y' and reclaims the old memory. Thread A then proceeds with the entanglement check on a dangling pointer. 98
6.8 Example, before and after heaps **C** and **D** merge into **B**. Afterwards, the down-pointer from **B** into **D** has become an internal pointer, and therefore the indicated chunk may be unpinned. 98

8.1 Speedups in comparison to sequential baseline (group 1). 112

8.2 Speedups in comparison to sequential baseline (group 2). 113
8.3 Two examples of loop-based parallelism using Java Streams, both of which operate on integers i in the range $0 \leq i < N$. The former is a parallel for-loop, the latter a parallel filter. 118
8.4 Example binary fork-join in Go. The expression `pardo(f,g)` runs the functions `f` and `g` in parallel using goroutines (Go's lightweight threads) and channel synchronization. 118

List of Tables

8.1 Comparison with sequential baseline: times, max residencies, overheads (OV), speedups (SU), and space blowups (BU). 111
8.2 MPL vs OCaml: Times (seconds) and max residencies (GB) of MPL (column **M**) and OCaml (column **O**). The ratios $\frac{O}{M}$ are the performance of OCaml relative to MPL. Larger ratios are better for MPL. 116
8.3 MPL vs Java: Times (seconds) and max residencies (GB) of MPL (column **M**) and Java (column **J**). The ratios $\frac{J}{M}$ are the performance of Java relative to MPL. Larger ratios are better for MPL. 118
8.4 MPL vs Go: Times (seconds) and max residencies (GB) of MPL (column **M**) and Go (column **G**). The ratios $\frac{G}{M}$ are the performance of Go relative to MPL. Larger ratios are better for MPL. 119
8.5 MPL vs C++: Times (seconds) and max residencies (GB) of MPL (column **M**) and C++ (column **C**). The ratios $\frac{M}{C}$ are the performance of MPL relative to C++. **Note: smaller ratios are better for MPL**. 121
8.6 Times (seconds), max residencies (GB), and percent differences of MPL relative to MPL^{dd}, which has detection disabled. The percentages in parentheses are the overhead of entanglement detection. The "geomean" is the geometric mean of the ratios $\text{MPL}/\text{MPL}^{\text{dd}}$. 123
8.7 Performance improvement ratio due to tracking candidates, including number of graph queries performed both with and without candidate tracking. 125

Chapter 1

Introduction

Nearly every computing device available today is a parallel computer, ranging from smartphones with 10 cores, workstations with dozens of cores [141], servers with hundreds of cores [51], and high-end machines with thousands of cores [125]. Given the mainstream availability of multicore computers, parallel programming today is increasingly important and relevant. However, parallel programmers face a number of important challenges, making it difficult to write software that is simultaneously correct, efficient, and scalable. Many of these challenges stem from operations on shared memory, where data races can cause potentially disastrous race-conditions, leading to complex and unpredicable bugs [10–12, 38, 40, 41, 61, 104, 113, 146].

Researchers have argued for decades that functional programming can make parallelism simpler and safer by helping programmers avoid race conditions [20, 26, 28, 68, 77, 79, 99, 102, 116, 119, 120, 138, 143, 162]. Any program which is "purely functional" (free of side-effects) naturally avoids all races and is deterministic by default. More generally, functional programming languages and their characteristic strong type systems enable programmers to control memory effects and isolate any potential bugs introduced by the use of in-place updates. Functional programming also allows for expressing parallel algorithms elegantly and succinctly in terms of implicitly-parallel higher-order functions (e.g. map, reduce, filter, scan, etc.) on collections of data. Many parallel functional languages have been developed going back to the 1980s and 90s, including multiLisp [77], Id [20], and NESL [26, 28], and more recently several forms of both parallel Haskell [79, 90, 99, 102, 119], and parallel ML [68, 76, 81, 82, 116, 120, 138, 143, 144, 153, 162]. Some of these languages only support pure or mutation-free functional programs, but others such as Parallel ML [18, 76, 153, 154] also allow for side effects.

Although functional programming has many benefits for parallelism, an elephant in the room remains: efficiency. After decades of research, parallel functional programming languages continue to underperform in comparison to their procedural/imperative counterparts. One of the primary reasons for this is memory: by eschewing in-place updates, functional programs allocate data at a high rate [16, 17, 21, 56, 57, 73, 74, 102]. Efficient memory management techniques have been developed for sequential functional languages, but this remains an open problem in the parallel setting. The central problem is that parallelism increases demand for memory (by increasing allocation rates, as many processors can allocate simultaneously), causing traditional memory management techniques to buckle under the increased pressure.

In this thesis, we tackle the problem of efficient automatic memory management for parallel

functional programs. The crux of our approach is identifying a memory property, called ***disentanglement***, which occurs naturally in parallel functional programs, and which is commonly found in effectful parallel programs as well. We then design and implement automatic memory management techniques which take advantage of the disentanglement property for improved efficiency and scalability. Our techniques allow for allocations and garbage collections to proceed independently and in parallel across a dynamic partitioning of memory into many disjoint heaps, enabling provably efficient parallel GC [18].

Informally, disentanglement ensures that **concurrent threads remain oblivious to each other's allocations**. More specifically, disentanglement restricts access to memory objects that are allocated in the "past", as determined by sequential dependencies. This restriction prevents ***entanglement***—i.e., cross-pointers—between objects allocated by concurrent threads during execution. That is, for any two threads t_1 and t_2 which execute concurrently, disentanglement ensures that t_1 will never acquire a pointer to data allocated by t_2, and vice versa. Individual threads are therefore able to manage their own memory independently (e.g., collect garbage and compact), without needing to synchronize with other concurrent threads.

Disentanglement holds for a wide variety of parallel programs, including both functional programs as well as programs with effects. In particular, purely functional programs (with no in-place updates) are disentangled by construction. More generally, we prove (Theorem 1) that disentanglement is implied by a classic correctness condition called ***determinacy-race-freedom*** [63], which allows for in-place updates while still guaranteeing deterministic execution. Disentanglement therefore allows the programmer to utilize in-place updates in a disciplined (i.e., deterministic and race-free) manner.

The fact that disentanglement allows for in-place updates is important for practical efficiency. Although functional programming generally eschews in-place updates in favor of immutability, we wish to allow for in-place updates if the programmer deems it profitable. Along these lines, a classic technique for improving performance in a functional setting is to hide effects behind a pure interface. This offers the best of both worlds: functions which are both efficient *and* pure (and therefore safe-for-parallelism by default, because the client of the interface is incapable of observing any side-effects). For example, a sequence library can provide purely functional data-parallel operations (map, scan, filter, reduce, etc.) and utilize mutable arrays under-the-hood to improve performance (e.g., for constant-time random access and good data locality). In this case, because typical implementations of these operations are determinacy-race-free, this approach is naturally disentangled.

Moving beyond deterministic programming, we observe an interesting phenomenon: parallel programmers sometimes intentionally interleave atomic in-place updates and accesses in shared memory with the goal of improving performance. A couple examples include techniques such as *priority updates* [134] and *deterministic reservations* [32], both of which utilize a "small amount" of non-determinism to avoid unnecessary synchronization. This non-determinism is desirable from a performance perspective, and safe if programmed with care. Perhaps surprisingly, we show that disentanglement is permissive enough to express these non-deterministic parallel programming techniques. Essentially, as long as concurrent threads only operate on pre-allocated data (i.e., do not expose any fresh allocations), they may communicate freely in a disentangled manner.

We wish to allow for programmers to utilize in-place updates freely, as long as the program

remains disentangled. This immediately raises a question: is it possible to violate disentanglement? And, if so, what are the consequences? Indeed, unrestricted use of in-place updates can lead to *entanglement* (i.e., violations of disentanglement). Furthermore, if our memory management system were to assume disentanglement but not enforce it, then this could result in incorrect and unpredictable behavior. Specifically, if an entangled program were executed, a garbage collector which assumes disentanglement might incorrectly reclaim an object by missing a cross-pointer. This could cause the program to crash, or (worse) return an incorrect result. To avoid this unsafe behavior and preserve memory safety for the programmer, disentanglement must be enforced automatically.

To enforce disentanglement, we present a dynamic approach, where individual memory accesses are monitored during execution, and if entanglement is detected, then the program is (safely) terminated. This allows for all disentangled programs to safely run to completion, including those that are effectful and/or non-deterministic. Our approach therefore avoids ruling out disentangled programs. We make these guarantees precise by formulating soundness and completeness properties (Theorems 2 and 3). Roughly speaking, soundness (a.k.a. "no missed alarms") says that if entanglement is not detected, then the execution is disentangled; similarly, completeness (a.k.a. "no false alarms") says that if execution is disentangled, then entanglement is not detected.

Because entanglement detection occurs dynamically and affects runtime performance, it is essential that it can be made efficient and scalable. Our approach takes inspiration from a long line of work on dynamic race detection for parallel programs [24, 50, 63, 64, 104, 122, 123, 151, 157]. While race detection remains expensive in practice (with overheads exceeding an order of magnitude for sequential runs, e.g., [151, 157]), we show that entanglement can be detected dynamically on-the-fly with close to zero overhead in practice. Entanglement detection therefore can remain "turned on" without noticeably affecting performance, allowing us to rely upon detection to ensure disentanglement.

With entanglement detection, we can safely turn our attention towards the implementation of a memory manager which assumes disentanglement and exploits this property for improved efficiency and scalability. Here, we take advantage of a separation property afforded by disentanglement: memory objects allocated by concurrently executing threads cannot point at each other. We take advantage of this separation by assigning each thread its own heap, in which the thread performs all of its allocations. We then organize memory as a (dynamic) tree of heaps which mirrors the structure of parallelism in the program: the leaves of the tree correspond to concurrently executing threads, and the internal nodes correspond to (suspended) parent threads, waiting for their children to complete. The tree dynamically grows and collapses as the computation proceeds, mirroring the dynamic nesting of parallel tasks, i.e., as new threads are created at forks and as threads are destroyed at joins. This design allows concurrently executing threads to allocate and reclaim memory with no synchronization. Furthermore, it allows concurrent threads to update and access data allocated by shared ancestors without needing to synchronize or *promote* (copy) data; instead, disentanglement makes it possible to delay promotions until opportune moments, such as during garbage collections, or even avoid promotions entirely, if desired for performance. In particular, we find that it is generally more efficient to not promote at all. Avoiding promotions also simplifies the design of the entanglement detector, which is coupled closely with memory management.

A significant component of this thesis is the design and implementation of the MPL ("maple") compiler and run-time system for Parallel ML [9, 18, 153]. MPL extends the MLton [106] compiler for Standard ML with support for nested fork-join parallelism. In MPL, we implement all of our proposed memory management techniques, including parallel garbage collection and entanglement detection. The design of MPL is unique in many ways, especially in its close coupling between scheduling and memory management, where a work-stealing scheduler is simultaneously responsible for assigning both threads and heaps to different processors.

To evaluate MPL and our techniques, we develop a comprehensive benchmark suite for Parallel ML by porting C/C++ implementations from state-of-the-art benchmark suites and libraries, including PBBS [14, 32, 133], ParlayLib [34], Ligra [131], GBBS [54], and PAM [148]. These benchmarks include sophisticated parallel algorithms from various problem domains, including graphs, text processing, digital audio processing, image analysis and manipulation, numerical algorithms, computational geometry, and others. All of these benchmarks are naturally disentangled, which provides further evidence that disentanglement is widely applicable: in C/C++, the original authors of these benchmarks had no knowledge of disentanglement, nor any need to ensure it.

In multiple empirical evaluations, utilizing MPL and our benchmark suite, we show that our techniques are highly efficient and scalable. On 72 processors, in comparison to a fast sequential baseline, MPL achieves between 14-60x speedup while on average using *less* space than the baseline. We also show that MPL outperforms (in terms of both space and time) existing industrial-strength functional and procedural languages, including multicore OCaml, Java, and Go, often by a wide margin. Finally, we find that MPL is competitive with low-level, memory-unsafe languages such as C++. Specifically, including the cost of automatic memory management and GC, MPL is on average less than 2x slower than C++ on 72 processors, while having approximately the same memory footprint.

Overview

The main chapters and contributions of this thesis are as follows:

- Examples of disentangled parallel algorithms written in Parallel ML (Chapter 2), including (i) purely functional algorithms, (ii) effectful and race-free algorithms, and (iii) more general effectful and non-deterministic algorithms. Regardless of their use of effects, all of these examples are "mostly functional" in the sense of being expressed almost entirely in terms of functions with pure functional specifications.
- A formal theory of *disentanglement* (Chapter 3), including a definition in terms of a graph-based semantics and a proof establishing the relationship between determinacy-race-freedom and disentanglement properties.
- Efficient memory management techniques for disentangled programs (Chapter 4), including fully parallel GC algorithms.
- An *entanglement detection* algorithm (Chapter 5) which ensures disentanglement dynamically during execution, including proofs of soundness ("no missed alarms") and completeness ("no false alarms"), and efficient implementation techniques which ensure nearly zero overhead in practice.

- The implementation of the MPL compiler for Parallel ML (Chapter 6). MPL provides a robust implementation of the full Standard ML language extended with support for nested fork-join parallelism. In MPL, we implement all of our automatic memory management techniques, including both entanglement detection and provably efficient parallel GC.
- The Parallel ML Benchmark Suite (Chapter 7), consisting of over 30 sophisticated parallel benchmarks ported from state-of-the-art C/C++ benchmark suites and libraries, as well as an accompanying library of key data structures and algorithms.
- A comprehensive empirical evaluation (Chapter 8), demonstrating that our memory management techniques are practical, with low time overhead, good scalability, and low memory footprint.

Peer-Reviewed Publications

This thesis contains work that also appeared in the following publications.

- *Disentanglement in Nested-Parallel Programs* [153], at POPL'20. The contributions of this paper include the definition of disentanglement and its connection with determinacy-race-freedom (Chapter 3), disentangled memory management techniques (Chapter 4, Sections 4.1-4.4), and the first version of the MPL implementation (subsumed here by Chapter 6). The paper also presents an empirical evaluation, using benchmarks and comparisons which have been incorporated into the Parallel ML Benchmark Suite (Chapter 7) and our expanded evaluation (Chapter 8).
- *Entanglement Detection with Near-Zero Cost* [154], at ICFP'22. The contributions of this paper include our entanglement detection algorithm (Chapter 5), proofs of its soundness, completeness, and efficiency (Sections 5.4 and 5.5), memory management techniques for entanglement detection (Section 5.6), an efficient implementation in MPL (Section 6.8), and an empirical evaluation (Section 8.7).

Thesis Statement

Altogether, our results in both theory and practice support the following thesis statement:

> **Through *disentanglement*—a common memory property—it is possible to automatically manage the memory of parallel functional programs efficiently and with good scalability.**

Chapter 2

Parallel Functional Programming with Parallel ML

The setting for this work is a programming language we call Parallel ML, which is based on the Standard ML (SML) functional programming language. Parallel ML extends SML with nested fork-join parallelism by providing the programmer with a single construct called `par`. The expression $\texttt{par}(f, g)$ evaluates $f()$ and $g()$ in parallel and returns their results as a tuple. Throughout this thesis, all parallelism (in our Parallel ML algorithms and benchmarks) is expressible in terms of just `par` alone.

val `par`: $(\texttt{unit} \to \alpha) \times (\texttt{unit} \to \beta) \to \alpha \times \beta$

In this chapter, we present a number of examples of parallel algorithms written in Parallel ML. These examples include purely functional codes (with no in-place updates), efficient pure libraries which use effects under the hood for improved performance, as well as "mostly functional" parallel algorithms that require atomic in-place updates (such as atomic compare-and-swap operations) in specific situations, but otherwise are written in a functional style. We use standard functional programming language features, including tuples, algebraic datatypes, pattern matching, higher-order functions, mutable references and arrays, etc.

Note that all examples in this chapter are disentangled, but we do not discuss disentanglement here, deferring instead to Chapter 3.

Parallel Tuples

The code examples in this chapter use the notation $(e_1 \parallel e_2)$, which resembles the normal ML notation for a binary tuple, written (e_1, e_2). That is, $(- \parallel -)$ is the parallel equivalent of a normal "sequential" tuple. It can be implemented in terms of the primitive `par` construct, defined as follows.

$$(e_1 \parallel e_2) \triangleq \texttt{par}(\texttt{fn}() \Rightarrow e_1,\ \texttt{fn}() \Rightarrow e_2)$$

Arrays

When operating on arrays, we write $|a|$ for the length of the array, $a[i]$ for reading the ith element, and $a[i] := v$ for updating the ith element with value v. In actual Parallel ML code,

```
1 // Sequential loop: fold−left, using "accumulator" variable a.
2 // Computes g(··· g(a, f(i)) ··· , f(j − 1))
3 fun foldl (g : β × α → β) (a : β) (i, j, f: int → α) : β =
4   if i ≥ j then
5     a
6   else
7     foldl g (g(a, f(i))) (i + 1, j, f)
```

Figure 2.1: Sequential left-fold

these are written respectively `Array.length`(a), `Array.sub`(a, i), and `Array.update`(a, i, v).

Granularity control

To ensure that the cost of `par` (i.e., the cost of exposing parallelism) is well-amortized, we will use manual granularity control in the form of constant thresholds, as is the standard approach in many parallel languages. Solving the so-called *granularity control problem* is an on-going area of active research [2, 7, 121], orthogonal to the issues of disentanglement and memory management considered in this thesis.

2.1 Purely Functional Algorithms

2.1.1 Parallel Reduction

Many purely functional parallel algorithms are expressible using a single higher-order function called `reduce`. The reduction primitive computes the "sum" of a collection of elements in parallel. The "sum" here is relative to an associative (not necessarily commutative) binary function $g : \alpha \times \alpha \rightarrow \alpha$ together with a corresponding identity element z. The `reduce` function then takes a triple (i, j, f) representing the elements $[f(i), \ldots, f(j-1)]$, and outputs the sum of these elements.

An implementation in Parallel ML is shown in Figure 2.2, using a divide-and-conquer algorithm, where the input range is split in two, the two halves are each individually summed in parallel, and then finally the overall sum is computed. Below a threshold, for granularity control, the algorithm switches to a sequential fold (Figure 2.1).

2.1.2 Maximum Contiguous Subsequence Sum

The *maximum contiguous subsequence sum* problem, also known as the *maximum subarray problem*, is the problem of finding a contiguous subarray which has largest overall sum. A classic parallel solution is based on divide-and-conquer, where contiguous segments of the input are summarized by a four tuple (ℓ, r, b, t): the values ℓ and r are respectively the maximum prefix and suffix; the value b is the best solution within the segment, and t is the total sum of the contiguous segment. A simple associative function, `combine`, can then be used to compute the summary of a larger segment in terms of the summaries of its two halves. For example, the new

```
1  // parallel reduce, computes the "sum" of [f(i), f(i+1), ..., f(j−1)]
2  // with respect to associative function g with identity element z
3  fun reduce (g : α × α → α) (z : α) (i, j, f: int → α) : α =
4    if j − i ≤ GRAIN_THRESHOLD then
5      // if smaller than constant threshold, do sequential instead of parallel
6      foldl g z (i, j, f)
7    else
8      let
9        val mid = ⌊(i + j)/2⌋
10       val (l, r) = (reduce g z (i, mid, f) ∥ reduce g z (mid, j, f))
11     in
12       g(l, r)
13     end
```

Figure 2.2: Divide-and-conquer parallel reduction in Parallel ML.

```
1  fun combine((ℓ1, r1, b1, t1), (ℓ2, r2, b2, t2)) =
2    (max(ℓ1, t1 + ℓ2),
3     max(r2, r1 + t2),
4     max(r1 + ℓ2, max(b1, b2)),
5     t1 + t2)
6
7  fun singleton v = let val p = max(v, 0) in (p, p, p, v) end
8
9  fun mcss (a: real array) =
10   let
11     val (_, _, b, _) = reduce combine (0, 0, 0, 0) (0, |a|, fn i ⇒ singleton(a[i]))
12   in
13     b
14   end
```

Figure 2.3: Maximum contiguous subsequence sum

best solution for the combined summary is $\max(r_1 + \ell_2, \max(b_1, b_2))$, i.e., either one of the best solutions b_1 or b_2 found so far, or a new best solution $r_1 + \ell_2$ which crosses the halfway point, consisting of the best suffix on the left combined with the best prefix on the right.

In Figure 2.3, we present an implementation of this classic parallel algorithm in Parallel ML. The function `mcss` takes as input an array of real numbers, and outputs the maximum sum amongst all contiguous subsequences. It is written in terms of a single call to `reduce`, which "automates" the divide-and-conquer process. In this case, the function `singleton` is used to summarize each individual element as a four-tuple (ℓ, r, b, t), which the `reduce` then combines in parallel as described above.

2.1.3 Sparse Matrix-Vector Multiplication

The function `sparseMxV` in Figure 2.4 implements a purely functional sparse matrix-vector multiplication. It takes a sparse matrix M and a dense vector V as argument, and returns a dense

```
1  type sparse_row_vector = (int × real) array   // (index, value) pairs
2  type sparse_matrix = sparse_row_vector array
3
4  type dense_vector = real array
5
6  fun sparseMxV(M: sparse_matrix, V: dense_vector) : dense_vector =
7    let
8      fun f(i,x) = V[i] * x
9      fun rowSum(r: sparse_row_vector) = reduce (op+) 0 (0,|r|,fn j ⇒ f(r[j]))
10   in
11     tabulate(0, |M|, fn i ⇒ rowSum(M[i]))
12   end
```

Figure 2.4: Sparse matrix-vector multiplication

vector as result.

In this code, sparse matrices are represented as arrays of sparse vectors. Each sparse vector is encoded with the type `(int × real) array`, which is an array of index-value pairs. In particular, each (i, x) in a sparse vector indicates that the ith index of the vector has the value x. Every unmentioned index takes the value 0.

The multiplication itself is implemented here in terms of a `tabulate` to construct the output vector. The `tabulate` function is described in Section 2.2; at a high level, it simply allocates and initializes an array in parallel by evaluating the function given as argument at each index. For sparse matrix-vector multiplication, each index of the output is computed as the sum across one sparse row of the matrix. This summation is succinctly expressed as a `reduce` operation on a row r, using floating-point addition as the combining function, where where each element $(i, x) \in r$ contributes $V[i] \cdot x$ towards the sum.

2.1.4 Tokenization

Tokenization is the problem of splitting an input text into tokens, where tokens are separated by a delimiter. For example, in a CSV (comma-separated-value) file, we can compute the fields of one row by tokenizing using commas as delimiters. Alternatively, we can compute the lines of a file by using newline characters as the delimiter.

In Figure 2.5, we implement a function `tokens` which takes two arguments: a function `isDelim` : `char` → `bool` and an input text t : `char array`. The function `isDelim` is a predicate which indicates whether or not a character should be considered a delimiter. The output of `tokens` is a `char array array`, where the ith element of the output is the ith token. Delimiters are omitted from the output.

The first step of the `tokens` function is to determine where the boundaries between tokens are. Specifically, an index i is a ***boundary*** if it is either the *start* or *end* of a token, defined as follows. For the purposes of these definitions, we pretend there are delimiters at $t[-1]$ and $t[n]$ where $n = |t|$; in the actual code, we handle these cases explicitly by checking for the cases $i = 0$ and $i = n$.

- Index i is the ***start*** of a token if $t[i-1]$ is a delimiter, and $t[i]$ is not a delimiter.

```
1  fun tokens(isDelim: char → bool, t: char array) : char array array =
2    let
3      val n = |t|
4
5      fun isBoundary(i) =
6        if i = n then not (isDelim(t[n − 1]))
7        else if i = 0 then not (isDelim(t[0]))
8        else
9          let
10           val d1 = isDelim(t[i − 1])
11           val d2 = isDelim(t[i])
12         in
13           (d1 andalso not d2) orelse (d2 andalso not d1)
14         end
15
16     val B = filter (0, n+1, fn i ⇒ i) isBoundary   // array of boundary indices
17     val m = Array.length(B)
18     val count = m/2   // note: m is even
19   in
20     // array of tokens
21     tabulate(0, count, fn i ⇒
22       // the i^th token starts at index B[2i] and ends at B[2i + 1]
23       tabulate(B[2i], B[2i+1], fn j ⇒ t[j])
24     )
25   end
```

Figure 2.5: Tokenization

- Index i is the ***end*** of a token if $t[i-1]$ is not a delimiter, and $t[i]$ is a delimiter.

To compute boundary indices, we implement a function `isBoundary` which tests whether or not index i is a boundary, and then perform a parallel filter (the implementation of which is described in more detail in Section 2.2). The filter considers indices $i \in [0, \ldots, n]$ (note: inclusive of n) and outputs an array B of all such indices i satisfying `isBoundary`(i).

There will always be an even number of boundary indices: for every start index of a token, there will also be an end index for the same token. Furthermore, because the output of the filter is stable, the start and end index of every token will be adjacent in the array of boundaries. In particular, the elements of the ith token lie between indices $B[2i]$ and $B[2i+1]$ where B is the array of boundaries. Therefore, to construct the output array of tokens, we can perform a nested tabulate. (The `tabulate` function, described in described in Section 2.2, allocates and initializes an array in parallel by evaluating a function at each index.)

2.2 Parallel Array Operations

In this section we develop data-parallel operations on arrays, including tabulate, scan (parallel prefix sums), filter, and flatten. These implementations utilize in-place updates for efficiency, but nevertheless have purely functional specifications.

```
fun for(i, j, f: int → unit) =
  if i ≥ j then () else (f(i); for(i+1, j, f))

fun parfor(i, j, f: int → unit) =
  if j − i ≤ GRAIN_THRESHOLD then
    for(i, j, f)
  else
    let val mid = ⌊(i + j)/2⌋
    in (parfor(i, mid, f) ‖ parfor(mid, j, f));
       ()
    end
```

Figure 2.6: Sequential and parallel for-loops.

```
fun tabulate(i, j, f: int → α) : α array =
  let
    val n = j − i
    val a = Array.allocate(n)
  in
    parfor(0, n, fn k ⇒ a[k] := f(i + k));
    a
  end
```

Figure 2.7: Parallel array tabulation

2.2.1 Tabulate

We first implement sequential and parallel for-loops (Figure 2.6), which take arguments (i, j, f), and evaluate the function f on each index between i and j. The expected use of these functions is to pass an effectful function f: `int → unit` as argument. Note that the parallel for-loop, `parfor`, reverts to a sequential loop below the granularity threshold.

Using a parallel for-loop, it is then straightforward to implement the `tabulate` function, where `tabulate`(i, j, f) produces the array $[f(i), f(i+1), \ldots, f(j-1)]$. This function simply allocates and array and then, in parallel for each index, evaluates f at that index and writes the result into the array at the appropriate position, as shown in Figure 2.7.

2.2.2 Scan (Parallel Prefix Sums)

We implement a block-based three-phase scan [47], which is illustrated in Figure 2.8. The first phase sums within the blocks. The second phase then does a scan on these partial sums. The results of the second phase are used as "offsets" for the third phase, which rereads the input along with the offsets to do a scan within each block, each starting from an offset.

An implementation in Parallel ML in shown in Figure 2.9. The function `scan` takes as argument an associative function $g : \alpha \times \alpha \to \alpha$ (such as binary addition, maximum, etc.) together with an identity z for that function. It then takes a triple $(\texttt{lo}, \texttt{hi}, f)$ which represents the data $[f(\texttt{lo}), \ldots, f(\texttt{hi} - 1)]$, and computes prefix sums with respect to g, outputting an array of length `hi-lo+1` where the ith output element is the prefix sum of the first i elements.

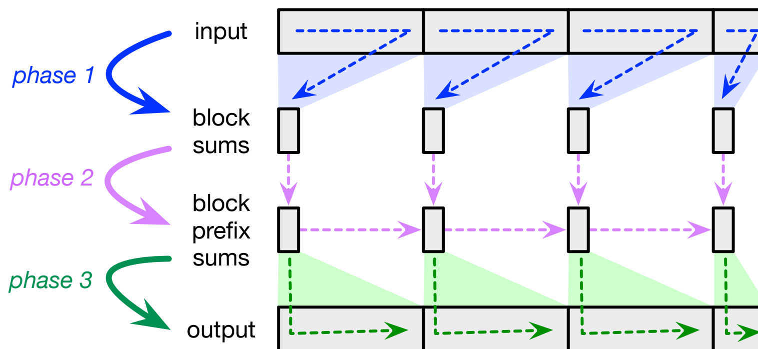

Figure 2.8: Three phases of scan

This is implemented in terms of previously discussed functions, including parallel for-loops (`parfor`), sequential folds (`foldl`), and array tabulations (`tabulate`). A constant `BLOCK_SIZE` is used for granularity control, where the input is broken up into many blocks, and sums are computed sequentially within each block, and in parallel across blocks.

2.2.3 Filter

The `filter` function takes a triple $(\texttt{lo}, \texttt{hi}, f)$ which represents the data $[f(\texttt{lo}), \ldots, f(\texttt{hi} - 1)]$, and an index-based predicate $p : \texttt{int} \rightarrow \texttt{bool}$ which indicates whether or not the element $f(i)$ should be kept. The output is an array containing elements at indices which satisfied the predicate.

Similar to `scan`, our `filter` implementation (Figure 2.10) is block-based, and consists of three phases. In the first phase, it begins by counting (in parallel across the blocks) the number of elements which satisfy the predicate within each block. Next, in a second phase, it computes prefix sums of the block-counts, which will be used as offsets for writing blocks in the output. Finally, in the third phase, the input is read again, and for each block starting at the appropriate offset, elements which satisfy the predicate are written to the output.

Alternative approaches. There are many ways a `filter` could be implemented. The advantage of this implementation is that has a low cost in terms of the amount of intermediate writes to memory: excluding the cost of writing the output to memory, this filter implementation only performs $O(N/B)$ memory writes where N is the number of input elements and B is the block size. In contrast, an alternative implementation might first map the predicate across the input, or might do a plus-scan across the entire input to calculate an offset for every input element individually. These alternatives would require $O(N)$ writes to memory, which is significantly larger than the $O(N/B)$ writes incurred by the implementation presented in Figure 2.10.

Impure and/or expensive predicate functions. The filter presented in Figure 2.10 calls the predicate p twice for each input element: once in the first phase, and again in the third phase. However, if p is an impure function (e.g., if it has a side-effect), then it could be unsafe

```
fun sequentialScan (g: α × α → α) (z: α) (lo, hi, f: int → α) : α array =
  let
    val n = hi - lo
    val R = Array.allocate(n + 1)
    fun bump((j, a), x) = (R[j] := a; (j + 1, g(a, x)))
    val (_, total) = foldl bump (0, z) (lo,hi,f)
  in
    R[n] := total;
    R
  end


fun scan (g: α × α → α) (z: α) (lo, hi, f: int → α) : α array =
  if hi - lo ≤ BLOCK_SIZE then
    // base case: sequential below the block-size threshold
    sequentialScan g z (lo,hi,f)
  else
    let
      val n = hi - lo
      val m = ⌈n/BLOCK_SIZE⌉   // number of blocks
      // Phase 1: compute block-sums in parallel
      val blockSums =
        tabulate(0, m, fn b ⇒
          let
            val start = lo + (b * BLOCK_SIZE)
            val stop = min(start + BLOCK_SIZE, hi)
          in
            foldl g z (start,stop,f)
          end)
      // Phase 2: recursively scan across block-sums
      val blockPrefixSums = scan g z (0, m, fn i ⇒ blockSums[i])
      val R = Array.allocate(n + 1)   // output array
    in
      // Phase 3: pass over blocks again, computing a sequential scan within each
      parfor(0, m, fn b ⇒
        let
          val start = lo + (b * BLOCK_SIZE)
          val stop = min(start + BLOCK_SIZE, hi)
          fun bump((j, a), x) = (R[j] := a; (j + 1, g(a, x)))
        in
          foldl bump (b * BLOCK_SIZE, blockPrefixSums[b]) (start,stop,f);
          ()
        end);
      R[n] := blockPrefixSums[m];
      R
    end
```

Figure 2.9: Parallel scan (prefix sums)

```
fun filter (lo, hi, f: int → α) (p: int → bool) : α array =
  let
    val n = hi - lo
    val m = ⌈n/BLOCK_SIZE⌉   // number of blocks
    // Phase 1: count the number of survivors in each block
    val counts =
      tabulate(0, m, fn b ⇒
        let
          val start = lo + (b * BLOCK_SIZE)
          val stop = min(start + BLOCK_SIZE, hi)
        in
          foldl (op+) 0 (start, stop, fn i ⇒ if p(i) then 1 else 0)
        end)
    // Phase 2: compute the offset of each block
    val offsets = scan (op+) 0 (0, m, fn i ⇒ counts[i])
    val R = Array.allocate(offsets[m]) // output array
  in
    // Phase 3: output the survivors of each block, starting at the appropriate offset
    parfor(0, m, fn b ⇒
      let
        val start = lo + (b * BLOCK_SIZE)
        val stop = min(start + BLOCK_SIZE, hi)
        fun bump(j,i) = if p(i) then (R[j] := f(i); j+1) else j
      in
        foldl bump (offsets[b]) (start, stop, fn i ⇒ i)
      end);
    R
  end
```

Figure 2.10: Parallel filter

to call p more than once per element. Alternatively, if p is an expensive function, it could be advantageous for performance to only call p once per element.

In either case, an alternative filtering procedure can be used which ensures that the predicate is evaluated exactly once per element. The idea is, in the first phase, instead of computing the counts for each block, instead, we perform a full sequential filter on each block, producing an intermediate state of type α `array array`, where the elements in the ith array contain the survivors of the ith block filter. This intermediate representation then needs to be "flattened" into an output array of type α `array`. (See the implementation of `flatten` in Section 2.2.4, below.) The number of intermediate writes performed by this alternative implementation of `filter` is $O(N/B+S)$ where N is the number of input elements, B is the block size, and S is the size of the output (i.e., the number of elements which satisfy the predicate).

2.2.4 Flatten

The `flatten` function takes as argument an α `array array` and in parallel concatenates the contents, producing an output of type α `array`. An implementation of `flatten` is shown in

```
fun flatten(A: α array array) : α array =
  let
    val n = |A|    // number of arrays being flattened
    val offsets = scan (op+) 0 (0, n, fn i ⇒ |A[i]|)
    val R = Array.allocate(offsets[n])
  in
    parfor(0, n, fn i ⇒
      parfor(0, |A[i]|, fn j ⇒
        R[offsets[i] + j] := A[i][j]
      )
    );
    R
  end
```

Figure 2.11: Parallel flatten

Figure 2.11. This implementation uses a plus-scan to determine the offsets for each array, and then uses nested parallel for-loops to write arrays to the output.

In practice, note however that this implementation of `flatten` does not necessarily have good load-balancing, as the arrays given as argument might differ wildly in size. To improve the efficiency of flatten, we can use a block-based strategy, where the input is broken into (logical) blocks of uniform size. In this approach, the start of each logical block can be computed by binary searching on the offsets.

2.2.5 Discussion: Fusion with Function Composition

The above examples operate on data of the form (i, j, f): `int` × `int` × (`int` → α) which can abstractly be thought of as representing a sequence $[f(i), f(i+1), \ldots, f(j-1)]$ with elements of type α. In comparison to representing sequences with arrays directly, this approach is advantageous because it allows us to achieve "fusion" automatically by relying on standard compiler optimizations [87, 155]. For example, in the `mcss` algorithm (Figure 2.3), the call to `reduce` takes a function (**fn** i ⇒ `singleton`($a[i]$)) as argument which fuses the calls to `singleton` into the `reduce` operation, avoiding the need to physically instantiate (in memory) the results of the `singleton` calls.

This is an instance of a more general functional programming technique known as *delayed sequences*, which efficiently supports a wide variety of standard operations (including maps, reduces, scans, filters, flattens, etc.) without generating unnecessary intermediate results [155]. In contrast to many existing techniques for fusion (e.g. [48, 52, 53, 82, 88, 101, 103, 147]), the delayed sequence technique is notable because it requires no special compiler support, and can be implemented entirely as a library using standard functional programming features such as higher-order functions and algebraic datatypes.

2.3 Non-deterministic Parallel Algorithms

Parallel programmers sometimes employ atomic operations such as compare-and-swap (and test-and-set, fetch-and-add, etc.) to improve efficiency. These techniques are typically non-deterministic: by interleaving atomic accesses and updates in shared memory, different interleavings in different executions (due to differences in scheduling) may result in different outcomes. Nevetheless, this non-determinism can be desirable from the programmer's perspective for improving performance.

2.3.1 Parallel BFS

As an example, consider a parallel breadth-first-search (BFS) graph traversal which uses atomic compare-and-swap operations, as shown in Figure 2.12.[1] Here, we use an abstract "sequence" library, provided by a module named `Seq`, supporting standard parallel operations such as tabulate, map, flatten, filter, reduce, etc. (whose implementations are discussed in Section 2.2).

The BFS consists of a series of rounds, where each round visits some of the vertices of the graph. Vertices are visited by setting their "parent" in the array P. In particular, when an edge (u, v) is traversed, we set $P[v]$ to u. Initially, the parent of every vertex is set to -1 (meaning: not yet visited).

The function `bfsRound` implements one round of BFS. It takes as input a "frontier" F which contains all vertices visited on the previous round. It then calls `edgeMap`, which is implemented in terms of standard parallel functions on sequences.[2] The call to `edgeMap` performs many calls to `tryVisit`(u, v) in parallel, one for each edge (u, v) where $u \in F$. The function `tryVisit`(u, v) attempts to visit v by atomically changing the value of $P[v]$ from -1 to u. If this succeeds, `tryVisit` returns `true`; if it fails (because v has already been visited), it returns `false`. The output of the `edgeMap` is the collection of vertices v such that `tryVisit`(u, v) succeeded. In this way, the `edgeMap` simultaneously accomplishes two things: (1) it visits vertices in parallel by setting their parents, and (2) it returns the set of vertices visited on this round (i.e., the next frontier).

This code has a determinacy race: on a single round, there might be two edges (u_1, v) and (u_2, v) that both have the same target vertex v. The two corresponding calls `tryVisit`(u_1, v) and `tryVisit`(u_2, v) will race to visit v, and only one will succeed, resulting in a non-deterministic choice between u_1 and u_2 as the parent of v. This non-determinism is desirable from a performance perspective, because it enables the algorithm to quickly "deduplicate" edges without having to first write all these edges out to an intermediate data structure.

Deterministic parent selection. If desired, the output of this BFS can be made deterministic by utilizing priority updates [134]. The idea is to compute, for each visited vertex, some deterministic selection of parent from amongst all possible parents. Here, we compute the maximum vertex identifier. To do so, the only function that needs to change is the `tryVisit` function. The new function, called `priorityTryVisit` (Figure 2.13) updates the parent of a vertex in two cases:

[1] This code is inspired by the Ligra graph framework [131].

[2] Note that this call to `edgeMap` uses `filter` with an impure predicate, which requires a different implementation for `filter` than the one presented in Figure 2.10. This nuance is discussed in Section 2.2.3.

```
1  structure Seq:
2  sig
3    type α t
4    type α seq = α t
5    val singleton: α → α seq
6    val map: (α → β) → α seq → β seq
7    val filter: (α → bool) → α seq → α seq
8    val flatten: α seq seq → α seq
9    ...
10 end
11
12 type graph
13 type vertex = int   // vertices labeled 0 to N-1
14 val numberOfVertices: graph → int
15 val neighbors: graph × vertex → vertex Seq.t   // out-neighbors of a vertex
16
17 // Do f(u,v) in parallel for every out-edge (u,v) where u ∈ S.
18 // Returns all vertices v where f(u,v) is true.
19 fun edgeMap(G: graph,
20             S: vertex Seq.t,
21             f: vertex × vertex → bool): vertex Seq.t =
22   Seq.flatten
23     (Seq.map (fn u ⇒ Seq.filter (fn v ⇒ f(u,v)) (neighbors(G,u))) S)
24
25 // breadth-first-search of G starting at vertex s, returning the array of parents
26 fun bfs(G: graph, s: vertex) : vertex array =
27   let
28     val N = numberOfVertices(G)
29     val P: vertex array = tabulate(0, N, fn _ ⇒ -1) // "parents" array
30
31     // Try to visit v by atomically changing P[v] from -1 to u.
32     // Returns true if success, or false if already visited.
33     fun tryVisit(u,v): bool = Array.compareAndSwap(P,v,-1,u)
34
35     fun bfsRound(F: vertex Seq.t): vertex Seq.t = edgeMap(G, F, tryVisit)
36
37     // main BFS loop, on current frontier F
38     fun loop(F) = if Seq.length(F) = 0 then () else loop(bfsRound(G,F))
39   in
40     tryVisit(s,s);            // mark s as visited, using self as parent
41     loop(Seq.singleton(s));   // BFS starting from s
42     P                         // return parents as result
43   end
```

Figure 2.12: BFS in Parallel ML. This algorithm is non-deterministic.

```
1  fun priorityTryVisit(u,v): bool =
2    let
3      val p = P[v]     // read the current parent of p
4      val isFirstVisit = (p = −1)
5    in
6      if u ≤ p then
7        false
8      else if Array.compareAndSwap(P, v, p, u) then
9        isFirstVisit   // even if the parent is updated many times, make sure
10                      // to visit v exactly once
11     else
12       priorityTryVisit(u,v)   // If u > p but the compareAndSwap fails, then some
13                               // other call to tryVisit must have succeeded by updating
14                               // the parent. Therefore, we need to try again.
15   end
```

Figure 2.13: Alternative implementation of `tryVisit` for parallel BFS, based on priority updates. The resulting parents computed for each vertex are deterministic.

it it hasn't been visited already, and also if a "larger" parent is found. To ensure that each visited vertex is included exactly once in the output of the `edgeMap`, we only return `true` in the case of a successful update which replaces an old parent value of -1.

By selecting parents deterministically, the final output of the BFS (the final parents array) will be the same on every execution, regardless of scheduling. However, the algorithm is still internally non-deterministic, in the sense that individual memory updates may be different on different executions. In particular, the total number of calls to `tryVisit` may differ between executions, depending on scheduling and the resulting contention between `compareAndSwap` operations. (In contrast, with non-deterministic selection of parent as shown in Figure 2.12, the number of calls to `tryVisit` is the same on every execution, but the final result is non-deterministic.)

2.3.2 Parallel Deduplication by Concurrent Hashing

Figure 2.14 implements a function `dedup` which deduplicates an input array X containing elements of type α. The algorithm here is based on concurrent hashing; therefore, in order to perform deduplication, we need additional inputs which specify how to hash an element of type α, how to test whether two elements are equal, etc. Specifically, these additional arguments are a hash function `hash`: $\alpha \to$ `int`, an equality test `eq`: $\alpha \times \alpha \to$ `bool`, and an "empty slot" value `empty`: α.[3] By passing these additional arguments, the `dedup` function is able to be polymorphic over type α.

[3]The advantage of the "empty slot" value (i.e., in comparison to using option types), is that it is typically more efficient when an appropriate value can be chosen. For example, if deduplicating a set of non-negative integers, the value -1 can be used to indicate an empty slot. More generally, any value of type α which does not appear in the input is a valid choice for the empty-slot value. If no suitable empty-slot value of type α can be chosen, then the input can be converted to use an option type: each element x can be mapped to `SOME(x)`, allowing `NONE` to be used as the empty slot.

```
1  // Deduplicate X by concurrent hashing. The first three arguments (hash, eq, and empty) are
2  // used to specify the hashing process:
3  //   – hash: a hash function for elements of type α.
4  //   – eq: an equality test for elements of type α.
5  //   – empty: an "empty slot" value.
6  fun dedup {hash: α → int, eq: α × α → bool, empty: α}
7            (X: α array) : α array =
8    let
9      val capacity = ⌈|X| · 10/9⌉   // In worst case (no duplicates), ensures at most 90% load
10     val data = tabulate(0, capacity, fn _ ⇒ empty)   // allocate hash-set
11
12     fun tryPut(x, i) =
13       eq(data[i],empty) andalso Array.compareAndSwap(data,i,empty,x)
14
15     fun insertLoop(x, i) =
16       if tryPut(x, i) then ()             // done: successful insert
17       else if eq(data[i],x) then ()       // done: x is a duplicate
18       else insertLoop(x, i + 1 mod |data|) // try next slot; wrap around if needed
19
20     fun insert(x) = insertLoop(x, hash(x) mod |data|)
21   in
22     parfor(0, |X|, fn i ⇒ insert(X[i]));   // batch insert
23     // finally, compact to remove empty slots, and return deduplicated elems
24     filter (0, |data|, fn i ⇒ data[i]) (fn i ⇒ not(eq(data[i], empty)))
25   end
```

Figure 2.14: Parallel deduplication by concurrent hashing

At a high level, the `dedup` function has three phases. First, it allocates an empty hash-set of sufficient capacity. Next, it inserts individual elements in parallel. Finally, it compacts the hash-set (using the `filter` function of Section 2.2.3) to remove any remaining empty slots.

The hashing process used here is a simple open addressing scheme with linear probing. The hash-set is of type α `array`, where initially, all slots of the hash-set contain the "empty slot" value. To insert an element x, the function `insertLoop` begins at index $i = \texttt{hash}(x) \textbf{ mod } |\texttt{data}|$, and then walks left-to-right (with wrap-around) looking for an empty slot. When an empty slot is found, we perform a compare-and-swap operation to attempt to atomically fill the slot with x. If this succeeds, then the insertion is finished; otherwise, if it fails, then the slot has been taken by another insertion, and the linear probing process must continue, to find another slot. Along the way, for every non-empty slot, we check if x is a duplicate, i.e., we check if another value x' has already been inserted which satisfies $\texttt{eq}(x, x')$.

The output of this function is non-deterministic, because the relative order of inserted elements is not enforced. For example, if two distinct elements x and y both hash to the same index, then in different executions, x and y may appear in different orders in the output. Furthermore, amongst a group of equal elements, this function does not guarantee which one is "chosen" by the deduplication.

Deterministic output. If desired, the output of this function can be made deterministic using the phase-concurrent hashing algorithm of Shun and Blelloch [132]. The high-level idea of their algorithm is to specify a priority on elements,[4] and the algorithm ensures that elements appear in priority order in the output (essentially by performing a small "insertion sort" during the hash-insertion loop). Internally, this algorithm is non-deterministic, because the relative timing of insertions is not enforced. Nevertheless, the output is guaranteed to be deterministic: the Shun-Blelloch algorithm produces the same elements, in the same order, on every execution.

[4]Specifying a priority on elements is straightforward: for example, the indices of elements in the input can be used, with priority given to smaller indices. The output would then match the result of a sequential hashing algorithm, which inserts input elements one-by-one, from left to right. In other situations, a different priority function might be more efficient, or could be used to ensure specific properties on the output.

Chapter 3

Disentanglement

In this chapter, we formalize the disentanglement property, which informally is the observation that **concurrent tasks often remain oblivious to each other's allocations**. Our formalization is based on a graph-based semantics embedded in a small ML-like core language. We then prove that disentanglement emerges naturally due to determinism and race-freedom: in particular, all *determinacy-race-free* programs are guaranteed to be disentangled (Theorem 1). We also discuss how disentanglement is more general than race-freedom, allowing for non-deterministic in-place updates and accesses in a variety of useful circumstances (including, for example, the parallel BFS example of Section 2.3).

3.1 Language and Graph Semantics

We consider a simple fork-join (nested-parallel) language that fully accounts for all memory operations by explicitly allocating memory for *all* data, mutable and immutable alike. To define disentanglement, during execution, the language constructs an execution trace called a *computation graph.* A computation graph records the history of a computation in terms of actions that are performed upon a shared memory and a partial order on these actions, which captures the structure of parallelism. The generality of computation graphs makes them suitable for defining both disentanglement and determinacy-race-freedom (Section 3.3).

Typically, a big-step semantics might be used to construct computation graphs. However, since our computational model permits both parallelism and side-effects, the semantics must account for fine-grained interleaving of (concurrent) computations. We therefore use a small-step semantics, which, due to possible interleavings of parallel steps that can affect a shared memory, is non-deterministic. In order to construct computation graphs in a small-step manner, we define *open computation graphs* (Section 3.1.3) which encode the structure of active parallel tasks, allowing each small step to extend the computation graph "at the right place".

Note that the small language considered here does not statically or dynamically enforce any guarantees of disentanglement and/or race-freedom. This is intentional, allowing us to define these properties as behaviors of execution that are not necessarily exhibited by all programs.

Variables	x, f		
Numbers	n	$\in$	$\mathbb{N}$
Memory Locations	ℓ		
Types	τ	$::=$	$\mathsf{nat} \mid \tau \times \tau \mid \tau \to \tau \mid \tau\ \mathsf{ref}$
Storables	s	$::=$	$n \mid \mathsf{fun}\ f\ x\ \mathsf{is}\ e \mid \langle \ell, \ell \rangle \mid \mathsf{ref}\ \ell$
Expressions	e	$::=$	$\ell \mid s \mid x \mid e\ e \mid \langle e, e \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{ref}\ e \mid !\,e \mid e := e \mid \langle e \parallel e \rangle$
Memory	μ	$\in$	*Locations* $\rightharpoonup$ *Storables*
Actions	α	$::=$	$\mathbf{A}\ell \Leftarrow s \mid \mathbf{R}\ell \Rightarrow s \mid \mathbf{W}\ell \Leftarrow s$
Computation Graphs	g	$::=$	$\bullet \mid \alpha \mid g \oplus g \mid g \otimes g$
Open Computation Graphs	G	$::=$	$[g] \mid g \oplus (G \otimes G)$

Figure 3.1: Syntax

3.1.1 Syntax

Figure 5.2 shows the syntax of the language.

Types. The types include a base type of natural numbers, as well as products (tuples/pairs), functions, and mutable references.

Memory Locations and Storables. To account precisely for memory operations, the language distinguishes between *storables* s, which are stored in memory, and *memory locations* ℓ. Storables consist of natural numbers, named recursive functions, pairs of memory locations, and mutable references to other memory locations. Locations are the only irreducible form of the language; that is, all terminating expressions eventually step to a memory location.

Expressions. Expressions consist of memory locations, storables, variables and applications, pairs and their projections, mutable references with explicit lookup and update, and the parallel pair $\langle e_1 \parallel e_2 \rangle$, which is used to execute e_1 and e_2 in parallel. For convenience we will use the syntactic sugar ($\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$) to mean ($\mathsf{fun}\ f\ x\ \mathsf{is}\ e_2$) e_1, where f does not appear free in e_2.

Memory. A separate memory μ is used to map locations to storables. We write $\mathrm{dom}(\mu)$ for the set of locations mapped by μ, $\mu(\ell)$ to look up the storable associated with ℓ, and $\mu[\ell \hookrightarrow s]$ to extend μ with a new mapping (with the implicit requirement that $\ell \notin \mathrm{dom}(\mu)$).

3.1.2 Computation Graphs and Actions

Traditionally, a nested parallel computation is represented by using a directed acyclic graph, or ***dag***, that consists of vertices and edges. Each vertex represents an executed instruction and each edge represents the control dependency between two instructions. We augment the dag by annotating every vertex with the ***action*** it performed upon shared memory. Actions, denoted α, can be one of the following:

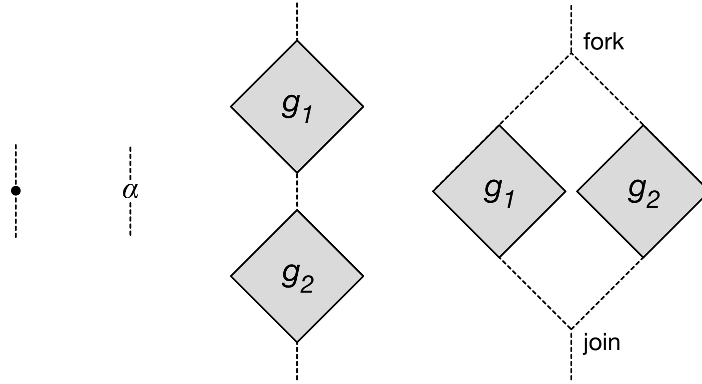

Figure 3.2: A series-parallel computation graph is either the empty graph •, a single action α, a sequential composition $g_1 \oplus g_2$, or a parallel composition $g_1 \otimes g_2$. The dashed lines are control dependencies, implicitly pointing down.

- $\mathbf{A}\ell \Leftarrow s$ is the allocation of location ℓ, initialized with contents s.
- $\mathbf{R}\ell \Rightarrow s$ is a read (lookup) at ℓ which returned s.
- $\mathbf{W}\ell \Leftarrow s$ is a write (update) at ℓ which stored s.

We call the dag augmented with actions a ***computation graph***. Due to the structure of nested (fork-join) parallelism, computation graphs have a *series-parallel* structure, as depicted in Figure 3.2. In particular, a computation graph can be any one of the following.

- The empty (no-op) graph, denoted •.
- A single action α.
- The sequential composition of graphs g_1 and g_2, denoted $g_1 \oplus g_2$, where there is an edge connecting the last vertex of g_1 to the first of g_2, indicating that all of g_1 happened before g_2.
- The parallel composition of graphs g_1 and g_2, denoted $g_1 \otimes g_2$, indicating that neither g_1 nor g_2 happened before the other. In this case there are two special vertices which arise: a *fork* and a corresponding *join*. The fork, which has out-degree two, is connected to each of the first vertices of g_1 and g_2. The join has in-degree two, and its incoming neighbors are the last vertices of g_1 and g_2.

3.1.3 Open Computation Graphs

As described thus far, computation graphs g can be understood as representing the history of "completed" computations. However, while a computation is in progress, we need a way of constructing its computation graph one step at a time. To do so, we exploit a specific structure dictated by the nesting of parallel tasks.

At every moment during execution, the tasks of a nested-parallel program can be organized into a tree structure, called a ***task tree***, where each node represents a task. Each task in the tree has either exactly two children (its subtasks) or no children. A characteristic feature of nested parallelism is that, when a task forks two subtasks, the task suspends its own execution

until both subtasks complete. Therefore in the task tree, each internal task is suspended, and the leaves are "active" tasks that may step in parallel. When both children of an internal task terminate, the corresponding leaves disappear from the tree and the internal task becomes a leaf, resuming its execution.

Each time a leaf task takes a step, it may perform an action that needs to be recorded in the computation graph. To locate where in the computation graph this new action should go, we partition the computation graph into many smaller computation graphs and organize them in a tree structure mirroring the task tree. We call this tree structure an ***open computation graph***, denoted G. In an open computation graph, each node records the local history of its corresponding task in the task tree.

There are two possible forms for an open computation graph G, corresponding to leaves and internal tasks, respectively.

$$
\begin{array}{rcll}
G & ::= & [g] & \text{leaf task} \\
& | & g \oplus (G_1 \otimes G_2) & \text{(suspended) internal task}
\end{array}
$$

Leaf tasks have the form $[g]$: each leaf is a computation that may be extended with a new action when the corresponding task takes a step (e.g., a step from $[g]$ to $[g \oplus (\mathbf{A}\ell \Leftarrow s)]$). The internal nodes of an open computation graph have the form $g \oplus (G_1 \otimes G_2)$ where g is the history of the corresponding task up until the moment it forked two subtasks, and G_1 and G_2 are the open computation graphs of its subtasks.

3.1.4 Operational Semantics

The operational semantics, defined in Figures 3.3 and 3.4, is a single-step relation

$$\mu \,;\, G \,;\, e \longmapsto \mu' \,;\, G' \,;\, e'.$$

Each step takes a memory μ, an open computation graph G, and an expression e and produces a new state consisting of μ', G', and e'. Steps are non-deterministic due to possible interleavings of rules ParL and ParR.

Allocation. The allocation rule Alloc is the only way to create new memory locations. It steps a storable s to a fresh location ℓ, extends the memory by mapping ℓ to s, and records $\mathbf{A}\ell \Leftarrow s$ in the computation graph.

Reading from Memory. There are four rules which read from memory: function application (rule App), pair projection (rules Fst and Snd), and reference lookup (rule Bang). The semantics does *not* distinguish between reads of mutable and immutable data. In rule App, the function at location ℓ_1 is applied to the argument at location ℓ_2. This is accomplished by reading from ℓ_1 (to acquire the source code of the function) and substituting both ℓ_1 and ℓ_2 into the function body e_b. Note that rule App performs a read at ℓ_1 but not at ℓ_2.

Writing to Memory. The rule Upd updates the storable at ℓ_1 to refer to ℓ_2. This is the only way the contents of an existing memory location can change during execution.

Execution $\boxed{\mu \,;\, G \,;\, e \longmapsto \mu' \,;\, G' \,;\, e'}$

$$\frac{\ell \notin \mathrm{dom}(\mu)}{\mu \,;\, [g] \,;\, s \longmapsto \mu[\ell \hookrightarrow s] \,;\, [g \oplus (\mathbf{A}\ell \Leftarrow s)] \,;\, \ell}\ \textsc{Alloc}$$

$$\frac{\mu(\ell_1) = \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b}{\mu \,;\, [g] \,;\, (\ell_1\ \ell_2) \longmapsto \mu \,;\, [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b)] \,;\, [\ell_1, \ell_2 \,/\, f, x]e_b}\ \textsc{App}$$

$$\frac{\mu(\ell) = \langle \ell_1, \ell_2 \rangle}{\mu \,;\, [g] \,;\, (\mathsf{fst}\ \ell) \longmapsto \mu \,;\, [g \oplus (\mathbf{R}\ell \Rightarrow \langle \ell_1, \ell_2 \rangle)] \,;\, \ell_1}\ \textsc{Fst}$$

$$\frac{\mu(\ell) = \langle \ell_1, \ell_2 \rangle}{\mu \,;\, [g] \,;\, (\mathsf{snd}\ \ell) \longmapsto \mu \,;\, [g \oplus (\mathbf{R}\ell \Rightarrow \langle \ell_1, \ell_2 \rangle)] \,;\, \ell_2}\ \textsc{Snd}$$

$$\frac{\mu(\ell_1) = \mathsf{ref}\ \ell_2}{\mu \,;\, [g] \,;\, (!\,\ell_1) \longmapsto \mu \,;\, [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2)] \,;\, \ell_2}\ \textsc{Bang}$$

$$\frac{}{\mu[\ell_1 \hookrightarrow \mathsf{ref}\ _] \,;\, [g] \,;\, (\ell_1 := \ell_2) \longmapsto \mu[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2] \,;\, [g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)] \,;\, \ell_2}\ \textsc{Upd}$$

$$\frac{}{\mu \,;\, [g] \,;\, \langle e_1 \parallel e_2 \rangle \longmapsto \mu \,;\, g \oplus ([\bullet] \otimes [\bullet]) \,;\, \langle e_1 \parallel e_2 \rangle}\ \textsc{Fork}$$

$$\frac{}{\mu \,;\, g \oplus ([g_1] \otimes [g_2]) \,;\, \langle \ell_1 \parallel \ell_2 \rangle \longmapsto \mu \,;\, [g \oplus (g_1 \otimes g_2)] \,;\, \langle \ell_1, \ell_2 \rangle}\ \textsc{Join}$$

$$\frac{\mu \,;\, G_1 \,;\, e_1 \longmapsto \mu' \,;\, G_1' \,;\, e_1'}{\mu \,;\, g \oplus (G_1 \otimes G_2) \,;\, \langle e_1 \parallel e_2 \rangle \longmapsto \mu' \,;\, g \oplus (G_1' \otimes G_2) \,;\, \langle e_1' \parallel e_2 \rangle}\ \textsc{ParL}$$

$$\frac{\mu \,;\, G_2 \,;\, e_2 \longmapsto \mu' \,;\, G_2' \,;\, e_2'}{\mu \,;\, g \oplus (G_1 \otimes G_2) \,;\, \langle e_1 \parallel e_2 \rangle \longmapsto \mu' \,;\, g \oplus (G_1 \otimes G_2') \,;\, \langle e_1 \parallel e_2' \rangle}\ \textsc{ParR}$$

Figure 3.3: Language dynamics (main computation steps).

Execution (cont.) $\boxed{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}$

$$\frac{\mu\,;G\,;e_1 \longmapsto \mu'\,;G'\,;e_1'}{\mu\,;G\,;(e_1\ e_2) \longmapsto \mu'\,;G'\,;(e_1'\ e_2)}\ \textsc{AppSL} \qquad \frac{\mu\,;G\,;e_2 \longmapsto \mu'\,;G'\,;e_2'}{\mu\,;G\,;(\ell_1\ e_2) \longmapsto \mu'\,;G'\,;(\ell_1\ e_2')}\ \textsc{AppSR}$$

$$\frac{\mu\,;G\,;e_1 \longmapsto \mu'\,;G'\,;e_1'}{\mu\,;G\,;\langle e_1, e_2\rangle \longmapsto \mu'\,;G'\,;\langle e_1', e_2\rangle}\ \textsc{PairSL} \qquad \frac{\mu\,;G\,;e_2 \longmapsto \mu'\,;G'\,;e_2'}{\mu\,;G\,;\langle \ell_1, e_2\rangle \longmapsto \mu'\,;G'\,;\langle \ell_1, e_2'\rangle}\ \textsc{PairSR}$$

$$\frac{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}{\mu\,;G\,;(\mathsf{fst}\ e) \longmapsto \mu'\,;G'\,;(\mathsf{fst}\ e')}\ \textsc{FstS} \qquad \frac{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}{\mu\,;G\,;(\mathsf{snd}\ e) \longmapsto \mu'\,;G'\,;(\mathsf{snd}\ e')}\ \textsc{SndS}$$

$$\frac{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}{\mu\,;G\,;(\mathsf{ref}\ e) \longmapsto \mu'\,;G'\,;(\mathsf{ref}\ e')}\ \textsc{RefS} \qquad \frac{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}{\mu\,;G\,;(!\,e) \longmapsto \mu'\,;G'\,;(!\,e')}\ \textsc{BangS}$$

$$\frac{\mu\,;G\,;e_1 \longmapsto \mu'\,;G'\,;e_1'}{\mu\,;G\,;(e_1 := e_2) \longmapsto \mu'\,;G'\,;(e_1' := e_2)}\ \textsc{UpdSL} \qquad \frac{\mu\,;G\,;e_2 \longmapsto \mu'\,;G'\,;e_2'}{\mu\,;G\,;(\ell_1 := e_2) \longmapsto \mu'\,;G'\,;(\ell_1 := e_2')}\ \textsc{UpdSR}$$

Figure 3.4: Language dynamics (additional administrative rules).

Parallelism. Parallelism is accomplished through four rules: forking new tasks (rule Fork), joining completed tasks (rule Join), and subtask stepping (rules ParL and ParR). The Fork rule records the beginning of two new parallel tasks in the computation graph. When two subtasks have completed, rule Join assembles their results as a standard pair and records that the tasks have completed in the computation graph. The ParL and ParR rules non-deterministically interleave steps of the subtasks, recording their actions in the appropriate subgraph. Note that the shape of the open computation graph determines whether a parallel pair forks, evaluates the subtasks, or joins.

3.2 Example: Transposing Points in 2D

Consider a function `transpose`, shown in Figure 3.5 using an ML-like syntax, that takes an array of points in 2D space and transposes each point in parallel by swapping its x- and y-coordinates. For this example, we assume that the language has arrays, which are natural extensions of mutable references (whereas a `ref` is a single mutable location, an array is a sequence of many mutable locations). The function relies on a recursive function `tr` that takes two indices specifying a segment of the input array. If the segment has size 1 then the function allocates a fresh point whose coordinates are the derived from the x- and y-coordinates of the sole element in the segment.[1] Otherwise, the function splits the segment in the middle into two segments, and

[1]A more realistic implementation would control granularity by reverting to a sequential transpose below a threshold size.

```
type point = int × int

fun transpose
  (P: point array)
  (n: int) =
let
  fun tr i j =
    if j − i = 1 then
      let val (x, y) = P[i]
      in P[i] := (y, x)
      end
    else
      let
        val mid = ⌊(i + j)/2⌋
      in
        (tr i mid || tr mid j)
      end
in
  tr 0 n
end
```

Figure 3.5: The function `transpose` transposes each element in array P of length n.

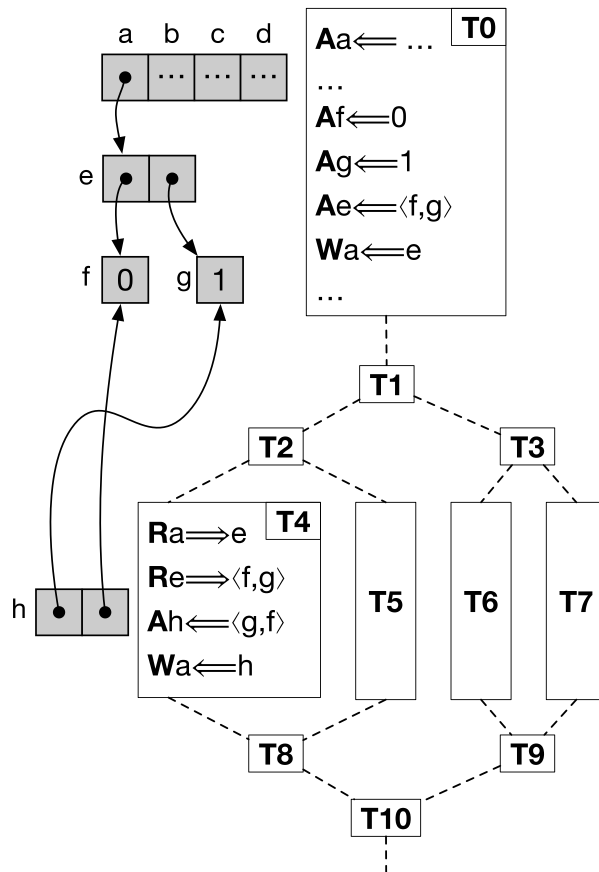

Figure 3.6: Computation graph for `transpose` on an input array of length 4.

transposes them recursively in parallel. Because this function performs constant work for each and every element of the array, the `transpose` function requires asymptotically linear work in n. Its span—the longest chain of dependencies—is logarithmic (in n). The function therefore exposes significant parallelism.

Figure 3.6 summarizes the computation graph for an execution of `transpose` with `n = 4` elements. The diagram uses a single vertex to represent entire tasks (sequential regions), and dashed lines to indicate control dependencies between tasks. All dashed lines implicitly point down. The gray square boxes represent memory objects and are labeled with their memory locations, and the solid arrows are pointers in memory. The input array consists of memory locations {a, b, c, d}, each of which points to a pair of locations which in turn point to integers. We assume that the root task **T0** allocates and initializes the input array and calls `transpose`, the root of which is the task **T1**. Task **T1** then forks two subtasks **T2** and **T3**, which in turn each fork two more (**T4**-**T7**). The tasks **T4**, **T5**, **T6**, and **T7** perform the steps of reading from the array, allocating new tuples with x- and y-coordinates swapped, and writing these tuples back into the array. We show the specific actions of **T4** and omit the actions of tasks **T5**-**T7**, which are all similar to **T4**. As depicted, the pointers show the state of memory *before* the write in **T4** occurs; after the write, location **a** should point to **h**. When the tasks **T4**, **T5**, **T6**, and **T7** all complete, they join "back up" with tasks **T8**, **T9**, and **T10**, at which point the computation is complete.

3.3 Definition of Disentanglement

To define disentanglement, we look more closely at the actions in the computation graph and define two notions—knowledge and use—where we say that actions *know* locations and *use* locations. An action ***knows*** a location ℓ if ℓ was allocated by the action itself or by an ancestor in the computation graph. An action ***uses*** a location ℓ if ℓ is being accessed by the action or ℓ is part of the storable being allocated, written, or read by the action. Specifically, the actions $\mathbf{A}\ell \Leftarrow s$, $\mathbf{W}\ell \Leftarrow s$, and $\mathbf{R}\ell \Rightarrow s$ each *use* exactly the locations $\ell \cup \mathsf{locs}(s)$. (The function $\mathsf{locs}(e)$, defined in Figure 3.7, is the set of locations mentioned by expression e.) We can then define disentanglement as the property that *every action uses only locations that it knows.*

Definition. The formal definition of disentanglement is given in Figure 3.8 as a judgement $A \vdash g\ \mathsf{de}$, which establishes that computation graph g is disentangled with respect to known allocations A. The judgement $A \vdash G\ \mathsf{de}$ similarly establishes disentanglement on open computation graphs G. Both judgements are given in terms of two auxiliary functions (defined in Figure 3.7): $\mathsf{A}(g)$ is the set of locations allocated by g, and $\mathsf{locs}(e)$ is the set of locations mentioned by expression e.

The rules establish for every action that all locations used by that action are known. For reads $\mathbf{R}\ell \Rightarrow s$ and writes $\mathbf{W}\ell \Leftarrow s$, this is verified by checking that ℓ and $\mathsf{locs}(s)$ are among the known allocations A. For allocation actions $\mathbf{A}\ell \Leftarrow s$, the rules only need to establish that $\mathsf{locs}(s)$ are among the known allocations, since ℓ is allocated by this action and therefore is certainly known. The set of known allocations A accumulates at sequential compositions $g_1 \oplus g_2$, allowing g_2 to know all allocations of g_1. Similarly, in open computation graphs $g \oplus (G_1 \otimes G_2)$, all

Expression Roots (Mentioned Locations)

$$\mathsf{locs}(\ell) \triangleq \{\ell\}$$
$$\mathsf{locs}(n) \triangleq \emptyset$$
$$\mathsf{locs}(\mathsf{fun}\ f\ x\ \mathsf{is}\ e) \triangleq \mathsf{locs}(\mathsf{fst}\ e) \triangleq \mathsf{locs}(\mathsf{snd}\ e) \triangleq \mathsf{locs}(\mathsf{ref}\ e) \triangleq \mathsf{locs}(!\,e) \triangleq \mathsf{locs}(e)$$
$$\mathsf{locs}(e_1\ e_2) \triangleq \mathsf{locs}(\langle e_1, e_2\rangle) \triangleq \mathsf{locs}(e_1 := e_2) \triangleq \mathsf{locs}(\langle e_1 \parallel e_2\rangle) \triangleq \mathsf{locs}(e_1) \cup \mathsf{locs}(e_2)$$

Allocations

$$\begin{aligned}
\mathsf{A}(\bullet) &\triangleq \emptyset \\
\mathsf{A}(\mathbf{A}\ell \Leftarrow s) &\triangleq \{\ell\} \\
\mathsf{A}(\mathbf{R}\ell \Rightarrow s) &\triangleq \emptyset \\
\mathsf{A}(\mathbf{W}\ell \Leftarrow s) &\triangleq \emptyset \\
\mathsf{A}(g_1 \oplus g_2) &\triangleq \mathsf{A}(g_1) \cup \mathsf{A}(g_2) \\
\mathsf{A}(g_1 \otimes g_2) &\triangleq \mathsf{A}(g_1) \cup \mathsf{A}(g_2) \\
\mathsf{A}([g]) &\triangleq \mathsf{A}(g) \\
\mathsf{A}(g \oplus (G_1 \otimes G_2)) &\triangleq \mathsf{A}(g) \cup \mathsf{A}(G_1) \\
&\quad \cup \mathsf{A}(G_2)
\end{aligned}$$

Allocations and Writes

$$\begin{aligned}
\mathsf{AW}(\bullet) &\triangleq \emptyset \\
\mathsf{AW}(\mathbf{A}\ell \Leftarrow s) &\triangleq \{\ell\} \\
\mathsf{AW}(\mathbf{R}\ell \Rightarrow s) &\triangleq \emptyset \\
\mathsf{AW}(\mathbf{W}\ell \Leftarrow s) &\triangleq \{\ell\} \\
\mathsf{AW}(g_1 \oplus g_2) &\triangleq \mathsf{AW}(g_1) \cup \mathsf{AW}(g_2) \\
\mathsf{AW}(g_1 \otimes g_2) &\triangleq \mathsf{AW}(g_1) \cup \mathsf{AW}(g_2) \\
\mathsf{AW}([g]) &\triangleq \mathsf{AW}(g) \\
\mathsf{AW}(g \oplus (G_1 \otimes G_2)) &\triangleq \mathsf{AW}(g) \cup \mathsf{AW}(G_1) \\
&\quad \cup \mathsf{AW}(G_2)
\end{aligned}$$

Figure 3.7: Auxiliary Definitions: expression roots, and graph allocations and writes.

allocations of g_1 are known to G_1 and G_2. Crucially, in parallel compositions, the two subgraphs do not know of each other's allocations.

Example. Returning to the `tranpose` example of Section 3.2, we can see that this computation is disentangled, as each action in Figure 3.6 only uses locations that were allocated by itself or ancestors.

3.4 Disentanglement and Race-Freedom

In this section, we show that disentanglement is guaranteed when a computation is free of a certain kind of race condition called a *determinacy race*. A ***determinacy race*** occurs when two concurrent actions both atomically access the same location, and at least one of these accesses modifies the location [63]. Determinacy races are race conditions on individual memory locations which emerge due to concurrent interleaving of atomic operations such as compare-and-swap, test-and-set, etc., as well as atomic loads and stores.[2] As the name suggests, the lack

[2]Determinacy races are distinct from *data races*. Within the context of a language memory model, a *data race* can be defined as two conflicting concurrent accesses which are not properly synchronized (e.g. "ordinary" loads and stores, which the semantics of a language may allow to be reordered), leading to incorrect behavior due to problems such as miscompilation or lack of atomicity [10, 11, 41, 42]. In contrast, the *determinacy races* we consider here are properly synchronized, i.e., well-defined within the language memory model. For example,

$\boxed{A \vdash g \text{ de}}$

$$\frac{}{A \vdash \bullet \text{ de}} \qquad \frac{\text{locs}(s) \subseteq A}{A \vdash (\mathbf{A}\ell \Leftarrow s) \text{ de}} \qquad \frac{\ell \in A \qquad \text{locs}(s) \subseteq A}{A \vdash (\mathbf{R}\ell \Rightarrow s) \text{ de}} \qquad \frac{\ell \in A \qquad \text{locs}(s) \subseteq A}{A \vdash (\mathbf{W}\ell \Leftarrow s) \text{ de}}$$

$$\frac{A \vdash g_1 \text{ de} \qquad A \uplus \mathsf{A}(g_1) \vdash g_2 \text{ de}}{A \vdash g_1 \oplus g_2 \text{ de}} \qquad \frac{A \vdash g_1 \text{ de} \qquad A \vdash g_2 \text{ de}}{A \vdash g_1 \otimes g_2 \text{ de}}$$

$\boxed{A \vdash G \text{ de}}$

$$\frac{A \vdash g \text{ de}}{A \vdash [g] \text{ de}} \qquad \frac{A \vdash g \text{ de} \qquad A \uplus \mathsf{A}(g) \vdash G_1 \text{ de} \qquad A \uplus \mathsf{A}(g) \vdash G_2 \text{ de}}{A \vdash g \oplus (G_1 \otimes G_2) \text{ de}}$$

Figure 3.8: Definition of disentanglement. Variable A denotes the set of known allocations.

of determinacy races is sufficient to guarantee determinism [60, 146].[3]

Determinacy-race-freedom. A computation with no determinacy races is ***determinacy-race-free*** (DRF). We can determine whether or not a computation is DRF by inspecting its computation graph. Specifically, we need to verify that for every pair of concurrent actions α_1 and α_2, which access the same location, that they are both read actions. A location is *accessed* when its contents are either inspected or modified. Specifically, each of $\mathbf{A}\ell \Leftarrow s$, $\mathbf{R}\ell \Rightarrow s$, and $\mathbf{W}\ell \Leftarrow s$ are considered to *access* location ℓ. The actions which modify a location are writes and allocations: both of $\mathbf{A}\ell \Leftarrow s$ and $\mathbf{W}\ell \Leftarrow s$ modify location ℓ. We treat allocations as modifications because allocations also initialize the location, which in an implementation requires a write to the location.

With this setup, we can formally define determinacy-race-freedom on computation graphs with a judgement $F \vdash g \text{ drf}$ which establishes that computation graph g is DRF with respect to a "forbidden" set of locations F. The definition is given in Figure 3.9, together with a corresponding judgement $F \vdash G \text{ drf}$ for open computation graphs G. These are defined in terms of another auxiliary function (defined in Figure 3.7): $\mathsf{AW}(g)$ is the set of locations allocated and written by g.

To see how the forbidden set F is used in the definition, consider the case for parallel composition. In order for $g_1 \otimes g_2$ to be DRF, we need to verify that every location modified by g_2 is not accessed by g_1, and vice-versa. We capture this constraint by extending the set of forbidden locations for g_1 with the allocated and written locations of g_2 (and vice-versa). Then at each individual action, we only need to verify that the accessed location is not forbidden. Note that

two threads simultaneously attempting to acquire the same lock constitutes a determinacy race. Another example is two concurrent threads racing to atomically test-and-set a shared flag. In both cases, the program may be non-deterministic, but its behavior is nevertheless well-defined. The language semantics of Section 3.1 allows for non-determinism in this manner, as all memory loads and stores are handled atomically. Our language semantics could easily be extended with other atomic operations, including compare-and-swap, test-and-set, fetch-and-add, etc.

[3] Assuming no other sources of non-determinism such as randomness.

$\boxed{F \vdash g \text{ drf}}$

$$\frac{}{F \vdash \bullet \text{ drf}} \qquad \frac{\ell \notin F}{F \vdash (\mathbf{A}\ell \Leftarrow s) \text{ drf}} \qquad \frac{\ell \notin F}{F \vdash (\mathbf{W}\ell \Leftarrow s) \text{ drf}} \qquad \frac{\ell \notin F}{F \vdash (\mathbf{R}\ell \Rightarrow s) \text{ drf}}$$

$$\frac{F \vdash g_1 \text{ drf} \qquad F \vdash g_2 \text{ drf}}{F \vdash g_1 \oplus g_2 \text{ drf}} \qquad \frac{F \cup \mathsf{AW}(g_2) \vdash g_1 \text{ drf} \qquad F \cup \mathsf{AW}(g_1) \vdash g_2 \text{ drf}}{F \vdash g_1 \otimes g_2 \text{ drf}}$$

$\boxed{F \vdash G \text{ drf}}$

$$\frac{F \vdash g \text{ drf}}{F \vdash [g] \text{ drf}} \qquad \frac{F \vdash g \text{ drf} \qquad F \cup \mathsf{AW}(G_2) \vdash G_1 \text{ drf} \qquad F \cup \mathsf{AW}(G_1) \vdash G_2 \text{ drf}}{F \vdash g \oplus (G_1 \otimes G_2) \text{ drf}}$$

Figure 3.9: Definition of determinacy-race-freedom. Variable F denotes a "forbidden" set of locations (that are allocated or updated by a concurrent task).

we do not accumulate forbidden locations in sequential compositions $g_1 \oplus g_2$, because in these cases we know that g_1 and g_2 did not happen concurrently.

Determinacy races can violate disentanglement. Intuitively, races can violate disentanglement, because two tasks can communicate by concurrently reading and writing at a shared memory location. For example, consider the program 'let x = ref 0 in $\langle x := 1 \parallel !\,x \rangle$'. This program allocates a shared location ℓ for the ref, and then spawns two subtasks. In one possible execution, the left-hand subtask gets to run completely before the right-hand subtask executes. In this case, the left-hand task allocates a location ℓ' for the value 1 and then writes a pointer to ℓ' at shared location ℓ. Next, the right-hand task executes, reading from ℓ and discovering ℓ', which violates disentanglement. In this situation, there was a determinacy race at ℓ.

Although races can violate disentanglement, it is possible to avoid this issue by preallocating any data that might possibly be shared amongst concurrent tasks. That is, we could rewrite the example program as 'let x = ref 0 in let y = 1 in $\langle x := y \parallel !\,x \rangle$', which is racy (non-deterministic) and yet disentangled. For more details about how to utilize determinacy races in disentangled programs, see Section 3.5.

Determinacy-race-freedom preserves disentanglement. When determinacy races are disallowed, disentanglement is guaranteed, because shared memory locations cannot be used to communicate pointers to freshly allocated locations. Theorem 1 establishes this result. The theorem states that if at any moment we pause a program and observe that it has (so far) been free of determinacy races, then the program also has been disentangled.

Theorem 1 (DRF $\Rightarrow$ DE)**.** For any $\emptyset\,;[\bullet]\,;e_0 \longmapsto^* \mu\,;G\,;e$ where $\mathsf{locs}(e_0) = \emptyset$, if $\emptyset \vdash G$ drf, then $\emptyset \vdash G$ de.

The full proof this theorem is presented below, in Section 3.4.1. At a high level, the proof relies on a single-step invariant which (roughly speaking) captures the the following property: "for every leaf task and for every location ℓ known by that task, if ℓ is not forbidden by DRF,

$$\boxed{A\,;F \vdash_\mu G\,; e \ \mathsf{drfde}}$$

$$\frac{A \vdash g \ \mathsf{de} \qquad \mathsf{locs}(e) \subseteq A \uplus \mathsf{A}(g) \qquad \begin{array}{c} F \vdash g \ \mathsf{drf} \\ \forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F.\, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \end{array}}{A\,; F \vdash_\mu [g]\,; e \ \mathsf{drfde}}$$

$$\frac{F \vdash g \ \mathsf{drf} \qquad A \vdash g \ \mathsf{de} \qquad \begin{array}{c} A \uplus \mathsf{A}(g)\,; F \cup \mathsf{AW}(G_2) \vdash_\mu G_1\,; e_1 \ \mathsf{drfde} \\ A \uplus \mathsf{A}(g)\,; F \cup \mathsf{AW}(G_1) \vdash_\mu G_2\,; e_2 \ \mathsf{drfde} \end{array}}{F\,; A \vdash_\mu g \oplus (G_1 \otimes G_2)\,; \langle e_1 \parallel e_2 \rangle \ \mathsf{drfde}}$$

$$\frac{A\,; F \vdash_\mu g \oplus (G_1 \otimes G_2)\,; e \ \mathsf{drfde}}{A\,; F \vdash_\mu g \oplus (G_1 \otimes G_2)\,; (\mathsf{fst}\ e) \ \mathsf{drfde}} \quad \textit{...similarly for } (\mathsf{snd}\ e),\ (\mathsf{ref}\ e),\ \textit{and } (!\,e)$$

$$\frac{\begin{array}{c} \neg(e_1 \ \mathsf{loc}) \\ \mathsf{locs}(e_2) \subseteq A \uplus \mathsf{A}(g) \end{array} \qquad A\,; F \vdash_\mu g \oplus (G_1 \otimes G_2)\,; e_1 \ \mathsf{drfde}}{A\,; F \vdash_\mu g \oplus (G_1 \otimes G_2)\,; (e_1\ e_2) \ \mathsf{drfde}}$$

...similarly for $\langle e_1, e_2 \rangle$ *and* $(e_1 := e_2)$

$$\frac{\ell_1 \in A \uplus \mathsf{A}(g) \qquad A\,; F \vdash_\mu g \oplus (G_1 \otimes G_2)\,; e_2 \ \mathsf{drfde}}{A\,; F \vdash_\mu g \oplus (G_1 \otimes G_2)\,; (\ell_1\ e_2) \ \mathsf{drfde}}$$

Figure 3.10: Strengthening of disentanglement with the guarantees of simultaneous determinacy-race-freedom.

then each $\ell' \in \mathsf{locs}(\mu(\ell))$ is known by that task." We present a judgement $A\,;F \vdash_\mu G\,;e\ \mathsf{drfde}$, defined in Figure 3.10, which states this property formally. The drfde judgement is also strong enough to imply both $F \vdash G\ \mathsf{drf}$ and $A \vdash G\ \mathsf{de}$. Initially, all of these properties hold of an initial state (i.e., for $\mu_0 = \emptyset$, $G_0 = [\bullet]$, and e_0). We prove Lemma 6, the single-step result that, given $A\,;F \vdash_\mu G\,;e\ \mathsf{drfde}$, if a step is taken to μ', G', and e' where $F \vdash G'\ \mathsf{drf}$, then $A\,;F \vdash_{\mu'} G'\,;e'\ \mathsf{drfde}$. The theorem follows by induction on the derivation of the $\longmapsto^*$ judgement.

Examples. All of the functions of Section 2.1 and Section 2.2 are determinacy-race-free and therefore disentangled.

3.4.1 Proof: Race-Freedom Preserves Disentanglement

Lemma 1. If $\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'$, then either
- $G = [g]$ and $G' = [g']$ or
- $G = [g]$ and $G' = g \oplus ([\bullet] \otimes [\bullet])$ or
- $G = g \oplus ([g_1] \otimes [g_2])$ and $G' = [g \oplus (g_1 \otimes g_2)]$ or
- $G = g \oplus (G_1 \otimes G_2)$ and $G' = g \oplus (G_1' \otimes G_2)$ or
- $G = g \oplus (G_1 \otimes G_2)$ and $G' = g \oplus (G_1 \otimes G_2')$.

Proof. By induction on the derivation of $\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'$. □

Lemma 2. If $\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'$,
then $\mathsf{A}(G) \subseteq \mathsf{A}(G')$ and $\mathsf{AW}(G) \subseteq \mathsf{AW}(G')$ and $\forall \ell \in \mathsf{dom}(\mu) \setminus (\mathsf{AW}(G') \setminus \mathsf{AW}(G)), \mu(\ell) = \mu'(\ell)$.

Proof. By induction on the derivation of $\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'$. □

Lemma 3. If $F \vdash G\ \mathsf{drf}$ then $F \cap \mathsf{AW}(G) = \emptyset$.

Proof. By induction on the derivation of $F \vdash G\ \mathsf{drf}$. □

Lemma 4. If $A\,;F \vdash_\mu G\,;e\ \mathsf{drfde}$ then $F \vdash G\ \mathsf{drf}$ and $A \vdash G\ \mathsf{de}$.

Proof. By induction on the derivation of $A\,;F \vdash_\mu G\,;e\ \mathsf{drfde}$. □

Lemma 5. If $A\,;F \vdash_\mu G\,;e\ \mathsf{drfde}$ and $F \subseteq F'$ and $\forall \ell \in \mathsf{dom}(\mu) \setminus F', \mu(\ell) = \mu'(\ell)$,
then $A\,;F' \vdash_{\mu'} G\,;e\ \mathsf{drfde}$

Proof. By induction on the derivation of $A\,;F \vdash_\mu G\,;e\ \mathsf{drfde}$. □

Lemma 6. For any $\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'$, if $A\,;F \vdash_\mu G\,;e\ \mathsf{drfde}$ and $F \vdash G'\ \mathsf{drf}$,
then $A\,;F \vdash_{\mu'} G'\,;e'\ \mathsf{drfde}$.

Proof. By induction on the derivation of $\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'$.

Case Fork. We have $G = [g]$ and $e = \langle e_1 \parallel e_2 \rangle$ and $\mu' = \mu$ and $G' = g \oplus ([\bullet] \otimes [\bullet])$ and $e' = \langle e_1 \parallel e_2 \rangle$. Assume $A\,;F \vdash_\mu [g]\,;\langle e_1 \parallel e_2 \rangle\ \mathsf{drfde}$ and $F \vdash g \oplus ([\bullet] \otimes [\bullet])\ \mathsf{drf}$. By inversion of $A\,;F \vdash_\mu [g]\,;\langle e_1 \parallel e_2 \rangle\ \mathsf{drfde}$, we have $F \vdash g\ \mathsf{drf}$ and $A \vdash g\ \mathsf{de}$ and $\mathsf{locs}(\langle e_1 \parallel e_2 \rangle) \subseteq A \uplus \mathsf{A}(g)$ and $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$. Note that $A \uplus \mathsf{A}(g) \uplus \mathsf{A}(\bullet)$ is defined and equal to $A \uplus \mathsf{A}(g)$. We show $A\,;F \vdash_\mu g \oplus ([\bullet] \otimes [\bullet])\,;\langle e_1 \parallel e_2 \rangle\ \mathsf{drfde}$, by showing:

- $F \vdash g$ drf, which was established above
- $A \vdash g$ de, which was established above
- $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}([\bullet]) \vdash_\mu [\bullet] ; e_1$ drfde, by showing:
  - $F \cup \mathsf{AW}([\bullet]) \vdash [\bullet]$ drf, by $[-]$ and $\bullet$ rules for drf
  - $A \uplus \mathsf{A}(g) \vdash [\bullet]$ de, by $[-]$ and $\bullet$ rules for de
  - $\mathsf{locs}(e_1) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}(\bullet)$, by $\mathsf{locs}(e_1) \subseteq \mathsf{locs}(\langle e_1 \| e_2 \rangle)$ and $\mathsf{locs}(\langle e_1 \| e_2 \rangle) \subseteq A \uplus \mathsf{A}(g)$ (established above) and $A \uplus \mathsf{A}(g) = A \uplus \mathsf{A}(g) \uplus \mathsf{A}(\bullet)$
  - $\forall \ell \in (A \uplus \mathsf{A}(g) \uplus \mathsf{A}(\bullet)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \bullet$, by $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$ (established above) and $A \uplus \mathsf{A}(g) = A \uplus \mathsf{A}(g) \uplus \mathsf{A}(\bullet)$
- $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}([\bullet]) \vdash_\mu [\bullet] ; e_2$ drfde, which can be established similarly to $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}([\bullet]) \vdash_\mu [\bullet] ; e_1$ drfde.

Case Join. We have $G = g \oplus ([g_1] \otimes [g_2])$ and $e = \langle \ell_1 \| \ell_2 \rangle$ and $\mu' = \mu$ and $G' = [g \oplus (g_1 \otimes g_2)]$ and $e' = \langle \ell_1, \ell_2 \rangle$. Assume $A ; F \vdash_\mu g \oplus ([g_1] \otimes [g_2]) ; \langle \ell_1 \| \ell_2 \rangle$ drfde and $F \vdash [g \oplus (g_1 \otimes g_2)]$ drf. By inversion of $A ; F \vdash_\mu g \oplus ([g_1] \otimes [g_2]) ; \langle \ell_1 \| \ell_2 \rangle$ drfde, we have $F \vdash g$ drf and $A \vdash g$ de and $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}([g_2]) \vdash_\mu [g_1] ; \ell_1$ drfde and $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}([g_1]) \vdash_\mu [g_2] ; \ell_2$ drfde. By inversion of $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}([g_2]) \vdash_\mu [g_1] ; \ell_1$ drfde, we have $F \cup \mathsf{AW}([g_2]) \vdash [g_1]$ drf and $A \uplus \mathsf{A}(g) \vdash [g_1]$ de and $\mathsf{locs}(\ell_1) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])$ and $\forall \ell \in (A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])) \setminus (F \cup \mathsf{AW}([g_2])), \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])$; note that A, $\mathsf{A}(g)$, and $\mathsf{A}(g_1)$ are mutually disjoint. By Lemma 3 with $F \cup \mathsf{AW}([g_2]) \vdash [g_1]$ drf, we have $(F \cup \mathsf{AW}([g_2])) \cap \mathsf{AW}([g_1]) = \emptyset$, which implies that $\mathsf{AW}(g_2) \cap \mathsf{AW}(g_1) = \emptyset$ and $\mathsf{A}(g_2) \cap \mathsf{A}(g_1) = \emptyset$. By inversion of $A \uplus \mathsf{A}(g) \vdash [g_1]$ de, we have $A \uplus \mathsf{A}(g) \vdash g_1$ de. By inversion of $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}([g_1]) \vdash_\mu [g_2] ; \ell_2$ drfde, we have $F \cup \mathsf{AW}([g_1]) \vdash [g_2]$ drf and $A \uplus \mathsf{A}(g) \vdash [g_2]$ de and $\mathsf{locs}(\ell_2) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])$ and $\forall \ell \in (A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])) \setminus (F \cup \mathsf{AW}([g_1])), \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])$; note that A, $\mathsf{A}(g)$, and $\mathsf{A}(g_2)$ are mutually disjoint. By Lemma 3 with $F \cup \mathsf{AW}([g_1]) \vdash [g_2]$ drf, we have $(F \cup \mathsf{AW}([g_1])) \cap \mathsf{AW}([g_2]) = \emptyset$, which implies that $\mathsf{AW}(g_1) \cap \mathsf{AW}(g_2) = \emptyset$ and $\mathsf{A}(g_1) \cap \mathsf{A}(g_2) = \emptyset$. By inversion of $A \uplus \mathsf{A}(g) \vdash [g_2]$ de, we have $A \uplus \mathsf{A}(g) \vdash g_2$ de. Recall that A, $\mathsf{A}(g)$, $\mathsf{A}(g_1)$, and $\mathsf{A}(g_2)$ are mutually disjoint. Therefore $A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$ is defined and equal to $A \uplus \mathsf{A}(g) \uplus \mathsf{A}(g_1) \uplus \mathsf{A}(g_2)$. We show $A ; F \vdash_\mu [g \oplus (g_1 \otimes g_2)] ; \langle \ell_1, \ell_2 \rangle$ drfde, by showing:

- $F \vdash g \oplus (g_1 \otimes g_2)$ drf, by inversion of $F \vdash [g \oplus (g_1 \otimes g_2)]$ drf (which was assumed)
- $A \vdash g \oplus (g_1 \otimes g_2)$ de, by $- \oplus -$ and $- \otimes -$ rules for de and by showing
  - $A \vdash g$ de, which was established above
  - $A \uplus \mathsf{A}(g) \vdash g_1$ de, which was established above
  - $A \uplus \mathsf{A}(g) \vdash g_2$ de, which was established above
- $\mathsf{locs}(\langle \ell_1, \ell_2 \rangle) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$, by $\mathsf{locs}(\ell_1) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])$ (established above) and $A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1]) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$ and $\mathsf{locs}(\ell_2) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])$ (established above) and $A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2]) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$
- $\forall \ell \in (A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$: Let $\ell \in (A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])) \setminus F$. Because $A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)]) = A \uplus \mathsf{A}(g) \uplus \mathsf{A}(g_1) \uplus \mathsf{A}(g_2)$, we can consider three (mutually exclusive) cases:

- $\ell \in (A \uplus \mathsf{A}(g)) \setminus F$: Recall that $\mathsf{AW}(g_1)$ and $\mathsf{AW}(g_2)$ are disjoint. Therefore, we can consider two (not mutually exclusive) cases:
  - $\ell \notin \mathsf{AW}(g_1)$: By $\forall \ell \in (A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])) \setminus (F \cup \mathsf{AW}([g_1])), \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])$ (established above) with $\ell \in (A \uplus \mathsf{A}(g)) \setminus F$ and $\ell \notin \mathsf{AW}(g_1)$, we have $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])$. Furthermore, $A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2]) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$.
  - $\ell \notin \mathsf{AW}(g_2)$: By $\forall \ell \in (A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])) \setminus (F \cup \mathsf{AW}([g_2])), \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])$ (established above) with $\ell \in (A \uplus \mathsf{A}(g)) \setminus F$ and $\ell \notin \mathsf{AW}(g_2)$, we have $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])$. Furthermore, $A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1]) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$.
- $\ell \in \mathsf{A}(g_1) \setminus F$: By $\forall \ell \in (A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])) \setminus (F \cup \mathsf{AW}([g_2])), \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])$ (established above) with $\ell \in \mathsf{A}(g_1) \setminus F$ and $\mathsf{AW}(g_2) \cap \mathsf{A}(g_1) = \emptyset$ (established above), we have $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1])$. Furthermore, $A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_1]) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$.
- $\ell \in \mathsf{A}(g_2) \setminus F$: By $\forall \ell \in (A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])) \setminus (F \cup \mathsf{AW}([g_1])), \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])$ (established above) with $\ell \in \mathsf{A}(g_2) \setminus F$ and $\mathsf{AW}(g_1) \cap \mathsf{A}(g_2) = \emptyset$ (established above), we have $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2])$. Furthermore, $A \uplus \mathsf{A}(g) \uplus \mathsf{A}([g_2]) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$.

Case ParL. We have $G = g \oplus (G_1 \otimes G_2)$ and $e = \langle e_1 \parallel e_2 \rangle$ and $\mu ; G_1 ; e_1 \longmapsto \mu' ; G_1' ; e_1'$ and $G' = g \oplus (G_1' \otimes G_2)$ and $e' = \langle e_1' \parallel e_2 \rangle$. Assume $A ; F \vdash_\mu g \oplus (G_1 \otimes G_2) ; \langle e_1 \parallel e_2 \rangle \text{ drfde}$ and $F \vdash g \oplus (G_1' \otimes G_2) \text{ drf}$. By inversion of $A ; F \vdash_\mu g \oplus (G_1 \otimes G_2) ; \langle e_1 \parallel e_2 \rangle \text{ drfde}$ we have $F \vdash g \text{ drf}$ and $A \vdash g \text{ de}$ and $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}(G_2) \vdash_\mu G_1 ; e_1 \text{ drfde}$ and $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}(G_1) \vdash_\mu G_2 ; e_2 \text{ drfde}$. By inversion of $F \vdash g \oplus (G_1' \otimes G_2) \text{ drf}$, we have $F \vdash g \text{ drf}$ and $F \cup \mathsf{AW}(G_2) \vdash G_1 \text{ drf}$ and $F \cup \mathsf{AW}(G_1) \vdash G_2 \text{ drf}$. By the induction hypothesis using $\mu ; G_1 ; e_1 \longmapsto \mu' ; G_1' ; e_1'$ with $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}(G_2) \vdash_\mu G_1 ; e \text{ drfde}_1$ and $F \cup \mathsf{AW}(G_2) \vdash G_1 \text{ drf}$ (established above), we have $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}(G_2) \vdash_{\mu'} G_1' ; e_1' \text{ drfde}$. We show $A ; F \vdash_{\mu'} g \oplus (G_1' \otimes G_2) ; \langle e_1' \parallel e_2 \rangle \text{ drfde}$ by showing:

- $F \vdash g \text{ drf}$, which was established above
- $A \vdash g \text{ de}$, which was established above
- $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}(G_2) \vdash_{\mu'} G_1' ; e_1' \text{ drfde}$, which was established above
- $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}(G_1') \vdash_{\mu'} G_2 ; e_2 \text{ drfde}$: By Lemma 2 with $\mu ; G_1 ; e_1 \longmapsto \mu' ; G_1' ; e_1'$, we have $\mathsf{A}(G_1) \subseteq \mathsf{A}(G_1')$ and $\mathsf{AW}(G_1) \subseteq \mathsf{AW}(G_1')$ and $\forall \ell \in \mathsf{dom}(\mu) \setminus (\mathsf{AW}(G_1') \setminus \mathsf{AW}(G_1)), \mu(\ell) = \mu'(\ell)$. By Lemma 5 with $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}(G_1) \vdash_\mu G_2 ; e_2 \text{ drfde}$ and $F \cup \mathsf{AW}(G_1) \subseteq F \cup \mathsf{AW}(G_1')$ and $\forall \ell \in \mathsf{dom}(\mu) \setminus (F \cup \mathsf{AW}(G_1')), \mu(\ell) = \mu'(\ell)$ (noting $\mathsf{dom}(\mu) \setminus (F \cup \mathsf{AW}(G_1')) \subseteq \mathsf{dom}(\mu) \setminus (\mathsf{AW}(G_1') \setminus \mathsf{AW}(G_1)))$ we have $A \uplus \mathsf{A}(g) ; F \cup \mathsf{AW}(G_1') \vdash_{\mu'} G_2 ; e_2 \text{ drfde}$.

Case ParR: Similar to ParL.

Case Alloc. We have $G = [g]$ and $e = s$ and $\ell' \notin \mathsf{dom}(\mu)$ and $\mu' = \mu[\ell' \hookrightarrow s]$ and $G' = [g \oplus (\mathbf{A}\ell' \Leftarrow s)]$ and $e' = \ell'$. Assume $A ; F \vdash_\mu [g] ; s \text{ drfde}$ and $F \vdash [g \oplus (\mathbf{A}\ell' \Leftarrow s)] \text{ drf}$. By inversion of $A ; F \vdash_\mu [g] ; s \text{ drfde}$, we have $F \vdash g \text{ drf}$ and $A \vdash g \text{ de}$ and $\mathsf{locs}(s) \subseteq A \uplus \mathsf{A}(g)$ and $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$. By Lemma 3 with $F \vdash g \oplus (\mathbf{A}\ell' \Leftarrow s) \text{ drf}$, we

have $F \cap \mathsf{AW}(g \oplus (\mathbf{A}\ell' \Leftarrow s)) = \emptyset$, which implies that $\ell' \notin F$. Furthermore, $\ell' \notin A \uplus \mathsf{A}(g)$, because if $\ell \in A \uplus \mathsf{A}(g)$, then by $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$ (established above) with $\ell \in A \uplus \mathsf{A}(g)$, we would have $\mu(\ell')$ defined, but $\ell' \notin \mathsf{dom}(\mu)$. Therefore, $A \uplus \mathsf{A}(g \oplus (\mathbf{A}\ell' \Leftarrow s))$ is defined and equal to $A \uplus \mathsf{A}(g) \uplus \{\ell'\}$. We show $A ; F \vdash_{\mu[\ell' \hookrightarrow s]} [g \oplus (\mathbf{A}\ell' \Leftarrow s)] ; \ell'$ drfde, by showing:

- $F \vdash g \oplus (\mathbf{A}\ell' \Leftarrow s)$ drf, by inversion of $F \vdash [g \oplus (\mathbf{A}\ell' \Leftarrow s)]$ drf (which was assumed)
- $A \vdash g \oplus (\mathbf{A}\ell' \Leftarrow s)$ de, by $- \oplus -$ and $(\mathbf{A}- \Leftarrow -)$ rules for de and by showing
  - $A \vdash g$ de, which was established above
  - $\mathsf{locs}(s) \subseteq A \uplus \mathsf{A}(g)$, which was established above
- $\mathsf{locs}(\ell') \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{A}\ell' \Leftarrow s))$, by $\mathsf{locs}(\ell') = \{\ell'\}$ and $A \uplus \mathsf{A}(g \oplus (\mathbf{A}\ell' \Leftarrow s)) = A \uplus \mathsf{A}(g) \uplus \{\ell'\}$
- $\forall \ell \in (A \uplus \mathsf{A}(g \oplus (\mathbf{A}\ell' \Leftarrow s))) \setminus F, \mathsf{locs}(\mu[\ell' \hookrightarrow s](\ell)) \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{A}\ell' \Leftarrow s))$:
  Let $\ell' \in (A \uplus \mathsf{A}(g \oplus (\mathbf{A}\ell' \Leftarrow s))) \setminus F$. We can consider two (mutually exclusive) cases:
  - $\ell = \ell'$: By $\mathsf{locs}(\mu[\ell' \hookrightarrow s](\ell)) = s$ (because $\ell = \ell'$) and $\mathsf{locs}(s) \subseteq A \uplus \mathsf{A}(g)$ (established above), we have $\mathsf{locs}(\mu[\ell' \hookrightarrow s](\ell)) \subseteq A \uplus \mathsf{A}(g)$. Futhermore, $A \uplus \mathsf{A}(g) \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{A}\ell \Leftarrow' s))$.
  - $\ell \neq \ell'$: By $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$ (established above) with $\ell \in (A \uplus \mathsf{A}(g \oplus (\mathbf{A}\ell \Leftarrow' s))) \setminus F = (A \uplus \mathsf{A}(g) \uplus \{\ell'\}) \setminus F$ and $\ell \neq \ell'$, we have $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$. Futhermore, $A \uplus \mathsf{A}(g) \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{A}\ell' \Leftarrow s))$.

Case Upd. We have $\mu = \mu_0[\ell_1 \hookrightarrow s]$ and $G = [g]$ and $e = \ell_1 := \ell_2$ and $\mu' = \mu_0[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2]$ and $G' = [g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)]$ and $e' = \ell_2$. Assume $A ; F \vdash_{\mu_0[\ell_1 \hookrightarrow s]} [g] ; \ell_1 := \ell_2$ drfde and $F \vdash [g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)]$ drf. By inversion of $A ; F \vdash_{\mu_0[\ell_1 \hookrightarrow s]} [g] ; \ell_1 := \ell_2$ drfde, we have $F \vdash g$ drf and $A \vdash g$ de and $\mathsf{locs}(\ell_1 := \ell_2) \subseteq A \uplus \mathsf{A}(g)$ and $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu_0[\ell_1 \hookrightarrow s](\ell)) \subseteq A \uplus \mathsf{A}(g)$. Note that $A \uplus \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2))$ is defined and equal to $A \uplus \mathsf{A}(g)$. We show $A ; F \vdash_{\mu_0[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2]} [g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)] ; \ell_2$ drfde, by showing:

- $F \vdash g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)$ drf, by inversion of $F \vdash [g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)]$ drf (which was assumed)
- $A \vdash g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)$ de, by $- \oplus -$ and $(\mathbf{W}- \Leftarrow -)$ rules for de and by showing
  - $A \vdash g$ de, which was established above
  - $\ell_1 \in A \uplus \mathsf{A}(g)$, by $\ell_1 \in \mathsf{locs}(\ell_1 := \ell_2)$ and $\mathsf{locs}(\ell_1 := \ell_2) \subseteq A \uplus \mathsf{A}(g)$ (established above)
  - $\mathsf{locs}(\mathsf{ref}\ \ell_2) \subseteq A \uplus \mathsf{A}(g)$, by $\mathsf{locs}(\mathsf{ref}\ \ell_2) \subseteq \mathsf{locs}(\ell_1 := \ell_2)$ and $\mathsf{locs}(\ell_1 := \ell_2) \subseteq A \uplus \mathsf{A}(g)$ (established above)
- $\mathsf{locs}(\ell_2) \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2))$, by $\mathsf{locs}(\ell_2) \subseteq \mathsf{locs}(\ell_1 := \ell_2)$ and $\mathsf{locs}(\ell_1 := \ell_2) \subseteq A \uplus \mathsf{A}(g)$ (established above) and $A \uplus \mathsf{A}(g) = A \uplus \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2))$
- $\forall \ell \in (A \uplus \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2))) \setminus F, \mathsf{locs}(\mu_0[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2](\ell)) \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2))$:
  Let $\ell \in \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)) \setminus F$. We can consider two (mutually exclusive) cases:
  - $\ell = \ell_1$: By $\mathsf{locs}(\mu_0[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2](\ell)) = \mathsf{locs}(\mathsf{ref}\ \ell_2)$ (because $\ell = \ell_1$) and $\mathsf{locs}(\mathsf{ref}\ \ell_2) \subseteq \mathsf{locs}(\ell_1 := \ell_2)$ and $\mathsf{locs}(\ell_1 := \ell_2) \subseteq A \uplus \mathsf{A}(g)$ (established above), we have $\mathsf{locs}(\mu_0[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2](\ell)) \subseteq A \uplus \mathsf{A}(g)$. Futhermore, $A \uplus \mathsf{A}(g) = A \uplus \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2))$.

  - $\ell \neq \ell_1$: By $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu_0[\ell_1 \hookrightarrow s](\ell)) \subseteq A \uplus \mathsf{A}(g)$ (established above) with $\ell \in (A \uplus \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2))) \setminus F$ and $A \uplus \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)) = A \uplus \mathsf{A}(g)$, we have $\mathsf{locs}(\mu_0[\ell_1 \hookrightarrow s](\ell)) \subseteq A \uplus \mathsf{A}(g)$. From $\mathsf{locs}(\mu_0[\ell_1 \hookrightarrow s](\ell)) \subseteq A \uplus \mathsf{A}(g)$ and $\mu_0[\ell_1 \hookrightarrow s](\ell) = \mu_0[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2](\ell)$ (because $\ell \neq \ell_1$), we have $\mathsf{locs}(\mu_0[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2](\ell)) \subseteq A \uplus \mathsf{A}(g)$. Futhermore, $A \uplus \mathsf{A}(g) = A \uplus \mathsf{A}(g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2))$.

Case Bang. We have $G = [g]$ and $e = !\,\ell_1$ and $\mu(\ell_1) = \mathsf{ref}\ \ell_2$ and $\mu' = \mu$ and $G' = [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2)]$ and $e' = \ell_2$. Assume $A\,;F \vdash_\mu [g]\,;!\,\ell_1$ drfde and $F \vdash [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2)]$ drf. By inversion of $A\,;F \vdash_\mu [g]\,;!\,\ell_1$ drfde, we have $F \vdash g$ drf and $A \vdash g$ de and $\mathsf{locs}(!\,\ell_1) \subseteq A \uplus \mathsf{A}(g)$ and $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$. Note that $A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2))$ is defined and equal to $A \uplus \mathsf{A}(g)$. We show $A\,;F \vdash_\mu [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2)]\,;\ell_2$ drfde, by showing:

- $F \vdash g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2)$ drf, by inversion of $F \vdash [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2)]$ drf (which was assumed)
- $A \vdash g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2)$ de, by $- \oplus -$ and $(\mathbf{R}- \Rightarrow -)$ rules for de and by showing
  - $A \vdash g$ de, which was established above
  - $\ell_1 \in A \uplus \mathsf{A}(g)$, by $\ell_1 \in \mathsf{locs}(!\,\ell_1)$ and $\mathsf{locs}(!\,\ell_1) \subseteq A \uplus \mathsf{A}(g)$ (established above)
  - $\mathsf{locs}(\mathsf{ref}\ \ell_2) \subseteq A \uplus \mathsf{A}(g)$: By $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$ (established above) with $\ell_1 \in A \uplus \mathsf{A}(g)$ (just established), we have $\mathsf{locs}(\mu(\ell_1)) \subseteq A \uplus \mathsf{A}(g)$. Furthermore $\mu(\ell_1) = \mathsf{ref}\ \ell_2$.
- $\mathsf{locs}(\ell_2) \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2))$, by $\mathsf{locs}(\ell_2) \subseteq \mathsf{locs}(\mathsf{ref}\ \ell_2)$ and $\mathsf{locs}(\mathsf{ref}\ \ell_2) \subseteq A \uplus \mathsf{A}(g)$ (just established) and $A \uplus \mathsf{A}(g) = A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2))$
- $\forall \ell \in (A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2))) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2))$, by $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$ (established above) and $A \uplus \mathsf{A}(g) = A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2))$.

Cases Fst and Snd: Similar to Bang.

Case App. We have $G = [g]$ and $e = \ell_1\ \ell_2$ and $\mu(\ell_1) = \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b$ and $\mu' = \mu$ and $G' = [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b)]$ and $e' = [\ell_1, \ell_2\ /\ f, x]e_b$. Assume $A\,;F \vdash_\mu [g]\,;\ell_1\ \ell_2$ drfde and $F \vdash [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b)]$ drf. By inversion of $A\,;F \vdash_\mu [g]\,;\ell_1\ \ell_2$ drfde, we have $F \vdash g$ drf and $A \vdash g$ de and $\mathsf{locs}(\ell_1\ \ell_2) \subseteq A \uplus \mathsf{A}(g)$ and $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$. Note that $A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b))$ is defined and equal to $A \uplus \mathsf{A}(g)$. We show $A\,;F \vdash_\mu [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b)]\,;[\ell_1, \ell_2\ /\ f, x]e_b$ drfde by showing:

- $F \vdash [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b)]$ drf, by inversion of $F \vdash [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2)]$ drf (which was assumed)
- $A \vdash [g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b)]$ de, by $- \oplus -$ and $(\mathbf{R}- \Rightarrow -)$ rules for de and by showing
  - $A \vdash g$ de, which was established above
  - $\ell_1 \in A \uplus \mathsf{A}(g)$, by $\ell_1 \in \mathsf{locs}(\ell_1\ \ell_2)$ and $\mathsf{locs}(\ell_1\ \ell_2) \subseteq A \uplus \mathsf{A}(g)$ (established above)
  - $\mathsf{locs}(\mathsf{fun}\ f\ x\ \mathsf{is}\ e_b) \subseteq A \uplus \mathsf{A}(g)$: By $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$ (established above) with $\ell_1 \in A \uplus \mathsf{A}(g)$ (just established), we have $\mathsf{locs}(\mu(\ell_1)) \subseteq A \uplus \mathsf{A}(g)$. Furthermore $\mu(\ell_1) = \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b$.

- $\mathsf{locs}([\ell_1, \ell_2 \ / \ f, x]e_b) \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2))$: by $\mathsf{locs}([\ell_1, \ell_2 \ / \ f, x]e_b) \subseteq \mathsf{locs}(\mathsf{fun}\ f\ x\ \mathsf{is}\ e_b) \cup \{\ell_1, \ell_2\} = \mathsf{locs}(\mathsf{fun}\ f\ x\ \mathsf{is}\ e_b) \cup \mathsf{locs}(\ell_1\ \ell_2)$ and $\mathsf{locs}(\mathsf{fun}\ f\ x\ \mathsf{is}\ e_b) \subseteq A \uplus \mathsf{A}(g)$ (just established) and $\mathsf{locs}(\ell_1\ \ell_2) \subseteq A \uplus \mathsf{A}(g)$ (established above) and $A \uplus \mathsf{A}(g) = A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b))$
- $\forall \ell \in (A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b))) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b))$, by $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$ (established above) and $A \uplus \mathsf{A}(g) = A \uplus \mathsf{A}(g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b))$.

Case AppSL. We have $e = e_1\ e_2$ and $\mu\,;G\,;e_1 \longmapsto \mu'\,;G'\,;e_1'$ and $e' = e_1'\ e_2$. Assume $A\,;F \vdash_\mu G\,;e_1\ e_2\ \mathsf{drfde}$ and $F \vdash G'\ \mathsf{drf}$. By inversion of $A\,;F \vdash_\mu G\,;e_1\ e_2\ \mathsf{drfde}$ (noting $e_1 = \ell_1$ is impossible), we can consider two cases:

- $G = [g]$ and $F \vdash g\ \mathsf{drf}$ and $A \vdash g\ \mathsf{de}$ and $\mathsf{locs}(e_1\ e_2) \subseteq A \uplus \mathsf{A}(g)$ and $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$: We establish $A\,;F \vdash_\mu [g]\,;e_1\ \mathsf{drfde}$ by showing:
  - $F \vdash g\ \mathsf{drf}$, assumed in this case
  - $A \vdash g\ \mathsf{de}$, assumed in this case
  - $\mathsf{locs}(e_1) \subseteq A \uplus \mathsf{A}(g)$, by $\mathsf{locs}(e_1) \subseteq \mathsf{locs}(e_1\ e_2)$ and $\mathsf{locs}(e_1\ e_2) \subseteq A \uplus \mathsf{A}(g)$ (assumed in this case)
  - $\forall \ell \in (A \uplus \mathsf{A}(g)) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g)$, assumed in this case

  By the induction hypothesis using $\mu\,;[g]\,;e_1 \longmapsto \mu'\,;G'\,;e_1'$ with $A\,;F \vdash_\mu [g]\,;e_1\ \mathsf{drfde}$ and $F \vdash G'\ \mathsf{drf}$ (assumed), we have $A\,;F \vdash_{\mu'} G'\,;e_1'\ \mathsf{drfde}$. By Lemma 1 with $\mu\,;[g]\,;e_1 \longmapsto \mu'\,;G'\,;e_1'$, we can consider two cases:
  - $G' = [g']$: By inversion of $A\,;F \vdash_{\mu'} [g']\,;e_1'\ \mathsf{drfde}$, we have $F \vdash g'\ \mathsf{drf}$ and $A \vdash g'\ \mathsf{de}$ and $\mathsf{locs}(e_1') \subseteq A \uplus \mathsf{A}(g')$ and $\forall \ell \in (A \uplus \mathsf{A}(g')) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g')$. We show $A\,;F \vdash_{\mu'} [g']\,;e_1'\ e_2\ \mathsf{drfde}$ by showing:
    - $F \vdash g'\ \mathsf{drf}$, which was established above
    - $A \vdash g'\ \mathsf{de}$, which was established above
    - $\mathsf{locs}(e_1'\ e_2) \subseteq A \uplus \mathsf{A}(g')$, by $\mathsf{locs}(e_1'\ e_2) = \mathsf{locs}(e_1') \cup \mathsf{locs}(e_2)$ and $\mathsf{locs}(e_1') \subseteq A \uplus \mathsf{A}(g')$ (established above) and $\mathsf{locs}(e_2) \subseteq \mathsf{locs}(e_1\ e_2)$ and $\mathsf{locs}(e_1\ e_2) \subseteq A \uplus \mathsf{A}(g)$ (assumed in this (outer) case) and $\mathsf{A}(g) \subseteq \mathsf{A}(g')$ (by Lemma 2 with $\mu\,;[g]\,;e_1 \longmapsto \mu'\,;[g']\,;e_1'$)
    - $\forall \ell \in (A \uplus \mathsf{A}(g')) \setminus F, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus \mathsf{A}(g')$, which was established above
  - $G' = g \oplus ([\bullet] \otimes [\bullet])$: We show $A\,;F \vdash_{\mu'} g \oplus ([\bullet] \otimes [\bullet])\,;e_1'\ e_2\ \mathsf{drfde}$ by showing:
    - $\neg(e_1'\ \mathsf{loc})$, because $e_1' = \ell_1'$ is impossible (no rules for $\longmapsto$ can derive $\mu\,;[g]\,;e_1 \longmapsto \mu'\,;g \oplus ([\bullet] \otimes [\bullet])\,;\ell_1'$)
    - $\mathsf{locs}(e_2) \subseteq A \uplus \mathsf{A}(g)$, by $\mathsf{locs}(e_2) \subseteq \mathsf{locs}(e_1\ e_2)$ and $\mathsf{locs}(e_1\ e_2) \subseteq A \uplus \mathsf{A}(g)$ (assumed in this (outer) case)
    - $A\,;F \vdash_{\mu'} g' \oplus ([\bullet] \otimes [\bullet])\,;e_1'\ \mathsf{drfde}$, which was establised above (as $A\,;F \vdash_{\mu'} G'\,;e_1'\ \mathsf{drfde}$)
- $G = g \oplus (G_1 \otimes G_2)$ and $\neg(e_1\ \mathsf{loc})$ and $\mathsf{locs}(e_2) \subseteq A \uplus \mathsf{A}(g)$ and $A\,;F \vdash_\mu g \oplus (G_1 \otimes G_2)\,;e_1\ \mathsf{drfde}$: By the induction hypothesis using $\mu\,;g \oplus (G_1 \otimes G_2)\,;e_1 \longmapsto \mu'\,;G'\,;e_1'$ with $A\,;F \vdash_\mu g \oplus (G_1 \otimes G_2)\,;e_1\ \mathsf{drfde}$ and

$F \vdash G'$ drf (assumed), we have $A\,;F \vdash_{\mu'} G'\,;e_1'$ drfde. By Lemma 1 with $\mu\,;g \oplus (G_1 \otimes G_2)\,;e_1 \longmapsto \mu'\,;G'\,;e_1'$, we can consider three cases:

- $G_1 = [g_1]$ and $G_2 = [g_2]$ and $G' = [g \oplus (g_1 \otimes g_2)]$: By inversion of $A\,;F \vdash_{\mu'} [g \oplus (g_1 \otimes g_2)]\,;e_1'$ drfde, we have $F \vdash g \oplus (g_1 \otimes g_2)$ drf and $A \vdash g \oplus (g_1 \otimes g_2)$ de and $\mathsf{locs}(e_1') \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$ and $\forall \ell \in (A \uplus [g \oplus (g_1 \otimes g_2)]) \setminus F, \mathsf{locs}(\mu'(\ell)) \subseteq A \uplus [g \oplus (g_1 \otimes g_2)]$. We show $A\,;F \vdash_{\mu'} [g \oplus (g_1 \otimes g_2)]\,;e_1'\ e_2$ drfde by showing:
  - $F \vdash g \oplus (g_1 \otimes g_2)$ drf, which was established above
  - $A \vdash g \oplus (g_1 \otimes g_2)$ de, which was established above
  - $\mathsf{locs}(e_1'\ e_2) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$, by $\mathsf{locs}(e_1'\ e_2) = \mathsf{locs}(e_1') \cup \mathsf{locs}(e_2)$ and $\mathsf{locs}(e_1') \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$ (established above) and $\mathsf{locs}(e_2) \subseteq A \uplus \mathsf{A}(g)$ (assumed in this (outer) case) and $A \uplus \mathsf{A}(g) \subseteq A \uplus \mathsf{A}([g \oplus (g_1 \otimes g_2)])$
  - $\forall \ell \in (A \uplus [g \oplus (g_1 \otimes g_2)]) \setminus F, \mathsf{locs}(\mu'(\ell)) \subseteq A \uplus [g \oplus (g_1 \otimes g_2)]$, which was established above
- $G' = g \oplus (G_1' \otimes G_2)$: We show $A\,;F \vdash_{\mu'} g \oplus (G_1' \otimes G_2)\,;e_1'\ e_2$ drfde by showing:
  - $\neg(e_1'\ \mathsf{loc})$, because $e_1' = \ell_1'$ is impossible (no rules for $\longmapsto$ can derive $\mu\,;g \oplus (G_1 \otimes G_2)\,;e_1 \longmapsto \mu'\,;g \oplus (G_1' \otimes G_2)\,;\ell_1'$)
  - $\mathsf{locs}(e_2) \subseteq A \uplus \mathsf{A}(g)$, assumed in this (outer) case
  - $A\,;F \vdash_{\mu'} g' \oplus (G_1' \otimes G_2)\,;e_1'$ drfde, which was establised above (as $A\,;F \vdash_{\mu'} G'\,;e_1'$ drfde)
- $G' = g \oplus (G_1 \otimes G_2')$: Similar to $G' = g \oplus (G_1' \otimes G_2)$ case.

Cases AppSR, PairSL, PairSR, FstS, SndS, RefS, BangS, UpdSL, and UpdSR: Similar to AppSL.

□

Lemma 7. For any $\mu\,;G\,;e \longmapsto^* \mu'\,;G'\,;e'$, if $A\,;F \vdash_\mu G\,;e$ drfde and $F \vdash G'$ drf, then $A\,;F \vdash_{\mu'} G'\,;e'$ drfde.

Proof. By induction on the the derivation of $\mu\,;G\,;e \longmapsto^* \mu'\,;G'\,;e'$, using Lemma 6. □

3.5 Disentanglement Beyond Race-Freedom

Superficially, it may appear that disentanglement prevents determinacy races, because it does not allow concurrent tasks to have knowledge of each other's allocations. But this is not correct. Disentanglement permits many kinds of races, and is general enough to even permit arbitrary communication in some cases.

To understand the interplay between determinacy races and disentanglement, consider that any determinacy race between two concurrent tasks may be classified either as a *write-write* race or a *read-write* race. A ***write-write*** race occurs when both tasks modify the same location, whereas a ***read-write*** race occurs when one of the tasks reads a location that the other task modifies. Write-write races are always safe for disentanglement, because writes can never "discover" new locations. In the case of read-write races, however, we have to be careful to

ensure that the reading task does not discover new locations. That is, a read-write race is disentangled only when the data being written was allocated by a common ancestor. This leads to a simple but powerful observation: as long as all possibly shared data is pre-allocated, disentanglement permits arbitrary communication between concurrent tasks.

Examples

The following examples illustrate a number of use-cases for disentangled race conditions.

Speculative search. In a parallel speculative search, we can use a shared "found-it" flag to quit early as soon as a suitable element has been found. Specifically, we allocate the flag and then begin searching in parallel with many subtasks. When one of the subtasks finds a desirable element, it sets the flag; meanwhile, all subtasks regularly poll the flag to check if they can quit early. Therefore we have a read-write race, but this example is nevertheless disentangled because we allocate the flag before the subtasks begin.

We can extend this example to non-deterministically select one suitable element. To do this, we allocate a mutable pointer and then instruct each subtask to set the pointer to the element it finds (if any). Multiple subtasks might then race the update the pointer (a write-write race), but this is disentangled because none of the subtasks ever read the pointer. Once all subtasks complete, the pointer may be safely dereferenced.

Graph search. In graph search algorithms where the number of vertices in the graph is known, we can use one "visited" flag per vertex to guarantee that each vertex is processed at most once. All of these flags must be allocated when the search begins, so that the read-write races on the flags are safe for disentanglement. A similar technique is used in the parallel BFS discussed in Section 2.3.1, where atomic compare-and-swap operations are used to visit vertices in parallel.

Concurrent union-find. We can implement concurrent union-find (dynamic disjoint sets), for example as described in [32], on a fixed number of nodes. Due to concurrency, any path compression within the structure is typically non-deterministic (due to race conditions). That is, while a path is being compressed, another operation might traverse the path. With some care, utilizing properly synchronized atomic operations, these race conditions are benign. Union-find is a crucial subcomponent in graph algorithms such as minimum-spanning-tree, where one union-find node is used per vertex in the graph. When the number of vertices is known ahead-of-time, all nodes may be allocated at the start of the algorithm.

Deduplication by hashing. We can implement parallel deduplication using a concurrent hash table on a collection of known (i.e., already allocated in memory) elements, for example as shown in Section 2.3.2. In this example, many subtasks insert elements into a hash table concurrently and in parallel. When any individual subtask inspects a slot of the hash table, it may either find an empty slot, or an element which was previously inserted (possibly by a concurrent sibling/cousin task). This is disentangled because the elements inserted into the

hash table are allocated before all subtasks begin; therefore, even though a subtask may find an element that was inserted concurrently, the element is nevertheless safe to access.

Concurrent data structures. Any concurrent data structure (such as a queue, stack, hash table, etc.) is safe for disentanglement as long as all data associated with the structure can be pre-allocated. In particular, the size of the structure (including the cumulative sizes of its elements) must be bounded, so that sufficient space can be allocated up-front.[4] Such a collection may then be used by multiple concurrent tasks to communicate freely.

[4]This restriction is conservative, making no assumptions about the details of the data structure itself. The goal is to ensure that no task will inspect a piece of the structure that may have been allocated by another concurrent task. These restrictions can be relaxed for particular data structures or use-cases, by specializing operations to preserve disentanglement. Some of the examples in this section illustrate this approach.

Chapter 4

Disentangled Memory Management

To manage memory in an efficient and scalable manner, we take advantage of the memory separation property afforded by disentanglement. In particular, by assigning each task its own local heap for allocation, disentanglement guarantees that the heaps of concurrent tasks do not contain objects pointing at each other. This makes it possible for individual tasks to collect their heaps independently, without synchronizing with other (concurrent) tasks. By scheduling many such task-local GCs simultaneously, our approach is naturally parallel.

More concretely, we show that task-local heaps may be organized in a tree structure called the ***heap hierarchy*** which mirrors the (dynamic) structure of parallelism in the program. In the heap hierarchy, any pointer between two heap-allocated objects may be classified as either an *up-pointer*, *down-pointer*, *internal pointer*, or *cross-pointer*, depending on the ancestry relationship between heaps. Disentanglement guarantees that the heap hierarchy is free of cross-pointers; therefore, any disjoint collection of subtrees may be collected in parallel with no additional synchronization. We call this technique ***subtree collection***, because the scope of each collection is a subtree, i.e., a heap together with all of its descendants.

Subtree collections are facilitated by lightweight remembered sets, which remember down-pointers from ancestor heaps into descendant heaps.[1] Down-pointers are used as additional roots for collections. We also describe an optional *promotion* strategy which eliminates down-pointers by copying data upwards in the hierarchy. Promotion is not necessary for correctness, but can be useful for efficiency, and can easily be integrated into subtree collections (by performing a full promotion phase before the tracing phase of collection begins). Outside of down-pointer maintenance, the tracing algorithm for subtree collection is fairly unconstrained, and essentially any tracing collection algorithm can be used, including compaction algorithms.

To schedule garbage collections, we couple memory management closely with the task scheduler, which assigns tasks to processors. In our approach, the task scheduler additionally assigns heaps to processors, specifically by mirroring the assignment of tasks. That is, every task and its corresponding heap are always assigned to the same processor, which is beneficial for performance (e.g. for improved data locality), and additionally simplifies the implementation of the garbage collector. To perform garbage collection, a processor only needs to interrupt its currently assigned task; the processor then may safely collect its own assigned

[1]The management of remembered sets and down-pointers is reminiscent of generational garbage collections.

heap(s) without needing to synchronize with any other (concurrent) tasks. This approach to collection—where processors only collect their own assigned heaps—is essentially a special kind of subtree collection. We refer to these collections as ***local garbage collections***, or ***LGC*** for short.

The LGC technique alone does not cover all heaps in the hierarchy: any "internal" heap with at least two active descendants is not local to a single processor, and therefore will not be in scope of a local garbage collection. We therefore develop a second garbage collection mechanism which can reclaim memory in internal heaps without pausing active descendant tasks. At a high level, our approach is to use a snapshot-at-the-beginning strategy [161] together with a concurrent non-moving tracing algorithm. We refer to these as ***concurrent garbage collections***, or ***CGC***, for short. To spawn a CGC, only a single active task needs to be temporarily paused to take a snapshot; after the snapshot has been recorded, the task may be resumed and proceed concurrently with—and in parallel with—the CGC. This effectively places CGC off the critical path, ensuring that CGC does not interfere with parallelism.

To allow for CGCs to proceed concurrently and in parallel with other tasks and garbage collections throughout the hierarchy, we extend the heap hierarchy with a mechanism we call ***CGC-chaining***. In the extended heap hierarchy, ***CGC-heaps*** (i.e., heaps currently undergoing CGC) are explicitly kept separate from ***primary heaps***. Primary heaps therefore remain available for fresh allocations, without needing to synchronize with an ongoing CGC.

Altogether, in a disentangled heap hierarchy, the techniques presented in this chapter make it possible to garbage-collect any individual heap independently, with little-to-no additional synchronization between parallel tasks. In this way, our collection algorithms are naturally parallel and concurrent, and never need to "stop the world".

4.1 Preliminaries: Heaps and Heap Objects

A ***heap object***, or simply ***object***, is a contiguous section of memory that is allocated as a unit. Objects may store both non-pointer data (e.g. numbers) and pointers to other objects. During execution, programs allocate new objects and read and write existing objects. The objects of an execution form a ***memory graph*** where vertices are objects and (directed) edges are pointers between objects. Memory graphs evolve over time as the program executes: allocations add new vertices and (possibly) edges, and writes can delete existing edges, replacing them with new edges pointing at different objects.

A ***heap*** is, abstractly, a set of objects. Many heaps can exist simultaneously, but they must be disjoint: each object exists in at most one heap. Heaps are an abstract data type (we describe how to implement them in Chapter 6) that offer a variety of natural operations: creation of a fresh empty heap, allocation of a new object in a heap, deletion of an object from a heap, and moving an object from one heap to another. We also permit *merging* two heaps (unioning their contents), and querying which heap contains an object. We write $H(x)$ for the heap that contains object x; in general, this is a dynamic query, as objects may be moved between heaps.

The ***roots*** are the set of objects mentioned explicitly by the program state (e.g. in the semantics of Chapter 3, the roots of expression e are $\mathsf{locs}(e)$). As a program executes, the roots change. Every object in the memory graph is either ***live*** or ***garbage***, depending upon whether or not it

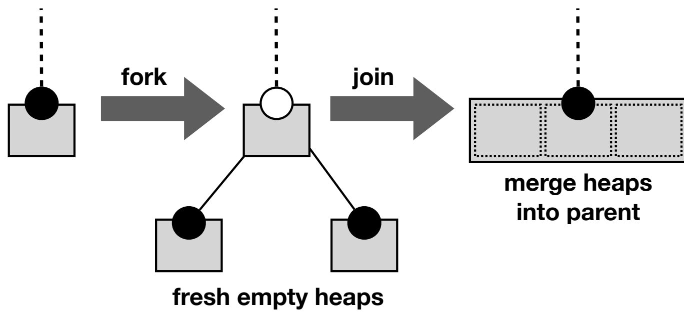

Figure 4.1: Forks and joins. Active tasks are black circles, and suspended tasks are white circles. Each task has a heap, drawn as a gray rectangle.

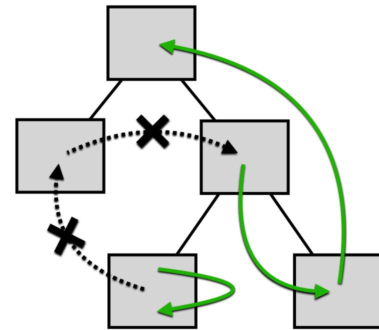

Figure 4.2: A disentangled heap hierarchy. Up, down, and internal pointers (solid) are permitted. Cross-pointers (dotted) are disallowed.

is reachable from the roots (by following pointers in the memory graph). As the program executes, live objects may become garbage by either (a) dropping a root, or (b) deleting an edge of the memory graph (with an update). Garbage objects will never again be used by the program, and so they may be de-allocated (reclaimed) by deleting them from their corresponding heaps. The goal of ***garbage collection*** is to reclaim space occupied by garbage objects.

4.2 Heap Hierarchy

We give each task its own heap and organize heaps into a tree that mirrors the task tree (Section 3.1.3). We call this tree of heaps the ***heap hierarchy***. Initially, there is a single root heap, corresponding to the initial task. When a task forks, its subtasks are initialized with two fresh (empty) heaps and, when both subtasks of a task complete, their heaps are merged with the heap of the parent task (see Figure 4.1). This puts heaps and tasks in a one-to-one correspondence.

4.2.1 Pointer Directions

In the heap hierarchy, we can use the ancestor/descendant relationships of heaps to give each memory graph pointer a direction: *up*, *down*, *internal*, or *cross*. A pointer from object x to object y is classified as follows:

- if $H(x)$ is a proper descendant of $H(y)$, the pointer is an ***up-pointer***;
- if $H(x)$ is a proper ancestor of $H(y)$, it is a ***down-pointer***;
- if $H(x) = H(y)$, it is a ***internal pointer***;
- otherwise, it is a ***cross-pointer***.

4.2.2 Guarantees of Disentanglement

As illustrated in Figure 4.2, disentanglement provides a strong guarantee on the directions of the pointers in the memory graph: there are no cross-pointers (Property 1). Furthermore, disentanglement guarantees that the roots of the program only point up (Property 2).

Property 1. *Throughout execution of a disentangled program, all pointers in the memory graph are either up-pointers, down-pointers, or internal pointers.*

Property 2. *Throughout execution of a disentangled program, for every task, every root of that task lies within either its own heap or an ancestor heap.*

If desired, these properties could be stated formally in a manner similar to the drfde judgement of Section 3.4. In particular, in Figure 3.10, the component highlighted in blue is essentially the statement of Property 1 (one would only need to eliminate the use of F which is specific to drfde), and the components highlighted in red capture Property 2. A formal proof can then proceed in a manner similar to the proof of Theorem 1. The gist of the proof is as follows. Initially when there are no allocated objects, both the memory graph and roots are empty, so both properties hold initially. We have Property 1 throughout the execution of a disentangled program because (a) allocations can only create up-pointers in the memory graph (because new allocations are always in the task's heap and by Property 2 the locations of the newly allocated storable lie within the task's heap or ancestor heaps), (b) writes can only create either up-pointers or down-pointers (again, because by Property 2 the written-to location and the stored location both lie within the task's heap or ancestor heaps), and (c) heap merges can only cause down-pointers to become up-pointers (and therefore cannot introduce cross-pointers). We have Property 2 throughout the execution of a disentangled program because new allocations are always in leaf heaps, and because at each read we are guaranteed by Property 1 that any newly obtained pointers are into the task's heap or ancestor heaps.

4.2.3 Relationship to Computation Graphs

The heap hierarchy directly implements the structure of allocations in an open computation graph G, where the memory locations of Section 3.1 are used as object identifiers. We derive the heap hierarchy corresponding to G as follows: if $G = [g]$ then it is just the single heap containing the objects $\mathsf{A}(g)$, otherwise the heap hierarchy of $G = g \oplus (G_1 \otimes G_2)$ is a heap containing the objects $\mathsf{A}(g)$ with two children which are the heap hierarchies of G_1 and G_2, respectively.

We can see that this correspondence between the heap hierarchy and an open computation graph is correct by examining forks, joins, and allocations. Forks are witnessed by replacing a leaf $[g]$ with $g \oplus ([\bullet] \otimes [\bullet])$, which is implemented by creating two empty heaps, corresponding to the new leaves $[\bullet]$. Joins occur when a graph $g \oplus ([g_1] \otimes [g_2])$ is replaced by $[g \oplus (g_1 \otimes g_2)]$; this corresponds to three heaps $h = \mathsf{A}(g)$, $h_1 = \mathsf{A}(g_1)$, and $h_2 = \mathsf{A}(g_2)$ being merged into a single heap $h \uplus h_1 \uplus h_2 = \mathsf{A}(g \oplus (g_1 \otimes g_2))$. Finally, for each allocation, a leaf $[g]$ is replaced by some $[g \oplus (\mathbf{A}\ell \Leftarrow s)]$. Since tasks store locally allocated data in their own heaps, this corresponds to extending the heap $\mathsf{A}(g)$ with a fresh location ℓ, forming a heap $\mathsf{A}(g) \uplus \{\ell\} = \mathsf{A}(g \oplus (\mathbf{A}\ell \Leftarrow s))$.

4.3 Subtree Collection

We describe an algorithm called ***subtree collection***, where a ***subtree*** consists of a heap and all of its descendants. As the name suggests, subtree collections are localized to a subtree of the heap hierarchy.

Utilizing Disentanglement. When performing collection on only a small region of the memory graph, it is necessary to find all incoming pointers (x, y) from live objects x outside the region to objects y inside the region, so that the set of live objects inside the region can be determined. In general however, knowing the set of live objects outside the region requires tracing the entire memory graph, which defeats the goal of a localized collection (cheaper collection with smaller scope). A common simplification made is to assume that all incoming pointers are live, which makes it possible to perform collection locally without needing to trace the entire memory graph (at the potential cost of preserving some dead objects). The idea is to explicitly keep track of incoming pointers in a so-called *remembered set.* Then, when a garbage collection begins, all incoming pointers are already known, and available to be used as additional roots.

In our case, disentanglement guarantees that all incoming pointers into a subtree are *down-pointers.* This is because of Property 1 and the fact that any up-pointer into a subtree must have originated from within the subtree. The fact that all incoming pointers into a subtree are down-pointers has multiple benefits. First, it means that in order to perform subtree collection, we only need to remember down-pointers. But more importantly, it means that a subtree collection only needs to access objects within or above the subtree. Since in a nested-parallel program, all ancestor tasks are suspended, this results in *independence* of subtree collections, which in turn enables a conceptually simple parallel garbage collection strategy: perform many disjoint (non-overlapping) subtree collections simultaneously across the hierarchy.

Subtree Collection. Subtree collection primarily consists of a *tracing* phase which identifies the set of *survivors* S within the subtree that are reachable from roots or down-pointers. Any object which is not a survivor is garbage. We also describe an optional *promotion* phase which eliminates down-pointers by copying data upwards in the hierarchy. Note that during promotion, objects may be moved to different heaps, in which case an object that originally was in-scope may become out-of-scope. In order to preserve disentanglement, objects are only ever be moved upwards in the hierarchy.

We now describe the tracing and (optional) promotion phases in more detail.

4.3.1 Tracing Phase

Let T be the set of heaps that lie within a subtree. We say that an object x is *in-scope* if $H(x) \in T$; otherwise, x is *out-of-scope.* For the subtree T, let R be the set of roots, and let D be the set of down-pointers (x, y) where $H(x) \notin T$ and $H(y) \in T$. The additional roots due to down-pointers are $R_D = \{y \mid (x, y) \in D\}$.

The tracing phase begins with the initial set of survivors $S \leftarrow \{x \in R \cup R_D \mid H(x) \in T\}$, i.e. the set of in-scope roots. Tracing proceeds by performing the following:

1. Pick a pointer (x, y) where $x \in S$ and $y \notin S$ and $H(y) \in T$. (If there are no such pointers, tracing is complete.)
2. Insert y into S.
3. Repeat.

Once tracing completes, the set S contains all live in-scope objects. After tracing, subtree collection completes by reclaiming the objects $\{x \notin S \mid H(x) \in T\}$.

4.3.2 Optional Promotion Phase

The goal of promotion is to eliminate down-pointers by moving objects upwards in the hierarchy. Promotion is motivated by efficiency: an object y which is referenced by an out-of-scope object x cannot be reclaimed by a subtree collection. Taking inspiration from generational collectors, rather than let such an object y persist through multiple collections, we can instead *promote* it to a higher heap which is collected less often. In this way, down-pointers are analogous to inter-generational pointers from old objects to young objects. By delaying the promotion of objects until garbage collection, promotion becomes very cheap, as the promotion of many objects can be batched and any performance artifacts of promotion can be hidden from the mutator program.

If used, promotion occurs before the tracing phase begins. Promotion proceeds by performing the following.

1. Let D be the set of candidate down-pointers (x, y) where $H(x) \notin T$ and $H(y) \in T$. (If there are no such down-pointers, promotion is complete.)
2. Pick $(x, y) \in D$ where $H(x)$ is shallowest amongst $\{H(x') \mid (x', y') \in D\}$.
3. Promote y by moving it to $H(x)$. (This promotion may create new down-pointers, including candidate down-pointers.)
4. Repeat.

Once promotion completes, there are no more down-pointers to in-scope objects from out-of-scope objects. Note that it is possible for there to be new down-pointers from promoted objects to out-of-scope objects, after promotion completes. However, promotion cannot create cross-pointers, because it only moves objects upwards in the hierarchy.

The order in which promotion processes down-pointers is important for efficiency: by operating from top to bottom, we guarantee that each object is promoted at most once. In particular, in step 2 of the promotion phase, it is crucial that $H(x)$ is shallowest amongst *all* candidate down-pointers. This guarantees, in chains of down-pointers, that the objects in the chain are promoted in order of shallowest to deepest. Otherwise, the deepest objects in the chain could be promoted multiple times.

4.3.3 Example

An example subtree collection, both with and without the optional promotion phase, is shown in Figure 4.3. In this example, there are five heaps depicted as large rectangles, and the three bottom-most heaps are in-scope for collection. The small squares are objects, the diamonds are root objects, and the arrows are pointers between objects. During collection, if promotion is

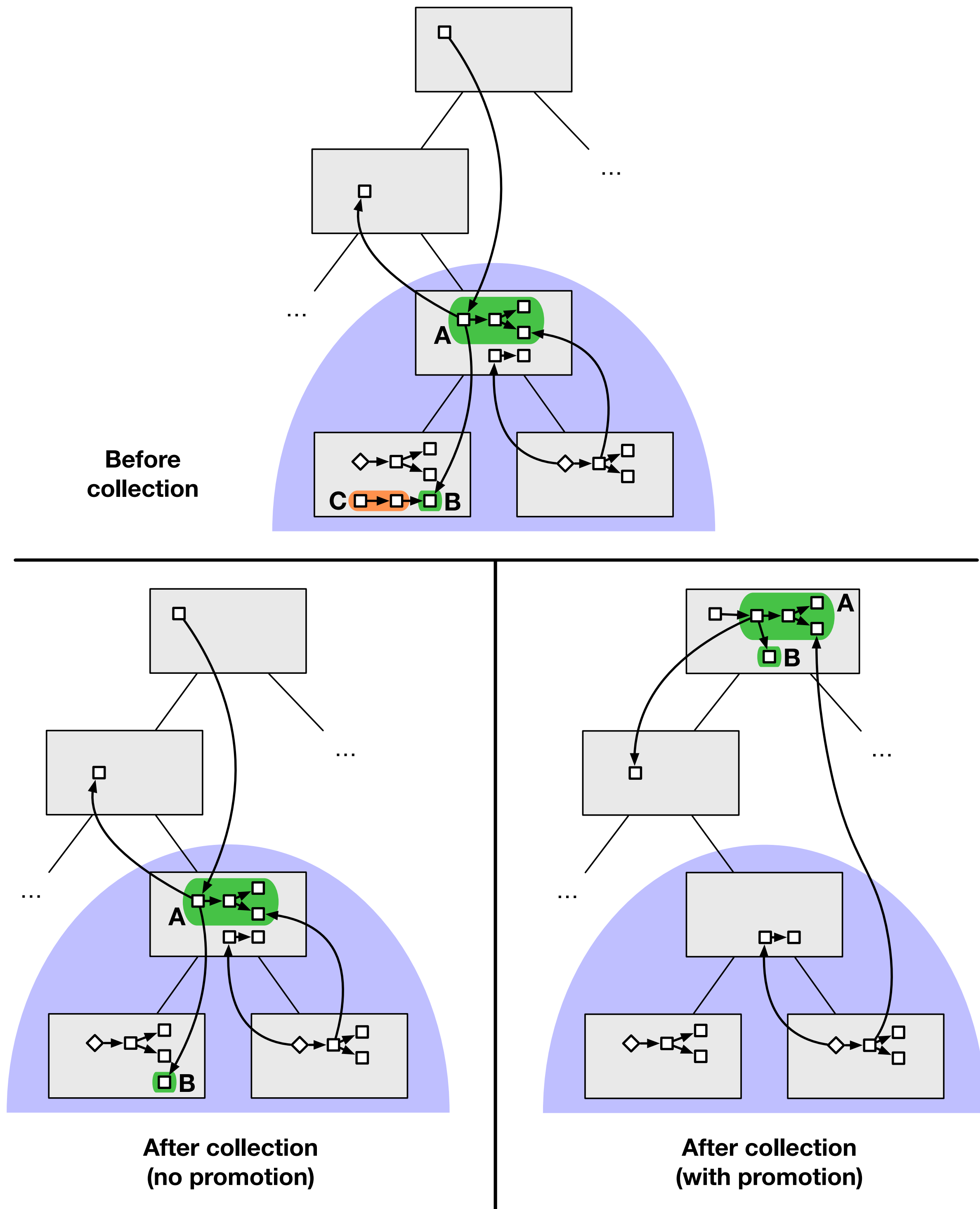

Figure 4.3: Before and after an example subtree collection of the bottom-most three heaps, with and without the optional promotion phase. The large rectangles are heaps, the squares are objects, and the diamonds are root objects. Highlighted groups **A** and **B** are kept live due to down-pointers. The group **C** is garbage and is reclaimed.

not used, then the highlighted groups of objects **A** and **B** are kept live due to down-pointers. If promotion is used, then **A** and **B** are promoted to the topmost heap. The group **C** is garbage, and is reclaimed.

4.3.4 Correctness

We now argue that subtree collection never reclaims an object that is reachable from the roots. Consider some live object x where initially $H(x) \in T$. There are two cases: either x is promoted to a heap outside the subtree, or it is not. In the former case, x will not be reclaimed because it becomes out-of-scope. In the latter case, consider that due to the lack of cross-pointers, after promotion completes, we have the guarantee that every path in the memory graph which ends at x is entirely contained within the subtree. Because we assumed that x is live, we know there exists a particular path $x_1, \ldots, x_n$ where x_1 is a root and $x_n = x$. This path ends at x, and so $H(x_i) \in T$ for each i. Since $H(x_1) \in T$ and x_1 is a root, we know that $x_1 \in S$ initially in the tracing phase. Therefore once tracing completes, we also know every $x_i \in S$ (because the path is contained within the subtree), including $x = x_n \in S$. No objects in S are reclaimed, so x is not reclaimed.

4.3.5 Independence of Subtree Collections

A subtree collection (consisting tracing together with an optional promotion phase) only accesses objects within the subtree or within ancestor heaps of the subtree, and furthermore only moves objects that lie within the subtree. This means that any two disjoint subtrees—that is, any two subtrees with no heaps in common—may be collected independently and in parallel, because any shared ancestors are guaranteed to be outside the scope of both collections. One subtlety is that two concurrent collections may promote two different objects into the same shared ancestor heap at the same time, however this scenario does not harm independence, because (a) insertions commute, and (b) neither collection will attempt to access the other's promoted objects. Subtree collections, in addition to being independent of other disjoint collections, are also independent of the actions of concurrent tasks. That is, a subtree collection may be performed locally upon the subtree without interrupting tasks that own heaps outside the subtree.

4.4 Scheduling and Local Garbage Collection (LGC)

Scheduling Preliminaries. In our setting of nested fork-join parallelism, any at moment during scheduling, each task can be classified as one of *ready*, *active*, *suspended*, or *completed*. A task is ***ready*** if it is available to be scheduled on a processor, but is not currently being executed. A task is ***active*** if it is currently being executed by a processor. A task is ***suspended*** if it is waiting for child tasks to complete. And finally, a task is ***completed*** if it has no more steps to take and has therefore terminated.

All computation occurs at active tasks. When an active task forks, it becomes suspended, and two (ready, or active if immediately scheduled on a processor) children are created. When

both children have completed, their parent may then resume as a ready task (or active, if immediately scheduled).

Local Heap Assignment. We now show how to assign heaps to processors based on the current state of the tasks during scheduling. The resulting heap assignment is a valid partioning of heaps into disjoint subtrees, each of which may be individually collected by a single processor by performing a subtree collection. We refer to each such collection as a ***local garbage collection***, or ***LGC*** for short. LGCs are a specialized form of subtree collection which is integrated with scheduling. In an LGC, the subtree being collected always consists of heaps which are "local" to a single processor. Specifically, to perform an LGC, a processor only needs to interrupt its current active task, and does not need to synchronize with any other active task assigned to a different processor.

Our assignment of heaps for local garbage collections is straightforward. For each processor which is currently executing an active task t, we assign that processor the largest subtree of heaps which contains t's heap but does not contain any heap of another active task. Any processor can compute its own heap assignment with the following algorithm. Here we write H_t for the heap corresponding to task t. The algorithm computes a set T of heaps which constitute the current heap assignment for a single processor.

1. Let t be the current active task being executed by a processor.
2. Initialize $T \leftarrow \{H_t\}$.
3. If t is the root task, then we are done.
4. Otherwise, let p be the (suspended) parent task of t, and let s be the sibling task of t.
5. If s is active or suspended, then we are done.
6. Otherwise (if s is either ready or completed), then:
    - Set $T \leftarrow T \cup H_p \cup H_s$
    - Set $t \leftarrow p$
    - Repeat from step 3.

Note that throughout the algorithm, T always constitutes a valid subtree. When the algorithm terminates, T corresponds to the largest subtree which contains the single active task assigned to a processor.[2] The subtree T may then be garbage collected via our subtree collection algorithm, and we call such a collection a *local garbage collection*.

Example. Figure 4.4 shows an example with two processors and two active tasks currently being executed by those processors. The local scopes of each processor, as computed by the above heap assignment, are highlighted.

[2] The subtree computed in this manner is largest only if every suspended task has at least one active descendant, which is guaranteed for any non-preemptive scheduler.

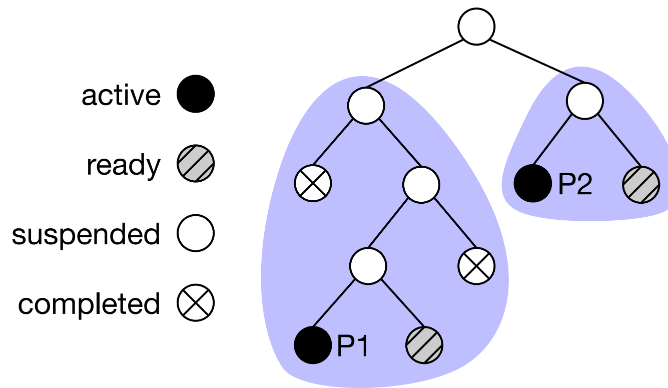

Figure 4.4: Example heap assignments for local collections, with two processors (**P1** and **P2**) and two corresponding active tasks.

4.5 Concurrent Garbage Collection (CGC)

Local collections alone do not cover all heaps in the hierarchy. For example, in Figure 4.4, the root of the hierarchy is not in scope of either collection. In general, shallow heaps (i.e., heaps close to the root) may have multiple active descendants, and therefore will not be in-scope of LGC. To collect such heaps, we present a concurrent garbage collection technique, called ***CGC***, for short.

CGC uses a non-moving concurrent tracing algorithm based on a snapshot-at-the-beginning strategy [161], and is therefore able to collect any heap without pausing all of the heap's active descendant tasks. In contrast, although LGC only needs to pause a single active task, there are potentially many ready (but not active) tasks in-scope of an LGC which must remain inactive until LGC completes. This makes it possible for LGC to compact memory without needing to worry about concurrent access from in-scope tasks. In this way, LGC and CGC are complementary: LGC offers fast parallel reclamation and compaction of new allocations near the leaves of the hierarchy, while CGC offers reclamation within shallow heaps.

4.5.1 Primary Heaps and CGC-heaps

As a concurrent collection technique, we allow the program the continue execution concurrently with in-progress CGCs (of which there could be many proceeding in parallel throughout the hierarchy). Specifically, we schedule CGCs off the critical path, and do not require any active tasks to wait for a CGC to complete.[3] This presents a technical challenge: when two completed parallel tasks join and their parent is ready to resume, the heap of the parent might be currently undergoing a CGC, and therefore unavailable for fresh allocations.

To solve this challenge, we extend hierarchical memory management by distinguishing between two types of heaps: ***primary heaps***, and ***CGC-heaps***. Primary heaps may be used for fresh allocations by active tasks (and also may be subjected to local garbage collections, by pausing active tasks). In contrast, CGC-heaps do not permit additional allocation. Each CGC-heap

[3]From the perspective of a computation graph, a CGC can be modeled as a task which is forked but not joined.

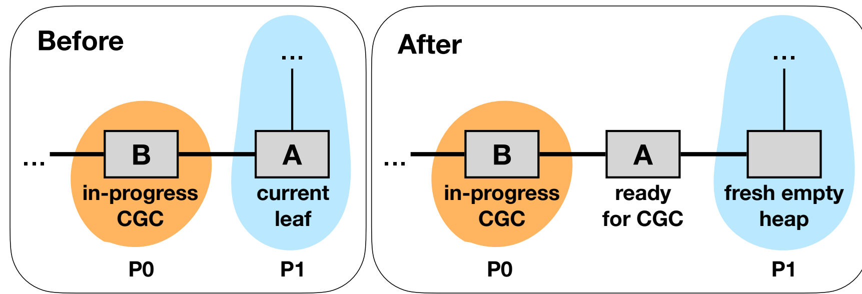

Figure 4.5: Spawning a new CGC-task: processor **P1** pushes heap **A** onto a CGC-chain and continues with a fresh (primary) heap.

corresponds to a single in-progress CGC, and the contents of a CGC-heap constitute the scope of the CGC. We associate each CGC-heap with a ***CGC-task***, which is spawned and scheduled similar to normal tasks, and is responsible for performing the CGC. We refer to normal tasks as ***primary tasks*** to distinguish them from CGC-tasks.

4.5.2 CGC Chaining

In the heap hierarchy, CGC-heaps are distinguished from primary heaps by augmenting each primary heap with a ***CGC-chain***: a list of CGC-heaps that are either ready for CGC or are currently undergoing CGC by another processor. That is, each primary heap has zero or more CGC-heaps attached to it. Spawning a CGC-task creates a new CGC-heap and extends the CGC-chain. As soon as all CGCs in a chain have completed, we merge all of these heaps back into their corresponding primary heap.

Spawning CGC-tasks. Figure 4.5 illustrates how a CGC-task is spawned and the corresponding CGC-chain is extended. In the figure, processor **P1** has decided to spawn a CGC-task for its current primary leaf heap, labeled **A**. The processor carries this out by pushing **A** onto the local CGC-chain, spawning the CGC-task, and finally continuing with a fresh primary heap. In this way, heap **A** becomes a CGC-heap, corresponding to a new CGC-task. The scheduler may then assign the new CGC-task to a processor as it sees fit. In the figure, we also show a second processor **P0** performing another CGC on a different CGC-heap (**B**), uninterrupted by the newly spawned CGC-task.

Joining primary tasks. When two children complete and join with their parent task, there may be non-empty CGC-chains (i.e., in-progress CGC-tasks) at the children, at the parent, or both. There are two cases for joins, shown in Figure 4.6. Both cases consider joining child primary heaps **B** and **C** into parent primary heap **A**.

In case 1, neither **B** nor **C** have any chained heaps, so the join is simple: we merge the two child heaps (**B** and **C**) into the parent's primary heap (**A**) and then resume the parent task.

In case 2, at least one child has a chained heap. In this case, we concatenate the chains, and also push the parent heap (**A**) into the chain to the left of any chained heaps on **B** and **C**. When

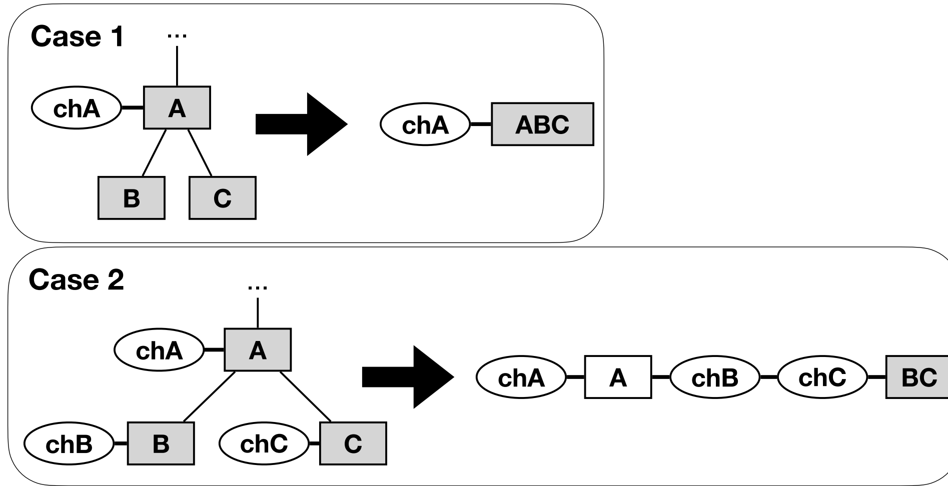

Figure 4.6: Two cases handling CGC-chains at joins. The rectangles are heaps, and the ovals are chains containing one or more heaps. Primary heaps are shaded.

the parent heap **A** is pushed into the chain, we do not collect it; rather, we treat this heap as though a collection has already finished on it.

The special handling of the parent heap in case 2 of Figure 4.6 ensures that immutable pointers between objects are properly tracked. Specifically, we ensure that immutable pointers only point upward in the heap tree, or to the left in a CGC-chain. This invariant is necessary for preserving the correctness of the snapshotting technique for CGCs. We discuss this in more detail below: Section 4.5.3 describes pointer invariants, and Section 4.5.4 describes the snapshotting and tracing algorithms for CGC.

4.5.3 Pointer Directions, Revisited

To establish invariants on the directions of pointers under CGC chaining, we extend our prior classification of pointers with two additional classifications: *left* and *right* pointers, which are pointers between heaps within a CGC-chain. Note that under CGC chaining, the heap structure is no longer a tree, and therefore the prior definitions for up, down, internal, and cross pointers (Section 4.2) are not immediately applicable. We can recover the prior definitions by identifying a *representative heap* for each object. The representative heap of an object is a primary heap; note that the primary heaps alone form a rooted tree structure, and therefore can be used to identify up, down, and cross-pointers.

Similar to before, we write $H(x)$ for the heap that contains an object x, which could be either a primary or a CGC-heap. We define the ***representative heap*** of an object, denoted $RH(x)$, as follows.

$$RH(x) \triangleq \begin{cases} H(x), & \text{if } H(x) \text{ is a primary heap} \\ h, & \text{o.w., where } h \text{ is the primary heap of the CGC chain containing } H(x) \end{cases}$$

We then classify a pointer from x to y as follows.

- *internal pointer*, if $H(x) = H(y)$;

- *up-pointer*, if $RH(x)$ is a descendant of $RH(y)$.
- *down-pointer*, if $RH(x)$ is an ancestor of $RH(y)$.
- *cross-pointer*, if neither $RH(x)$ nor $RH(y)$ is an ancestor of the other.
- ***left-pointer***, if $H(x)$ and $H(y)$ lie in the same chain (i.e. $RH(x) = RH(y)$) and $H(y)$ is to the left of $H(x)$ in the chain.
- ***right-pointer***, if $H(x)$ and $H(y)$ lie in the same chain (i.e. $RH(x) = RH(y)$) and $H(y)$ is to the right of $H(x)$ in the chain.

The first four (internal, up, down, and cross) are identical to before, but now appropriately rephrased in terms of primary and CGC-heaps. The two new classifications are *left-* and *right-pointers*, which lie within a chain.

Pointer invariants, revisited. All immutable pointers are either internal, up, or left-pointers. In this way, left-pointers are analogous to up-pointers. Right-pointers are analogous to down-pointers: to create a right-pointer from x to y, the object x has to mutable, and there must have been an in-place update of x to point to y.

Invariants are preserved by joins. Inspecting Figure 4.6 more closely, we see that the two cases correctly preserve these invariants. In particular, after a join, a down-pointer becomes either internal or a right-pointer. Similarly, a join may cause an up-pointer to become either an internal or a left-pointer.

4.5.4 CGC Snapshotting and Tracing

To trace memory and collect garbage, just like any collection algorithm, CGC requires a (potentially conservative) set of roots. Here, we use a SATB (i.e. snapshot-at-the-beginning [161]) collection algorithm. At the time a CGC-task is spawned, we record the current root set of the current active primary task; these are called the *snapshotted roots*. Then, we rely on a *SATB write barrier* to additionally remember any object which becomes unreachable due to an in-place update by an active primary task. The only remaining additional roots needed are those corresponding to down-pointers and right-pointers; these are remembered explicitly. Note that any incoming left-pointers and up-pointers are covered by the snapshot.

Tracing. For a CGC-heap h, let R_h be the set of snapshotted roots of the heap, and let D_h be the set of down- and right-pointers (x, y) where $H(x) \neq h$ and $H(y) = h$. The additional roots due to down- and right-pointers are $RD_h = \{y \mid (x, y) \in D_h\}$. The CGC tracing algorithm begins with the initial set of survivors $S_h \leftarrow R_h \cup RD_h$. Tracing proceeds as follows.

1. Pick a pointer (x, y) where $x \in S_h$ and $y \notin S_h$ and $H(y) = h$. (If there are no such pointers, tracing is complete.)
2. Insert y into S_h.
3. Repeat.

When tracing completes, the set S_h contains all live in-scope objects. After tracing, CGC completes by reclaiming the objects $\{x \in h \mid x \notin S_h\}$.

SATB write barrier. Here, we assume that the SATB write-barrier (executed by concurrent primary tasks) inserts objects directly into CGC survivor sets before any pointer is overwritten. In particular, consider an in-place update of an object x which is about to overwrite the pointer (x, y). The SATB write-barrier checks if $H(x)$ is a CGC-heap, and also if $H(x) = H(y)$. If both conditions hold, then the write-barrier inserts y into $S_{H(x)}$, i.e., the current CGC survivor set of the CGC-heap $H(x)$. This will cause the CGC tracing procedure to trace y as well as all objects reachable from y within heap $H(x)$.

In Section 6.7, we describe the SATB write-barrier in more detail, especially in regards to managing the concurrency between the CGC-task and the write-barrier. For the algorithm above, it suffices to assume that the write-barrier executes atomically with respect to one step of tracing.

4.5.5 CGC Scheduling

By placing CGCs off the critical path, we ensure that the work of CGC does not interfere with parallelism. However, because CGC-tasks are placed off the critical path, it is possible that the scheduler will accidentally ignore CGC-tasks, leading to an unbounded delay in collection. To ensure this does not occur, we propose a mild prioritization of CGC-tasks in the scheduler as follows. Immediately after each join (where two child heaps are merged into a parent heap and the primary parent task is resumed), the scheduler checks if there is CGC-task available to be scheduled in the CGC-chain of the parent. If there is, then the scheduler immediately schedules the CGC-task on a processor; otherwise, the scheduler continues as normal. This approach ensures that the responsiveness of CGC-tasks is bounded.

4.6 Collection Policy

Utilizing LGC and CGC techniques, any heap in the hierarchy can be collected independently, and individual garbage collections can proceed concurrently with—and in parallel with—other parallel tasks and garbage collections elsewhere in the heap hierarchy. In this way, LGC and CGC can be seen as building blocks which can be used and adapted for a variety of garbage collection policies.

For our implementation in Chapter 6, we describe a specific policy which is provably work-efficient, and which utilizes both LGC (on the critical path) and CGC (off the critical path). This policy is described in detail in Section 6.7.3, where we also analyze its work-efficiency. We also note that, based on the work in this thesis, Arora et al. [18] developed a collection policy which is provably efficient in terms of both work- and space-bounds. One limitation of that work, however, is that it effectively places CGC on the critical path, which increases the *span*, a.k.a., critical path length. In future work, we aim to design a garbage collection policy which offers simultaneous guarantees for work, space, and span.

Chapter 5

Entanglement Detection

In the presence of entanglement, the garbage collection techniques of Chapter 4 might reclaim an object incorrectly by missing a cross-pointer. To avoid this unsafe behavior, we enforce disentanglement with a dynamic approach, where individual memory accesses are monitored during execution, and if entanglement is detected, then the program is (safely) terminated. This allows for all disentangled programs to run to completion, including those that are effectful and/or non-deterministic. We make these guarantees precise by formulating soundness and completeness properties (Theorems 2 and 3). Roughly speaking, soundness (a.k.a. "no missed alarms") says that if entanglement is not detected, then the execution is disentangled; similarly, completeness (a.k.a. "no false alarms") says that if execution is disentangled, then entanglement is not detected.

It is important to note that, as a dynamic approach, entanglement detection is execution-dependent. If an execution of a program exhibits entanglement, then we detect the entanglement and terminate. But on the exact same program—even on the same input—it is possible that a different execution might *not* exhibit entanglement, in which case we allow the execution to complete. This is possible because the outcome of a race condition might determine whether or not entanglement occurs during execution. Therefore, although our approach handles entanglement safely, a shortcoming is that we cannot prevent the *possibility* of entanglement. An important problem for future work is to give good diagnostics when entanglement is detected, to facilitate debugging.

Because entanglement detection occurs dynamically and affects runtime performance, it is essential that it can be made efficient and scalable. Our approach takes inspiration from a long line of work on dynamic race detection for parallel programs [24, 50, 63, 64, 104, 122, 123, 151, 157]. While race detection remains expensive in practice (with overheads exceeding an order of magnitude for sequential runs, e.g., [151, 157]), we show that entanglement can be detected dynamically on-the-fly with close to zero overhead in practice. This is due to a number of differences between data races and entanglement; for example, unlike typical race detectors, our entanglement detector does not need to maintain an "access history" for each individual memory location. We defer a more detailed discussion of differences between the two techniques to Section 9.2.

5.1 Overview

We begin by considering a language with (nested) fork-join parallelism and mutable references, and present a dynamic semantics that checks for entanglement by monitoring accesses to references (mutable objects) only. The dynamic semantics constructs a *computation graph* of the execution that represents the parallel tasks and their dependencies in terms of *fork* and *join* edges. To detect entanglement, the semantics checks that each object accessed is allocated by a task that precedes the current task in the computation graph.

We prove soundness and completeness for the semantics (Theorems 2 and 3), thereby establishing that to detect entanglement, it suffices to track operations on mutable objects. Notably, the semantics incurs no "overhead" for immutable data, even when such data is reachable through a mutable object. Because the only operations monitored by the semantics are dereferences, we are able to prove that the work overhead of entanglement detection is proportional to the number of dereference operations.

For entanglement checks, the semantics associates each allocated object with a vertex in the computation graph. Naïvely, this would require $O(N)$ additional space for N heap objects, to store one vertex identifier per object. To reduce this space cost, we show how to group allocations by sharing a single vertex identifier amongst many objects allocated by the same task. This reduces the additional space cost from $O(N)$ down to approximately $O(\min(N, M/B))$, where M is the total size of memory and B is a chunking factor. The quantity M/B therefore represents the number of "heap chunks" used to store objects, which is typically much smaller than N in practice.

Our semantics paves the way for an implementation, albeit an inefficient one. The idea is to represent the computation graph using a well-known "series-parallel order maintenance" data structure for checking the precedence relation needed for entanglement checks. Series-parallel order maintenance, or *SP-order maintenance* for short, is well-studied in the race detection literature and many solutions can achieve efficiency and scalability [24, 50, 63, 64, 104, 122, 123, 151, 157]. In practice, however, the constant factors for precedence queries are significant: we completed such a direct implementation and measured that it can incur as much as 2x overhead, primarily due to the cost of precedence queries.

As a final step, we optimize away many of the SP-order maintanence operations by observing that typically, only a small number of mutable objects can lead to entanglement at any moment. We refer to such objects as *entanglement candidates*. Throughout execution, we explicitly track the set of candidates and only perform graph queries on these objects; all queries on others are pruned away. We prove that this optimization does not lead to an asymptotic impact on our bounds in the worst case, and show empirically that it can dramatically improve efficiency and scalability by eliminating many SP-order maintenance operations.

5.2 Entanglement and Determinacy Races

In a disentangled program, tasks are not permitted to become ***entangled***: no task is allowed to obtain a pointer to an object that was allocated by a concurrent task. Entanglement is always caused by a determinacy race (Theorem 1). However, as we discussed previously in Section 2.3,

```
1  // initialize strings A[i..j] (exclusive at j)
2  fun init(A: string array, i: int, j: int) =
3    if j − i = 0 then () else
4    if j − i = 1 then A[i] := Int.toString i else
5    let val m = ⌊(i + j)/2⌋
6    in (init(A, i, m) || init(A, m, j)); ()
7    end
8
9  val A = Array.allocate 3
10 val _ = init(A, 0, 3)
11 val (x, y) = (A[0] ^ A[2] || A[1] ^ A[2])
12 val _ = print (x ^ y)
```

Figure 5.1: Example disentangled program.

some determinacy races are compatible with disentanglement. In particular, when all communication utilizes only pre-allocated memory (i.e. memory allocated by common ancestors in the fork-join task tree), then disentanglement is still guaranteed. This allows determinacy races to be utilized in a disentangled manner, which can be useful for efficiency in practice.

Figure 5.1 presents an example of a small program that, as presented, is disentangled. We then consider multiple variations on the example which create different combinations of determinacy races and (dis)entanglement. Our goal here is to illustrate the nuances of disentanglement, including its interaction with determinacy races, which can be tricky to reason about.

The code in Figure 5.1 operates on an array of strings, where each string is heap-allocated and immutable. We write $(e_1 \parallel e_2)$ to execute e_1 and e_2 in parallel, wait for both to complete, and return their results as a tuple. The operation `^` denotes string concatenation.

The example defines a function `init` (lines 2-7) which in parallel initializes an array A between two indices i and j by storing a freshly allocated string at each index. On line 10, the example calls `init` on an array of size 3, which results in contents `["0","1","2"]`. It then in parallel concatenates a few elements of the array, resulting in x =`"02"` and y =`"12"` (line 11). Finally, it concatenates x and y and prints out `"0212"` (line 12). As written, the code is free of determinacy races.

Example: race-free and disentangled. As presented in Figure 5.1, this code is disentangled. There are multiple ways we could go about showing this. One way is to observe that the code is determinacy-race-free, which ensures disentanglement (Theorem 1). Another approach is to consider all of the allocations that occur in the computation, and where each allocated objects is used. The allocations of this computation include: the array A, the three strings stored in the array (at each $A[i]$), and the two strings allocated in parallel on line 11. The array A is allocated before everything else in the computation, so it is always safe to use. The strings stored in the array are allocated by the calls `init`$(0, 1)$, `init`$(1, 2)$, and `init`$(2, 3)$ which are respectively performed by three parallel tasks. These tasks perform no reads on the array and only update disjoint indices, so none of them acquires a cross-reference. Next, on line 11, the three strings within A are used, but this is safe, because all tasks from within the call to `init` on the previous

Variables	x, f		
Numbers	n	$\in$	$\mathbb{N}$
Memory Locations	ℓ		
Types	τ	$::=$	$\mathsf{nat} \mid \tau \times \tau \mid \tau \to \tau \mid \tau\ \mathsf{ref}$
Storable Values	s	$::=$	$n \mid \mathsf{fun}\ f\ x\ \mathsf{is}\ e \mid \langle \ell, \ell \rangle \mid \mathsf{ref}\ \ell$
Expressions	e	$::=$	$\ell \mid s \mid x \mid e\ e \mid \langle e, e \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{ref}\ e \mid !\,e \mid e := e \mid \langle e \parallel e \rangle$
Memory	μ	$\in$	*Locations* $\rightharpoonup$ *Storable Values*
Allocation Map	α	$\in$	*Locations* $\rightharpoonup$ *Vertices*
Task Tree	T	$::=$	$\mathsf{Leaf}(v) \mid \mathsf{Par}(v, T, T)$
Vertices	u, v, w		
Computation Graphs	G		
Program State	S	$::=$	$(\mu\ ;\alpha\ ; G\ ; T\ ; e) \mid \mathsf{ERROR}(\mu\ ; \alpha\ ; G\ ; T\ ; e)$

Figure 5.2: Syntax

line are guaranteed to complete before line 11. Similarly, line 12 is permitted to use the two strings allocated on line 11 for the same reason.

Example: racy and disentangled. It is possible to change Figure 5.1 so that the program has a determinacy race but is still disentangled. In particular, consider replacing line 11 with the following.

val $((x, y), _) = ((A[0]\ \hat{}\ A[2] \parallel A[1]\ \hat{}\ A[2]) \parallel A[2] := A[0])$

This code has a race which causes the reads at $A[2]$ to return either `"0"` or `"2"`. Nevertheless, it is disentangled, because both of these strings were allocated by preceding tasks (namely, from within the call to `init`, which completes before line 11 begins).

Example: racy and entangled. It is also possible to change Figure 5.1 so that the program is entangled due to a determinacy race. Consider replacing line 11 with the following (where the intent is that the string `"!"` is allocated dynamically).

val $((x, y), _) = ((A[0]\ \hat{}\ A[2] \parallel A[1]\ \hat{}\ A[2]) \parallel A[2] := \text{“!”})$

This introduces a third task which allocates a string `"!"` and writes it at $A[2]$, causing a determinacy race: in some executions x will be `"02"` but in others it will be `"0!"` (and similarly for y). This change makes the program entangled, because the tasks which read the contents of $A[2]$ might obtain a pointer to the string `"!"`, which is allocated by a concurrent task.

5.3 Language and Graph Semantics

At a high level, the idea for entanglement detection is to only check dereferences of mutable data (i.e., **ref** cells). We present the detection algorithm by embedding it in the operational semantics of a small ML-like language with (nested) fork-join parallelism, mutable references,

Execution with Entanglement Detection $\boxed{S \longmapsto S'}$

$$\frac{\ell \notin \mathrm{dom}(\mu)}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, s) \longmapsto (\mu[\ell \hookrightarrow s] \,;\, \alpha[\ell \hookrightarrow v] \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \ell)}\ \textsc{Alloc}$$

$$\frac{\mu(\ell_1) = \mathsf{fun}\ f\ x\ \mathsf{is}\ e}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \ell_1\ \ell_2) \longmapsto (\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, [\ell_1, \ell_2 \,/\, f, x]e)}\ \textsc{App}$$

$$\frac{\mu(\ell) = \langle \ell_1, \ell_2 \rangle}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \mathsf{fst}\ \ell) \longmapsto (\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \ell_1)}\ \textsc{Fst}$$

$$\frac{\mu(\ell) = \langle \ell_1, \ell_2 \rangle}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \mathsf{snd}\ \ell) \longmapsto (\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \ell_2)}\ \textsc{Snd}$$

$$\frac{\mu(\ell) = \mathsf{ref}\ \ell' \qquad \alpha(\ell') \preccurlyeq_G v}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, !\,\ell) \longmapsto (\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \ell')}\ \textsc{Bang-Pass}$$

$$\frac{\mu(\ell) = \mathsf{ref}\ \ell' \qquad \alpha(\ell') \not\preccurlyeq_G v}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, !\,\ell) \longmapsto \mathsf{ERROR}(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \ell')}\ \textsc{Bang-Detect}$$

$$\frac{}{(\mu[\ell_1 \hookrightarrow \mathsf{ref}\ _] \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \ell_1 := \ell_2) \longmapsto (\mu[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2] \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(v) \,;\, \ell_2)}\ \textsc{Upd}$$

$$\frac{v, w \notin \mathsf{Vertices}(G) \qquad G' = \mathit{fork}(G, u, v, w)}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Leaf}(u) \,;\, \langle e_1 \parallel e_2 \rangle) \longmapsto (\mu \,;\, \alpha \,;\, G' \,;\, \mathsf{Par}(u, \mathsf{Leaf}(v), \mathsf{Leaf}(w)) \,;\, \langle e_1 \parallel e_2 \rangle)}\ \textsc{Fork}$$

$$\frac{w \notin \mathit{vertices}(G) \qquad G' = \mathit{join}(G, u, v, w)}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Par}(_, \mathsf{Leaf}(u), \mathsf{Leaf}(v)) \,;\, \langle \ell_1 \parallel \ell_2 \rangle) \longmapsto (\mu \,;\, \alpha \,;\, G' \,;\, \mathsf{Leaf}(w) \,;\, \langle \ell_1, \ell_2 \rangle)}\ \textsc{Join}$$

$$\frac{(\mu \,;\, \alpha \,;\, G \,;\, T_1 \,;\, e_1) \longmapsto (\mu' \,;\, \alpha' \,;\, G' \,;\, T_1' \,;\, e_1')}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Par}(v, T_1, T_2) \,;\, \langle e_1 \parallel e_2 \rangle) \longmapsto (\mu' \,;\, \alpha' \,;\, G' \,;\, \mathsf{Par}(v, T_1', T_2) \,;\, \langle e_1' \parallel e_2 \rangle)}\ \textsc{ParL}$$

$$\frac{(\mu \,;\, \alpha \,;\, G \,;\, T_2 \,;\, e_2) \longmapsto (\mu' \,;\, \alpha' \,;\, G' \,;\, T_2' \,;\, e_2')}{(\mu \,;\, \alpha \,;\, G \,;\, \mathsf{Par}(v, T_1, T_2) \,;\, \langle e_1 \parallel e_2 \rangle) \longmapsto (\mu' \,;\, \alpha' \,;\, G' \,;\, \mathsf{Par}(v, T_1, T_2') \,;\, \langle e_1 \parallel e_2' \rangle)}\ \textsc{ParR}$$

Figure 5.3: Execution with entanglement detection (main computation steps).

Execution with Entanglement Detection (cont.) $\boxed{S \longmapsto S'}$

$$\frac{(\mu\,;\alpha\,;G\,;T\,;e_1) \longmapsto (\mu'\,;\alpha'\,;G'\,;T'\,;e_1')}{(\mu\,;\alpha\,;G\,;T\,;e_1\ e_2) \longmapsto (\mu'\,;\alpha'\,;G'\,;T'\,;e_1'\ e_2)}\ \textsc{App-SL}$$

$$\frac{(\mu\,;\alpha\,;G\,;T\,;e_1) \longmapsto \mathsf{ERROR}(\mu'\,;\alpha'\,;G'\,;T'\,;e_1')}{(\mu\,;\alpha\,;G\,;T\,;e_1\ e_2) \longmapsto \mathsf{ERROR}(\mu'\,;\alpha'\,;G'\,;T'\,;e_1'\ e_2)}\ \textsc{App-SLE}$$

$$\frac{(\mu\,;\alpha\,;G\,;T\,;e_2) \longmapsto (\mu'\,;\alpha'\,;G'\,;T'\,;e_2')}{(\mu\,;\alpha\,;G\,;T\,;\ell_1\ e_2) \longmapsto (\mu'\,;\alpha'\,;G'\,;T'\,;\ell_1\ e_2')}\ \textsc{App-SR}$$

$$\frac{(\mu\,;\alpha\,;G\,;T\,;e_2) \longmapsto \mathsf{ERROR}(\mu'\,;\alpha'\,;G'\,;T'\,;e_2')}{(\mu\,;\alpha\,;G\,;T\,;\ell_1\ e_2) \longmapsto \mathsf{ERROR}(\mu'\,;\alpha'\,;G'\,;T'\,;\ell_1\ e_2')}\ \textsc{App-SRE}$$

... and similarly for pairs (Pair-SL, Pair-SR, Pair-SLE, Pair-SRE) and updates (Upd-SL, Upd-SR, Upd-SLE, Upd-SRE)

$$\frac{(\mu\,;\alpha\,;G\,;T\,;e) \longmapsto (\mu'\,;\alpha'\,;G'\,;T'\,;e')}{(\mu\,;\alpha\,;G\,;T\,;\mathsf{fst}\ e) \longmapsto (\mu'\,;\alpha'\,;G'\,;T'\,;\mathsf{fst}\ e')}\ \textsc{Fst-S}$$

$$\frac{(\mu\,;\alpha\,;G\,;T\,;e) \longmapsto \mathsf{ERROR}(\mu'\,;\alpha'\,;G'\,;T'\,;e')}{(\mu\,;\alpha\,;G\,;T\,;\mathsf{fst}\ e) \longmapsto \mathsf{ERROR}(\mu'\,;\alpha'\,;G'\,;T'\,;\mathsf{fst}\ e')}\ \textsc{Fst-SE}$$

... and similarly for second projection (Snd-S, Snd-SE), refs (Ref-S, Ref-SE), and dereferences (Bang-S, Bang-SE).

$$\frac{(\mu\,;\alpha\,;G\,;T_1\,;e_1) \longmapsto \mathsf{ERROR}(\mu'\,;\alpha'\,;G'\,;T_1'\,;e_1')}{(\mu\,;\alpha\,;G\,;\mathsf{Par}(v,T_1,T_2)\,;\langle e_1 \parallel e_2\rangle) \longmapsto \mathsf{ERROR}(\mu'\,;\alpha'\,;G'\,;\mathsf{Par}(v,T_1',T_2)\,;\langle e_1' \parallel e_2\rangle)}\ \textsc{ParLE}$$

... and similarly for par-right step (ParRE).

Figure 5.4: Execution with entanglement detection (administrative rules).

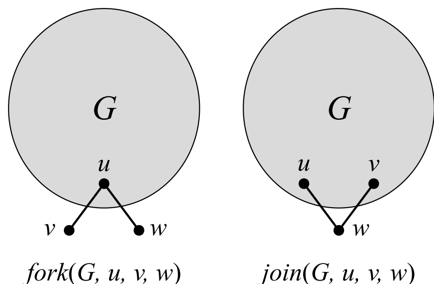

Figure 5.5: Functions *fork* and *join* on computation graphs.

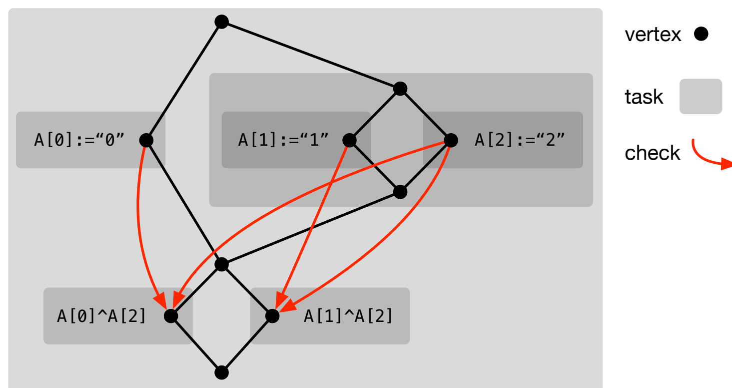

Figure 5.6: Example dag and entanglement checks for the disentangled program in Figure 5.1.

and a shared memory. This language is similar to that of Chapter 3: it has identical expressions and values, and it similarly maintains a dynamic task tree and constructs a computation graph. The computation graph is used in this setting to check for entanglement.

The syntax of the language is shown in Figure 5.2, and the dynamics are given in Figure 3.3. The operational semantics is a small-step semantics of the form $S \longmapsto S'$ where a single program state consists of five components: a memory μ, an allocation map α, a computation graph G, a task tree T, and an expression e. There is also an explicit error state, written ERROR$(\mu;\alpha;G;T;e)$, which indicates when entanglement has been detected (see Section 5.3.2). Computation graphs and task trees are discussed in Section 5.3.1. The allocation map α is used for detection, and is discussed in Section 5.3.2. The memory μ is used to map memory locations to their contents. An explicit allocation step (rule Alloc) extends the memory μ, producing $\mu[\ell \hookrightarrow s]$ where ℓ is a fresh location and s is the contents of that location. In this way, storable values always "take one more step". Memory locations are the only irreducible term of the language.

5.3.1 Parallelism, Task Trees, and Computation Graphs (Dags)

Similar to Chapter 3, the language here supports nested fork-join parallelism with a "parallel tuple" expression $\langle e_1 \parallel e_2 \rangle$, which runs e_1 and e_2 in parallel, waits for both to complete, and finally evaluates to a tuple of their results. To identify which parallel tuples are currently being evaluated, the semantics maintains a ***task tree*** T, which can either be a leaf of the form $\mathsf{Leaf}(v)$, or an internal "par" node denoted $\mathsf{Par}(v, T_1, T_2)$. The elements v stored in the task tree are vertices for the computation graph, described below.

In rule Fork, a parallel tuple begins execution by transitioning from a leaf in the task tree to a par-node, with two leaves as children. Then, steps may occur either on the left or the right non-deterministically via rules ParL and ParR. Eventually, when both child tasks have completed, rule Join transitions back to a leaf in the task tree (resuming the execution of the parent task) and converts the parallel tuple to a standard tuple.

Computation Graphs (Dags). Throughout execution, the semantics maintains a directed, acyclic graph (a.k.a., dag) which summarizes the execution and is used to check for entanglement. Dags are extended at each fork and join using a “current vertex” stored in the task tree. In rule Fork where a leaf task currently has vertex u, the function $\mathit{fork}(G, u, v, w)$ extends graph G with two new vertices v and w for the child tasks, and two edges (u, v) and (u, w), indicating that the children began executing after vertex u. Symmetrically, in rule Join where two leaf siblings have currently have vertices u and v, the function $\mathit{join}(G, u, v, w)$ extends graph G with one new vertex w and edges (u, w) and (v, w), indicating that the continuation (at w) began executing after the children completed. This maintenance of graphs is illustrated in Figure 5.5.

In a dag G, we say that vertex u ***precedes*** vertex v, denoted $u \preccurlyeq_G v$, if there exists a path in the dag from u to v. Note that $\preccurlyeq$ is a partial order. For any pair of vertices u and v, if $u \not\preccurlyeq_G v$ and also $v \not\preccurlyeq_G u$, then we say that u and v are ***concurrent***.

Relationship with Chapter 3. The presentation of task trees and computation graphs in this chapter is similar to that of Chapter 3, but differs slightly. In Chapter 3, the notion of an *open computation graph* encodes both the task tree and the dag simultaneously, in the same structure. Here, we instead choose to separate the two. We leave the implementation of the computation graph abstract, and rely on abstract queries of the form $u \preccurlyeq_G v$ on dags G. This is intentional: the implementation of the computation graph will ultimately be handled by an *SP-order maintenace* structure (see Section 6.8) which allows for efficient updates and queries on the graph. Our detection algorithm is agnostic to the implementation details of computation graphs and queries, and therefore we choose to leave the computation graph abstract. This approach also elucidates the overhead of entanglement detection, by making graph queries explicit in the semantics.

5.3.2 Entanglement Detection

Entanglement occurs when a task acquires a memory location that was allocated by a concurrent task. To detect entanglement, the semantics tags each memory location ℓ with a vertex $\alpha(\ell)$, indicating where in the computation the location was allocated. This occurs in rule Alloc, where both the memory μ and the map α are extended with the new location (guaranteeing $\mathrm{dom}(\mu) = \mathrm{dom}(\alpha)$). When dereferencing a mutable reference, we check for entanglement by inspecting the *result* of the read. Specifically, when a read at location ℓ returns some other location ℓ', we compare $\alpha(\ell')$ against the current vertex v. If $\alpha(\ell') \preccurlyeq_G v$, then this access is safe for disentanglement and the execution may proceed with rule Bang-Pass. However, if $\alpha(\ell') \not\preccurlyeq_G v$, then we have detected entanglement (rule Bang-Detect).

In short, entanglement is detected whenever rule Bang-Detect is used during execution, which results in a stuck program state: any state of the form $\mathsf{ERROR}(\mu\,;\alpha\,;G\,;T\,;e)$ cannot step. In this way, the semantics terminates an execution as soon as entanglement is detected. All other steps allow the computation to proceed as normal.

Note that immutable reads are never checked. For example, in rule Fst, we could have included $\alpha(\ell_1) \preccurlyeq_G v$ as one of the premises, but it is intentionally left out. This is because, as we will discuss more carefully in Section 5.4, reads of immutable data are always safe for disentanglement.

5.3.3 Example Revisited

The dag in Figure 5.6 summarizes the execution of the example program from Figure 5.1. Each black circle is a vertex, and the solid black edges (implicitly pointing down) are execution dependencies between vertices. The tasks of the computation are illustrated as shaded gray boxes, such that at any moment throughout execution, the nesting of the shaded gray boxes represents the task tree. We draw red, curved arrows to summarize where entanglement checks occur. A red curved arrow from u to v indicates that a mutable dereference was performed at v, returning some object allocated at u. The intent here is that arrays operate analogously to mutable references: indexing into an array checks the resulting object for entanglement in the same manner as rules Bang-Pass and Bang-Detect of the semantics. For example, reading $A[0]$ and $A[2]$ (Figure 5.1, line 11) discovers the strings `"0"` and `"2"` which are checked to ensure both were allocated previously in the computation (i.e. not by a concurrent task). The example program is disentangled, and indeed in Figure 5.6 we see that entanglement is never detected, because for every red curved edge from u to v, there is a path $u \preccurlyeq v$ in the dag.

5.4 Soundness and Completeness

In our approach, entanglement is considered to have been detected whenever rule Bang-Detect is used in an execution. In this setting, soundness can be stated as a preservation property for disentanglement, i.e., that steps taken without detecting entanglement preserve disentanglement. Similarly, completeness is the property that, if disentanglement is preserved by a step, then entanglement is not detected. In other words, soundness is “no missed alarms”, and completeness is “no false alarms”.

The disentanglement invariant, written S de, is defined in Figure 5.7.[1] It consists of two components: root disentanglement and memory disentanglement.

The ***root disentanglement*** judgement, written $\alpha\,;G\,;T\,;e$ rootsde, establishes that each task only currently uses locations allocated at or before its associated vertex. We state this formally in terms of $\mathsf{locs}(e)$, the set of locations mentioned directly by an expression, defined in the natural way: for example, $\mathsf{locs}(e_1\ e_2) = \mathsf{locs}(e_1) \cup \mathsf{locs}(e_2)$ and $\mathsf{locs}(\ell) = \{\ell\}$. The judgement $\alpha\,;G\,;T\,;e$ rootsde is then established inductively on both the structure of e and the task tree T, where the task tree is used to delimit each task. Observe for example that at each leaf task, we have $\forall \ell \in \mathsf{locs}(e).\ \alpha(\ell) \preccurlyeq_G v$, i.e. that every location ℓ which is mentioned by the expression was allocated before the current vertex v in the computation.

The ***memory disentanglement*** judgement, written $\mu\,;\alpha\,;G$ memde, establishes that each immutable location of the memory only points “backwards” in the computation. For example, if ℓ stores the tuple $\langle \ell_1, \ell_2 \rangle$ then $\alpha(\ell_1) \preccurlyeq_G \alpha(\ell)$ and similarly for ℓ_2. Thus, if it is safe to access ℓ then it is similarly safe to access the contents of ℓ. This formalizes our intuition that immutable data is always safe for disentanglement. In contrast, if ℓ stores a mutable reference ref ℓ', then ℓ' might have been written there by a concurrent task, hence why rule Bang-Pass and rule Bang-Detect need to check this explicitly.

[1] The definition here is similar to the disentanglement definitions of Chapter 3, but now appropriately rephrased in terms of abstract computation graphs and explicit task trees.

Disentanglement $\boxed{S \ \text{de}}$

$$\frac{\mu ; \alpha ; G \ \ \text{memde} \qquad \alpha ; G ; T ; e \ \ \text{rootsde}}{(\mu ; \alpha ; G ; T ; e) \ \ \text{de}} \qquad \frac{\mu ; \alpha ; G \ \ \text{memde} \qquad \alpha ; G ; T ; e \ \ \text{rootsde}}{\text{ERROR}(\mu ; \alpha ; G ; T ; e) \ \ \text{de}}$$

Memory Disentanglement $\boxed{\mu ; \alpha ; G \ \ \text{memde}}$

$$\frac{\forall \ell \in \text{imm}(\mu).\ \forall \ell' \in \text{locs}(\mu(\ell)).\ \alpha(\ell') \preccurlyeq_G \alpha(\ell)}{\mu ; \alpha ; G \ \ \text{memde}} \qquad \text{imm}(\mu) \triangleq \{\ell \in \text{dom}(\mu) \mid \mu(\ell) \neq \text{ref } _\}$$

Root Disentanglement $\boxed{\alpha ; G ; T ; e \ \ \text{rootsde}}$

$$\frac{\forall \ell \in \text{locs}(e).\ \alpha(\ell) \preccurlyeq_G v}{\alpha ; G ; \text{Leaf}(v) ; e \ \ \text{rootsde}} \qquad \frac{\alpha ; G ; T_1 ; e_1 \ \ \text{rootsde} \qquad \alpha ; G ; T_2 ; e_2 \ \ \text{rootsde}}{\alpha ; G ; \text{Par}(v, T_1, T_2) ; \langle e_1 \parallel e_2 \rangle \ \ \text{rootsde}}$$

$$\left.\frac{\alpha ; G ; \text{Par}(v, T_1, T_2) ; e \ \ \text{rootsde}}{\alpha ; G ; \text{Par}(v, T_1, T_2) ; (\text{fst } e) \ \ \text{rootsde}}\right\} \textit{...similarly for } (\text{snd } e), (\text{ref } e), \textit{ and } (!\, e)$$

$$\left.\begin{array}{c} \dfrac{\begin{array}{c}\neg(e_1 \text{ loc}) \\ \alpha ; G ; \text{Par}(v, T_1, T_2) ; e_1 \ \ \text{rootsde} \qquad \forall \ell \in \text{locs}(e_2).\ \alpha(\ell) \preccurlyeq_G v\end{array}}{\alpha ; G ; \text{Par}(v, T_1, T_2) ; (e_1\ e_2) \ \ \text{rootsde}} \\[2ex] \dfrac{\alpha(\ell_1) \preccurlyeq_G v \qquad \alpha ; G ; \text{Par}(v, T_1, T_2) ; e_2 \ \ \text{rootsde}}{\alpha ; G ; \text{Par}(v, T_1, T_2) ; (\ell_1\ e_2) \ \ \text{rootsde}} \end{array}\right\} \begin{array}{l}\textit{...similarly for } \langle e_1, e_2 \rangle \\ \textit{and } (e_1 := e_2)\end{array}$$

Figure 5.7: Single-step disentanglement invariant, consisting of memory property for all immutable locations, and disentanglement property for all program "roots".

Note that S de is defined for both regular program states as well as ERROR states. This makes it possible to identify any state as disentangled, regardless of whether or not the dynamic semantics claims to have detected entanglement, which is key to the statement of the completeness theorem, below.

To state soundness and completeness, we identify “no-error” program states, which are those in which the dynamic semantics has *not* detected entanglement.

$$\frac{}{(\mu\,;\alpha\,;G\,;T\,;e)\ \ \mathsf{noerror}}$$

The soundness and completeness theorems (Theorems 2 and 3) are then given in terms of single steps. Roughly speaking, soundness says that if entanglement is not detected, then the step is disentangled; similarly, completeness says that if the step is disentangled, then entanglement is not detected.

Theorem 2 (Soundness)**.** If S de and $S \longmapsto S'$ and S' noerror, then S' de.

Theorem 3 (Completeness)**.** If S de and $S \longmapsto S'$ and S' de, then S' noerror.

5.4.1 Completeness Proof

Theorem 3 states: if S de and $S \longmapsto S'$ and S' de, then S' noerror. In the following proof, note that for any state $S' = \mathsf{ERROR}(\ldots)$, we have $\neg(S'$ noerror), and therefore a contradiction may be obtained by deriving a judgement of the form ERROR$(\ldots)$ noerror.

Proof. By induction on the derivation of $S \longmapsto S'$.

All cases with a right-hand side S' where S' noerror trivially satisfy the consequent and therefore are omitted. We now consider the remaining cases (i.e., those with a right-hand side $S' = \mathsf{ERROR}(\ldots)$).

Case Bang-Detect. We have $\mu\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;\,!\,\ell$ de and $\mu(\ell) = \ell'$ and $\alpha(\ell') \not\preccurlyeq_G v$. By assumption, this takes a step to where we have $\mu\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;\ell'$ de. This implies $\alpha\,;G\,;\mathsf{Leaf}(v)\,;\ell'$ rootsde which in turn yields $\alpha(\ell') \preccurlyeq_G v$. However, this contradicts the assumption $\alpha(\ell') \not\preccurlyeq_G v$. Therefore Bang-Detect does not appear in the derivation $S \longmapsto S'$.

The remainder of the cases all follow a similar structure, where the inductive hypothesis is used to derive a contradiction, thereby demonstrating that the corresponding rule does not appear in the derivation of $S \longmapsto S'$ in the antecedent of the theorem statement. As an example, consider case App-SLE. We have $\mu\,;\alpha\,;G\,;T\,;e_1\,e_2$ de and ERROR$(\mu'\,;\alpha'\,;G'\,;T'\,;e_1'\,e_2)$ de. From these we have $\mu\,;\alpha\,;G\,;T\,;e_1$ de and ERROR$(\mu'\,;\alpha'\,;G'\,;T'\,;e_1')$ de. Inductively, we have ERROR$(\mu'\,;\alpha'\,;G'\,;T'\,;e_1')$ noerror, which is a contradiction. The rest of the cases (App-SRE, Pair-SLE, Pair-SRE, Upd-SLE, Upd-SRE, Fst-SE, Snd-SE, Ref-SE, Bang-SE, ParLE, and ParRE) all proceed similarly.

□

5.4.2 Soundness Proof

The proof of the soundness result (Theorem 2) is more involved, as it amounts to showing that all uses of immutable data are always safe for disentanglement.

An interesting case in the soundness result is rule Alloc, which establishes the connection between the *memory disentanglement* and *root disentanglement* properties discussed above. When allocating a new (immutable) memory location, in order to satisfy the memory disentanglement property, we have to establish that the contents of the new location only point backwards, to other previously allocated locations. Because each of these other locations is a "root" of the current expression, we obtain this property from the root disentanglement invariant. Another interesting case in the proof is an immutable read. For example, in rule Fst, we have to re-establish the root disentanglement property for the result of the read. By appealing to memory disentanglement, which ensures that the contents of the immutable location only point "backwards" in the computation, we are able to do so. In this way, disentanglement of the memory and the roots work in tandem, enabling us to prove that immutable data never needs to be checked for entanglement.

Supporting Lemmas and Definitions

The soundness result relies on two supporting lemmas, Lemmas 8 and 9. Here, we write $\mathsf{vert}(T)$ for the "root vertex" of T, i.e. $\mathsf{vert}(\mathsf{Leaf}(u)) = u$ and $\mathsf{vert}(\mathsf{Par}(u, _, _)) = u$.

Lemma 8. When taking a step, the root vertex either stays the same or moves forward.

Formally, for any $\mu ; \alpha ; G ; T ; e \longmapsto^* \mu' ; \alpha' ; G' ; T' ; e'$, we have $\mathsf{vert}(T) \preccurlyeq_{G'} \mathsf{vert}(T')$.

Lemma 9. Taking a step extends the computation graph.

Formally, for any $\mu ; \alpha ; G ; T ; e \longmapsto^* \mu' ; \alpha' ; G' ; T' ; e'$ and $u \preccurlyeq_G v$, we have $u \preccurlyeq_{G'} v$.

Main proof

The soundness result, Theorem 2, states the following: if S de and $S \longmapsto S'$ and S' noerror, then S' de.

Proof. By induction on the derivation of $S \longmapsto S'$. All cases with the right-hand side $S' = \mathsf{ERROR}(\ldots)$ do not satisfy the the antecedent S' noerror and therefore are omitted. (For example, rule Bang-Detect.) The remaining cases follow.

Case Alloc. We have $\mu ; \alpha ; G ; \mathsf{Leaf}(v) ; s$ de. Need to show $\mu' ; \alpha' ; G ; \mathsf{Leaf}(v) ; \ell$ de where $\mu' = \mu[\ell \hookrightarrow s]$ and $\alpha' = \alpha[\ell \hookrightarrow v]$ and $\ell \notin \mathsf{dom}(\mu)$. By $\mu ; \alpha ; G ; \mathsf{Leaf}(v) ; s$ de we have $\forall \ell \in \mathsf{locs}(s).\ \alpha(\ell) \preccurlyeq_G v$ and therefore $\mu' ; \alpha' ; G$ memde. We also have $\alpha' ; G ; \mathsf{Leaf}(v) ; \ell$ rootsde because $\alpha'(\ell) = v$. Therefore $\mu' ; \alpha' ; G ; \mathsf{Leaf}(v) ; \ell$ de.

Case App-SL. We have $\mu ; \alpha ; G ; T ; (e_1\ e_2)$ de and therefore (for T either Leaf or Par) we have $\forall \ell \in \mathsf{locs}(e_2).\ \alpha(\ell) \preccurlyeq_G \mathsf{vert}(T)$ as well as $\mu;\alpha;G;T;e_1$ de. By induction, we have $\mu';\alpha';G';T';e_1'$ de and therefore both $\mu' ; \alpha' ; G'$ memde as well as $\alpha' ; G' ; T' ; e_1'$ rootsde. By Lemmas 8 and 9 we have $\forall \ell \in \mathsf{locs}(e_2).\ \alpha(\ell) \preccurlyeq_{G'} \mathsf{vert}(T) \preccurlyeq_{G'} \mathsf{vert}(T')$, and therefore (for T' either Leaf or Par) we have $\alpha' ; G' ; T' ; (e_1'\ e_2)$ rootsde. Altogether, these yield $\mu' ; \alpha' ; G' ; T' ; (e_1'\ e_2)$ de.

Cases Pair-SL and Upd-SL proceed similarly as App-SL.

Case App-SR. We have $\mu ; \alpha ; G ; T ; (\ell_1\ e_2)$ de and therefore (for T either Leaf or Par) we have $\alpha(\ell_1) \preccurlyeq_G \mathsf{vert}(T)$ as well as $\mu ; \alpha ; G ; T ; e_2$ de. By induction we have $\mu' ; \alpha' ; G' ; T' ; e_2'$ de and therefore both $\mu' ; \alpha' ; G'$ memde as well as $\alpha' ; G' ; T' ; e_2'$ rootsde. By Lemmas 8 and 9, we have $\alpha(\ell_1) \preccurlyeq_{G'} \mathsf{vert}(T')$ and therefore (for T' either Leaf or Par) we have $\alpha' ; G' ; T' ; (\ell_1\ e_2')$ rootsde which in turn yields $\mu' ; \alpha' ; G' ; T' ; (\ell_1\ e_2')$ de.

Cases Pair-SR and Upd-SR proceed similarly as App-SR.

Case App. We have $\mu ; \alpha ; G ; \mathsf{Leaf}(v) ; (\ell_1\ \ell_2)$ de where $\mu(\ell_1) = \mathsf{fun}\ f\ x\ \mathsf{is}\ e$, and therefore $\mu;\alpha;G$ memde as well as $\alpha;G;\mathsf{Leaf}(v);(\ell_1\ \ell_2)$ rootsde. From this we have $\alpha(\ell_1) \preccurlyeq_G v$ and also $\alpha(\ell_2) \preccurlyeq_G v$. Additionally, because $\ell_1 \in \mathsf{imm}(\mu)$, we have $\forall \ell' \in \mathsf{locs}(\mathsf{fun}\ f\ x\ \mathsf{is}\ e).\ \alpha(\ell') \preccurlyeq_G \alpha(\ell_1)$. The same holds for all $\ell' \in \mathsf{locs}(e)$ because $\mathsf{locs}(\mathsf{fun}\ f\ x\ \mathsf{is}\ e) = \mathsf{locs}(e)$. We therefore have $\forall \ell' \in \mathsf{locs}([\ell_1, \ell_2\ /\ f, x]e).\ \alpha(\ell') \preccurlyeq_G v$ because $\mathsf{locs}([\ell_1, \ell_2\ /\ f, x]e) \subseteq \mathsf{locs}(e) \cup \{\ell_1, \ell_2\}$ and previously we determined both $\alpha(\ell_1) \preccurlyeq_G v$ and $\alpha(\ell_2) \preccurlyeq_G v$. Therefore $\alpha;G;\mathsf{Leaf}(v);[\ell_1, \ell_2\ /\ f, x]e$ rootsde. Altogether, this yields $\mu ; \alpha ; G ; \mathsf{Leaf}(v) ; [\ell_1, \ell_2\ /\ f, x]e$ de.

Case Fst-S. We have $\mu ; \alpha ; G ; T ; (\mathsf{fst}\ e)$ de and therefore (for T either Leaf or Par) we have $\mu ; \alpha ; G ; T ; e$ de. By induction, we have $\mu' ; \alpha' ; G' ; T' ; e'$ de and therefore both $\mu';\alpha';G'$ memde as well as $\alpha';G';T';e'$ rootsde. Therefore (for T' either Leaf or Par) we have $\alpha' ; G' ; T' ; (\mathsf{fst}\ e')$ rootsde which in turn yields $\mu' ; \alpha' ; G' ; T' ; (\mathsf{fst}\ e')$ de.

Cases Snd-S, Ref-S, and Bang-S proceed similarly as Fst-S.

Case Fst. We have $\mu ; \alpha ; G ; \mathsf{Leaf}(v) ; (\mathsf{fst}\ \ell)$ de. Need to show $\mu ; \alpha ; G ; \mathsf{Leaf}(v) ; \ell_1$ de where $\mu(\ell) = \langle \ell_1, \ell_2 \rangle$. By $\mu ; \alpha ; G ; \mathsf{Leaf}(v) ; (\mathsf{fst}\ \ell)$ de we have $\mu ; \alpha ; G$ memde and therefore $\alpha(\ell_1) \preccurlyeq_G \alpha(\ell)$. By $\alpha ; G ; \mathsf{Leaf}(v) ; (\mathsf{fst}\ \ell)$ rootsde we have $\alpha(\ell) \preccurlyeq_G v$. Together, these imply $\alpha(\ell_1) \preccurlyeq_G v$, which in turn gives us $\alpha;G;\mathsf{Leaf}(v);\ell_1$ rootsde and therefore $\mu;\alpha;G;\mathsf{Leaf}(v);\ell_1$ de.

Case Snd proceeds symmetrically to Fst.

Case Bang-Pass. We have $\mu ; \alpha ; G ; \mathsf{Leaf}(v) ; (!\,\ell)$ de and $\mu(\ell) = \mathsf{ref}\ \ell'$ and $\alpha(\ell') \preccurlyeq_G v$. Need to show $\mu ; \alpha ; G ; \mathsf{Leaf}(v) ; \ell'$ de, which follows immediately from the assumptions.

Case Upd. Let μ_1 be the memory before the update (i.e. $\mu_1 = \mu[\ell_1 \hookrightarrow \mathsf{ref}\ _]$) and $\mu_2 = \mu[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2]$ be the memory after. We have $\mu_1;\alpha;G;\mathsf{Leaf}(v);(\ell_1 := \ell_2)$ de and therefore $\mu_1;\alpha;G$ memde as well as $\alpha(\ell_2) \preccurlyeq_G v$. Therefore we have $\alpha ; G ; \mathsf{Leaf}(v) ; \ell_2$ rootsde. Because $\ell \notin \mathsf{imm}(\mu_2)$, we also have $\mu_2 ; \alpha ; G$ memde. Together, these yield $\mu_2 ; \alpha ; G ; \mathsf{Leaf}(v) ; \ell_2$ de.

Case Fork. We have $\mu;\alpha;G;\mathsf{Leaf}(u);\langle e_1 \parallel e_2 \rangle$ de. Need to show $\mu;\alpha;G';\mathsf{Par}(u, \mathsf{Leaf}(v), \mathsf{Leaf}(w))$; $\langle e_1 \parallel e_2 \rangle$ de where $G' = \mathit{fork}(G, u, v, w)$. From $\alpha;G;\mathsf{Leaf}(u);\langle e_1 \parallel e_2 \rangle$ rootsde we have $\alpha(\ell) \preccurlyeq_G u$ for every $\ell \in \mathsf{locs}(e_1) \cup \mathsf{locs}(e_2)$. By the definition of forking, we have $u \preccurlyeq_G v$ and $u \preccurlyeq_G w$. Therefore $\alpha(\ell) \preccurlyeq_{G'} u \preccurlyeq_G v$ for every $\ell \in \mathsf{locs}(e_1)$, and similarly $\alpha(\ell) \preccurlyeq_G u \preccurlyeq_{G'} w$ for all ℓ in e_2. Together, these imply $\alpha ; G' ; \mathsf{Leaf}(v) ; e_1$ rootsde and $\alpha ; G' ; \mathsf{Leaf}(w) ; e_2$ rootsde which in turn yield $\alpha ; G' ; \mathsf{Par}(u, \mathsf{Leaf}(v), \mathsf{Leaf}(w)) ; \langle e_1 \parallel e_2 \rangle$ rootsde. We also have $\mu ; \alpha ; G$ memde and therefore $\mu ; \alpha ; G'$ memde since G' is an extension of G. Altogether, these imply $\mu ; \alpha ; G' ; \mathsf{Par}(u, \mathsf{Leaf}(v), \mathsf{Leaf}(w)) ; \langle e_1 \parallel e_2 \rangle$ de.

Case Join. We have $\mu ; \alpha ; G ; \mathsf{Par}(_, \mathsf{Leaf}(u), \mathsf{Leaf}(v)) ; \langle \ell_1 \parallel \ell_2 \rangle$ de and therefore $\alpha ; G ; \mathsf{Leaf}(u) ; \ell_1$ rootsde and $\alpha ; G ; \mathsf{Leaf}(v) ; \ell_2$ rootsde. Need to show $\mu ; \alpha ; G' ; \mathsf{Leaf}(w) ; \langle \ell_1, \ell_2 \rangle$ de where $G' = \mathit{join}(G, u, v, w)$. From $\alpha ; G ; \mathsf{Leaf}(u) ; \ell_1$ rootsde and we have $\alpha(\ell_1) \preccurlyeq_G u$, which by extension gives us $\alpha(\ell_1) \preccurlyeq_{G'} u$. By the definition of joining we have $u \preccurlyeq_{G'} w$, so $\alpha(\ell_1) \preccurlyeq_{G'} w$. Similarly, $\alpha(\ell_2) \preccurlyeq_G v \preccurlyeq_{G'} w$. Together these yield $\alpha ; G' ; \mathsf{Leaf}(w) ; \langle \ell_1, \ell_2 \rangle$ rootsde. We also have $\mu ; \alpha ; G$ memde and therefore $\mu ; \alpha ; G'$ memde by extension. Altogether, these imply $\mu ; \alpha ; G' ; \mathsf{Leaf}(w) ; \langle \ell_1, \ell_2 \rangle$ de.

Case ParL. We have $\mu;\alpha;G;\mathsf{Par}(v, T_1, T_2);\langle e_1 \parallel e_2 \rangle$ de and therefore both $\alpha;G;T_1;e_1$ rootsde and $\alpha ; G ; T_2 ; e_2$ rootsde as well as $\mu ; \alpha ; G$ memde. Therefore $\mu ; \alpha ; G ; T_1 ; e_1$ de. By induction we have $\mu' ; \alpha' ; G' ; T_1' ; e_1'$ de and therefore $\mu' ; \alpha' ; G'$ memde and $\alpha' ; G' ; T_1' ; e_1'$ rootsde. Altogether therefore we have $\mu' ; \alpha' ; G' ; \mathsf{Par}(v, T_1', T_2) ; \langle e_1' \parallel e_2 \rangle$ de.

Case ParR proceeds symmetrically to ParL.

□

5.5 Entanglement Detection Cost Analysis

We bound the work and space of our entanglement detection algorithm. Theorems 4 and 5 follow directly from the observation that entanglement detection only introduces three sources of overhead: maintaining the computation graph, mapping locations to vertices in the graph, and performing graph queries at dereference operations. The work of maintaining the computation graph can be charged to the overall work of the computation, as each step makes at most a constant number of additions to the computation graph. Mapping locations to vertices in the graph can be performed by storing vertex identifiers in memory along with the contents of each location. The remaining costs are isolated to details of how the computation graph is maintained and queried, so for the sake of brevity here, we leave these costs abstract. At a high level, the idea is to use SP-order maintenance, a well-studied problem with many solutions available "off-the-shelf" with low overhead in practice. More details about graph maintenance and queries are provided in Section 6.8.

Theorem 4 (Work of Entanglement Detection)**.** For a program with work (total number of steps) W, execution with entanglement detection requires $O(W + D \cdot W_q)$ work in total, where D is the number of dereference operations, and W_q is an upper bound on the work required for a graph query.

Theorem 5 (Space of Entanglement Detection)**.** At any point during execution, entanglement detection requires $O(N + S_g)$ additional space, where N is the current number of heap objects, and S_g is the current space required to maintain the computation graph. A similar bound holds for live (reachable) memory.

5.5.1 Utilizing Heap Chunks to Optimize Space

A common implementation strategy is to represent heaps as lists of "chunks", where each chunk is a fairly large region of contiguous memory (e.g. one or more pages). When heap chunks are task-local, we can significantly reduce the number of vertex labels stored for entanglement detection. In particular, our implementation (Chapter 6) guarantees that all objects within a chunk were allocated by the same task (or one of that task's completed subtasks). We can therefore assign one vertex identifier per chunk. For N heap objects using a total of M space, this reduces the amount of additional space needed from N (one vertex identifier per heap object) down to approximately $\min(N, M/B)$ (one vertex identifier per chunk). Because typical memory objects are small, and therefore $N \approx M$, this is a significant improvement. The following theorem formalizes the bound.

Theorem 6 (Chunked Space of Entanglement Detection)**.** Using task-local heap chunks, at any point during execution, entanglement detection requires $O(\min(N, M/B) + T + S_g)$ additional space, where M is the current total heap size, N is the number of heap objects, B is the minimum size of a heap chunk, T is the current number of active tasks, and S_g is the current space required to maintain the computation graph.

Proof. It suffices to bound the number of heap chunks. Consider a task t and let M_t be the size of t's local heap, split across k_t heap chunks, with N_t heap objects in those chunks. We assume bump-allocation within each chunk, which ensures that the amount of memory allocated in any two consecutive heap chunks is at least B. Hence, we have $k_t \leq 1 + 2M_t/B$. Also, because large objects are given their own chunks, we have $k_t \leq N_t$. Putting these two upper bounds together and summing over all active tasks yields a bound of $O(\min(N, M/B) + T)$ heap chunks. □

5.6 Entanglement Candidates

The entanglement detection semantics in Section 5.3 describes how to check whether an execution is entangled: for every dereference of a **ref** cell, perform a query on the computation graph (Figure 3.3, rules Bang-Pass and Bang-Detect). Here we show that a significant number of these queries can be pruned away dynamically, resulting in significant performance improvement in practice.

The high-level idea is to annotate each **ref** cell with a bit that indicates whether or not the **ref** requires a graph query when it is dereferenced, to check for entanglement. Any **ref** that is marked is called an ***entanglement candidate*** (or simply *candidate* for short). Throughout execution, **ref** cells are dynamically marked and unmarked; that is, a **ref** might at various points throughout execution be marked as a candidate and later unmarked when it returns to a safe state. On each dereference $!x$, we first check if x is marked as a candidate. If so, then we do a graph query to check for entanglement, consistent with the semantics. But if x is not marked, then we skip the graph query.

Below, we describe how candidates are marked and unmarked throughout execution (Section 5.6.1) and analyze the cost of this algorithm (Section 5.6.2). We first focus on **ref** objects, and then generalize our techniques to handle arrays (Section 5.6.3). Next, we provide some intuition for how candidate tracking can make a significant impact in practice (Section 5.6.4). Finally, in Section 5.6.5, we connect the idea of candidates with the semantics of Section 5.3 and argue that this technique is a valid optimization (i.e., it does not affect correctness of detection).

5.6.1 Marking and Unmarking Candidates

Objects are born safe. When a task first allocates a **ref**, at that moment, the **ref** can only contain data that the task already had access to. Therefore, each **ref** begins its life unmarked, indicating that the ref is safe (i.e., not a candidate). As long as a **ref** is never updated, it remains safe, similar to immutable data.

Marking candidates at updates. Intuitively, when a **ref** is updated, it might become a candidate and need to be marked. Here we leverage an observation about entanglement. Consider a reference x which currently contains a pointer to an object y, and suppose that a task becomes entangled by performing the dereference $!x$ and acquiring a pointer to y. At this moment, the pointer from x to y in memory must be a *down-pointer*, i.e., $H(x)$ must be an ancestor of $H(y)$ in the heap hierarchy (where $H(x)$ is the heap that contains object x; see Section 4.2). **In other words, in order to acquire a cross-pointer, a task must read a down-pointer.** Therefore,

any **ref** which contains a down-pointer must be marked as a candidate. We specifically mark candidates at updates: when a task performs an update $x := y$, if this creates a down-pointer from x to y, then the task marks x as a candidate. Note that if the pointer from x to y is not a down-pointer, then the **ref** is not marked as a candidate.

Unmarking candidates in leaf heaps. Any **ref** which does not contain a down-pointer is no longer a candidate and should be unmarked. To unmark a candidate x, we wait until $H(x)$ is a leaf heap (which occurs naturally due to tasks joining with their parents). At this point, because the heap has no descendants, we know that any pointer from x is no longer a down-pointer, and thus x is safe. Therefore, we unmark all candidates within a heap whenever a heap becomes a leaf. Specifically, whenever two completed tasks join, we unmark all candidates in their parent heap.

Another strategy for unmarking would be to detect when a down-pointer is overwritten with an internal or up-pointer. We choose not to use this strategy because it is not efficient for large mutable objects, such as `array` objects (described in Section 5.6.3). In particular, as long as an array contains at least one down-pointer, it must remain marked. Attempting to unmark arrays at individual updates would require counting the number of down-pointers dynamically, which would be prohibitively expensive.

Candidate sets. To facilitate unmarking candidates in bulk when a heap becomes a leaf, we give each heap a *candidate set* which is the set of objects within that heap that are currently marked as a candidate. When a candidate is marked, it is added to the corresponding set. To unmark candidates of a heap, we iterate through the candidate set and unmark each object individually. The candidate set is then cleared.

5.6.2 Cost Analysis of Tracking Candidates

Tracking candidates does not impact the asymptotic costs of execution and has low overhead. The space overhead of candidate sets is $O(1)$ per candidate, because each candidate is only stored in one candidate set. The algorithm takes constant work on each update, to mark a candidate and add it to a candidate set. At joins, it traverses the candidate set of a single heap to clear candidates. This incurs constant work per candidate, and can be charged to the cost of the update that marked the candidate. When an object is dereferenced, we incur constant overhead to first check if the object is a candidate.

Our algorithm for tracking candidates is conservative because the program may also delete a down-pointer (causing an object which is marked as a candidate to no longer have any down-pointers), but the algorithm does not attempt to track this. It would be prohibitively expensive to attempt to track individual down-pointer deletions, especially with candidate arrays (Section 5.6.3), which may have a large number of down-pointers at any moment. Our algorithm avoids this cost by unmarking candidates "lazily" at joins.

5.6.3 Candidate Arrays

Generalizing the above algorithm for mutable `array` objects is straightforward. Similar to **ref** cells, we give each array a single bit. If an update $a[i] := x$ creates a down-pointer from a to x, then we mark a as a candidate. Later, when the heap $H(a)$ becomes a leaf, we unmark a.

In this way, we (conservatively) only track whether or not the whole array is a candidate, rather than attempting to precisely track every index of the array separately, which would be costly in practice. For example, consider an array of pointers and suppose several tasks are adding and deleting down-pointers to it, in parallel. Determining when a particular delete removes the last down pointer from the array would be prohibitively expensive (it would require counting the number of down-pointers and updating this every time the array is mutated). Instead, our algorithm unmarks the array (as a candidate) after all descendant tasks complete. In this way, the algorithm only incurs constant overhead for the entire array.

5.6.4 Asymptotically Fewer Graph Queries

By tracking candidates, we asymptotically reduce the number of graph queries for many parallel algorithms and primitives. In particular, consider parallel operations on mutable arrays such as map, filter, and scan (a.k.a., prefix sums). These operations read their input array and allocate a fresh output array for the result. Without candidates, reading the input requires $O(n)$ entanglement checks for an input array of size n, and therefore would naïvely require $O(n)$ graph queries for entanglement detection. With candidates, we may reduce the number of graph queries from $O(n)$ down to $O(1)$, and often even zero, as we observe in our experiments (Section 8.7.2).

The reduction in queries occurs when the input array is not a candidate. Whether or not an input is a candidate depends on how that input was generated. Typically, the input is generated by a similar operation (e.g. the input to a filter may be generated by a previous map); in such cases, the input is not a candidate. For example, the map operation allocates an array, populates it (in parallel) with down-pointers, and then joins back up. When this completes, all down-pointers become internal, and therefore the resulting array is not a candidate for later operations.

5.6.5 Candidates in the Detection Semantics

We now describe how the notion of candidate can be defined in terms of the semantics of Section 5.3, allowing us to argue that tracking objects with down-pointers suffices for entanglement detection. We formally define a location ℓ as a candidate as follows. The definition says that location ℓ is a candidate if there exists a leaf task that is allowed to access ℓ but who would become entangled if it would dereference ℓ. This is precisely the condition in which entanglement is detected by the semantics.

Definition 1. In a program state $(\mu\,;\alpha\,;G\,;T\,;e)$, consider a location ℓ where $\mu(\ell) = \text{ref}\ \ell'$. We say that ℓ is an ***entanglement candidate*** if there exists some $\mathsf{Leaf}(v)$ in the task tree T such that $\alpha(\ell) \preccurlyeq_G v$ and $\alpha(\ell') \not\preccurlyeq_G v$.

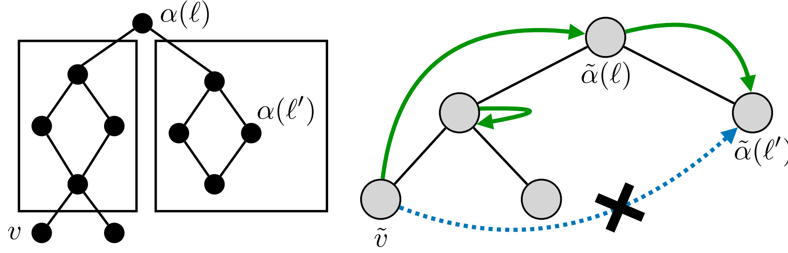

Figure 5.8: A partial dag on the left and its corresponding task tree on the right. Each node of the tree corresponds to contracted sub-dags, shown delimited by boxes. In the tree, disentangled pointers may point up, down, or internally to a node. An example entangled pointer is shown in dotted blue.

For example, consider the array A in the example of Figure 5.6, whose code was shown in Figure 5.1. Even though the array is mutable, the reads during `A[0]^A[2]` (and during `A[1]^A[2]`) cannot cause entanglement because the contents of the array (`A[0]`, `A[1]`, and `A[2]`) were all allocated *before* ($\preccurlyeq_G$) all leaf tasks. That is, while it is being read from (Figure 5.1, line 11), the array A is not a candidate and we can elide the graph queries. Note that A was a candidate earlier in the computation—specifically, while it was being initialized during the call to `init` (Figure 5.1, line 10). It just so happens that the array was never read while it was a candidate. As soon as the tasks within the call to `init` join, the array A is no longer a candidate.

Relating task trees and computation graphs. According to Definition 1, whether or not an object is a candidate depends on the current state of the computation graph G and how it relates to the current task tree, T. We can relate them as follows. Given a partial computation dag G, we can derive the task tree by contracting the dag: label each vertex of the dag so that (i) a fork and its corresponding join get the same label, and (ii) all vertices between them also get that label. If we contract all the vertices with the same label, we get a tree that is isomorphic to the task tree (upto labels); the labels map vertices of the dag to nodes in the task tree.

We use $\tilde{u}$ to refer to the label of vertex u, and define a partial order $\leq$ on the nodes of the task tree: $\tilde{u} \leq \tilde{v}$ if $\tilde{u}$ is an ancestor of $\tilde{v}$. The direction of pointers between objects (up, down, internal, and cross, as described in Section 4.2) can then be expressed in terms of the labels assigned to vertices. In particular, for a location ℓ, let $\alpha(\ell)$ be its allocation vertex in the dag and let $\tilde{\alpha}(\ell)$ be its label in the task tree. Based on their positions in the tree, we can classify a pointer from ℓ to ℓ' as follows: (i) *up pointer* if $\tilde{\alpha}(\ell) > \tilde{\alpha}(\ell')$, (ii) *internal pointer* if $\tilde{\alpha}(\ell) = \tilde{\alpha}(\ell')$, (iii) *down pointer* if $\tilde{\alpha}(\ell) < \tilde{\alpha}(\ell')$, and (iv) *cross pointer* otherwise: when $\tilde{\alpha}(\ell) \nleq \tilde{\alpha}(\ell')$ and $\tilde{\alpha}(\ell') \nleq \tilde{\alpha}(\ell)$.

For example, consider the dag and its tree in Figure 5.8. The dag has two fork-join pairs that correspond to subcomputations that have finished, as shown by the boxes; so every vertex within a box gets the same label. All other vertices are not between a fork-join pair, so they get different labels. After contracting the boxes, we get the tree on the right (with gray nodes). The figure also shows an up-pointer, an internal pointer, and a down-pointer, shown with solid green arrows. An example cross pointer is illustrated as a dotted blue arrow.

Candidates have down-pointers. The following statement formalizes the intuition that candidate objects (as given by Definition 1) have down-pointers.

Theorem 7. Assuming disentanglement (i.e., no cross-pointers),

$$\ell \text{ is a candidate} \Leftrightarrow \exists\, \ell' : \ell' \in \mathsf{locs}(\mu(\ell)) \wedge \tilde{\alpha}(\ell) < \tilde{\alpha}(\ell')$$

Proof. We can prove the right-to-left implication as follows: if there is a down pointer from $\tilde{\alpha}(\ell)$, it means that $\tilde{\alpha}(\ell)$ is an internal (non-leaf) node of the task tree and has at least one path to a leaf such that $\tilde{\alpha}(\ell')$ is not on it, i.e., $\exists$ leaf $v\ : \tilde{\alpha}(\ell) < \tilde{v} \wedge \tilde{\alpha}(\ell') \nleq \tilde{v}$. It follows from Lemma 10 that $\exists$ leaf $v\ : \alpha(\ell) \preccurlyeq_G v \wedge \alpha(\ell') \not\preccurlyeq_G v$, and therefore ℓ is indeed a candidate.

For the left-to-right implication, we have a leaf v and location ℓ' such that $\alpha(\ell) \preccurlyeq_G v$ and $\alpha(\ell') \not\preccurlyeq_G v$. There are three cases for the pointer from ℓ to ℓ':

1. Case $\tilde{\alpha}(\ell') \leq \tilde{\alpha}(\ell)$. From Lemma 10, we have $\tilde{\alpha}(\ell) \leq \tilde{v}$, and therefore $\tilde{\alpha}(\ell') \leq \tilde{v}$; applying the lemma a second time reveals $\alpha(\ell') \preccurlyeq_G v$, which is a contradiction because of the assumption that $\alpha(\ell') \not\preccurlyeq_G v$.
2. Case $\tilde{\alpha}(\ell') \nleq \tilde{\alpha}(\ell)$ and $\tilde{\alpha}(\ell) \nleq \tilde{\alpha}(\ell')$. Therefore the pointer from ℓ to ℓ' is a cross-pointer, violating our disentanglement assumption.
3. Case $\tilde{\alpha}(\ell) < \tilde{\alpha}(\ell')$. This satisfies the right-hand side of the desired implication.

□

The above proof relies on the following lemma, which establishes that for leaves, the tree ordering is equivalent to the dag ordering. This result can be proven by induction over individual steps of execution; formally, this would require augmenting the stepping judgement $S \longmapsto S'$ with the vertex labeling $\tilde{u}$. The proof is then mechanical, so we omit it for brevity.

Lemma 10. Throughout execution, for any vertex u and leaf v, $u \preccurlyeq_G v \Leftrightarrow \tilde{u} \leq \tilde{v}$.

The result of Theorem 7 establishes that marking candidates when down-pointers are created is sufficient for detecting entanglement. Therefore, because we unmark candidates conservatively, our candidate tracking algorithm is a valid optimization.

Chapter 6

The MPL Compiler for Parallel ML

To realize our goal of efficient and scalable parallel functional programming, we developed MPL ("maple"), a compiler and runtime system for Parallel ML. MPL extends the MLton [106] compiler for Standard ML, and provides efficient support for nested fork-join parallelism.

MPL inherits many features from MLton, especially in terms of compilation itself, where MPL and MLton are essentially identical. The main differences between the two systems are localized to the runtime system, where we implement a thread scheduler and memory management system based on the techniques developed in this thesis. In this chapter we summarize many important implementation details, especially concerning the implementation of the hierarchical heap architecture, scheduler, and garbage collector.

Acknowledgements

MPL is the culmination of over a decade of work from multiple contributors, including Ram Raghunathan, Stefan Muller, Adrien Guatto, Jatin Arora, Rohan Yadav, Larry Wang, Guy Blelloch, Umut Acar, Matthew Fluet, as well as myself. Around 2018, I took over as the lead developer of MPL. The origins of MPL can be traced back to Daniel Spoonhower's multicore extensions to MLton, developed for his thesis [143].

6.1 Scheduler

MPL features a work-stealing scheduler [36] utilizing Arora-Blumofe-Plaxton concurrent deques [19]. The scheduler maps user threads (one-shot continuations, implemented as heap-allocated call-stacks) onto worker threads (OS threads, specifically `pthreads`). We use one worker thread per processor, and therefore for simplicity we refer to these simply as "processors".

Initially, there is a single user-thread being executed by one processor. At each steal, the scheduler creates a new user-thread to execute the stolen work. The scheduler coordinates with the runtime system to create new heaps and merge existing heaps (at forks and joins), as described in this section.

MPL's scheduler is unique in that it is written mostly at the source level (i.e., in ML itself) with special runtime system calls where necessary. This is advantageous for compilation,

```
 1  structure MPL: sig
 2    structure Thread: sig
 3      type t   // first-class thread
 4      type thread = t
 5      val getCurrent: unit → thread
 6      val newThread: (unit → unit) → thread   // Fresh output thread is suspended.
 7                                              // (Must switchTo the new thread
 8                                              // to execute it.)
 9
10      val switchTo: thread → unit   // switchTo(t) suspends the current thread, and
11                                    // resumes t. The current thread will
12                                    // be resumed when someone switches back.
13    end
14
15    structure Heaps: sig
16      type depth = int
17      val getCurrentDepth: Thread.t → depth
18      val switchToHeapAtDepth: Thread.t × depth → unit
19      val attachSiblingAtDepth: Thread.t × Thread.t × depth → unit
20      val mergeSiblings: Thread.t * Thread.t → unit
21      val mergeIntoParent: Thread.t → unit
22    end
23  end
```

Figure 6.1: Auxiliary functions used by the scheduler. The modules `Thread` and `Heaps` are implemented in the run-time system and linked as foreign functions.

because (as observed also in [144]) it allows for certain optimizations to by implemented by relying entirely upon standard compilation optimizations. In particular, the Cilk [37, 70] fast/slow clone optimization can be expressed using higher-order functions.

6.1.1 Thread and Heap Maintenance

The scheduler coordinates with the run-time system to manage threads and heaps using the interface shown in Figure 6.1. The `MPL.Thread` module provides first-class threads, and support for hierarchical heaps is provided by the `MPL.Heaps` module. Both of these modules are implemented by functions in the run-time system, with bindings provided at the source level via MPL's foreign-function interface.

Instead of manipulating heaps as first-class objects, the scheduler implicitly associates heaps with first-class threads. A single thread may have many heaps associated with it, as illustrated in Figure 6.2. These heaps correspond to a path of heaps in the heap hierarchy. To distinguish the "current" heap of a thread (in which the thread performs all allocations), we equip each heap with a *depth*. For example, in Figure 6.2, heap **D** has depth 3. Threads control which heap they allocate in by indexing heaps by their depths.

The run-time system provides multiple operations to manipulate heaps, as shown in the `Heaps` module of Figure 6.1.

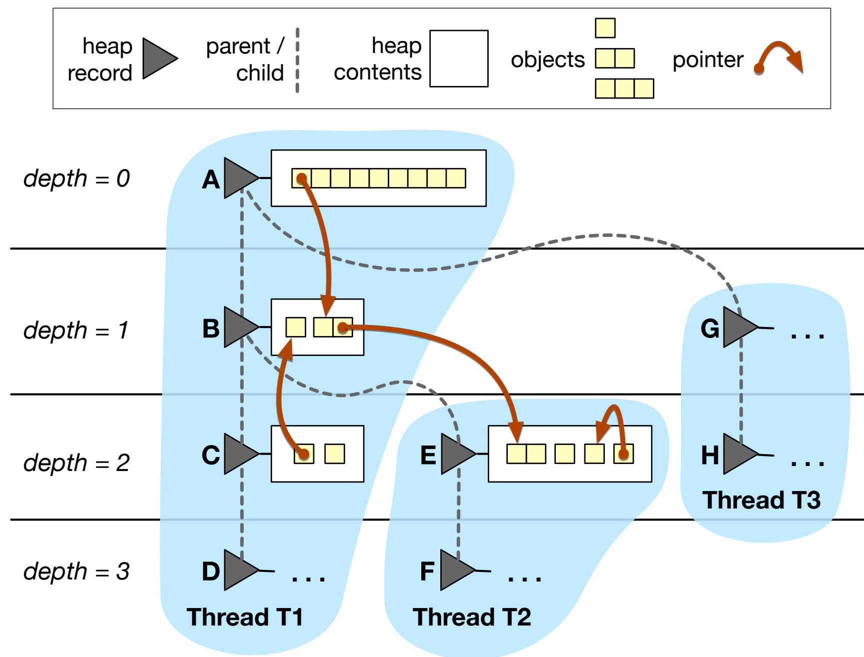

Figure 6.2: Example threads and heaps. Each thread has a list of associated heaps at various depths, corresponding to a path of heaps in the heap hierarchy.

- The function `getCurrentDepth` returns the depth of the current heap of a thread.
- The function `switchToHeapAtDepth` takes a thread t and a depth d as argument, and switches to the heap at depth d in the heap list associated with t. If there is no such heap, a fresh heap at depth d is created.[1]
- The function `attachSiblingAtDepth` takes two threads t_1 and t_2 as well as a depth d as arguments, and informs the runtime system that t_1 and t_2's heaps at depth d are siblings in the heap hierarchy. For example, in Figure 6.2, thread **T2** was attached as a sibling of **T1** at depth 2.
- The function `mergeSiblings` takes two threads t_1 and t_2 as arguments, and merges their heap lists with a "zip": heaps from t_1 and t_2 at the same depth are merged, resulting in a new heap list. When `mergeSiblings` completes, the new heap list is given to thread t_1, and thread t_2 is left with an empty associated heap list. This operation is only ever used on sibling threads. (For example, in Figure 6.2, threads **T1** and **T2** are siblings, but **T2** and **T3** are not.)
- The function `mergeIntoParent` takes a thread t as argument and merges its current heap into the parent of that heap. This requires that the thread has at least two heaps in its heap list.

[1]For efficiency, heap creation is delayed until the next failed *limit check*. That is, heaps are instantiated lazily as needed, which helps reduce fragmentation and decreases the overhead of heap maintenance. This is discussed in Section 6.3.

```
1  structure Scheduler: sig
2    type job
3    val spawn: (unit → unit) → job
4    val tryCancel: job → bool
5
6    type sync_var
7    val freshSyncVar: MPL.Thread.t → sync_var
8    val leftSynchronize: sync_var → unit
9    val rightSynchronize: sync_var → unit
10 end
```

Figure 6.3: Simplified interface of the work-stealing scheduler.

```
1 val S = Scheduler.freshSyncVar(MPL.Thread.getCurrent())
2 val _ = Scheduler.spawn (fn () ⇒ e2; Scheduler.rightSynchronize(S))
3 val _ = e1
4 val _ = Scheduler.leftSynchronize(S)    // wait for rightSynchronize(S)
5 // at this point, expressions e1 and e2 are both guaranteed to have completed
```

Figure 6.4: Example usage of sync_vars, as provided by MPL's scheduler. The call to leftSynchronize blocks until the corresponding call to rightSynchronize completes.

```
1  datatype sync_var = S of {incounter: int ref, guardedThread: MPL.Thread.t}
2  fun freshSyncVar(t) = S {incounter = ref(2), guardedThread = t}
3
4  // Note: leftSynchronize must only be called by guardedThread
5  fun leftSynchronize(S{incounter, guardedThread}) =
6    if Atomic.fetchAndAdd(incounter, −1) = 1 then
7      () // rightSynchronize already happened, so no need to block
8    else
9      // rightSynchronize hasn't happened yet.
10     // Switch away from this thread (it will be resumed when rightSynchronize happens)
11     MPL.Thread.switchTo(getIdleSchedulerThread(...))
12
13 // Note: this call never returns. Should be called as the final operation of some other thread.
14 // (Must not be called by guardedThread.)
15 fun rightSynchronize(S{incounter, guardedThread}) =
16   if Atomic.fetchAndAdd(incounter, −1) = 1 then
17     // leftSynchronize already happened; guardedThread is ready to be resumed
18     MPL.Thread.switchTo(guardedThread)
19   else
20     // leftSynchronize hasn't happened yet
21     MPL.Thread.switchTo(getIdleSchedulerThread(...))
```

Figure 6.5: Implementation of thread synchronization in MPL, using atomic fetch-and-add operations and switching between first-class threads.

6.1.2 Scheduler Jobs and Synchronization

At the source-level, we implement a standard work-stealing scheduler. In Figure 6.3, we show a simplified interface which is sufficient for implementing `par` (see Section 6.1.3). Some additional functionality is omitted here for brevity.

The scheduler provides support for first-class *jobs* which may be migrated between processors for parallelism. Jobs are thin wrappers around first-class functions of type `unit → unit`, and are stored in scheduler queues and migrated between processors. Executing a job is as simple as calling its associated function. Under the hood, the scheduler executes jobs by creating and switching-to fresh threads (using the functions `newThread` and `switchTo`, as shown in Figure 6.1).

Cancellation. To support the Cilk-style clone optimization (described below, in Section 6.1.3), the scheduler allows for jobs to be *cancelled* using the function `tryCancel`, which either succeeds or fails. Cancelling a job will only succeed if the job has not already begun execution; otherwise, the job cannot be cancelled and `tryCancel` will return `false`, indicating failure. If `tryCancel` succeeds, it returns `true`.

Thread synchronization. For synchronization between threads, the scheduler provides a type `sync_var` which can be used to block execution of a thread until all dependencies have completed. The interface here is specialized for binary fork-join parallelism, with two synchronization functions called `leftSynchronize` and `rightSynchronize`, respectively. When a `sync_var` is created, a thread t is passed as an argument; we refer to this thread as being *guarded* by the `sync_var`. The function `leftSynchronize` may then later be called by the guarded thread; this call will block until `rightSynchronize` is called by some other thread. An example usage of these functions is shown in Figure 6.4.

The `sync_var` type and corresponding functions can be implemented in terms of atomic read-modify-write operations and switching between first-class threads, as shown in Figure 6.5. At a high level, the idea is to allocate an *in-counter* which counts the number of outstanding dependencies. In our implementation, which is specialized for binary fork-join parallelism, this count is always initially 2.[2] The functions `leftSynchronize` and `rightSynchronize` then atomically fetch-and-decrement the in-counter, allowing them each to determine whether or not the other synchronization operation has happened yet. This race condition has two possibilities: either the `leftSynchronize` occurs first, or the `rightSynchronize` occurs first. In the former case, the `leftSynchronize` switches away from the guarded thread, and then the `rightSynchronize` later switches to it, resuming the guarded thread. In the latter case, the `leftSynchronize` is able to continue executing the guarded thread without any switching.

Note that for both `leftSynchronize` and `rightSynchronize`, if the guarded thread is not ready to be resumed, then these operations "return to the scheduler" by switching to an idle scheduler thread. As a result, the current processor will resume normal scheduler operations (such as attempting to steal ready-to-be-executed jobs from other processors).

[2]This could easily be generalized to a larger number of dependencies if desired, and even allows for specifying a dynamically determined number of dependencies.

```
1  fun par(f: unit → α, g: unit → β) : α × β =
2    let
3      val T1 = MPL.Thread.getCurrent()
4      val d = MPL.Heaps.getCurrentDepth(T1)
5      val rightSideThread = ref (NONE: MPL.Thread.t option)
6      val rightSideResult = ref (NONE: β option)
7      val syncVar = Scheduler.freshSyncVar(T1)
8
9      // Spawn the "slow clone" of right-hand side.
10     // If executed on a different processor, will be executed inside a fresh thread (T2).
11     val rightSide = Scheduler.spawn(fn () ⇒
12       let
13         val T2 = MPL.Thread.getCurrent()
14         val _ = MPL.Heaps.switchToHeapAtDepth(T2, d + 1)
15         val _ = MPL.Heaps.attachSiblingAtDepth(T1, T2, d + 1)
16         val r2 = g()
17       in
18         rightSideThread := SOME(T2);
19         rightSideResult := SOME(r2);
20         Scheduler.rightSynchronize(syncVar)
21       end)
22
23     // Advance to depth d + 1 and execute the left-hand side
24     val _ = MPL.Heaps.switchToHeapAtDepth(T1, d + 1)
25     val r1 = f()
26
27     // Get the right-hand side result
28     val r2 =
29       if Scheduler.tryCancel(rightSide) then
30         // rightSide successfully cancelled: execute "fast clone" on same processor.
31         (MPL.Heaps.mergeIntoParent(T1);
32          MPL.Heaps.switchToHeapAtDepth(T1, d);
33          g()) // execute g directly, with no additional synchronization
34       else
35         // synchronize with the "slow clone" (rightSide, executed by different processor)
36         let
37           val _ = Scheduler.leftSynchronize(syncVar)
38           val SOME(T2) = !rightSideThread
39           val _ = MPL.Heaps.mergeSiblings(T1, T2)   // merge right heaps into left
40           val _ = MPL.Heaps.mergeIntoParent(T1)
41           val _ = MPL.Heaps.switchToHeapAtDepth(T1, d)
42           val SOME(r2) = !rightSideResult
43         in
44           r2
45         end
46   in
47     (r1, r2)
48   end
```

Figure 6.6: Simplified presentation of the implementation of par in MPL.

6.1.3 Implementing the `par` Function

As described in Chapter 2, we provide the programmer with a single parallel primitive called `par`, which takes two functions f and g as arguments, evaluates $f()$ and $g()$ in parallel, and finally returns their results as a tuple. This primitive is implemented mostly at the source level, with calls into the MPL run-time system as necessary.

A simplified presentation of the implementation of `par` is shown in Figure 6.6, using the interfaces of Figures 6.1 and 6.3. For a call `par`(f, g) which occurs at a heap of depth d, the idea is to spawn a job which executes $g()$ in a heap of depth $d + 1$, and then reuse the current thread to execute $f()$, also at depth $d + 1$. We refer to the execution of $f()$ as the "left-hand side", and similarly refer to the execution of $g()$ as the "right-hand side".

Fast/Slow Clones. When the left-hand side (the call to $f()$) is finished, it is possible that the spawned right-side job has not yet been scheduled. We could immediately synchronize with this job, which would have the effect of switching back to the scheduler and then creating a fresh thread for the right-side job. This overhead is substantial, and can be avoided, as demonstrated by the *clone optimization* in the Cilk implementation [37, 70]. We refer to the right-side job as the ***slow clone*** of $g()$; this version of $g()$ has to perform synchronization code and requires a fresh thread to execute. To avoid the overhead of the slow clone, before synchronizing with the slow clone, the `par` function first attempts to cancel the right-side job. If this is successful, then $g()$ can be executed on the same thread with no additional synchronization. We refer to this call to $g()$ as the ***fast clone***. In a highly-parallel program (where the parallelism of the program is much larger than the number of available processors), many calls to `par` will be able to take advantage of the fast clone.

For MPL, in the fast-clone case (where $g()$ is executed on the same thread), we "revert" the new heap that was used to execute $f()$ at depth $d + 1$, and execute $g()$ at depth d rather than $d + 1$. This is accomplished by merging the current heap (containing any allocations of $f()$) back into the parent heap before executing $g()$. In the case where the slow clone is executed on a different processor, we need to retrieve the right-hand-side result, and also need to merge the right-hand-side heaps. Both of these are accomplished using the references `rightSideResult` and `rightSideThread`, which are updated by the right-hand-side job when it completes, and before it synchronizes. This way, after the original thread T_1 calls `leftSynchronize`, it can safely read the right-hand-side thread and merge its heaps. After merging heaps, it is safe (with respect to disentanglement) to read the right-side result (Figure 6.6, line 42).

6.2 Block Allocator

All significant allocations performed by MPL are backed by a block allocation system. At a high level, this system is based roughly on the Hoard memory allocator [25].

Blocks and superblocks. We logically divide the virtual memory space into fixed-size ***blocks*** of 2^k bytes (we use $k = 12$), appropriately aligned such that the low-order bits of the beginning of each block are zeroed. Blocks are organized into contiguous groups called ***superblocks***,

where each superblock contains 2^s blocks (we use $s = 7$). Each superblock has a size class, indicating how much memory it can supply with a single allocation. A size class of 1 means that the superblock can be used to supply allocations of single blocks; a size class of 2 means that the superblock supplies allocations of 2 contiguous blocks, etc. Size classes are powers-of-two to ensure that any superblock can be used for any size class, up to 2^s (the superblock size).

Free-lists and fullness groups. Superblocks are organized into processor-local free-lists, to ensure fast processor-location allocation of blocks. Based on the Hoard algorithm, superblocks are also organized into ***fullness groups*** based on how many blocks within the superblock are currently in use. To select a superblock to supply an allocation, each processor always selects the fullest superblock from the appropriate size class within its processor-local freelist of available superblocks. This ensures that mostly-unused superblocks eventually become completely free. Any superblock which is completely free may then be reassigned to a different size class, if needed.

Processors hold onto their superblocks. In a deviation from the Hoard algorithm, we do not migrate superblocks between processors. If no free superblock is available for allocation in a processor's local freelist, then the processor simply allocates a new superblock from the OS. The advantages of this approach are that (i) it it simpler to implement, (ii) it has low contention, and (iii) it has good locality, as memory initialized by a processor will eventually become available for re-allocation by the same processor. The downside is that it potentially uses more memory than necessary. Nevertheless, because all processors in the MPL runtime system are treated homogeneously by the scheduler, and because the number of processors is fixed throughout an execution, this approach works well. In the future, if these constraints are relaxed, it may become necessary to implement Hoard-style superblock migrations.

Megablocks. Beyond superblocks, for large allocations (those exceeding the largest superblock size class, which in our case is 512KB), we implement a ***megablock*** allocator. Megablocks are stored in a global freelist, accessible by any processor, and are organized by size class. To allocate a megablock, a processor acquires a global lock on the global freelist, and searches for an appropriate megablock. If none is found, a new megablock is `mmap`'ed from the OS. When a megablock is freed, it is always returned to the global freelist. In our current implementation, megablocks are never returned to the OS, but this could be easily be changed in the future if desired. We allow megablocks of up to 2^{18} blocks.

Very large allocations. Finally, beyond megablocks, for extremely large allocations (those exceeding the largest megablock size class, which in our case is more than 1GB of memory in a single allocation), we `mmap` and `munmap` memory directly from the OS on every allocation and free.

6.3 Heaps and Heap Objects

Heaps and Chunks. In the run-time system, we implement heaps as linked lists of ***heap chunks***, with each chunk consisting of one or more contiguous blocks (Section 6.2) of memory. This strategy makes it possible to merge two heaps without copying any data: instead, we merge two heaps simply by linking together their chunks-lists, which takes constant time.

At the front of each chunk is a ***chunk descriptor*** containing various metadata needed for allocation and garbage collection, including *frontier* and *limit* pointers. The ***frontier*** points to the beginning of the unallocated space within a chunk, and the ***limit*** points to the end of the chunk. To perform allocations, the compiled source manipulates the frontier to bump-allocate within the "current" heap chunk until the limit is reached. The compiled code includes explicit ***limit checks*** to determine when a new chunk is needed: when a limit check fails due to lack of space, a call is made into the runtime system to allocate a new chunk and extend the current heap, or perform a garbage collection if needed.

Lazy Heap Creation. To reduce fragmentation and decrease the cost of instantiating new heaps, MPL instantiates new heaps lazily at failed limit checks. Specifically, when a limit check fails and a fresh chunk is needed, MPL performs a call into the runtime system. The runtime call allocates the fresh chunk at the current thread's depth, and a fresh heap is instantiated only if no heap at that depth exists yet. This way, every heap in the system contains at least one chunk, and chunks aren't abandoned until they are mostly full.

Heap Objects. Within the allocated region of a chunk (between the chunk start and frontier), MPL stores ***heap objects***. The heap object model of MPL is inherited from MLton [106]; for completeness, we describe the details here.

Each heap objects consist of a *header* together with a *payload*. The ***header*** stores GC metadata specific to that object (accessible only by the compiler and runtime system), and the ***payload*** stores data corresponding to source-level values. Although the payload data is determined by the source-level program, its representation and layout is ultimately controlled by the compiler. As a whole-program optimizing compiler, MPL will choose low-level memory representations for data depending on the context in which that data is used. That is, two values that both have the same source-level type might have different memory representations in the compiled code. Similarly, the same source-level function might be copied multiple times in the compiled code, with each copy specialized for different data representations. This is especially true for polymorphic source-level functions, which are monomorphized at compile-time by making (at least) one copy per distinct monomorphic instantiation.

One of the main optimizations performed by MPL (inherited from MLton) during compilation is *data flattening*, which changes the memory representations of objects with the goal of eliminating unnecessary allocations. That is, while an object of type `(int × int) array` could be represented by an array of pointers to tuples, it is likely more efficient (both in terms of total memory usage and data locality) to use an "array-of-structs" layout, avoiding the use of pointers entirely. In this way, MPL deviates from the theoretical semantics of Section 3.1 in which disentanglement was defined. In particular, for ease and simplicity of presentation, the

formal semantics of Section 3.1 explicitly allocates a memory location for *all* data, even "small" types such as integers. But this would not be efficient in practice. Data flattening optimizations are crucial for efficiency; we have measured runtime and space improvements of 2x or more due to flattening. These optimizations appear to be safe for disentanglement, because flattening only eliminates allocations. We therefore leave the flattening optimizations turned on. We have encountered no correctness issues due to flattening in our experiments and benchmarks.

Heap Object Types and Tags. MPL permits many different forms of payloads with flexible sizes and layouts. To be able to trace through memory, the GC system needs to know the size and layout of each payload. Therefore, in the header of each object, MPL stores an ***object tag***. The object tag keeps track of four pieces of information: (1) an ***object type*** (described below), (2) whether or not the object is mutable, (3) the number of bytes of non-pointer data, and (4) the number of pointers to other objects. This information is sufficient for the GC system to determine the size of an object, as well as whether or not it contains pointers to other objects (and at what offsets those pointers are located). Object tags are efficiently represented as indices into a static ***object tag table***. MPL reserves 19 bits in the header for a tag index, allowing up to 2^{19} different object tags in single executable. Typical executables only need (at most) a few hundred distinct tags.

MPL uses three distinct object types: *normal* objects, *stack* objects, and *sequence* objects. Most objects in a typical program are ***normal*** objects, which are used to represent tuples, records, and (mutable) references. A normal object consists of some number of non-pointer fields followed by zero or more pointers, as indicated by the object tag. The ***sequence*** object type is used to represent (mutable) arrays and (immutable) vectors. Objects of this type have a larger GC header, which additionally stores the length of the sequence (in terms of number of elements). Each element of a sequence object is laid out like a normal object, but with no GC header. The GC header of the sequence itself describes the layout of each element (e.g. the number of non-pointer and pointer fields and their offsets). Finally, objects of the ***stack*** type are used to store call-stacks for first-class threads in the source program. These first-class threads are used extensively in the implementation of our scheduler (Section 6.1). First-class threads themselves are represented as normal objects, containing a distinguished pointer to the stack.

The compiled program directly manipulates call-stacks by pushing and popping function frames which store local data needed by individual function calls. Frames can contain pointers to heap objects. To make it possible for the GC to identify where these pointers are located on the stack, each frame has a corresponding ***frame index*** which is used to look up information about the layout of the frame in a ***frame info table***. The frame info table is stored statically in the compiled code of each executable, similar to the static object tag table.

Single- and Multi-object Chunks. Typically, a heap chunk stores many heap objects: at its smallest, a heap chunk is at least a page (4KB) in size, and most heap objects are on the order of tens (or possibly hundreds) of bytes. However, some objects are large. Sequence objects in particular can be very large: MPL supports arrays containing up to $2^{63} - 1$ elements. Stack objects also can grow quite large, depending on the maximum stack depth of a program.

To support large objects, MPL distinguishes between ***single-object chunks*** and ***multi-***

object chunks. As the names suggest, a single-object chunk stores only a single object, where as a multi-object chunk stores many objects adjacent to one another. Sequence objects (arrays and vectors) are only given their own single-object chunk if the sequence is at least half a page in size. MPL chooses to give all stacks their own chunk, to facilitate stacks growing if necessary.

The advantage of single-object chunks is that it allows for large objects to be moved between heaps without copying any data. In particular, during garbage collections, if a large array needs to be moved between heaps, this can be accomplished in constant time (by unlinking and then relinking the chunk into a different chunk-list). The same is true for single-object chunks containing stacks. The running time performance advantages of this optimization are significant, and easily outweigh the (small amount of) fragmentation introduced by single-object chunks.

6.4 Parallel Initialization of Sequence Objects

As described in Section 6.3, the sequence object type is used to represent (mutable) arrays and (immutable) vectors, which can be large (up to $2^{63}-1$ elements). When a fresh region of memory is allocated for a sequence object, it could contain arbitrary bits (i.e., the fresh memory is not guaranteed to be zero'ed or otherwise initialized to some known safe value). If a fresh sequence object is tagged as containing pointers but is left uninitialized, then the GC system may attempt to interpret unknown bits as a pointer, leading to a crash (or worse). Fresh sequence objects therefore need to be initialized with values which are safe-for-GC.

Many compilers and runtime systems—including MLton, on which MPL is based—perform this initialization with a sequential loop on a single processor, at the moment a sequence is allocated. This approach is a non-starter for parallelism, as it introduces long delays onto the critical path: any algorithm which in parallel generates many elements stored in an array would be effectively sequentialized.

Parallelization with Raw Sequences. MPL parallelizes the allocation and initialization of sequence objects in a three-step process.

1. First, a region of memory for a sequence object is allocated, and the GC header is initialized. In the GC header, the object is marked as a ***raw sequence***: any sequence object marked as *raw* might contain unknown bits, and therefore should not be scanned by the GC sytem. When MPL's garbage collector encounters a sequence marked as raw, it skips over the contents of the sequence.
2. Next, the raw sequence is initialized in parallel: for each field of the sequence that eventually may contain a pointer, a GC-safe value is written (e.g., a null-pointer). This step is integrated with the scheduler, using ordinary tasks which are spawned and scheduled just like any other parallel computation in MPL.
3. Finally, the sequence is unmarked, and is no longer raw. At this point, the GC can safely scan the sequence.

After completing this three-step process, the sequence can then be filled with normal ML values in parallel.

6.5 Heap Queries

The MPL runtime system relies heavily on the ability to query which heap contains an object, and to do so efficiently. This presents an interesting challenge, because MPL creates fresh heaps and merges heaps at a high rate: typically, each call to `par` creates at least one fresh heap, and then later performs at least one heap merge. The frequency of heap merges precludes any implementation which iterates through the heap on each merge to update a heap ownership field, even if this field was stored on a per-chunk basis. Such an approach would be prohibitively expensive. For efficient heap queries, we need an algorithm which is effectively constant-time at heap merges.

Here we leverage the observation that heap queries are essentially an instance of the classic union-find problem. Based on this observation, we choose to support efficient heap queries in MPL with a path-compressing union-find data structure. In particular, given the address of some object, MPL can find the heap which contains the object via the following procedure. First, the chunk descriptor of the chunk that contains the object can be found in constant time by zeroing the low-order bits of the object's address. In the chunk descriptor, we store a pointer to a union-find node corresponding to the heap which contains the chunk. We then follow parent-pointers in the union-find tree to find the representative union-find node for this tree in the union-find forest (path-compressing along the way, if needed). The representative union-find node then contains a pointer to the heap record of the appropriate heap.

Note that, to find chunk descriptors in constant time, we require that chunks are appropriately aligned at power-of-two addresses. This is guaranteed by the block allocator: each chunk consists of one or more contiguous blocks, and blocks themselves are aligned.

6.5.1 Memory Reclamation for Union-Find Nodes

In the union-find forest, path compression will splice out many union-find nodes. To avoid a space leak, these nodes must be reclaimed. Detecting precisely when a union-find node is no longer in use is not obvious, because many processors may perform heap queries concurrently. We therefore defer the work of reclaiming union-find nodes to garbage collections.

At LGCs, rather than attempt to determine which nodes are and aren't still in use, we instead simply construct a fresh heap structure with fresh heap records and fresh union-find nodes, in a flat structure. This way, we ensure that all old union-find nodes and heap records are no longer in use, and may be reclaimed all at once in bulk. This approach essentially performs a custom scavenging copying collection on heap records and union-find nodes at each collection.

For CGCs, we use a similar approach. However, note that in the case of CGC, we cannot simply reclaim all old union-find and heap nodes, because some of these may still be in use by other processors. Essentially, in the case of CGC, the union-find data-structure behaves as a (wait-free) concurrent data structure, and suffers from the same correctness issues for reclamation as other concurrent data structures do [69, 105].

When reclaiming union-find nodes during CGC, rather than directly free old nodes, we instead *retire* them, using a variant of the DEBRA epoch-based reclamation algorithm [43]. The high-level idea of epoch-based reclamation is to associate each retired node with an *epoch*, which is a period of execution that is agreed upon by all processors. Any node which was retired

at least two epochs prior to the current epoch is safe to free. To help each other free nodes, processors periodically attempt to advance the current epoch. When a processor successfully advances the epoch, it may safely free nodes that it retired two epochs prior.

6.5.2 Allocation for Heap Records and Union-Find Nodes

To support efficient allocation of heap records and union-find nodes, we implement an allocator which is specialied for fixed-size small objects. Similar to MPL's block allocator (Section 6.2), the fixed-size allocator is based on processor-local freelists. Note that the memory of the fixed-size allocator is backed by blocks from the blocks allocator, ensuring that all of MPL's allocations ultimately are backed by a single mechanism which communicates with the OS to map and unmap virtual memory.

6.6 Remembered Sets and Write Barriers

We equip each heap with a depth and a *remembered set* in order to efficiently implement the collection algorithms described in Chapter 4. The depth is simply the depth of the heap in the heap hierarchy, which is easily maintained at forks and joins. The remembered sets store entries of the form (x, y), indicating that object x held a down- or right-pointer to y. This approach supports a strategy we call *pinning*, which is described in more detail in Section 6.8.4.

Remembered sets are maintained by a *write barrier*, which is a small piece of code that inserted in the compiled program before certain writes to memory. Our implementation has a write barrier for every update of pointer data that might result in a down-pointer. At the write barrier for the update $x[i] := y$, we compare the depths of $\mathit{heap}(x)$ and $\mathit{heap}(y)$: if $\mathit{heap}(y)$ is deeper than $\mathit{heap}(x)$, then we remember the down-pointer by inserting the entry (x, y) in the remembered set for $\mathit{heap}(y)$. Note that disentanglement guarantees that x is either an ancestor or a descendant of y, which is why we can determine their relative position in the hierarchy simply by comparing their depths.

The contents of the remembered set are used to facilitate fast management of down- and right-pointers during both LGC and CGC. In particular, both collection algorithms use the objects $\{y \,|\, (_, y) \in \text{RememberedSet}\}$ as additional roots for collection after filting the remembered set to remove invalid entries. Remembered set entries can become invalid if the corresponding down- or right-pointer has become due to heap merges. That is, for any remembered-set entry (x, y), if $\mathit{heap}(x) = \mathit{heap}(y)$, then the entry is no longer needed (because the down- or right-pointer has become an internal pointer), and so the entry is removed.

6.7 Garbage Collection

In MPL, garbage collections are triggered at failed limit checks based on an amortization policy. When a GC is triggered, MPL chooses between LGC and CGC based on a shallowness criterion, which concentrates the work of CGC on shallow heaps, and relies on LGC for all other heaps (e.g., heaps close to the leaves of the hierarchy). In this section, we describe the implementa-

tions of LGC (Section 6.7.1) and CGC (Section 6.7.2), as well as the amortization policy which determines when LGCs and CGCs occur (Section 6.7.3).

6.7.1 LGC

We implemented LGC by adapting a Cheney-style copying collector [49], which copies and compacts surviving objects into a fresh set of heaps. The tracing phase performs a Cheney-style collection on each in-scope heap, beginning at the deepest local heap and progressing to the shallowest local heap. In each heap, objects are copied by individually evacuating pointers which point into the from-space, and we install forwarding pointers so that references to old objects may be updated. By processing from deepest to shallowest, we guarantee that when a heap is processed, all up-pointers into the heap have already been evacuated.

Note that the Cheney-style collection algorithm used for LGC requires that all in-scope tasks remain inactive until the LGC completes. Otherwise, if an in-scope task were active, it could witness a forwarding pointer installed on a relocated object.

MPL ensures that in-scope tasks remain inactive by integrating with the scheduler. Each in-scope task corresponds to one job in the scheduler queue; therefore, to ensure that these tasks remain inactive, LGC temporarily removes the appropriate jobs from the scheduler queue. In particular, we implement the local heap assignment algorithm of Section 4.4 by walking upwards from the leaf, iteratively removing jobs from the scheduler queue that were added by the current thread. Each such job corresponds to a sibling task which is ready but inactive. By removing these job from the scheduler queue, MPL ensures that the corresponding tasks remain inactive. Later, when the LGC completes, all of the jobs removed in this process are restored by placing them back into the scheduler queue. From a scheduling perspective, this effectively treats each LGC as an additional dependency for each in-scope inactive task.

Simplified Heap Assignment. In our current implementation, MPL deviates slightly from the local heap assignment algorithm of Section 4.4, by not distinguishing between completed and active tasks. This simplifies the implementation of the heap assignment algorithm: removing one job from the scheduler queue corresponds to claiming a single ancestor heap to be included in LGC. However, the consequence is that any heap of a completed child task (as well its parent) is excluded from LGC until the sibling completes. Note that in our experiments (Chapter 8), we observe that MPL is highly space-efficient in general, suggesting that in practice, this simplification does not have significant impact on the precision of garbage collection. In future work, we plan to revisit this issue, to measure the impact precisely.

Example. Returning to Figure 6.2, consider that **T1** called `par` at depth 2 and is now currently working on the left-side task, at depth 3. If **T1** triggers an LGC, it may attempt to remove the corresponding right-side job (not shown in the figure). There are then two possibilities.

1. Suppose another processor has already begun working on the right-side job. Then thread **T1** fails to remove the job, and the scope of the LGC is limited to only heap **D**. When the LGC completes, because no jobs were removed, no jobs need to be returned to the scheduler queue.

2. Suppose the right-side job has not yet begun execution on another processor. Then **T1** will successfully remove the right-side job and thereby claim the heap **C** for LGC. Next, **T1** may attempt to claim heap **B** by removing another right-side job; however, **T1** will fail to claim **B**, because the appropriate right-side job has already begun execution on another processor (thread **T2** in the figure). Therefore, the scope of the LGC will consist of two heaps: **D** and **C**. When the LGC completes, the single job that was removed (to claim **C**) is placed back into the scheduler queue.

6.7.2 CGC

For CGC, we implemented a non-moving mark-and-sweep algorithm based on a SATB (i.e., snapshot-at-the-beginning [161]) strategy. Our CGC algorithm iteratively processes objects in depth-first order in two passes. The first pass marks reachable objects. After marking reachable objects, all chunks in which no objects were reachable are freed. Next, the algorithm performs a second pass over the reachable objects, to unmark them.

For the SATB algorithm, we augment each heap with a ***forgotten set***, which contains all objects x such that a mutable update deleted a pointer to x. The forgotten set is updated at write barriers. Additionally, when a processor inserts an element into the forgotten set, it marks the object, thereby ensuring that the same object is not added multiple times. In this way, forgotten set maintenance "helps" concurrent collection.

Coordinating access between a CGC task and other tasks adding elements to the forgotten set requires care. For efficiency, we implement forgotten sets with a distributed "bag" data structure consisting of P separate lists, one for each processor. When processor i adds an element to the forgotten set, it adds the element to the ith list. This ensures that insertions are low-contention.

The CGC task itself accesses the forgotten set by removing batches of inserted elements (which must then be scanned for collection). To end the marking phase of CGC, the CGC task attempts to *close* the forgotten set. If it succeeds, then the marking phase is complete, and no more elements will be added to the forgotten set. If it fails, then an additional element (or multiple elements) must have been added to the forgotten set by another processor. In this case, the CGC will remove the new element(s), scan it for collection, and then try again to close the forgotten set (repeating until it succeeds). This process is guaranteed to eventually succeed, because the write barrier marks each element before adding it to the forgotten set, and only adds to the forgotten set if an element is not already marked.

6.7.3 Amortization Policy for LGC and CGC

A simple amortization policy for LGC and CGC is as follows. We keep track of the size (i.e., the amount of allocated memory) of each heap h, denoted $|h|$. We then annotate each heap h with a counter, $S(h)$, which is the amount of memory in h that survived the most recent garbage collection where h was in-scope. That is, when a collection is completed on a heap h, we reset the counter $S(h)$ to the new size, i.e., $S(h) \leftarrow |h|$. Fresh allocations in a heap h increase $|h|$ but do not affect $S(h)$. Note that counters $S(h)$ and $|h|$ are summed when heaps merge.

For a particular local heap assignment T on a single processor (Section 4.4), our policy dictates that the processor should perform an LGC whenever $\sum_{h \in T} |h| \geq k \sum_{h \in T} S(h)$, where $k > 1$ is an amortization parameter, set at run-time. The parameter k can be adjusted to control the cost of garbage collection: increasing k decreases the frequency of garbage collection, which decreases the time cost and increases the space cost.

For CGC, we use a similar policy. For any primary leaf heap h, our policy spawns a CGC task to garbage-collect h if the following two conditions are met:

1. The size of the heap must be large enough. Specifically, we do not collect a heap h unless $|h| \geq k \cdot S(h)$, where $k > 1$ is an amortization parameter, set at run-time.
2. The heap must be *shallow*. Specifically, the heap must be at depth d_{shallow} or less, where d_{shallow} is a parameter set at run-time.

The shallowness condition is a heuristic, to ensure that CGC and LGC cooperate. In particular, note that the first condition (i.e., $|h| \geq k \cdot S(h)$) is also a satisfying condition of LGC. For shallow heaps, rather than perform LGC eagerly, we instead prioritize CGC. This prioritization is motivated by the observation that shallow heaps are typically larger than other heaps: as tasks "join up" and merge their heaps, data naturally flows upwards, into shallow heaps, eventually accumulating at the root heap.

Work-efficiency. The combination of our CGC and LGC amortization policies produces an overall collection policy which is work-efficient. In particular, for an execution that performs W work in total, this policy ensures that the total work of garbage collection is at most $O(W)$. (The *work* of execution is the total number of instructions performed, excluding the cost of garbage collection.)

The argument for work-efficiency is straightforward. Garbage collection requires a linear amount of work; therefore, each LGC of a local heap assignment T pays $O(\sum_{h \in T} |h|)$, and each CGC of an internal heap h pays $O(|h|)$. These costs can be charged against the cost of allocations. For each individual heap h, at the beginning of any garbage collection on that heap (either LGC or CGC), there are $A(h) = |h| - S(h)$ fresh allocations that have not yet participated in a garbage collection. Assigning a potential of $A(h)$ to the heap yields an amortized garbage collection cost, per heap, of $A(h)/(k-1)$, where $k > 1$ is the amortization constant. Summing over all heaps and all garbage collections yields a total cost of $A_{\mathsf{tot}}/(k-1)$, where A_{tot} is the total amount of memory allocated throughout execution. Because $A_{\mathsf{tot}} \leq W$ (i.e., every allocation is at least one unit of work), we have that the total cost of garbage collection is bounded by $O(W)$.

Separate Control over LGC and CGC. The above policy uses a single amortization parameter k for both LGC and CGC, but this is not necessary. In MPL, we give each of LGC and CGC their own amortization parameters: k_{LGC} and k_{CGC}, respectively. This allows us to separately control the eagerness of LGC and CGC. By default, MPL makes CGC more eager than LGC, with the settings $k_{\mathsf{CGC}} = 2$ and $k_{\mathsf{LGC}} = 8$. In our experience, because CGC typically operates on larger heaps, it is advantageous for space efficiency to run CGC more frequently. In contrast, because LGCs is scheduled on the critical path, it is advantageous for running time efficiency to run LGC less frequently. The space impact of increasing the LGC amortization parameter is small, because LGC generally operates on smaller heaps.

For a shallowness threshold, we use $d_{\mathsf{shallow}} = 2$. That is, by default, MPL only spawns CGCs for heaps that have at most two ancestors. This ensures that CGC is concentrated on a small number of (generally large) shallow heaps.

LGC Scope Heuristic. As a heuristic, LGC claims as few heaps as possible while still ensuring that enough memory is in-scope. In particular, if an LGC is allowed to trace up to M memory, then the LGC will select as few heaps as possible to guarantee at least $c \cdot M$ memory is in-scope of the collection (where $0 < c < 1$ is an adjustable parameter; MPL uses $c = 3/4$ by default). This heuristic is motivated by two observations: (i) it is beneficial for parallelism to select smaller LGC scopes, and (ii) the heap hierarchy generally is top- and bottom-heavy, with many middle-range heaps which are nearly empty.

For example, consider a leaf heap h together with its parent h', both local to a single processor. Suppose an LGC is triggered where $|h| = 90$ and $|h'| = 10$, with parameter $c = 0.9$. Instead of collecting both heaps, the LGC will only collect h, because h alone constitutes 90% of the available collectible memory. This leaves the parent heap h' unclaimed, allowing for a sibling task (if any) to be scheduled before the LGC completes.

6.8 Entanglement Detection Implementation

6.8.1 Vertex Identifiers and SP-order maintenance

To implement graph queries for entanglement detection, we use DePa [156], an "off-the-shelf" SP-order maintenance data structure which is optimized for binary nested parallelism and is suitable for MPL's parallel primitives. In this data structure, each vertex is represented by an identifier packed into one or more machine words, and graph queries are efficiently performed by comparing vertex identifiers, requiring work proportional to the number of words in the identifier. The size of a identifier is determined by the dynamic nesting depth of the corresponding task during execution. In the common case, because dynamic nesting depths of highly-parallel programs are typically small, a vertex identifier is a single 64-bit word. At forks and joins, the scheduler reassigns the current vertex identifier of the current thread to match that specified by our algorithm (Chapter 5).

6.8.2 Read and Write Barriers for Detection

Marking entanglement candidates. In MPL's write barrier, we additionally mark the mutated object as an entanglement candidate if the write creates a down-pointer. To mark candidates, we reserve a bit in the object's GC header which indicates whether or not the object is a candidate.

Inserting read barriers. For entanglement detection, MPL inserts read barriers for reads (dereferences) of mutable objects. In Parallel ML, ref-cells and arrays are the only mutable objects, meaning that the only operations needing a read barrier are the operations which retrieve

elements from refs and arrays, including standard dereferences as well as atomic compare-and-swaps:

```
val !: α ref → α
val sub: α array × int → α
val refCompareAndSwap: α ref × α × α → α
val arrayCompareAndSwap: α array × int × α × α → α
```

In the compiler front-end, we begin by conservatively marking all of these operations as needing a read barrier. However, not all actually result in a read barrier in the generated code. In the compiler backend, MPL eventually chooses concrete, bit-level representations for all data. In this step, we only add read barriers if MPL chooses to represent the contents of the ref (or array) with a pointer.

For example, consider an object x of type `int` **ref** and corresponding dereference `!`x. MPL never chooses to indirect the integer through a pointer, so no read barrier is inserted. But with types such as `(int × int)` **ref**, MPL may choose to either (a) inline the integers into the ref, or (b) allocate the tuple separately and represent the contents of the cell with a pointer. A read barrier is inserted only in the latter case, where MPL chooses to allocate the contents separately.

Read barrier fast and slow path. The read barrier has two behaviors: a fast path for objects that are not entanglement candidates, and a slow path otherwise. The fast path is implemented by codegen in the compiler. For each dereference `!`x, we generate code which first optimistically performs the read, and then by inspects the header of x to check if x is marked as a candidate. If the read barrier sees that x is *not* marked as a candidate, then the fast path is satisfied, and execution continues as normal (i.e., the graph query is elided). If the read barrier sees that x is marked as a candidate, then we fall back on the slow path. In the slow path, MPL performs a graph query to check for entanglement. The graph query is implemented as a function call into the run-time system (written in C and linked with the generated code).

Note that the read barrier synchronizes with the write barrier, which might concurrently update an object and mark it as a candidate. The order of operations is important. The read barrier inspects the mark *after* it performs the read, while the write barrier sets the mark *before* it performs the update. Abstractly, there are two states for an object, determined by the state of the mark: either (i) it is definitely not a candidate, or (ii) it is *possibly* a candidate. By placing the mark between the read and write, we ensure that the state transition (from non-candidate to possibly-candidate) linearizes when the mark is set.

6.8.3 Memory Management for Detection

Vertex identifiers in heap chunks. In each chunk descriptor, MPL additionally stores a vertex identifier, which is shared across all objects within the chunk. When a new chunk is allocated, it is assigned the current vertex identifier of the thread that allocated the chunk. Chunk boundaries are aligned, allowing fast access to chunk descriptors by zeroing-out the low-order bits of an object's address. This makes it possible (in constant time) to look up the vertex of the thread which allocated an object. The read barrier therefore can check for entanglement by inspecting the vertex stored in the chunk descriptor of the object in question.

Reassignment of vertices during GC. In MPL, the GC design incorporates a local copying collector which compacts the live objects of each thread into a new set of heap chunks. These new chunks must be assigned an appropriate vertex identifier. Here, we take advantage of the fact that after two threads join, their corresponding graph vertices are indistinguishable with respect to entanglement detection. That is, using the definitions of Section 5.6, for any two vertices u and v such that $\tilde{u} = \tilde{v}$ and any leaf vertex w, we have $u \preccurlyeq_G w$ if and only if $v \preccurlyeq_G w$. When picking a vertex identifier for a new chunk during GC, it is therefore safe to use any vertex identifier of any chunk in the heap being compacted.

6.8.4 Chunk Pinning: Handling the Possibility of Entanglement

We now describe how to ensure that execution remains memory-safe, even at the moment when entanglement occurs. This requires care, because there is a race between LGC (which compacts thread-local memory by copying objects) and entanglement (which allows a thread to reach into another thread's local memory).

To illustrate the problem, consider two concurrent threads A and B as shown in Figure 6.7. Thread A begins executing a dereference `!x`, where `x` is shared between the two threads. By reading from `x`, thread A acquires a pointer to an object `y` which was allocated by B.[3] Next, suppose thread B performs a garbage collection which copies `y` to a new location `y'`, forwards the down-pointer (from `x`) to point to `y'`, and finally reclaims `y`. When A proceeds with the entanglement check on `y`, it will then read from reclaimed memory, and possibly crash (or worse).

Pinning. To make the entanglement check memory-safe, we have to ensure that it is safe for a concurrent thread to access the chunk descriptor of any object which is potentially entangled. Here we take advantage of a key property (discussed in Section 5.6): entanglement can only occur due to a read of a down-pointer. When MPL creates a down-pointer, it executes a write barrier (which adds the down-pointer to a remembered set for GC). In the write barrier, we mark the target object of each down-pointer as *pinned*; then, in LGC, we ensure that these objects are not moved (i.e. the address of a pinned object must not change during a collection). Any chunk that contains a pinned object is called a *pinned chunk*. By preserving the addresses of pinned objects, the GC implicitly ensures that pinned chunks are not reclaimed. Therefore, it is safe at any moment for a thread to access the descriptor of a pinned chunk, to inspect the vertex identifier stored there to detect entanglement.

In the example of Figure 6.7, the pinning technique fixes the race by ensuring that object `y` is pinned. The GC performed by thread B therefore is required to preserve `y` (and the chunk containing `y`), allowing the entanglement check performed by A to proceed safely.

Unpinning. A pinned object may be later unpinned as soon as there are no more down-pointers which point at the object. As heaps merge upwards due to thread joins, down-pointers

[3]At this point, it would be problematic for thread A to perform a garbage collection, because it is holding a cross-pointer. Our implementation therefore disallows GCs from being triggered within a read barrier, specifically by ensuring that no limit check is placed between the dereference and the entanglement check.

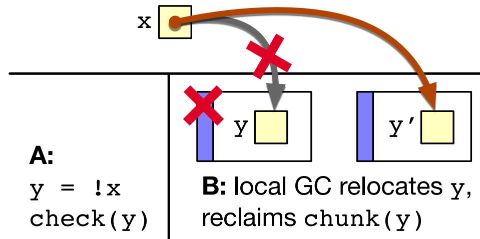

Figure 6.7: Example of race between local GC and the entanglement check. Thread A first acquires a pointer to y. Meanwhile, B forwards y to y' and reclaims the old memory. Thread A then proceeds with the entanglement check on a dangling pointer.

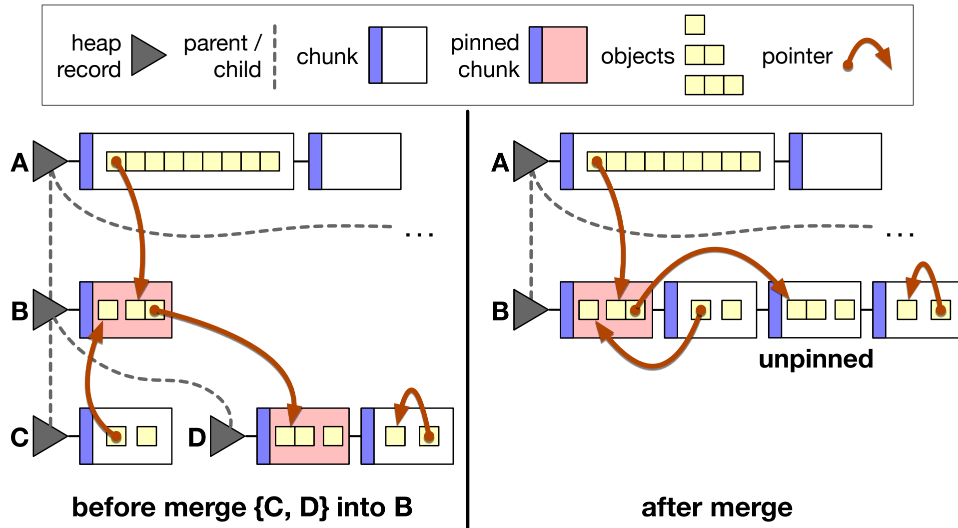

Figure 6.8: Example, before and after heaps **C** and **D** merge into **B**. Afterwards, the down-pointer from **B** into **D** has become an internal pointer, and therefore the indicated chunk may be unpinned.

naturally become internal, which enables unpinning. We could unpin objects at heap merges: whenever a heap is merged into its parent, if this causes all down-pointers incident upon a pinned object to become internal, then the object may be safely unpinned. However, unpinning objects at every heap merge would be inefficient, because merges are frequent. We therefore instead perform unpinning in bulk during GC.

Unpinning integrates naturally with MPL's handling of remembered sets (Section 6.6). During GC, MPL scans the remembered set to identify down-pointers that have become internal due to prior joins. During this process, we identify the set of objects which must remain pinned (due to the existence of a down-pointer). All other pinned objects are then unpinned.

Example. Figure 6.8 shows an example where pinned chunks become unpinned due to a heap merge. In the figure, there are four heaps shown on the left, labeled **A** through **D**. Each heap consists of a list of chunks, and within each chunk are various objects of different sizes allocated by the program, with pointers to other objects. Pinned chunks are shaded in light red: these chunks contain pinned objects, i.e., objects which are pointed-to by a down-pointer from an ancestor heap. On the left in the figure, there are two pinned chunks: one in heap **B**, and one in heap **D**. The right-side of the figure illustrates the heap structure after **C** and **D** are merged into their parent. As a result of the merge, a down-pointer (from **B** into **D**) becomes an internal pointer within **B**. The corresponding object therefore may be unpinned, as may its containing chunk (which no longer contains any pinned objects).

Cost of chunk pinning. Chunk pinning makes it possible to handle the possibility of entanglement. We measured the impact of pinning, in comparison to a promotion-based LGC algorithm (as described in Chapter 4), and found that overall it offers a slight improvement in time and space in comparison to promotion, with approximately −3% and −7% difference (respectively) in time and space on average across the benchmarks in our evaluation. This slight improvement may seem surprising, given that one disadvantage of pinning is that it introduces additional fragmentation into the system (by preventing the GC from relocating pinned objects to compact memory). However, note that this fragmentation is short-lived: as the program "joins back up", objects naturally become unpinned, as illustrated in Figure 6.8.

The space loss due to fragmentation appears to be outweighed by other advantages of pinning. For example, the pinning strategy allows for a more space-efficient implementation of remembered sets for down-pointers. In particular, for a promotion-based collection, MPL uses remembered-set entries of the form (x, i, y), indicating that $x[i]$ is a down-pointer to y; this representation allows for the GC to update $x[i]$ to point to the new version of y when y is relocated by collection. Under the pinning strategy, because the addresses of pinned objects are kept fixed (and thus down-pointers do not need to be updated by GC), we do not need to store this additional information in the remembered set. Therefore, under the pinning strategy, we instead use remembered-set entries of the form (x, y), where x is an object which contains a down-pointer to y. Then, by comparing $\mathit{heap}(x)$ against $\mathit{heap}(y)$, we can determine whether or not it is safe to unpin y. We therefore save space by shrinking the size of each individual remembered-set entry. Furthermore, if there are multiple down-pointers incident upon an object, we only need to keep one entry (the one with $\mathit{heap}(x)$ of minimum depth). These dif-

ferences can result in significant space savings in the remembered set, especially in programs with a large number of down-pointers.

Chapter 7

The Parallel ML Benchmark Suite

To evaluate the performance of MPL and provide examples of efficient and scalable Parallel ML programs, we developed the Parallel ML Benchmark Suite.[1] The suite consists of sophisticated parallel benchmarks from various problem domains, including graphs, text processing, digital audio processing, image analysis and manipulation, numerical algorithms, computational geometry, and others. We ported many of these benchmarks to Parallel ML from existing state-of-the-art parallel C/C++ benchmark suites and libraries including PBBS [14, 32, 133], ParlayLib [34, 155], Ligra [131], GBBS [54], and PAM [148]. Even though these benchmarks were originally written in C/C++, all of these benchmarks are naturally disentangled.

The Parallel ML Benchmark Suite is supported by MPLLib,[2] a library of key parallel algorithms and data-structures for parallel programming. MPLLib provides data structures such as sequences, sets, dictionaries, matrices, adjacency graphs, oct-trees, simplicial complexes (meshes), etc., and supports a wide variety of parallel operations on these data structures. The library also provides utilities for parallel I/O and processing for text, image (`.ppm`, `.gif`), and audio (`.wav`) files, as well as utilities for benchmarking.

In this chapter, we present an overview and discussion of our benchmarks and the library that supports them, including programming details for some of the key data structures and operations provided by MPLLib.

Acknowledgements

The Parallel ML Benchmark Suite includes contributions from multiple collaborators and colleagues, including Jatin Arora, Guy Blelloch, Umut Acar, Larry Wang, and Troels Henriksen.

7.1 Graph Algorithms

These algorithms are ported from Ligra [131] and GBBS [54]. We represent graphs in an adjacency table, specifically in a "compressed sparse row" format. In this representation, all edges of the graph are compressed into a single `vertex array`, and the graph additionally stores

[1] https://github.com/MPLLang/parallel-ml-bench
[2] https://github.com/MPLLang/mpllib

an array of offsets, indicating where the neighbors of a vertex are located. Specifically, the set of neighbors of a vertex v lie between indices `offsets[`v`]` and `offsets[`$v+1$`]` in the compressed neighbor array. The degree of a vertex therefore can also be efficiently computed as `offsets[`$v+1$`] - offsets[`v`]`. Here, vertices are labeled 0 to $N-1$.

Based on this representation, we implement an abstract graph interface, providing functions for querying degrees or retrieving the neighbors of vertices, both of which are supported with constant work. The graph library also provides efficient implementations of vertex subsets and Ligra-inspired parallel "edge map" operations.

Benchmarks. Our graph-based benchmarks are as follows.

- **Bfs** computes a breadth-first search. The input is a randomly generated power-law graph [44] with approximately 16.7M vertices and 199M edges, symmetrized.[3]
- **Centrality** computes single-source betweenness centrality. The input graph is the same as for the BFS benchmark.
- **Low-d-decomp** computes a low-diameter decomposition of a graph. The input graph is the same as for BFS.
- **Max-indep-set** computes a maximal independent set, i.e., a maximal set of non-adjacent vertices. The input graph is the same as for BFS.
- **Triangle-count** counts the number of triangles in an undirected graph. The input is a randomly generated power-law graph with approximately 1M vertices and 19.6M edges, symmetrized.

7.2 Computational Geometry

Some of the algorithms in this section share an implementation of an oct-tree, which organizes points into a rooted tree by recursively partitioning into rectangular regions. We implement oct-trees as an algebraic datatype, consisting of either a node with children, or a leaf. The leaves are "chunked" for efficiency; in particular, rectangular regions containing fewer than 50 points are represented as (unordered) arrays. This significantly improves space efficiency by avoiding redundant nodes, and improves cache efficiency for leaf-heavy operations. For example, in the nearest-neighbors benchmarks, the nearest neighbor of any point is likely stored at the same leaf.

The construction of the oct-tree is parallelized and included in the benchmark measurements. To construct an oct-tree, we partition points via a 2^D-way filter (for D-dimensional space; e.g., 4-way filter for 2-dimensional space) and then recursively build the children in parallel.

For delaunay, we implement a library which abstractly provides operations on meshes (a.k.a., simplicial complexes). The library supports key operations such as rip-and-tent, as well as mesh hopping, where the simplex containing a point can be found by walking through the mesh starting at any point, using "point-outside" queries to determine the direction towards

[3] A symmetrized graph is an undirected graph where each edge is represented as two directed edges.

any point from any simplex. To support mesh hopping, the simplex must be convex. Our library therefore provides primitives for constructing an appropriate convex boundary for any set of points.

The nearest-neighbors algorithm, in addition to being a standalone benchmark here, is also used as a subroutine for the delaunay algorithm, to aid in point location within a mesh. In particular, the delaunay algorithm iteratively inserts batches of points, and periodically between batches recomputes a nearest-neighbor data structure. This is used to locate the nearest neighbor within the current mesh, before mesh-hopping, to reduce the number of simplexes traversed by mesh-hopping.

For the range-query benchmark, we use Parallel Augmented Maps [148], which we ported to Parallel ML. Specifically we use weight-balanced binary trees, augmented for range queries as described in the PAM paper. In the future, this benchmark could be further optimized by using PaC-trees [55], which conceptually are a "chunked" version of PAM trees.

Benchmarks. Our computational geometry benchmarks are the following.

- **Delaunay** computes a Delaunay triangulation of 1M uniformly random points within a circle, using the algorithm by Blelloch, Gu, Shun, and Sun [33].
- **Nearest-nbrs** computes all nearest neighbors within a set of 2D points (i.e. for each point, the nearest other point within the set) by constructing an intermediate quad-tree and then querying it in parallel. The input is 1M points distributed uniformly randomly within a circle.
- **Quickhull** computes the convex hull of 20M uniformly random points distributed within a circle.
- **Range-query** performs rectangular queries on a collection of 2-dimensional points, where for each query, it counts the number of points within the rectangle. This is implemented using Parallel Augmented Maps [148]. In total there are 1M points in the database, and 1M queries on those points are performed in parallel.

7.3 Images and Audio

These benchmarks generally operate on either image data (of type `pixel array`, where pixels store RGB color information) or audio data (of type `real array`, where each element is a floating-point sample in the range $[-1, +1]$). To support these operations we implemented file support for `.ppm`, `.gif`, and `.wav` file types. The parallel versions of both the reverb and seam-carving algorithms are described in detail in the author's online blog.[4] At a high level, the reverb algorithm is similar to a parallel prefix sums algorithm, but specialized for 2-dimensional data, where the size of second the dimension is determined by the reverb delay parameter. Parallel seam-carving is accomplished by a dynamic programming algorithm which partions the image into triangular-blocked strips, which is more efficient (in terms of both cache efficiency and parallelism) than the simpler row-major parallelization strategy, providing up to 5x performance improvement at scale.

[4] https://shwestrick.github.io/

Benchmarks. Our image and audio processing benchmarks are as follows.

- **Raytracer** computes an image of 1000×1000 pixels by ray-tracing. This implementation is provided by Troels Henriksen.[5]
- **Seam-carve** is a parallel implementation of the seam-carving technique [22] for content-aware rescaling. This benchmark removes 100 vertical seams from a panoramic image of approximately 1.5M pixels.
- **Tinykaboom** is a port of a C++ graphics application[6] which procedurally generates an explosion effect. It generates 10 frames of 100×100 pixels each.
- **Reverb** applies an artificial reverberation effect to an audio file. The input is approximately 4 minutes long with a sample rate of 44.1 kHz at 16 bits per sample.

7.4 Text Processing

- **Dedup** takes a collection of words and removes duplicates by deterministic hashing. The input text is approximately 148MB with 16.5M words and 65K unique words.
- **Grep** is a parallel implementation of the Unix `grep` utility, with parallelization both across the lines as well as within long lines. In particular, we perform a prefix-doubling parallel search within long lines to find the first occurrence of the pattern. The benchmark simply performs a direct comparison to match against the pattern, and for simplicity does not support regular expressions. The input text is approximately 148MB with 16.5M lines and 386K matching lines.
- **Wc** is a parallel implementation of the Unix `wc` utility, which computes the number of lines, the number of words, and the number of bytes in a text file. The input text is approximately 1.7GB with 216M lines.
- **Suffix-array** computes the suffix array of a uniformly random input text of 10M characters.
- **Palindrome** finds the longest (contiguous) substring which is a palindrome using a polynomial rolling hash. The input is 1M characters.
- **Tokens** separates a text into tokens, using whitespace as delimiters. The input text is approximately 148MB with 16.5M tokens.

7.5 Numerical Algorithms

- **Dense-matmul** multiplies two 1024×1024 dense matrices of 64-bit floating-point elements using the simple $O(n^3)$-work, four-way divide-and-conquer cache-oblivious algorithm.
- **Sparse-mxv** multiplies a sparse matrix (2M rows, 200M nonzero entries) with a vector of length 2M.

[5] https://github.com/athas/raytracers
[6] https://github.com/ssloy/tinykaboom

- **Primes** generates all prime numbers that are less than 100M (approximately 5.8M primes) with a parallel sieve.
- **Bignum-add** performs addition on two bignums of 500M bytes each. Bignums are represented as sequences of bytes, where each byte represents one digit in radix 128. This leaves one bit leftover in each byte, which is used in the algorithm as a carry bit.
- **Integrate** calculates the integral of $\sqrt{1/x}$ for $x \in [1, 1000]$ with a midpoint-rectangular Riemann sum across $n = 500\text{M}$ points.
- **Linearrec** solves a linear recurrence of the form $R_i = x_i R_{i-1} + y_i$; the input is 200M pairs of doubles (x_i, y_i).
- **Linefit** finds a line of best fit (by least-squares) for 500M 2D points (pairs of doubles).
- **Mcss** computes the maximum contiguous subsequence sum of an array of 500M doubles.

7.6 Other Algorithms

- **Msort-strings** performs parallel mergesort on a collection of strings taken from an English corpus. The input is 38MB with 4.1M strings of approximately 10 characters each on average.
- **Msort-int64** performs parallel mergesort on 64-bit integers. The input contains 20M uniformly random integers, generated by a hash function.
- **Nqueens** counts the number of solutions to the 13×13 n-queens problem.

Chapter 8

Evaluation

To evaluate the performance of MPL, we perform a number of empirical evaluations across over 30 benchmarks from the Parallel ML Benchmark Suite (Chapter 7). Collectively, these comparisons demonstrate that MPL is efficient and scalable in terms of both time and space usage. In particular, we show that MPL generally outperforms existing state-of-the-art procedural and functional parallel language implementations which have automatic memory management, including Java, Go, and multicore OCaml. Furthermore, we show that MPL is competitive with low-level, memory-unsafe languages such as C++.

8.1 Overview

Overheads and Scalability

In Section 8.3, we begin by comparing MPL against the MLton [106] compiler for (sequential) Standard ML. Because MPL extends MLton, the two systems are able to compile exactly the same benchmarks[1] and are very similar in terms of the details of compilation itself. The main difference between MLton and MPL is their runtime systems: whereas MLton only supports sequential execution and has a sequential GC, our MPL is fully parallel. Therefore, this comparison allows us to determine the overheads and scalability of our memory management techniques, in comparison to an efficient sequential baseline. We consider a number of differences in memory management between MPL and MLton, especially regarding heap architecture and garbage collection policy.

On executions using up to 72 processors, we observe that MPL achieves between 14-60x speedup over (sequential) MLton while on average using less space. Note that in general, parallel execution can require more memory than sequential execution—as much as a factor P on P processors, in theory. MPL therefore is able to efficiently manage memory to avoid this blowup when possible, without significant impact on runtime scalability.

[1] Specifically, we use MLton to compile the sequential elision of each benchmark, where parallel tuples are replaced by sequential tuples.

Problem-Based Cross-Language Comparisons

To determine how MPL fares against other state-of-the-art parallel language implementations, we perform a number of "problem-based" comparisons. The problem-based methodology, as popularized by PBBS [14, 133], considers benchmarks with well-defined input and output specifications. That is, if program A and program B both solve the same problem (by having the same input and output, down to individual bits, if possible), then it is reasonable to compare them broadly in terms of performance, even in A and B are based on different languages and systems. We use this approach to compare MPL against a multiple other parallel languages, including multicore OCaml, Java, Go, and C++.

OCaml. In Section 8.4, we compare MPL against multicore OCaml [139], a state-of-the-art parallel functional language implementation. Multicore OCaml supports fork-join parallelism, and has a source-level language which is essentially the same as MPL. Therefore, for this comparison, we are able to use very similar benchmark codes. We ported multiple of our benchmarks to OCaml, and also ported multiple OCaml benchmarks to MPL. The results of this comparison show that MPL is approximately 2x faster while also using 2x less space at scale (on 72 processors).

Java and Go. In Section 8.5, we compare against Java and Go to determine how MPL's performance fares against well-known memory-safe procedural languages. In these comparisons, we consider the classic problem of sorting, where we compare the fastest parallel sorting implementations we could find in each language. We also consider five other highly parallel benchmarks representing a variety of characteristics, including both compute-bound and memory-bound benchmarks, as well as benchmarks with both low and high rates of allocation.

We observe that, although both Java and Go are sometimes faster than MPL on one processor, they do not scale as well. At scale (on 72 processors), MPL is on average 2x faster than Go and over 3x faster than Java. MPL also uses less space than both Go and Java on average (approximately 4x less than Java and 30% less than Go). Finally, we note that the running time performance advantage of MPL is highest for benchmarks with a high rate of allocation. This results show that MPL generally outperforms both Java and Go (in terms of both time and space), often by a wide margin, and the performance advantage appears to be due to efficient and scalable memory management.

C++. All of the above comparisons consider languages that are memory-safe and have automatic memory management. However, memory-unsafe languages such as C++ are often favored by programmers for achieving high performance, expecially in the world of shared-memory multicore parallelism. Therefore, it is interesting to consider how MPL fares in comparison to C++, to establish a ballpark number for MPL's "absolute" performance. In Section 8.6, we perform such a comparison using the same problem-based methodology as the rest of our experiments. The C++ benchmarks in this comparison are taken from state-of-the-art suites and libraries, including PBBS [14, 32, 133], ParlayLib [34], and Ligra [131].

We note that although this comparison is unfair against MPL (due to the additional safety guarantees provided by MPL), we do not focus here on language-level differences between MPL

and C++. Rather, the point of this comparison to determine whether or not MPL is competitive in performance with the fastest available low-level programming techniques.

At a high level, we observe that on 72 processors, MPL is on average less than 2x slower than C++ while using a similar amount of memory. In three cases, MPL is within 30% of C++, and in one case, MPL matches the performance of C++. These measurements include the cost (both space and time) of automatic memory management and GC. This comparison demonstrates that MPL is generally competitive with (i.e., within a factor 2 of) the fastest low-level parallel programming techniques. With further engineering, we believe this performance gap can be narrowed, and eventually closed.

Overhead of Entanglement Detection

As described in Chapter 5, MPL achieves memory safety via entanglement detection, which monitors individual reads and writes during execution to ensure disentanglement. In Section 8.7, we measure the overhead of these techniques by measuring MPL's performance both with and without detection (i.e, by enabling and disabling detection at compile-time). Across the board, we observe close to zero overhead in terms of both time and space. We also consider the performance improvement due to our *entanglement candidates* algorithm (Section 5.6), which dynamically eliminates unnecessary graph queries when checking for entanglement. We observe that, by tracking candidates, we are able to bring the number of graph queries down to 0 in most cases, and achieve performance improvements of up to a factor of 2x. The entanglement candidates algorithm is therefore key to the low overhead of entanglement detection.

8.2 Methodology and Experimental Setup

To measure run times, we run each benchmark 20 times back-to-back and report the average, excluding initialization (e.g., loading the input from file), warmup, and teardown. To measure space usage, we measure the average of the maximum resident set size (as reported by Linux) of 20 back-to-back runs of the benchmark. Back-to-back runs are executed in the same program instance to ensure that the effect of memory management amortization thresholds is taken into account (for example, a garbage collection might run only once every five runs). We write T_P for time on P processors, and similarly R_P for the max residency on P processors. For the sequential baseline runs, we write T_s. Unless stated otherwise, all run times are in seconds, and all space numbers (max residencies) are in GB.

We run all of our experiments on a 72-core Dell PowerEdge R930 consisting of 4×2.4GHz Intel (18-core) E7-8867 v4 Xeon processors, 1TB of memory, and running Ubuntu version `16.04.7` with Linux version `4.10.0-40-generic x86_64`. All of our MPL experiments use MPL version `0.3`. In Section 8.4, we use multicore OCaml version `5.0.0+dev4-2022-06-14` with default settings and the library `domainslib` version `0.4.2`. In Section 8.5, we use Java OpenJDK version `11.0.14` with the `G1GC` collector, which we found yielded the best performance. We used the following runtime settings to control the number of threads:

```
-XX:ParallelGCThreads=N
-Djava.util.concurrent.ForkJoinPool.common.parallelism=N
```

For Go, we use version `1.18.4`. In Section 8.6, we use `g++` version `10.3.0` with the `jemalloc` library, and the compiler flags `-O3`, `-march=native`, `-std=c++17`, and `-mcx16`.

The code for these experiments is publicly available on GitHub.[2]

8.3 Overheads and Scalability

We evaluate the end-to-end scalability of our memory management techniques by comparing against the MLton compiler for Standard ML. Because MPL extends MLton, the two systems are very similar. As baseline we use the sequential versions of our benchmarks, which are derived from parallel benchmarks by replacing parallel tuples with sequential ones. This approach enables us to measure the cost of our memory management techniques by keeping the underlying algorithms the same for both parallel and sequential versions.

The results of this comparison are shown in Table 8.1 and Figures 8.1 and 8.2. Figures 8.1 and 8.2 show the speedup of our benchmarks in comparison to the sequential baseline on up to 72 processors; we split these into two separate plots to make them easier to read. Table 8.1 shows measurements for sequential, uniprocessor runs ($P = 1$), and parallel runs with 72 processors ($P = 72$), including time and space usage, as well as overheads, speedups, and memory blowups.

Overhead. The overhead—measured by the ratio T_1/T_s of uniprocessor to sequential time—is relatively small, ranging between a factor of 1 (no overhead) to 4.27 (327% slower) in comparison to the sequential baseline, with a geometric average of 1.5x (see "geomean" in Table 8.1). These overheads include all overheads of parallelism, including the cost of scheduling as well as parallel memory management. The largest overhead in comparison to MLton is on the `primes` benchmark (approximately 4x). We verified that this overhead is due to compilation (rather than memory management or scheduling) by adjusting MPL's compile-time thresholds for function specialization[3] and measuring the performance of `primes` across different settings. With a larger threshold, the uniprocessor overhead of `primes` becomes 25%, and the speedup on 72 processors increases to 22x.

Speedup. The speedup—measured by the ratio T_s/T_P of the sequential running time to the P-processor running time—shows the benefits of parallelism over the sequential. In Table 8.1, we observe between 14-60x speedup, with an average of 30x. A speedup of $T_s/T_P = P$ would indicate perfect speedup, i.e., full utilization of all P processors. Perfect speedup is uncommon, even for embarrassingly parallel benchmarks, due to overheads of parallelism and memory bottlenecks on modern multicores. Typically, we expect to see speedups scale linearly with the number of processors but then plateau as the memory bandwidth of the machine is reached, particularly for "memory-bound" benchmarks.

[2] `https://github.com/MPLLang/parallel-ml-bench`

[3] Specifically, we increased the `-polyvariance-small` threshold. This setting is inherited from MLton, and controls the "polyvariance" optimization pass, which duplicates higher-order functions at each use, if the size of the function is smaller than some threshold. By increasing this threshold, we allow the compiler to inline and specialize larger higher-order functions, which increases code size but can substantially improve performance in some cases.

	Time (s)					Space (GB)				
			OV		SU			BU_1		BU_{72}
Benchmark	T_s	T_1	$\frac{T_1}{T_s}$	T_{72}	$\frac{T_s}{T_{72}}$	R_s	R_1	$\frac{R_1}{R_s}$	R_{72}	$\frac{R_{72}}{R_s}$
bfs	11.4	18.7	**1.64**	.420	**27**	35	7.4	**0.2**	5.7	**0.2**
bignum-add	2.47	4.05	**1.64**	.069	**36**	8.6	3.0	**0.3**	3.1	**0.4**
centrality	14.5	18.6	**1.28**	.466	**31**	33	6.6	**0.2**	5.7	**0.2**
dedup	3.05	6.14	**2.01**	.127	**24**	11	2.8	**0.3**	8.9	**0.8**
delaunay	8.24	11.9	**1.44**	.379	**22**	2.7	1.1	**0.4**	2.1	**0.8**
dense-matmul	1.88	2.85	**1.52**	.048	**39**	.11	.046	**0.4**	.091	**0.8**
grep	1.44	2.12	**1.47**	.040	**36**	4.6	.61	**0.1**	.85	**0.2**
integrate	3.06	3.11	**1.02**	.052	**59**	.0020	.013	**6.5**	.050	**25.0**
linearrec	3.34	5.02	**1.50**	.226	**15**	77	6.7	**0.1**	21	**0.3**
linefit	2.00	2.42	**1.21**	.146	**14**	8.5	8.2	**1.0**	8.2	**1.0**
low-d-decomp	6.67	8.24	**1.24**	.215	**31**	28	6.8	**0.2**	6.1	**0.2**
max-indep-set	12.1	16.3	**1.35**	.341	**35**	18	6.9	**0.4**	5.5	**0.3**
mcss	1.90	4.64	**2.44**	.078	**24**	4.3	4.1	**1.0**	4.1	**1.0**
msort-int64	3.36	3.98	**1.18**	.077	**44**	5.0	.66	**0.1**	.91	**0.2**
msort-strings	1.31	3.05	**2.33**	.070	**19**	1.5	.67	**0.4**	1.7	**1.1**
nearest-nbrs	1.31	1.67	**1.27**	.038	**34**	1.5	.72	**0.5**	2.2	**1.5**
nqueens	1.23	1.64	**1.33**	.029	**42**	.0020	.044	**22.0**	.26	**130.0**
palindrome	1.09	1.64	**1.50**	.032	**34**	.20	.061	**0.3**	.10	**0.5**
primes	1.74	7.43	**4.27**	.122	**14**	1.5	.26	**0.2**	.37	**0.2**
quickhull	2.53	3.51	**1.39**	.113	**22**	15	13	**0.9**	20	**1.3**
range-query	14.5	17.0	**1.17**	.297	**49**	13	4.4	**0.3**	4.2	**0.3**
raytracer	2.93	3.28	**1.12**	.057	**51**	.39	.090	**0.2**	.41	**1.1**
reverb	1.00	1.32	**1.32**	.042	**24**	6.9	1.5	**0.2**	1.9	**0.3**
seam-carve	12.0	15.5	**1.29**	.859	**14**	.54	.090	**0.2**	.56	**1.0**
sparse-mxv	1.33	2.45	**1.84**	.049	**27**	2.1	4.5	**2.1**	4.4	**2.1**
suffix-array	4.36	7.19	**1.65**	.146	**30**	6.3	1.2	**0.2**	1.8	**0.3**
tinykaboom	2.45	2.46	**1.00**	.041	**60**	.0054	.0073	**1.4**	.038	**7.0**
tokens	1.56	1.93	**1.24**	.043	**36**	13	1.1	**0.1**	1.1	**0.1**
triangle-count	4.93	5.37	**1.09**	.113	**44**	2.7	.79	**0.3**	1.2	**0.4**
wc	4.18	7.69	**1.84**	.130	**32**	1.9	3.6	**1.9**	3.6	**1.9**
geomean			**1.48**		**30**			**0.43**		**0.73**

Table 8.1: Comparison with sequential baseline: times, max residencies, overheads (OV), speedups (SU), and space blowups (BU).

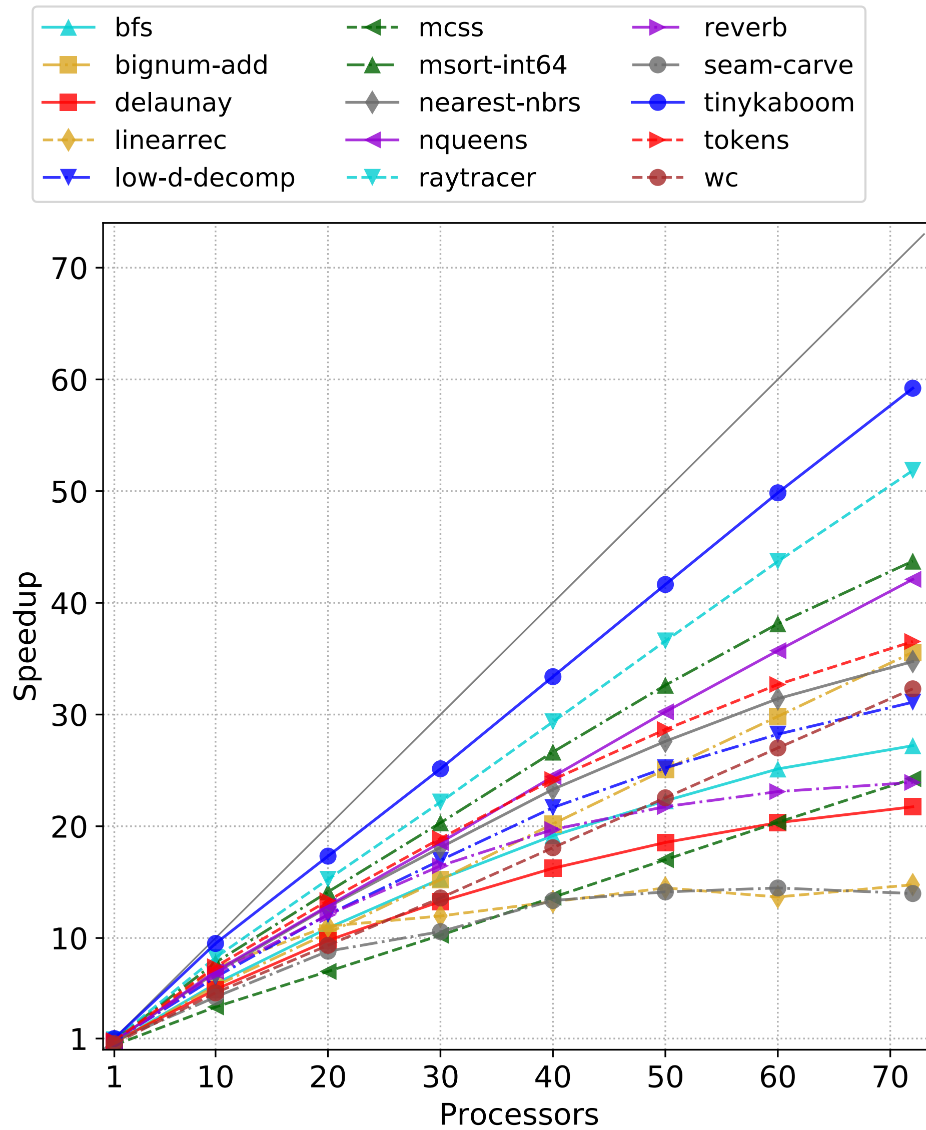

Figure 8.1: Speedups in comparison to sequential baseline (group 1).

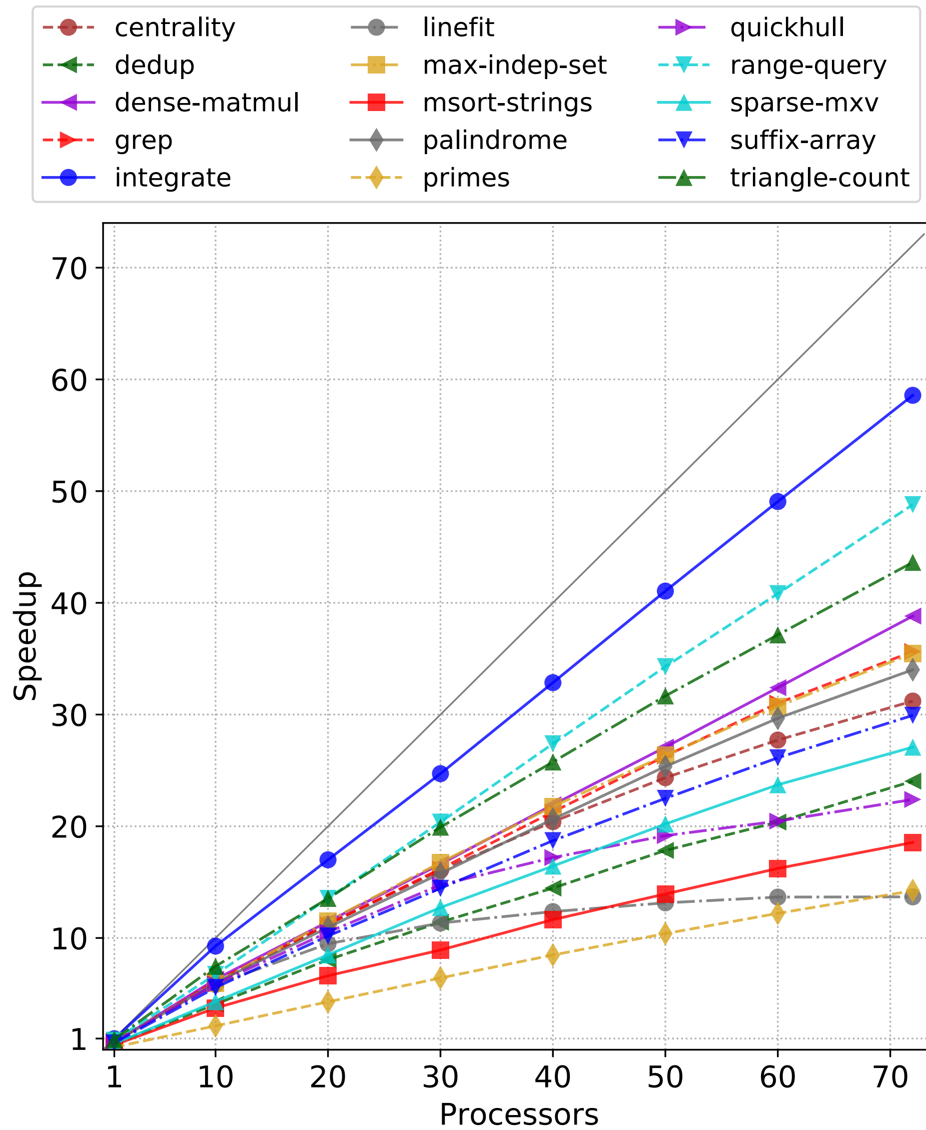

Figure 8.2: Speedups in comparison to sequential baseline (group 2).

In Figures 8.1 and 8.2, we observe two primary behaviors, as expected. Most benchmark can be classified as either compute-bound or memory-bound: the compute-bound benchmarks (e.g. raytracer, tinykaboom, integrate, dense-matmul) all scale approximately linearly, whereas the memory-bound benchmarks (e.g. seam-carve, reverb, delaunay, linefit) each initially scale linearly and then plateau as the memory bandwidth of the machine is reached.

One example of a memory-bound benchmark is linefit. Despite low overall speedup (only 14x on 72 processors), MPL's parallel performance on this benchmark is nearly optimal.[4] In particular, consider that the linefit algorithm has to go over the input data twice; given that each element is 16 bytes and the input is 500M elements, the total number of bytes that need to be read is 16 GB. Our machine has a peak read bandwidth of 140 GB/sec, which implies a peak performance of $16/140 = .114$ seconds. We achieve .146 seconds on 72 processors, which is reasonably close to the peak bandwidth.

Another example of a memory-bound benchmark is seam-carve, which has relatively low speedup (14x on 72 processors) due to three factors. First, seam-carving not highly parallel: in an image of width w and height h, seam-carving has $O(wh)$ work and $O(h)$ span, leaving only $O(w)$ parallelism, which for typical images is small. Second, seam-carving only does a small amount of compute (a few arithmetic instructions) per memory access. Third, seam-carving has a high allocation and reclamation rate: this particular implementation is "pure" in the sense that removing one seam does not modify the input image, so in total the benchmark allocates approximately 100 copies of the input image, which stresses the memory management system. In light of these bottlenecks, a speedup of 14x for seam-carving is admirable.

The `primes` benchmark also has reasonably low speedup, which is explainable entirely due to its uniprocessor overhead, discussed above. Indeed, on this benchmark, MPL has an excellent "self-speedup" of $T_1/T_{72} = 61$, and looking closely at the speedup of `primes` in Figure 8.2, we observe an almost perfectly straight line.

These results show that the speedups are significant and that they are usually inversely correlated with the overheads, showing that the scalability is overall quite good.

Space Efficiency. The space blowup on P processors–defined by the ratio R_P/R_s of P-processor space usage to sequential–summarizes the additional memory required to support parallelism and parallel memory management.

In Table 8.1, we immediately observe that MPL has low blowups across the board in comparison to the sequential baseline. Surprisingly, on average, MPL uses *less* memory than MLton on both uniprocessor and parallel runs. This is due to MPL's collection policy, which is more eager than MLton's, in anticipation of parallelism. By default, MLton uses a single heap with an amortization ratio of 8x, and therefore will use up to 8x more memory than the working set size. MPL uses two amortization ratios—one for LGC, and one for CGC, as described in Section 6.7.3—and is more eager for CGC, with an amortization ratio of 2x. In this way, MPL gets a space benefit both from generational effects (where LGC essentially functions as a fast nursery collector) as well as from a lower overall amortization ratio.

We have observed that, by increasing MPL's GC amortization ratios, we can decrease the

[4] In our later comparison with C++ (Section 8.6), we observe that MPL matches the performance of C++ on the linefit benchmark.

sequential overhead of MPL relative to MLton while increasing the memory blowup. For example, increasing the CGC amortization ratio to 8x brings the sequential time overhead of nearest-nbrs to within 20% but increases the sequential memory blowup to 1.5x; similarly, on dedup, the sequential time overhead improves slightly (down to 1.8x from 2x) while the memory usage approximately doubles. However, at scale, we do not witness an improvement in running time from this change. This is why we choose to make MPL more eager with CGC by default: improved space performance with little effect on running time at scale. We note also that because of MPL's chunked heaps, there is a potential space benefit for large objects stored in single-object chunks, which do not need to be copied during LGC, as described in Section 6.3. We plan to investigate the performance advantages of this optimization in future work.

On 72 processors, MPL achieves blowups in the range 0.1-2.1x in all but three cases. The three exceptions are integrate (25x), nqueens (130x), and tinykaboom (7x). We attribute these blowups to the unavoidable costs of parallelism, including a constant overhead (the memory of the scheduler) as well as a multiplicative factor: on P processors, any parallel algorithm might need as much as a factor of P more memory than sequential. All three benchmarks have small overall memory usage on 1 processor, which amplifies the cost of any constant overheads in the observed blowups. Note that MPL has "self-blowups" R_{72}/R_1 in the range 4-6x for these benchmarks. Self-blowup can be a useful measure for the multiplicative space cost of parallelism. In these cases, it is low in comparison to the number of processors, P.

For 18 out of 30 benchmarks, MPL uses significantly less space on 72 processors than MLton uses sequentially. This suggests that MPL's memory management and garbage collection strategies are space-efficient and scalable. Furthermore, MPL's low overheads and good speedups (as discussed above) demonstrate that the runtime cost of automatic memory management is well-amortized and does not interfere with parallelism.

8.4 Comparison with Multicore OCaml

The surface languages supported by the MPL compiler and OCaml are essentially the same, raising the question of how the two systems compare, and making it possible to perform such a comparison by using very similar benchmarks. In this section, we perform a modest comparison between MPL and OCaml. Before we dive into details, we note that beyond the similarity of the languages, the two systems have different aims. Our main goal is to develop a fully parallel/distributed memory manager that comes with theoretical guarantees on efficiency. The multicore extensions to the OCaml compiler prioritize backward compatibility with sequential codes, especially with respect to performance [139]. The currently available version of multicore OCaml uses a generational collector with a concurrent, non-moving major GC and a stop-the-world parallel minor GC.

Benchmarks. We consider four OCaml-specific benchmarks taken from the OCaml Sandmark suite,[5] which we ported to MPL. These benchmarks have the prefix "SM:" in tables and figures. We also use five benchmarks from Section 8.3: raytracer, primes, msort-int64, msort-

[5] https://github.com/ocaml-bench/sandmark

	T_1			T_{72}			R_1			R_{72}		
	O	M	$\frac{O}{M}$	O	M	$\frac{O}{M}$	O	M	$\frac{O}{M}$	O	M	$\frac{O}{M}$
tokens	4.87	1.93	**2.52**	.819	.043	**19.05**	1.5	1.1	**1.36**	14	1.1	**12.73**
msort-int64	11.2	3.98	**2.81**	.479	.077	**6.22**	1.1	.66	**1.67**	6.8	.91	**7.47**
primes	8.20	7.43	**1.10**	.168	.122	**1.38**	.45	.26	**1.73**	1.7	.37	**4.59**
raytracer	5.30	3.28	**1.62**	.161	.057	**2.82**	.080	.090	**0.89**	.41	.41	**1.00**
SM:lu-decomp	2.96	1.21	**2.45**	.256	.154	**1.66**	.048	.063	**0.76**	.41	.21	**1.95**
msort-strings	4.10	3.05	**1.34**	.119	.070	**1.70**	.36	.67	**0.54**	1.7	1.7	**1.00**
SM:nbody	3.19	4.33	**0.74**	.227	.192	**1.18**	.0060	.0087	**0.69**	.15	.096	**1.56**
SM:game-of-life	3.07	1.70	**1.81**	.073	.079	**0.92**	.021	.054	**0.39**	.17	.12	**1.42**
SM:binarytrees5	1.25	1.84	**0.68**	.088	.096	**0.92**	.067	.080	**0.84**	.76	1.1	**0.69**
geomean			**1.50**			**2.22**			**0.88**			**2.22**

Table 8.2: MPL vs OCaml: Times (seconds) and max residencies (GB) of MPL (column **M**) and OCaml (column **O**). The ratios $\frac{O}{M}$ are the performance of OCaml relative to MPL. Larger ratios are better for MPL.

strings, and tokens. The raytracer implementations are taken from Troels Henriksen's comparison.[6] We ported the four other benchmarks (primes, msort-int64,[7] msort-strings, and tokens) from MPL to OCaml. Because of the similarities between Parallel ML and OCaml, and because multicore OCaml supports fork-join parallelism, the resulting benchmark codes are very similar.

Results. Table 8.2 presents the results of the comparison, with running times T_p and max residency R_p on both $p = 1$ and $p = 72$ processors. The columns labeled "O" are results for OCaml, and the columns "M" are results for our MPL. The column $\frac{O}{M}$ is the performance of OCaml relative to MPL; larger ratios are in favor of MPL.

In terms of running times, MPL is generally faster than OCaml and scales better. On 1 core, OCaml requires approximately 1.5x more time than MPL. On 72 cores, the gap is approximately 2.2x, and MPL is faster than OCaml in all but two benchmarks (where the gap is less than 10%). In terms of memory usage, MPL is generally competitive with OCaml. On 1 core, MPL uses approximately 15% more space on average. On 72 cores, OCaml uses 2x more space (1.75x when exluding the maximum "outlier"). The results show that MPL performs well in practice, and the fully parallel/distributed GC of MPL can deliver good overall scalability, work efficiency, and space efficiency.

The largest difference in both running time and memory is the tokens benchmark, where MPL is 19x faster than OCaml and uses 13x less space. To identify the reason, we measured space usage across different numbers of repetitions of the tokens benchmark on 72 processors. (In our experiments, we average over 20 repetitions to obtain reliable timings and account for amortization in memory management policies.) On a single repetition, both MPL and OCaml use approximately 1GB of memory. As the number of repetitions increases, the space usage of

[6]https://github.com/athas/raytracers

[7]For the msort-int64 benchmark, we allow OCaml to use its native `int` type, which is 63 bits wide.

MPL stays approximately constant. In contrast, OCaml's memory usage increases linearly with repetitions. For example, OCaml uses approximately 1GB to run the benchmark once, 1.7GB to run it twice back-to-back, 2.5GB to run it three times, etc. We verified that this trend continues: OCaml needs approximately 18GB to complete 40 repetitions. Investigating further, we inserted an explicit call to `Gc.full_major()` between each repetition. This brought the memory usage down to approximately 1.5GB and improved the run time on 72 processors by a factor of 2 (still this is 50% more space and 9x slower compared to MPL).

8.5 Comparison with Java and Go

We compare with Java and Go, two industry-strength memory-safe procedural languages with well-engineered automatic memory management. Both languages support fork-join parallelism out-of-the-box, which makes it possible to write efficient divide-and-conquer parallel programs as well as parallel loop-based algorithms. Examples of parallel constructs in both Java and Go are shown in Figures 8.3 and 8.4, and described in more detail below.

For these comparisons, we consider seven of our problem-based benchmarks: linefit, mcss, msort-int64, msort-strings, primes, sparse-mxv, and tokens. All of these benchmarks are highly parallel, and represent a variety of characteristics, including both compute-bound and memory-bound benchmarks, as well as benchmarks with both low and high rates of allocation. (For example, mcss is compute-bound, linefit is memory-bound, primes has a low rate of allocation, and tokens has a high rate of allocation.) We also consider the classic problem of sorting (msort-int64 and msort-strings benchmarks), where we compare the fastest parallel sorting algorithms we could find.

At a high level, the takeaway from these comparisons is that MPL can outperform modern memory-safe procedural languages, often by a wide margin, and the performance advantage appears to be due to memory management. In particular, although both Java and Go are sometimes faster than MPL on one processor, they do not scale as well. At scale (on 72 processors), MPL is faster than both Java and Go on all benchmarks, and on average is 2x faster than Go and over 3x faster than Java. The performance advantage of MPL is largest for benchmarks that have a high rate of allocation, such as the tokens benchmark. Furthermore, MPL uses less space than both Go and Java in almost all cases (both uniprocessor and parallel runs). On 72 processors, MPL uses on average approximately 30% less space than Go and 4x less space than Java. These results suggest that MPL's main performance advantage is due to efficient and scalable memory management.

Java Benchmarks and Results

In Java, we implement benchmarks using the Fork/Join Framework [97] and `java.util.stream` library which provides parallel operations on logical streams of data. (The Java streams library is supported under the hood by the Fork/Join framework.) For example, the code shown in Figure 8.3 demonstrates a couple examples of how the Java Streams can be used for parallelism. For the sorting comparisons (msort-int64, msort-strings), we use the standard `parallelSort` function from the `java.util.Arrays` library, which is provided as part of the Fork/Join frame-

```
IntStream.range(0, N).parallel().forEach(i → {
  // ... body of parallel for-loop, where 0 ≤ i < N
});

int[] filterResults =   // parallel filter
  IntStream.range(0, N).parallel()
  .filter(i → { /* ... predicate using i ... */ })
  .toArray(); // output results as an array
```

Figure 8.3: Two examples of loop-based parallelism using Java Streams, both of which operate on integers i in the range $0 \leq i < N$. The former is a parallel for-loop, the latter a parallel filter.

```
func pardo(f, g func()) {
  done := make(chan bool)
  go func(){g(); done<-true}()
  f()
  <-done
}
```

Figure 8.4: Example binary fork-join in Go. The expression `pardo(f,g)` runs the functions `f` and `g` in parallel using goroutines (Go's lightweight threads) and channel synchronization.

	T_1			T_{72}			R_1			R_{72}		
	J	M	$\frac{J}{M}$	J	M	$\frac{J}{M}$	J	M	$\frac{J}{M}$	J	M	$\frac{J}{M}$
linefit	12.6	2.42	**5.21**	1.07	.146	**7.33**	28	8.2	**3.41**	30	8.2	**3.66**
mcss	6.29	4.64	**1.36**	.441	.078	**5.65**	11	4.1	**2.68**	30	4.1	**7.32**
msort-int64	2.28	3.98	**0.57**	.263	.077	**3.42**	4.2	.66	**6.36**	2.4	.91	**2.64**
msort-strings	.947	3.05	**0.31**	.075	.070	**1.07**	1.1	.67	**1.64**	1.3	1.7	**0.76**
primes	8.90	7.43	**1.20**	.181	.122	**1.48**	1.2	.26	**4.62**	1.8	.37	**4.86**
sparse-mxv	1.71	2.45	**0.70**	.086	.049	**1.76**	15	4.5	**3.33**	17	4.4	**3.86**
tokens	4.01	1.93	**2.08**	.458	.043	**10.65**	19	1.1	**17.27**	20	1.1	**18.18**
geomean			**1.12**			**3.29**			**4.26**			**4.06**

Table 8.3: MPL vs Java: Times (seconds) and max residencies (GB) of MPL (column **M**) and Java (column **J**). The ratios $\frac{J}{M}$ are the performance of Java relative to MPL. Larger ratios are better for MPL.

work.

Results. Table 8.3 presents the results of the Java comparison, with running times T_p and max residency R_p on both $p = 1$ and $p = 72$ processors. The columns labeled "J" are results for Java, and the columns "M" are results for our MPL. The column $\frac{J}{M}$ is the performance of Java relative to MPL; larger ratios are in favor of MPL.

On 72 processors, MPL is faster than Java on all benchmarks, with an average of 3.3x. In terms of space, MPL uses less space in all cases except one. The performance gap (for both space and time) is often wide, especially for space, where MPL uses 4x less space than Java on average. On uniprocessor runs, MPL and Java have similar performance on average. There are a three benchmarks where Java is faster than MPL on a single processor (msort-* and sparse-mxv benchmarks). The largest gap on a single processor is msort-strings, where Java is 3x faster than MPL. However, on these benchmarks, MPL scales better: for example, on 72 processors, MPL is 10% faster on msort-strings, 75% faster on sparse-mxv, and 3x faster than Java on msort-int64.

	T_1			T_{72}			R_1			R_{72}		
	G	M	$\frac{G}{M}$	G	M	$\frac{G}{M}$	G	M	$\frac{G}{M}$	G	M	$\frac{G}{M}$
linefit	7.24	2.42	**2.99**	.168	.146	**1.15**	9.3	8.2	**1.13**	9.5	8.2	**1.16**
mcss	6.21	4.64	**1.34**	.106	.078	**1.36**	4.8	4.1	**1.17**	4.9	4.1	**1.20**
msort-int64	3.41	3.98	**0.86**	.266	.077	**3.45**	1.4	.66	**2.12**	2.0	.91	**2.20**
msort-strings	1.32	3.05	**0.43**	.111	.070	**1.59**	.67	.67	**1.00**	1.0	1.7	**0.59**
primes	2.47	7.43	**0.33**	.140	.122	**1.15**	.45	.26	**1.73**	.53	.37	**1.43**
sparse-mxv	4.85	2.45	**1.98**	.089	.049	**1.82**	6.8	4.5	**1.51**	6.1	4.4	**1.39**
tokens	4.38	1.93	**2.27**	.478	.043	**11.12**	2.5	1.1	**2.27**	2.5	1.1	**2.27**
geomean			**1.12**			**2.13**			**1.49**			**1.35**

Table 8.4: MPL vs Go: Times (seconds) and max residencies (GB) of MPL (column **M**) and Go (column **G**). The ratios $\frac{G}{M}$ are the performance of Go relative to MPL. Larger ratios are better for MPL.

The largest gap, in terms of both time and space, is the tokens benchmarks, where on 72 processors MPL is 10x faster and uses 18x less space. This benchmark has a high rate of allocation: it extracts the tokens of a file of size 148MB, which contains approximately 16.5 million tokens (average length of each token: 9 characters). That is, it allocates 16.5 million strings in parallel, of total size approximately 148MB. We measured that Java spends approximately half of its time allocating strings. Specifically, we developed an alternate version of the benchmark which pre-allocates the output strings; when the pre-allocation cost is excluded from the measurements, Java's 72-processor running time improves by approximately 2x. However, this is still 4-5x slower than MPL. Going further, we used the same approach to exclude the cost of all dynamic allocations (e.g., including temporary intermediate arrays), and found that Java's 72-processor runtime improves significantly, down to 0.053s, which is only approximately 25% slower than MPL (0.043s on 72 processors, including the cost of allocation and memory management). Therefore, Java's limited scalability on this benchmark (self-speedup of 9x) appears to be due to the cost of dynamic allocation and memory management, especially for high rates of allocation. In contrast, MPL manages the high rate of allocation efficiently and with good scalability (self-speedup of 45x).

Go Benchmarks and Results

Utilizing native "goroutines" and channels, Go directly supports nested parallelism. In particular, the code shown in Figure 8.4 shows how to implement a binary fork-join primitive called `pardo` which is similar to the `par` primitive we provide in MPL. The Go expression `pardo(f,g)` runs the functions `f` and `g` in parallel, and allows for arbitrary nesting, i.e., the functions `f` and `g` may themselves call `pardo` if desired. This makes it possible to write efficient divide-and-conquer parallel programs in Go, which we do here.

Results. Table 8.4 presents the results of the Go comparison, with running times T_p and max residency R_p on both $p = 1$ and $p = 72$ processors. The columns labeled "G" are results for Go,

and the columns "M" are results for our MPL. The column $\frac{G}{M}$ is the performance of Go relative to MPL; larger ratios are in favor of MPL.

We first observe that, in terms of both time and space efficiency, MPL and Go are generally competitive. On average, Go is 10% slower on one processor and approximately 2x slower on 72 processors. All space ratios fall within the range 0.59-2.3x, with averages of 1.5x on uniprocessor runs and 1.35x on 72-processor runs.

Comparing running times more closely, we see that Go is in three cases faster than MPL on a single core (msort-int64, msort-strings, and primes), but does not scale as well: on 72 processors, MPL is 1.6-3.5x faster for sorting, and approximately 15% faster for primes. The largest gap between the two systems is the tokens benchmark, where MPL is 11x faster on 72 processors. On this benchmark, Go maintains a relatively low total space usage but only achieves a self-speedup T_1/T_{72} of approximately 9x. This is low in comparison to Go's self-speedup on the other benchmarks (e.g. 43x on linefit, 58x on mcss, and 18x on primes).

As discussed in the Java comparison, the tokens benchmark has a high rate of allocation, suggesting that the low speedup in this case is due to memory management and GC. For Go, we found that disabling GC entirely (by setting `GOGC=-1`) improves Go's time performance on the tokens benchmark by approximately 2-3x. However, this is still approximately 4-6x slower than MPL. We developed an alternative version of the Go tokens benchmark which separately pre-allocates as much memory as possible (including the output strings, and all temporary intermediate arrays). Excluding the cost of these pre-allocations from the measurements, we find that the Go running time improves significantly, down to only 1.8x slower than MPL on 72 processors, which is in the same ballpark as the other benchmarks. Therefore, Go's limited scalability on this benchmark (only approximately 9x self-speedup) appears to be due to the cost of dynamic allocation and memory management. In contrast, MPL is able to manage the high rate of allocation in the tokens benchmark efficiently and with good scalability (self-speedup of 45x).

8.6 Comparison with C++

Low-level, memory-unsafe languages such as C++ are widely used for high-performance shared-memory multicore parallelism, largely due to the level of control over memory layout and representation they offer, which is key to achieving high performance in many cases. It is therefore interesting to consider how MPL fares against the fastest low-level techniques.

We consider here 15 problem-based benchmarks, including sophisticated parallel algorithms from domains such as graph processing (e.g. edge-balanced parallel bfs), computational geometry (e.g. delaunay triangulation, nearest neighbors, and convex hull), text processing (e.g. parallel string search and tokenization), various numerical algorithms (e.g., bignums, linear regression and linear recurrence, numerical integration, matrix and vector multiplications, etc.), as well as the classic sorting problem. The C++ benchmarks in this comparison are taken from state-of-the-art suites and libraries, including PBBS [14, 32, 133], ParlayLib [34], and Ligra [131]. We ported all of these benchmarks to MPL; interestingly, all of these benchmarks were naturally disentangled.

This comparison in unfair against MPL, because MPL offers guarantees which C++ does not,

	T_1			T_{72}			R_1			R_{72}		
	C	M	$\frac{M}{C}$	C	M	$\frac{M}{C}$	C	M	$\frac{M}{C}$	C	M	$\frac{M}{C}$
bfs	8.47	18.7	**2.21**	.188	.420	**2.23**	5.4	7.4	**1.37**	6.1	5.7	**0.93**
bignum-add	1.89	4.05	**2.14**	.036	.069	**1.92**	1.5	3.0	**2.00**	2.3	3.1	**1.35**
delaunay	3.48	11.9	**3.42**	.160	.379	**2.37**	.40	1.1	**2.75**	1.2	2.1	**1.75**
grep	1.00	2.12	**2.12**	.021	.040	**1.90**	.49	.61	**1.24**	1.2	.85	**0.71**
integrate	1.85	3.11	**1.68**	.030	.052	**1.73**	.015	.013	**0.87**	.76	.050	**0.07**
linearrec	.921	5.02	**5.45**	.080	.226	**2.83**	4.8	6.7	**1.40**	5.6	21	**3.75**
linefit	2.16	2.42	**1.12**	.149	.146	**0.98**	8.0	8.2	**1.02**	8.8	8.2	**0.93**
mcss	.972	4.64	**4.77**	.038	.078	**2.05**	4.0	4.1	**1.02**	4.8	4.1	**0.85**
msort-int64	2.17	3.98	**1.83**	.058	.077	**1.33**	.65	.66	**1.02**	1.4	.91	**0.65**
nearest-nbrs	1.01	1.67	**1.65**	.022	.038	**1.73**	.22	.72	**3.27**	.96	2.2	**2.29**
primes	1.58	7.43	**4.70**	.072	.122	**1.69**	.17	.26	**1.53**	.92	.37	**0.40**
quickhull	.941	3.51	**3.73**	.036	.113	**3.14**	2.6	13	**5.00**	3.5	20	**5.71**
sparse-mxv	2.13	2.45	**1.15**	.046	.049	**1.07**	4.2	4.5	**1.07**	4.9	4.4	**0.90**
tokens	1.09	1.93	**1.77**	.025	.043	**1.72**	.70	1.1	**1.57**	1.4	1.1	**0.79**
wc	3.12	7.69	**2.46**	.052	.130	**2.50**	1.8	3.6	**2.00**	2.5	3.6	**1.44**
geomean			**2.38**			**1.86**			**1.59**			**1.01**

Table 8.5: MPL vs C++: Times (seconds) and max residencies (GB) of MPL (column **M**) and C++ (column **C**). The ratios $\frac{M}{C}$ are the performance of MPL relative to C++. **Note: smaller ratios are better for MPL**.

namely, memory safety. However, we do not focus here on differences at the language-level between C++ and MPL. Instead, our goal is to establish a ballpark measure for the "absolute" efficiency of MPL, taking into account the performance overhead of automatic memory management and GC. With this in mind, we do not expect MPL to outperform C++. Rather, in this section, we show that MPL is generally competive with C++. That is, on average, MPL is less than 2x slower than C++ on 72 processors. Notably, we show that MPL is highly competitive in terms of space usage: on average, on 72 processors, MPL has the same memory footprint as C++ across these benchmarks.

Results. Table 8.5 presents the results of our C++ comparison, with running times T_p and max residency R_p on both $p = 1$ and $p = 72$ processors. The columns labeled "C" are results for C++, and the columns "M" are results for our MPL. The columns $\frac{M}{C}$ show the performance of MPL relative to C++, where smaller ratios are in favor of MPL. **Note**: in contrast to our prior comparisons (such as in Sections 8.4 and 8.5), here we report MPL's overhead rather than its performance advantage. This is because, in general, we do not expect MPL to outperform C++. Instead, we are interested in establishing the performance gap between the performance that MPL offers and the fastest low-level techniques.

We observe that on 72 processors, MPL is on average less than 2x slower than C++ while using a similar amount of memory. These measurements include the cost (both space and time) of automatic memory management and GC. Runtime overheads of MPL range from 0.98-3.14x,

with 9 out of 15 benchmarks incurring less than 2x overhead, and 2 benchmarks (linefit and sparse-mxv) within ±10% of C++. On the linefit benchmark, as discussed in Section 8.3, both MPL and C++ have nearly optimal performance for our test machine.

On uniprocessor runs, MPL has higher runtime overhead (2.4x on average) than it does at scale (1.9x). That is, MPL has better self-scalability than C++. In other words, this demonstrates that the overheads of memory safety and automatic memory management are well-parallelized and scale well. These overheads include the cost of managing the dynamic tree of heaps, of parallel garbage collection, but also for other safety features in the MPL language, including (for example) out-of-bounds array checks, all of which perform well in parallel.

In terms of space usage, there are three benchmarks where MPL uses significantly more space than C++: linearrec, nearest-nbrs, and quickhull. All three of these benchmarks operate on similar data: an array containing pairs of double-precision floating-point numbers (representing 2-dimensional points in Euclidean space). The additional space usage of MPL in these cases appears to be due to memory representation, determined by compilation. In particular, the MPL compiler has multiple data flattening compilation passes which attempt to "unbox" data (i.e., eliminate heap allocations by flattening this data into its container). Thus, on a MPL array of type `(real × real) array`, the compiler controls whether or not the tuples in the array are flattened into the array, or indirected by a pointer. In these three benchmarks where MPL has higher space usage (linearrec, nearest-nbrs, and quickhull), the compiler does not succeed in flattening these tuples into the array, resulting in higher space usage. The runtime performance is also affected: both linearrec and quickhull incur approximately 3x runtime overhead in comparison to C++, due to the cost of additional indirection the corresponding loss of data locality. (For example, the cost of random access to access a point in the MPL quickhull benchmark likely incurs two cache misses, whereas in C++ it would only incur at most one.) We believe that, by controlling data flattening more carefully, the performance gap on these benchmarks can be closed. In future work, we plan to revisit this issue.

Nevertheless, MPL has (on average) the same memory footprint on 72 processors as C++. Additionally, we observe that the space overhead of MPL is on average smaller on 72 processors than it is on one processor, i.e., MPL's memory usage per processor is lower than C++ in these benchmarks. Notably, on 9 out of 15 benchmarks, MPL uses less memory than C++ at scale. These results demonstrate that MPL generally is highly space-efficient and can compete with low-level manual memory management.

8.7 Evaluation of Entanglement Detection

We evaluate our entanglement detection techniques in two parts. First, we compare MPL against itself with entanglement detection disabled, to measure the overhead of entanglement detection. We observe close to zero overhead across the board (in terms of both time and space), with approximately 1% overhead on average. Next, we show that the efficiency of entanglement detection is largely due to our entanglement candidates algorithm, which eliminates unnecessary entanglement checks. We perform this comparison by measuring the time performance of detection with and without the candidates tracking algorithm. Our results show that the candidates algorithm brings the number of entanglement checks down to 0 in multiple cases,

	T_1		T_{72}		R_1		R_{72}	
	MPL^{dd}	**MPL**	MPL^{dd}	**MPL**	MPL^{dd}	**MPL**	MPL^{dd}	**MPL**
bfs-tree	19.9	20.5 (+3%)	.433	.442 (+2%)	7.8	7.8 (+0%)	68	68 (+0%)
centrality	16.2	16.0 (-1%)	.432	.426 (-1%)	6.9	6.9 (+0%)	5.9	5.9 (+0%)
dedup-strings	5.35	5.52 (+3%)	.114	.122 (+7%)	2.6	2.6 (+0%)	6.4	6.7 (+5%)
delaunay	8.72	9.18 (+5%)	.323	.335 (+4%)	.59	.58 (-2%)	1.3	1.3 (+0%)
dense-matmul	2.78	2.79 (+0%)	.048	.048 (+0%)	.064	.063 (-2%)	.31	.31 (+0%)
game-of-life	1.81	1.73 (-4%)	.067	.067 (+0%)	.061	.061 (+0%)	.35	.35 (+0%)
grep	2.09	2.05 (-2%)	.038	.038 (+0%)	1.1	1.1 (+0%)	1.4	1.4 (+0%)
low-d-decomp	8.16	8.37 (+3%)	.406	.392 (-3%)	6.9	6.9 (+0%)	14	13 (-7%)
msort-strings	2.70	2.77 (+3%)	.060	.058 (-3%)	.62	.62 (+0%)	1.7	1.7 (+0%)
nearest-nbrs	1.54	1.54 (+0%)	.046	.047 (+2%)	.26	.26 (+0%)	1.6	1.6 (+0%)
nqueens	1.60	1.61 (+1%)	.028	.029 (+4%)	.045	.045 (+0%)	.32	.32 (+0%)
palindrome	1.62	1.69 (+4%)	.031	.032 (+3%)	.061	.063 (+3%)	.36	.36 (+0%)
primes	7.42	7.56 (+2%)	.123	.120 (-2%)	.27	.27 (+0%)	.56	.56 (+0%)
quickhull	3.22	3.40 (+6%)	.103	.110 (+7%)	2.4	2.4 (+0%)	5.9	6.1 (+3%)
range-query	14.4	15.4 (+7%)	.250	.248 (-1%)	4.5	4.5 (+0%)	4.5	4.6 (+2%)
raytracer	3.50	3.30 (-6%)	.058	.058 (+0%)	.11	.11 (+0%)	.54	.55 (+2%)
reverb	1.34	1.31 (-2%)	.042	.041 (-2%)	1.5	1.5 (+0%)	2.1	2.1 (+0%)
seam-carve	16.2	16.2 (+0%)	.808	.827 (+2%)	.091	.091 (+0%)	.71	.71 (+0%)
skyline	7.31	7.57 (+4%)	.250	.251 (+0%)	.88	.88 (+0%)	25	25 (+0%)
suffix-array	5.54	5.74 (+4%)	.113	.115 (+2%)	1.2	1.2 (+0%)	1.7	1.6 (-6%)
tinykaboom	2.47	2.48 (+0%)	.042	.042 (+0%)	.0093	.011 (+18%)	.31	.31 (+0%)
tokens	1.96	1.86 (-5%)	.043	.042 (-2%)	1.1	1.1 (+0%)	1.4	1.4 (+0%)
triangle-count	4.74	4.57 (-4%)	.192	.191 (-1%)	2.5	2.5 (+0%)	14	14 (+0%)
geomean		1.01x		1.01x		1.01x		1.0x

Table 8.6: Times (seconds), max residencies (GB), and percent differences of MPL relative to MPL^{dd}, which has detection disabled. The percentages in parentheses are the overhead of entanglement detection. The "geomean" is the geometric mean of the ratios MPL/MPL^{dd}.

and provides running time improvements of up to a factor of 2x.

The comparisons in this section are taken from an earlier version of the Parallel ML Benchmark Suite. The code is available on GitHub.[8]

8.7.1 With and Without Entanglement Detection

We compare MPL against itself with entanglement detection disabled to determine the overheads of entanglement detection itself. Here, we write MPL^{dd} for the version of MPL that has detection disabled. There are two main differences with entanglement detection disabled: (i) the read barrier is disabled, which could have a significant impact on performance, and (ii) entanglement candidates are no longer tracked.

[8] https://github.com/MPLLang/entanglement-detect

Entanglement detection overheads are small. Table 8.6 shows results on 1 and 72 processors. Columns T_1 and T_{72} show the run-time for MPL and $\mathsf{MPL}^{\mathsf{dd}}$, with the additional cost of MPL relative to $\mathsf{MPL}^{\mathsf{dd}}$ shown as percentage for each quantity. Observe that the MPL times are usually within ±5% of $\mathsf{MPL}^{\mathsf{dd}}$. 18 out of 23 benchmarks considered here have less than 2% time overhead, and we observe a max time overhead of 7% in only two cases on 72 processors. On average across all benchmarks (geometric mean of the ratios MPL/$\mathsf{MPL}^{\mathsf{dd}}$), the time overhead is approximately 1% on both 1 and 72 processors.

Entanglement detection scales well. On both 1 and 72 processors, we observe similar time overheads across all benchmarks. Entanglement detection therefore has no noticeable impact on scalability.

Space overheads are small. Columns R_1 and R_{72} show the space usage for MPL and $\mathsf{MPL}^{\mathsf{dd}}$, with the additional cost of MPL relative to $\mathsf{MPL}^{\mathsf{dd}}$ shown as percentage for each quantity. For both 1-core and 72-core runs, there is almost no noticable space overhead. Only one benchmark (`tinykaboom`) registers above 10% space overhead, but only for sequential runs, where the overall footprint is small: while $\mathsf{MPL}^{\mathsf{dd}}$ uses approximately 9 MB, MPL uses 11 MB. Entanglement detection therefore appears to have a small constant space overhead. At scale, with memory on the order of gigabytes, this overhead is negligible.

8.7.2 Improvement Due To Entanglement Candidates

We now demonstrate that the low overhead of entanglement detection is due to the elimination of unnecessary graph queries. Our technique for eliminating unnecessary graph queries is our candidate tracking algorithm (Section 5.6). To measure the performance improvement due to tracking candidates, we run a version of MPL that still performs detection, but without candidate tracking. We focus here only on the benchmarks which have a large number of graph queries (i.e., a large number of dereferences of mutable pointers to heap-allocated objects). The performance of the omitted benchmarks is not significantly affected by the candidate tracking algorithm (specifically, on average, the performance change is ±1%). That is, by tracking candidates, we improve performance only in the cases where it is needed, and do not harm performance otherwise.

The results of this comparison are presented in Table 8.7. In the table, the "Improvement Ratio" columns are calculated as $T_{\text{off}}/T_{\text{on}}$, where T_{off} is the time with candidate tracking disabled, and T_{on} is the time with it enabled. Improvement ratios larger than 1 indicate a performance improvement due to the candidate tracking algorithm. We provide improvement ratios for both uniprocessor runs ($P = 1$) as well as parallel runs ($P = 72$). We also collect additional data, including the number of graph queries performed (both with and without candidate tracking), and the number of times a candidate was marked (which is an upper bound on the number of candidate objects).

We first observe that by tracking candidates, we are able to improve performance in almost all cases, often by a significant margin. The benchmark with the biggest change is `msort-strings`, with approximately 2x improvement. Other improvements range from 2% up to a factor of 1.6x.

	Improvement Ratio		# Graph Queries		
	$P = 1$	$P = 72$	Naïve	w/ Candidates	# Candidate Marks
bfs-tree	0.99x	0.99x	16M	16M	64
dedup-strings	**1.17x**	**1.12x**	122M	0	0
delaunay	**1.44x**	**1.25x**	237M	0	0
low-d-decomp	0.99x	**1.02x**	1M	0	62
msort-strings	**1.87x**	**1.95x**	247M	0	0
nearest-nbrs	**1.60x**	**1.23x**	73M	0	0
quickhull	**1.62x**	**1.20x**	184M	0	0
skyline	**1.16x**	**1.07x**	174M	1M	1K
triangle-count	**1.02x**	0.95x	3M	0	2

Table 8.7: Performance improvement ratio due to tracking candidates, including number of graph queries performed both with and without candidate tracking.

Inspecting the number of graph queries performed, we see that our approach successfully elides a significant number of unnecessary graph queries, and in all but 2 benchmarks, the number of graph queries goes down to 0. For example, under the naïve strategy (tracking disabled), the `msort-strings` benchmark performs approximately 250M graph queries, but after tracking candidates, none are performed.

This elimination of unnecessary graph queries is key to the near-zero overhead of entanglement detection. Note that although the graph queries are elided, there is still overhead for the "fast path" of the read barrier on mutable objects, which inspects the header of the dereferenced object to determine whether or not it is a candidate. However, this overhead is not significant enough to be a concern: as evidenced by the results in comparison with vanilla MPL (in Table 8.6), the cost of the fast path is nearly zero.

There are three interesting benchmarks here which do not see significant improvement: `bfs-tree`, `low-d-decomp`, and `triangle-count`. In the case of `low-d-decomp` and `triangle-count`, although tracking candidates successfully eliminates all graph queries, the number of queries needed in the first place is low relative to the work performed by the benchmark, so the opportunity for improvement is small (e.g. the input for `low-d-decomp` has approximately 200M edges but performs only 1M queries when the candidate tracker is disabled). In contrast, the `bfs-tree` benchmark achieves no performance improvement because no graph queries are eliminated. This is the only benchmark does not see a significant number of queries eliminated due to the candidates algorithm. The reason in this case is that the algorithm uses a shared array to associate state with each element of the input, and negotiates access to this state via a lock-free algorithm. As soon as one piece of this state is updated, the shared array becomes a candidate, and all further accesses to it require full checks.

Finally, it's worth emphasizing that the number of candidate marks (i.e., the number of times any object is marked as a candidate) is small across the board. This is an upper bound on the number of candidate objects, and therefore bounds the space overhead of tracking candidates (i.e., it bounds the size of the candidate set during execution). These results confirm our hypothesis that only a small number of objects typically pose a risk for entanglement.

8.7.3 Entangled Tests

As mentioned, all benchmarks used in our performance evaluation above are naturally disentangled. To validate our detector, we also developed a set of entanglement test cases, including both synthetic tests as well as variations of some of our benchmarks. Our detector successfully found entanglement in all instances.

One interesting case of entanglement we developed was a variation of the hash-deduplication algorithm, similar to that described in Section 2.3.2. Note that the implementation we show in Section 2.3.2 is disentangled. We were able to introduce entanglement by changing this code, to mix allocations with insertions. In particular, in the entangled version, we copy each element immediately before it is inserted, and insert the copy instead of the original. This causes entanglement because, when inspecting a slot in the hash table, a task may observe a pointer to an element that was allocated and inserted by another (concurrent) task. In contrast, in the disentangled version, all inserted elements are allocated before the insertions begin.

Chapter 9

Related Work

9.1 Parallel Memory Management

Nearly all high level languages today support automatic memory management and numerous techniques for incorporating parallelism, concurrency, and real-time features into memory managers have been developed. Jones et al. [86] provides an excellent survey. Here, we contrast the disentanglement-based memory management techniques proposed in Chapter 4 with prior systems that use processor-local or thread-local heaps combined with a shared global heap that must be collected cooperatively [15, 21, 56–58, 102].

The Doligez-Leroy-Gonthier (DLG) parallel collector [56, 57] employs this design, with the invariant that there are no pointers from the shared global heap into any processor-local heap and no cross pointers between processor local-heaps. To maintain this invariant, all mutable objects are allocated in the shared global heap and (transitively reachable) data is promoted (copied) from a processor-local heap to the shared global heap when updating a mutable object. This approach penalizes allocations and updates for mutable data and thus increases the cost of common scheduling and communication actions, such as migrating a user-level thread or returning the result of a child task.

The Manticore garbage collector [21] is a variant of the DLG design, where the Appel semi-generational collector [16] is used for collection of the processor-local heaps. As with the DLG design, Manticore collector can incur large promotion overheads. Recent work [96] has considered extending the Manticore language with mutable state via software transactional memory, but observed that promotions lead to efficiency problems.

The two-level hierarchical model that does not allow pointers from the global to the local heaps incur large overheads when an object allocated locally must be shared, which can happen often in nested-parallel programs due to scheduling actions, which migrate tasks between processes or workers. Adaptations of the two-level model to concurrent and parallel systems therefore devised techniques to relax this invariant. For example, the Glasgow Haskell Compiler (GHC) uses a garbage collector [102] that allows pointers from global to local heaps and relies on a read barrier to promote (copy) data to the global heap when accessed. Although Haskell is a pure language, there are significant side effects due to lazy evaluation. GHC therefore combines elements of the DLG and Domani et al. [58] collectors for improved handling of

side effects. The Multicore OCaml project also utilizes a variant of the two-level heap model, offering multiple strategies for collection based on different relaxations of the invariants maintained between local and global heaps [139]. As a result of this work, OCaml version 5.0 features a two-level collector with a concurrent, non-moving major GC and a stop-the-world parallel minor GC.

In contrast to these prior approaches, in our work, we associate heaps with tasks rather than system-level threads or processors. The result is a dynamic hierarchy that mirrors the structure of the computation; here, we take advantage of the structure of fork-join parallel computations, which are naturally hierarchical. The hierarchy can be arbitrarily deep in principle and grows and shrinks as the computation proceeds. To support sharing, we allow pointers between heaps that have ancestor-descendant relationships. For example, a heap can point to an object allocated in its parent, and a parent can point to an object allocated in its children. The only kind of pointer that is not allowed is a cross-pointer between concurrent heaps. This approach enables taking advantage of important properties of parallel programs, e.g., we can return the result of a child task and migrate threads without copying (promoting) data, and concurrent threads can share the data allocated by their ancestors, and disentangled effects do not require an immediate promotion of data.

In the sequential setting, region-based memory management [80, 126, 130, 150] shares some similarities with hierarchical heaps. In a region-based system, a program dynamically creates and destroys *regions*, into which individual objects may be allocated; thus, regions (only) support bulk deallocation (but also supporting garbage collection has been considered [59]). Statically-scoped regions [75, 150] are organized as a stack, while dynamically-scoped regions [75, 80] impose no particular relationship between regions. In general, pointers from one region to any other region are supported; a type-and-effect system [150] or linear types [66, 152] can be used to guarantee that pointers into deallocated regions will never be followed. The flexibility of allocating new objects into any available region allows for arbitrary memory graphs and avoids the need to promote objects from one region to another, but with the overhead of explicitly managing the set of available regions, rather than implicitly having a single allocation frontier.

Nearly all of the work reviewed above relies on the idea of organizing memory as a hierarchy of heaps, some shallow like most other work, and some possibly deep, like our work. The general idea of hierarchical heaps goes back to 1990s. Early approaches in procedural languages such as Split-C [91], Co-Array Fortran [114], and Titanium [159], differentiate between memory that is local and remote to a thread. Alpern et al. developed abstract models of uniprocessor and multiprocessor machines as hierarchies of memories [13]. More recently, the technique was employed in the Sequioa language, which allows the programmer to designate tasks to run on a fixed memory hierarchy by specifying the mapping between tasks and levels [62]. The work on Legion [23] builds on Sequioa by allowing the programmer to control data sharing and locality using types and by allowing more dynamic hierarchies. One difference between these approaches and our approach is that in our approach, the memory hierarchy mirrors the evolution of the computation automatically, growing and shrinking dynamically as the computation proceeds.

This dynamic and automatic management of hierarchical memory was first proposed in 2015 [4] and realized concretely for functional programs [120]. A recent paper [76] extended the

technique to support for isolated effects at the sequential portions of the parallel computation. Handling of more general effects remained unknown until the results presented in this thesis.

9.2 Race Detection

There has been significant work on detecting races in parallel and concurrent programs. Our entanglement detection technique is similar to race detection in the sense that it concerns a property of memory accesses. Prior research on race detection can broadly be divided into two lines of work: (i) those that target task-parallel programs, and (ii) those that target general concurrency. The first line of work assumes, as we do here, task parallelism, where a program may create many (e.g., millions of) fine-grained threads, which synchronize in a structured fashion. The second line of work assumes a general concurrency setting, where programs contain a small number of coarse-grained threads that may synchronize by using locks, synchronization variables, etc. Techniques from this second line of work do not scale to task-parallel programs because of their coarse-grained threads assumption (e.g,. [123]), and are less directly relevant to this thesis.

Race Detection for Task Parallel Programs

Many algorithms for race detection in task-parallel programs, such as fork-join programs, have been proposed [24, 50, 63, 64, 104, 122, 123, 151, 157]. These algorithms all revolve around an ordering data structure, sometimes called a *series-parallel order* or *SP-order* data structure, which keeps track of whether two instructions are sequentially dependent or can be executed in parallel. Experiments with state-of-the-art race detection show over an order of magnitude overhead in sequential runs, and parallel runs with race detection typically run slower than the sequential baseline even with over a dozen cores [151]. All of the above work considers nested parallelism with fork-join and async-finish constructs, which result in similar dependency structures. More recent work considers race detection for futures and establishes worst-case bounds, though the overheads are no longer constant [157].

Our entanglement detection techniques share the same basic structure as these race detectors, in the sense that for entanglement detection we similarly record the structure of the computation as a graph and query the graph with the help of an order maintenance data structure. However, entanglement detection is more efficient than race detection. This is due to a few differences between the two techniques.

One of the most significant overheads of race detection is incurred by the maintenance of "access histories", which track the history of reads and writes on each individual memory location [104, 123, 151]. For example, an array of size N requires N access histories for race detection (one for each index). In contrast, entanglement detection requires only a single annotation (one vertex identifier) for the whole array. Furthermore, as we show in Section 5.5, the space overhead of the annotations for entanglement detection can be further reduced by grouping together the allocations of individual threads, resulting in essentially negligible space overhead. Maintenance of access histories for race detection also incurs a time penalty, particularly on reads, where the thread identifier of the reader is logged at each access.

Finally, we note that entanglement detection benefits from compiler optimizations which perform data inlining (also called "flattening" or "unboxing"). For example, when operating on an array of unboxed integers, entanglement detection requires neither a read barrier nor a write barrier, and therefore incurs no overhead. In contrast, race detection still needs to monitor these operations.

Race Detection for General Concurrency

In this thesis we consider task-parallel programs that typically generate a very large number of fine-grained tasks or threads, that synchronize in a structured fashion. There has been much work on race detection for more general concurrent programs, which use a small number of coarse-grained threads that can synchronize in an unstructured manner, using locks and other synchronization primitives. Early work proposes the lock-set algorithm [129], which can lead to false positives. Subsequent work proposed precise techniques by using vector-clocks to capture the happens-before relations between threads [65]. Followup work has proposed hybrid approaches that combine lock sets and vector clocks, trading off efficiency and precision [115, 160]. Most approaches use dynamic, on-the-fly race detection though there has also been some work on predicting data races [89, 140].

A general assumption in the general concurrency domain is that threads are coarse grained, and the total number of threads is small. These approaches, therefore, do not work well for task parallel programs, where the number fine-grained threads can be very large [123]. Another limitation of dynamic race-detection under general concurrency is that these techniques are unable to account for logical (potentially unrealized) parallelism and remain sensitive to scheduling decisions. In contrast, race-detection for task parallelism can account for logical parallelism, as can the entanglement detection techniques presented here.

9.3 Parallel Programming Languages

In this thesis, we consider the Parallel ML functional programming language which extends Standard ML with parallelism. As with the Standard ML language, Parallel ML supports references and destructive updates, and allows writing both purely functional and impure (imperative) programs.

Parallel ML builds on a rich history of research or parallel programming languages, including both procedural and functional ones. Programming languages such as Cilk/Cilk++ [37, 71, 84], Cilk-F/L [135, 136], I-Cilk [107], and Intel TBB [85] extend C/C++ with task parallelism but they all require manual memory management, which is especially challenging for parallel programs. The Rust language offers a type-safe option for systems-level programming [127] and can ensure memory safety under certain assumptions.

Extensions of the Java language to support parallelism include the Fork/Join Framework [97], and Habanero Java [83], both of which support automatic memory management. The X10 [46, 98] language is designed with concurrency and parallelism from the beginning and supports both imperative and object-oriented features. Even though these languages simplify writing parallel programs by managing memory automatically, avoiding concurrency bugs can still be

challenging, because of the lax control over side effects that these languages offer. Motivated partly by this concern, research on Deterministic Parallel Java [38, 39] develops type systems to guarantee determinism.

Modern type-safe functional programming languages offer substantial control over side effects [72, 92, 93, 95, 100, 117, 118, 124, 145, 149], which help programmers avoid unintentional race conditions. Recent projects include Manticore [67, 68] MultiMLton [138, 162], SML# [116], multicore OCaml [139], and both current and prior work on disentanglement and MPL [4, 18, 76, 120, 153, 154]. There has also been significant progress on parallel and concurrent Haskell [45, 87], including work on memory management techniques [102].

We note that even though disentanglement and entanglement have so far been applied to functional programming languages, they are fundamentally a language-agnostic property, and thus could be applied to procedural languages.

Scheduling

Almost all modern parallel programming languages utilize a thread scheduler in the run-time system which migrates threads between processors to utilize processors as best as possible within some context. Many scheduling algorithms have been designed to improve a variety of metrics, including time [3, 7, 8, 19, 36], responsiveness and interactivity [107–111, 137], space [29, 35, 112], and locality [1, 27, 30, 31].

Granularity Control

An important parameter in many parallel codes is the "granularity" or the "grain" at which computations revert from parallel to sequential. In the current state of the art, researchers and practitioners typically control granularity manually by optimizing their codes to switch from parallel to sequential codes at a certain granularity, e.g., small input sizes. This is the technique used in our benchmarks as well as those that we compare against. This manual approach to granularity control has several important drawbacks and there have been recent works that propose solutions that can automate or semi-automate granularity control [5–7, 121].

One interesting observation is that the memory management techniques presented in this thesis naturally adapt to changes to grain size. For example, if the grain size is decreased, then the depth of the heap hierarchy increases, and in response, the scope of LGC increases. In other words, the amount of memory collected by LGC is mostly invariant under different grain sizes.

Chapter 10

Concluding Remarks

10.1 Discussion

Statically Checking for Disentanglement

In this thesis, we use a dynamic entanglement detection technique to enforce disentanglement during execution. It is important to emphasize that, as a dynamic approach, entanglement detection is execution-dependent. Due to non-determinism, our detector cannot prevent the *possibility* of entanglement. An important problem for future work therefore is to develop static tests that can prevent the possibility of entanglement entirely.

To ensure disentanglement statically, one natural approach could be to use a type system. Designing a suitable type system for disentanglement seems possible, but challenging: entanglement is an undecidable property, and it emerges due to subtle interaction between non-determinism and aliasing. A conservative type system could be used to ensure disentanglement (for example by forbidding all in-place updates), but this would result in significant loss in efficiency. Existing type and effect systems for determinism (e.g., [38, 39]) could possibly be adapted to ensure disentanglement, but these would also forbid the non-deterministic programs that we wish to allow.

As we discuss in Sections 2.3 and 3.5, many efficient parallel algorithms are inherently non-deterministic, and one of the advantages of disentanglement is that it allows for non-determinism. Some examples considered in this thesis include breadth-first-search, betweenness centrality, delaunay triangulation, deduplication, low-diameter decomposition, maximal independent set, and others. Ideally, a static test for disentanglement should allow for these examples, as well as other non-deterministic parallel programming idioms which are commonly used to improve efficiency in practice.

We note that various systems for enforcing "safe non-determinism" have been proposed (e.g., [40, 94]). It would be interesting to explore whether these systems can be adapted to ensure disentanglement, and (if so) what disentangled programs would be permitted.

Entanglement Management

Our experience with developing a parallel benchmark suite shows that many parallel programs are naturally disentangled. Perhaps surprisingly, we have found that this holds even for programs that were originally written in low level languages such as C/C++. But, as we consider a broader set of parallel programs and include concurrent programs, which use communication between concurrently executing tasks (or threads), entanglement will arise naturally. In future work, we plan to use entanglement detection to bring the benefits of disentanglement-based memory management to entangled programs by detecting entanglement at run-time and managing it automatically. The result should be a fully general parallel functional programming language which is just as efficient as the implementation presented in this thesis, but which also supports effects and communication with no restrictions.

Disentanglement Beyond Fork-Join

Although we focus on fork-join parallelism, the definition of disentanglement could be generalized to other forms of parallelism. At a high level, the idea would be represent computations as computation graphs, and similarly define disentanglement as the property that every instruction which uses a location is sequentially dependent upon the instruction which allocated that location. This generalized definition makes no assumption about the structure of the computation graph, and therefore could in principle be applied to arbitrary computations, including other structured forms of parallelism (e.g., futures [77, 78], async-finish [98]), as well as more general unstructured parallelism. We conjecture that determinacy-race-free programs are disentangled for arbitrary computations; we leave the proof to future work.

In more general settings, an interesting research avenue is to determine how to structure memory so that disentanglement can be exploited for improved efficiency and scalability. For example, programs with futures might require that memory be restructured dynamically when a future is touched (to ensure that after a touch, the continuation can safely access memory allocated by the touched future).

Distributed Computing

In this thesis, we focus on shared-memory, multicore parallelism. Moving beyond multicore parallelism, we note that disentanglement appears to be directly applicable in the field of distributed computing, where separation of memory between remote nodes is essential for efficiency. Any distributed program which communicates only via message passing is essentially disentangled by construction, allowing for local memory management without interfering with remote nodes. More generally, if a distributed program utilizes a distributed shared memory scheme, disentanglement could provide an opportunity to implement distributed shared memory more efficiently, by restricting remote access to descendants (in the task tree).

Heterogeneous Computing

Hardware such as GPUs provide an opportunity to accelerate data-parallel functional operations (including map, scan, reduce, filter, etc.). GPU implementations of these operations can

be supported in a functional setting in various ways. For example, the Futhark language implementation compiles a dialect of parallel ML and produces efficient GPU code [81, 82]. Alternatively, one could implement a high-level functional library, utilizing low-level GPU code under-the-hood for improved efficiency, and link the GPU code via a foreign-function interface (e.g., as demonstrated by Yadav and Houghton [158]). Our MPL compiler features a rich foreign-function interface and is capable of offering GPU support in this manner.[1]

For a library that utilizes the GPU via foreign functions, there are a few options for memory management. One option would be to manually copy data to and from the GPU, and ensure that any data allocated on the GPU is properly reclaimed. For example, a library could implement data-parallel operations on the GPU by (1) copying data to the GPU, (2) performing a parallel operation on that data, (3) copying the data back, and (4) freeing the GPU-allocated memory. This approach, however, can incur significant cost due to explicit data migration. To avoid this cost, another option would be to utilize "unified" memory, which allows for the same data to be accessed both by the CPU and the GPU, and automatically migrates data between the two, on demand [128]. The garbage collector could then reclaim and recycle unified memory, similar to standard memory.

In the context of disentangled memory management (Chapter 4), a GPU kernel is essentially the same as an active leaf task, and could be scheduled and managed in a similar manner. This would allow us to schedule GPU kernels cooperatively with other work in the system, including both parallel CPU tasks and garbage collections. In this way, it seems possible to support heterogenous computing within the framework of disentangled memory management, and it would be interesting to explore this avenue in future work.

Closing the Performance Gap

As we show in Chapter 8, MPL is generally competitive with low-level, memory-unsafe programs written in C++. On average, MPL has the same memory footprint at scale, and is within a factor of 2x in terms of running times. With further research and engineering, we believe it is possible to close the performance gap entirely.

For closing the gap, perhaps the most important angle to investigate is data flattening. MPL performs a number of data flattening passes during compilation, which it inherits from the MLton compiler. These passes inline data into their containers; for example, an array of type `(int × int) array` has two possible representations: (i) as an array of pointers to heap-allocated tuples, or (ii) as a flattened array, with no pointers, where the tuples are inlined into the array itself. The flattened representation helps eliminate unnecessary allocations and avoids the cost of indirection, and is usually more efficient in practice. Therefore, in general, MPL/MLton perform flattening aggressively at compile time. However, flattening can increase space usage by reducing sharing. To combat a significant increase in space usage, the compiler has a number of heuristics which limit the impact of flattening.

Data flattening is essential for efficiency in some cases, but MPL/MLton do not always "make the right choice" at compile time. One interesting example is the quickhull benchmark, on which MPL has approximately 3x time overhead in comparison to C++. On this benchmark,

[1] Specifically, MPL inherits the MLton FFI (`http://www.mlton.org/ForeignFunctionInterface`) which is the same interface used by Yadav and Houghton [158].

we attempted to increase the aggressiveness of flattening at compile time by modifying the compiler slightly. The result was nearly a 2x performance boost (due primarily to improved data locality, i.e., fewer cache misses), bringing the overall time performance to within 60% of C++. This suggests that by controlling flattening, we may be able to reduce the performance gap between MPL and C++ significantly.

When it matters for performance, the programmer needs control over memory representation and layout. An interesting research question for future work is to offer such control in a high-level language such as Parallel ML while preserving the desirable qualities of the language (e.g., its simple yet effective type system).

10.2 Conclusion

Functional programming languages provide programmers with the ability to control effects and avoid unintentional race-conditions. In this way, functional languages are well-suited for parallelism, especially when it comes to reasoning about correctness. But parallel functional language imlementations have long underperformed in comparison to their procedural and imperative counterparts, largely due to the overhead of automatic memory management and garbage collection.

In this thesis, we establish a disentanglement property which has broad applicability to parallel programs, including functional programs as well as more generally any program that uses effects in a disciplined manner. We then design memory management techniques that take advantage of disentanglement, and implement a memory manager for Parallel ML which is naturally parallel, efficient, and scalable. Our implementation outperforms existing memory managed language implementations, and is competitive with low-level, memory-unsafe languages such as those based on C. This result takes an important step towards closing the performance gap between low-level (e.g., memory-unsafe) and high-level parallel languages, such as functional languages, which offer important correctness benefits that are crucial for parallel programming.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 499–518, 2011.

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013.

[4] Umut A. Acar, Guy Blelloch, Matthew Fluet, Stefan K. Muller, and Ram Raghunathan. Coupling memory and computation for locality management. In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[5] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)*, 26:e23, 2016.

[6] Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. Performance challenges in modular parallel programs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 381–382, 2018. ISBN 978-1-4503-4982-6.

[7] Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 769–782, 2018. ISBN 978-1-4503-5698-5.

[8] Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. Provably and practically efficient granularity control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 214–228, 2019. ISBN 978-1-4503-6225-2.

[9] Umut A. Acar, Jatin Arora, Matthew Fluet, Ram Raghunathan, Sam Westrick, and Rohan Yadav. Mpl: A high-performance compiler for parallel ml, 2020. `https://github.com/MPLLang/mpl`.

[10] Sarita V. Adve. Data races are evil with no exceptions: technical perspective. *Commun. ACM*, 53(11):84, 2010.

[11] Sarita V. Adve and Hans-Juergen Boehm. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, 2010. doi: 10.1145/1787234.1787255. URL https://doi.org/10.1145/1787234.1787255.

[12] T. R. Allen and D. A. Padua. Debugging Fortran on a shared memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, August 1987.

[13] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 600–608 vol.2, Oct 1990. doi: 10.1109/FSCS.1990.89581.

[14] Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. The problem-based benchmark suite (pbbs), V2. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 445–447. ACM, 2022. doi: 10.1145/3503221.3508422. URL https://doi.org/10.1145/3503221.3508422.

[15] Todd A. Anderson. Optimizations in a private nursery-based garbage collector. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, pages 21–30, 2010.

[16] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989. URL http://www.cs.princeton.edu/fac/~appel/papers/143.ps.

[17] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, January 1996. URL ftp://daffy.cs.yale.edu/pub/papers/shao/stack.ps.

[18] Jatin Arora, Sam Westrick, and Umut A. Acar. Provably space efficient parallel functional programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)"*, 2021.

[19] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[20] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, October 1989.

[21] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John H. Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC)*, pages 51–57, 2011.

[22] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, page 10–es, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781450378369. doi: 10.1145/1275808.1276390. URL https://doi.org/10.1145/1275808.1276390.

[23] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on*

High Performance Computing, Networking, Storage and Analysis, pages 1–11, Nov 2012. doi: 10.1109/SC.2012.71.

[24] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 133–144, 2004.

[25] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[26] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[27] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA*, 2004.

[28] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994.

[29] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 12–23, 1997.

[30] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *In the Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 501–510, 2008.

[31] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–366, 2011.

[32] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP '12*, pages 181–192, 2012.

[33] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 467–478, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342100. doi: 10.1145/2935764.2935766. URL `https://doi.org/10.1145/2935764.2935766`.

[34] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib - A toolkit for parallel algorithms on shared-memory multicore machines. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 507–509. ACM, 2020. doi: 10.1145/3350755.3400254. URL `https://doi.org/10.1145/3350755.3400254`.

[35] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded

computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[36] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[37] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[38] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 97–116, 2009.

[39] Robert L Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *First USENIX Conference on Hot Topics in Parallelism*, 2009.

[40] Robert L. Bocchino Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 535–548. ACM, 2011.

[41] Hans-Juergen Boehm. How to miscompile programs with "benign" data races. In *3rd USENIX Workshop on Hot Topics in Parallelism, HotPar'11, Berkeley, CA, USA, May 26-27, 2011*, 2011.

[42] Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 68–78. ACM, 2008. doi: 10.1145/1375581.1375591. URL https://doi.org/10.1145/1375581.1375591.

[43] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 261–270. ACM, 2015. doi: 10.1145/2767386.2767436. URL https://doi.org/10.1145/2767386.2767436.

[44] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SIAM SDM*, 2004.

[45] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007*, pages 10–18, 2007.

[46] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented

approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005.

[47] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *1990 ACM/IEEE Conference on Supercomputing (SC)*, page 666–675, 1990. ISBN 0897914120.

[48] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. Size and access inference for data-parallel programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI)*, page 130–144, 1991. ISBN 0897914287.

[49] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[50] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, 1998.

[51] Intel Corp. Knights landing (knl): 2nd generation intel xeon phi processor. In *Intel Xeon Processor E7 v4 Family Specification*, 2017. https://ark.intel.com/products/series/93797/Intel-Xeon-Processor-E7-v4-Family.

[52] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, page 315–326, 2007.

[53] Alain Darte. On the complexity of loop fusion. In *IEEE Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1999.

[54] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Trans. Parallel Comput.*, 8(1):4:1–4:70, 2021. doi: 10.1145/3434393. URL https://doi.org/10.1145/3434393.

[55] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. Pac-trees: supporting parallel and compressed purely-functional collections. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 108–121. ACM, 2022. doi: 10.1145/3519939.3523733. URL https://doi.org/10.1145/3519939.3523733.

[56] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Portland, OR, January 1994. ACM Press. URL ftp://ftp.inria.fr/INRIA/Projects/para/doligez/DoligezGonthier94.ps.gz.

[57] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993. URL file://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/publications/concurrent-gc.ps.gz.

[58] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 76–87, Berlin, June 2002. ACM Press. URL http://www.cs.technion.ac.il/~erez/publications.html.

[59] Martin Elsman. A stack machine for region based programs. In SPACE [142]. URL http://www.diku.dk/topps/space2001/program.html#MartinElsman.

[60] Perry A. Emrath and Davis A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[61] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '91*, pages 580–588, November 1991.

[62] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0.

[63] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[64] Jeremy T. Fineman. Provably good race detection that runs in parallel. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2005.

[65] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. *SIGPLAN Not.*, 44(6):121–133, June 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542490.

[66] Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. Linear regions are all you need. In *Proceedings of the 15th Annual European Symposium on Programming (ESOP)*, March 2006.

[67] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008.

[68] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

[69] Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, UK, 2004. URL https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193.

[70] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[71] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *21st Annual ACM Symposium on Parallelism in*

Algorithms and Architectures, pages 79–90, 2009.

[72] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the ACM Symposium on Lisp and Functional Programming (LFP)*, pages 22–38. ACM Press, 1986.

[73] Marcelo J. R. Gonçalves. *Cache Performance of Programs with Intensive Heap Allocation and Generational Garbage Collection*. PhD thesis, Department of Computer Science, Princeton University, May 1995.

[74] Marcelo J. R. Gonçalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *Record of the 1995 Conference on Functional Programming and Computer Architecture*, June 1995.

[75] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 282–293, Berlin, June 2002. ACM Press. ISBN 1-58113-463-0.

[76] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 81–93, 2018.

[77] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17. ACM, 1984.

[78] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[79] Kevin Hammond. Why parallel functional programming matters: Panel statement. In *Reliable Software Technologies - Ada-Europe 2011 - 16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings*, pages 201–205, 2011.

[80] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[81] Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, University of Copenhagen, Universitetsparken 5, 2100 KÃ¸benhavn, 11 2017.

[82] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062354. URL http://doi.acm.org/10.1145/3062341.3062354.

[83] Shams Mahmood Imam and Vivek Sarkar. Habanero-java library: a java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*,

pages 75–86, 2014.

[84] *Intel Cilk++ SDK Programmer's Guide*. Intel Corporation, October 2009. Document Number: 322581-001US.

[85] TBB. *Intel(R) Threading Building Blocks*. Intel Corporation, 2009. Available from http://www.threadingbuildingblocks.org/documentation.php.

[86] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[87] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010.

[88] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Int. Workshop on Languages and Compilers for Parallel Computing*, 1993.

[89] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 157–170. ACM, 2017.

[90] Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. Efficient tree-traversals: Reconciling parallelism and dense data representations. 5 (ICFP), aug 2021. doi: 10.1145/3473596. URL https://doi.org/10.1145/3473596.

[91] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 262–273, New York, NY, USA, 1993. ACM. ISBN 0-8186-4340-4. doi: 10.1145/169627.169724. URL http://doi.acm.org/10.1145/169627.169724.

[92] Lindsey Kuper and Ryan R Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84. ACM, 2013.

[93] Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 2–14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594312. URL http://doi.acm.org/10.1145/2594291.2594312.

[94] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: Quasi-deterministic parallel programming with lvars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 257–270, New York, NY, USA, 2014. ACM.

[95] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*

(PLDI), Orlando, Florida, USA, June 20-24, 1994, pages 24–35, 1994.

[96] Matthew Le and Matthew Fluet. Partial aborts for transactions via first-class continuations. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 230–242, 2015. ISBN 978-1-4503-3669-7.

[97] Doug Lea. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*, pages 36–43, 2000.

[98] Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 25–36. ACM, 2010. doi: 10.1145/1693453.1693459. URL https://doi.org/10.1145/1693453.1693459.

[99] Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 107–118, 2007.

[100] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7.

[101] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. Exploiting vector instructions with generalized stream fusion. *Commun. ACM*, 60(5):83–91, 2017.

[102] Simon Marlow and Simon L. Peyton Jones. Multicore garbage collection with local heaps. In Hans-Juergen Boehm and David F. Bacon, editors, *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, pages 21–32. ACM, 2011.

[103] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, page 49–60, 2013. ISBN 9781450323260.

[104] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33, 1991.

[105] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distributed Syst.*, 15(6):491–504, 2004. doi: 10.1109/TPDS.2004.8. URL https://doi.org/10.1109/TPDS.2004.8.

[106] MLton. MLton web site. http://www.mlton.org, n.d.

[107] Stefan Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. Responsive parallelism with futures and state. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[108] Stefan K. Muller and Umut A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 71–82, 2016.

[109] Stefan K. Muller, Umut A. Acar, and Robert Harper. Responsive parallel computation:

Bridging competitive and cooperative threading. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 677–692, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8.

[110] Stefan K. Muller, Umut A. Acar, and Robert Harper. Competitive parallelism: Getting your priorities right. *Proc. ACM Program. Lang.*, 2(ICFP):95:1–95:30, July 2018. ISSN 2475-1421.

[111] Stefan K. Muller, Sam Westrick, and Umut A. Acar. Fairness in responsive parallelism. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2019, 2019.

[112] Girija J. Narlikar. Scheduling threads for low space requirement and good locality. In *11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 83–95, 1999.

[113] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[114] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. ISSN 1061-7264. doi: 10.1145/289918.289920. URL http://doi.acm.org/10.1145/289918.289920.

[115] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In Rudolf Eigenmann and Martin C. Rinard, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, pages 167–178. ACM, 2003.

[116] Atsushi Ohori, Kenjiro Taura, and Katsuhiro Ueno. Making sml# a general-purpose high-performance language, 2018. Unpublished Manuscript.

[117] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based on sampling functions. *ACM Trans. Program. Lang. Syst.*, 31(1):4:1–4:46, December 2008. ISSN 0164-0925.

[118] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 71–84, 1993.

[119] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, pages 383–414, 2008.

[120] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 392–406, New York, NY, USA, 2016. ACM.

[121] Mike Rainey, Ryan R. Newton, Kyle C. Hale, Nikos Hardavellas, Simone Campanoni, Peter A. Dinda, and Umut A. Acar. Task parallel assembly language for uncompromising parallelism. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pages 1064–1079. ACM, 2021.

[122] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 368–383. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-16611-2.

[123] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, 2012.

[124] John C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 39–46, New York, NY, USA, 1978. ACM.

[125] Dan Robinson. Hpe shows the machine — with 160tb of shared memory. *Data Center Dynamics*, May 2017.

[126] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, August 1967.

[127] Rust Team. Rust language, 2019. URL https://www.rust-lang.org/.

[128] Nikolay Sakharnykh. Maximizing unified memory performance in cuda. https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/, 2017. URL https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/.

[129] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[130] Jacob T. Schwartz. Optimization of very high level languages (parts i and ii). *Computer Languages*, 2–3(1):161–194,197–218, 1975.

[131] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPOPP '13*, pages 135–146, New York, NY, USA, 2013. ACM.

[132] Julian Shun and Guy E. Blelloch. Phase-concurrent hash tables for determinism. In Guy E. Blelloch and Peter Sanders, editors, *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, pages 96–107. ACM, 2014. doi: 10.1145/2612669.2612687. URL https://doi.org/10.1145/2612669.2612687.

[133] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, 2012. ISBN 978-1-4503-1213-4.

[134] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing con-

tention through priority updates. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 152–163, 2013.

[135] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. Proactive work stealing for futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 257–271, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6225-2. doi: 10.1145/3293883.3295735. URL http://doi.acm.org/10.1145/3293883.3295735.

[136] Kyle Singer, Kunal Agrawal, and I-Ting Angelina Lee. Scheduling I/O latency-hiding futures in task-parallel platforms. In Bruce M. Maggs, editor, *1st Symposium on Algorithmic Principles of Computer Systems, APOCS 2020, Salt Lake City, UT, USA, January 8, 2020*, pages 147–161. SIAM, 2020. doi: 10.1137/1.9781611976021.11. URL https://doi.org/10.1137/1.9781611976021.11.

[137] Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. Priority scheduling for interactive applications. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 465–477, 2020.

[138] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Multimlton: A multicore-aware runtime for standard ml. *Journal of Functional Programming*, FirstView: 1–62, 6 2014.

[139] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. Retrofitting parallelism onto ocaml. *Proc. ACM Program. Lang.*, 4(ICFP):113:1–113:30, 2020.

[140] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 387–400. ACM, 2012.

[141] A. Sodani. Knights landing (knl): 2nd generation intel xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, Aug 2015.

[142] SPACE. *Proceedings of the Second workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'01)*, London, January 2001. URL http://www.diku.dk/topps/space2001/.

[143] Daniel Spoonhower. *Scheduling Deterministic Parallel Programs.* PhD thesis, Carnegie Mellon University, May 2009. URL https://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf.

[144] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 91–100, New York, NY, USA, 2009. ACM.

[145] Guy L. Steele, Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94,

pages 472–492, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0.

[146] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 218–231. ACM Press, 1990.

[147] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, page 205–217, 2015. ISBN 9781450336697.

[148] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. PAM: parallel augmented maps. In Andreas Krall and Thomas R. Gross, editors, *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 290–304. ACM, 2018. doi: 10.1145/3178487.3178509. URL https://doi.org/10.1145/3178487.3178509.

[149] Tachio Terauchi and Alex Aiken. Witnessing side effects. *ACM Trans. Program. Lang. Syst.*, 30(3):15:1–15:42, May 2008. ISSN 0164-0925.

[150] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997. URL http://www.diku.dk/research-groups/topps/activities/kit2/infocomp97.ps.

[151] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 83–94, 2016.

[152] David Walker. On linear types and regions. In SPACE [142]. URL http://www.diku.dk/topps/space2001/program.html#DavidWalker.

[153] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. Disentanglement in nested-parallel programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)"*, 2020.

[154] Sam Westrick, Jatin Arora, and Umut A. Acar. Entanglement detection with near-zero cost. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2022, 2022.

[155] Sam Westrick, Mike Rainey, Daniel Anderson, and Guy E. Blelloch. Parallel block-delayed sequences. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 61–75. ACM, 2022. doi: 10.1145/3503221.3508434. URL https://doi.org/10.1145/3503221.3508434.

[156] Sam Westrick, Larry Wang, and Umut A. Acar. DePa: Simple, provably efficient, and practical order maintenance for task parallelism. *CoRR*, abs/2204.14168, 2022. doi: 10.48550/arXiv.2204.14168. URL https://doi.org/10.48550/arXiv.2204.14168.

[157] Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. Parallel determinacy race detection for futures. In Rajiv Gupta and Xipeng Shen, editors, *PPoPP '20: 25th ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020, pages 217–231. ACM, 2020. doi: 10.1145/3332466.3374536. URL https://doi.org/10.1145/3332466.3374536.

[158] Rohan Yadav and Brandon Houghton. Adding fast cuda bindings to standard ml, 2017. URL https://github.com/rohany/StandardML-GPU.

[159] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[160] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 221–234. ACM, 2005.

[161] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

[162] Lukasz Ziarek, K. C. Sivaramakrishnan, and Suresh Jagannathan. Composable asynchronous events. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 628–639, 2011.