Parallel Algorithms for Scheduling Data-Graph Computations

by

William Cleaburn Hasenplaugh

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2016

Author . . . .
Department of Electrical Engineering and Computer Science
January 29, 2016

Certified by . . . .
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Parallel Algorithms for Scheduling Data-Graph Computations

by

William Cleaburn Hasenplaugh

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

A ***data-graph computation*** — popularized by such programming systems as Pregel, GraphLab, Galois, Ligra, PowerGraph, and GraphChi — is an algorithm that iteratively performs local updates on the vertices of a graph. During each round of a data-graph computation, a user-supplied ***update function*** atomically modifies the data associated with a vertex as a function of the vertex's prior data and that of adjacent vertices. A ***dynamic*** data-graph computation updates only an active subset of the vertices during a round, and those updates determine the set of active vertices for the next round. In this thesis, I explore two ways of scheduling deterministic parallel data-graph computations that provide performance guarantees culminating in theoretical contributions to graph theory and practical, high-performance systems. In particular, I describe a system called Prism which processes dynamic and static data-graph computations on arbitrary graphs using a technique called ***chromatic scheduling***. Using a vertex-coloring to identify independent sets of vertices, which may be safely processed in parallel, Prism serializes through the colors and processes the independent sets in parallel, thus executing data-graph computations deterministically and without the use of costly atomic instructions (e.g., Compare-And-Swap). Prism supports dynamic data-graph computations deterministically and work-efficiently through the introduction of ***multibag*** and ***multivector*** data structures.

Prism requires a vertex-coloring, and since graphs are generally not supplied with one, it is necessary to find one as a preprocessing step. Furthermore, the runtime of Prism is linear in the number of colors and thus motivates a study in this thesis of fast parallel coloring algorithms that provide vertex-colorings with few colors in practice. At the core of the analysis of these coloring algorithms lies a new result about the maximum depth of a random priority dag, the dag that results from randomly ordering vertices and directing edges from lower to higher numbered vertices in the order. In particular, when the largest degree Δ in the graph $G = (V, E)$ is less than $\ln |V|$, I show a tight bound on the longest path: $\Theta(\ln V / \ln(e \ln V/\Delta))$ with high probability. When Δ is greater than $\ln |V|$, the longest path in the dag is simply $\Theta\left(\min\{\Delta, \sqrt{E}\}\right)$, also with high probability.

I also present a system called Laika which processes data-graph computations for the special, but important, case of graphs representing physical simulations. Such graphs typically have vertices with coordinates in 3D space and are connected to other "nearby" vertices. We take advantage of these two properties to execute physical simulations, cast as data-graph computations, that make efficient use of cache resources. I analyze a contrived graph construction — a ***random cube graph*** — as a proxy for the mesh graphs that arise in physical simulations: n vertices are uniformly randomly assigned positions in the unit cube and have edges connecting them to any other vertices that are within a distance $r = O\left(V^{-1/3}\right)$. For such a graph and given a cache sufficiently large to hold M vertices, I improve on previous theory to show that a fraction $O(M^{-1/3})$ of edges will connect to vertices not in the cache, whereas previous theory held that this "miss rate" is $O(M^{-1/4})$. LAIKA also guarantees linear speedup for any random cube graph $G = (V, E)$ with constant average degree for any number of workers $P = O\left(V/\lg^2 V\right)$.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

This thesis is dedicated to my favorites: Marcy, Ella, and the Blueberry.

Acknowledgments

Charles E. Leiserson initially cautioned me about coming to MIT as a PhD student, citing the fact that they make you "do stuff." You see, I had a comfortable job at Intel and I could have just joined his lab temporarily as a visiting scientist instead. He used air quotes to make the "stuff" seem ominous and it was not until I found myself on the business end of this thesis that I truly realized what he meant. I applied anyway and I will never forget the look on his face when he realized that I was not kidding: I really had never taken a computer science class before. Needless to say, I required more advising than average, but he was up to the task. Charles is an outstanding advisor, teacher, and human being; he always does the right thing, sleep be damned. Would you like evidence? Just look at the acknowledgments section of any thesis bearing his signature.

It has been a privilege to work with the rest of my world-class thesis committee: Thanks to Nir Shavit for making computer science research cool and working on the hard problems. Thanks to Erik D. Demaine for putting the intellectual ball up just high enough that I might dare to slam it [176].

My friends and collaborators from the Supertech group deserve thanks: Maryam Mehri Dehnavi, Tim Kaler, Bradley Kuszmaul, I-Ting Angelina Lee, Edya Ladan Moses, Tao B. Schardl, Jim Sukha, and Yuan Tang.

Many thanks to my many other collaborators: Dan Alistarh, Tyler Denniston, Fredo Durand, Vineet Gopal, Predrag Gruevski, Syed Kamran Haider, Fredrik Kjolstad, Tuan Andrew Nguyen, Jonathan Ragan-Kelley, Parker Tew, and James J. Thomas.

Thanks to my friends and collaborators over many years at Intel: Michael Adler, Pritpal Ahuja, Brad Burres, Tom Dmukauskas, and Samantika Sury.

Three people have had a particularly profound impact on my life and mind and I would definitely not be writing a PhD thesis at MIT if not for Simon Steely, Jr., Joel Emer, and Tryggve Fossum. Simon and I worked together for many years and I learned that regardless of what idea I might think I have about computer architecture, Simon likely patented it when I was a toddler. Joel guided, clarified, and taxonomized much of the work I did with Simon and others at Intel to our great benefit and he always took time to help guide my career. What compelled Tryggve to bring me into his group all those years ago when I clearly knew nothing about computer architecture? I don't know, but I'm glad he did. He has been a great friend and mentor, and he taught me much of what I know about how processors work and, importantly, *everything* I know about how processors *should* work.

Mark Neifeld, my former advisor at the University of Arizona, deserves much gratitude. He taught me how to be a scientist. Thanks to Brian King, fellow alumni of Mark's group at Arizona, for his friendship and guidance.

Incidentally, Steven Pinker's latest book [167] influenced my writing enough that I would like to thank him here.[1]

Suzanne Patton, Bill and Leilani Hasenplaugh, and Jessica Hasenplaugh[2] deserve special thanks for being supportive and keeping things lively. Rona, Joel, Jill, Jeremy, Victoria, and the rest of Marcy's family have my gratitude for welcoming me into their lives.

Finally, I want to acknowledge that I would never have made it past the first week, and certainly not the full five years, of the PhD program if not for my wife and best friend Marcy. She models what I want for our kids and myself: she does work that matters in the world and she does it well. She's also a relentless kale pusher.

[1]Disclaimer: this does not constitute an invitation for Steven Pinker to critique my writing.

[2]Some of you may know her as "Birdgirl."

Contents

1 Introduction **15**

1.1 Executing dynamic data-graph computations deterministically using chromatic scheduling . . . 18

1.2 Ordering heuristics for parallel graph coloring . . . 23

1.3 Cache-efficient data-graph computations for physical simulations . . . 27

2 Related Work **33**

3 Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling **39**

3.1 Introduction . . . 39

3.2 The Prism algorithm . . . 49

3.3 The multibag data structure . . . 52

3.4 Analysis of Prism . . . 56

3.5 Empirical evaluation . . . 58

3.6 The Prism-R Algorithm . . . 64

3.7 The multivector data structure . . . 68

3.8 Analysis and Evaluation of Prism-R . . . 73

3.9 Conclusion . . . 74

4 Ordering Heuristics for Parallel Graph Coloring 77

4.1 Introduction 77

4.2 The Jones-Plassmann algorithm 85

4.3 Jones-Plassmann with random ordering 87

4.4 The largest-first and smallest-last heuristics 94

4.5 New ordering heuristics 98

4.6 Empirical evaluation 103

4.7 Implementation techniques 108

4.8 The saturation-degree heuristic 111

4.9 Related work 112

4.10 Conclusion 113

5 Cache-Efficient Data-Graph Computations for Physical Simulations 115

5.1 Introduction 115

5.2 Reordering Vertices for Cache Locality 122

5.3 The Laika data-graph computation scheduler 134

5.4 Theoretical analysis of Laika 139

5.5 The Mass-Spring-Dashpot model 145

5.6 Experimental results 152

5.7 Conclusion 161

6 Conclusion 163

A The Cilk Model of Multithreading 165

List of Figures

1-1 Example of chromatic scheduling 18

1-2 Google's PageRank algorithm implemented in a Jacobi and Gauss-Seidel style 19

1-3 PageRank cast as a dynamic data-graph computation 20

1-4 Pseudocode for a serial greedy graph-coloring algorithm 24

1-5 The Jones-Plassmann parallel priority-dag scheduling algorithm 25

1-6 Example of dag scheduling 26

1-7 A dragon and a bunny 28

1-8 A 2D Hilbert space-filling curve 29

1-9 A priority function on vertices with spatial dimensions using the Hilbert space-filling curve 30

1-10 A Russian street dog named Laika 31

2-1 Compressed sparse row graph layout 35

3-1 Google's PageRank algorithm implemented in a Jacobi and Gauss-Seidel style 40

3-2 PageRank cast as a dynamic data-graph computation 41

3-4 Serial data-graph computation algorithm 44

3-6 The PRISM data-graph computation algorithm 51

3-7 The multibag data structure 53

3-8 The MB-INSERT multibag operation 54

3-9 The MB-COLLECT multibag operation 56

3-12 Empirical speedup of PRISM relative to a serial implementation 64

3-13 PRISM scalability versus the number of colors used to color the input graph . 65

3-14 The PRISM-R data-graph computation algorithm 66

3-15 The FLATTEN log tree operation 69

3-16 The Identity and Reduce log tree operations 69
3-17 Example of a log tree reducer 70
3-18 The MV-Collect multivector operation 71
3-19 The MV-Insert multivector operation 72

4-1 The Greedy serial graph-coloring algorithm 78
4-4 The Jones-Plassmann (JP) parallel coloring algorithm 82
4-6 Existential lower bound for the longest path in a random priority dag 92
4-9 An adversarial input graph for the “largest-degree-first” ordering heuristic . . 97
4-10 The “smallest-log-degree-last” ordering heuristic 101
4-15 The “saturation-degree” ordering heuristic 109

5-1 The Stanford dragon 116
5-2 The Jones-Plassmann parallel priority-dag scheduling algorithm 119
5-3 Data-graph computations support hyperedges 120
5-4 Data-graph computations support tetrahedra 120
5-5 Priority function based on the Hilbert space-filling curve 122
5-6 Measured and predicted miss rates averaged across all graphs 124
5-7 Measured and predicted memory bandwidths averaged across all graphs . . . 125
5-8 The probability of cache miss in Laika related to the volume of a spherical cap 127
5-9 Distribution of edge lengths for the cube, random, bunny, and dragon graphs 130
5-10 Measured miss rates for the bunny, dragon, cube, and rand graphs 131
5-11 A bunny emerges from a block of stone 132
5-13 Priority function for the Laika scheduling algorithm 134
5-14 The Laika data-graph computation scheduling algorithm 135
5-15 Initialization function for Laika 135
5-16 The Laika-Worker kernel 136
5-17 Helper functions for Laika 138
5-19 Example of the Mass-Spring-Dashpot model 145
5-20 Example of the Mass-Spring-Dashpot model with detail for a single vertex . . 147
5-21 Length distribution after many iterations of the Mass-Spring-Dashpot model . 148
5-22 Kinetic Energy in the Mass-Spring-Dashpot model per iteration over entire simulation 150

5-23 Kinetic Energy in the Mass-Spring-Dashpot model per iteration at the end of the simulation . . . 151
5-24 The bulk-synchronous parallel (BSP) and Lax data-graph computation schedulers . . . 152

List of Tables

3-3 Convergence rates for Jacobi and Gauss-Seidel implementations of PageRank 43
3-5 Performance summary of dynamic data-graph schedulers 46
3-10 PageRank performance of PRISM 60
3-11 Performance of ROUND-ROBIN, CHROMATIC, and PRISM schedulers for various application benchmarks 62

4-2 Coloring quality performance measurements for serial ordering heuristics . . . 80
4-3 Running time performance measurements for serial ordering heuristics 81
4-5 Summary of ordering-heuristic behavior for graph coloring 84
4-7 Running time performance measurements for parallel coloring algorithms on real-world graphs 95
4-8 Running time performance measurements for parallel coloring algorithms on synthetically generated graphs 96
4-11 Parameters for generating recursive matrices 104
4-12 Description of input graphs 104
4-13 Running time performance measurements for parallel coloring algorithms on real-world graphs 105
4-14 Running time performance measurements for parallel coloring algorithms on synthetically generated graphs 106

5-12 Comparison of performance with and without reordering vertices according to the Hilbert space-filling curve 133
5-18 Performance comparison with and without using a NUMA-aware randomized work-stealing scheduler 140
5-25 Data structure memory usage for data-graph computation schedulers 154

5-26 Graph suite for LAIKA 154
5-27 Measured runtimes for all data-graph computation schedulers and all graph topologies 156
5-28 Measured runtimes for all data-graph computation schedulers and all graph sizes 157
5-29 Instructions per cycle as a function of graph size 158
5-30 Memory-level hardware performance counter measurements of data-graph computation schedulers 159
5-31 Instruction-level hardware performance counter measurements of data-graph computation schedulers 160

Chapter 1

Introduction

Suppose that you are responsible for designing aerodynamic improvements for a critically important fighter jet that has become obsolete. If successful, you may turn the tide of war to your country's advantage, but you are confronted with three constraints:

Quality — You must be sure that your modification is of sound design: Not only must it be better than the current design, it must be reliable.

Time — Lives are lost with every second that your solution is delayed!

Cost — You have only a modest budget and thus can hire very few experts.

How would you proceed? Perhaps you could develop a computerized model for a new *aileron* (i.e., the hinged flap attached to the trailing edge of each wing), which is used to perform the tactically important *roll maneuver*. Next, you could use the finite element method [53, 189] to simulate how wind interacts with the shape of the aileron, ultimately allowing you to measure how much lateral force can be applied to the jet in a roll maneuver. The finite element method operates on a discretized approximation to the aileron, in the form a mesh graph: The finer the mesh, the better the approximation. Here, you encounter a conflict between the quality and time constraints: you need a high-quality estimate of the forces acting on the wing, but the finer you make the mesh, the longer it takes to simulate. It may be possible to use parallel programming to reduce the required simulation time, but parallel programming is famously complex and error-prone. Would it violate the cost constraint if you hired a parallel programming expert?

By indulging this example (i.e., my naive approach to simulating the interaction of air with an aileron) I have unwittingly demonstrated an unfortunate reality: the domain expert (e.g., the aerodynamicist, data scientist, biochemist, etc.) and the algorithms expert (i.e., the person who makes the program go fast, allegedly me) are rarely the same person. Yet, it is important that the work in both domains be done well. The essential motivation in my research is then to help decouple domain-specific expertise from algorithm-specific expertise, enabling more productive work on important,[1] real-life problems. My general approach to this problem is to develop new, practical systems which abstract the scheduling and data-structuring techniques required to achieve high performance from the expression of the domain-specific application. In addition, I analyze these systems theoretically to show that they will perform well, regardless of the input. In particular, this thesis presents new, high-performance systems that simplify the process of expressing a specific, though broad, class of algorithms applied to graphs (e.g., the finite element method and other physical simulations, machine learning algorithms, numerical methods, algorithms from operations research, etc.).

This thesis represents a collaborative effort, unifying work developed and published with several other researchers. In all cases, the contributions described herein are shared with my collaborators and I will identity them in each chapter. When I say, "In Chapter X I will demonstrate that all science is either physics or stamp collecting," I mean that I will present the proof and that I co-discovered the fact with my collaborator, in this case Ernest Rutherford.[2]

Data-graph computations

There is an active body of research which seeks to simplify the expression of a class of iterative algorithms applied to graphs, called ***data-graph computations***, and within this research community there are many popular systems, such as Pregel [145], GraphLab [142,143], Galois [128], Ligra [177] , PowerGraph [90] , and GraphChi [129]. A data-graph computation is an algorithm that iteratively performs local updates on the vertices of a graph and during each round, a user-supplied ***update function*** atomically modifies the data associated with a vertex as a function of the vertex's prior data and that of adjacent vertices. Many thousands

[1]Computer science is abstract and can be appreciated for its own sake, but its ultimate reason for being is to solve just the sort of problem described above. Indeed, modern computers may not have come to exist without the urgency of war [56,74].

[2]For clarity, this is a fictitious example: I have not collaborated with Ernest Rutherford, nor is there any proof of his assertion to my knowledge.

of software developers currently use the data-graph computation engines mentioned above to write simple, fast, and scalable implementations of graph algorithms without needing to know anything about parallel programming. That is, domain experts in physics, machine learning, biochemistry, etc. write the update functions and algorithms experts write the underlying scheduling software to optimize performance. This decoupling of domain-specific expertise and algorithm-specific expertise is a productivity improvement over prior approaches relying on highly specialized, and thus comparatively rare, programmers. Motivated by such productivity improvements, companies are raising substantial venture capital[3] to monetize this revolution in "easy" parallel programming. Nonetheless, the users of such systems face obstacles: at present their programs are either slow, nondeterministic, or double-buffered.[4] In this thesis, I demonstrate that none of these limitations are necessary. In particular, I explore various ways of scheduling deterministic, parallel data-graph computations that provide theoretical performance guarantees culminating in practical, high-performance systems.

In Chapter 2, I give a short history of how the data-graph model of computation came to be and illustrate opportunities for improvement among existing frameworks. In Section 1.1 I introduce one such improvement, a system called PRISM, that processes data-graph computations on arbitrary graphs using a technique called ***chromatic scheduling***. Chromatic scheduling uses a vertex-coloring to identify independent sets of vertices which may be safely processed in parallel without risk of a ***determinacy race***: A determinacy race occurs if two workers[5] may concurrently access a common memory address and at least one of them performs a write. PRISM requires a vertex-coloring as a preprocessing step, and so I investigate deterministic, parallel graph coloring in Section 1.2, the results of which are of independent interest.

In Section 1.3, I introduce a system called LAIKA, which is designed to process data-graph computations for the special, but important, case of mesh graphs used in physical simulations [65, 124]. Such graphs typically have vertices with coordinates in 3D space and are connected to other "nearby" vertices. We take advantage of these properties to execute physical simulations, cast as parallel data-graph computations, making more efficient use of cache resources.

[3]For instance, as of November 2015, GraphLab has been productized by the startup Dato with the support of $25.25M in venture capital.

[4]A double-buffered data-graph computation features two copies of the application data and toggling in alternate cycles one is read-only and the other is write-only.

[5]A "worker" in this context is the same as a "process" in Netzer and Miller's work on race conditions [156].

1.1 Executing dynamic data-graph computations deterministically using chromatic scheduling

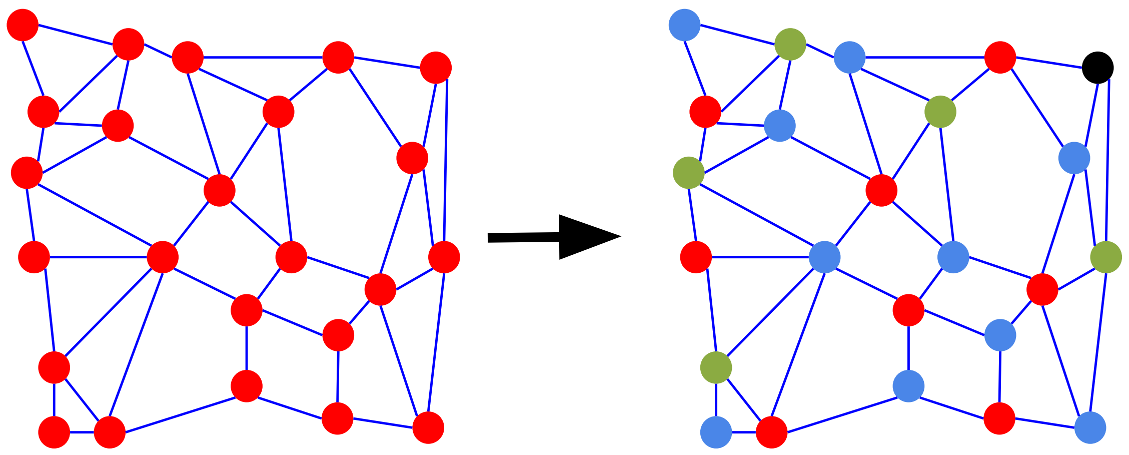

Figure 1-1: Example of how a graph can be partitioned into independent sets of vertices denoted by color, each set of which is able to be executed simultaneously without risk of data races. Iterating through the colors serially and executing the corresponding independent sets in parallel is a technique called ***chromatic scheduling***.

In this section I summarize joint work with Charles E. Leiserson, Tim Kaler, and Tao B. Schardl that was presented at the 2014 ACM Symposium on Parallelism in Algorithms and Architectures under the title "Executing dynamic data-graph computations deterministically using chromatic scheduling" [121].

In this research, I demonstrate theoretically and experimentally that general data-graph computations can be made to be deterministic without giving up high-performance building on a known technique called chromatic scheduling [2, 18, 142], in fact, while increasing performance. First, one finds a vertex-coloring of the graph as depicted in Figure 1-1, an assignment of colors to vertices such that no two neighboring vertices share the same color, and then loop through the colors serially. Since each subset of the vertices of a given color form an independent set (i.e., no two members share an edge) they may be updated simultaneously without causing a determinacy race, assuming that the update function applied to a vertex v reads the data associated with all of its neighbors and writes only the data associated with v. Thus, chromatic scheduling enables deterministic parallel execution of a data-graph computation without *any* concurrent operations on data. PRISM removes the overhead of mutual-exclusion locks or other atomic operations that would be required in a design with concurrent data modification, while also providing a deterministic execution irrespective of the details of the update function. By contrast, other systems [143] ensure atomicity of update functions that concurrently modify data through a nondeterministic locking protocol.

Jacobi vs. Gauss-Seidel

```
PageRank-Jacobi($G, d, \epsilon$)
let $G = (V, E)$
$\delta = \infty$
for $v \in V$
    $P_v = 1/|V|$
while $\delta < \epsilon$
    $\delta = \infty$
    for $v \in V$
        $\hat{P}_v = (1 - d)/|V|$
        for $u \in N(v)$
            $\hat{P}_v = \hat{P}_v + d \cdot P_u/\deg(u)$
        $\delta = \min\{\delta, (P_v - \hat{P}_v)/P_v\}$
    $P = \hat{P}$
return $P$
```

```
PageRank-Gauss-Seidel($G, d, \epsilon$)
let $G = (V, E)$
$\delta = \infty$
for $v \in V$
    $P_v = 1/|V|$
while $\delta < \epsilon$
    $\delta = \infty$
    for $v \in V$
        $\hat{p} = (1 - d)/|V|$
        for $u \in N(v)$
            $\hat{p} = \hat{p} + d \cdot P_u/\deg(u)$
        $\delta = \min\{\delta, (P_v - \hat{p})/P_v\}$
        $P_v = \hat{p}$
return $P$
```

Figure 1-2: Two implementations of Google's PageRank algorithm [35], where PageRanks for each vertex are updated iteratively and stop once the estimates for all vertices in a round change by less than a fraction ϵ. PageRank-Jacobi is a Jacobi-style or double-buffered implementation where in each round estimates for each vertex are based exclusively on the estimates of the previous round. PageRank-Gauss-Seidel is a Gauss-Seidel-style or in-place implementation where a vertex v is updated in round $r + 1$ based on the most recent estimates of each neighbor in $N(v)$, some of which may be from round r and the rest from round $r + 1$. The latter approach converges more quickly, as will be demonstrated in Chapter 3.

One common assumption among many of the existing data-graph computation systems described in Chapter 2 is that the update function may be applied to all vertices simultaneously. For instance, if the vertex data is "double-buffered" and we alternate between the buffers in alternating time steps such that at any one time, one buffer is read-only and one buffer is write-only, then this is a safe assumption. Indeed, many algorithms are written this way, and such algorithms avoid overheads due to concurrent data access. Much like the difference between the Jacobi [199] and Gauss-Seidel [120] iterative methods of solving linear systems of equations, however, an "in-place" or single-buffering method like Gauss-Seidel is often superior in both memory usage and convergence rate [188]. An example of such an application is Google's PageRank algorithm, originally used to *rank* the relevance of web *pages* returned by their search engine [35]. Specifically, for a graph $G = (V, E)$ and a damping factor $d \in [0, 1]$, the pagerank P_v of each vertex $v \in V$ can be circularly defined as a function of its neighbors $N(v) = \{u \in V : (u, v) \in E\}$:

$$P_v = \frac{1-d}{|V|} + d \sum_{\forall u \in N(v)} \frac{P_u}{|N(u)|}.$$

An implementation of the Jacobi and Gauss-Seidel-style implementations of the PageRank algorithm can be found in Figure 1-2, where the primary difference between PageRank-Jacobi and PageRank-Gauss-Seidel is the indentation levels of lines 12 and 25, respectively. That is, in PageRank-Gauss-Seidel we use a scalar temporary variable $\hat{p}$ to collect updates to any particular vertex, whereas PageRank-Jacobi requires a temporary for every vertex, the vector $\hat{P}$, and is updated in bulk at the end of each iteration of the **while** loop.

Dynamic data-graph computations

```
PageRank-Dynamic($G, d, \epsilon$)
let $G = (V, E)$
$P_{1:|V|} = |V|^{-1}$
$r = 0$
$Q_r = V$
while $Q_r \neq \emptyset$
    $Q_{r+1} = \emptyset$
    for $v \in Q_r$
        $\hat{p} = (1-d)/|V|$
        for $u \in N(v)$
            $\hat{p} = \hat{p} + d \cdot P_u / \deg(u)$
        if $(P_v - \hat{p})/P_v > \epsilon$
            $Q_{r+1} = Q_{r+1} \cup v \cup N(v)$
        $P_v = \hat{p}$
    $r = r + 1$
return $P$
```

Figure 1-3: An implementation of Google's PageRank algorithm [35] using a ***dynamic*** update rule. The vertex set Q_r is updated on round r and any vertex v which changes by more than a fraction ϵ and its neighbors $N(v)$ are included in the set Q_{r+1} for execution in the next round. By only updating vertices that change considerably in the previous round, PageRank-Dynamic manages to avoid work and converge to a solution more efficiently.

There are data-graph computations where an additional benefit in convergence rate can be had by only updating a subset of vertices in each round.[6] Such an algorithm is called a ***dynamic*** data-graph computation, an example of which is a variant of PageRank, shown in Figure 1-3, where only certain vertices are updated in a round: typically those that change significantly in the previous round and their neighbors. Practitioners find that this dynamic version of PageRank is faster and gives them "relevant" answers, even if the static version (i.e., one that updates every vertex every round) yields a slightly "better" answer. Many algorithms in machine learning feature this tradeoff, including loopy belief propagation [154, 165], coordinate descent [60], co-EM [160], alternating least-squares [107], singular-value decomposition [89], and matrix factorization [193].

To demonstrate the relative convergence rates of PageRank-Jacobi, PageRank-Gauss-Seidel, and PageRank-Dynamic, I ran each one on a collection of graphs, described and used extensively in Chapter 4, and summarized the results in Table 3-3 in Chapter 3. The static Gauss-Seidel method performs 2.5 times fewer updates in geometric mean than the Jacobi method, and the dynamic Gauss-Seidel method performs 3.5 times fewer updates in geometric mean than the Jacobi method. This performance advantage motivates my interest in supporting dynamic in-place data-graph computations. Unfortunately, an in-place data-graph computation must cope with data races when updating neighboring vertices deterministically in parallel, and a dynamic data-graph computation must make careful use of data structures to ensure that the resulting algorithm is ***work-efficient***: the overheads incurred in an effort to parallelize the computation are at most linear in the work used by the serial implementation. I seek these features not only because I am not a savage, but because determinism is essential to the development of reliable software [28, 132] and work-efficiency is a virtue in an increasingly power-constrained world [78]. In this research, I address both challenges in developing and analyzing systems that enable deterministic, work-efficient, in-place, dynamic data-graph computations while still guaranteeing good parallel performance.

[6]In particular, the convergence rate in terms of the total work performed (i.e., number of *vertex* updates times the number of neighbors in each update) can be improved by updating only a subset of vertices in any given round, the subset typically being the vertices whose values changed significantly in the previous round.

Performance guarantees

I present the performance of Prism in the language of work-span analysis [52, Ch. 27], an overview of which is in Appendix A. For now, it suffices to understand that the ***work*** of a computation is the total number of instructions executed, and the ***span*** corresponds to the longest path of dependencies in the parallel program. For convenience, I assume[7] that a single update executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span. Under this assumption, on a data-graph G of degree Δ, Prism executes the updates in round r on the activation set Q_r containing vertices colored using χ colors on P processors using $O(Q_r + P)$ work and $O(\chi(\lg(Q_r/\chi) + \lg \Delta) + \lg P)$ span.[8] While the theoretical parallelism depends on the size of Q_r, I present extensive empirical evidence that Prism is highly scalable in practice. Surprisingly, the "price of determinism" incurred by the use of chromatic scheduling instead of the more common locking strategy appears to be negative for real-world applications. For instance, as detailed in Chapter 3, Prism executes 1.1–2.3 times faster than GraphLab's comparable, but nondeterministic, locking strategy on a machine with 12 Intel Core-i7 (Nehalem) processor cores

Prism behaves deterministically as long as every update is ***pure***: it modifies no data except for that associated with its target vertex. This assumption precludes the update function from modifying global variables to aggregate or collect values. To support this common use pattern, I will discuss an extension to Prism called Prism-R in Chapter 3. It executes dynamic data-graph computations deterministically even when updates modify global variables using associative operations (e.g., a reducer hyperobject [76]). Prism-R uses a new "multivector" data structure whose contents are ordered deterministically, whereas Prism uses a simpler "multibag" data structure. Nonetheless, Prism-R executes in the same theoretical bounds as Prism despite its more complicated implementation. Empirically, Prism-R is only 1.04 times slower in geometric mean than Prism and outperforms the nondeterministic lock-based version of GraphLab on all application benchmarks described in Chapter 3.

[7]Other assumptions about the work and span of an update can easily be made at the potential expense of complicating the analysis.

[8]For convenience, I abuse the meaning of the notation Q_r depending on the context. In pseudocode, Q_r is the set of vertices that are executed in round r. In big-O notation, it refers to the size of the set Q_r and all neighboring vertices (i.e., the work incurred by executing every vertex in Q_r).

1.2 Ordering heuristics for parallel graph coloring

In this section I summarize joint work with Charles E. Leiserson, Tim Kaler, and Tao B. Schardl that was presented at the 2014 ACM Symposium on Parallelism in Algorithms and Architectures under the title "Ordering heuristics for parallel graph coloring" [102].

Graph coloring is a heavily studied problem with many real-world applications, including the scheduling of conflicting jobs [6, 73, 146, 197], register allocation [34, 42, 43], high-dimensional nearest-neighbor search [14], sparse-matrix computation [50, 119, 172], etc. Formally, a ***(vertex)-coloring*** of an undirected graph $G = (V, E)$ is an assignment of a ***color*** $v.color$ to each vertex $v \in V$ such that for every edge $(u, v) \in E$, we have $u.color \neq v.color$, that is, no two adjacent vertices have the same color. The ***graph-coloring problem*** is the problem of determining a coloring which uses as few colors as possible.

I am interested in the graph coloring problem as a means to speed up chromatic scheduling [2, 18, 121] of parallel data-graph computations. As we saw in Section 1.1, a chromatic scheduler first colors the vertices of the data-graph G and then sequences through the colors, scheduling all vertices of the same color in parallel. The time to perform a data-graph computation thus depends both on how long it takes to initially color G and on the number of colors produced by the graph-coloring algorithm: more colors means less parallelism. Although the coloring can be performed offline for some data-graph computations, for others the coloring must be produced online, and one must accept a trade-off between coloring ***quality*** — the number of colors used — and the time to produce the coloring.

Although the problem of finding an ***optimal*** coloring of a graph — a coloring using the fewest colors possible — is in NP-complete [79], heuristic "greedy" algorithms work reasonably well in practice. Welsh and Powell [197] introduced the original ***greedy*** coloring algorithm, shown in Figure 1-4, which iterates over the vertices and assigns each vertex the smallest color not yet assigned to a neighbor.

Ordering heuristics

In practice, however, greedy coloring algorithms tend to produce much better colorings than the $\Delta + 1$ bound implies, and moreover, the order in which a greedy coloring algorithm colors the vertices affects the quality of the coloring. To reduce the number of colors used by a greedy coloring algorithm, practitioners therefore employ ***ordering heuristics*** to determine the order in which the algorithm colors the vertices [4, 32, 32, 50, 118, 141, 147, 197].

The literature includes many studies of ordering heuristics and how they affect running time and coloring quality in practice. Here are three popular heuristics:

R The ***random*** ordering heuristic [118] colors vertices in a uniformly random order.

LF The ***largest-degree-first*** ordering heuristic [197] colors vertices in order of decreasing degree.

SL The ***smallest-degree-last*** ordering heuristic [4, 147] colors the vertices in the order induced by first removing all the lowest-degree vertices from the graph, then recursively coloring the resulting graph, and finally coloring the removed vertices.

The experimental results overviewed in Chapter 4 indicate that I have listed these heuristics in order of coloring quality from worst to best and in order of running time from fastest to slowest.

```
GREEDY($G$)
42 let $G = (V, E, \rho)$
43 for $v \in V$ in order of decreasing $\rho(v)$
44     $C = \{1, 2, \ldots, \deg(v) + 1\}$
45     for $u \in N(v) : \rho(u) > \rho(v)$
46         $C = C - \{u.color\}$
47     $v.color = \min C$
```

Figure 1-4: Pseudocode for a serial greedy graph-coloring algorithm. Given a vertex-weighted graph $G = (V, E, \rho)$, where the priority of a vertex $v \in V$ is given by $\rho(v)$, GREEDY colors each vertex $v \in V$ in decreasing order according to $\rho(v)$.

Although an ordering heuristic can be viewed as producing a permutation of the vertices of a graph $G = (V, E)$, it is convenient to think of an ordering heuristic as producing an injective (1-to-1) ***priority function*** $\rho : V \to \mathbb{R}$ (breaking ties randomly). Figure 1-4 gives the pseudocode for GREEDY, a greedy coloring algorithm. GREEDY takes a vertex-weighted graph $G = (V, E, \rho)$ as input, where ρ is a priority function produced by some ordering heuristic. Each step of GREEDY simply selects the uncolored vertex with the highest priority according to ρ and colors it with the smallest available color.

Parallel greedy coloring via dag scheduling

There is a historical tension between coloring quality and the parallel scalability of greedy graph coloring. While the traditional ordering heuristics are efficient using GREEDY, I show in Chapter 4 that any parallelization of them requires worst-case span of $\Omega(V)$ for a general

```
JP($G$)
48  let $G = (V, E, \rho)$
49  parallel for $v \in V$
50      $v.pred = \{u \in N(v) : \rho(u) > \rho(v)\}$
51      $v.succ = \{u \in N(v) : \rho(u) < \rho(v)\}$
52      $v.counter = |v.pred|$
53  parallel for $v \in V$
54      if $v.pred == \emptyset$
55          JP-UPDATE($v$)

JP-UPDATE($v$)
56  UPDATE($v$)
57  parallel for $u \in v.succ$
58      if JOIN($u.counter$) $== 0$
59          JP-UPDATE($u$)

UPDATE($v$)
60  // Finds lowest available color
61  $C = \{1, 2, \ldots, |v.pred| + 1\}$
62  parallel for $u \in v.pred$
63      $C = C - \{u.color\}$
64  $v.color = \min C$
```

Figure 1-5: The Jones-Plassmann [118] parallel priority-dag scheduling algorithm, shown here as JP, is a generalization of Jones and Plassmann's original distributed vertex-coloring algorithm. A pictorial example of JP can be found in Figure 1-6. JP uses a recursive helper function JP-UPDATE to process a vertex using the user-supplied UPDATE function once all of its predecessors have been updated, recursively calling JP-UPDATE in line 59 for any successor u who is eligible to be updated (i.e., when $u.counter == 0$). The function JOIN decrements its argument and returns the post-decrement value. To implement the original Jones-Plassmann vertex-coloring algorithm, the UPDATE(v) function merely assigns the lowest available color to v not already taken by any of its predecessors $v.pred$.

graph $G = (V, E)$, with the exception of the random ordering heuristic. Of the various attempts to parallelize greedy coloring [49, 66, 144], the algorithm first proposed by Jones and Plassmann [118] extends the greedy algorithm in a straightforward manner, uses work linear in size of the graph, and is deterministic given a small random seed. Furthermore, Jones and Plassmann's original paper demonstrates good theoretical parallel performance for $O(1)$-degree graphs using the random ordering heuristic.

Figure 1-5 gives the pseudocode for JP, which colors a given graph $G = (V, E, \rho)$ in the order specified by the priority function ρ. The JP and JP-UPDATE functions form a general dag scheduling algorithm,[9] where the user-supplied UPDATE function implements a specific data-graph computation, in this case vertex-coloring. The algorithm begins in lines 50 and 51 by partitioning the neighbors of each vertex into ***predecessors*** — vertices with larger priorities — and ***successors*** — vertices with smaller priorities. The priority function ρ orients the edges in the graph such that the resulting directed graph forms a dag, as depicted in Figure 1-6. JP uses the recursive JP-UPDATE helper function to update a vertex $v \in V$ once all vertices in $v.pred$ have been updated. Initially, lines 53–55 in JP scan the vertices of V to find every vertex that has no predecessors and updates each one using JP-UPDATE. Within a call to JP-UPDATE(v), line 56 calls the user-supplied function UPDATE to, in this case, assign a color to v, and the loop on lines 57–59 broadcasts in parallel to all of v's successors the fact that v is updated. For each successor $u \in v.succ$, line 58 tests whether all of u's predecessors have already been updated, and if so, line 59 recursively calls JP-UPDATE(u).

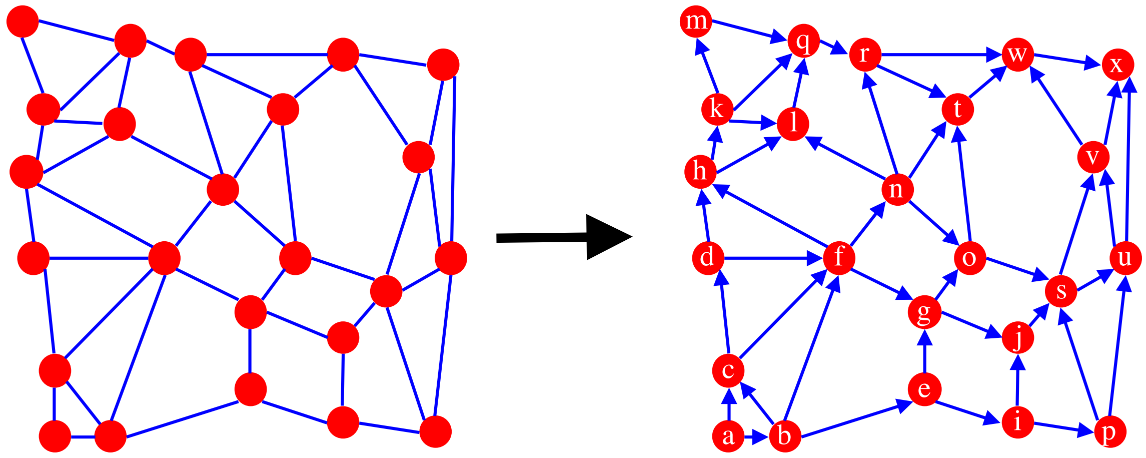

Figure 1-6: An alternative to chromatic scheduling which yields a deterministic output is dag scheduling. A priority function $\rho : V \to \mathbb{R}$ is used to create a partial order on the vertices and orienting an edge from low to high priority results in a dag. For simplicity, the vertices are shown with a random permutation of letters $\{a, b, \ldots, x\}$, in lieu of a random priority value. The vertices are processed in dag order: a vertex is not processed until all of its predecessors have been processed.

[9] Actually, Jones and Plassmann specifically designed a distributed coloring algorithm using a random priority function and analyzed it for $O(1)$-degree graphs. Here, I merely generalize their scheduling technique for use with generic update functions.

Parallel ordering heuristics

Jones and Plassmann showed that the expected length of the longest path in a random priority dag G_ρ, the dag induced on $G = (V, E)$ by a random priority function ρ, is $O(\lg V / \lg \lg V)$ for any $O(1)$-degree graph, leading to an upper bound on the running time of the corresponding parallel algorithm. In Chapter 4 I give matching upper and lower bounds for the expected length of the longest directed path in a random priority dag of arbitrary degree. In particular, for a graph $G = (V, E)$ of degree $\Delta = O(\lg V)$, the expected length of the longest path in G_ρ is $\Theta\left(\ln V / \ln\left(e \ln V / \Delta\right)\right)$ with high probability. For $\Delta = \Omega(\lg V)$, the expected length of the longest path is $\Theta(\min\{\Delta, \sqrt{E}\})$.

Although JP with a random priority function scales well in theory and in practice, it is one of the weaker ordering heuristics in terms of coloring quality (i.e., it tends to produce many colors). Of the other heuristics, it is possible to construct adversarial graphs that cause JP with largest-first and smallest-last to scale poorly, requiring $\Omega(V)$ time. Consequently, I focus on alternatives to largest-first and smallest-last that provide comparable coloring quality while guaranteeing good parallel performance. Specifically, in Chapter 4 I will describe two new parallel ordering heuristics — "largest-log-degree-first" and "smallest-log-degree-last" — which resemble largest-first and smallest-last, respectively, but which scale provably well when used with JP, requiring at most $O\left(\lg \Delta\right)$ times more span, for Δ-degree graphs. They both provide good parallel scalability in theory and practice: they achieve speedups of 6–8 times on a machine with 12 Intel Core-i7 (Nehalem) processor cores and are resilient to adversarial graphs.

1.3 Cache-efficient data-graph computations for physical simulations

In this section I summarize joint work with Predrag Gruevski, Charles E. Leiserson, and James J. Thomas called "Cache-efficient data-graph computations for physical simulations" [96].

Chapter 5 investigates an important special case of scheduling data-graph computations: the specific problem of performing physical simulations on mesh graphs. For example, the domain-specific language Simit [124] can be used to describe physical simulations (e.g., fluid dynamics [17], the n-body problem [168,183,196], etc.) on a static ***mesh graph***, a wire mesh discretization of a continuous object in 3D space,[10] like the scary dragon and cuddly bunny depicted in Figure 1-7. The Simit compiler generates code for an update function which is applied to all vertices over many time steps (e.g., typically millions), where the update function typically approximates some physical force (e.g., Newton's laws of motion [157]). The scheduling technique described in Chapter 5 is general to other physical simulation systems (e.g., Liszt [65]), but we use Simit as an illustrative motivation.

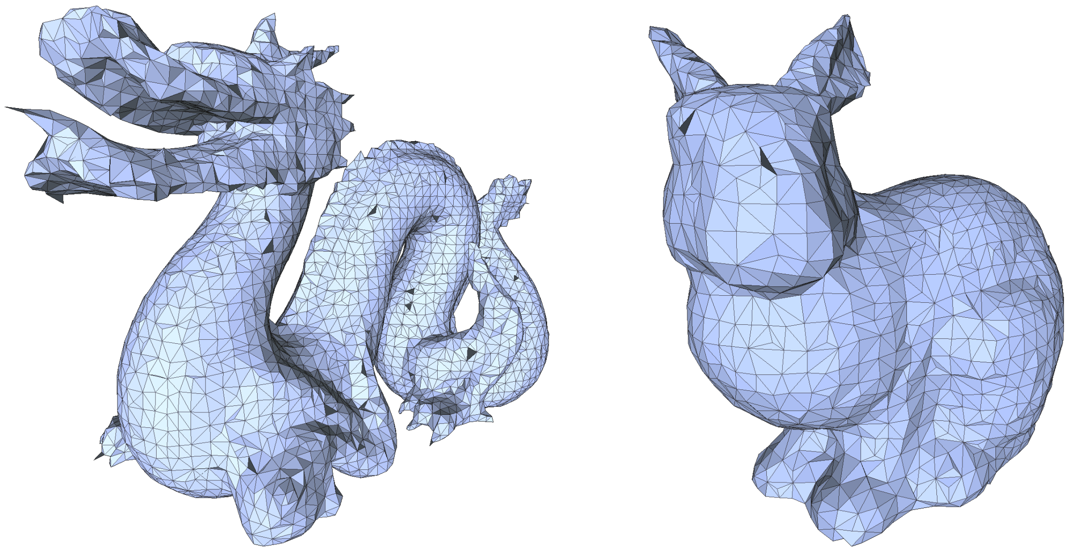

Figure 1-7: Mesh graph examples where lines correspond to edges and intersections of lines correspond to vertices.

While chromatic scheduling enables high parallelism without data races, it can be inefficient for cache usage. For instance, to process the update function of a vertex v of color c the worker needs to read data associated with all of its neighbors $N(v)$, but by virtue of being in different color sets by definition, each vertex $w \in N(v)$ can not be processed until after all vertices of color c have been processed. This squanders the potential cache advantage of processing the neighbors of v soon after v itself is processed, while they are still in cache.

An alternative approach to chromatic scheduling is ***dag scheduling*** [118], depicted in Figures 1-5 and 1-6 and used extensively in my research on parallel graph coloring. In dag scheduling, the graph is turned into a dag through the use of a priority function $\rho : V \to \mathbb{R}$. In particular, an undirected edge connecting vertices v and w is oriented as (v, w) if $\rho(v) > \rho(w)$ (ties are broken by comparing the vertex numbers). The vertices are then processed in dag order, meaning that a vertex v may be processed only once all of its predecessors have been

[10]Simit is capable of describing objects in a space of arbitrary dimension, but is generally used to specify 3D simulations.

processed. This scheduling approach affords us the opportunity to process vertices shortly after they are read by their neighbors, a potential caching advantage.

In Chapter 5, I will describe a variation of dag scheduling optimized for the special case of physical simulations which uses a priority function designed to exploit cache resources while minimizing the overhead introduced by scheduling logic. I will also demonstrate empirically that this system, called LAIKA, is able to transform a traditionally memory bandwidth-bound problem [7, 65, 70, 83, 91, 139, 162] into a compute-bound problem on a machine with 12 Intel Core-i7 (Nehalem) processor cores. For example, I tested the performance of LAIKA executing a physical simulation, typical of the type expressed in Simit, on a set of four graphs, each of which is approximately 1.9GB in size. LAIKA is 10.89 times faster running on 12 cores than on 1 core and LAIKA running on 12 cores is 44.27 times faster than a naive serial implementation.[11]

The Hilbert space-filling curve

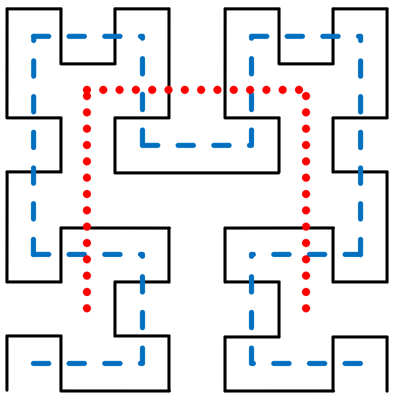

Figure 1-8: Three recursion levels of a 2D Hilbert space-filling curve [106].

LAIKA makes use of a new priority function that makes dag scheduling of data-graph computations on mesh graphs especially cache-efficient. This priority function is first used to reorder the vertices in the graph to improve ***spatial locality***.[12] In particular, we use the bounding box of the mesh graph in 3D space to normalize the graph to the unit cube. Then,

[11]This baseline is equivalent to Simit's current shared-memory implementation.

[12]A cache block $\mathcal{C}$ is made up of B words. Suppose an algorithm reads a particular word, bringing $\mathcal{C}$ into the cache. If it then reads a different word in $\mathcal{C}$ while it is still in cache, this is an example of spatial locality.

we decompose the unit cube into a regular $2^k \times 2^k \times 2^k$ grid, each grid point of which is assigned a scalar value corresponding to the closest point on the discretized the Hilbert space-filling curve [106]. A 2D example of the Hilbert space-filling curve[13] is given in Figure 1-8. The red dotted curve is the first recursion level and illustrates the basic inverted "U" shape. The blue dashed curve shows how each quadrant is partitioned into four independent first-level Hilbert curves (up to rotations) of half the size in each dimension. The black solid curve illustrates the third recursion level. All vertices are assigned to the closest grid point and assigned the corresponding scalar value along the Hilbert curve, as depicted in Figure 1-9. This scalar value is a vertex's value in ***Hilbert curve space***. Since some vertices may be assigned to the same grid point, ties are broken in the priority function randomly. Thus, the vertices are processed in the order dictated by the Hilbert curve iterating through the 3D grid.

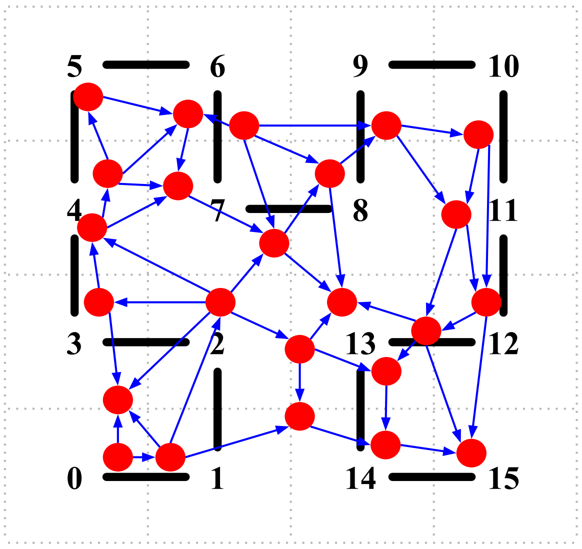

Figure 1-9: Example of how a locally-connected graph in 2 dimensions is mapped to a dag via a second-order Hilbert curve priority function. Each vertex is mapped to its closest grid point in the discretized Hilbert curve. Among vertices mapping to the same grid point, ties are broken randomly.

Intuitively, we see why the Hilbert curve might be a good ordering for the vertices by considering that mesh graphs are locally connected, meaning that the neighborhood of a vertex is typically nearby in 3D space. One well-known property of the Hilbert curve is that points that are close together in Hilbert curve space are also close in 3D space [92]. However,

[13]Many other space-filling curves exist [10, 39, 173] and many would be appropriate for use in LAIKA, however, we specifically chose the Hilbert curve because it is known to produce better cache behavior, up to constant factors, than others which are simpler to compute (e.g., Z-order [153]).

it is also true that randomly chosen points that are close together in 3D space are quite likely to be close in Hilbert curve space, as well [152, 192]. This property leads to excellent cache behavior, since the neighbors of each vertex are close in 3D space for mesh graphs, and will thus tend to also be close in memory.

Figure 1-10: The Russian street dog Laika is one of the first and most famous animals to travel through space.

We call the priority-dag scheduling algorithm using the Hilbert curve priority function, LAIKA.[14] In Chapter 5, I describe a novel theoretical contribution to the relationship between distances in 3D space and distances in Hilbert curve space, improving on the results of Tirthapura, Seal, and Aluru [192], which leads to a stronger bound on cache performance for LAIKA. In particular, for n vertices uniformly randomly distributed in the unit cube and connected to other vertices within a distance r — a ***random cube graph*** — Tirthapura, Seal, and Aluru's bound states that a traversal of such a graph, reordered using the Hilbert curve priority function,[15] and using a cache sufficiently large to hold M vertices will incur $O(M^{-1/4})$ misses per edge. I will show that, in fact, we incur only $O(M^{-1/3})$ misses per edge. Furthermore, LAIKA achieves linear expected speedup on any random cube graph $G = (V, E)$ with constant expected degree for any number of workers $P = O(V/\lg^2 V)$.

The use of space-filling curves for locality-preserving load-balancing is a known technique. Algorithms for the n-body problem [183,196], database layout and scheduling [152], resource

[14]We take naming inspiration from the graph processing libraries GraphLab [142], which is named after a Labrador Retriever, and GraphChi [129], which is named after a Chihuahua. Laika, pictured in Figure 1-10, was a Russian street dog that was used in early space exploration [8] and we chose this name because Laika is a dog who travels through space, much like the Hilbert curve.

[15]Tirthapura, Seal, and Aluru's paper does not actually discuss priority functions, I merely extrapolate their results to the context of data-graph computations of locally connected physical simulations.

scheduling [138], and dynamic load balancing [101] all use variations on the general theme of mapping ND space onto a 1D curve that is subsequently partitioned among P processors. My contribution to the empirical aspect of this research area is the use of the Hilbert space-filling curve with dag scheduling and the development of Laika, which scales to large datasets while preserving deterministic execution and excellent cache usage.

Chapter 2

Related Work

In recent years, there has been growing interest in the development of frameworks for storage and analysis of data on large compute clusters, Hadoop [55, 59] being among among the most popular of these. Hadoop breaks up large datasets into pieces distributed across many shared-memory multicore nodes in a cluster, each of which communicates via a message-based network protocol. Users supply computations, or ***map*** operators, that are evaluated over each of the pieces independently and other computations, or ***reduce*** operators, that combine the results. Many problems can be cast into the Hadoop model, but in many cases the Hadoop approach is far less effecient than more specialized methods optimized for graph algorithms, as we will explore throughout this research.

The idea behind recent big data frameworks, including Hadoop, is to decouple scheduling and data layout from the expression of the computation, enabling high programmer productivity and portable, best-in-class performance. Iterative graph algorithms, however, are one class of problem not well-suited to the Hadoop approach. In particular, graphs are difficult to split into completely independent sets (with no crossing edges) for the map phase of a Hadoop computation, so the maps are often wasteful [145]. The idea of decoupling data and scheduling from the expression of the algorithm, however, is useful for designing frameworks for graph algorithms, even if Hadoop itself is ill-suited to the task.

A new paradigm

Malewicz et al. proposed the Pregel programming interface [145] and Low et al. developed the GraphLab framework [142] to abstract scheduling from the application-specific specification of iterative graph algorithms, initially targeting machine learning algorithms. These systems popularized the “think like a vertex” mindset of graph-centric algorithm frameworks, and many such systems have been subsequently developed, optimizing different corners of the design space. In particular, the Pregel / GraphLab interface is used to express a data-graph computation, described in Chapter 1, with a user-specified update function and application-specific data. The update function is iteratively applied to some subset of vertices, taking as inputs the data associated with each vertex's neighbors. Many interesting big data algorithms, including Google's PageRank, can be easily expressed under this model. Examples of traditional graph algorithms that can be cast as data-graph computations include maximal independent set, maximal matching, vertex and edge coloring, breadth-first search, and triangle counting [29]. Other examples arise in numerical methods and machine-learning algorithms on sparse matrices, including loopy belief propagation [154, 165], coordinate descent [60], co-EM [160], alternating least-squares [107], singular-value decomposition [89], and matrix factorization [193]. The systems which currently implement the data-graph computation interface, however, do not support work-efficient, deterministic, dynamic, in-place updates, nor do any of them exploit the special structure of mesh graphs in physical simulations to optimize performance. My collaborators and I solve both problems in Chapters 3 and 5. Rather, most work in the field of data-graph computation engines is concerned with performance-engineering techniques applied to the simplest version of the data-graph computation problem (i.e., on every iteration update every vertex, simultaneously, in parallel), which amounts to the design of data layout and algorithmic techniques optimized for random accesses to memory.

Existing Systems

In the distributed setting, Pregel [145], Distributed GraphLab [142], PowerGraph [90], Naiad [155], PowerLyra [47], and GPS [174] provide a parallel programming model focused on graphs. These systems often aim to minimize communication across nodes through data replication and partitioning, as well as to avoid synchronization overheads. For example, PowerGraph [90] accounts for the highly skewed degree distribution in real-world power-law graphs by splitting popular vertices across nodes to perform partial aggregation.

In the shared-memory setting, many systems have been developed for multicore machines and have typically shown significantly better performance than distributed frameworks given sufficient memory capacity. Ligra [177] is an efficient multicore system that supports update functions for both vertices and edges. Ligra switches between multiple edge mapping implementations depending on the number of active vertices to take advantage of the choice of dense vs. sparse updates. Galois [158] provides a multicore framework which does dataflow-style scheduling with user-defined priorities and also provides a vertex-update interface. GraphMat [190] converts vertex-centric programs with Scatter-Gather-Apply stages per step into sparse matrix-vector primitives, and on many graph algorithms offers the best performance among published systems. Its authors also find its performance to be within 1.2x of hand-optimized code. Two other relevant systems are GraphChi [129] and X-Stream [171], which focus on streaming through data sequentially from either a disk or from DRAM, and performing random access on a much smaller fraction of the data stored in DRAM or cache respectively. Polymer [204] is a NUMA-optimized framework that provides the Ligra interface and implements optimizations similar to both distributed and shared-memory systems. It focuses on minimizing both random access and cross-NUMA-node access.

Memory access optimizations

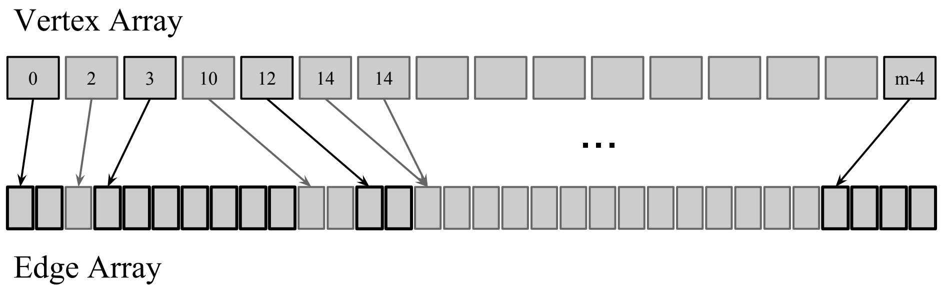

Figure 2-1: Graphs can be stored in memory on a single cache-coherent multicore in a compressed sparse row format. The vertex array contains vertex data and an index into an edge array, which contains vertex IDs of the associated neighbors.

To optimize memory accesses in particular, many of the currently fastest systems try to lay out data in a sequential manner and to reduce working sets. Ligra, Galois and GraphMat use a "Compressed Sparse Row" layout for edges, as depicted in Figure 2-1, where the edge lists for each vertex are sequential in memory, to turn most accesses to edge data into sequential scans. Many of these systems also use compressed data structures (e.g., using a bit vector to track active nodes or even compressing IDs in edge lists [180]). The fastest systems for multicores still perform random accesses to DRAM for vertex data, however, and these accesses consume most of their execution time. Finally, vertex-centric updates like those in PageRank are analogous to sparse matrix-vector multiply problems, for which a great variety of data layouts and parallelization techniques have been studied [13, 198, 202]. These include layouts such as "Compressed Sparse Blocks" that maintain locality in both rows and columns [37], and cache-oblivious Hilbert curve orderings [98, 201]. It should be noted that a sparse matrix-vector multiply is a special-case of a single iteration of a data-graph computation, where a general data-graph computation makes use of a user-supplied update function and user-defined per-vertex state whereas the matrix-vector multiply performs an inner product. In both cases, however, the data access pattern is the same, perhaps motivating GraphMat [190] to use routines similar to those used in sparse matrix libraries in its own data-graph computation implementation.

Researchers have made much progress in recent decades on an important and pervasive special case of data-graph computations: physical simulations and graphics applications. Brandt [31] developed the multigrid adaptive solver for boundary-value problems, an algorithmic approach to reducing the overall work required in such physical simulations, which exploits the observation that physically nearby vertices behave similarly and thus may be initially approximated coarsely. Warren and Salmon [196] developed an efficient, albeit nondeterministic, parallel algorithm for n-body simulation, which uses a Z-order curve [153] to generate better empirical cache usage. Similarly, Singh et al. [183] use orthogonal recursive bisection [75, p. 37–62] to empirically improve the performance of the fast multipole method [95]. Neither work provides any theoretical guarantees of performance or analysis of the expected cache behavior. Hoppe uses a similar intuition for ordering vertices in computer graphics applications [109], showing empirically that a greedy algorithm for storing adjacent vertices can reduce memory bandwidth requirements, though the resulting algorithm remains memory-bound. More generally, Asanovic et al. [7] analyzed the landscape of many programming patterns — the so-called "dwarves" — and classified physical simulations as memory-bound.

Some researchers have gone to more extreme lengths to combat the memory-boundedness of physical simulation. For instance, Goodnight et al. [91] applied Brandt's multigrid approach to a GPU implementation in order to take advantage of higher memory bandwidth only to find that the implementation is awkward for the GPU and that memory bandwidth still limits performance. Feichtinger et al. [70] continue in this path, painstakingly mapping a lattice Boltzmann method for computational fluid dynamics to a distributed GPU system, where the network bandwidth dominates. Devito et al. [65] developed a domain-specific language for describing mesh-based partial differential equation solvers, as a way of enabling scientists to take advantage of high GPU memory bandwidth, without the awkwardness of actually programming a GPU. Lindtjorn et al. [139] and Giefers, Plessl, and Förstner [83] have gone to more extreme measures to tame the memory bandwidth thirst of physical simulation: they use field-programmable gate array (FPGA) processor chips to hardcode the computational pipeline and shift the balance of hardware resources toward memory bandwidth. In Chapter 5 I demonstrate how my coauthors and I transform physical simulations on mesh graphs into a compute-bound problem, thus enabling high performance on shared-memory multicore computers, which are comparatively easy to program.

Chapter 3

Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling

3.1 Introduction

Many systems from physics, artificial intelligence, and scientific computing can be represented naturally as a ***data graph*** — a graph with data associated with its vertices and edges. For example, some physical systems can be decomposed into a finite number of elements whose interactions induce a graph. Probabilistic graphical models in artificial intelligence can be used to represent the dependency structure of a set of random variables. Sparse matrices can be interpreted as graphs for scientific computing.

A data-graph computation is an algorithm that performs "local" updates on the vertices of a data graph, taking as input data associated with a vertex and its neighbors. Several software systems have been implemented to support parallel data-graph computations, including GraphLab [142, 143], Pregel [145], Galois [158, 159], PowerGraph [90], Ligra[1] [177, 180], and GraphChi [129]. These systems can support "complex" data-graph computations, in which data can be associated with edges as well as vertices and updating a vertex v can modify any data associated with v, v's incident edges, and the vertices adjacent to v. For

[1]While Ligra does not technically execute data-graph computations, it is designed to implement similar algorithms by decoupling the scheduling and algorithm-specific code, as with the other data-graph computation frameworks.

ease in discussing chromatic scheduling, however, we shall principally restrict ourselves to "simple" data-graph computations (which correspond to "edge-consistent" computations in GraphLab), although most of our results straightforwardly extend to more complex models. Indeed, six out of the seven GraphLab applications described in [142, 143] are simple data-graph computations.

Jacobi vs. Gauss-Seidel

```
PageRank-Jacobi($G, d, \epsilon$)
**let** $G = (V, E)$
$\delta = \infty$
**for** $v \in V$
    $P_v = 1/|V|$
**while** $\delta < \epsilon$
    $\delta = \infty$
    **for** $v \in V$
        $\hat{P}_v = (1 - d)/|V|$
        **for** $u \in N(v)$
            $\hat{P}_v = \hat{P}_v + d \cdot P_u/\deg(u)$
        $\delta = \min\{\delta, (P_v - \hat{P}_v)/P_v\}$
    $P = \hat{P}$
**return** $P$
```

```
PageRank-Gauss-Seidel($G, d, \epsilon$)
**let** $G = (V, E)$
$\delta = \infty$
**for** $v \in V$
    $P_v = 1/|V|$
**while** $\delta < \epsilon$
    $\delta = \infty$
    **for** $v \in V$
        $\hat{p} = (1 - d)/|V|$
        **for** $u \in N(v)$
            $\hat{p} = \hat{p} + d \cdot P_u/\deg(u)$
        $\delta = \min\{\delta, (P_v - \hat{p})/P_v\}$
        $P_v = \hat{p}$
**return** $P$
```

Figure 3-1: Two implementations of Google's PageRank algorithm [35], where PageRanks for each vertex are updated iteratively and stop once the estimates for all vertices in a round change by less than a fraction ϵ. PageRank-Jacobi is a Jacobi-style or double-buffered implementation where in each round estimates for each vertex are based exclusively on the estimates of the previous round. PageRank-Gauss-Seidel is a Gauss-Seidel-style or in-place implementation where a vertex v is updated in round $r + 1$ based on the most recent estimates of each neighbor in $N(v)$, some of which may be from round r and the rest from round $r + 1$. The latter approach converges more quickly, as summarized in Table 3-3.

One common assumption among many of the existing data-graph computation systems described in Chapter 2 is that the update function may be applied to all vertices simultaneously. For instance, if the vertex data is "double-buffered" and we alternate between the buffers in alternating time steps such that at any one time, one buffer is read-only and one buffer is write-only, then this is a safe assumption. Indeed, many algorithms are written this way and such algorithms avoid overheads due to concurrent data access. However, much like the difference between the Jacobi [199] and Gauss-Seidel [120] iterative methods of solving linear systems of equations, an "in-place" or single-buffering method like Gauss-Seidel is often superior in both memory usage and convergence rate [188]. An example of such an application is Google's PageRank algorithm, originally used to *rank* the relevance of web *pages* returned by their search engine [35]. In particular, for a graph $G = (V, E)$ and a damping factor $d \in [0, 1]$, the pagerank P_v of each vertex $v \in V$ can be circularly defined as a function of its neighbors $N(v) = \{u \in V : (u, v) \in E\}$:

$$P_v = \frac{1-d}{|V|} + d \sum_{\forall u \in N(v)} \frac{P_u}{|N(u)|}.$$

An implementation of the Jacobi and Gauss-Seidel-style implementations of PageRank can be found in Figure 3-1, where the primary difference between PageRank-Jacobi and PageRank-Gauss-Seidel is the indentation levels of lines 76 and 89, respectively. That is, in PageRank-Gauss-Seidel we use a scalar temporary variable $\hat{p}$ to collect updates to any particular vertex, whereas PageRank-Jacobi requires a temporary for every vertex, the vector $\hat{P}$, and is updated in bulk at the end of each iteration of the **while** loop.

Dynamic data-graph computations

```
PageRank-Dynamic($G, d, \epsilon$)
 91  let $G = (V, E)$
 92  for $v \in V$
 93      $P_v = 1/|V|$
 94  $r = 0$
 95  $Q_r = V$
 96  while $Q_r \neq \emptyset$
 97      $Q_{r+1} = \emptyset$
 98      for $v \in Q_r$
 99          $\hat{p} = (1-d)/|V|$
100          for $u \in N(v)$
101              $\hat{p} = \hat{p} + d \cdot P_u/\deg(u)$
102          if $(P_v - \hat{p})/P_v > \epsilon$
103              $Q_{r+1} = Q_{r+1} \cup v \cup N(v)$
104          $P_v = \hat{p}$
105      $r = r + 1$
106  return $P$
```

Figure 3-2: An implementation of Google's PageRank algorithm [35] using a ***dynamic*** update rule. The vertex set Q_r is updated on round r and any vertex v which changes by more than a fraction ϵ and its neighbors $N(v)$ are included in the set Q_{r+1} for execution in the next round. By only updating vertices that change considerably in the previous round, PageRank-Dynamic manages to avoid work and converge to a solution more efficiently, as summarized in Table 3-3.

There are data-graph computations where an additional benefit in convergence rate can be had by only updating a subset of vertices in each round.[2] Such an algorithm is called a ***dynamic*** data-graph computation, an example of which is a variant of PageRank, shown in Figure 3-2, where only certain vertices are updated in a round: typically those that change significantly in the previous round and their neighbors. Practitioners find that this dynamic version of PageRank is faster and gives them "relevant" answers, even if the static version (i.e., one that updates every vertex every round) yields a slightly "better" answer. Many algorithms in machine learning feature this tradeoff, including loopy belief propagation [154, 165], coordinate descent [60], co-EM [160], alternating least-squares [107], singular-value decomposition [89], and matrix factorization [193]

To demonstrate the relative convergence rates of PageRank-Jacobi, PageRank-Gauss-Seidel, and PageRank-Dynamic, we ran each one on a collection of graphs, described and used extensively in Chapter 4, and summarized the results in Table 3-3. The static Gauss-Seidel method performs 2.5 times fewer updates in geometric mean than the Jacobi method and the dynamic Gauss-Seidel method performs 3.5 times fewer updates in geometric mean than the Jacobi method. This performance advantage motivates my interest in supporting dynamic in-place data-graph computations. Unfortunately, an in-place data-graph computation must cope with data races when updating neighboring vertices deterministically in parallel and a dynamic data-graph computation must make careful use of data structures to ensure that the resulting algorithm is ***work-efficient***: if the overheads incurred in an effort to parallelize the computation are at most linear in the work used by the serial implementation, the parallel execution is work-efficient. We seek these features because determinism is essential to the development of reliable software [28, 132] and work-efficiency is a virtue in an increasingly power-constrained world [78]. In this research, we address both challenges in developing and analyzing systems that enable deterministic, work-efficient, in-place, dynamic data-graph computations while still guaranteeing good parallel performance.

[2]In particular, the convergence rate in terms of the total work performed (i.e., number of *vertex* updates times the number of neighbors in each update) can be improved by updating only a subset of vertices in any given round, the subset typically being the vertices whose values changed significantly in the previous round.

Graph	$\lvert E\rvert$	$\frac{\lvert E\rvert}{\lvert V\rvert}$	Jacobi	Gauss-Seidel	Dynamic	$\frac{\text{Jacobi}}{\text{Gauss-Seidel}}$	$\frac{\text{Jacobi}}{\text{Dynamic}}$
com-orkut	117.2	38.1	21	11	8.3	1.91	2.53
liveJournal1	42.9	8.8	33	13	11.0	2.54	3.00
europe-osm	36.0	0.7	30	10	3.8	3.00	7.89
cit-Patents	16.5	2.7	32	12	8.2	2.67	3.90
as-skitter	11.1	1.0	34	13	11.1	2.62	3.06
wiki-Talk	4.7	1.9	33	14	13.0	2.36	2.54
web-Google	4.3	4.7	34	13	6.8	2.54	4.98
com-youtube	3.0	2.5	33	13	12.0	2.51	2.75

Table 3-3: Convergence rates for Jacobi (i.e., double-buffered) and Gauss-Seidel (i.e., in-place) implementations of Google's PageRank algorithm [35] using a damping factor of 0.85. Each algorithm is iterated until the PageRank estimate for each vertex changes by less than 1% in the round. The coloumns $|E|$ and $|E|/|V|$ give the number of edges (in millions) and the ratio of edges to vertices, respectively, for the graph named in the first column. The column "Jacobi" gives the total amount of work performed divided by the number of edges, for comparison across graphs, using a double-buffered approach estimates of a vertex's PageRank for round $r+1$ are a function of round r [188]. The column "Static" uses an in-place Gauss-Seidel implementation where the estimate of a vertex's PageRank in round $r+1$ uses the most recent estimates (i.e., some from round r and some from $r+1$) from each of its neighbors. The version in column "Dynamic" only updates vertices in a round that have changed or have neighbors that have changed by more than 1% in the previous round.

Problem statement

We formalize the computational model as follows. Let $G = (V, E)$ be a data graph. Denote the ***neighbors***, or ***adjacent vertices***, of a vertex $v \in V$ by $N(v) = \{u \in V : (u, v) \in E\}$. The ***degree*** of v is thus $\deg(v) = |N(v)|$, and the ***degree*** of G is $\deg(G) = \max\{\deg(v) : v \in V\}$. A ***(simple) dynamic data-graph computation*** is a triple $\langle G, f, Q_0\rangle$, where

- $G = (V, E)$ is a graph with data associated with each vertex $v \in V$;
- $f : V \to 2^V$ is an ***update function***; and
- $Q_0 \subseteq V$ is the initial ***activation set***.

The update $S = f(v)$ implicitly computes as a side effect a new value for the data associated with v as a function of the old data associated with v and v's neighbors. The update returns a set $S \subseteq N(v)$ of vertices that must be updated in the next round. For example, an update $f(v)$ might activate a neighbor u only if the value of v changes significantly. During a round

r of a dynamic data-graph computation, each vertex $v \in Q_r$ is updated at most once, that is, Q_r is a set, not a multiset.

A serial reference implementation

Before we address the issues involved in scheduling and executing dynamic data-graph computations in parallel, let us first hone our intuition with a serial implementation. Figure 3-4 gives the pseudocode for SERIAL-DDGC. This algorithm schedules the updates of a data-graph computation by maintaining a FIFO queue Q of activated vertices that have yet to be updated. Sentinel values enqueued in Q on lines 110 and 115 demarcate the rounds of the computation such that the set of vertices in Q after the rth sentinel has been enqueued is the activation set Q_r for round r.

```
SERIAL-DDGC($G, f, Q_0$)
107  for $v \in Q_0$
108      ENQUEUE($Q, v$)
109  $r = 0$
110  ENQUEUE($Q$, NIL) // Sentinel NIL denotes the end of a round.
111  while $Q \neq \{$NIL$\}$
112      $v$ = DEQUEUE($Q$)
113      if $v$ == NIL
114          $r$ += 1
115          ENQUEUE($Q$, NIL)
116      else
117          $S = f(v)$
118          for $u \in S$
119              if $u \notin Q$
120                  ENQUEUE($Q, u$)
```

Figure 3-4: Pseudocode for a serial algorithm to execute a data-graph computation $\langle G, f, Q_0 \rangle$. SERIAL-DDGC takes as input a data graph G and an update function f. The computation maintains a FIFO queue Q of activated vertices that have yet to be updated and sentinel values NIL, each of which demarcates the end of a round. An update $S = f(v)$ returns the set $S \subseteq N(v)$ of vertices activated by that update. Each vertex $u \in S$ is added to Q if it is not currently scheduled for a future update.

Given a data-graph $G = (V, E)$, an update function f, and an initial activation set Q_0, SERIAL-DDGC executes the data-graph computation $\langle G, f, Q_0 \rangle$ as follows. Lines 107–108 initialize Q to contain all vertices in Q_0. The **while** loop on lines 111–120 then repeatedly dequeues the next scheduled vertex $v \in Q$ on line 111 and executes the update $f(v)$ on line 117. Executing $f(v)$ produces a set S of activated vertices, and lines 118–120 check each vertex in S for membership in Q, enqueuing all vertices in S that are not already in Q.

We can analyze the time SERIAL-DDGC takes to execute one round r of the data-graph computation $\langle G, f, Q_0 \rangle$. Define the ***size*** of an activation set Q_r as

$$size(Q_r) = |Q_r| + \sum_{v \in Q_r} \deg(v) \ .$$

The size of Q_r is asymptotically the space needed to store all the vertices in Q_r and their incident edges using a standard sparse-graph representation, such as compressed-sparse-rows (CSR) format [187]. For example, if $Q_0 = V$, we have $size(Q_0) = |V| + 2|E|$ by the handshaking lemma [52, p. 1172–3]. Let us make the reasonable assumption that the time to execute $f(v)$ serially is proportional to $\deg(v)$. If we implement the queue as a dynamic (resizable) table [52, Section 17.4], then line 120 executes in $\Theta(1)$ amortized time. Of course, a linked list would suffice to append operations in $\Theta(1)$ time, but would not allow for convenient subsequent parallel iteration over its elements. All other operations in the **for** loop on lines 118–120 take $\Theta(1)$ time, and thus all vertices activated by executing $f(v)$ are examined in $\Theta(\deg(v))$ time. The total time spent updating the vertices in Q_r is therefore $\Theta(Q_r + \sum_{v \in Q_r} \deg(v)) = \Theta(size(Q_r))$, which is ***linear*** time: time proportional to the storage requirements for the vertices in Q_r and their incident edges.

Parallelizing dynamic data-graph computations

The salient challenge in parallelizing data-graph computations is to deal effectively with races between updates, that is, logically parallel updates that read and write common data. A ***determinacy race*** [71] (also called a ***general race*** [156]) occurs when two logically parallel instructions access the same memory location and at least one of them writes to that location. Two updates in a data-graph computation ***conflict*** if executing them in parallel produces a determinacy race. A parallel scheduler must manage or avoid conflicting updates to execute a data-graph computation correctly and deterministically.

The standard approach to preventing races associates a mutual-exclusion lock with each vertex of the data graph to ensure that an update on a vertex v does not proceed until all locks on v and v's neighbors have been acquired. Although this locking strategy prevents races, it can incur substantial overhead from lock acquisition and contention, hurting application performance, especially when update functions are simple. Moreover, because runtime happenstance can determine the order in which two logically parallel updates acquire locks,

Benchmark	$\lvert V \rvert$	$\lvert E \rvert$	χ	RRLocks	Cilk+ Locks	Prism	Prism-R
PR/G	916	5,105	43	15.5	14.3	9.7	12.6
PR/L	4,847	68,475	333	227.6	200.4	109.3	127.3
ID/2000	4,000	15,992	4	48.6	43.8	32.1	32.8
ID/4000	16,000	63,984	4	200.0	179.6	123.1	124.3
FBP/C1	87	265	2	8.7	8.9	6.9	7.0
FBP/C3	482	160	2	16.4	17.8	13.3	13.4
ALS/N	187	20,597	6	134.3	123.6	105.2	105.7

Table 3-5: Comparison of dynamic data-graph schedulers on seven application benchmarks. Column "*Graph*" identifies the input graph, and columns $|V|$ and $|E|$ specify the number of vertices and edges in the graph in thousands, respectively. All runtimes are in seconds and were calculated by taking the median 12-core execution time of 5 runs on an Intel Xeon X5650 with hyperthreading disabled. The runtimes of Prism and Prism-R include the time used to color the input graph. PR/G and PR/L run a PageRank algorithm on the web-Google [137] and soc-LiveJournal [9] graphs, respectively. ID/2000 and ID/4000 run an image denoise algorithm to remove Gaussian noise from 2D grayscale images of dimension 2000 by 2000 and 4000 by 4000. FBP/C1 and FBP/C3 perform belief propagation on a factor graph provided by the cora-1 and cora-3 datasets [148,184]. ALS/N runs an alternating least squares algorithm on the NPIC-500 dataset [150].

the data-graph computation can act nondeterministically: different runs on the same inputs can produce different results. Without repeatability, parallel programming is arguably much harder [28, 132]. Nondeterminism confounds debugging.

A known alternative to using locks is ***chromatic scheduling*** [2,18,142], which schedules a data-graph computation based on a coloring of the data-graph computation's ***conflict graph*** — a graph with an edge between two vertices if updating them in parallel would produce a race. For a simple data-graph computation, the conflict graph is simply the data graph itself with undirected edges. The idea behind chromatic scheduling is fairly simple. Chromatic scheduling begins by computing a ***(vertex) coloring*** of the conflict graph — an assignment of colors to the vertices such that no two adjacent vertices share the same color. Since no edge in the conflict graph connects two vertices of the same color, updates on all vertices of a given color can execute in parallel without producing races. To execute a round of a data-graph computation, the set of activated vertices Q is partitioned into χ ***color sets*** — subsets of Q containing vertices of a single color. Updates are applied to vertices in Q by serially stepping through each color set and updating all vertices within a color set in parallel. Indeed, the special case where the active set $Q == V$ is the entire graph (i.e., a static data-graph computation) can be executed using chromatic scheduling using Distributed GraphLab [142]. The result of a data-graph computation executed using chromatic scheduling is equivalent to that of a slightly modified version of SERIAL-DDGC that starts each round (immediately before line 115 of Figure 3-4) by sorting the vertices within its queue by color.

Chromatic scheduling avoids both of the pitfalls of the locking strategy. First, since only nonadjacent vertices in the conflict graph are updated in parallel, no races can occur, and the necessity for locks and their associated performance overheads are precluded. Second, by establishing a fixed order for processing different colors, any two adjacent vertices are always processed in the same order. The data-graph computation is therefore executed deterministically, as long as a deterministic coloring algorithm is used to color the conflict graph. While chromatic scheduling potentially loses parallelism because colors are processed serially, we shall see that this concern does not appear to be an issue in practice.

To date, chromatic scheduling has been applied to static data-graph computations [142], but not to dynamic data-graph computations. This chapter addresses the question of how to perform chromatic scheduling efficiently when the activation set changes on-the-fly, necessitating a data structure for maintaining dynamic sets of vertices in parallel.

Contributions

This chapter represents joint work with Charles E. Leiserson, Tim Kaler, and Tao B. Schardl that was presented at the 2014 ACM Symposium on Parallelism in Algorithms and Architectures under the title "Executing dynamic data-graph computations deterministically using chromatic scheduling" [121].

This chapter introduces PRISM, a chromatic-scheduling algorithm that executes dynamic data-graph computations in parallel efficiently in a deterministic fashion. PRISM employs a "multibag" data structure to manage an activation set as a list of color sets. The multibag achieves efficiency using "worker-local storage," which is memory locally associated with each "worker" thread executing the computation. By using the "multibag" and a deterministic coloring algorithm, PRISM guarantees to execute the data-graph computation deterministically.

We analyze the performance of PRISM using work-span analysis [52, Ch. 27], which is described in detail in Appendix A. The ***work*** of a computation is the total number of instructions executed, and the ***span*** corresponds to the longest path of dependencies in the parallel program. We shall make the reasonable assumption that a single update $f(v)$ executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span.[3] Under this assumption, on a degree-Δ data graph G colored using χ colors, PRISM executes the updates on the vertices in the activation set Q_r of a round r on P processors in $O(\mathit{size}(Q_r) + P)$ work and $O\left(\chi\left(\lg\left(Q_r/\chi\right) + \lg\Delta\right) + \lg P\right)$ span.

The "price of determinism" incurred by using chromatic scheduling instead of the more common locking strategy appears to be negative for real-world applications. This discovery is perhaps surprising since it would seem to be strictly harder to guarantee that the computation behave deterministically than to allow for nondeterministic behaviors. Nevertheless, as Table 3-5 indicates, on seven application benchmarks, PRISM executes 1.2–2.1 times faster than GraphLab's comparable, but nondeterministic, locking strategy, which we call RRLOCKS. This performance gap is not due solely to superior engineering or load balancing. A similar performance overhead is observed in a comparably engineered lock-based scheduling algorithm, CILK+LOCKS. PRISM outperforms CILK+LOCKS on each of the 7 application benchmarks and is on average (geometric mean) 1.4 times faster.

Our contribution is not a full-featured data-graph computation framework like GraphLab, Pregel, Galois, PowerGraph, Ligra, or GraphChi. Each of these systems is the result of countless hours of performance engineering and feature support. Instead, we provide a scheduling technique that could be adopted by any such framework to enable the deterministic execution of work-efficient, dynamic data-graph computations, which no existing framework currently supports[4] We use a modified shared-memory version of GraphLab in order to isolate the effect of our scheduling algorithms. Thus, the empirical comparisons in this chapter are apples-to-apples comparisons of scheduling strategies, not competitive comparisons with other systems.

PRISM behaves deterministically as long as every update is ***pure***: it modifies no data except for that associated with its target vertex. This assumption precludes the update function from modifying global variables to aggregate or collect values. To support this common use pattern, we describe an extension to PRISM, called PRISM-R, which executes dynamic data-graph computations deterministically even when updates modify global variables using

[3]Other assumptions about the work and span of an update can easily be made at the potential expense of complicating the analysis.

[4]Deterministic Galois [159] has added support for deterministic execution of dynamic data-graph computations by recursively removing and executing independent sets of vertices. However, their algorithm is not work-efficient and, as a result, is much slower than the nondeterministic version.

associative operations. PRISM-R replaces each multibag PRISM uses with a "multivector," maintaining color sets whose contents are ordered deterministically. PRISM-R executes in the same theoretical bounds as PRISM, but its implementation is more involved. Empirically PRISM-R is on average (geometric mean) only 1.07 times slower than PRISM and outperforms CILK+LOCKS on all but one of the seven application benchmarks.

Outline

The remainder of this chapter is organized as follows. Section 3.2 describes PRISM, the chromatic-scheduling algorithm for dynamic data-graph computations. Section 3.3 describes the multibag data structure PRISM uses to represent its color sets. Section 3.4 presents our theoretical analysis of PRISM. Section 3.5 describes a Cilk Plus [113] implementation of PRISM and presents empirical results measuring this implementation's performance on seven application benchmarks. Section 3.6 describes PRISM-R which executes dynamic data-graph computations deterministically even when update functions modify global variables using associative operations. Section 3.7 describes and analyzes the multivector data structure PRISM-R uses to represent deterministically ordered color sets. Section 3.8 analyzes PRISM-R both theoretically, using work-span analysis, and empirically. Section 3.9 offers some concluding remarks.

3.2 The Prism algorithm

This section presents PRISM, a chromatic-scheduling algorithm for executing dynamic data-graph computations deterministically. We describe how PRISM differs from the serial algorithm in Section 3.1, including how it maintains activation sets that are partitioned by color using the multibag data structure.

Figure 3-6 shows the pseudocode for PRISM, which differs from the SERIAL-DDGC routine from Figure 3-4 in two main ways: the use of a multibag data structure to implement Q, and the call to COLOR-GRAPH on line 121 to color the data graph.

A ***multibag*** Q represents a list $\langle C_0, C_1, \ldots, C_{\chi-1}\rangle$ of χ ***bags*** (unordered multisets) and supports two operations:

- MB-INSERT(Q, v, k) inserts an item v into bag C_k in Q. A multibag supports parallel MB-INSERT operations.

- MB-COLLECT(Q) produces a collection $\mathcal{C}$ that represents a list of the nonempty bags in Q, emptying Q in the process.

Although the multibag data structure supports duplicate items in a single bag, our implementation of PRISM actually ensures that no duplicate vertices are ever inserted into a bag.

PRISM calls COLOR-GRAPH on line 121 to color the given data graph $G = (V, E)$ and obtain the number χ of colors used. Although it is NP-complete to find an ***optimal*** coloring of a graph [79] — a coloring that uses the smallest possible number of colors — an optimal coloring is not necessary for PRISM to perform well, as long as the data graph is colored deterministically, in parallel,[5] and with sufficiently few colors in practice. Many parallel coloring algorithms exist that satisfy the needs of PRISM (see, for example, [5, 11, 86, 87, 102, 118, 126, 127, 140, 191]), however, our implementation of PRISM uses a multicore variant of the Jones and Plassmann algorithm [118] that produces a deterministic $(\Delta + 1)$-coloring of a Δ-degree graph $G = (V, E)$ in linear work and $O\left(\ln V + \lg \Delta \cdot \min\{\sqrt{E}, \Delta + \ln V / \ln(e \ln V/\Delta)\}\right)$ span, which is described in Chapter 4.

Let us now see how PRISM uses chromatic scheduling to execute a dynamic data-graph computation $\langle G, f, Q_0 \rangle$. After line 121 colors G, line 123 initializes the multibag Q with the initial activation set Q_0, and then the **while** loop on lines 124–133 executes the rounds of the data-graph computation. At the start of each round, line 125 collects the nonempty bags $\mathcal{C}$ from Q, which correspond to the nonempty color sets for the round. Lines 126–132 iterate through the color sets $C \in \mathcal{C}$ sequentially, and the **parallel for** loop on lines 127–132 processes the vertices of each C in parallel. For each vertex $v \in C$, line 129 performs the update $S = f(v)$, which returns a set S of activated vertices, and lines 130–132 insert into Q the vertices in S that have been activated.

Although a vertex u can be activated by multiple neighbors, it must only be updated at most once during a round. PRISM enforces this constraint[6] by using the atomic ***compare-and-swap*** operator [105, p. 480], which is available as a synchronization primitive on most machines and whose definition is given in lines 134–140. Lines 130–132 use the CAS primitive

[5]If the data-graph computation performs sufficiently many updates, a serial $\Theta(V + E)$-work greedy coloring algorithm, such as that introduced by Welsh and Powell [197], can suffice as well, since the time to color the graph would be sufficiently amortized against the work performed.

[6]This constraint may be enforced without the use of an atomic compare-and-swap operation by deduplicating the contents of Q at the start of each round. However, our empirical studies have shown that this limited use of atomics is beneficial in practice.

```
PRISM($G, f, Q_0$)
$\chi$ = COLOR-GRAPH($G$)
$r = 0$
$Q = Q_0$
while $Q \neq \emptyset$
    $\mathcal{C}$ = MB-COLLECT($Q$)
    for $C \in \mathcal{C}$
        parallel for $v \in C$
            $active[v]$ = FALSE
            $S = f(v)$
            parallel for $u \in S$
                if CAS($active[u]$, FALSE, TRUE)
                    MB-INSERT($Q, u, color[u]$)
        $r = r + 1$
```

```
CAS($current, test, value$)
begin atomic
if $current == test$
    $current = value$
    return TRUE
else
    return FALSE
end atomic
```

Figure 3-6: Pseudocode for PRISM, including the compare-and-swap synchronization primitive CAS. The procedure PRISM takes as input a data graph G, an update function f, and an initial activation set Q_0. The procedure COLOR-GRAPH colors a given graph and returns the number of colors it used. The procedures MB-COLLECT and MB-INSERT operate the multibag Q to maintain activation sets for PRISM. The variable r tracks the number of rounds executed.

to activate each vertex $u \in S$ by atomically setting $active[u]$ = TRUE, and if $active[u]$ was previously FALSE, then calling MB-INSERT. Thus, each vertex is inserted into Q at most once during a round.

Design considerations for the implementation of multibags

The theoretical performance of PRISM depends upon the properties of the multibag data structure. In particular, the multibag is carefully designed to ensure that PRISM is ***work-efficient*** — that is, it performs the same asymptotic work as the serial algorithm SERIAL-DDGC in Figure 3-4. Before examining the design of the multibag in Section 3.3, let us first explore why maintaining active color sets in PRISM in a work-efficient manner is tricky. Specifically, we shall consider two alternative strategies: bit vectors and an array of worker-local queues.

The bit-vector approach avoids the multibag altogether and simply manages activation sets using the bit vector *active* already used by PRISM. Recall that if $active[i]$ is TRUE, then the vertex $v_i \in V$ indexed by i is active. Suppose that *active* were the only data structure. To iterate over all activated vertices of color k, a **parallel for** could scan through *active*, updating the vertex v_i whenever $active[i]$ is TRUE and $color[i]$ is k. This scheme requires

$\Omega(V\chi)$ work per round of the computation, where χ is the number of colors returned by COLOR-GRAPH in line 121 of Figure 3-6, since the entire bit vector must be scanned χ times each round. At the cost of additional preprocessing, *active* could be organized such that vertices of the same color are assigned contiguous indexes. Even with this optimization, however, scanning *active* requires $\Omega(V)$ work each round, which is not work-efficient for dynamic computations that activate only a sparse subset of the vertices each round.

An alternative strategy that one might consider is to represent the active color sets using an array of worker-local queues. A straightforward implementation of this approach, however, is also not work-efficient. For a dynamic data-graph computation using χ colors and P processors, a total of $P\chi$ worker-local queues would be needed to maintain the set of active vertices, and $\Omega(P\chi)$ work would be required to collect all nonempty queues. As we shall see in Section 3.3, however, by using a carefully designed data structure to manage worker-local queues, we can obtain a work-efficient data structure for maintaining color sets.

3.3 The multibag data structure

This section presents the multibag data structure employed by PRISM. The multibag uses worker-local sparse accumulators [84] and an efficient parallel collection operation. We describe how the MB-INSERT and MB-COLLECT operations are implemented, and we analyze them using work-span analysis [52, Ch. 27]. When used in a P-processor execution of a parallel program, a multibag Q of χ bags storing n elements supports MB-INSERT in $\Theta(1)$ worst-case time and MB-COLLECT in $O(n+\chi+P)$ work and $O(\lg n+\chi+\lg P)$ span. Such a multibag storing k elements uses $O(P\chi+k)$ space.

A ***sparse accumulator (SPA)*** [84] implements an array that supports lazy initialization of its elements. A SPA T contains a sparsely populated array $T.array$ of elements and a log $T.log$, which is a list of indices of initialized elements in $T.array$. To implement multibags, we shall only need the ability to create a SPA, access an arbitrary SPA element, or delete all elements from a SPA. For simplicity, we shall assume that an uninitialized array element in a SPA has a value of NIL. When an array element $T.array[i]$ is modified for the first time, the index i is appended to $T.log$. An appropriately designed SPA T storing $n = |T.log|$ elements admits the following performance properties:

- Creating T takes $\Theta(1)$ work.

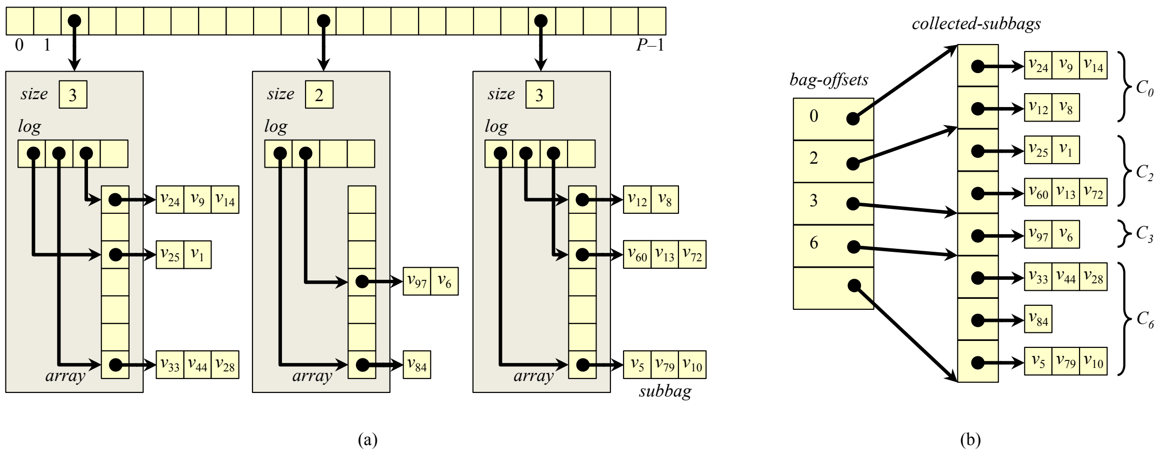

Figure 3-7: A multibag data structure. **(a)** A multibag containing 19 elements distributed across 4 distinct bags: $\{C_0, C_2, C_3, C_6\}$, representing vertices of colors 0, 2, 3, and 6, respectively. Each worker keeps track of its portion of a particular bag, its ***subbag***, using a worker-local SPA, thus avoiding initialization of unused subbags by maintaining a compact ***log*** pointing to the set of populated subbags. For example, bag C_6 is composed of three subbag contributions from the three active workers: $\{v_{33}, v_{44}, v_{28}\}$, $\{v_{84}\}$, and $\{v_5, v_{79}, v_{10}\}$. **(b)** The output of MB-COLLECT when executed on the multibag in **(a)**. Sets of subbags in *collected-subbags* are labeled with the bag C_k that their union represents.

- Each element of T can be accessed in $\Theta(1)$ work.
- Reading all k initialized elements of T takes $\Theta(k)$ work and $\Theta(\lg k)$ span.
- Emptying T takes $\Theta(1)$ work.

A multibag Q is an array of P worker-local SPA's, where P is the number of workers executing the program. We shall use p interchangeably to denote either a worker or that worker's unique identifier. Worker p's local SPA in Q is thus denoted by $Q[p]$. For a multibag Q of χ bags, each SPA $Q[p]$ contains an array $Q[p].array$ of size χ and a log $Q[p].log$. Figure 3-7(a) illustrates a multibag with $\chi = 7$ bags, 4 of which are nonempty. As Figure 3-7(a) shows, the worker-local SPA's in Q partition each bag $C_k \in Q$ into subbags $\{C_{k,0}, C_{k,1}, \ldots, C_{k,P-1}\}$, where $Q[p].array[k]$ stores subbag $C_{k,p}$.

Implementation of MB-INSERT *and* MB-COLLECT

The worker-local SPA's enable a multibag Q to support parallel MB-INSERT operations without creating races. Figure 3-8 shows the pseudocode for MB-INSERT. When a worker p executes MB-INSERT(Q, v, k), it inserts element v into the subbag $C_{k,p}$ as follows. Line 141 calls GET-WORKER-ID to get worker p's identifier. Line 142 checks if subbag $C_{k,p}$ stored in $Q[p].array[k]$ is initialized, and if not, lines 143 and 144 initialize it. Line 145 inserts v into $Q[p].array[k]$.

```
MB-INSERT($Q, v, k$)
141  $p = $ GET-WORKER-ID()
142  if $Q[p].array[k] == $ NIL
143      APPEND($Q[p].log, k$)
144      $Q[p].array[k] = $ new subbag
145  APPEND($Q[p].array[k], v$)
```

Figure 3-8: Pseudocode for the MB-INSERT multibag operation. MB-INSERT(Q, v, k) inserts the element v into the kth bag C_k of the multibag Q.

Conceptually, the MB-COLLECT operation extracts the bags in Q to produce a compact representation of those bags that can be read efficiently. Figure 3-7(b) illustrates the compact representation of the elements of the multibag from Figure 3-7(a) that MB-COLLECT returns. This representation consists of a pair $\langle$*bag-offsets*, *collected-subbags*$\rangle$ of arrays that together resemble the representation of a graph in a CSR format. The *collected-subbags* array stores all of the subbags in Q sorted by their corresponding bag's index. The *bag-offsets* array stores indices in *collected-subbags* that denote the sets of subbags comprised by each bag. In particular, in this representation, the contents of bag C_k are stored in the subbags in *collected-subbags* between indices *bag-offsets*$[k]$ and *bag-offsets*$[k+1]$.

Figure 3-9 sketches how MB-COLLECT converts a multibag Q stored in worker-local SPA's into the representation illustrated in Figure 3-7(b). Steps 1 and 2 create an array *collected-subbags* of nonempty subbags from the worker-local SPA's in Q. Each subbag $C_{k,p}$ in *collected-subbags* is tagged with the integer index k of its corresponding bag $C_k \in Q$. Step 3 sorts *collected-subbags* by these index tags, and Step 4 creates the *bag-offsets* array. Step 5 removes all elements from Q, thereby emptying the multibag.

Analysis of multibags

We now analyze the work and span of the multibag's MB-INSERT and MB-COLLECT operations, starting with MB-INSERT.

Lemma 1 *Executing* MB-INSERT *takes* $\Theta(1)$ *time in the worst case.*

PROOF. Consider each step of a call to MB-INSERT(Q, v, k). The GET-WORKER-ID procedure on line 141 obtains the executing worker's identifier p from the runtime system in $\Theta(1)$ time, and line 142 checks if the entry $Q[p].array[k]$ is empty in $\Theta(1)$ time. Suppose that $Q[p].log$ and each subbag in $Q[p].array$ are implemented as dynamic arrays that use a deamortized table-doubling scheme [36]. Lines 143–145 then take $\Theta(1)$ time each to append k to $Q[p].log$, create a new subbag in $Q[p].array[k]$, and append v to $Q[p].array[k]$. $\square$

The next lemma analyzes the work and span of MB-COLLECT.

Lemma 2 *In a P-processor parallel program execution, a call to* MB-COLLECT(Q) *on a multibag Q with χ bags whose contents are distributed across m distinct subbags executes in $O(m + \chi + P)$ work and $O(\lg m + \chi + \lg P)$ span.*

PROOF. We analyze each step of MB-COLLECT in turn. We shall use a helper procedure PREFIX-SUM(A), which computes the all-prefix sums of an array A of n integers in $\Theta(n)$ work and $\Theta(\lg n)$ span. (Blelloch [19] describes an appropriate implementation of PREFIX-SUM.) Step 1 replaces each entry in $Q[p].log$ in each worker-local SPA $Q[p]$ with the appropriate index-subbag pair $\langle k,\ C_{k,p}\rangle$ in parallel, which requires $\Theta(m + P)$ work and $\Theta(\lg m + \lg P)$ span. Step 2 gathers all index-subbag pairs into a single array. Suppose that each worker-local SPA $Q[p]$ is augmented with the size of $Q[p].log$, as Figure 3-7(a) illustrates. Executing PREFIX-SUM on these sizes and then copying the entries of $Q[p].log$ into *collected-subbags* in parallel therefore completes Step 2 in $\Theta(m + P)$ work and $\Theta(\lg m + \lg P)$ span. Step 3 can sort the *collected-subbags* array in $\Theta(m+\chi)$ work and $\Theta(\lg m+\chi)$ span using a variant of a parallel radix sort [24, 49, 203] as follows:

1. Divide *collected-subbags* into m/χ groups of size χ, and create an $(m/\chi) \times \chi$ matrix A, where entry A_{ij} stores the number of subbags with index j in group i. Constructing A can be done with $\Theta(m+\chi)$ work and $\Theta(\lg m+\chi)$ span by evaluating the groups in parallel and the subbags in each group serially.
2. Evaluate PREFIX-SUM on A^{T} (or, more precisely, the array formed by concatenating the columns of A in order) to produce a matrix B such that B_{ij} identifies which entries in the sorted version of *collected-subbags* will store the subbags with index j in group i. This PREFIX-SUM call takes $\Theta(m + \chi)$ work and $\Theta(\lg m + \lg \chi)$ span.
3. Create a temporary array T of size m, and in parallel over the groups of *collected-subbags*, serially move each subbag in the group to an appropriate index in T, as identified by B. Copying these subbags executes in $\Theta(m+\chi)$ work and $\Theta(\lg m+\chi)$ span.
4. Rename the temporary array T as *collected-subbags* in $\Theta(1)$ work and span.

```
MB-COLLECT($Q$)
  1. For each SPA $Q[p]$, map each bag index $k$ in $Q[p].log$ to the pair $\langle k,\ Q[p].array[k]\rangle$.
  2. Concatenate the arrays $Q[p].log$ for all workers $p \in \{0, 1, \ldots, P-1\}$ into a single array,
     collected-subbags.
  3. Sort the entries of collected-subbags by their bag indices.
  4. Create the array bag-offsets, where bag-offsets[$k$] stores the index of the first subbag
     in collected-subbags that contains elements of the $k$th bag.
  5. For $p = 0, 1, \ldots, P-1$, delete all elements from the SPA $Q[p]$.
  6. Return the pair $\langle$bag-offsets, collected-subbags$\rangle$.
```

Figure 3-9: Pseudocode for the MB-COLLECT multibag operation. Calling MB-COLLECT on a multibag Q produces a pair of arrays *collected-subbags*, which contains all nonempty subbags in Q sorted by their associated bag's index, and *bag-offsets*, which associates sets of subbags in Q with their corresponding bag.

Finally, Step 4 can scan *collected-subbags* for adjacent pairs of entries with different bag indices to compute *bag-offsets* in $\Theta(m)$ work and $\Theta(\lg m)$ span, and Step 5 can reset every SPA in Q in parallel using $\Theta(P)$ work and $\Theta(\lg P)$ span. Totaling the work and span of each step completes the proof. □

Remark 3 *Let Q be a multibag in a P-processor execution with m distinct subbags that represents bags whose indices lie in the range $[0, k]$. Then Q may be treated as a multibag representing k bags so that* MB-COLLECT(Q) *executes in $O(m + k + P)$ work and $O(\lg m + k + \lg P)$ span.*

Although different executions of a program can store the elements of Q in different numbers m of distinct subbags, notice that m is never more than the total number of elements in Q.

3.4 Analysis of Prism

This section analyzes the performance of PRISM using work-span analysis [52, Ch. 27]. We derive bounds on the work and span of PRISM for any simple data-graph computation $\langle G, f, Q_0\rangle$. Recall that we make the reasonable assumptions that a single update $f(v)$ executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span, and that the update only activates vertices in $N(v)$. These work and span bounds can be used to characterize the data-graph computations on which PRISM achieves good parallel scalability. In particular, we show that on a data-graph on n vertices colored using χ colors that PRISM achieves good parallel speedup whenever the average work per round is much greater than $P\ \chi \lg n$

Let us first analyze the work and span of PRISM for one round of a data-graph computation.

Theorem 4 *Suppose that* PRISM *colors a Δ-degree data graph $G = (V, E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0 \rangle$. Then, on P processors,* PRISM *executes updates on all vertices in the activation set Q_r for a round r using $O(\mathit{size}(Q_r) + P)$ work and $O(\chi(\lg(Q_r/\chi) + \lg \Delta) + \lg P)$ span.*

PROOF. Let us first analyze the work and span of one iteration of lines 126–132 in PRISM, which perform the updates on the vertices belonging to one color set $C \in Q_r$. Consider a vertex $v \in C$. Lines 128 and 129 execute in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span. For each vertex u in the set S of vertices activated by the update $f(v)$, Lemma 1 implies that lines 131–132 execute in $\Theta(1)$ total work. The **parallel for** loop on lines 130–132 therefore executes in $\Theta(S)$ work and $\Theta(\lg S)$ span. Because $|S| \leq \deg(v)$, the **parallel for** loop on lines 127–132 thus executes in $\Theta(\mathit{size}(C))$ work and $\Theta(\lg C + \max_{v \in C} \lg(\deg(v))) = O(\lg C + \lg \Delta)$ span.

By processing each of the χ color sets belonging to Q_r, lines 126–132 therefore executes in $\Theta(\mathit{size}(Q_r) + \chi)$ work and $O(\chi(\lg(Q_r/\chi) + \lg \Delta))$ span. Lemma 2 implies that line 125 executes MB-COLLECT in $O(Q_r + \chi_r + P)$ work and $O(\lg Q_r + \chi_r + \lg P)$ span where $\chi_r = \max_{v \in Q_r} \{\mathit{color}[v]\}$. Note that we take advantage here of the observation made in remark 3. The theorem follows since $|Q_r| + \chi_r \leq \mathit{size}(Q_r) + 1$ □

Theoretical scalability of PRISM

Dynamic data-graph computations typically run for multiple rounds until a convergence criteria is met. We will now generalize Theorem 4 to prove work and span bounds for PRISM when executing a sequence of rounds.

Theorem 5 *Suppose that* PRISM *colors a Δ-degree data graph $G = (V, E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0 \rangle$ in r rounds applying updates to the activation sets $Q_0, Q_1, \ldots, Q_{r-1}$. Define the multiset $\mathcal{U} = \biguplus_{i=0}^{r-1} Q_i$ so that $|\mathcal{U}| = \sum_{i=0}^{r-1} |Q_i|$ and $\mathit{size}(\mathcal{U}) = \sum_{i=0}^{r-1} \mathit{size}(Q_i)$, where the symbol $\uplus$ indicates a* ***multiset sum****.[7] Then, on P processors,* PRISM *executes the data-graph computation using $O(\mathit{size}(\mathcal{U}) + rP)$ work and $O(r\ \chi(\lg((\mathcal{U}/r)/\chi) + \lg \Delta) + r \lg P)$ span.*

[7]A multiset sum $\mathcal{M} = \biguplus_{i \in I} \mathcal{M}_i$ has multiplicity of element m equal to $\mathcal{M}(m) = \sum_{i \in I} \mathcal{M}_i(m)$ for all $m \in M$.

PROOF. The work bound follows directly from Theorem 4 by taking the sum of work performed in each of the r rounds of PRISM. The total span of PRISM is equal to the sum of each round's span which by Theorem 4 is bounded by $\sum_{i=0}^{r-1}(\chi(\lg(Q_i/\chi) + \lg\Delta) + \lg P)$. Observing that $\sum_{i=0}^{r-1}\chi\lg(Q_i/\chi) \le r\ \chi\lg((\mathcal{U}/r)/\chi)$ completes the proof. □

Given Theorem 5 we can compute the parallelism of PRISM for a data-graph computation that applies a multiset $\mathcal{U}$ of updates over r rounds. The following corollary expresses the parallelism of PRISM in terms of the average size of the activation sets in a sequence of rounds.

Corollary 6 *Suppose* PRISM *executes a data-graph computation in r rounds during which it applies a multiset $\mathcal{U}$ of updates. Define the average number of updates per round $U_{avg} = |\mathcal{U}|/r$ and the average work per round $W_{avg} = size(\mathcal{U})/r$. Then* PRISM *has $\Omega(W_{avg}/(\chi(\lg(U_{avg}/\chi) + \lg\Delta)))$ parallelism.*

PROOF. Follows from Theorem 5 by computing the parallelism as the ratio of the work and span and then performing substitution. □

Corollary 6 implies that PRISM achieves near perfect linear parallel speedup on P processors for a graph of n vertices when the average work performed in each round $W_{avg} \gg P\ \chi\lg n$.

3.5 Empirical evaluation

This section explores the performance properties of PRISM from an empirical perspective. We describe three experiments designed to investigate the synchronization costs, dynamic-scheduling overheads, and scalability properties of PRISM. For the first experiment, on a suite of 12 benchmark graphs, PRISM executed between 1.0 and 2.1 times faster than a nondeterministic locking protocol on PageRank [35], exhibiting a geometric-mean speedup of a factor of 1.5, a substantial advantage in synchronization costs. The second experiment shows that the slowdown that PRISM incurs for dynamic scheduling using multibags, compared with static scheduling, is only about 1.16 when all vertices are activated in every round. This experiment shows that PRISM can be effective even for relatively densely activated graphs. The third experiment shows that PRISM scales well and is relatively insensitive to the number of colors needed to color the data graph, as long as there is sufficient parallelism.

Experimental setup

All of the benchmarks presented in this section were run on an Intel Xeon X5650 machine with 12 processor cores running at 2.67-GHz with hyperthreading disabled. Our test machine has 49 GB of DRAM, two 12-MB L3-caches, each shared among 6 cores, and private L2- and L1-caches of sizes 128 KB and 32 KB, respectively.

As a platform for our experiments, we implemented a new parallel execution engine within GraphLab [143] that uses Intel Cilk Plus[8] [113] to expose parallelism. The new execution engine and all of our scheduling algorithms were designed to be compatible with the original GraphLab API in order to facilitate a fair evaluation of the relative merits of different scheduling methodologies. In particular, to better understand the performance properties of PRISM, we developed four scheduling algorithms for comparison:

SERIAL-DDGC — is an implementation of the serial scheduling algorithm from Figure 3-4. SERIAL-DDGC provides a serial performance baseline for measuring the parallel speedup achieved by the other, more complex, scheduling algorithms for dynamic data-graph computations.

CILK+LOCKS — is a lock-based scheduling algorithm for dynamic data-graph computations. During each round, CILK+LOCKS updates only an active subset of the vertices in the graph. It uses a locking scheme to avoid executing conflicting updates in parallel. The locking scheme associates a shared-exclusive (i.e., reader-writer) lock [54] with each vertex in the graph. Prior to executing an update $f(v)$, vertex v's lock is acquired exclusively, and a shared lock is acquired for each $u \in N(v)$. A global ordering of locks is used to avoid deadlock.

RRLOCKS — is the lock-based dynamic scheduling algorithm implemented by the round-robin ***sweep scheduler*** in the original shared-memory version of GraphLab. A bit vector *active* is used to represent the active set of vertices. During each round, RRLOCKS scans each vertex in the active set in a round-robin fashion, conditionally updating a vertex v_i if *active*$[i]$ is TRUE. To avoid races, a locking strategy is used to coordinate updates that conflict.

[8]All code was compiled with Intel's ICC version 13.1.1.

Graph	$\lvert V \rvert$	$\lvert E \rvert$	χ	Cilk+ Locks	Prism	Prism-R	*Coloring*
cage15	5,154	94,044	17	36.9	35.5	35.6	12%
liveJournal	4,847	68,475	333	36.8	21.7	22.3	12%
randLocalDim25	1,000	49,992	36	26.7	14.4	14.6	18%
randLocalDim4	1,000	41,817	47	19.5	12.5	13.7	14%
rmat2Million	2,097	39,912	72	22.5	16.6	16.8	12%
powerGraph2M	2,000	29,108	15	12.1	9.8	10.1	13%
3dgrid5m	5,000	15,000	6	10.3	10.3	10.4	7%
2dgrid5m	4,999	9,999	4	17.7	8.9	9.0	4%
web-Google	916	5,105	43	3.9	2.4	2.4	8%
web-BerkStan	685	7,600	200	3.9	2.4	2.7	8%
web-Stanford	281	2,312	62	1.9	0.9	1.0	11%
web-NotreDame	325	1,469	154	1.1	0.8	0.8	12%

Table 3-10: Performance of Prism versus Cilk+Locks when executing $10 \cdot |V|$ updates of the PageRank [35] data-graph computation on a suite of six real-world graphs and six synthetic graphs. Column "*Graph*" identifies the input graph, and columns $|V|$ and $|E|$ specify the number of vertices and edges in the graph in thousands, respectively. Column χ gives the number of colors Prism used to color the graph. Columns "Cilk+Locks," "Prism," and "Prism-R" present 12-core running times in seconds for each respective scheduler. Each running time is the median of 5 runs. Column "*Coloring*" gives the percentage of Prism's running time spent coloring the graph. Prism-R, discussed in Section 3.6, provides deterministic support for associative operations on global variables.

RRColor — is a coloring-based dynamic scheduling algorithm that uses a bit vector *active* to represent the active set of vertices. Instead of using locks to coordinate conflicting updates, however, RRColor uses a vertex-coloring of the graph. At the start of the computation, RRColor partitions the vertices by color and stores them in static arrays. For a graph colored using χ colors, each round of the computation is divided into χ color steps. During the kth color step, RRColor scans all color-k vertices and conditionally updates a color-kd vertex v_i if *active*$[i]$ is TRUE.

Overheads for locking and for chromatic scheduling

We compared the overheads associated with coordinating conflicting updates of a dynamic data-graph computation using locks versus using chromatic scheduling. We evaluated these overheads by comparing the 12-core execution times for Prism and Cilk+Locks to execute the PageRank [35] data-graph computation on a suite of graphs. We used PageRank for this study because of its comparatively cheap update function, which makes overheads due to scheduling more pronounced. PageRank updates a vertex v by first scanning v's incoming edges to aggregate the data from its incoming neighbors, and then by scanning v's outgoing edges to activate its outgoing neighbors.

We executed the PageRank application on a suite of six synthetic and six real-world graphs. The six real-world graphs came from the Stanford Large Network Dataset Collection (SNAP) [136], and the University of Florida Sparse Matrix Collection [57]. The six synthetic graphs were generated using the "randLocal," "powerLaw," "gridGraph," and "rMatGraph" generators included in the Problem Based Benchmark Suite [179]. We chose the graphs in this suite to be large enough to stress the memory system and thus make parallel speedup comparatively difficult. That is, given the random access inherent in data-graph computations, we expect most references to vertex data to come from DRAM, making DRAM bandwidth a scarce shared commodity. Since the span of PRISM is superconstant, however, for a fixed number of workers, increasing the size of the graph only increases parallelism, making good parallel speedup comparatively easy. Thus, we have pessimistically chosen the graphs in the suite to be large enough to make DRAM bandwidth a shared bottleneck but not unduly larger.

We observed that PRISM often performs slightly fewer rounds of updates than CILK+LOCKS when both are allowed to run until convergence. Wishing to isolate scheduling overheads, we controlled this variation by explicitly setting the total number of updates on a graph to 10 times the number of vertices.

Table 3-10 presents the empirical results for this study. Table 3-10 shows that over the 12 benchmark graphs, PRISM executes between 1.0 and 2.1 times faster than CILK+LOCKS on PageRank, exhibiting a geometric-mean speedup of a factor of 1.5. Moreover, from Table 3-10 we see that an average of 10.9% of PRISM's total running time is spent coloring the data graph, which is approximately equal to the cost of executing $|V|$ updates. PRISM colors the data-graph once to execute the data-graph computation, however, meaning that its cost can be amortized over all of the updates in the data-graph computation. By contrast, the locking scheme implemented by CILK+LOCKS incurs overhead for every update. Before updating a vertex v, CILK+LOCKS acquires each lock associated with v and every vertex $u \in N(v)$. For simple data-graph computations whose update functions perform relatively little work, this step can account for a significant fraction of the time to execute an update.

Benchmark	χ	*Updates*	RRLocks	RRColor	Prism	Prism-R
PR/L	333	48,475K	35.25	14.5	17.7	18.4
ID/2000	4	40,000K	63.15	50.1	59.2	59.9
FBP/C3	2	16,001K	11.9	8.8	8.8	8.9
ID/1000	4	10,000K	15.7	12.6	14.9	15.0
PR/G	43	9,164K	3.1	1.3	2.1	2.2
FBP/C1	2	8,783K	5.9	4.7	4.8	4.8
ALS/N	6	1,877K	65.7	52.4	52.8	53.5

Table 3-11: Performance of three schedulers on the seven application benchmarks from Table 3-5, modified so that all vertices are activated in every round. Column "*Updates*" specifies the number of updates performed in the data-graph computation. Columns "RRLocks," "RRColor," "Prism," and "Prism-R" list the 12-core running times in seconds for the respective schedulers to execute each benchmark. Each running time is the median of 5 runs. The Prism-R algorithm, which provides deterministic support for associative operations on global variables, will be discussed in Section 3.6.

Dynamic-scheduling overhead

To investigate the overhead of using multibags to maintain activation sets, we compared the 12-core running times of Prism, RRColor, and RRLocks on the seven benchmark applications from Table 3-5. For this study, we modified the benchmarks slightly for each scheduler in order to provide a fair comparison. First, because Prism typically executes fewer updates than a static data-graph computation scheduler, we modified the update functions for each application so that every update on a vertex v always activates all vertices $u \in N(v)$. This modification guarantees that Prism executes the same set of updates each round as RRLocks and RRColor, while still incurring the overhead that Prism requires in order to maintain a dynamic set of active vertices. Thus, we compare the worst case conditions for Prism with respect to scheduling overhead with the best case conditions for RRLocks and RRColor.

Table 3-11 presents the results of these tests, revealing that the overhead Prism incurs to maintain its activation sets using a multibag. As can be seen from the figure, Prism is 1.0 to 1.6 times slower than RRColor on the benchmarks with a geometric-mean relative slowdown of 1.16. That is, for static data-graph computations, Prism incurs only an aggregate 16% slowdown through the use of a multibag, as opposed to the simple array used by RRColor, which suffices for static scheduling. The Prism algorithm, which can also support *dynamic* activation sets efficiently, incurred minimal overhead for the multibag data structure. Prism outperformed RRLocks on all benchmarks, achieving a geometric-mean speedup of 30% due to RRLocks's lock overhead. Thus, Prism incurs relatively little overhead by maintaining activation sets with multibags.

The relative overhead of RRColor and Prism depends on the percentage of vertices active during a given round. As a typical example, RRColor is approximately 1.09 times faster than Prism on the image denoise benchmark when 80% of the vertices are active each round, but is 1.11 times slower when 5% or less of the vertices are active each round. As part of an effort to incorporate the Prism scheduling paradigm into an existing data-graph computation framework (e.g., GraphLab, Pregel, etc.), one might consider using a heuristic to switch between the use of a bitvector and a multibag depending on the density of the activation set. A simple heuristic such as a fixed threshold on the relative density of the activation set (e.g., 10% of the vertices) would likely suffice to maintain activation sets with good performance: if fewer than 10% of vertices are active, use a multibag, otherwise use a bitvector.

Scalability of Prism

To measure the scalability of Prism, and Cilk+Locks, we compared their 12-core runtimes to the serial reference implementation Serial-DDGC. Figure 3-12 shows the empirical 12-core speedups relative to Serial-DDGC of Prism and Cilk+Locks on seven application benchmarks. (Data for Prism-R is also included, which will be discussed in Section 3.8. In geometric mean, Cilk+Locks achieved 5.73 times speedup, Prism achieved 7.56 times speedup, and Prism-R achieved 7.42 times speedup.

In order to study the effect of the number χ of colors used to color the application's data graph on the parallel scalability of Prism, we measured the parallelism T_1/T_∞ and the 12-core speedup T_1/T_{12} of Prism while executing the image-denoise application as we varied the number of colors used. The image-denoise application performs belief propagation to remove Gaussian noise added to a gray-scale image. The data graph for the image-denoise application is a two-dimensional grid in which each vertex represents a pixel, and there is an edge between any two adjacent pixels. The Color-Graph procedure invoked in line 121 of Figure 3-6 typically colors this data-graph with just 4 colors.

To perform this study, we artificially increased χ by repeatedly taking a random nonempty subset of the largest set of vertices with the same color and assigning a new color to those vertices. Using this technique, we ran the image-denoise application on a

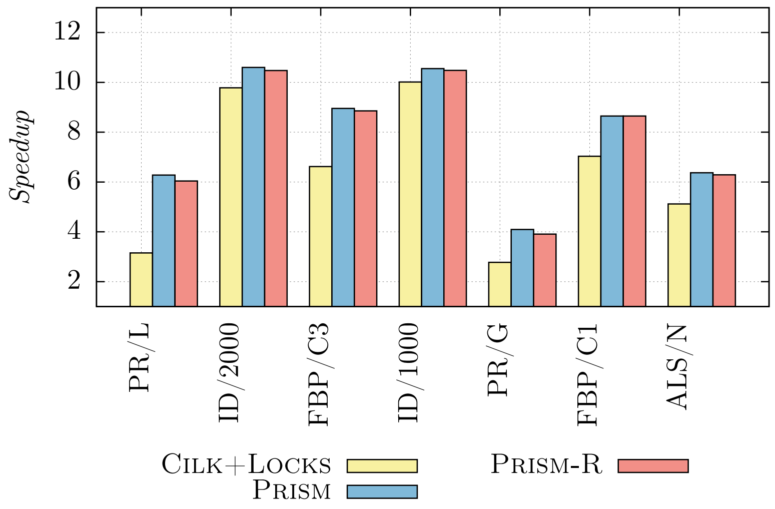

Figure 3-12: Empirical speedup relative to Serial-DDGC on 12 processor cores. Shown are the empirical speedups T_s/T_{12} of Cilk+Locks, Prism, and Prism-R, where T_s is the runtime of the serial scheduling algorithm Serial-DDGC and T_{12} is the runtime of the particular algorithm on 12 cores. The Prism-R algorithm is discussed in Section 3.6.

500-by-500 pixel input image for values of χ between 4 and 250,000, the last data point corresponding to a coloring that assigns all pixels distinct colors. Figure 3-13 plots the results of these tests. Although the parallelism of Prism is inversely proportional to χ, Prism's speedup on 12 cores is relatively insensitive to χ, as long as the parallelism is greater than about 120. This result is consistent with the rule of thumb that a program with at least $10P$ parallelism should achieve nearly perfect linear speedup on P processors [52, p. 783].

3.6 The Prism-R Algorithm

This section introduces Prism-R, a chromatic-scheduling algorithm that executes a dynamic data-graph computation deterministically even when updates modify global ***reducer variables*** using associative operations such as a reducer hyperobject [76]. While the chromatic scheduling technique employed by Prism ensures that there are no data races on the vertex data of the graph, the order in which updates are made to a reducer variable among vertices of a common color can yield a nondeterministic result to the final reducer variable value. The multivector data structure, which is a theoretical improvement to the multibag, is used by Prism-R to maintain activation sets that are partitioned by color and ordered deterministically. We describe an extension of the model of simple data-graph computations

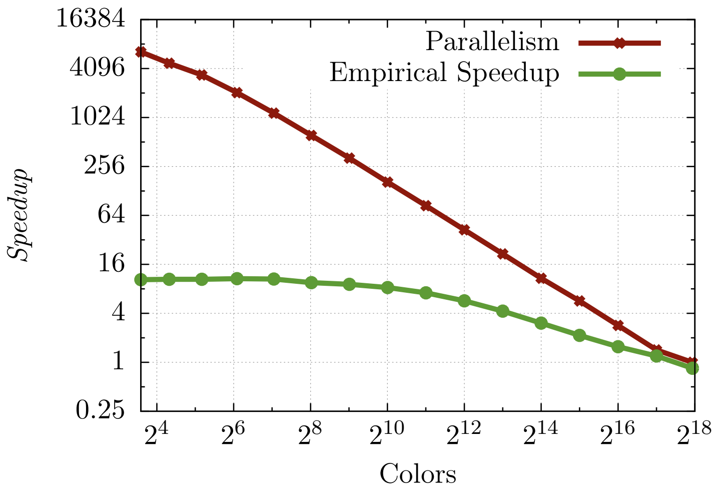

Figure 3-13: Scalability of PRISM on the image-denoise application as a function of χ, the number of colors used to color the data graph. The parallelism T_1/T_∞ is plotted together with the empirical speedup T_1/T_{12} achieved on a 12-core execution. Parallelism values were measured using the Cilkview scalability analyzer [103].

that permits an update function to perform associative operations on global variables using a parallel reduction mechanism. In this extended model, PRISM-R executes dynamic data-graph computations deterministically while achieving the same work and span bounds as PRISM.

Data-graph computations with global reductions

Several frameworks for executing data-graph computations allow updates to modify global variables in limited ways. Pregel aggregators [145], and GraphLab's sync mechanism [143], for example, both support data-graph computations in which an update can modify a global variable in a restricted manner. These mechanisms coordinate parallel modifications to a global variable using ***parallel reductions*** [20, 45, 112, 116, 125, 130, 149, 169], that is, they coordinate these modifications by applying them to local ***views*** (copies) of the variable and then ***reducing*** (combining) those copies together using a binary ***reduction operator***.

A ***reducer (hyperobject)*** [76, 133] is a general parallel reduction mechanism provided by Cilk Plus and other dialects of Cilk. A reducer is defined on an arbitrary data type T, called a ***view type***, by defining an IDENTITY operator and a binary REDUCE operator for views of type T. The IDENTITY operator creates a new view of the reducer. The binary REDUCE operator defines the reducer's reduction operator. A reducer is a particularly general

```
PRISM-R($G, f, Q_0$)
146  $\chi$ = COLOR-GRAPH($G$)
147  $r = 0$
148  $updates = 0$
149  $Q = Q_0$
150  while $Q \neq \emptyset$
151      $\mathcal{C}$ = MV-COLLECT($Q$)
152      for $C \in \mathcal{C}$
153          parallel for $i = 1, 2, \ldots, |C|$
154              $\langle v, p \rangle = C[i]$
155              if $p == priority[v]$
156                  $rank[f(v)] = updates + i$
157                  $priority[v] = \infty$
158                  $S = f(v)$
159                  parallel for $u \in S$
160                      if PRIORITYWRITE($priority[u], rank[f(v)]$)
161                          MV-INSERT($Q, \langle u, rank[f(v)] \rangle, color[u]$)
162              $updates = updates + |C|$
163          $r = r + 1$
```

```
PRIORITYWRITE($current, value$)
164  begin atomic
165  if $current > value$
166      $current = value$
167      return TRUE
168  else
169      return FALSE
170  end atomic
```

Figure 3-14: Pseudocode for PRISM-R. The algorithm takes as input a data graph G, an update function f, and an initial activation set Q_0. COLOR-GRAPH colors a given graph and returns the number of colors it used. The procedures MV-COLLECT and MV-INSERT operate the multivector Q to maintain activation sets for PRISM-R. PRISM-R updates the value of *updates* after processing each color set and r after each round of the data-graph computation.

reduction mechanism because it guarantees that, if its REDUCE operator is associative, then the final result in the global variable is deterministic: every parallel execution of the program produces the same result. Other parallel reduction mechanisms, including Pregel aggregators and GraphLab's sync mechanism, provide this guarantee only if the reduction operator is also commutative.

Although PRISM is implemented in Cilk Plus, PRISM does not produce a deterministic result if updates modify global variables using a noncommutative reducer. The reason for this is, in part, that the order of vertices within in a multibag depends on how the computation was scheduled among participating workers. As a result, the order in which lines 127–132 of PRISM in Figure 3-6 evaluates the vertices in a color set C is nondeterministic. If two updates on vertices in C modify the same reducer, then the relative order of these modifications can differ between runs of PRISM, even if a single worker happens to execute both updates.

PRISM-R

PRISM-R is an extension to PRISM that executes dynamic data-graph computations deterministically even when update functions are allowed to perform associative operations on global variables. The semantics of PRISM-R mimic that of SERIAL-DDGC when its queue of active vertices is stable sorted by color at the start of each round. In this modified version of SERIAL-DDGC updates to active vertices of the same color are applied in increasing order of their insertion into the queue. PRISM-R guarantees that the result of associative reductions performed by update functions reflect this same order.

Figure 3-14 shows the pseudocode for PRISM-R which differs from PRISM in its use of alternate data structure to maintain partitioned activation sets and in its use of a priority deduplication strategy for avoiding multiple updates to the same vertex in a round.

A ***multivector*** is used by PRISM-R to represent a list of χ ***vectors*** (ordered multisets). It supports the operations MV-INSERT and MV-COLLECT, which are analogous to the multibag operations MB-INSERT and MB-COLLECT, respectively. Each vector maintained by a multivector has serial semantics, meaning that the order of elements within each vector is deterministic and equivalent to the insertion order in an execution of the serial elision of the parallel program. Section 3.7 describes and analyzes the implementation of the multivector data structure.

The serial semantics of the multivector are not alone sufficient to ensure that updates are ordered deterministically in an execution of the serial elision of the program. Consider, for example, a round of PRISM that updates the three vertices x, y, z in parallel. Suppose that y activates u and both x and z activate a common neighbor v. The atomic compare-and-swap operator used by PRISM on line 131 of Figure 3-6 ensures that x and z will not both insert v into the activation set, but which of the two succeeds is nondeterministic. Inserting these two activated vertices into a multivector would produce either the order u, v or v, u depending on whether x or z activated v.

To eliminate this source of nondeterminism, PRISM-R assigns each update $f(v)$ a unique integer $\mathit{rank}[f(v)]$ on line 156 of Figure 3-14 that orders updates applied during a round according to their execution order in an execution of the serial elision of PRISM-R. Instead of maintaining a bit vector denoting whether or not a vertex is active PRISM-R maintains an integer array *priority* of priorities. For each active vertex v the value $\mathit{priority}[v]$ is equal to the smallest rank of any update $f(u)$ that activated v in the previous round. The priority of a vertex v is reset on line 157 before applying $f(v)$ by setting $\mathit{priority}[v] = \infty$.

For each vertex $u \in N(v)$ activated by update $f(v)$, PRISM-R uses an atomic ***priority-write*** operator [178] to set $\mathit{priority}[u] = \min\{\mathit{priority}[u], \mathit{rank}[f(v)]\}$ and inserts the vertex-priority pair $\langle u, \mathit{rank}[f(v)]\rangle$ into the multivector if the priority write is successful on line 160. The color sets returned by MV-COLLECT on line 151 can contain multiple vertex-priority pairs for each active vertex. On lines 153–161 PRISM-R iterates over the vertex-priority pairs $\langle v, p\rangle$ in a color set and only applies the update $f(v)$ if $\mathit{priority}[v] == p$. Since $\mathit{priority}[v]$ is equal to the lowest ranked update that activated v, PRISM-R updates each active vertex exactly once during a round in the same order as a serial execution.

3.7 The multivector data structure

This section introduces the multivector data structure, which provides a theoretical improvement to the multibag. The multivector data structure maintains several vectors (dynamic arrays), each supporting a parallel append operation. Each vector has serial semantics, that is, the order of elements within any vector is equivalent to their insertion order in an execution of the serial elision of the Cilk parallel program. The multivector can be used in place of the multibag to provide a stronger encapsulation of nondeterminism in programs whose behavior depends on the ordering of elements in each set. This section assumes familiarity with the Cilk execution model [77], as well as its implementation of reducers [76].

A ***multivector*** represents a list of χ ***vectors*** (ordered multisets). It supports the operations MV-INSERT and MV-COLLECT, which are analogous to the multibag operations MB-INSERT and MB-COLLECT, respectively. Our implementation relies on properties of a work-stealing runtime system. Consider a parallel program modeled by a computation dag A in the Cilk model of multithreading. The ***serial execution order*** $R(A)$ of the program lists the vertices of A according to the order they would be visited if an execution of the serial elision of the underlying Cilk program were executed, which corresponds to a left-to-right depth-first execution of the dag.

A work-stealing scheduler partitions $R(A)$ into a sequence $R(A) = \langle t_0, t_1, \ldots, t_{M-1}\rangle$, where each ***trace*** $t_i \in R(A)$ is a contiguous subsequence of $R(A)$ executed by exactly one worker. A multivector represents each vector as a sequence of ***trace-local subvectors*** — subvectors that are modified within exactly one trace. The ordering properties of traces imply that concatenating a vector's trace-local subvectors in order produces a vector whose

```
FLATTEN($L, A, i$)
$A[i] = L$
if $L.left \neq$ NIL
    spawn FLATTEN($L.left, A, i - L.right.size - 1$)
if $L.right \neq$ NIL
    FLATTEN($L.right, A, i - 1$)
sync
```

Figure 3-15: Pseudocode for the FLATTEN operation for a log tree. FLATTEN performs a post-order parallel traversal of a log tree to place its nodes into a contiguous array.

```
IDENTITY()
$L =$ new log-tree node
$L.sublog =$ new vector
$L.size = 1$
$L.left =$ NIL
$L.right =$ NIL
return $L$
```

```
REDUCE($L_l, L_r$)
$L =$ IDENTITY()
$L.size = L_l.size + L_r.size + 1$
$L.left = L_l$
$L.right = L_r$
return $L$
```

Figure 3-16: Pseudocode for the IDENTITY and REDUCE log-tree reducer operations. The IDENTITY operation creates and returns a new log-tree node L. The REDUCE(L_l, L_r) operation concatenates a left log-tree node L_l with a right log-tree node L_r.

elements appear in the serial execution order. The multivector data structure assumes that a worker can query the runtime system to determine when it starts executing a new trace.

The log-tree reducer

A multivector stores its nonempty trace-local subvectors in a ***log tree***, which represents an ordered multiset of elements and supports $\Theta(1)$-work append operations. A log tree is a binary tree in which each node L stores a dynamic array $L.sublog$. The ordered multiset that a log tree represents corresponds to a concatenation of the tree's dynamic arrays in a post-order tree traversal. Each log-tree node L is augmented with the size of its subtree $L.size$ counting the number of log-tree nodes in the subtree rooted at L. Using this augmentation, the operation FLATTEN($L, A, L.size - 1$) described in Figure 3-15 flattens a log tree rooted at L of n nodes and height h into a contiguous array A using $\Theta(n)$ work and $\Theta(h)$ span.

To handle parallel MV-INSERT operations, a multivector employs a ***log-tree reducer***, that is, a Cilk Plus reducer whose view type is a log tree. Figure 3-16 presents the pseudocode for the IDENTITY and REDUCE operations for the log-tree reducer.

```
A(R)
188  Log-Insert(R, e_1)
189  spawn B(R)
190  Log-Insert(R, e_7)
191  sync
192  Log-Insert(R, e_8)

B(R)
193  Log-Insert(R, e_2)
194  spawn Log-Insert(R, e_3)
195  Log-Insert(R, e_4)
196  Log-Insert(R, e_5)
197  sync
198  Log-Insert(R, e_6)

Log-Insert(R, e)
199  L = Get-Local-View(R)
200  Append(L.subblog, e)
```

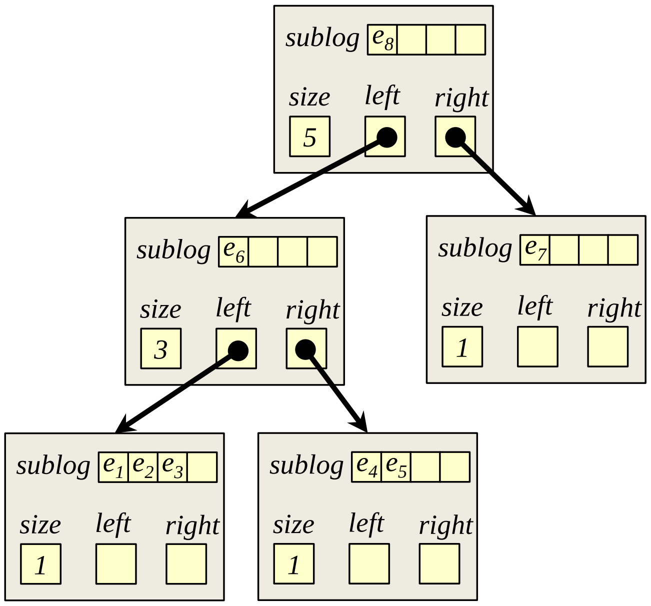

Figure 3-17: The state of a log-tree reducer R after a work-stealing execution of A(R). Steals occur on line 189 of A and line 195 of B partitioning the execution into 5 traces. The ordered multiset $(e_1, e_2, \ldots, e_8)$ is represented by 5 trace-local sublogs ordered according to a post-order traversal of the log tree.

The Identity operation creates a new log-tree node with an empty sublog. The Reduce(L_l, L_r) operation creates a new root node L and assigns $L.left = L_l$ and $L.right = L_r$. Updates are performed using a log-tree reducer R by first obtaining a local view L of the log-tree reducer using a runtime-provided function Get-Local-View(R) and appending elements to $L.sublog$. A log tree's Flatten operation uses a post-order traversal to order the log tree's nodes, which results in an ordering identical to that which would be obtained by using a linked-list reducer in place of the log-tree reducer.

The log-tree reducer's Reduce operation is logically associative, that is, for any three log-tree reducer views a, b, and c, the views produced by Reduce(Reduce(a, b), c) and Reduce(a, Reduce(b, c)) represent the same ordered multiset.

Figure 3-17 illustrates the state of a log-tree reducer R following the execution of a fork-join parallel function $A(R)$. Steals occur on line 189 of A and line 195 of B. The log-tree reducer partitions this execution of $A(R)$ into 5 traces each of which corresponds to one node in the tree. The first trace corresponds to the worker that begins the execution of $A(R)$ and each steal creates two additional traces: one corresponding to the stolen continuation of the spawned function, and another corresponding to the portion of the program following the associated **sync** statement.

To maintain trace-local subvectors, a multivector Q consists of an array of P worker-local SPA's, where P is the number of processors executing the computation, and a log-tree reducer. The SPA $Q[p]$ for worker p stores the trace-local subvectors that worker p has appended since the start of its current trace. The log-tree reducer $Q.\textit{log-reducer}$ stores all nonempty subvectors created.

Let us see how MV-Insert and MV-Collect are implemented.

Figure 3-19 sketches the MV-Insert(Q, v, k) operation to insert element v into the vector $C_k \in Q$. MV-Insert differs from MB-Insert in two ways. First, when a new subvector is created and added to a SPA, lines 206–207 additionally append that subvector to $Q.\textit{log-reducer}$, thereby maintaining the log-tree reducer. Second, lines 202–203 reset the contents of the SPA $Q[p]$ after worker p begins executing a new trace, thereby ensuring that $Q[p]$ stores only trace-local subvectors.

```
MV-Collect($Q$)
  1. Flatten the log-reducer tree so that all subvectors in the log appear in a contiguous
     array collected-subvectors.
  2. Sort the subvectors in collected-subvectors by their vector indices using a stable sort.
  3. Create the array vector-offsets, where vector-offsets[$k$] stores the index of the first
     subvector in collected-subvectors that contains elements of the vector $C_k \in Q$.
  4. Reset $Q$.log-reducer, and for $p = 0, 1, \ldots, P - 1$, reset $Q[p]$.
  5. Return the pair ⟨vector-offsets, collected-subvectors⟩.
```

Figure 3-18: Pseudocode for the MV-Collect multivector operation. Calling MV-Collect on a multivector Q produces a pair ⟨*vector-offsets*, *collected-subvectors*⟩ of arrays, where *collected-subvectors* contains all nonempty subvectors in Q sorted by their associated vector's color, and *vector-offsets* associates sets of subvectors in Q with their corresponding vector.

Figure 3-18 sketches the details of the MV-Collect operation, which returns a pair ⟨*subvector-offsets*, *collected-subvectors*⟩ analogous to the return value of MB-Collect. The procedure MV-Collect differs from MB-Collect primarily in that Step 1, which replaces Steps 1 and 2 in MB-Collect, flattens the log tree underlying $Q.\textit{log-reducer}$ to produce the unsorted array *collected-subvectors*. MV-Collect also requires that *collected-subvectors* be sorted using a stable sort on Step 2. The integer sort described in the proof of Lemma 2 for MB-Collect is a suitable stable sort for this purpose.

```
MV-INSERT(Q, v, k)
201 p = GET-WORKER-ID()
202 if worker p began a new trace since last insert
203     reset Q[p]
204 if Q[p].array[k] == NIL
205     Q[p].array[k] = new subvector
206     L = GET-LOCAL-VIEW(Q.log-reducer)
207     APPEND(L.sublog, Q[p].array[k])
208 APPEND(Q[p].array[k], v)
```

Figure 3-19: Pseudocode for the MV-INSERT multivector operation. MV-INSERT(Q, v, k) inserts an element v into the kth vector C_k maintained by the multivector Q.

Analysis of multivector operations

We now analyze the work and span of the MV-INSERT and MV-COLLECT operations, starting with MV-INSERT.

Lemma 7 *Executing* MV-INSERT *takes* $\Theta(1)$ *time in the worst case.*

PROOF. Resetting the SPA $Q[p]$ on line 203 can be done in $\Theta(1)$ worst-case time with an appropriate SPA implementation, and appending a new subvector to a log tree takes $\Theta(1)$ time. The theorem thus follows from the analysis of MB-INSERT in Lemma 1. □

Lemma 8 bounds the work and span of MV-COLLECT.

Lemma 8 *Consider a computation A with span $T_\infty(A)$, and suppose that the contents of a multivector Q of χ vectors are distributed across m subvectors. Then a call to* MV-COLLECT(Q) *incurs $\Theta(m + \chi)$ work and $\Theta(\lg m + \chi + T_\infty(A))$ span.*

PROOF. Flattening the log-tree reducer in Step 1 is accomplished in two steps. First, the FLATTEN operation writes the nodes of the log tree to a contiguous array. Execution of FLATTEN has span proportional to the depth of the log tree, which is bounded by $O(T_\infty(A))$, since at most $O(T_\infty(A))$ reduction operations can occur along any path in A, and REDUCE for log trees executes in $\Theta(1)$ work [76]. Second, using a parallel-prefix sum computation, the log entries associated with each node in the log tree can be packed into a contiguous array, incurring $\Theta(m)$ work and $\Theta(\lg m)$ span. Step 1 thus incurs $\Theta(m)$ work and $O(\lg m + T_\infty(A))$ span. The remaining steps of MV-COLLECT, which are analogous to those of MB-COLLECT and analyzed in Lemma 2, execute in $\Theta(\chi + \lg m)$ span. □

3.8 Analysis and Evaluation of Prism-R

This section presents a theoretical work-span analysis of PRISM-R, demonstrating that its work and span are asymptotically equivalent to PRISM. This section also discusses PRISM-R's empirical performance relative to PRISM, which was evaluated in Section 3.5. In particular, PRISM-R is only 2-7% slower than PRISM, overall, while providing deterministic support for associative operations on global variables.

Work-span analysis of Prism-R

We begin by analyzing the work and span of PRISM-R for simple data-graph computations that perform associative operations on global variables. In this extended model, PRISM-R executes dynamic data-graph computations deterministically while achieving the same work and span bounds as PRISM.

Theorem 9 *Let G be a Δ-degree data graph. Suppose that* PRISM-R *colors G using χ colors. Then* PRISM-R *executes updates on all vertices in the activation set Q_r for a round r of a simple data-graph computation $\langle G, f, Q_0 \rangle$ in $O(\mathit{size}(Q_r))$ work and $O(\chi(\lg(Q_r/\chi) + \lg \Delta))$ span.*

PROOF. PRISM-R can perform a priority write to its *active* array with $\Theta(1)$ work, and it can remove duplicates from the output of MV-COLLECT in $O(\mathit{size}(Q_r))$ work and $O(\lg(\mathit{size}(Q_r))) = O(\lg Q_r + \lg \Delta)$ span. The theorem follows by applying Lemmas 7 and 8 appropriately to the analysis of PRISM in Theorem 4. □

Theorem 10 *Suppose that* PRISM-R *colors a Δ-degree data graph $G = (V, E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0 \rangle$ in r rounds applying updates to the activation sets $Q_0, Q_1, \ldots, Q_{r-1}$. Define the multiset $\mathcal{U} = \biguplus_{i=0}^{r-1} Q_i$ so that $|\mathcal{U}| = \sum_{i=0}^{r-1} |Q_i|$ and $\mathit{size}(\mathcal{U}) = \sum_{i=0}^{r-1} \mathit{size}(Q_i)$. Then* PRISM-R *executes the data-graph computation using $O(\mathit{size}(\mathcal{U}))$ work and $O(r \cdot \chi(\lg((\mathcal{U}/r)/\chi) + \lg \Delta))$ span.*

PROOF. By Theorem 9 PRISM-R executes a round of a data-graph computation using the same asymptotic work and span as PRISM. We mirror the arguments in Theorem 5 to bound the work and span of PRISM-R for a sequence of rounds. □

Given Theorem 10 we can compute the parallelism of PRISM-R for a data-graph computation that applies a multiset $\mathcal{U}$ of updates over r rounds. The following corollary expresses the parallelism of PRISM-R in terms of the average size of the activation sets in a sequence of rounds.

Corollary 11 *Suppose* PRISM-R *executes a data-graph computation in r rounds during which it applies a multiset $\mathcal{U}$ of updates. Define the average number of updates per round $U_{avg} = |\mathcal{U}|/r$ and the average work per round $W_{avg} = size(\mathcal{U})/r$. Then* PRISM-R *has $\Omega(W_{avg}/(\chi(\lg(U_{avg}/\chi) + \lg \Delta)))$ parallelism.*

PROOF. Follows from Theorem 10 by computing the parallelism as the ratio of the work and span and then performing substitution. □

Empirical evaluation of PRISM-R

PRISM-R provides deterministic support for associative operations on global variables at the cost of additional complexity versus PRISM, specifically in the maintenance of activation sets. Nonetheless, PRISM-R guarantees the same asymptotic work and span as PRISM. Empirically, we find that PRISM-R suffers a geomean slowdown of only 2–7% versus PRISM in various scenarios. In particular, the 12-core performance for each dynamic data-graph computation application featured in Table 3-5 demonstrate that for real-world applications PRISM-R is 7% slower in geometric mean than PRISM. in Table 3-11 we see that PRISM-R is only 1.8% slower than PRISM for static versions of the applications featured in Table 3-5 (i.e., all vertices are updated every round). Finally, in Table 3-10 we present the 12-core performance of PRISM-R on PageRank [35] for a suite of six synthetic and six real-world graphs. In this case, PRISM-R is 3.5% slower in geometric mean than PRISM.

3.9 Conclusion

Researchers over multiple decades have soberly advised the rest of the community that the difficulty of parallel programming can be greatly reduced by using some form of deterministic parallelism [15,16,21,28,62,63,71,72,82,100,110,161,164,185,200]. With a deterministic parallel program, the programmer observes no logical concurrency, that is, no nondeterminacy in the behavior of the program due to the relative and nondeterministic timing of communicating processes (e.g., when two processes try to acquire a lock simultaneously). The semantics of a *deterministic* parallel program are therefore serial and reasoning about such a program's correctness is theoretically no harder than reasoning about the correctness of a serial program, which is already sufficiently hard for most people. Testing, debugging, and formal verification is simplified by determinism, because there is no need to consider all possible relative timings (i.e., interleavings) of operations on shared mutable state.

The behavior of PRISM corresponds to a variant of SERIAL-DDGC that sorts the activated vertices in its queue by color at the start of each round. Whether PRISM executes a given data graph on 1 processor or many, it always behaves the same way. With PRISM-R, this property holds even when the update function can perform reductions (e.g., associative operators on global variables). By contrast, lock-based schedulers provide no such a guarantee of determinism. Instead, updates in a round executed by a lock-based scheduler appear to execute according to some linear order, the so-called *sequential consistency* model employed by GraphLab [142, 143] and others. This order is nondeterministic due to races on the acquisition of locks.

Blelloch, Fineman, Gibbons, and Shun [22] argue that deterministic programs can be fast compared with nondeterministic programs, and they document many examples where the overhead for converting a nondeterministic program into a deterministic one is small. They even document a few cases where this "price of determinism" is *slightly* negative. To their list, we add the execution of dynamic data-graph computations as having a price of determinism which is *significantly* negative. We leave as an open problem how one might support dynamic data-graph computations deterministically in the face of dynamically changing graphs, while maintaining time and space bounds similar to PRISM. We also put forward a related open problem: how might high quality colorings be maintained efficiently in the face of dynamically changing graphs?

Chapter 4

Ordering Heuristics for Parallel Graph Coloring

4.1 Introduction

A ***(vertex)-coloring*** of an undirected graph $G = (V, E)$ is an assignment of a ***color*** $v.color$ to each vertex $v \in V$ such that for every edge $(u, v) \in E$, we have $u.color \neq v.color$, that is, no two adjacent vertices have the same color. We were motivated to work on graph coloring in the context of "chromatic scheduling" [2, 18, 121] of parallel "data-graph computations." A ***data graph*** is a graph with data associated with its vertices and edges. A ***data-graph computation*** is an algorithm implemented as a sequence of "updates" on the vertices of a data graph $G = (V, E)$, where ***updating*** a vertex $v \in V$ involves computing a new value associated with v as a function of v's old value and the values associated with the ***neighbors*** of v: the set of vertices adjacent to v in G, denoted $N(v) = \{u \in V : (v, u) \in E\}$. To ensure atomicity of each update, rather than using mutual-exclusion locks or other nondeterministic means of data synchronization, chromatic scheduling first colors the vertices of G and then sequences through the colors, scheduling all vertices of the same color in parallel. The time to perform a data-graph computation thus depends both on how long it takes to color G and on the number of colors produced by the graph-coloring algorithm: more colors means less parallelism. Although the coloring can be performed offline for some data-graph computations, for other computations the coloring must be produced online, and one must accept a trade-off between coloring ***quality*** — number of colors — and the time to produce the coloring.

Although the problem of finding an ***optimal*** coloring of a graph — a coloring using the fewest colors possible — is in NP-complete [79], heuristic “greedy” algorithms work reasonably well in practice. Welsh and Powell [197] introduced the original ***greedy*** coloring algorithm, which iterates over the vertices and assigns each vertex the smallest color not assigned to a neighbor. For a graph $G = (V, E)$, define the ***degree*** of a vertex $v \in V$ by $\deg(v) = |N(v)|$, the number of neighbors of v, and let the ***degree*** of G be $\Delta = \max_{v \in V} \{\deg(v)\}$. Welsh and Powell show that the greedy algorithm colors a graph G with degree Δ using at most $\Delta + 1$ colors.

Ordering heuristics

```
Greedy(G)
let $G = (V, E, \rho)$
for $v \in V$ in order of decreasing $\rho(v)$
    $C = \{1, 2, \ldots, \deg(v) + 1\}$
    for $u \in N(v)$ such that $\rho(u) > \rho(v)$
        $C = C - \{u.color\}$
    $v.color = \min C$
```

Figure 4-1: Pseudocode for a serial greedy graph-coloring algorithm. Given a vertex-weighted graph $G = (V, E, \rho)$, where the priority of a vertex $v \in V$ is given by $\rho(v)$, Greedy colors each vertex $v \in V$ in decreasing order according to $\rho(v)$.

In practice, however, greedy coloring algorithms tend to produce much better colorings than the $\Delta + 1$ bound implies, and moreover, the order in which a greedy coloring algorithm colors the vertices affects the quality of the coloring.[1] To reduce the number of colors a greedy coloring algorithm uses, practitioners therefore employ ***ordering heuristics*** to determine the order in which the algorithm colors the vertices [4, 32, 118, 147].

The literature includes many studies of ordering heuristics and how they affect running time and coloring quality. Here are six of the more popular heuristics:

[1] In fact, for any graph $G = (V, E)$, some ordering of V causes a greedy algorithm to color G optimally, although finding such an ordering is NP-hard [151].

FF The ***first-fit*** ordering heuristic [141, 197] colors vertices in the order they appear in the input graph representation.

R The ***random*** ordering heuristic [118] colors vertices in a uniformly random order.

LF The ***largest-degree-first*** ordering heuristic [197] colors vertices in order of decreasing degree.

ID The ***incidence-degree*** ordering heuristic [50] iteratively colors an uncolored vertex with the largest number of colored neighbors.

SL The ***smallest-degree-last*** ordering heuristic [4, 147] colors the vertices in the order induced by first removing all the lowest-degree vertices from the graph, then recursively coloring the resulting graph, and finally coloring the removed vertices.

SD The ***saturation-degree*** ordering heuristic [32] iteratively colors an uncolored vertex whose colored neighbors use the largest number of distinct colors.

Tables 4-2 and 4-3 summarizes our empirical evaluation of the six ordering heuristics above run on our suite of real-world and synthetic graphs. The measurements were taken using the same machine and methodology as was used for Tables 4-13 and 4-14. As Tables 4-2 and 4-3 show, we found that, in order, FF, R, LF, SL, and SD generally produce better colorings at the cost of greater running times, confirming the findings of Gebremedhin and Manne [80], who also rank the relative quality of R, LF, ID, and SD in this order. ID was outperformed in both time and quality by SL. The figure indicates that LF tends to produce better colorings than FF and R at some performance cost, and SL produces better colorings than LF at additional cost. We found that SD produces the best colorings overall, at the cost of a 4.5 geometric-mean slowdown versus SL.

Although an ordering heuristic can be viewed as producing a permutation of the vertices of a graph $G = (V, E)$, we shall find it convenient to think of an ordering heuristic H as producing an injective (1-to-1) ***priority function***[2] $\rho : V \to \mathbb{R}$. We shall use the notation $\rho \in H$ to mean that the ordering heuristic H produces a priority function ρ.

[2]If the rule for an ordering heuristic allows for ties in the priority function (the priority function is not injective), we shall assume that ties are broken randomly. Formally, suppose that an ordering heuristic H produces a priority function ρ_H which may contain ties. We extend ρ_H to a priority function ρ that maps each vertex $v \in V$ to an ordered pair $\langle \rho_H(v), \rho_{\mathrm{R}}(v) \rangle$, where the priority function ρ_{R} is produced by the random ordering heuristic R. To determine which of two vertices $u, v \in V$ has higher priority, we compare the ordered pairs $\rho(u)$ and $\rho(v)$ lexicographically. Notwithstanding this subtlety, we shall still adopt the simplifying convenience of viewing the priority function as mapping vertices to real numbers. In fact, the range of the priority function can be any linearly ordered set.

Figure 4-1 gives the pseudocode for Greedy, a greedy coloring algorithm. Greedy takes a vertex-weighted graph $G = (V, E, \rho)$ as input, where $\rho : V \to \mathbb{R}$ is a priority function produced by some ordering heuristic. Each step of Greedy simply selects the uncolored vertex with the highest priority according to ρ and colors it with the smallest available color. Generally, for a coloring algorithm A and ordering heuristic H, let A-H denote the coloring algorithm A that runs on vertex-weighted graphs whose priority functions are produced by H. In this way, we separate the behavior of the coloring algorithm from that of the ordering heuristic.

	C						
Graph	FF	R	LF	ID	SL	SD	*Spark*
com-orkut	175	132	87	86	83	76	
liveJournal1	352	330	323	325	322	326	
europe-osm	5	5	4	4	3	3	
cit-Patents	17	21	14	14	13	12	
as-skitter	103	81	71	72	70	70	
wiki-Talk	102	85	72	57	56	51	
web-Google	44	44	45	45	44	44	
com-youtube	57	46	32	28	28	26	
constant1M	33	32	32	34	34	26	
constant500K	52	52	52	55	53	44	
graph500-5M	220	220	159	157	158	147	
graph500-2M	206	208	153	152	153	141	
rMat-ER-2M	12	12	11	11	11	8	
rMat-G-2M	27	27	15	15	15	11	
rMat-B-2M	105	105	67	67	67	59	
big3dgrid	4	7	7	4	7	5	
cliqueChain400	399	399	399	399	399	399	
path-10M	2	3	3	2	2	2	

Table 4-2: Coloring quality performance measurements for six serial ordering heuristics used by Greedy in Figure 4-1, where measurements for real-world graphs appear above the center line and those for synthetic graphs appear below. The "*Spark*" column contains bar graphs that pictorially represent the coloring quality for each of the ordering heuristics. The height of the bar for the coloring quality C_H of ordering heuristic H is proportional to C_H. Section 4.6 details the experimental setup and graph suite used.

Using any of these six ordering heuristics, Greedy can be made to run in $\Theta(V + E)$ time. Although some of these ordering heuristics involve more bookkeeping than others, achieving these theoretical bounds for Greedy-FF, Greedy-R, Greedy-LF, Greedy-ID, and Greedy-SL is straightforward [85, 147]. Despite conjectures to the contrary [50, 85], Greedy-SD can also be made to run in $\Theta(V + E)$ time, as we shall show in Section 4.8.

	T_S						
Graph	FF	R	LF	ID	SL	SD	*Spark*
com-orkut	2.23	3.39	3.54	44.13	10.59	46.60	
liveJournal1	0.89	2.05	2.34	17.93	4.69	19.75	
europe-osm	1.32	13.36	17.15	48.59	19.87	52.73	
cit-Patents	0.50	1.62	2.00	9.82	3.21	10.08	
as-skitter	0.24	1.70	2.43	9.41	2.79	9.94	
wiki-Talk	0.09	0.35	0.49	2.79	0.61	2.90	
web-Google	0.09	0.22	0.25	1.68	0.47	1.77	
com-youtube	0.06	0.19	0.25	1.50	0.35	1.55	
constant1M	0.90	1.13	1.16	16.07	2.96	17.23	
constant500K	0.74	0.88	0.84	14.20	1.97	15.51	
graph500-5M	1.83	3.14	3.69	25.19	8.43	35.29	
graph500-2M	0.52	0.77	0.98	8.09	2.22	11.68	
rMat-ER-2M	0.47	0.93	1.07	10.10	2.22	9.13	
rMat-G-2M	0.48	0.92	1.18	9.17	2.59	9.07	
rMat-B-2M	0.50	0.83	1.00	8.44	2.41	8.64	
big3dgrid	0.41	3.34	4.07	13.61	4.77	15.30	
cliqueChain400	0.05	0.05	0.05	0.81	0.08	2.06	
path-10M	0.18	1.95	2.49	7.34	2.58	7.96	

Table 4-3: Performance measurements for six serial ordering heuristics used by Greedy in Figure 4-1, where measurements for real-world graphs appear above the center line and those for synthetic graphs appear below. The "*Spark*" column contains bar graphs that pictorially represent the serial running time for each of the ordering heuristics. The height of the bar for the serial running time T_S of ordering heuristic H is proportional to the log of T_S. Section 4.6 details the experimental setup and graph suite used.

In practice, to produce a better quality coloring tends to cost more in running time. That is, the six heuristics, which are listed in increasing order of coloring quality, are also listed in increasing order of running time. The only exception is Greedy-ID, which is dominated by Greedy-SL in both coloring quality and runtime.

Parallel greedy coloring

```
JP($G$)
let $G = (V, E, \rho)$
parallel for $v \in V$
    $v.pred = \{u \in V : (u, v) \in E$ and $\rho(u) > \rho(v)\}$
    $v.succ = \{u \in V : (u, v) \in E$ and $\rho(u) < \rho(v)\}$
    $v.counter = |v.pred|$
parallel for $v \in V$
    if $v.pred == \emptyset$
        JP-Color($v$)
```

```
JP-Color($v$)
$v.color$ = Get-Color($v$)
parallel for $u \in v.succ$
    if Join($u.counter$) == 0
        JP-Color($u$)
```

```
Get-Color($v$)
$C = \{1, 2, \ldots, |v.pred| + 1\}$
parallel for $u \in v.pred$
    $C = C - \{u.color\}$
return $\min C$
```

Figure 4-4: The Jones-Plassmann (JP) parallel coloring algorithm. JP uses a recursive helper function JP-Color to process a vertex once all of its predecessors have been colored. JP-Color uses the helper routine Get-Color to find the smallest color available to color a vertex v.

There is a practical tradeoff between coloring quality and the parallel scalability of greedy graph coloring. While the traditional ordering heuristics FF, LF, ID, and SL are efficient using Greedy, it can be shown that any parallelization of them requires worst-case span of $\Omega(V)$ for a general graph $G = (V, E)$. Of the various attempts to parallelize greedy coloring [49, 66, 144], the algorithm first proposed by Jones and Plassmann [118] extends the greedy algorithm in a straightforward manner, uses work linear in size of the graph, and is deterministic given a small (e.g., $O(\lg V)$ random bits) random seed. Jones and Plassmann's original paper demonstrates good theoretical parallel performance for $O(1)$-degree graphs using the random ordering heuristic R, though their implementation failed to generate a parallel speedup. Unfortunately, in practice, R tends to produce colorings of relatively poor quality relative to the other traditional ordering heuristics. But the other traditional ordering heuristics are all vulnerable to adversarial graph inputs which cause JP to operate in $\Omega(V)$ time and thus exhibit poor parallel scalability. Consequently, there is need for new ordering heuristics for JP that can achieve both good coloring quality and guaranteed fast parallel performance.

Figure 4-4 gives the pseudocode for JP, which colors a given graph $G = (V, E, \rho)$ in the order specified by the priority function ρ. The algorithm begins in lines 217 and 218 by partitioning the neighbors of each vertex into ***predecessors*** — vertices with larger priorities — and ***successors*** — vertices with smaller priorities. JP uses the recursive JP-Color helper function to color a vertex $v \in V$ once all vertices in $v.pred$ have been colored. Initially, lines 220–222 in JP scan the vertices of V to find every vertex that has no predecessors and colors each one using JP-Color. Within a call to JP-Color(v), line 223 calls Get-Color to assign a color to v, and the loop on lines 224–226 broadcasts in parallel to all of v's successors the fact that v is colored. For each successor $u \in v.succ$, line 225 tests whether all of u's predecessors have already been colored, and if so, line 226 recursively calls JP-Color on u.

Jones and Plassmann analyzed the performance of JP-R for $O(1)$-degree graphs. Although they do not discuss using the naive FF ordering heuristic, it is apparent that there exist adversarial input orderings for which their algorithm would fail to scale. For example, if the graph $G = (V, E)$ is simply a chain of vertices and the input order of V corresponds to their in order in the chain, JP-FF exhibits no parallelism. Jones and Plassmann show that a random ordering produced by R, however, allows the algorithm to run in $O(\lg V / \lg \lg V)$ expected time on this chain graph — and on any $O(1)$-degree graph, for that matter. Section 4.3 of this chapter extends their analysis of JP-R to arbitrary-degree graphs.

Although JP-R scales well in theory, when it comes to coloring quality, R is one of the weaker ordering heuristics, as we have noted. Of the other heuristics, JP-LF and JP-SL suffer from the same problem as FF, namely, it is possible to construct adversarial graphs that cause them to scale poorly, which we explore in Section 4.4. The ID heuristic tends to produce worse colorings than SL, and since Greedy-ID also runs more slowly than Greedy-SL, we have dropped ID from consideration. Moreover, because of our motivation to use the coloring algorithm for online chromatic scheduling, where the performance of the coloring algorithm cannot be sacrificed for marginal improvements in the quality of coloring, we also have dropped the SD heuristic. Since SD produces the best-quality colorings of the six ordering heuristics, however, we see parallelizing it as an interesting opportunity for future research.

Consequently, this chapter focuses on alternatives to the LF and SL ordering heuristics that provide comparable coloring quality while exhibiting the same resilience to adversarial graphs that R shows compared with FF. Specifically, we introduce two new randomized ordering heuristics — "largest-log-degree-first" (LLF) and "smallest-log-degree-last" (SLL) — which resemble LF and SL, respectively, but which scale provably well when used with JP. We demonstrate that JP-LLF and JP-SLL provide good parallel scalability in theory and practice and are resilient to adversarial graphs.

Table 4-5 summarizes our empirical findings. The data suggest that the LLF and SLL ordering heuristics produce colorings that are nearly as good as LF and SL, respectively. With respect to performance, our implementations of JP-LLF and JP-SLL actually operate slightly faster on 1 processor than our highly tuned implementations of GREEDY-LF and GREEDY-SL, respectively, and they scale comparably to JP-R.

H	H'	$\frac{C_{H'}}{C_H}$	$\frac{\text{GREEDY-}H}{\text{JP-}H'_1}$	$\frac{\text{JP-}H'_1}{\text{JP-}H'_{12}}$
FF	R	1.011	0.417	7.039
LF	LLF	1.021	1.058	7.980
SL	SLL	1.037	1.092	6.082

Table 4-5: Summary of ordering-heuristic behavior on a suite of 8 real-world graphs and 10 synthetic graphs when run on a machine with 12 Intel Xeon X5650 processor cores. Column H lists three serial heuristics traditionally used for GREEDY, and column H' lists parallel heuristics for JP, of which LLF and SLL are introduced in this chapter. Column "$C_{H'}/C_H$" shows the geometric mean of the ratio of the number of colors the parallel heuristic uses compared to the serial heuristic. Column "GREEDY-H/JP-H'_1" shows the geometric mean of the ratio of serial running times of GREEDY with the serial heuristic versus JP with the analogous parallel heuristic when run on 1 processor. Column "JP-H'_1/JP-H'_{12}" shows the geometric mean of the speedup of each parallel heuristic going from 1 processor to 12.

Outline

This chapter represents joint work with Charles E. Leiserson, Tim Kaler, and Tao B. Schardl that was presented at the 2014 ACM Symposium on Parallelism in Algorithms and Architectures under the title "Ordering heuristics for parallel graph coloring" [102].

The remainder of this chapter is organized as follows. Section 4.2 reviews the asynchronous parallel greedy coloring algorithm first proposed by Jones and Plassmann [118]. We show how JP can be extended to handle arbitrary-degree graphs and arbitrary priority functions. Using work-span analysis [52, Ch. 27], we show that JP colors a Δ-degree graph $G = (V, E, \rho)$ in $\Theta(V + E)$ work and $O(L \lg \Delta + \lg V)$ span, where L

is the length of the longest path in G along which the priority function ρ decreases. Section 4.3 analyzes the performance of JP-R, showing that it operates using linear work and $O\left(\ln V + \ln\Delta \cdot \min\{\sqrt{E}, \Delta + \ln V/\ln(e\ln V/\Delta)\}\right)$ span. Section 4.4 shows that there exist "adversarial" graphs for which JP-LF and JP-SL exhibit limited parallel speedup. Section 4.5 analyzes the LLF and SLL ordering heuristics. We show that, given a Δ-degree graph G, JP-LLF colors $G = (V, E, \rho)$ using $\Theta(V + E)$ work and $O\left(\ln V + \ln\Delta\left(\min\{\sqrt{E}, \Delta\} + \ln\Delta \ln V/\ln(e\ln V/\Delta)\right)\right)$ expected span, while JP-SLL colors $G = (V, E, \rho)$ using same work and an additive $\Theta(\lg\Delta \lg V)$ additional span. Section 4.6 evaluates the performance of JP-LLF and JP-SLL on a suite of 8 real-world and 10 synthetic benchmark graphs. Section 4.7 discusses the software engineering techniques used in our implementation of JP-R, JP-LLF, and JP-SLL. Section 4.8 introduces an algorithm for computing the SD ordering heuristic using $\Theta(V + E)$ work. Section 4.9 discusses related work, and Section 4.10 offers some concluding remarks.

4.2 The Jones-Plassmann algorithm

This section reviews JP, the parallel greedy coloring algorithm introduced by Jones and Plassmann [118], whose pseudocode is given in Figure 4-4. We first review the dag model of dynamic multithreading and work-span analysis [52, Ch. 27]. Then we describe how JP can be modified from Jones and Plassmann's original algorithm to handle arbitrary-degree graphs and arbitrary priority functions. We analyze JP with an arbitrary priority function ρ and show that on a Δ-degree graph $G = (V, E, \rho)$, JP runs in $\Theta(V + E)$ work and $O(L\lg\Delta + \lg V)$ span, where L is the longest path in the "priority dag" of G induced by ρ.

The dag model of dynamic multithreading

We shall analyze the parallel performance of JP using the dag model of dynamic multithreading introduced by Blumofe and Leiserson [25, 26] and described in more detail in Appendix A. The dag model views the executed computation resulting from running a parallel algorithm as a ***computation dag*** A, in which each vertex denotes an instruction, and edges denote parallel control dependencies between instructions. Although the model encompasses other parallel control constructs, for our purposes, we need only understand that the execution of a **parallel for** loop can be modeled as a balanced binary tree of vertices in the dag, where the leaves of the tree denote the initial instructions of the loop iterations.

To analyze the performance of a dynamic multithreading program theoretically, we assume that the program executes on an ***ideal parallel computer***: each instruction executes in unit time, the computer has ample memory bandwidth, and the computer supports concurrent writes and read-modify-write instructions [105] without incurring overheads due to contention.

Given a dynamic multithreading program whose execution is modeled as a dag A, we can bound the parallel running time $T_P(A)$ of the computation as follows. The ***work*** $T_1(A)$ is the number of strands in the computation dag A. The ***span*** $T_\infty(A)$ is the length of the longest path in A. A deterministic algorithm with work T_1 and span T_∞ can always be executed on P processors in time T_P satisfying $\max\{T_1/P, T_\infty\} \leq T_p \leq T_1/P + T_\infty$ [25, 26, 33, 69, 94]. The ***speedup*** of an algorithm on P processors is T_1/T_P, which is at most P in theory, since $T_P \geq T_\infty$. The ***parallelism*** T_1/T_∞ is the greatest theoretical speedup possible for any number P of processors.

Analysis of JP

To analyze the performance of JP, it is convenient to think of the algorithm as coloring the vertices in the partial order of a "priority dag," similar to the priority dag described by Blelloch, Fineman, and Shun [23]. Specifically, on a vertex-weighted graph $G = (V, E, \rho)$, the priority function ρ induces a ***priority dag*** $G_\rho = (V, E_\rho)$, where $E_\rho = \{(u, v) \in V \times V : (u, v) \in E \text{ and } \rho(u) > \rho(v)\}$. Notice that G_ρ is a dag, because ρ is an injective function and thus induces a total order on the vertices V. We shall bound the span of JP running on a graph G in terms of the ***depth*** of G_ρ, that is, the length of the longest path through G_ρ. We analyze JP in two steps.

First, we bound the work and span of calls during the execution of JP to the helper routine GET-COLOR(v), which returns the minimum color not assigned to any vertex $u \in v.pred$.

Lemma 12 *The helper routine* GET-COLOR*, shown in Figure 4-4, can be implemented so that during the execution of* JP *on a graph* $G = (V, E, \rho)$*, a call to* GET-COLOR(v) *for a vertex* $v \in V$ *costs* $\Theta(k)$ *work and* $\Theta(\lg k)$ *span, where* $k = |v.pred|$.

PROOF. Implement the set C in GET-COLOR as an array whose ith entry initially stores the value i. The ith element from this array can be removed by setting the ith element to ∞. With this implementation, lines 228–229 execute in $\Theta(k)$ work and $\Theta(\lg k)$ span. The min operation on line 230 can be implemented as a parallel minimum reduction in the same bounds. □

Second, we show that JP colors a graph $G = (V, E, \rho)$ using work $\Theta(V + E)$ and span linear in the depth of the priority dag G_ρ.

Theorem 13 *Given a Δ-degree graph $G = (V, E, \rho)$ for some priority function ρ, let G_ρ be the priority dag induced on G by ρ, and let L be the depth of G_ρ. Then* JP(G) *runs in $\Theta(V + E)$ work and $O(L \lg \Delta + \lg V)$ span.*

PROOF. Let us first bound the work and span of JP-COLOR excluding any recursive calls. For a single call to JP-COLOR on a vertex $v \in V$, Lemma 12 shows that line 223 takes $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span. The JOIN operation on line 225 can be implemented as an atomic decrement-and-fetch operation [105] on the specified counter. Hence, excluding the recursive call, the loop on lines 224–226 performs $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span to decrement the counters of all successors of v.

Because JP-COLOR is called once per vertex, the total work that JP spends in calls to JP-COLOR is $\Theta(V + E)$. Furthermore, the span of JP-COLOR is the length of any path of vertices in G_ρ, which is at most L, times $\Theta(\lg \Delta)$. Finally, the loop on lines 216–219 executes in $\Theta(V + E)$ work and $\Theta(\lg V + \lg \Delta)$ span, and the parallel loop on lines 220–222, excluding the call to JP-COLOR, executes in $\Theta(V + E)$ work and $\Theta(\lg V)$ span. □

4.3 Jones-Plassmann with random ordering

This section bounds the depth of a priority dag G_ρ induced on a Δ-degree graph[3] $G = (V, E, \rho)$ by a random priority function ρ in R. We show that the expected depth of G_ρ is $\Theta\left(\min\{\sqrt{E}, \Delta + \ln V / \ln(e \ln V/\Delta)\}\right)$. This bound extends Jones and Plassmann's $O(\lg V / \lg \lg V)$ bound for the depth of G_ρ when $\Delta = \Theta(1)$ [118]. Combined with Theorem 13, our new bound implies that the expected span of JP-R is $O\left(\ln V + \ln \Delta \cdot \min\{\sqrt{E}, \Delta + \ln V / \ln(e \ln V/\Delta)\}\right)$.

To bound the depth of a priority dag G_ρ induced on a graph G by $\rho \in$ R, let us start by bounding the number of length-k paths in G_ρ. Each path in G_ρ corresponds to a unique ***simple*** path in G, that is, a path in which each vertex in G appears at most once. The following lemma bounds the number of length-k simple paths in G.

[3]Throughout this chapter we assume that graphs are simple and connected.

Lemma 14 *The number of length-k simple paths in any Δ-degree graph $G = (V, E)$ is at most* $|V| \cdot \min\{\Delta^{k-1}, (2|E|/(k-1))^{k-1}\}$.

PROOF. Consider selecting a length-k simple path $p = \langle v_1, \ldots, v_k \rangle$ in G. There are $|V|$ choices for v_1, and for all $i \in \{1, \ldots, k-1\}$, given a choice of $\langle v_1, \ldots, v_i \rangle$, there are at most $\deg(v_i)$ choices for v_{i+1}. Hence there are at most $J = |V| \cdot \prod_{i=1}^{k-1} \deg(v_i)$ simple paths in G of length k. Let $V_k \subseteq V$ denote some set of $k-1$ vertices in V, and let $\delta = \max_{V_{k-1}} \{\sum_{v \in V_{k-1}} \deg(v)/(k-1)\}$ be the maximum average degree of any such set. Then we have $J \leq |V| \cdot \delta^{k-1}$.

The proof follows from two upper bounds on δ. First, because $\deg(v) \leq \Delta$ for all $v \in V$, we have $\delta \leq \Delta$. Second, for all $V_{k-1} \subseteq V$, we have $\sum_{v \in V_{k-1}} \deg(v) \leq \sum_{v \in V} \deg(v) = 2|E|$ by the handshaking lemma [52, p. 1172–3], and thus $\delta \leq 2|E|/(k-1)$. $\square$

Intuitively, the bound on the expected depth of G_ρ follows by arguing that although the number of simple length-k paths in a graph G might be exponential in k, for sufficiently large k, the probability is tiny that any such path is a path in G_ρ. To formalize this argument, we consider the cases where Δ is large and small separately, specifically when $\Delta > \ln n$ and when $\Delta \leq \ln n$.

The longest path in random priority dags of large degree

We show that when Δ is larger than $\ln|V|$, the depth of the random priority dag G_ρ, induced on $G = (V, E, \rho)$ by the random priority function ρ, has a tight bound $\Theta\left(\min\{\Delta, \sqrt{E}\}\right)$. The following two lemmas show the upper bound and lower bound, respectively.

Lemma 15 *Let $G = (V, E)$ be a Δ-degree graph for any $\Delta > \ln n$, let $n = |V|$ and $m = |E|$, and let G_ρ be a priority dag induced on G by a random priority function $\rho \in$* R. *For any constant $c > 0$ and sufficiently large n, there exists a directed path of length $e^2 \cdot \min\{\Delta, \sqrt{m}\} + (1+c)\ln n$ in G_ρ with probability at most n^{-c}.*

PROOF. Let $p = \langle v_1, \ldots, v_k \rangle$ be a length-k simple path in G. Because ρ is a random priority function, ρ induces each possible permutation among $\{v_1, \ldots, v_k\}$ with equal probability. If p is a directed path in G_ρ, then we must have that $\rho(v_1) < \rho(v_2) < \cdots < \rho(v_k)$. Hence, p is a length-k path in G_ρ with probability at most $1/k!$. If J is the number of length-k simple paths in G, then by the union bound, the probability that a length-k directed path exists in G_ρ is at most $J/k!$, which is at most $J(e/k)^k$ by Stirling's approximation [52, p. 57] [186].

We consider the cases when $\Delta < \sqrt{m}$ and $\Delta \geq \sqrt{m}$, separately. When $\Delta < \sqrt{m}$, letting $k = e^2\Delta + (1+c)\ln n$ and $J = n \cdot \Delta^{k-1}$, the theorem follows from the facts that $k \geq (1+c)\ln n$ and $k \geq e^2\Delta$:

$$\begin{aligned}
J\,(e/k)^k &= n \cdot \Delta^{k-1}\,(e/k)^k \\
&\leq n \cdot (e\Delta/k)^k \\
&= n \cdot \exp\left(-k \ln\left(k/e\Delta\right)\right) \\
&= n \cdot \exp\left(-\left(e^2\Delta + (1+c)\ln n\right)\ln\left(\frac{e^2\Delta + (1+c)\ln n}{e\Delta}\right)\right) \\
&\leq n \cdot \exp\left(-\left(e^2\Delta + (1+c)\ln n\right)\ln\left(\frac{e^2\Delta}{e\Delta}\right)\right) \\
&= n \cdot \exp\left(-\left(e^2\Delta + (1+c)\ln n\right)\ln e\right) \\
&\leq n \cdot \exp\left(-\left((1+c)\ln n\right)\right) \\
&= n^{-c}.
\end{aligned}$$

When $\Delta \geq \sqrt{m}$, let $k = e^2\sqrt{m} + (1+c)\ln n$. By Lemma 14, the number of length-k simple paths is at most $n(2m/(k-1))^{k-1} \leq n(4m/k)^k$, and thus the probability that a length-k path exists in G_ρ is at most $n(4em/k^2)^k$. The theorem follows from the facts that $k \geq (1+c)\ln n$ and $k^2 \geq e^4 m$. □

Lemma 16 *There exists a Δ-degree graph $G = (V, E)$ with $\Delta > \ln|V|$ such that the length of the longest directed path in G_ρ is at least* $\max\{\Delta, \sqrt{|E|}\}$*, where G_ρ is the priority dag induced on G by any priority function ρ.*

PROOF. Let G be formed by two subgraphs. The first is a clique, $K_{\Delta+1}$, with $\Delta+1$ vertices and $\Delta(\Delta+1)$ edges. The second subgraph is a graph with $n - \Delta - 1$ vertices and no edges, thus $\Delta < \sqrt{|E|} < \Delta + 1$. The longest path in $K_{\Delta+1}$ is $\Delta + 1$ irrespective of the ordering by the priority function ρ. □

Corollary 17 *Let $G = (V, E)$ be a Δ-degree graph for any $\Delta > \ln|V|$ and let G_ρ be a priority dag induced on G by a random priority function $\rho \in$ R. The expected length of the longest path in G_ρ is* $\Theta\left(\min\{\Delta, \sqrt{E}\}\right)$.

PROOF. The proof follows from Lemmas 15 and 16. □

The longest path in random priority dags of small degree

We show that when Δ is smaller than $\ln|V|$, the depth of the random priority dag G_ρ, induced on $G = (V, E, \rho)$ by random priority function ρ, has a tight bound $\Theta(\ln V/\ln(e\ln V/\Delta))$. Lemmas 18 and 20 show the upper bound and lower bound, respectively.

Lemma 18 *Let $G = (V, E)$ be a Δ-degree graph for any $\Delta \le \ln|V|$, let $n = |V|$, and let G_ρ be a priority dag induced on G by a random priority function $\rho \in \mathrm{R}$. For any $c > 0$ and sufficiently large n, there exists a directed path of length $(1+e+c)\ln n/\ln(e\ln n/\Delta)$ in G_ρ with probability at most n^{-c}.*

Proof. Let $p = \langle v_1, \ldots, v_k\rangle$ be a length-k simple path in G. Because ρ is a random priority function, ρ induces each possible permutation among $\{v_1, \ldots, v_k\}$ with equal probability. If p is a directed path in G_ρ, then we must have that $\rho(v_1) < \rho(v_2) < \cdots < \rho(v_k)$. Hence, p is a length-k path in G_ρ with probability at most $1/k!$. If J is the number of length-k simple paths in G, then by the union bound, the probability that a length-k directed path exists in G_ρ is at most $J/k!$, which is at most $J(e/k)^k$ by Stirling's approximation [52, p. 57] [186].

Let $a = 1+e+c$, let $k = a\ln n/\ln(e\ln n/\Delta)$, and let $J = n\cdot\Delta^{k-1}$. Using the fact that $\ln(x) \ge 1 - 1/x$ for all $x > 0$, which follows[4] from the primitive fact that $e^x > 1 + x$ for all x, the probability that there exists a directed path of length at least k is at most

$$\begin{aligned}
J(e/k)^k &\le n(e\Delta/k)^k \\
&= n\cdot\exp(-k\ln(k/e\Delta)) \\
&= n\cdot\exp\left(-\frac{a\ln n}{\ln(e\ln n/\Delta)}\ln\left(\frac{a\ln n}{\ln(e\ln n/\Delta)}\cdot\frac{1}{e\Delta}\right)\right) \\
&= n\cdot\exp\left(-\frac{a\ln n}{\ln(e\ln n/\Delta)}\left(\ln(e\ln n/\Delta) - \ln e + \ln\left(\frac{a}{e\ln(e\ln n/\Delta)}\right)\right)\right) \\
&\le n\cdot\exp\left(-\frac{a\ln n}{\ln(e\ln n/\Delta)}\left(\ln(e\ln n/\Delta) - \ln e + \left(1 - \left(\frac{a}{e\ln(e\ln n/\Delta)}\right)^{-1}\right)\right)\right) \\
&= n\cdot\exp\left(-\frac{a\ln n}{\ln(e\ln n/\Delta)}\left(\ln(e\ln n/\Delta) - \left(\frac{a}{e\ln(e\ln n/\Delta)}\right)^{-1}\right)\right) \\
&= n\cdot\exp\left(-\frac{a\ln n}{\ln(e\ln n/\Delta)}\left(\ln(e\ln n/\Delta) - \frac{e}{a}\ln(e\ln n/\Delta)\right)\right)
\end{aligned}$$

[4]Rearranging the identity $e^{-y} \ge 1 - y$ shows us that $y \ge 1 - e^{-y}$. Substituting $y = \ln x$ gives us $\ln x \ge 1 - 1/x$ for all $x > 0$.

$$= n \cdot \exp\left(-\left(a - e\right)\ln n\right)$$
$$= n^{1+e-a}$$
$$= n^{-c}.$$

□

Lemma 19 *For any positive integers d and any real number $k > 0$, $\left(\left(k-1\right)/k\right)^d \leq k/\left(k+d\right)$.*

Proof. The proof is by induction on d.

The base case, with $d = 1$, holds, since $k^2/\left(k^2-1\right) > 1$:

$$\begin{aligned}\frac{k-1}{k} &\leq \frac{k^2}{k^2-1}\cdot\frac{k-1}{k}\\ &\leq \frac{k^2}{\left(k-1\right)\left(k+1\right)}\cdot\frac{k-1}{k}\\ &= \frac{k}{k+1}.\end{aligned}$$

Assuming that the lemma holds for $d-1$, the inductive step follows similarly:

$$\begin{aligned}\left(\frac{k-1}{k}\right)^d &= \left(\frac{k-1}{k}\right)^{d-1}\cdot\frac{k-1}{k}\\ &\leq \frac{k}{k+d-1}\cdot\frac{k-1}{k}\\ &= \frac{k-1}{k+d-1}\\ &\leq \frac{k\left(k+d-1\right)}{k\left(k+d-1\right)-d}\cdot\frac{k-1}{k+d-1}\\ &= \frac{k\left(k+d-1\right)}{\left(k+d\right)\left(k-1\right)}\cdot\frac{k-1}{k+d-1}\\ &= \frac{k}{k+d}.\end{aligned}$$

□

Lemma 20 *Let $G = (V, E)$ be a Δ-degree graph for any $\Delta < \ln|V|$, let $n = |V|$, and let G_ρ be a priority dag induced on G by a random priority function $\rho \in R$. For any $c > 0$ and sufficiently large n, the length of the longest directed path in G_ρ is less than $\left((e/4) - \log_n\left(c\ln n\right)\right)\ln n/\ln\left(e\ln n/\Delta\right)$ with probability at most n^{-c}.*

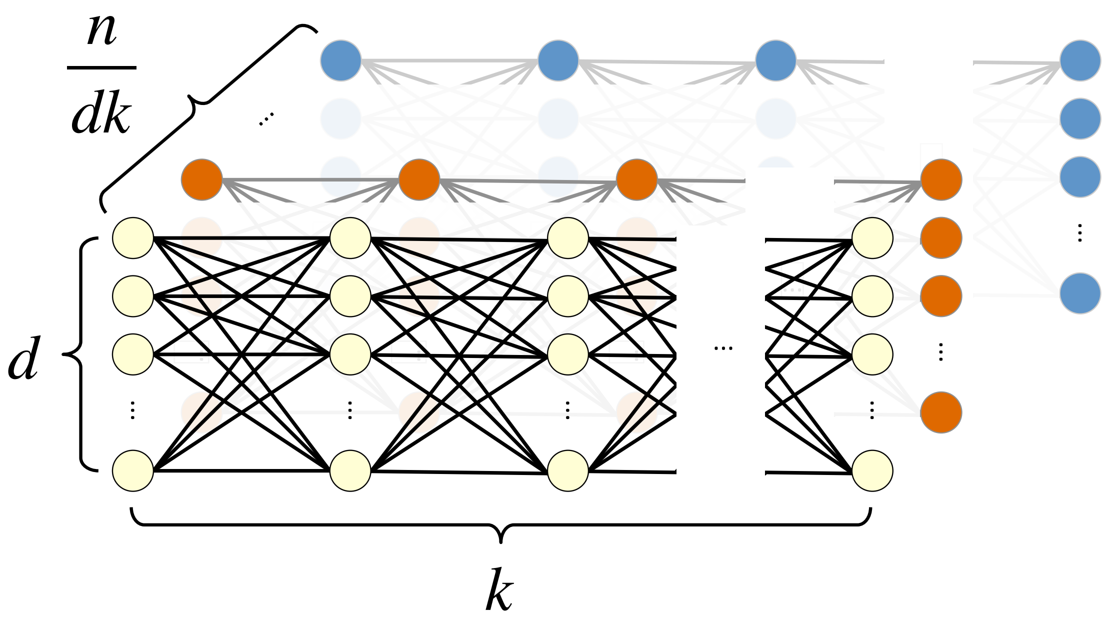

Figure 4-6: Example graph demonstrating an existential lower bound for the longest path in a random priority dag. The graph consists of n/dk independent subgraphs, each with k columns and d rows. Each column is connected to its adjacent columns as a complete bipartite graph.

Proof. Let G be the graph depicted in Figure 4-6, where $d = \Delta/2$, and let $G_{i,\rho}$ be the priority dag induced on the ith slice of G by the random priority function ρ, where a "slice" is a sequence of k columns each adjacent pair of which is connected as a complete bipartite graph. For convenience and without loss of generality, assume that the priority function $\rho : V \to [0, 1]$ maps each vertex uniformly randomly to the unit interval. If the jth column of $G_{i,\rho}$ has at least one vertex mapped to the interval $[j/k, (j+1)/k)$ by ρ for all j, then a directed path of length k would exist in $G_{i,\rho}$. Since every vertex in every column is connected to every other vertex in the adjacent columns, having at least one vertex in the k such non-overlapping intervals would guarantee a directed path in $G_{i,\rho}$. Each individual column has at least one vertex in the required interval with probability $(1 - 1/k)^d$. The probability that all columns in $G_{i,\rho}$ have at least one vertex in the required interval is then $\left(1 - (1 - 1/k)^d\right)^k$. Let $a = ((e/4) - \log_n(c \ln n))$, which is less than $e/4$ for all $n, c > 0$, and note that $n^{e/4} < n/dk$ for all $n > e$ and the given values of d and k. Since each identical slice is independent and by Lemma 19, the probability that no slice has a k-length directed path is at most

$$
\begin{aligned}
&\le \left(1-\left(1-(1-1/k)^d\right)^k\right)^{n/dk}\\
&\le \left(1-(1-(k/(k+d)))^k\right)^{n/dk}\\
&= \left(1-(d/(k+d))^k\right)^{n/dk}\\
&\le \left(1-(d/2k)^k\right)^{n/dk}\\
&= (1-\exp(-k\ln(2k/d)))^{n/dk}\\
&= (1-\exp(-k\ln(4k/\Delta)))^{n/dk}\\
&= \left(1-\exp\left(-k\ln\left(\frac{4}{\Delta}\cdot\frac{a\ln n}{\ln(e\ln n/\Delta)}\right)\right)\right)^{n/dk}\\
&= \left(1-\exp\left(-k\left(\ln(e\ln n/\Delta)+\ln\left(\frac{4a}{e\ln(e\ln n/\Delta)}\right)\right)\right)\right)^{n/dk}\\
&\le (1-\exp(-k\ln(e\ln n/\Delta)))^{n/dk}\\
&= \left(1-\exp\left(-\frac{a\ln n}{\ln(e\ln n/\Delta)}\ln(e\ln n/\Delta)\right)\right)^{n/dk}\\
&= \left(1-n^{-a}\right)^{n/dk}\\
&\le \left(1-n^{-a}\right)^{n^{e/4}}\\
&\le \exp\left(-n^{(e/4)-a}\right)\\
&= \exp\left(-n^{\log_n(c\ln n)}\right)\\
&= \exp(-c\ln n)\\
&= n^{-c}.
\end{aligned}
$$

□

Corollary 21 *Let $G=(V,E)$ be a Δ-degree graph for any $\Delta \le \ln|V|$ and let G_ρ be a priority dag induced on G by a random priority function $\rho \in \mathrm{R}$. The expected length of the longest path in G_ρ is $\Theta(\ln V/\ln(e\ln V/\Delta))$.*

Proof. The proof follows from Lemmas 18 and 20. □

The longest path in random priority dags of arbitrary degree

We combine the previous results for the cases where Δ is larger and smaller than $\ln|V|$, respectively, to show a tight bound on the depth of the random priority dag G_ρ induced on the arbitrary graph $G = (V, E, \rho)$ by the random priority function ρ. Then, we show that JP-R colors arbitrary graphs with low span.

Theorem 22 *Let $G = (V, E)$ be a Δ-degree graph for any $\Delta \leq \ln|V|$ and let G_ρ be a priority dag induced on G by a random priority function $\rho \in \mathrm{R}$. The expected length of the longest path in G_ρ is $\Theta\left(\min\{\sqrt{E}, \Delta + \ln V / \ln(e \ln V/\Delta)\}\right)$.*

Proof. The proof follows from Corollaries 17 and 21. □

Corollary 23 *Given a priority dag G_ρ induced on graph $G = (V, E, \rho)$ by random priority function $\rho \in \mathrm{R}$,* JP-R *colors all vertices of G with expected span $O\left(\ln V + \ln\Delta \cdot \min\{\sqrt{E}, \Delta + \ln V / \ln(e \ln V/\Delta)\}\right)$.*

Proof. Theorems 13 and 22 imply the corollary. □

4.4 The largest-first and smallest-last heuristics

This section shows that the largest-first (LF) and smallest-last (SL) ordering heuristics can inhibit parallel speedup when used by JP. We examine a "clique-chain" graph and show that JP-LF incurs $\Omega(\Delta^2)$ span to color a Δ-degree clique-chain graph $G = (V, E)$, whereas JP-R colors G, incurring only $O(\ln V + \Delta\ln\Delta + \ln\Delta \lg V / \ln(e\ln V/\Delta))$ expected span. We formally review the SL ordering heuristic and observe that this formulation of SL means that JP-SL requires $\Omega(V)$ span to color a path graph $G = (V, E)$.

Tables 4-7 and 4-8 summarize the performance of FF, LF and SL on a suite of 8 real-world and 10 synthetic benchmark graphs. The number of edges, ratio of edges to vertices and maximum degree of each benchmark graph is given in Table 4-12.

The LF ordering heuristic

The LF ordering heuristic colors the vertices of a graph $G = (V, E, \rho)$ for some ρ in LF in order of decreasing degree. Formally, $\rho \in \mathrm{LF}$ is defined for a vertex $v \in V$ as $\rho(v) = \langle \deg(V), \rho_{\mathrm{R}}(v)\rangle$, where ρ_{R} is randomly chosen from R.

			GREEDY	JP			
Graph	H	C_H	T_S	T_1	T_{12}	T_S/T_1	T_1/T_{12}
com-orkut	FF	175	2.23	4.16	0.817	0.54	5.09
	LF	87	3.54	6.43	1.067	0.55	6.02
	SL	83	10.59	12.94	8.264	0.82	1.57
liveJournal1	FF	352	0.89	1.69	0.275	0.52	6.15
	LF	323	2.34	2.89	0.365	0.81	7.91
	SL	322	4.69	4.76	2.799	0.98	1.70
europe-osm	FF	5	1.32	∞	∞	∞	∞
	LF	4	17.15	5.16	0.587	3.33	8.79
	SL	3	19.87	∞	∞	∞	∞
cit-Patents	FF	17	0.50	0.99	0.152	0.50	6.47
	LF	14	2.00	1.52	0.211	1.31	7.22
	SL	13	3.21	3.05	1.579	1.05	1.93
as-skitter	FF	103	0.24	0.55	0.109	0.45	5.00
	LF	71	2.43	0.69	0.133	3.51	5.21
	SL	70	2.79	1.19	0.733	2.35	1.62
wiki-Talk	FF	102	0.09	0.23	0.046	0.38	4.99
	LF	72	0.49	0.37	0.073	1.30	5.12
	SL	56	0.61	0.57	0.293	1.08	1.93
web-Google	FF	44	0.09	0.20	0.036	0.47	5.62
	LF	45	0.25	0.29	0.042	0.88	6.85
	SL	44	0.47	0.53	0.278	0.89	1.92
com-youtube	FF	57	0.06	0.16	0.027	0.39	6.07
	LF	32	0.25	0.24	0.040	1.03	6.12
	SL	28	0.35	0.36	0.181	0.98	1.99

Table 4-7: Performance measurements for a set of real-world graphs taken from Stanford's SNAP project [136]. The column heading H denotes that the priority function used for the experiment in a particular row was produced by the ordering heuristic listed in the column. The average number of colors used by the corresponding ordering heuristic and graph is C_H. The time in seconds of GREEDY, JP with 1 worker and with 12 workers is given by T_S, T_1 and T_{12}, respectively, where a value of ∞ indicates that the program crashed due to excessive stack usage. Details of the experimental setup and graph suite can be found in Section 4.6.

Although LF has been used in parallel greedy graph-coloring algorithms in the past [4,85], Figure 4-9 illustrates a Δ-degree "clique-chain" graph $G = (V, E)$ for which JP-LF incurs $\Omega(\Delta^2)$ span to color, but JP-R colors with only $O(\ln V + \Delta \ln \Delta + \ln V / \ln (e \ln V/\Delta))$ expected span. Conceptually, the ***clique-chain*** graph comprises a set of cliques of increasing size that are connected in a "chain" such that JP-LF is forced to color these cliques sequentially from largest to smallest. Figure 4-9 illustrates a Δ-degree clique-chain graph

			GREEDY	JP			
Graph	*H*	C_H	T_S	T_1	T_{12}	T_S/T_1	T_1/T_{12}
constant1M	FF	33	0.90	1.70	0.230	0.53	7.40
	LF	32	1.16	2.96	0.386	0.39	7.68
	SL	34	2.96	5.09	2.023	0.58	2.52
constant500K	FF	52	0.74	1.26	0.286	0.59	4.42
	LF	52	0.84	2.55	0.444	0.33	5.73
	SL	53	1.97	3.50	1.435	0.56	2.44
graph500-5M	FF	220	1.83	2.86	0.560	0.64	5.11
	LF	159	3.69	3.99	0.649	0.92	6.15
	SL	158	8.43	9.45	5.576	0.89	1.69
graph500-2M	FF	206	0.52	0.98	0.208	0.53	4.72
	LF	153	0.98	1.34	0.221	0.73	6.06
	SL	153	2.22	2.72	1.559	0.81	1.75
rMat-ER-2M	FF	12	0.47	1.11	0.169	0.42	6.60
	LF	11	1.07	1.72	0.204	0.62	8.45
	SL	11	2.22	3.07	1.362	0.72	2.25
rMat-G-2M	FF	27	0.48	0.88	0.130	0.55	6.74
	LF	15	1.18	1.42	0.200	0.83	7.09
	SL	15	2.59	3.09	1.712	0.84	1.81
rMat-B-2M	FF	105	0.50	0.84	0.151	0.60	5.53
	LF	67	1.00	1.28	0.191	0.79	6.68
	SL	67	2.41	2.84	1.691	0.85	1.68
big3dgrid	FF	4	0.41	1.68	0.173	0.24	9.69
	LF	7	4.07	1.53	0.198	2.66	7.72
	SL	7	4.77	2.60	1.074	1.83	2.42
cliqueChain400	FF	399	0.05	0.09	0.224	0.51	0.40
	LF	399	0.05	∞	∞	∞	∞
	SL	399	0.08	0.14	0.265	0.55	0.54
path-10M	FF	2	0.18	∞	∞	∞	∞
	LF	3	2.49	0.76	0.092	3.26	8.27
	SL	2	2.58	∞	∞	∞	∞

Table 4-8: Performance measurements for five classes of synthetically generated graphs: constant degree, rMat, 3D grid, clique chain and path. The column headings are equivalent to those in Table 4-7.

$G = (V, E)$, where 3 evenly divides Δ. This clique-chain graph contains a sequence of cliques $\mathcal{K} = \{K_1, K_4, \ldots, K_{\Delta-2}\}$ of increasing size, each pair of which is separated by two additional vertices forming a linear chain. Specifically, for $r \in \{1, 4, \ldots, \Delta - 2\}$, each vertex $u \in K_r$ is connected to each vertex $u \in K_{r+3}$ by a path $\langle u, x_{r+1}, x_{r+2}, v\rangle$ for distinct vertices

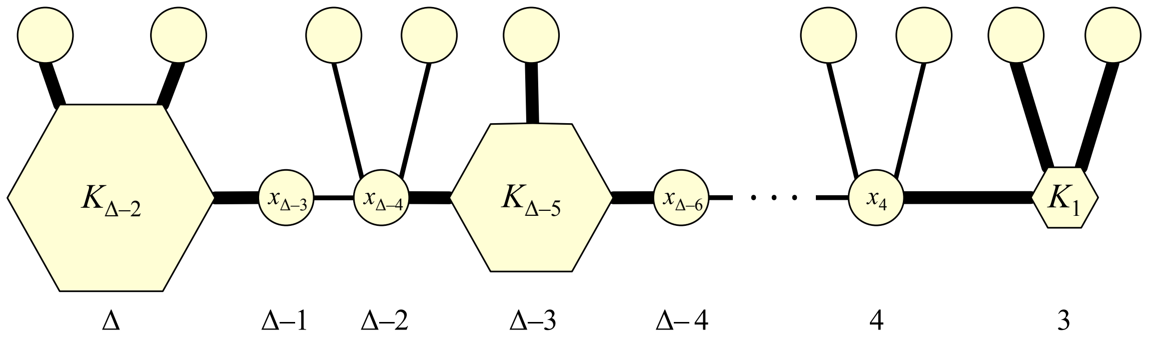

Figure 4-9: A Δ-degree clique-chain graph G, which Theorem 24 shows is adversarial for JP-LF. This graph contains $\Theta(\Delta^2)$ vertices arranged as a chain of cliques. Each hexagon labeled K_r represents a clique of r vertices, and circles represent individual vertices. A thick edge between an individual vertex and a clique indicates that the vertex is connected to every vertex within the clique. A label below an individual vertex indicates the degree of the associated vertex, and a label below a clique indicates the degree of every vertex within that clique.

$x_{r+1}, x_{r+2} \in V$. Additional vertices, shown above the chain in Figure 4-9, ensure that the degree of each vertex in K_r is $r+2$, and the degrees of the vertices x_{r+1} and x_{r+2} are $r+3$ and $r+4$, respectively. Clique-chain graphs of other degrees are structured similarly.

Theorem 24 *For any $\Delta > 0$, there exists a Δ-degree graph $G = (V, E)$ such that* JP-LF *colors G in $\Omega(\Delta^2)$ span and* JP-R *colors G in $O(\ln V + \Delta \ln \Delta + \ln \Delta \lg V / \ln(e \ln V/\Delta))$ expected span.*

PROOF. Assume without loss of generality that 3 evenly divides Δ and that G is a clique-chain graph. The span of JP-R follows from Corollary 23. Because JP-LF trivially requires $\Omega(1)$ span to process each vertex in G, the span of JP-LF on G can be bounded by showing that the length of the longest path p in the priority dag G_ρ induced on G by any priority function ρ in LF is $\Delta^2/6 + \Delta/2 + 2$. Because LF assigns higher priority to higher-degree vertices, p starts at some vertex in $K_{\Delta-2}$, which has degree Δ, and passes through the $\Delta-2$ vertices in $K_{\Delta-2}$ followed by $x_{\Delta-3}$ and $x_{\Delta-4}$.[5] The remainder of p is a longest path through the clique-chain graph G' of degree $\Delta-3$ in the remaining graph $G - K_{\Delta-2} - \{x_{\Delta-3}, x_{\Delta-4}\}$, which has a longest path p' of length $|p'| = (\Delta-3)^2/6 + (\Delta-3)/2 + 2$ by induction. The length of p is thus $\Delta + |p'| = \Delta^2/6 + \Delta/2 + 2$. □

[5]Notice that it does not matter how ties are broken in the priority function.

The SL ordering heuristic

We focus on the formulation of the SL ordering heuristic due to Allwright et al. [4], because our experiments indicate that it gives colorings using fewer colors than other formulations [147].

Given a graph $G = (V, E)$, the SL ordering heuristic produces a priority function ρ via an iterative algorithm that assigns priorities to the vertices V in rounds to induce an ordering on V. For $i \geq 0$, let $G_i = (V_i, E_i)$ denote the subgraph of G remaining at the start of round i, and let δ_i denote an upper bound on the smallest degree of any vertex $v \in V_i$. Assume that $\delta_0 = 1$. At the start of round i, remove all vertices $v \in V_i$ such that $\deg(v) \leq \max\{\delta_{i-1}, \min_{v \in V_i}\{\deg(v)\}\}$. For a vertex v removed in round i, a priority function $\rho \in \text{SL}$ is defined as $\rho(v) = \langle i, \rho_{\text{R}}(v)\rangle$ where $\rho_{\text{R}} \in \text{R}$ is a random priority function.

The following theorem shows that there exist graphs for which JP-SL incurs a large span, whereas JP-R incurs only a small span.

Theorem 25 *There exists a class of graphs such that for any $G = (V, E, \rho)$ in the class and for any priority function $\rho \in \text{SL}$,* JP-SL *incurs $\Omega(V)$ span and* JP-R *incurs $O(\lg V / \lg\lg V)$ span.*

Proof. Consider the algorithm to compute the priority function ρ for all vertices in a path graph G. By induction over the rounds, the graph G_i at the start of round i is a path with $|V| - 2i + 2$ vertices, and in round i the 2 vertices at the endpoints of G_i will be removed. Hence $\lceil |V|/2 \rceil$ rounds are required to assign priorities for all vertices in G. A similar argument shows that the resulting priority dag G_ρ contains a path of length $|V|/2$ along which the priorities strictly decrease. JP-SL trivially incurs $\Omega(1)$ span through each vertex in the longest path in G_ρ. Since there are $\Theta(V)$ total vertices along the path and by Corollary 23 with $\Delta = \Theta(1)$, the theorem follows. $\square$

We shall see in Section 4.5 that it is possible to achieve coloring quality comparable to LF and SL, but with guaranteed parallel performance comparable to JP-R.

4.5 New ordering heuristics

This section describes the largest-log-degree-first (LLF) and smallest-log-degree-last (SLL) ordering heuristics. Given a Δ-degree graph G, we show that the expected depth of the priority dag G_ρ induced on G by a priority function $\rho \in$ LLF is $O\left(\min\{\Delta, \sqrt{E}\} + \ln\Delta \ln V / \ln(e \ln V/\Delta)\right)$. The same bound applies to the depth of a priority dag G_ρ induced on a graph G by a priority function $\rho \in$ SLL, though $O(\lg\Delta \lg V)$ additional span is required to calculate ρ using the method given in Figure 4-10. Combined with Theorem 13, these bounds imply that the expected span of JP-LLF is $O\left(\ln V + \ln\Delta \cdot \left(\min\{\sqrt{E}, \Delta\} + \ln\Delta \ln V / \ln(e \ln V/\Delta)\right)\right)$ and the expected span of JP-SLL is $O\left(\ln\Delta \ln V + \ln\Delta \left(\min\{\sqrt{E}, \Delta\} + \ln\Delta \ln V / \ln(e \ln V/\Delta)\right)\right)$.

The LLF ordering heuristic

The **LLF *ordering heuristic*** orders the vertices in decreasing order by the logarithm of their degree. More precisely, given a graph $G = (V, E, \rho)$ for some $\rho \in$ LLF, the priority of each $v \in V$ is equal to $\rho(v) = \langle \lceil \lg(\deg(v)) \rceil, \rho_{\mathrm{R}}(v) \rangle$, where $\rho_{\mathrm{R}} \in$ R is a random priority function and $\lg x$ denotes $\log_2 x$. [6] For a given graph G, the following theorem bounds the depth of the priority dag G_ρ induced by $\rho \in$ LLF.

Theorem 26 *Let $G = (V, E)$ be a Δ-degree graph, and let G_ρ be the priority dag induced on G by a priority function $\rho \in$ LLF. The expected length of the longest directed path in G_ρ is $O\left(\min\{\Delta, \sqrt{E}\} + \ln\Delta \ln V / \ln(e \ln V/\Delta)\right)$.*

Proof. Consider a length-k path $p = \langle v_1, \ldots, v_k \rangle$ in G_ρ. Let $G(\ell) \subseteq G_\rho$ be the subdag of G_ρ induced by those vertices $v \in V$ for which $\rho(v) = \lceil \lg(\deg(v)) \rceil = \ell$. Suppose that $v_i \in G(\ell)$ for some $v_i \in p$. Since $\lceil \lg(\deg(v_{i-1})) \rceil \geq \lceil \lg(\deg(v_i)) \rceil$ for all $i > 1$, we have $v_{i-1} \in G(\ell')$ for some $\ell' \geq \ell$. We can therefore decompose p into a sequence of paths $p = \left\langle p_{\lceil \lg\Delta \rceil}, \ldots, p_0 \right\rangle$ such that each subpath $p_\ell \in p$ is a path through $G(\ell)$. By definition of LLF, the subdag $G(\ell)$ is a dag induced on a graph with degree 2^ℓ by a random priority function.

By Corollary 23, the expected length of p_ℓ is $O(2^\ell + \ln V / \ln\left(e \ln V / 2^\ell\right))$. Linearity of expectation therefore implies that the expected length of the longest path in G is at most

$$\mathrm{E}\left[|p|\right] \leq \sum_{\ell=0}^{\lceil \lg\Delta \rceil} O\left(2^\ell + \frac{\ln V}{\ln(e \ln V / 2^\ell)}\right)$$

[6]The theoretical results in this section assume only that the base b of the logarithm is a constant. In practice, however, it is possible that the choice of b could have impact on the coloring quality or runtime of JP-LLF. We studied this trade-off and found that there is only a minor dependence on b. In general, the coloring quality and runtime of JP-LLF smoothly transitions from the behavior of JP-LF for small b and the behavior of JP-R for large b, sweeping out a Pareto-efficient frontier of reasonable choices. We chose $b = 2$ for our experiments, because $\log_2 x$ can be calculated conveniently by native instructions on modern architectures.

$$\begin{aligned}
&\le \sum_{\ell=0}^{\lceil \lg \Delta \rceil} O\left(2^\ell + \frac{\ln V}{\ln(e \ln V/\Delta)}\right) \\
&\le O\left(\Delta + \frac{\lg \Delta \ln V}{\ln(e \ln V/\Delta)}\right) \\
&\le O\left(\Delta + \frac{\ln \Delta \ln V}{\ln(e \ln V/\Delta)}\right) .
\end{aligned}$$

To establish the $\sqrt{E}$ bound, observe that at most $E/2^\ell$ vertices have degree at least 2^ℓ. Consequently, for $\ell > \lg \sqrt{E}$, the depth of $G(\ell)$ can be at most $E/2^\ell$. Hence we have

$$\begin{aligned}
\mathrm{E}\left[|p|\right] &\le \sum_{\ell=0}^{\lceil \lg \sqrt{E} \rceil} O\left(2^\ell\right) + \sum_{\ell=\lceil \lg \sqrt{E} \rceil}^{\infty} E/2^\ell + \sum_{\ell=0}^{\lceil \lg \Delta \rceil} O\left(\frac{\ln V}{\ln(e \ln V/\Delta)}\right) \\
&\le O\left(\sqrt{E} + \frac{\lg \Delta \ln V}{\ln(e \ln V/\Delta)}\right) \\
&\le O\left(\sqrt{E} + \frac{\ln \Delta \ln V}{\ln(e \ln V/\Delta)}\right) .
\end{aligned}$$

□

Corollary 27 *Given a graph $G = (V, E, \rho)$ for some $\rho \in$ LLF, JP-LLF colors all vertices in G with expected span $O\left(\ln V + \ln \Delta \cdot \left(\min\{\sqrt{E}, \Delta\} + \ln \Delta \ln V / \ln(e \ln V/\Delta)\right)\right)$.*

PROOF. The corollary follows from Theorem 13. □

The SLL ordering heuristic

To understand the **SLL *ordering heuristic***, it is convenient to consider in isolation how to compute its priority function. The pseudocode in Figure 4-10 for SLL-ASSIGN-PRIORITIES describes algorithmically how to perform this computation on a given graph $G = (V, E)$. As Figure 4-10 shows, a priority function $\rho \in$ SLL can be computed by iteratively removing low-degree vertices from G in rounds. The priority of a vertex $v \in V$ is the round number in which v is removed, with ties broken randomly. As with SL, SLL colors the vertices of G in the reverse order in which they are removed, but SLL-ASSIGN-PRIORITIES determines when to remove a vertex using a degree bound that grows exponentially. SLL-ASSIGN-PRIORITIES considers each degree bound for a maximum of r rounds. Effectively, a vertex is removed from G based on the logarithm of its degree in the remaining graph.

```
SLL-ASSIGN-PRIORITIES($G, r$)
231 let $G = (V, E)$
232 $i = 1$
233 $U = V$
234 let $\Delta$ be the degree of $G$
235 let $\rho_{\mathrm{R}} \in \mathrm{R}$ be a random priority function
236 for $d = 0$ to $\lg \Delta$
237     for $j = 1$ to $r$
238         $Q = \{u \in U : |N(u) \cap U| \le 2^d\}$
239         parallel for $v \in Q$
240             $\rho(v) = \langle i, \rho_{\mathrm{R}}(v) \rangle$
241         $U = U - Q$
242         $i = i + 1$
243 return $\rho$
```

Figure 4-10: Pseudocode for SLL-ASSIGN-PRIORITIES, which computes a priority function $\rho \in \mathrm{SLL}$ for the input graph. The input parameter r denotes the maximum number of times SLL-ASSIGN-PRIORITIES is permitted to remove vertices of at most a particular degree 2^d on lines 237–242.

We can formalize the behavior of SLL as follows. Given a graph G, let $G_i = (V_i, E_i)$ denote the subgraph of G remaining at the start of round i. As Figure 4-10 shows, for each $d \in \{0, 1, \ldots, \lg \Delta\}$, SLL-ASSIGN-PRIORITIES executes r rounds in which it removes vertices $v \in V_i$ such that $\deg(v) \le 2^d$ in G_i.[7]

For a given graph G, the following theorem bounds the depth of the priority dag G_ρ induced by a priority function $\rho \in \mathrm{SLL}$.

Theorem 28 *Let $G = (V, E)$ be a Δ-degree graph, and let G_ρ be the priority dag induced on G by a random priority function $\rho \in \mathrm{SLL}$. The expected length of the longest directed path in G_ρ is $O\left(\min\{\Delta, \sqrt{E}\} + \ln \Delta \ln V / \ln(e \ln V/\Delta)\right)$.*

PROOF. We begin with an argument similar to the proof of Theorem 26. Let $p = \langle v_1, \ldots, v_k \rangle$ be a length-k path in G_ρ, and let $G(\ell) \subseteq G_\rho$ be the subdag of G_ρ induced by those vertices $v \in V$, where $\rho(v) = \ell$. Since lines 237–242 of SLL-ASSIGN-PRIORITIES remove vertices with degree at most 2^d exactly r times for each $d \in [0, \ldots, \lg \Delta]$, we have that $\lfloor \rho(v)/r \rfloor = d$, and thus the degree of $G(\ell)$ is at most $2^{\lfloor \ell/r \rfloor}$. Suppose that $v_i \in G(\ell)$ for some $v_i \in p$.

[7]As with LLF, the degree cutoff 2^d on line 238 of Figure 4-10 could be b^d for an arbitrary constant base b with no harm to the theoretical results. We explored the choice of base empirically, but found that there was only a minor dependence on b. Generally, JP-SLL smoothly transitions from the behavior of JP-SL for small b to the behavior of JP-R and for large b. We therefore chose $b = 2$ for our experiments because of its implementation simplicity.

Since $\rho(v_{i-1}) \leq \rho(v_i)$ for all $i > 1$, we have $v_{i-1} \in G(\ell')$ for some $\ell' \geq \ell$. We can therefore decompose p into a sequence of paths $p = \langle p_{\lceil r \lg \Delta \rceil}, \ldots, p_0 \rangle$ where each $p_\ell \in p$ is a path in $G(\ell)$. By definition of SLL, the subdag $G(\ell)$ is a dag induced on a subgraph with degree at most $2^{\lfloor \ell/r \rfloor}$ by a random priority function.

By Corollary 23, the expected length of p_ℓ is $O\left(2^{\lfloor \ell/r \rfloor} + \ln V / \ln\left(e \ln V / 2^{\lfloor \ell/r \rfloor}\right)\right)$. Linearity of expectation therefore implies that

$$\begin{aligned}
\mathrm{E}\left[|p|\right] &= \sum_{\ell=0}^{\lceil r \lg \Delta \rceil} O\left(2^{\lfloor \ell/r \rfloor} + \frac{\ln V}{\ln\left(e \ln V / 2^{\lfloor \ell/r \rfloor}\right)}\right) \\
&\leq \sum_{\ell=0}^{\lceil r \lg \Delta \rceil} O\left(2^{\lfloor \ell/r \rfloor} + \frac{\ln V}{\ln\left(e \ln V / \Delta\right)}\right) \\
&\leq O\left(\Delta + \frac{\ln \Delta \ln V}{\ln\left(e \ln V / \Delta\right)}\right) \ .
\end{aligned}$$

Next, because at most $E/2^{\lfloor \ell/r \rfloor}$ vertices can have degree at least $2^{\lfloor \ell/r \rfloor}$, we have for $\ell > r \lg \sqrt{E}$ that the longest path through the subdag $G(\ell)$ is no longer than $E/2^{\lfloor \ell/r \rfloor}$. We thus conclude that

$$\begin{aligned}
\mathrm{E}\left[|p|\right] &\leq \sum_{\ell=0}^{\lceil r \lg \sqrt{E} \rceil} O\left(2^{\lfloor \ell/r \rfloor}\right) + \sum_{\ell=\lceil r \lg \sqrt{E} \rceil}^{\infty} E/2^{\lfloor \ell/r \rfloor} + \sum_{\ell=0}^{\lceil r \lg \Delta \rceil} O\left(\frac{\ln V}{\ln\left(e \ln V / 2^{\lfloor \ell/r \rfloor}\right)}\right) \\
\mathrm{E}\left[|p|\right] &\leq O\left(\sqrt{E}\right) + O\left(\sqrt{E}\right) + \sum_{\ell=0}^{\lceil r \lg \Delta \rceil} O\left(\frac{\ln V}{\ln\left(e \ln V / \Delta\right)}\right) \\
&\leq O\left(\sqrt{E} + \frac{\ln \Delta \ln V}{\ln\left(e \ln V / \Delta\right)}\right) \ .
\end{aligned}$$

□

Corollary 29 *Given a graph $G = (V, E, \rho)$ for some $\rho \in$ SLL, JP-SLL colors all vertices in G with expected span $O\left(\ln \Delta \ln V + \ln \Delta \left(\min\{\sqrt{E}, \Delta\} + \ln \Delta \ln V / \ln\left(e \ln V / \Delta\right)\right)\right)$.*

PROOF. The procedure SLL-ASSIGN-PRIORITIES calls the parallel loop on line 239 $O(\lg \Delta)$ times, each of which has expected span $O(\lg V)$. The proof then follows from Theorems 13 and 28. □

4.6 Empirical evaluation

This section evaluates the LLF and SLL ordering heuristics empirically using a suite of eight real-world and ten synthetic graphs. We describe the experimental setup used to evaluate JP-R, JP-LLF, and JP-SLL, and we compare their performance with Greedy-FF, Greedy-LF, and Greedy-SL. We compare the ordering heuristics in terms of the quality of the colorings they produce and their execution times. We conclude that LLF and SLL produce colorings with quality comparable to LF and SL, respectively, and that JP-LLF and JP-SLL scale well. We also show that the engineering quality of our implementations appears to be competitive with ColPack [81], a publicly available graph-coloring library. Our source code and data are available from `http://supertech.csail.mit.edu`.

Experimental setup

To evaluate the ordering heuristics, we implemented JP using Intel Cilk Plus [113] and engineered it to use the parallel ordering heuristics R, LLF, and SLL. To compare these parallel codes against their serial counterparts, we implemented Greedy in C to use the FF, LF, or SL ordering heuristics. In order to empirically evaluate the potential parallel performance of the serial ordering heuristics, we also engineered JP to use FF, LF, or SL. We evaluated our implementations on a dual-socket Intel Xeon X5650 with a total of 12 processor cores operating at 2.67-GHz (hyperthreading disabled); 49 GB of DRAM; 2 12-MB L3-caches, each shared between 6 cores; and private L2- and L1-caches with 128 KB and 32 KB, respectively. Each measurement was taken as the median of 7 independent trials, and the averages of those measurements reported in Tables 4-13 and 4-14 were taken across 5 independent random seeds.

These implementations were run on a suite of eight real-world graphs and ten synthetic graphs. The real-world graphs came from the Large Network Dataset Collection provided by Stanford's SNAP project [136]. The synthetic graphs consist of the adversarial graphs described in Section 4.4 and a set of graphs from three classes: constant degree, 3D grid, and "recursive matrix" (rMat) [40, 44]. The adversarial graphs — cliqueChain400 and path-10M — are described in Figure 4-9 with $\Delta = 400$ and Theorem 25 with $|V| = 10,000,000$, respectively. The constant-degree graphs — constant1M and constant500K — have 1M and 500K vertices and constant degrees of 100 and 200, respectively. These graphs were generated such that every pair of vertices is equally likely to be connected and every vertex has the

Graph	$\lvert V\rvert$	a	b	c	d
graph500-5M	5M	0.57	0.19	0.19	0.05
graph500-2M	2M	0.57	0.19	0.19	0.05
rMat-ER-2M	2M	0.25	0.25	0.25	0.25
rMat-G-2M	2M	0.45	0.15	0.15	0.25
rMat-B-2M	2M	0.55	0.15	0.15	0.15

Table 4-11: Parameters for the generation of rMat graphs [44], where $a+b+c+d=1$ and $b=c$, when the desired graph is undirected. An rMat graph is built by adding $|E|$ edges independently at random using the following rule: Let k be the number of 1's in a binary representation of i. As each edge is added, the probability that the ith vertex v_i is selected as an endpoint is $(a+c)^k(b+d)^{\lg n-k}$.

Graph	$\lvert E\rvert$	$\lvert E\rvert \,/\, \lvert V\rvert$	Δ
com-orkut	117.2M	38.1	33,313
liveJournal1	42.9M	8.8	20,333
europe-osm	36.0M	0.7	9
cit-Patents	16.5M	2.7	793
as-skitter	11.1M	1.0	35,455
wiki-Talk	4.7M	1.9	100,029
web-Google	4.3M	4.7	6,332
com-youtube	3.0M	2.6	28,754
constant1M	50.0M	50.0	100
constant500K	50.0M	99.9	200
graph500-5M	49.1M	5.9	121,495
graph500-2M	19.2M	9.2	70,718
rMat-ER-2M	20.0M	9.5	44
rMat-G-2M	20.0M	9.5	938
rMat-B-2M	19.8M	9.4	14,868
big3dgrid	29.8M	3.0	6
cliqueChain400	3.6M	132.4	400
path-10M	10.0M	1.0	2

Table 4-12: Number of edges, ratio of edges to vertices and maximum vertex degree for a collection of real-world and synthetic graphs, which lie above and below the center line, respectively.

same degree. The graph big3dgrid is a 3-dimensional grid on 10M vertices. The rMat graphs were generated using the parameters in Table 4-11.

			Greedy			JP			
Graph	H	C_H	T_S	H'	$C_{H'}$	T_1	T_{12}	T_S/T_1	T_1/T_{12}
com-orkut	FF	175	2.23	R	132	4.44	0.817	0.50	5.43
	LF	87	3.54	LLF	98	5.74	0.846	0.62	6.79
	SL	83	10.59	SLL	84	9.90	1.865	1.07	5.31
liveJournal1	FF	352	0.89	R	330	2.08	0.231	0.43	8.98
	LF	323	2.34	LLF	326	2.23	0.286	1.05	7.80
	SL	322	4.69	SLL	327	4.03	0.704	1.16	5.73
europe-osm	FF	5	1.32	R	5	4.04	0.391	0.33	10.34
	LF	4	17.15	LLF	4	4.93	0.473	3.48	10.41
	SL	3	19.87	SLL	3	7.28	1.232	2.73	5.91
cit-Patents	FF	17	0.50	R	21	1.08	0.163	0.46	6.67
	LF	14	2.00	LLF	14	1.46	0.160	1.37	9.11
	SL	13	3.21	SLL	14	2.90	0.519	1.11	5.58
as-skitter	FF	103	0.24	R	81	0.58	0.114	0.42	5.07
	LF	71	2.43	LLF	72	0.63	0.106	3.84	5.99
	SL	70	2.79	SLL	71	1.04	0.269	2.67	3.88
wiki-Talk	FF	102	0.09	R	85	0.28	0.053	0.31	5.28
	LF	72	0.49	LLF	70	0.34	0.050	1.43	6.78
	SL	56	0.61	SLL	62	0.55	0.124	1.12	4.43
web-Google	FF	44	0.09	R	44	0.21	0.029	0.44	7.44
	LF	45	0.25	LLF	44	0.27	0.030	0.94	8.92
	SL	44	0.47	SLL	44	0.50	0.093	0.94	5.44
com-youtube	FF	57	0.06	R	46	0.18	0.026	0.36	6.86
	LF	32	0.25	LLF	33	0.22	0.028	1.11	7.97
	SL	28	0.35	SLL	28	0.35	0.073	1.01	4.75

Table 4-13: Performance measurements for a set of real-world graphs taken from Stanford's SNAP project [136]. The column heading H denotes that the priority function used for the experiment in a particular row was produced by the ordering heuristic listed in the column. The average number of colors used by the corresponding ordering heuristic and graph is C_H. The time in seconds of Greedy, JP with 1 worker and with 12 workers is given by T_S, T_1 and T_{12}, respectively. Details of the experimental setup and graph suite can be found in Section 4.6.

Coloring quality of R, LLF, and SLL

Tables 4-13 and 4-14 present the coloring quality of the three parallel ordering heuristics R, LLF, and SLL alongside that of their serial counterparts FF, LF, and SL.

The number of colors used by LLF was comparable to that used by LF on the vast majority of the 18 graphs. Indeed, LLF produced colorings that were within 2 colors of LF on all synthetic graphs and all but 2 real-world graphs: com-orkut and liveJournal1.

			Greedy			JP			
Graph	H	C_H	T_S	H'	$C_{H'}$	T_1	T_{12}	T_S/T_1	T_1/T_{12}
constant1M	FF	33	0.90	R	32	1.93	0.255	0.47	7.55
	LF	32	1.16	LLF	32	2.70	0.323	0.43	8.35
	SL	34	2.96	SLL	32	4.63	0.610	0.64	7.59
constant500K	FF	52	0.74	R	52	1.50	0.190	0.49	7.89
	LF	52	0.84	LLF	52	2.01	0.273	0.42	7.34
	SL	53	1.97	SLL	52	3.33	0.498	0.59	6.69
graph500-5M	FF	220	1.83	R	220	2.99	0.558	0.61	5.35
	LF	159	3.69	LLF	160	3.74	0.542	0.99	6.89
	SL	158	8.43	SLL	162	7.63	1.056	1.10	7.23
graph500-2M	FF	206	0.52	R	208	1.01	0.212	0.51	4.77
	LF	153	0.98	LLF	154	1.24	0.151	0.79	8.19
	SL	153	2.22	SLL	156	2.25	0.324	0.99	6.94
rMat-ER-2M	FF	12	0.47	R	12	1.25	0.149	0.37	8.40
	LF	11	1.07	LLF	12	1.63	0.198	0.66	8.25
	SL	11	2.22	SLL	11	3.13	0.506	0.71	6.18
rMat-G-2M	FF	27	0.48	R	27	0.91	0.144	0.53	6.33
	LF	15	1.18	LLF	17	1.34	0.204	0.88	6.54
	SL	15	2.59	SLL	15	2.75	0.432	0.94	6.36
rMat-B-2M	FF	105	0.50	R	105	0.86	0.149	0.58	5.78
	LF	67	1.00	LLF	68	1.18	0.149	0.85	7.94
	SL	67	2.41	SLL	68	2.38	0.376	1.01	6.31
big3dgrid	FF	4	0.41	R	7	1.66	0.178	0.25	9.31
	LF	7	4.07	LLF	7	1.89	0.216	2.15	8.76
	SL	7	4.77	SLL	7	2.63	0.307	1.81	8.57
cliqueChain400	FF	399	0.05	R	399	0.09	0.012	0.50	7.77
	LF	399	0.05	LLF	399	0.12	0.015	0.41	7.70
	SL	399	0.08	SLL	399	0.16	0.024	0.47	6.70
path-10M	FF	2	0.18	R	3	0.85	0.074	0.21	11.54
	LF	3	2.49	LLF	3	0.98	0.083	2.54	11.87
	SL	2	2.58	SLL	3	1.36	0.169	1.90	8.04

Table 4-14: Performance measurements for five classes of synthetically generated graphs: constant degree, rMat, 3D grid, clique chain and path. The column headings are equivalent to those in Table 4-13.

Similarly, SLL produced colorings that were within 3 colors of SL for all synthetic graphs and all but 2 real-world graphs: liveJournal1 and wiki-Talk.

The liveJournal1 graph appears to benefit little from the ordering heuristics we considered. Every heuristic uses more than 300 colors, and the biggest difference between the number of colors used by any heuristic is less than 10.

The wiki-Talk and com-orkut graphs appear to benefit from ordering heuristics and illustrate what we believe is a coarse hierarchy of coloring quality in which FF $<$ R $<$ LLF $<$ LF $<$ SLL $<$ SL. On com-orkut, LLF produced a coloring of size 98, which was better than the 175 and 132 colors used by FF and R, respectively, but not as good as the 87 colors used by LF. In contrast, SLL nearly matched the superior coloring quality of SL, producing a coloring of size 84. On wiki-Talk, SLL produced a coloring of size 62, which was better than LF, LLF, R, and FF by a margin of between 8 to 40 colors, but not as good as SL, which used only 56 colors. These trends appear to exist, in general, for most of the graphs in the suite.

Scalability of JP-R, JP-LLF, and JP-SLL

The parallel performance of JP was measured by computing the speedup it achieved on 12 cores and by comparing the 1-core runtimes of JP to an optimized serial implementation of Greedy. These results are summarized in Tables 4-13 and 4-14.

Overall, JP-LLF obtains a geometric-mean speedup — the ratio of the runtime on 1 core to the runtime on 12 cores — of 7.83 on the eight real-world graphs and 8.08 on the ten synthetic graphs. Similarly, JP-SLL obtains a geometric-mean speedup of 5.36 and 7.02 on the real-world and synthetic graphs, respectively.

Tables 4-13 and 4-14 also include scalability data for JP-FF, JP-LF, and JP-SL. Historically, JP-LF has been used with mixed success in practical parallel settings [4,85,118,175]. Despite the fact that it offers little in terms of theoretical parallel performance guarantees, we have measured its parallel performance for our graph suite, and indeed JP-LF scales reasonably well: $\text{JP-LF}_1/\text{JP-LF}_{12} = 6.8$ as compared to $\text{JP-LLF}_1/\text{JP-LLF}_{12} = 8.0$ in geometric mean, not including cliqueChain400, which is omitted since JP-LF crashes due to excessive stack usage on cliqueChain400. The omission of cliqueChain400 highlights the dangers of using algorithms without good performance guarantees: it is difficult to know if the algorithm will behave badly given any particular input. In this respect, JP-FF is particularly vulnerable to adversarial inputs, as we can see by the fact that it crashes on europe-osm, which is not even intentionally adversarial. We also see this vulnerability with JP-SL, as well as generally poor scalability on the entire suite.

To measure the overheads introduced by using a parallel algorithm, the runtime T_1 of JP on 1 core was compared with the runtime T_S of an optimized implementation of GREEDY. This comparison was performed for each of the three parallel ordering heuristics we considered: R, LLF, and SLL. The serial runtime of GREEDY using FF is 2.5 times faster than JP-R on 1 core for the eight real-world graphs and 2.3 times faster on the ten synthetic graphs. We conjecture that GREEDY gains its advantage due to the spatial-locality advantage that results from processing the vertices in the linear order they appear in the graph representation. JP-LLF and JP-SLL on 1 core, however, are actually faster than GREEDY with LF and SL by 43.3% and 19% on the eight real-world graphs and 6% and 3% on the whole suite, respectively.

In order to validate that our implementation of GREEDY is a credible baseline, we compared it with a publicly available graph-coloring library, COLPACK [81], developed by Gebremedhin et al. and found that the two implementations appeared to achieve similar performance. For example, using the SL ordering heuristic, GREEDY is 19% faster than COLPACK in geometric-mean across the graph suite, though GREEDY is slower on 5 of the 16 graphs and as much 2.22 times slower for as-skitter.

4.7 Implementation techniques

This section describes the techniques we employed to implement JP and GREEDY for the evaluation in Section 4.6. We describe three techniques — join-trees [68], bit-vectors, and software prefetching — that improve the practical performance of JP. Where applicable, these same techniques were used to optimize the implementation of GREEDY. Overall, applying these techniques yielded a speedup of between 1.6 and 2.9 for JP and a speedup of between 1.2 and 1.6 for GREEDY on the rMat-G-2M, rMat-B-2M, web-Google, and as-skitter graphs used in Section 4.6.

Join trees for reducing memory contention

Although the theoretical analysis of JP in Section 4.2 does not concern itself with contention, the implementation of JP works to mitigate overheads due to contention. The pseudocode for JP in Figure 4-4 shows that each vertex u in the graph has an associated counter $u.counter$.

```
GREEDY-SD($G$)
244 let $G = (V, E)$
245 for $v \in V$
246     $v.adjColors = \emptyset$
247     $v.adjUncolored = N(v)$
248     PUSHORADDKEY($v, Q[0][|v.adjUncolored|]$)
249 $s = 0$
250 while $s \geq 0$
251     $v =$ POPORDELKEY($Q[s][\max$ KEYS$(Q[s])]$)
252     $v.color = \min(\{1, 2, \ldots, |v.adjUncolored| + 1\} - v.adjColors)$
253     for $u \in v.adjUncolored$
254         REMOVEORDELKEY($u, Q[|u.adjColors|][|u.adjUncolored|]$)
255         $u.adjColors = u.adjColors \cup \{v.color\}$
256         $u.adjUncolored = u.adjUncolored - \{v\}$
257         PUSHORADDKEY($u, Q[|u.adjColors|][|u.adjUncolored|]$)
258         $s = \max\{s, |u.adjColors|\}$
259     while $s \geq 0$ and $Q[s] == \emptyset$
260         $s = s - 1$
```

Figure 4-15: The GREEDY-SD algorithm computes a coloring for the input graph $G = (V, E)$ using the SD heuristic. Each uncolored vertex $v \in V$ maintains a set $v.adjColors$ of colors used by its neighbors and a set $v.adjUncolored$ of uncolored neighbors of v. The PUSHORADDKEY method adds a specified key, if necessary, and then adds an element to that key's associated set. The POPORDELKEY and REMOVEORDELKEY methods remove an element from a specified key's associated set, deleting that key if the set becomes empty. The variable s maintains the maximum saturation degree of G.

Line 225 of JP-COLOR executes a JOIN operation on $u.counter$. Although Section 4.2 describes how JOIN can treat $u.counter$ as a join counter [51] and update $u.counter$ using an atomic decrement and fetch operation, the cache-coherence protocol [163] on the machine serializes such atomic operations, giving rise to potential memory contention. In particular, memory contention may harm the practical performance of JP on graphs with large-degree vertices.

Our implementation of JP mitigates overheads due to contention by replacing each join counter $u.counter$ with a join tree having $\Theta(|u.pred|)$ leaves. In particular, each join tree was sized such that an average of 64 predecessors of u map to each leaf through a hash function that maps predecessors to random leaves. We found that the join tree reduces T_1 for JP by a factor of 1.15 and reduces T_{12} for JP by between 1.1 and 1.3.

Bit vectors for assigning colors

To color vertices more efficiently, the implementation of JP uses vertex-local bit vectors to store information about the availability of low-numbered colors. Because JP assigns to each vertex the lowest-numbered available color, vertices tend to be colored with low-numbered colors. To take advantage of this observation, we store a 64-bit word per vertex u to track the colors in the range $\{1, 2, \ldots, 64\}$ that have already been assigned to a neighbor of u. The bit vector on $u.vec$ is computed as a "self-timed" OR reduction that occurs during updates on u's join tree. Effectively, as each predecessor v of u executes JOIN on u's join tree, if $v.color$ is in $\{1, 2, \ldots, 64\}$, then v OR's the word $2^{v.color-1}$ into $u.vec$. When GET-COLOR(u) subsequently executes, GET-COLOR first scans for the lowest unset bit in $u.vec$ to find the minimum color in $\{1, 2, \ldots, 64\}$ not assigned to a neighbor of u. Only when no such color is available does GET-COLOR(u) scan its predecessors to assign a color to u.

We discovered that a large fraction of vertices in a graph can be colored efficiently using this practical optimization. We found that this optimization improved T_{12} for JP by a factor of 1.4 to 2.2, and a similar optimization sped up the implementation of GREEDY by a factor of 1.2 to 1.6.

Software prefetching

We used software prefetching to improve the latency of memory accesses in JP. In particular, JP uses software prefetching to mitigate the latency of the indirect memory access encountered when accessing the join trees of the successors of a vertex v on line 224 of JP-COLOR in Figure 4-4. This optimization improves T_{12} for JP by a factor of 1.2 to 1.5.

Interestingly, our implementation of GREEDY did not appear to benefit from using software prefetching in a similar context, specifically, to access the predecessors of a vertex on line 212 of GREEDY in Figure 4-1. We suspect that because GREEDY only reads the predecessors of a vertex on this line and does not write them, the processor hardware is able to generate many such reads in parallel, thereby mitigating the latency penalty introduced by cache misses.

4.8 The saturation-degree heuristic

Our experiments with serial heuristics detailed in Tables 4-2 and 4-3 indicate that the SD heuristic tends to provide colorings with higher quality than the other heuristics we have considered, confirming similar findings by Gebremedhin and Manne [80]. Although we leave the problem of devising a good parallel algorithm for SD as an open question, we were able to devise a linear-time serial algorithm for the problem, despite conjectures in the literature [50, 85] that superlinear time is required. This section briefly describes our linear-time serial algorithm for SD.

Figure 4-15 gives pseudocode for the GREEDY-SD algorithm, which implements the SD heuristic. Rather than trying to define a priority function for SD, the figure gives the coloring algorithm GREEDY-SD itself, since the calculation of such a priority function would color the graph as a byproduct. At any moment during the execution of the algorithm, the ***saturation degree*** of a vertex v as the number $|v.adjColors|$ of distinct colors of v's neighbors, and the ***effective degree*** of v as $|v.adjUncolored|$, its degree in the as yet uncolored graph.

The main loop of GREEDY-SD (lines 250–260) first removes a vertex v of maximum saturation degree from Q (line 251) and colors it (line 252). It then updates each uncolored neighbor $u \in v.adjUncolored$ of v (lines 253–258) in three steps. First, it removes u from Q (line 254). Next, it updates the set $u.adjUncolored$ of u's ***effective neighbors*** — u's uncolored neighbors in G — and the set $u.adjColors$ of colors used by u's neighbors (lines 255–256). Finally, it enqueues u in Q based on u's updated information (lines 257–258).

The crux of GREEDY-SD lies in the operation of the queue data structure Q, which is organized as an array of ***saturation tables***, each of which supports the three methods PUSHORADDKEY, POPORDELKEY, and REMOVEORDELKEY described in the caption of Figure 4-15. A saturation table can support these operations in $\Theta(1)$ time and allow its keys K to be read in $\Theta(K)$ time. At the start of each main loop iteration, entry $Q[i]$ stores the uncolored vertices in the graph with saturation degree i in a saturation table. The PUSHORADDKEY, POPORDELKEY, and REMOVEORDELKEY methods maintain the invariant that, for each table $Q[i]$, each key $j \in$ KEYS$(Q[i])$ is associated with a nonempty set of vertices, such that each vertex $v \in Q[i][j]$ has saturation degree i and effective degree j.

Theorem 30 GREEDY-SD *colors a graph* $G = (V, E)$ *according to the* SD *ordering heuristic in* $\Theta(V + E)$ *time.*

Proof. PushOrAddKey, PopOrDelKey, and RemoveOrDelKey operate in $\Theta(1)$ time, and a given saturation table's key set K can be read in $\Theta(K)$ time. Line 251 can thus find a vertex v with maximum saturation degree s in $\Theta(|\text{Keys}(Q[s])|)$ time. Line 252 can color v in $\Theta(\deg(v))$ time, and lines 258–260 maintain s in $\Theta(s)$ time. Because $s + |\text{Keys}(Q[s])| \leq \deg(v)$, lines 250–260 evaluate v in $\Theta(\deg(v))$ time. The handshaking lemma [52, p. 1172–3] implies the theorem, because each vertex in V is evaluated once. □

4.9 Related work

Parallel coloring algorithms have been explored extensively in the distributed computing domain [5,11,86,87,118,126,127,140]. These algorithms are evaluated in the message-passing model, where nodes are allowed unlimited local computation and exchange messages through a sequence of synchronized rounds. Kuhn [126] and Barenboim and Elkin [11] independently developed $O(\Delta+\lg^* n)$-round message passing algorithms to compute a deterministic greedy coloring.

Several greedy coloring algorithms have been described in synchronous PRAM models. Goldberg, Plotkin, and Shannon [86] describe an algorithm for finding a greedy coloring of $O(1)$-degree graphs in $O(\lg n)$ time in the EREW PRAM model using a linear number of processors. They observe that their technique can be applied recursively to color Δ-degree graphs in $O(\Delta \lg \Delta \lg n)$ time. Their strategy incurs $\Omega(\lg \Delta(V + E))$ (superlinear) work, however.

Çatalyürek et al. [40] present the algorithm Iterative, which first speculatively colors a graph G and then fixes coloring conflicts, that is, corrects the coloring where two adjacent vertices are assigned the same color. The process of fixing conflicting colors can introduce new conflicts, though the authors observe empirically that comparatively few iterations suffice to find a valid coloring. We ran Iterative on our test system and found that JP-LLF uses 13% fewer colors and takes 19% less time in geometric mean of number of colors and relative time, respectively, over all graphs in our test suite. Furthermore, we found that JP-SLL uses 17% fewer colors, but executes in twice the time of Iterative. We do not know the extent to which the optimizations enjoyed by our algorithms could be adopted by speculative-coloring algorithms, however, and so it is likely too soon to draw conclusions about comparisons between the strategies.

4.10 Conclusion

Because of the importance of graph coloring, considerable effort has been invested over the years to develop ordering heuristics for serial graph-coloring algorithms. For the traditional "serial" LF and SL ordering heuristics, we have developed "parallel" analogues — the LLF and SLL heuristics, respectively — which approximate the traditional orderings, generating colorings of comparable quality while offering provable guarantees on parallel scalability. The correspondence between serial ordering heuristics and their parallel analogues is fairly direct for LF and LLF . LLF colors any two vertices whose degrees differ by more than a factor of 2 in the same order as LF. In this sense, LLF can be viewed as a simple coarsening of the vertex ordering used by LF. Although SLL is inspired by SL, and both heuristics tend to color vertices of smaller degree later, the correspondence between SL and SLL is not as straightforward. We relied on empirical results to determine the degree to which SLL captures the salient properties of SL.

We had hoped that the coarsening strategy LLF and SLL embody would generalize to the other serial ordering heuristics, and we are disappointed that we have not yet been able to devise parallel analogues for the other ordering heuristics, and in particular, for SD. Because the SD heuristic appears to produce better colorings in practice than all of the other serial ordering heuristics, SD appears to capture an important phenomenon that the others miss.

The problem with applying the coarsening strategy to SD stems from the way that SD is defined. Because SD determines the order to color vertices while serially coloring the graph itself, it seems difficult to parallelize, and it is not clear how SD might correspond to a possible parallel analogue. Thus, it remains an intriguing open question as to whether a parallel ordering heuristic exists that captures the same "insights" as SD while offering provable guarantees on scalability.

SD has a high running time, if only a constant factor higher than optimal. One way to potentially improve the overall running time without compromising the coloring quality of SD is to exploit the "deferred coloring property:" If $G = (V, E, \rho)$ is a graph colored with χ colors when ordered according to the priority function ρ, then $G = (V, E, \rho')$ is colored with at most χ colors, where $V_\chi = \{v \in V : |N(v)| < \chi\}$ and $\rho'(v) = \langle \textsc{IsMember}(v, V_\chi), \rho(v) \rangle$. That is, if a vertex with less than χ neighbors is assigned a color, it can never be assigned a color larger than χ. We could exploit the deferred coloring property by successively removing sets of vertices with degree less than a specified value, until the χ-core remains [117]. The resulting χ-core may be colored using SD at a reduced cost, since there are fewer remaining vertices. Finally, the low-degree vertices may be cheaply colored in the reverse order of their removal, using a greedy algorithm. An open question remains: under what circumstances or for what inputs is this a parallel algorithm with provably good span?

Chapter 5

Cache-Efficient Data-Graph Computations for Physical Simulations

5.1 Introduction

Scientists, governments, and companies are wrestling with the "other" Moore's Law: the amount of data in the world doubles every two years [194]. Untold treasures lie within this data and thus a battalion of data wonks are marching on Insightsville. One such type of big data problem is the space of physical simulations (e.g., fluid dynamics [17], the n-body problem [168, 183, 196], computer graphics, etc.). We investigate the specific problem of performing physical simulations on large datasets quickly and deterministically. Figure 5-1 depicts a typical dataset: a mesh graph, where vertices have positions in 3D space and edges connect physically nearby vertices. A function, typically some approximation of physical forces (e.g., Newton's laws of motion [157]), that operates on each vertex and its neighbors is applied to all vertices over many (e.g., at least millions of) time steps. This research introduces a software platform called LAIKA that performs these physical simulations in a fast and provably scalable manner on a shared-memory multicore computer.

In recent years, there has been growing interest in developing frameworks for the storage and analysis of data on large compute clusters, Hadoop [55,59] being among among the most popular of these. Hadoop distributes data across many shared-memory multicore nodes,

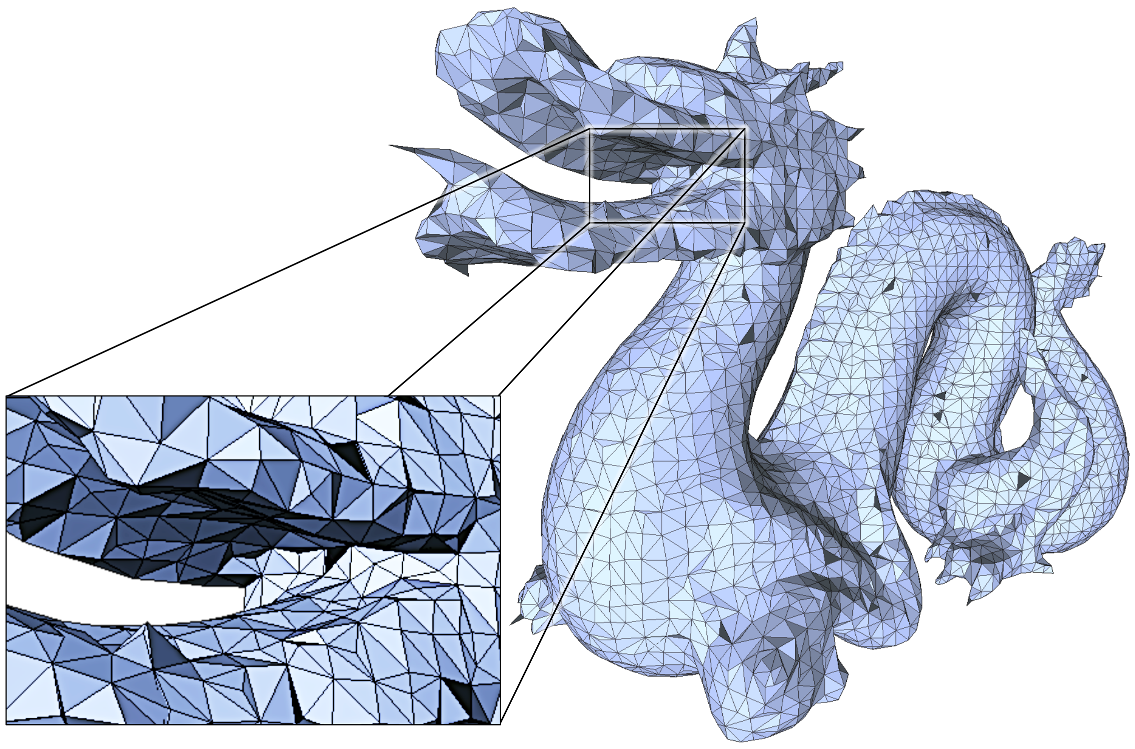

Figure 5-1: The Stanford dragon [1] mesh graph where lines correspond to edges and intersections of lines correspond to vertices.

which communicates via a message-based network protocol. Users supply ***map*** operators, that operate on each data item independently, and ***reduce*** operators, that combine the results. Many problems can be cast into the Hadoop model, but in many cases the Hadoop approach is less effecient than more specialized methods optimized for graph algorithms.

The idea behind recent big data frameworks, like Hadoop, is to decouple scheduling and data layout from the expression of the computation, enabling high programmer productivity and portable, best-in-class performance. Iterative graph algorithms, however, are one class of problem that is not well-suited to the Hadoop approach. For example, graphs are difficult to split into completely independent sets (i.e., with no crossing edges) for the map phase of a Hadoop computation, so the maps are often wasteful. Nonetheless, the idea of decoupling scheduling and data layout from the expression of the algorithm is useful for designing frameworks for graph algorithms, even if Hadoop itself is ill-suited to the task.

Data-graph computations

In response to the shortcomings of applying Hadoop and similar systems to graph problems, Low et al. developed the GraphLab framework [142, 143] for iterative graph algorithms, primarily targeting machine learning algorithms. In particular, GraphLab is a framework for implementing a ***data-graph computation***, which consists of a graph $G = (V, E)$, where each vertex has associated user-specified data, and a user-specified ***update function*** which is applied to every vertex, taking as inputs the data associated with the associated neighbors. On each *round* or *time step* the update function is applied to some subset of, or all, vertices. Many interesting big data algorithms, including Google's PageRank, can be conveniently expressed under this model.

Chromatic scheduling

Recently, Kaler et al. [121] demonstrated that general data-graph computations could be made to be deterministic without giving up high-performance, in fact, while increasing performance. Their system PRISM is 1.2-2.1 times faster than the nondeterministic GraphLab framework on a representative suite of data-graph computations in a multicore setting. The algorithm Kaler et al. proposed is called PRISM, which uses a well-known technique, ***chromatic scheduling*** [2, 18, 142].

In chromatic scheduling one finds a valid coloring of the graph as depicted in Figure 1-1, an assignment of colors to vertices such that no two neighboring vertices share the same color,[1] and then serializes through the colors. Since each subset of the graph of a given color is an independent set (i.e., no two members share an edge) they may be processed simultaneously without causing a data race. This strategy assumes that the update function applied to a vertex v reads the data associated with all of its neighbors $N(v) = \{w \in V | (v, w) \in E\}$ and writes only the data associated with v. Chromatic scheduling is a powerful technique because it allows the parallel execution of a data-graph computation without any concurrent operations on data, removing the overhead of mutual-exclusion locks incurred by GraphLab or other atomic operations that would be required in a design with concurrency.

While chromatic scheduling does a good job of enabling high parallelism without any concurrency, it can be inefficient for cache usage. For instance, in order to process the update function of a vertex v of color c, the worker needs to read data associated with all of its neighbors, denoted $N(v)$. However, by virtue of being in different color sets, by definition, each vertex $w \in N(v)$ can not be processed until after all vertices of color c

[1]We define a "neighbor" of a vertex $v \in V$ in a graph $G = (V, E)$ to be any $w \in V$ such that $(v, w) \in E$ or $(w, v) \in E$.

have been processed, potentially squandering the potential cache advantage of processing the neighbors of v soon after v itself is processed, while they are still in cache.

Dag scheduling

An alternative approach to chromatic scheduling is ***dag scheduling*** [118], depicted in Figure 1-6 and used extensively by Hasenplaugh et al. [102] in the context of graph coloring. In dag scheduling, the graph is turned into a dag through the use of a priority function $\rho : V \to \mathbb{R}$. In particular, an undirected edge connecting vertices v and w is oriented as (v, w) if $\rho(v) < \rho(w)$ (ties can be broken randomly or by comparing vertex numbers). The vertices are then processed in dag order, meaning that a vertex v may be processed only once all of its predecessors $v.pred = \{w \in N(v) : \rho(w) < \rho(v)\}$ have been processed. A relatively simple implementation of dag scheduling is called JP, described in Figure 5-2. A counter at each vertex is initialized with the number of predecessors in the dag. Then, after a vertex v is updated the worker atomically decrements the counters for all successors $v.succ = N(v) \setminus v.pred$, recursively updating any successors whose counters drop to zero. This scheduling approach affords us the opportunity to process vertices shortly after they are read by their neighbors, a potential caching advantage. We will explore a technique for achieving such cache behavior for a special class of mesh graphs used in physical simulations in Section 5.3.

Simit: A domain-specific language for specifying physical simulations

Simit [124] is a language used by scientists to describe physical simulations (e.g., fluid dynamics, the n-body problem, computer graphics, etc.) and used by us to motivate the scheduling of such simulations. Simit generates a mesh graph (i.e., a wire mesh discretization of a continuous 3D object) of an object in physical 3D space, like the one depicted in Figure 5-1. These meshes can contain vertices at intersections of line segments, hyperedges (e.g., triangular faces), and tetrahedron volumes, each of which has associated data. An immediate hurdle presents itself when trying to cast operations on such a mesh graph as a data-graph computation: data-graphs do not natively support hyperedges or tetrahedra. Simit is a programming language, and thus the compiler can intervene to represent the mesh graph as a data-graph where each vertex has a ***type***.

```
JP($G$)
261 let $G = (V, E, \rho)$
262 parallel for $v \in V$
263     $v.pred = \{u \in N(v) : \rho(u) > \rho(v)\}$
264     $v.succ = \{u \in N(v) : \rho(u) < \rho(v)\}$
265     $v.counter = |v.pred|$
266 parallel for $v \in V$
267     if $v.pred == \emptyset$
268         JP-UPDATE($v$)
```

```
JP-UPDATE($v$)
269 UPDATE($v$)
270 parallel for $u \in v.succ$
271     if JOIN($u.counter$) == 0
272         JP-UPDATE($u$)
```

Figure 5-2: The Jones-Plassmann [118] parallel priority-dag scheduling algorithm, shown here as JP, is a generalization of Jones and Plassmann's original distributed vertex-coloring algorithm. JP uses a recursive helper function JP-UPDATE to process a vertex using the user-supplied UPDATE function once all of its predecessors have been updated, recursively calling JP-UPDATE in line 59 for any successor u who is eligible to be updated (i.e., when $u.counter == 0$). The function JOIN decrements its argument and returns the post-decrement value.

We see in Figure 5-3 how a face (or hyperedge) connecting vertices B, D, and E, for example, can be represented as a new type of vertex (i.e., the blue squares in the figure) which is conntected to B, D, and E by individual edges. In addition, a tetrahedron can be viewed as a set of four adjacent triangular faces (or hyperedges). In Figure 5-4 we see such an example, where a third type of vertex (i.e., the purple diamond in the figure) is connected to four face type vertices. Finally, the update function, which is generated by the Simit compiler, can generate an update function which takes the type of the vertex as a parameter and jumps to the relevant code, as in a `case` statement.

The Hilbert space-filling curve

We propose a new priority function for use with dag scheduling of data-graph computations for the special case of physical simulations, like those generated by Simit. This priority function is used to reorder the vertices in the graph to exploit improved cache behavior as we explore in Section 5.2. In particular, we use the bounding box of the graph in 3D space to normalize the graph to the unit cube. Then, we decompose the unit cube into a regular $2^k \times 2^k \times 2^k$ grid, each grid point of which is assigned a scalar value by the Hilbert space-

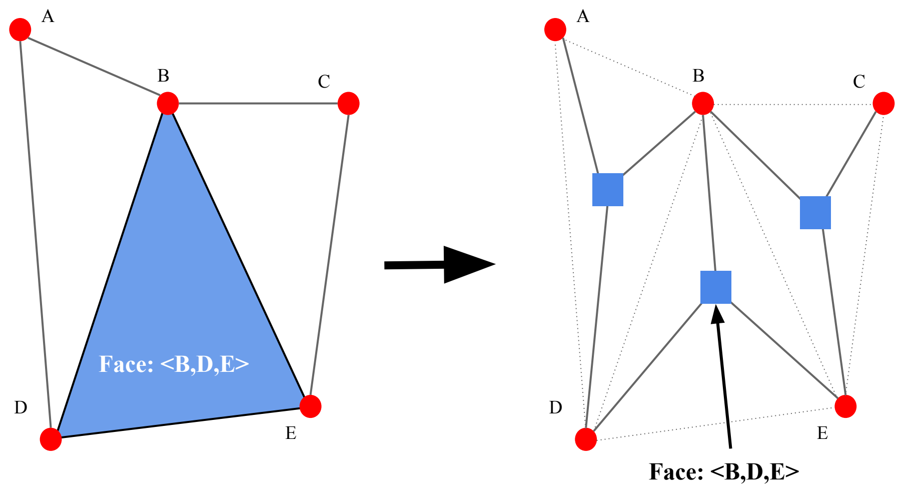

Figure 5-3: Graphs generated by the language Simit feature hyperedges, an example of which is in blue on the left. Hyperedges are represented by different types of vertices in the resulting data-graph computation. The square vertices in the figure represent hyperedges and have associated per-hyperedge data.

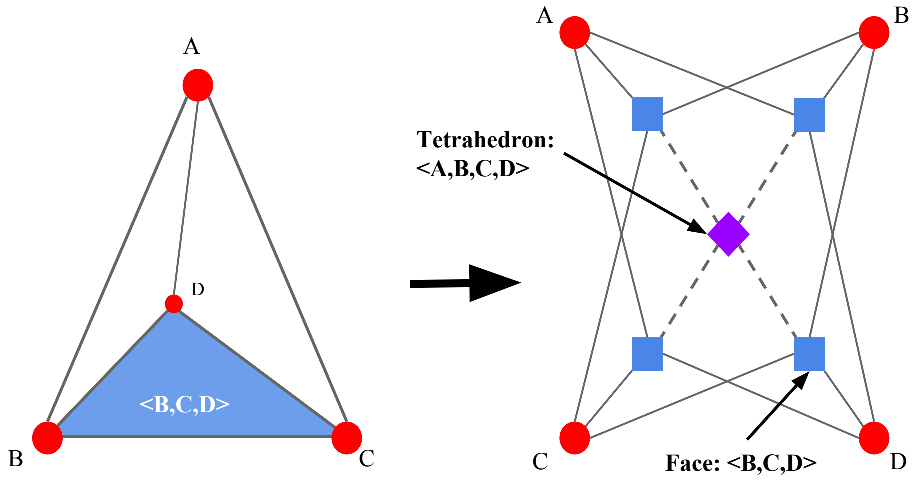

Figure 5-4: Graphs generated by the language Simit have tetrahedrons, as depicted on the left above. A tetrahedron is composed of four hyperedges (or faces), an example of which is in blue on the left. Tetrahedra are represented by different types of vertices in the resulting data-graph computation. The diamond vertex on the right represents a tetrahedron and is connected to its four constituent hyperedges.

filling curve [106]. A 2D example of the Hilbert space-filling curve is given in Figure 1-8. The red dotted curve is the first recursion level and illustrates the basic inverted “U” shape. The blue dashed curve shows how each quadrant is partitioned into four independent first-level Hilbert curves (up to rotations) of half the size in each dimension. The black solid curve illustrates the third recursion level. All vertices are assigned to the closest grid point and assigned the corresponding the scalar value along the Hilbert curve, as depicted in Figure 5-5. This scalar value is known as a point's value in ***Hilbert curve space***. Since some vertices may be assigned to the same grid point, ties are broken in the ordering randomly. Thus, the vertices are processed in the order dictated by the Hilbert curve iterating through the 3D grid. We call the priority-dag scheduling algorithm, using the Hilbert curve priority function, Laika.

The Hilbert curve has another convenient property that we can exploit toward the goal of partitioning a mesh graph to improve cache usage. That is, a subinterval in Hilbert curve space corresponds to a compact subspace in 3D space which has a low surface area to volume ratio [166,183,196]. Since mesh graphs are locally connected, we would then expect that the relative number of edges crossing from one such subspace to another would be low [152]. Thus, to distribute the computation among P different workers in a multicore system, we merely split the vertices, presorted on the Hilbert priority function, evenly in P chunks while exposing relatively few inter-processor edges. The use of space-filling curves for locality-preserving load-balancing is a known technique. Algorithms for the n-body problem [183, 196], database layout and scheduling [152], resource scheduling [138], and dynamic load balancing [101] all use variations on the general theme of mapping ND space onto a 1D curve that is subsequently partitioned among P processors. We extend this field by offering an improved analysis of the relationship between distance in Hilbert curve space and distance in memory, when vertices are ordered by the Hilbert priority function.

Chapter organization

This chapter represents a collaborative effort with Predrag Gruevski, Charles E. Leiserson, and James J. Thomas and also exists in a standalone unpublished manunscript called "Cache-efficient data-graph computations for physical simulations" [96].

We will explore the theoretical and experimental cache behavior of Hilbert-ordered data-graph computations of locally-connected mesh graphs in Section 5.2. In Section 5.3 we will describe a new scheduling algorithm Laika which exploits this cache advantage and test its performance in Section 5.6 using an example physical simulation described in Section 5.5.

5.2 Reordering Vertices for Cache Locality

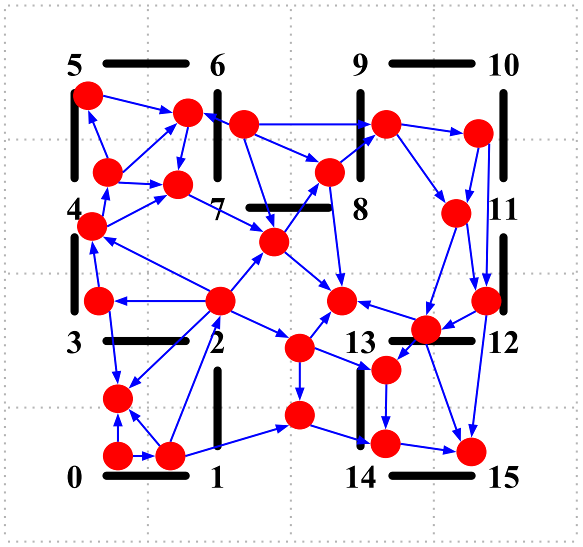

Figure 5-5: Example of how a locally-connected graph in 2D is mapped to a dag via a second-order Hilbert priority function. Each vertex is mapped to its closest grid point in the discretized Hilbert curve. Among vertices mapping to the same Hilbert grid point, ties are broken randomly.

In this section, we describe the rationale behind and empirical evidence supporting our use of the Hilbert space-filling curve as a way of mapping a 3D space onto the real line. Specifically, we use this mapping to reorder the vertices of a 3D, locally-connected mesh graph to gain cache locality. We provide a theoretical analysis which shows that a random cube graph — a specific type of mesh graph — reordered in this fashion exhibits good cache behavior. We then show empirically that a random cube graph is an adequate proxy for a mesh graph that is generated using a tetrahedralization of a surface mesh,[2] which is the typical method in the space of physical simulations. For instance, we used a popular open-source program `TetGen` [181, 182] to generate the tetrahedral mesh graphs in our test suite, described in Section 5.6.

Figure 5-5 illustrates a set of points on the unit square overlaid with a second-order 2D Hilbert curve; the priority function and Hilbert curve both extend naturally to 3 dimensions. The ***Hilbert priority function***, $\rho_H : V \to \mathbb{R}$, for a vertex is equal to the value along the closest Hilbert curve grid point, breaking ties between vertices nearest to the same Hilbert curve grid point at random. The Hilbert priority function takes a parameter k which indicates

[2]There are many strategies for tetrahedralizing a surface mesh, which means tesselating the volume encased by the surface mesh with tetrahedra. The tetrahedra are then subdivided into additional smaller tetrahedra until the desired level of granularity is achieved, a process called "mesh refinement."

the order of the Hilbert curve recursion, where one thinks of the curve dividing up a cubic space into $2^k \times 2^k \times 2^k$ blocks. This priority function is used to sort the vertices in input graphs that are known to be locally-connected. We pick k such that $2^{3k} = O(n)$ and thus we can sort the vertices in linear time using a counting sort [52, Ch. 8].[3]

We analyze the canonical ***traversal*** of a mesh graph to approximate the cache behavior of a single iteration of a static data-graph computation on the same graph. In particular, we stream through the vertices in sequence while keeping a cache of nearby vertices, and we measure the number of neighbors that lie outside the cache. For example, during the course of a traversal with a cache of size M vertices, we arrive at vertex number v, and the cache is stocked with all vertices in the range $[v - M/2, v + M/2 - 1]$. Any neighbor of v outside that range would incur a cache miss. In order to simulate a traversal of the graphs in our suite, we measured the absolute difference $|v - w|$ for every neighboring pair v and w of vertices, where we abuse the notation v to mean both the vertex v and v's location within the vertex array. We used the cumulative distribution of the pairwise differences to simulate the miss rate per edge for all cache sizes of size M. We can see in Figure 5-6 that reordering the vertices according to the Hilbert priority function depicted in Figure 5-5 yields excellent cache behavior. For instance, when using a cache of 2048 vertices — roughly the size of the L2 cache of modern Intel processors — less than 13% of the neighbors lie outside the L2 cache.

The gray line in Figure 5-6 is an upper bound on the cache miss rate resulting from an analysis due to Tirthapura, Seal, and Aluru [192]. They analyze a generic ***recursively defined space-filling curve***, which is any partitioning of a 3D unit cube that recursively divides the cube into 8 ordered, equal-sized subvolumes, specifically, those that would result from centering the cube at the origin and separating the portions that lie in each octant. The leaves of such a partitioning may then be lexicographically compared by appending the respective orders from each recursion level. Tirthapura, Seal, and Aluru leave as an open problem the challenge of analyzing specific space-filling curves to achieve tighter bounds.

[3]For problems generated by Simit and many other physical simulations, there are generally sufficiently many time steps that the cost of any preprocessing, even $O(n \log n)$-time sorting, is completely amortized.

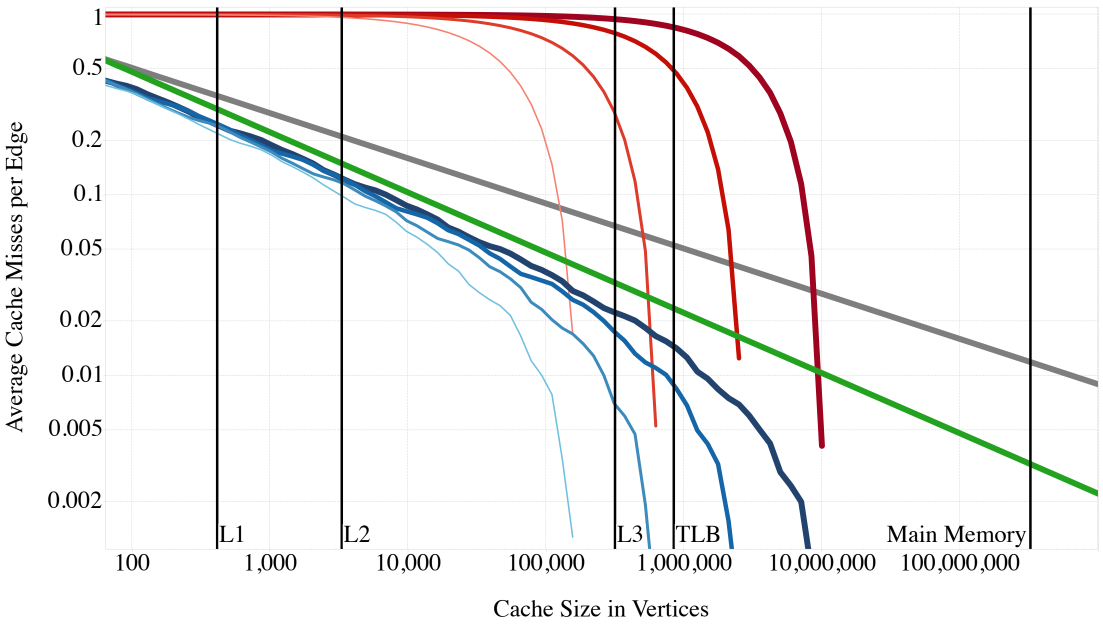

Figure 5-6: Measured and predicted miss rates (i.e., fraction of neighbors that would miss a cache of the size given by the X-axis) for a selection of graphs described in Section 5.6 as a function of cache size, in number of vertices. Various cache sizes from our Intel Xeon [111] test system are listed as vertical black lines. The red lines refer to graphs whose vertices are ordered randomly, in four sizes: the thinnest line corresponds to the smallest graph (29.8MB) and the thickest line corresponds to the largest graph (1.9GB). The blue lines correspond to the same graphs as in the red lines, except that their vertices are ordered using the Hilbert priority function. The gray line is an upper bound on the cache miss rate due to an analysis Tirthapura, Seal, and Aluru [192]. The green line is an upper bound on the cache miss rate resulting from an the analysis in this section.

Definition 1 *Given distance parameter r and norm*[4] *L^P, an n-vertex* ***random cube graph*** *$G = (V, E)$ is generated by choosing the position $\vec{p}_v$ of each vertex v uniformly randomly in the unit cube for all $v \in V$ and defining the edge set $E = \{(u, v) \in V \times V : \|\vec{p}_u - \vec{p}_v\|_P < r\}$.*

That is, vertices in a random cube graph separated by a distance at most r under the L^P norm are connected by an edge.[5] The average degree of a vertex v in a random cube graph is at most the expected number of vertices that fall within a sphere of radius r. Since the vertices in G are distributed uniformly randomly in the unit cube, the expected degree

[4]The L^P norm of an m-dimensional vector $\vec{x} = \{x_1, x_2, \ldots, x_m\}$ equals $\left(|x_1|^P + |x_2|^P + \ldots |x_m|^P\right)^{1/P}$.

[5]A small subtly exists at the borders of the random cube graph. In particular, when the radius-r sphere centered on a vertex v falls outside the unit cube, the sphere "wraps around" to the other side of the cube in a toroidal fashion. In practice, only a tiny fraction of edges are made this way, but we define random cube graphs this way for analytical convenience.

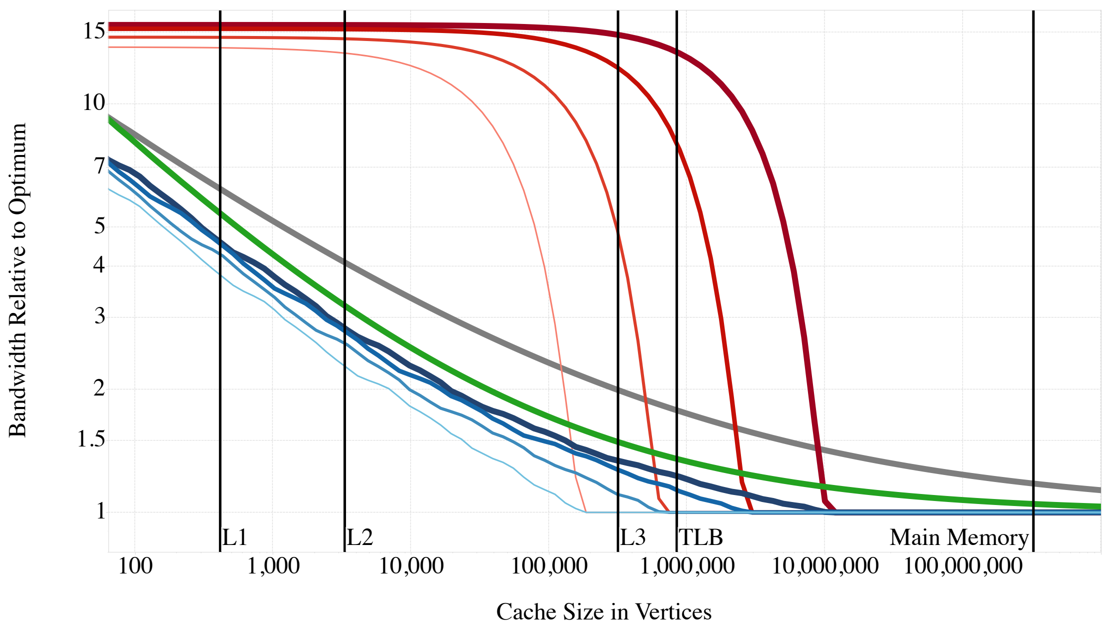

Figure 5-7: Measured and predicted amounts of bandwidth relative to optimum for a cache size given by the X-axis (i.e., the average number of times a vertex needs to be read into a cache of the size of the X-axis). The horizontal asymptote of 1 is due to the fact that each vertex must be read at least once. The meaning of each line is the same as in Figure 5-6.

is equal to the volume of the sphere in the L^P norm times the number of potential neighbors $|V|$, a fact that will be exploited in Lemma 31.

Let the vertices of a random cube graph $G = (V, E)$ be ordered by a recursively defined space-filling curve. Tirthapura, Seal, and Aluru find that the expected number of edges connecting vertices separated by a distance (i.e., in the order) of more than $|V|^{1-\alpha}/2$ is $O(V^{(3+\alpha)/4})$, when $|E| / |V| = O(1)$. A traversal of G would yield $O(V)$ work. Thus, the miss rate per vertex equals $O(V^{(3+\alpha)/4}/V) = O(V^{(-1+\alpha)/4})$. With a cache of size $M = V^{1-\alpha}$, the miss rate per vertex would equal $O(M^{-1/4})$.

Improved expected miss rate for recursive space-filling curves

Using an improved analysis, we can show that the expected miss rate of the traversal of a random cube graph, reordered using a recursive space-filling curve (e.g., the Hilbert curve [106]) with a cache of size M vertices is $O(M^{-1/3})$, the green line in Figure 5-6, improving on the $O(M^{-1/4})$ bound of Tirthapura, Seal, and Aluru, the gray line in Figure 5-6. In order to visualize the proof technique in this section, consider a vertex v in a subcube C of size $2^{-j} \times 2^{-j} \times 2^{-j}$ for some $2^{-j} \in [0, 1]$ within the unit cube and aligned with the recursive decomposition of the space-filling curve, as depicted in Figure 5-8. We can calculate the probability that another vertex w in the random cube graph is connected to v, yet lies outside C.

Lemma 31 *Given distance parameter r and norm L^P, let $G = (V, E)$ be a random cube graph, let G be decomposed into a grid of $2^{-j} \times 2^{-j} \times 2^{-j}$-sized subcubes, and let M be the expected number of vertices per subcube. The expected number of edges that connect vertices in different subcubes is $O\left(r^4 V^2 \sqrt[3]{V/M}\right)$ for all L^P.*

Proof. Consider a vertex v in a subcube C and another vertex w, to be placed uniformly randomly in the unit cube. The probability that w is placed within a distance r of v, yet outside C, is equal to the fraction of the volume of the radius-r sphere under the L^P norm centered on v that lies outside of C. This volume is at most the sum of the volume of the spherical caps — the section of a sphere that lies on one side of a plane intersecting the sphere — that lie on the other side (i.e., outside of C) of the planes coincident with the faces of C. When v is at a distance h of a face of C, the volume of such a spherical cap [123] is at most $4r^2(r - h)$, maximized by the L^∞ norm [67]. Thus, we find the expected value of V_{cap} by integrating over h:

$$\begin{aligned} V_{\text{cap}} &= 4r^2 \int_0^r (r - h)\, dh \\ &= 4r^4 - \frac{1}{2} 4r^4 \\ &= 2r^4 \end{aligned}$$

The position of v is uniformly distributed within C, so we can merely integrate over the shell of C that is within a distance r of each face. A 2D illustration of this construction under the L^2 norm is given in Figure 5-8 where we integrate over the shaded portion of C, the example square in the figure. We need only to multiply V_{cap} by the area of each face, 2^{-2j}, and multiply by the 6 faces to find an upper bound on the probability that w is connected to v, but not in C, resulting in a probability of $6r^4 2^{-2j}$. There are 2^{3j} total subcubes, so the probability that a particular pair of vertices are connected, but lie in different subcubes is then at most $6r^4 2^j$. There are at most $|V|^2/2$ pairs of vertices, so the expected number of edges crossing between different subcubes is $3r^4 |V|^2 2^j$.

The average number of vertices per subcube M is $2^{-3j}|V|$, so $2^j = \sqrt[3]{|V|/M}$. Thus, the expected number of edges that connect vertices in different subcubes is $3r^4|V|^2\sqrt[3]{|V|/M} = O\left(r^4V^2\sqrt[3]{V/M}\right)$.

□

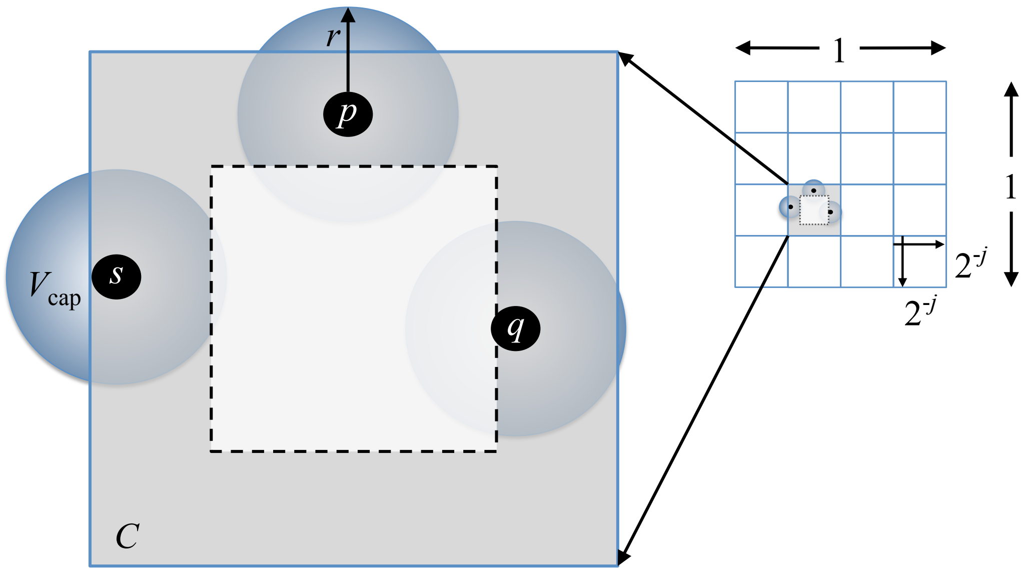

Figure 5-8: A 2D analogy to a random cube graph, showing three hypothetical vertices s, p, and q. Each vertex has a different distance to the border of the subcube, C, where the probability that another vertex is randomly placed outside of C, but within a radius r, is equal to the volume of the corresponding spherical cap, e.g., V_{cap}.

Lemma 32 *Let $G = (V, E)$ be a random cube graph graph subdivided by a recursive space-filling curve. Given any value of $M \in [72\ln|V|, |V|]$, let j be the minimum recursion level such that $M \geq 2^{-3j}|V| + \sqrt{(2^{-3j}|V|)(9\ln|V|)}$. The probability that any subcube at the jth recursion level holds more M vertices is less than $|V|^{-2}$.*

Proof. For convenience, let $n = |V|$ and let $M' = 2^{-3j}n$ be the expected number of vertices per subcube. Let X_v an indicator variable where a value of 1 indicates that vertex v is in a particular subcube for all $v \in V$. We use the Chernoff bound [48]

$$\Pr\{X \geq (1+\sigma)\mu\} \leq \exp\left(-\sigma^2\mu/3\right) \quad \forall\sigma \in [0,1], \tag{5.1}$$

letting μ be the mean of random variable $X = \sum_v X_v$, to show that the probability that a particular subcube holds more than M vertices is less than n^{-3}. By the lemma statement,

$M \geq 2^{-3j}n + \sqrt{(2^{-3j}n)(9\ln n)}$, and substituting M', $M \geq \left(1 + \sqrt{(9\ln n)/M'}\right) M'$. Thus, $\sqrt{(9\ln n)/M'} \leq 1$, since $8M' > M$ and $M \geq 72\ln n$. Applying Equation (5.1) with $\mu = 2^{-3j}n$ and $\sigma = \sqrt{(9\ln n)/M'}$, the probability that a particular subcube holds $X \geq M$ vertices is at most

$$\begin{aligned}
\Pr\left\{X \geq \left(1 + \sqrt{\frac{9\ln n}{M'}}\right) M'\right\} &\leq \exp\left(-\frac{9\ln n}{M'}\frac{M'}{3}\right) \\
&= \exp\left(-3\ln n\right) \\
&= n^{-3}.
\end{aligned}$$

There are at most n subcubes, so by the union bound, the probability that any subcube holds more than M vertices is at most $n \cdot n^{-3} = |V|^{-2}$. □

Lemma 33 *Let $G = (V, E)$ be a random cube graph with distance parameter r and norm L^P. Given a cache of M vertices for any $M \in [72\ln|V|, |V|]$, a traversal of G, reordered using a recursive space-filling curve, incurs $O\left(r^4 V \sqrt[3]{V/M}\right)$ expected misses per vertex for all L^P.*

Proof. Consider a recursion level of the space-filling curve used to order the vertices in the random cube graph such that each subcube is of size $2^{-j} \times 2^{-j} \times 2^{-j}$. For convenience, let $n = |V|$ and let $M' = 2^{-3j}n$ be the expected number of vertices held by subcube. Let j be the minimum integer value such $2^{-3j}n + \sqrt{(2^{-3j}n)(9\ln n)} \leq M$.

Let X_i be the number of vertices held by the ith subcube. If each $2^{-j} \times 2^{-j} \times 2^{-j}$-sized subcube holds fewer than M vertices (i.e., $X_i < M\ \forall i$), then the expected number of edges connecting vertices in different subcubes is at most $O\left(r^4 n^2 \sqrt[3]{n/M}\right)$ by Lemma 31 for any given norm L^P. Each such edge could cause a miss in an M-vertex cache. The fact that not all subcubes have M vertices can only reduce the number of cache misses, as all pairwise distances between vertices in memory would decrease upon removing vertices from sparsely filled subcubes.

Finally, the probability that not all $2^{-j} \times 2^{-j} \times 2^{-j}$-sized subcubes hold fewer than M vertices is at most n^{-2}, and the since number of edges is at most n^2, the expected number of misses is at most

$$\Pr\left\{X_i < M \quad \forall i\right\} \cdot O\left(r^4 n^2 \sqrt[3]{\frac{n}{M'}}\right) + \Pr\left\{\overline{X_i < M \quad \forall i}\right\} \cdot n^2 \leq O\left(r^4 n^2 \sqrt[3]{\frac{n}{M'}}\right) + n^{-2} \cdot n^2$$

$$\leq O\left(r^4 n^2 \sqrt[3]{\frac{n}{M'}}\right).$$

And since $8M' > M$ and there are $|V|$ vertices, the expected number of misses per vertex is $O\left(r^4 V \sqrt[3]{V/M}\right)$ □

Theorem 34 *Let $G = (V, E)$ be a random cube graph with distance parameter r and norm L^P. Given a cache of M vertices for any $M \in [72 \ln |V|, |V|]$, a traversal of G, reordered using a recursive space-filling curve, incurs $O(M^{-1/3})$ expected misses per vertex for all L^P when $\mathrm{E}[|E| / |V|] = O(1)$.*

Proof. The expected number of edges $d = \mathrm{E}[|E| / |V|]$ in a random cube graph is equal to the volume of the radius-r sphere under the L^P norm times the number of vertices $|V|$. To maximize the number of expected misses, we choose $L^P = L^1$ to maximize r. The volume of the radius-r sphere in 3 dimensions under the L^1 norm is $(4/3)r^3$ and thus $r = \sqrt[3]{3d/(4|V|)}$. By Lemma 33 and given that $d = O(1)$, the expected number of misses per vertex is at most

$$\begin{aligned}
O\left(r^4 V \sqrt[3]{\frac{V}{M'}}\right) &= O\left(\left(\frac{3d}{4V}\right)^{\frac{4}{3}} V \sqrt[3]{\frac{V}{M'}}\right) \\
&= O\left(\frac{3d}{4V} V \sqrt[3]{\frac{3dV}{4VM'}}\right) \\
&= O\left(\frac{3d}{4} \sqrt[3]{\frac{3d}{4M'}}\right) \\
&= O\left(M^{-1/3}\right).
\end{aligned}$$

□

Goodness of approximation

Why should we believe that good cache behavior for traversals of random cube graphs would generalize to tetrahedralized mesh graphs? That is, we rely heavily on the way in which random cube graphs are generated, particularly that vertices are uniformly randomly distributed, to show that traversals of them with a cache of M vertices leads to only $O(M^{-1/3})$ cache misses per vertex. Furthermore, we can see in Figure 5-9 that the distribution of edge lengths appears quite different for random cube graphs and the others in our graph suite. For

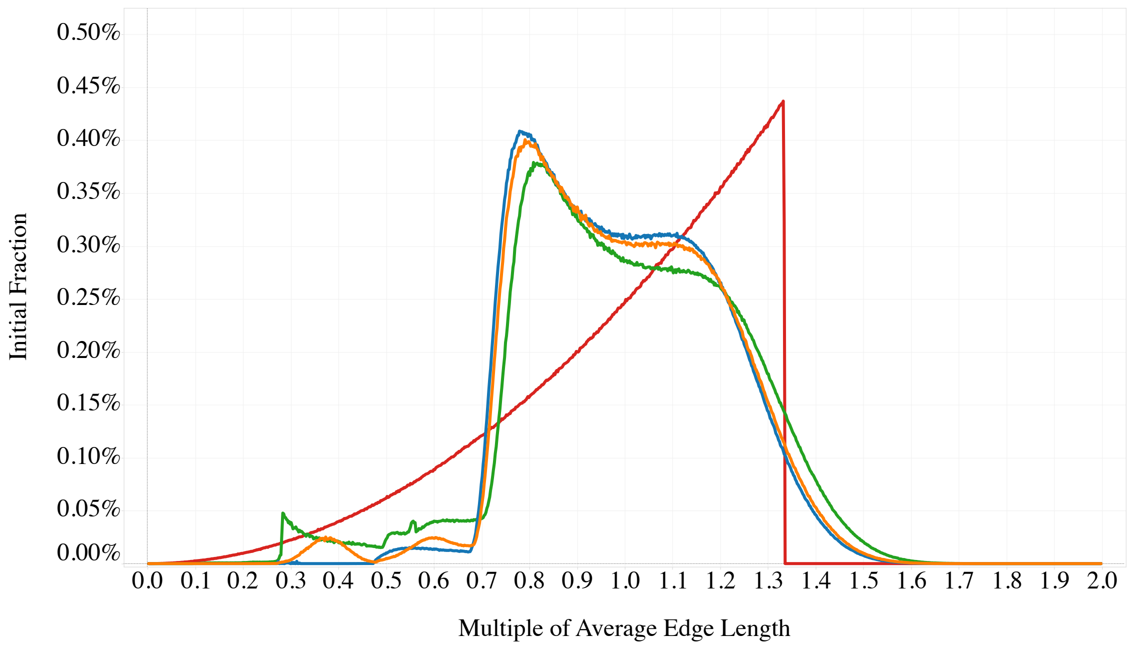

Figure 5-9: The distribution of edge lengths, relative to each graph's average edge length, for four different graphs described in Section 5.6. The first three graphs, bunny (orange), dragon (green), and cube (blue), were generated using a tetrahedralizing mesh refinement engine `TetGen` [181, 182]. The last graph, rand (red), is a random cube graph.

instance, notice that the cube, bunny, and dragon graphs all appear to have the same peculiar distribution of edge length, presumably an artifact of the particular tetrahedralization algorithm used by `TetGen`. When we examine the actual miss rate behavior in Figure 5-10, however, we see that the cube and rand graphs are similar despite their differences in edge length.

Why should we believe that good cache behavior for traversals of mesh graphs filling the unit cube would generalize to arbitrary topologies? That is, practitioners simulate complex models and generally not cubes. We see in Figure 5-10 that the miss rate curves for the bunny and dragon graphs seem to track the general slope of rand and cube, but with a constant offset.[6] To account for the difference in the shapes of realistic mesh graphs, we analyze a generalization of random cube graphs.

[6]The Y-axis in Figure 5-10 uses a log scale, so the constant offset corresponds to a constant multiplicative factor higher miss rate.

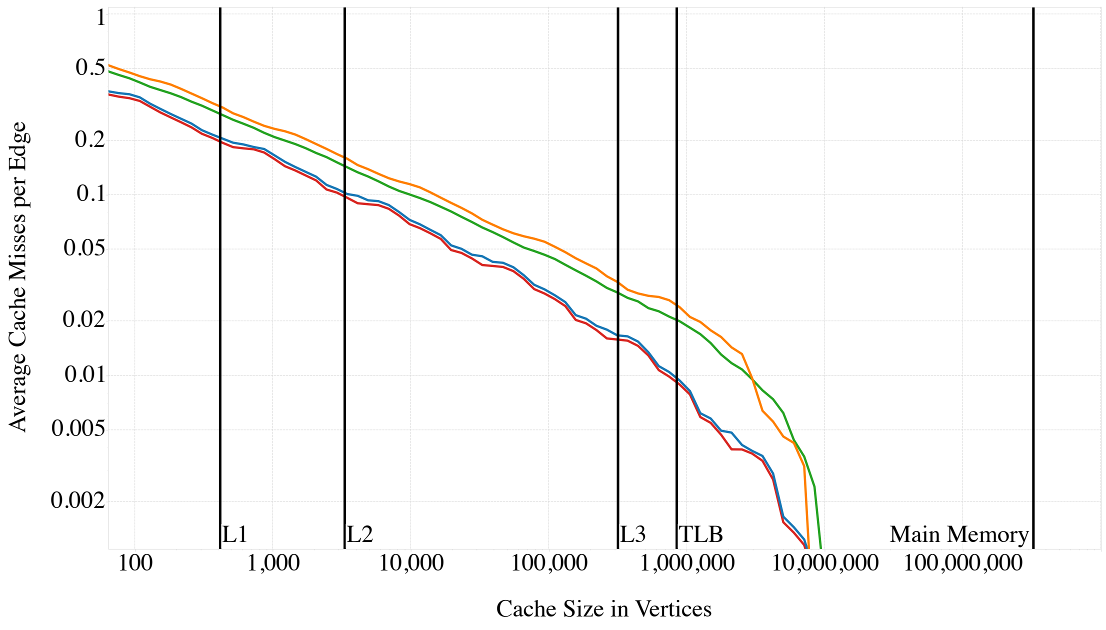

Figure 5-10: Measured miss rates (i.e., fraction of neighbors that would miss a cache of the size given by the X-axis) for a selection of similar sized graphs described in Section 5.6 as a function of cache size, in number of vertices. Various cache sizes from our Intel Xeon [111] test system are listed as vertical black lines. The first three graphs, bunny (orange), dragon (green), and cube (blue), were generated using a tetrahedralizing mesh refinement engine `TetGen` [181, 182]. The last graph, rand (red), is a random cube graph.

Definition 2 *Given distance parameter r and norm L^P, an n-vertex* ***random arbitrary-topology graph*** *$G = (V, E)$ with surface $\mathcal{S}$ is generated by choosing the position $\vec{p}_v$ of each vertex v uniformly randomly within $\mathcal{S}$ for all $v \in V$ and defining the edge set $E = \{(u, v) \in V \times V : \|\vec{p}_u - \vec{p}_v\|_P < r\}$.*

Each such graph has a fill-factor p, the fraction of the unit cube that is occupied by the surface.[7] For example, the bunny graph occupies 16.8% of the unit cube and the dragon graph occupies 26.6% of the unit cube. We will see that the offset in miss rate is directly related to the fraction of the unit cube occupied by the topology.

Lemma 35 *Given distance parameter r and norm L^P, let $G = (V, E)$ be an a random arbitrary-topology graph with surface $\mathcal{S}$, filling some fraction $p \in (0, 1]$ of the unit cube. Given a cache of M vertices for any $M \in [72 \ln |V|, |V|]$, a traversal of G reordered using a recursive space-filling curve incurs $O\left(\left(r^4 V/p^2\right) \sqrt[3]{V/pM}\right)$ misses per vertex for all L^P.*

[7]It is assumed that the surface is translated and scaled to fit in the unit cube.

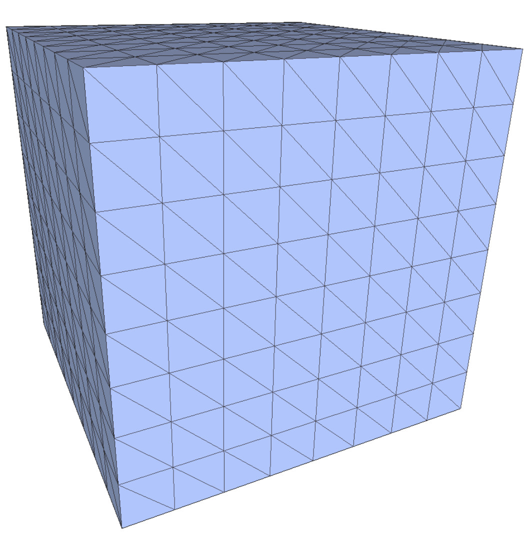
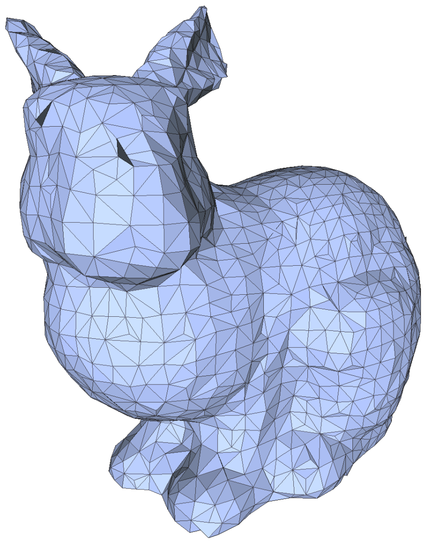

Figure 5-11: The cube and bunny mesh graphs, described in detail in Section 5.6.

PROOF. First, we imagine that we start with a random cube graph, with vertices ordered using a recursive space-filling curve. Then, we proceed to remove vertices which lie outside the surface $\mathcal{S}$, noting that the distance in memory between all remaining vertex pairs either stays the same or decreases by one after each removal. We continue to remove vertices in this manner, revealing e.g., the angel inside Michelangelo's block of marble[8] or the bunny inside the cube, as in Figure 5-11. Thus, by Lemma 33 the total expected number of misses out of an M-vertex cache is no more than with a $(|V|/p)$-vertex random cube graph,

$$O\left(\frac{r^4V^2}{p^2}\sqrt[3]{\frac{V}{pM}}\right).$$

Since G has $|V|$ vertices, the expected number of misses per vertex is $O\left(\left(r^4V/p^2\right)\sqrt[3]{V/pM}\right)$.

□

Lemma 36 *Given distance parameter r and norm L^P, let $G = (V, E)$ be an a random arbitrary-topology graph with surface $\mathcal{S}$, filling some fraction $p \in (0, 1]$ of the unit cube. Given a cache of M vertices for any $M \in [72 \ln |V|, |V|]$, a traversal of G reordered using a recursive space-filling curve incurs $O\left((1/p)M^{-1/3}\right)$ misses per vertex for all L^P and all $r = O\left((p/V)^{1/3}\right)$.*

PROOF. The proof follows from Lemma 35. To maximize the expected number of misses per vertex, we maximize $r = O\left((p/V)^{1/3}\right)$, the value of r that would yield a $(|V|/p)$-vertex

[8]"Ho visto un angelo nel marmo ed ho scolpito fino a liberarlo" – Michelangelo Buonarroti [195].

random cube graph with constant average degree by Theorem 34. Thus, the expected number of total misses is

$$\begin{aligned}
O\left(r^4 \frac{V^2}{p^2} \sqrt[3]{\frac{V}{pM}}\right) &= O\left(\left(\frac{p}{V}\right)^{\frac{4}{3}} \frac{V^2}{p^2} \sqrt[3]{\frac{V}{pM}}\right) \\
&= O\left(\frac{pV^2}{Vp^2} \sqrt[3]{\frac{pV}{pVM}}\right) \\
&= O\left(\frac{V}{p} \sqrt[3]{\frac{1}{M}}\right).
\end{aligned}$$

Since G has $|V|$ expected vertices, the expected number of misses per vertex is $O\left((1/p)M^{-1/3}\right)$. $\square$

The practical implications of Theorem 34 and Lemma 36 can be seen in Table 5-12, which summarizes how reordering the vertices of our graph suite, described in detail in Section 5.6, results in a real performance advantage. For example, Lax, the baseline scheduler which simply updates each vertex in parallel, achieves a speedup of 4.51–5.95 times using the Hilbert priority function ρ_H to order the vertices compared to the baseline which orders the vertices using the random priority function ρ_R.

Scheduler	T_1	$\frac{T_{1,\rho_R}}{T_{1,\rho_N}}$	T_{12}	$\frac{T_{12,\rho_R}}{T_{12,\rho_N}}$
	$\rho_R \,/\, \rho_N$		$\rho_R \,/\, \rho_N$	
Lax	89.25 / 19.78	4.51	9.96 / 1.67	5.95
BSP	91.59 / 21.16	4.33	10.21 / 1.91	5.35
Laika	93.64 / 21.95	4.27	10.56 / 2.02	5.24
Prism	96.44 / 33.92	2.84	11.03 / 3.74	2.95
JP	61.37 / 29.82	2.06	7.34 / 6.25	1.17

Table 5-12: Performance comparison of data-graph computation schedulers random ordering of vertices, denoted ρ_R, and reordering of vertices using the Hilbert priority curve, denoted ρ_H. Each runtime is the geometric mean across the largest size of four different graph topologies described in Section 5.6. T_1 and T_{12} are the runtimes on 1 and 12 workers, respectively.

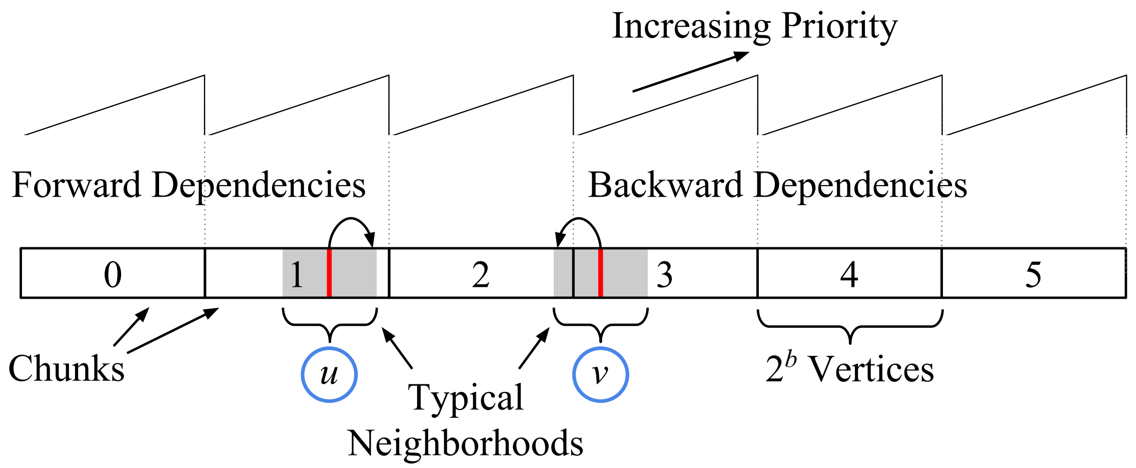

Figure 5-13: Diagram describing the scheduling algorithm LAIKA detailed in Figure 5-14. The diagram consists of 2^b-vertex "chunks", where b is a configuration parameter, each of which is processed serially. Due to the locality resulting from the Hilbert priority function described in Section 5.2, the neighbors of vertices labeled u and v, respectively, predominantly lie in the shaded regions surrounding them. Chunks are processed serially, so it is not necessary to keep track of predecessors and successors that lie within a single chunk. As a result, vertex u merely executes its update function and incurs no overhead for updating the counters of its neighbors, as with JP. The vertex v illustrates a common phenomenon, where vertices near the beginning of a chunk (i.e., chunk 3) have successors toward the end of the previous chunk (i.e., chunk 2), a *backward dependency*, which is why LAIKA processes chunks in two phases, obviating the need to track such dependencies.

5.3 The Laika data-graph computation scheduler

In this section we describe a new scheduling algorithm called LAIKA which takes advantage of the reordering of the vertices by the Hilbert priority function. It is a priority-dag scheduling algorithm, like JP, in that each vertex has a dependency counter which is decremented for each predecessor that is processed. But many of the (atomic) decrements that would normally appear in JP are removed by design in LAIKA in order to reduce overhead. Details of JP can be found in Section 5.1 and Figures 1-6 and 5-2. A diagram depicting LAIKA can be found in Figure 5-13, and pseudocode can be found in Figures 5-14—5-17. The algorithm breaks up the vertices into contiguous chunks of size 2^b, where b is a configuration parameter, each of which is processed serially in two phases.[9] The first half of each chunk is processed in the first phase and the second half in the second phase. The barrier between the two phases obviates the need to protect against a data race between neighboring vertices in different

[9] LAIKA is deterministic given a fixed value of b, irrespective of the number P of workers. In practice, we use the rule of thumb that a program with at least $10P$ parallelism should achieve nearly perfect linear speedup on P processors [52, p. 783] to find the maximum b such that $|V|/2^b > 10P$.

phases, as depicted in Figure 5-13. Thus, by reordering the vertices with the Hilbert priority function such that most neighbors of a vertex v lie within a 2^b-vertex window centered on v, as dictated by Lemma 36, LAIKA effectively removes the bulk of the scheduling overhead incurred by JP.

```
LAIKA($G, b, numRounds$)
273 LAIKA-INIT($G, b, \rho \in R$)
274 $P$ = GET-NUM-WORKERS( )
275 $Q$ = ALLOCATE-QUEUES($P$)
276 $cnt = 0$
277 for $p = 1$ to $P$
278     spawn LAIKA-WORKER($G, Q, b, cnt, numRounds, P$)
```

Figure 5-14: The LAIKA data-graph computation scheduling algorithm. First, LAIKA calls LAIKA-INIT, detailed in Figure 5-15, to initialize the graph, using a random priority function (i.e., $\rho \in R$) to break ties between vertices at the same position within their respective (different) chunks. Next, LAIKA allocates the work queues, through which the workers will share the work of processing chunks. Finally, LAIKA spawns P independent instances of the function LAIKA-WORKERS, detailed in Figure 5-16, to process the graph over *numRounds* rounds.

```
LAIKA-INIT($G, b, \rho$)
279 let $G = (V, E)$
280 parallel for $v \in V$
281     $N'(v) = \{w \in N(v) :$ SAME-PHASE$(w, v) \wedge$ DIFFERENT-CHUNK$(w, v)\}$
282     $v.pred = \{w \in N'(v) : \rho(w) > \rho(v)\}$
283     $v.counter = |v.pred|$
284     $v.succ = \{w \in N'(v) : \rho(w) < \rho(v)\}$
```

Figure 5-15: Initialization function for LAIKA, which finds for each vertex the predecessor and successor vertex sets, which lie in different chunks and the same phase (i.e., the same half of each vertex's respective chunk). The counter value for each vertex is initialized to be the cardinality of the predecessor set. The function DIFFERENT-CHUNK(w, v) returns TRUE if w and v are in different chunks $\left(\text{i.e., } \lfloor w/2^b \rfloor \neq \lfloor v/2^b \rfloor\right)$ and FALSE otherwise. The function SAME-PHASE(w, v) returns TRUE if w and v are in the same phase $\left(\text{i.e., } \lfloor (w \pmod{2^b}) / 2^{b-1} \rfloor == \lfloor (v \pmod{2^b}) / 2^{b-1} \rfloor\right)$ and FALSE otherwise.

Laika uses an explicit randomized work-stealing scheduler[10] to coordinate the processing of chunks among P workers. Work is exposed to other workers via a set of P concurrent queues [105, Ch. 10], allocated in line 275 of Figure 5-14, all of which are visible to all workers. Each worker executes the function Laika-Worker in parallel, coordinating their work through the work queues, $Q[0{:}P\text{-}1]$, and a shared counter *cnt* initialized in the parent function Laika.

```
Laika-Worker($G, Q, b, cnt, numRounds, P$)
285  let $G = (V, E)$
286  $p$ = Get-Worker-ID( )
287  $start[0] = 0$
288  $start[1] = 2^{b-1}$
289  $end[0] = 2^{b-1}$
290  $end[1] = 2^{b}$
291  $N = |V| / P$
292  for $round = 1$ to $numRounds$
293      for $phase = 0$ to 1
294          Atomic-Add($cnt, N$)
295          Push-Chunks($Q[p], N \cdot p, N, start[phase]$)
296          while $cnt > 0$
297              $\langle c, idx \rangle$ = Pop($Q$[Rand( ) $\pmod P$]))
298              if $c \neq$ NIL
299                  $v = V[idx + c \cdot 2^{b}]$
300                  if Process-Chunk($v, Q[p], b$) == $end[phase]$
301                      Atomic-Dec($cnt$)
302          Barrier( )
```

Figure 5-16: The Laika-Worker function is executed by each worker, where the round number and phase number is maintained by each worker independently (i.e., in lines 292 and 293) through a shared variable, *cnt*. Each chunk is initially pushed onto the work queue in line 295 and the shared counter *cnt* is incremented for each chunk on line 294. Then, each chunk is processed serially on line 300, where workers steal chunks to process at random on line 297 from the P work queues. Chunks can be "shelved" when the current vertex within the chunk has unmet dependencies. When a worker resolves a vertex's dependencies, the chunk is enabled and pushed onto the work queue. When a chunk is completed, the shared variable *cnt* is decremented. When *cnt* goes to 0, the workers all meet at a barrier on line 302, before resuming the next phase.

[10]Programs written in Cilk [113] implicitly use a randomized work-stealing scheduler, which is embedded in the Cilk runtime system, even though Cilk programs do not explicitly comprehend workers or how work is allocated among the workers. By contrast, Laika uses an application-specific runtime to coordinate the work in the data-graph computation using a randomized work-stealing algorithm [26]. Laika uses a set of queues that are allocated on the heap to expose the work to the other workers, whereas the Cilk runtime system uses the stack of each worker to do so [104, Appendix A].

Here we describe the logical flow of the function LAIKA-WORKER in Figure 5-14. Each worker independently counts the rounds of the data-graph computation and the two phases of each round. Each phase begins with all workers pushing the chunks that nominally "belong" to them into their respective queues, as implemented in function PUSH-CHUNKS in Figure 5-17. For example, there are $N = |V|/P$ chunks per worker[11] and worker p pushes chunks $[N \cdot p, N \cdot (p+1) - 1]$ into $Q[p]$ in line 295. The counter *cnt* is incremented once for each chunk, in line 294, and it will be decremented once for each completed chunk in line 301. Next, the workers proceed to independently steal work from randomly selected queues in line 297. Specifically, a successful steal retrieves the pair $\langle c, idx \rangle$, where c is a chunk and *idx* is the index of the next vertex to be processed in line 300. The worker will work on chunk c, as implemented by the function PROCESS-CHUNK in Figure 5-17, until it either finishes the last vertex in the phase or it cannot proceed due to an unmet dependency. Let w be the next vertex to be processed in c, and consider the case that it has a dependency counter greater than 0. The worker must prematurely stop processing c, and it is ***shelved***. The worker indicates that it has been shelved by decrementing the dependency counter of w. Thus, when the last predecessor of w is processed, w's dependency counter will be decremented to a value of -1, indicating that w had been shelved when it was the next vertex to be processed in its respective chunk. We use this mechanism to allow a worker to discover that a chunk has been enabled and should be processed. Finally, the workers all meet at a barrier in line 302 once they see that the shared counter $cnt == 0$, indicating that the final chunk has been completed for the current phase. This process repeats until all phases of all rounds have been completed.

Next, we describe the logical flow of the function PROCESS-CHUNK(v, Q, b) in Figure 5-17. PROCESS-CHUNK sequentially processes each vertex in the chunk, starting with v, until it cannot proceed due to an unmet dependency, as detected by the two logical clauses in line 305. In the first clause, if the dependency counter $v.counter$ is greater than 0, then the counter will be atomically decremented to potentially shelve the chunk.[12] If another worker happens to perform the final decrement to $v.counter$ in between the evaluation of the two clauses, then the value of $v.counter$ after the call to DEC-AND-FETCH will be -1, which indicates that all dependencies for v have been met. If DEC-AND-FETCH$(v.counter)$ returns

[11]We assume for convenience and without loss of generality that P evenly divides $|V|$.

[12]We assume that the conditionals are executed in the standard left to right order and will break as soon as the condition is met. Thus, $v.counter$ will not be decremented if $v.counter == 0$

```
PUSH-CHUNKS($Q$, $chunk$, $numChunks$, $start$)
303 parallel for $c = 0$ to $numChunks - 1$
304     PUSH($Q, \langle chunk + c, start \rangle$)

PROCESS-CHUNK($v, Q, b$)
305 while $v.counter == 0$ or DEC-AND-FETCH($v.counter$) $== -1$
306     UPDATE($v$)
307     $v.counter = |v.pred|$
308     for $w \in v.succ$
309         if DEC-AND-FETCH($w.counter$) $== -1$
310             $c = \lfloor w/2^b \rfloor$
311             $idx = w \mod 2^b$
312             PUSH($Q, \langle c, idx \rangle$)
313     $v = v + 1$
314     if $v \equiv 0 \pmod{2^{b-1}}$
315         return $v$
316 return $v$
```

Figure 5-17: The function PUSH-CHUNKS merely pushes the range of chunks "belonging" to the worker calling the function, specifically $[chunk, chunk + numChunks - 1]$, each with the vertex position within the chunk of the starting vertex, given by *start*. The function PROCESS-CHUNK attempts to make as much progress as possible on the chunk, starting with vertex v, until it either finishes the chunk on line 314 or reaches a vertex with unmet dependencies on line 305 (i.e., the counter value indicates that the current vertex v still has unprocessed predecessors). For each vertex v a sequence of steps occurs. First, the user-supplied UPDATE function (i.e., the only part of LAIKA that is specific to the data-graph computation application) is called. Second, the counter $v.counter$ is reset on line 307 for the following round. Then, for each successor w in $v.succ$, the counter $w.counter$ is decremented on line 309 and w is pushed onto the work queue on line 312 if the decrement enabled it.

any value greater than -1, it means that the dependencies are not met and the chunk will be shelved. If v is indeed enabled (i.e., all vertices in the set $v.pred$ have been updated), then the user-supplied UPDATE(v) function is called and the counter $v.counter$ is reset to $|v.pred|$, in lines 306 and 307, respectively. Next, the counters for each successor, w, in the set $v.succ$ are decremented. If any post-decrement counter value $w.counter$ equals -1, it means that w had been previously shelved and is now enabled, in which case it is pushed into the work queue in line 312. Finally, PROCESS-CHUNK moves on to the next vertex in line 313 and tests if the new vertex is the last one in the phase in line 314, in which case it returns.

NUMA-aware scheduling

Laika features an explicit randomized work-stealing scheduler, and so there exists the opportunity to bias theft decisions (i.e., which queue to steal from) to optimize for Non-Uniform Memory Access (NUMA) [104]. In particular, we implemented two variations of Laika:

Random — When processor p enables a chunk, it is placed in the pth work queue. Processors randomly select a work queue to "steal" work. This algorithm mirrors the Cilk runtime system, which uses a deque per worker to expose available work to other workers.

NUMA-aware — When processor p enables a chunk c, it is placed in the work queue that "owns" it, specifically the $\lfloor N/c \rfloor$th work queue, where N is the number of chunks per worker. Processors attempt to steal work from their own work queue first and, failing that, resort to stealing from a randomly selected queue. Finally, the vertex and edge arrays are allocated using a NUMA-aware memory allocation, such that the portions of each array owned by a particular worker are allocated at the closest memory controller [104, ch. 5.4].

We find that the NUMA-aware variation is superior, but with only a modest advantage. The data summarized in Table 5-18 demonstrates that the NUMA-aware variation is never outperformed by the random variation, but that the advantage decreases with increasing graph size. Furthermore, the NUMA-aware variation performs $O(1)$ work per steal attempt, thus, by the standard analysis of randomized work-stealing schedulers [26], both variations enjoy the same asymptotic runtime guarantees.

5.4 Theoretical analysis of Laika

In this section, we show that Laika is work efficient on any input graph and that on random cube graphs, Laika can achieve linear expected speedup with the number of workers. We only consider ***ordinary*** data-graph computations, those with serial $\text{Update}(v)$ functions with $O(N(v))$ work for all $v \in V$, though other assumptions can be made at the expense of more cumbersome analysis. In addition, we only consider random cube graphs as opposed to arbitrary graphs generated by a mesh refinement engine, such as `TetGen` [181, 182]. In

Size	T_1	$\frac{T_{1,S_R}}{T_{1,S_N}}$	T_{12}	$\frac{T_{12,S_R}}{T_{12,S_N}}$
	S_R / S_N		S_R / S_N	
0	22.73 / 22.37	1.02	2.30 / 2.10	1.09
1	25.29 / 24.01	1.05	2.26 / 2.19	1.03
2	23.94 / 23.23	1.03	2.12 / 2.07	1.03
3	21.93 / 21.95	1.00	2.04 / 2.02	1.01

Table 5-18: Performance comparison of LAIKA with and without using a NUMA-aware randomized work-stealing scheduler and data allocation across all sizes of input graph. The column headings S_R and S_N correspond to the randomized and NUMA-aware policies, respectively. Each runtime is the geometric mean across four graph topologies described in Section 5.6. T_1 and T_{12} are the runtimes on 1 and 12 workers, respectively.

Section 5.6 we demonstrate empirically that the theoretical insights earned in this section are also relevant to graphs "in the wild."

Lemma 37 LAIKA *is work-efficient for any ordinary data-graph computation on any graph* $G = (V, E)$.

PROOF. From inspection of the pseudocode in Figures 5-14—5-17, one can see that UPDATE is called once per vertex per round, just as in the baseline serial execution. Furthermore, there are at most $|E|$ calls to DEC-AND-FETCH made to counters in line 309. Thus, it suffices to show that the overhead incurred by the shelving and subsequent enabling of chunks in a given round requires at most $O(V + E)$ work. When a chunk c is shelved, it is due to a particular vertex having unmet dependencies. A vertex v can be responsible for shelving a chunk c at most once, since the chunk can only be subsequently enabled once all predecessors of v have been updated. Consequently, when the chunk is later processed, at least one vertex, specifically v, will be updated before being shelved again. Because shelving a chunk costs only $O(1)$ work (i.e., a single DEC-AND-FETCH), the total cost due to shelving chunks can be at most $O(V)$. The baseline serial algorithm performs $O(V + E)$ work since it is an ordinary data-graph computation, and thus LAIKA is work-efficient. □

Lemma 38 *Let* $G = (V, E)$ *be a random cube graph with average degree* d. *For any* $c > 0$, *the maximum degree* $\max_{v \in V} |N(v)|$ *is greater than* $2d + 3(4 + c) \ln |V|$ *with probability less than* $|V|^{-(3+c)}$.

PROOF. Let r be the radius of the sphere such that vertices within a distance r are connected by an edge. Since the average degree in G is d, the distance parameter r and norm L^P must be set such that the probability that a vertex w is connected to v is $d/|V|$. Consider a vertex v in a random cube graph $G = (V, E)$ and let X_w be a indicator variable where $X_w == 1$ denotes the event that a vertex w is connected to vertex v. In order to bound the probability that more than $2d+3(4+c)\ln|V|$ vertices are connected to v, we use the Chernoff bound [48] formulated for deviations of more than 1 times the mean:

$$\Pr\{D \geq (1+\beta)\cdot \mathrm{E}[D]\} \leq \exp\left(-\frac{\beta}{3}\mathrm{E}[D]\right) \quad \forall\beta > 1,$$

where $D = \sum_w X_w$ [52, p. 1203]. Then, we have $D = |N(v)|$ and thus, by the lemma statement, $\mathrm{E}[D] = d$. Let $\beta = (d + 3(4+c)\ln|V|)/d \geq 1$ so that

$$\begin{aligned}\Pr\{|N(v)| \geq 2d + 3(4+c)\ln|V|\} &\leq \exp\left(-\frac{\beta}{3}\mathrm{E}[|N(v)|]\right)\\ &\leq \exp\left(-\frac{1}{3}\frac{d+3(4+c)\ln|V|}{d}d\right)\\ &\leq \exp\left(-\frac{1}{3}(d+3(4+c)\ln|V|)\right)\\ &\leq \exp\left(-\frac{1}{3}(3(4+c)\ln|V|)\right)\\ &\leq \exp(-(4+c)\ln|V|)\\ &\leq |V|^{-(4+c)}.\end{aligned}$$

We use the union bound across all $|V|$ vertices to see that G has maximum degree greater than $2d + 3(4+c)\ln|V|$ with probability at most $|V|\cdot|V|^{-(4+c)} = |V|^{-(3+c)}$. $\square$

Lemma 39 *Let $G = (V, E)$ be a Δ-degree graph,*[13] *let $n_G = |V|$, and let G_ρ be a priority dag induced on G by a random priority function ρ. For any $c > 0$, there exists a directed path of length $e^2 \cdot \max\{\Delta, (4+c)\ln n\}$ in G_ρ for any $n \geq n_G$ with probability at most $n^{-(3+c)}$.*

PROOF. Let $p = \langle v_1, v_2, \ldots, v_k\rangle$ be a length-k path in G. Because ρ is a random priority function, ρ induces each possible permutation among $\{v_1, v_2, \ldots, v_k\}$ with equal probability. If p is a directed path in G_ρ, then we must have that $\rho(v_1) < \rho(v_2) < \ldots < \rho(v_k)$. Hence, p is a length-k path in G_ρ with probability at most $1/k!$, which is at most $(e/k)^k$ by Stirling's

[13]A Δ-degree graph $G = (V, E)$ has $|N(v)| \leq \Delta$ for all $v \in V$.

approximation [52, p. 57] [186]. There are at most $n_G \Delta^k$ paths in G, since we have n_G potential starting points and, for each step in the path, each vertex has at most Δ potential neighbors.

We use the union bound to show that no directed path of length $k \geq e^2 \cdot \max\{\Delta, (4+c)\ln n\}$, exists in G_ρ. In particular, the probability that a length-k path exists in G_ρ is at most

$$\begin{aligned}
n_G \left(\frac{e\Delta}{k}\right)^k &= n_G \cdot \exp\left(-k \ln\left(\frac{k}{e\Delta}\right)\right) \\
&\leq n_G \cdot \exp\left(-k \ln\left(\frac{e^2 \max\{\Delta, (4+c)\ln n\}}{e\Delta}\right)\right) \\
&\leq n_G \cdot \exp\left(-k \ln\left(\frac{e^2\Delta}{e\Delta}\right)\right) \\
&\leq n_G \cdot \exp(-k) \\
&\leq n_G \cdot \exp\left(-\left(e^2 \max\{\Delta, (4+c)\ln n\}\right)\right) \\
&\leq n_G \cdot \exp(-(4+c)\ln n) \\
&\leq n_G \cdot n^{-(4+c)} \\
&\leq n^{-(3+c)}.
\end{aligned}$$

□

Theorem 40 *Let $G = (V, E)$ be a random cube graph with $n = |V|$ vertices and average degree d. For any $c > 0$ and for any positive $b < \lg n - 1$,* Laika *with chunk size 2^b executes an ordinary data-graph computation on G using more than $e^2 (2d + 3(4+c)\ln n)^2 2^b$ span with probability less than $n^{-(2+c)}$.*

Proof. Consider the algorithm Laikb, which is a synchronous version of Laika. That is, each chunk will be processed synchronously in parallel: the kth vertex from all chunks will be processed before the $k+1$st vertex from any chunk is processed. Since Laika is at least as fast as Laikb on all inputs, it suffices to show that the span of Laikb satisfies the lemma statement. The dependent edges between vertices at the kth position within each chunk form the priority dag G_k induced by a random priority function. Thus, the depth of each such dag, denoted $L(G_k)$, is governed by Lemma 39. The span of Laikb applied to G will be less than the product of

1. the maximum degree (recall: UPDATE(v) is serial and uses $|N(v)|$ work), and
2. the maximum depth among the 2^b random priority dags, $\{G_1, G_2, \ldots, G_{2^b}\}$, and
3. the total number of priority dags: $(\max_{v \in V} |N(v)|) \cdot \left(\max_{k \in [1,2^b]} L(G_k)\right) \cdot 2^b$.

Let A_Δ be the event that the maximum degree in G exceeds $\Delta = 2d + 3(4+c)\ln n$ and let A_{G_k} be the event that the depth of the priority dag G_k exceeds $e^2\Delta$, for all $k \in [1, 2^b]$. The event $\overline{A_\Delta} \cap \{\bigcap_{\forall k} \overline{A_{G_k}}\}$ would imply that the overall span of LAIKB applied to G is at most $e^2(2d + 3(4+c)\ln n)^2 2^b$, since $\Delta > (4+c)\ln n$ when $\overline{A_\Delta}$ is true, as required by Lemma 39. We use De Morgan's law [58] to find the negation of this event, and following from Lemmas 38 and 39, we see that the probability of the event $A_\Delta \cup \{\bigcup_{\forall k} A_{G_k}\}$ is thus at most

$$\begin{aligned}
&\leq \Pr\{A_\Delta\} + \sum_{k\in[1,2^b]} \Pr\{A_{G_k}\} \\
&\leq \Pr\{A_\Delta\} + \sum_{k\in[1,2^b]} \left[\Pr\{A_{G_k} | A_\Delta\} \Pr\{A_\Delta\} + \Pr\left\{A_{G_k} | \overline{A_\Delta}\right\} \Pr\left\{\overline{A_\Delta}\right\}\right] \\
&\leq \Pr\{A_\Delta\} + \sum_{k\in[1,2^b]} \left[1 \cdot \Pr\{A_\Delta\} + \Pr\left\{A_{G_k} | \overline{A_\Delta}\right\} \cdot 1\right] \\
&\leq \Pr\{A_\Delta\} + \sum_{k\in[1,2^b]} \left[n^{-(3+c)} + \Pr\left\{A_{G_k} | \overline{A_\Delta}\right\} \cdot 1\right] \\
&\leq \Pr\{A_\Delta\} + \sum_{k\in[1,2^b]} \left[n^{-(3+c)} + n^{-(3+c)}\right] \\
&\leq \Pr\{A_\Delta\} + 2 \cdot 2^b \cdot n^{-(3+c)} \\
&\leq n^{-(3+c)} + 2 \cdot 2^b \cdot n^{-(3+c)} \\
&\leq n \cdot n^{-(3+c)} \\
&\leq n^{-(2+c)}.
\end{aligned}$$

□

Corollary 41 *Let $G = (V, E)$ be a random cube graph $G = (V, E)$ with average degree d. For any $c > 0$ and any number of workers $P < 2|E| / (e(2d + 3(4+c)\ln |V|))^2$, there exists a choice of chunk size 2^b such that* LAIKA *executes an ordinary data-graph computation on G using P workers in $O((V+E)/P)$ expected time.*

Proof. The expected work of the data-graph computation described in the corollary statement is $2|E|$ by linearity of expectation:

$$\begin{aligned}
\mathrm{E}\left[\sum_{v\in V} |N(v)|\right] &= \sum_{v\in V} \mathrm{E}\left[|N(v)|\right] \\
&= \sum_{v\in V} d \\
&= d\,|V| \\
&= 2\,|E|\,.
\end{aligned}$$

For convenience, let $n = |V|$. Let A_S denote the event that the data-graph computation in the corollary statement has span exceeding $e^2\left(2d + 3\left(4+c\right)\ln n\right)^2$, which occurs with probability at most $n^{-(2+c)}$ by Theorem 40, letting 2^b equal 1. Thus, the expected parallelism — the ratio of work to span — is at least

$$\begin{aligned}
&\geq \Pr\left\{\overline{A_S}\right\} \frac{2\,|E|}{e^2\left(2d+3\left(4+c\right)\ln n\right)^2} + \Pr\left\{A_S\right\}\cdot 1 \\
&\geq \left(1 - n^{-(2+c)}\right) \frac{2\,|E|}{e^2\left(2d+3\left(4+c\right)\ln n\right)^2} + n^{-(2+c)}\cdot 1 \\
&\geq \frac{2\,|E|}{e^2\left(2d+3\left(4+c\right)\ln n\right)^2} + n^{-(2+c)}\left(1 - \frac{2\,|E|}{e^2\left(2d+3\left(4+c\right)\ln n\right)^2}\right) \\
&\geq \frac{2\,|E|}{e^2\left(2d+3\left(4+c\right)\ln n\right)^2} + n^{-(2+c)}\left(1 - n^2\right) \\
&\geq \frac{2\,|E|}{e^2\left(2d+3\left(4+c\right)\ln n\right)^2}.
\end{aligned}$$

Laika is a randomized work-stealing scheduler [26], and so the expected runtime is at most the work $O(E+V)$ divided by the number P of workers, plus the span $e^2\left(2d+3\left(4+c\right)\ln n\right)^2$. By constraining our choice of P to be less than the parallelism, however, the first term dominates, and thus Laika runs in $O((E+V)/P)$ time. A similar analysis holds for smaller values of P and correspondingly larger values of 2^b. □

Corollary 42 Laika *achieves linear speedup for any ordinary data-graph computation applied to an* $O(1)$*-degree random cube graph* $G = (V, E)$ *using* $P = O(V/\lg^2 V)$ *workers.*

Proof. The proof follows from Corollary 41. □

5.5 The Mass-Spring-Dashpot model

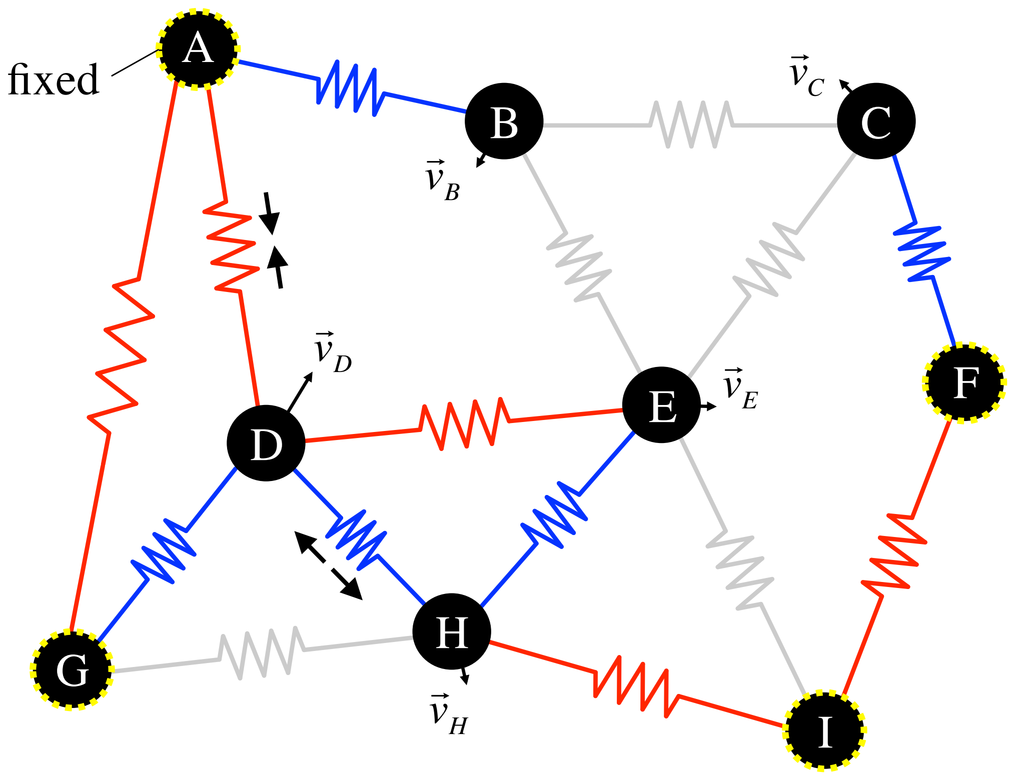

Figure 5-19: An example of the Mass-Spring-Dashpot model. The vertices encircled by a dashed yellow line are fixed in place, modeled as having infinite mass. The springs obey Hooke's Law, generating a force $k(1 - x/L)$ along the direction of the edge, where k is the stiffness coefficient, L is the rest length, and x is the distance between the endpoints. The average edge length (i.e., the physical distance between the endpoints) is the "rest length" for all springs. The gray springs are quite close to the rest length and thus exert little force. The red springs are stretched out to have length longer than the rest length, generating a restoring force (e.g., the edge (A, D)). The blue springs are compressed and generate a separating force (e.g., the edge (D, H)). The net velocity (e.g., $\vec{v}_H$), induced by the net forces acting on each vertex, is shown with magnitude (i.e., length) and direction for each vertex.

This section describes an example physical simulation which is used to test the performance of LAIKA. We use the ***Mass-Spring-Dashpot model*** as a proxy for other types of physical simulations (e.g., the finite element method [53], etc.). It can be thought of as a set of vertices in 3D space connected by springs, as depicted in Figure 5-19. Each spring has a "rest length" L: springs stretched to be longer than L try to pull the endpoints together and springs compressed to be shorter than L try to push the endpoints apart. The simulation steps through time, incrementally updating the position and velocity of each vertex until a steady-state is achieved where the spring forces cancel and each vertex is at rest. We use the Mass-Spring-Dashpot model because it is among the simplest physical models that is also realistic. By simplest, we mean that it has a low number of instructions per byte of vertex data, which exacerbates the effect of scheduling overheads relative to simulations with a higher number of instructions per vertex. Thus, performance characteristics in the Mass-Spring-Dashpot model would be pessimal for most other physical simulations (i.e., those that perform more work per vertex).

The Mass-Spring-Dashpot model is an example of a physical system under Newton's laws of motion [157]. In particular, by Newton's second law, each vertex $u \in V$ in a graph $G = (V, E)$ is subject to the ordinary differential equation

$$\begin{aligned} m_u \frac{\partial \vec{v}_u}{\partial t} &= \vec{F}_u, \\ \frac{\partial \vec{p}_u}{\partial t} &= \vec{v}_u, \end{aligned}$$

where the position, velocity, and mass of vertex u is given by $\vec{p}_u$, $\vec{v}_u$, and m_u, respectively. The force term[14] $\vec{F}_u$ is solely a function of the neighborhood of vertex u, $N(u)$, and thus the Mass-Spring-Dashpot model is easily expressed as a data-graph computation, an example of which is in Figure 5-19. One component of $\vec{F}_u$ is a spring force that exists between every pair of connected vertices. By Newton's third law, this force acts in equal and opposite directions between the connected vertices. The spring force is a consequence of Hooke's law [108], where a spring with *rest length* L, *stiffness coefficent* k, and length x exerts a force $k(1 - x/L)$ along the direction of the spring's orientation. When $x > L$ (i.e., the spring is stretched out), Hooke's law implies that the spring will produce a negative restoring force and when $x < L$ (i.e., the spring is compressed), it will produce a positive separating force, as depicted in Figure 5-20. For example, a spring between two vertices u and w exerts the following force on vertex u:

$$\vec{F}_{u,w} = \underbrace{k\left(1 - \frac{\|\vec{p}_u - \vec{p}_w\|}{L}\right)}_{\text{magnitude}} \cdot \underbrace{\frac{\vec{p}_u - \vec{p}_w}{\|\vec{p}_u - \vec{p}_w\|}}_{\text{direction}} .$$

[14]We use the arrow notation in $\vec{F}_u$ to denote the vector nature of the variable associated with vertex u, thus being comprised of both a magnitude $\left\|\vec{F}_u\right\|$ and a direction $\hat{F}_u = \vec{F}_u / \left\|\vec{F}_u\right\|$. The hat notation in $\hat{F}_u$ denotes a unit-length vector in the direction of $\vec{F}_u$.

Such a force exists for every edge $(u, w) \in E$.

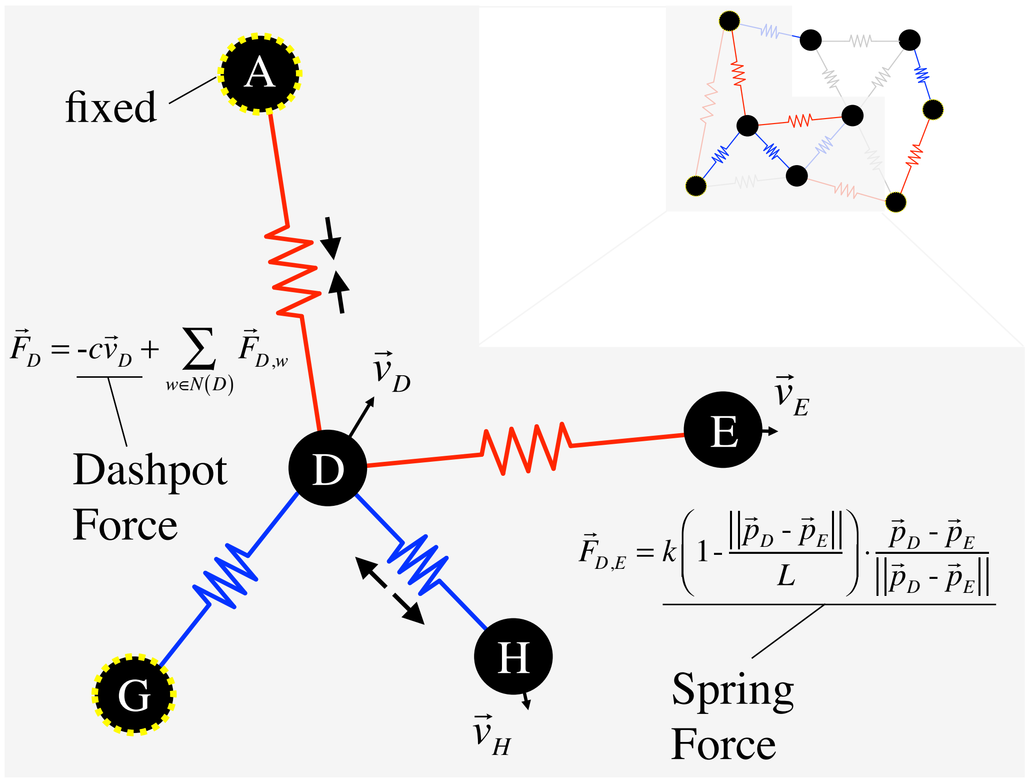
fixed
A
$\vec{F}_D = \underbrace{-c\vec{v}_D}_{} + \sum_{w \in N(D)} \vec{F}_{D,w}$
Dashpot Force
$\vec{v}_D$
D
E
$\vec{v}_E$
G
H
$\vec{v}_H$
$\vec{F}_{D,E} = \underbrace{k\left(1 - \frac{\|\vec{p}_D - \vec{p}_E\|}{L}\right) \cdot \frac{\vec{p}_D - \vec{p}_E}{\|\vec{p}_D - \vec{p}_E\|}}_{}$
Spring Force

Figure 5-20: Detail on the Mass-Spring-Dashpot model depicted in Figure 5-19 for vertex D. Vertex D has two neighbors, G and H, which have separating forces pushing D upward. Vertex D has two other neighbors, A and E, which have restoring forces pulling D up and to the right. Each spring force is given by the equation shown e.g., for $\vec{F}_{D,E}$, where $\vec{p}_D$ is the position in 3D space of vertex D and $\|\vec{p}_D - \vec{p}_E\|$ is the Euclidian distance between vertices D and E. The net force, $\vec{F}_D$, also includes the dashpot term, which is the velocity, $\vec{v}_D$, multiplied by a negative coefficient (e.g., wind resistance). The net force influences both the velocity and position of vertex D through the ordinary differential equation given in Newton's second law [157]: $m_D\,(\partial\vec{v}_D/\partial t) = \vec{F}_D$ and $(\partial\vec{p}_D/\partial t) = \vec{v}_D$.

The Mass-Spring-Dashpot model also features the dashpot force,[15] which is a type of resistance force that complements the spring forces. A vertex u travelling with velocity $\vec{v}_u$ will incur a force $-c\vec{v}_u$ in the opposite direction of travel, where c is the ***dashpot coefficient***.

[15]The dashpot force is also sometimes called a *damper* or *drag* force and is the principal reason that the Mass-Spring-Dashpot system does not oscillate indefinitely.

Thus, the net force on each vertex $u \in V$ is

$$\vec{F}_u = \underbrace{-c\vec{v}_u}_{\text{dashpot force}} + \underbrace{\sum_{w \in N(u)} \vec{F}_{u,w}}_{\text{net spring force}} .$$

For each vertex, this force may be calculated by examining the position and velocity of every neighbor w in $N(u)$, as depicted in Figure 5-20.

In practice, an implementation of the Mass-Spring-Dashpot model uses a numerical solution to Newton's laws to update the position and velocity of each vertex, ultimately pushing the edges in the graph to be closer to the rest length. For example, the initial distribution of edge lengths for four graphs in our graph suite are shown in Figure 5-9 and after 10,000 iterations, the edge lengths evolve to the distribution shown in Figure 5-21.

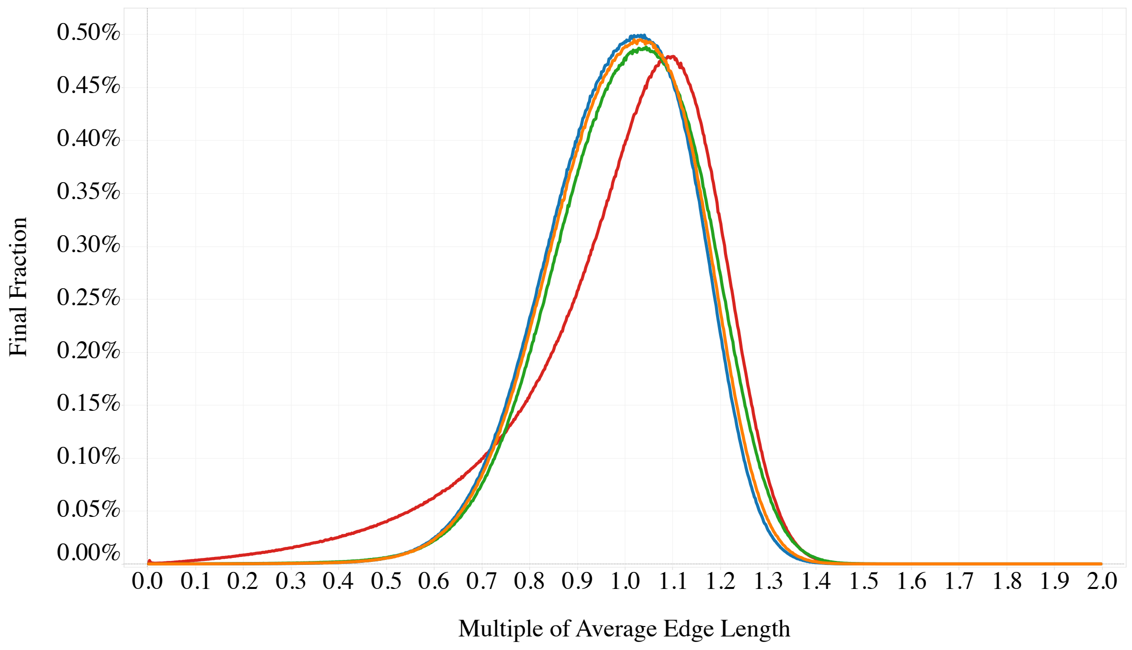

Figure 5-21: Distribution of edge lengths after 10,000 iterations for each graph topology: bunny (orange), dragon (green), cube (blue), and rand (red).

A numerical solution for the Mass-Spring-Dashpot model

We use a standard finite difference method [30, 61] to implement the Mass-Spring-Dashpot model. In particular, we approximate the derivative of the position and velocity vectors[16] at a time t using finite differences, as follows:

$$\begin{aligned}\frac{\partial \vec{v}_u(t)}{\partial t} &\approx \frac{\vec{v}_u(t+\frac{1}{2}\partial t) - \vec{v}_u(t-\frac{1}{2}\partial t)}{\partial t},\\ \frac{\partial \vec{p}_u(t)}{\partial t} &\approx \frac{\vec{p}_u(t) - \vec{p}_u(t-\partial t)}{\partial t}.\end{aligned}$$

These approximations lead to a straightforward update function implementing Newton's laws. First, we use the central difference approximation to update the velocity vector:

$$\begin{aligned}m_u \frac{\vec{v}_u(t+\partial t) - \vec{v}_u(t)}{\partial t} &= \vec{F}_u(t+\frac{1}{2}\partial t),\\ \vec{v}_u(t+\partial t) &= \vec{v}_u(t) + \partial t \frac{\vec{F}_u(t+\frac{1}{2}\partial t)}{m_u}.\end{aligned}$$

Next, we use the backward difference approximation to update the position vector, using the newly updated value of $\vec{v}_u(t+\partial t)$:

$$\begin{aligned}\frac{\vec{p}_u(t+\partial t) - \vec{p}_u(t)}{\partial t} &= \vec{v}_u(t+\partial t),\\ \vec{p}_u(t+\partial t) &= \vec{p}_u(t) + \partial t \vec{v}_u(t+\partial t).\end{aligned}$$

We approximate the velocity vector $\vec{v}_u(t)$ at time t using the central difference because it produces a better estimate than the backward difference used for the position vector $\vec{p}_u(t)$. The central difference requires that we evaluate the force vector $\vec{F}_u(t+\partial t/2)$ in the middle of the tth time step. Evaluating the force vector is comparatively easy: we merely add a scalar multiple of the velocity vector to the position vector (i.e., use $\vec{p}_u(t) + \partial t \vec{v}_u(t)/2$ in place of $\vec{p}_u(t)$ for for all force calculations involving vertex u). If we were to also use the central difference approximation for the position vector, however, we would need to evaluate the expensive force vector twice. This tradeoff yields good convergence properties with minimal extra computational overhead and is thus favored by practitioners in budget-conscious scenarios [61, 93].

[16]We use the notation $\vec{F}_u(t)$, $\vec{v}_u(t)$, and $\vec{p}_u(t)$ to refer to the force, velocity, and position of vertex u at time t, respectively.

A specific instance of the Mass-Spring-Dashpot model

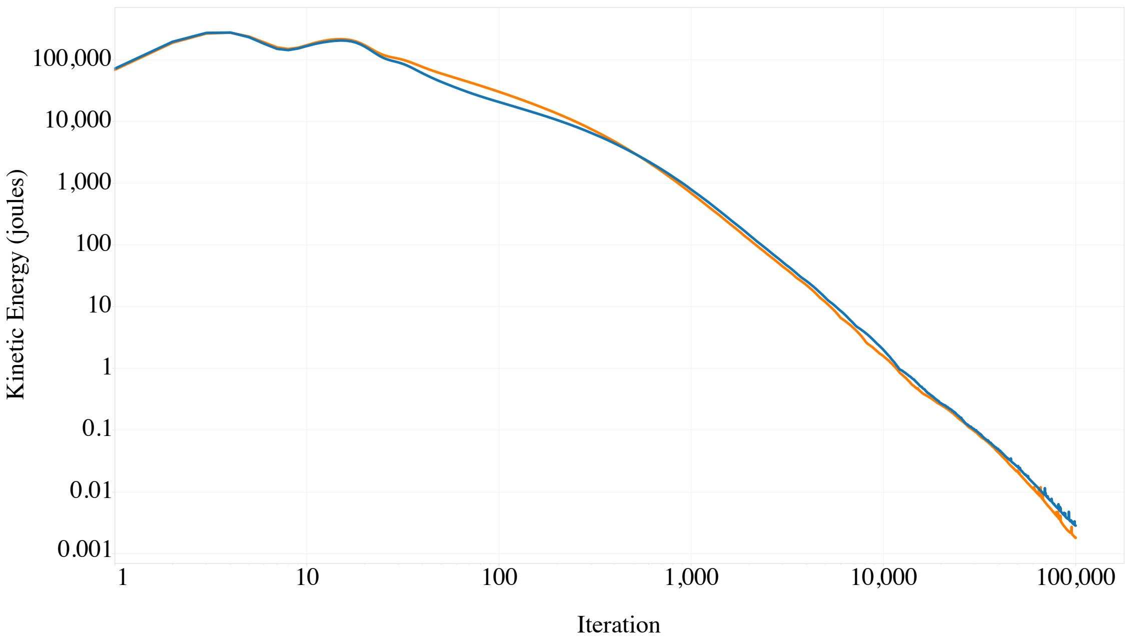

Figure 5-22: The kinetic energy, in joules, of the entire graph over 100,000 iterations of the Mass-Spring-Dashpot model for an in-place implementation (i.e., in orange) and a double-buffered implementation (i.e., in blue). At any given point in time, the kinetic energy is calculated as $\sum_{u \in V} m_u \|\vec{v}_u\|^2/2$, where $\vec{v}_u$ and m_u are the velocity and mass of vertex u, respectively.

We designed a specific instance of the Mass-Spring-Dashpot model, cast as a data-graph computation, that allows us to compare performance properties across a set of parallel schedulers, the results of which are detailed in Section 5.6. Given an input graph $G = (V, E)$, where each vertex $u \in V$ has an associated position $\vec{p}_u$ in 3D space,[17] and a number of iterations, each scheduler updates each vertex once per iteration in a scheduler-specific order. We measure the time to execute the iterations, thus excluding the initialization time for each scheduler, because we want to isolate the effects of scheduling. In addition, most physical simulations typically execute sufficiently many iterations that any initialization time is completely amortized away.

We ***anchor*** a set of vertices in each graph that is independent of the ordering of the vertices so that experiments differing only in the order of the vertices will converge to the

[17]The code supports an arbitrary number of dimensions via a compile-time constant.

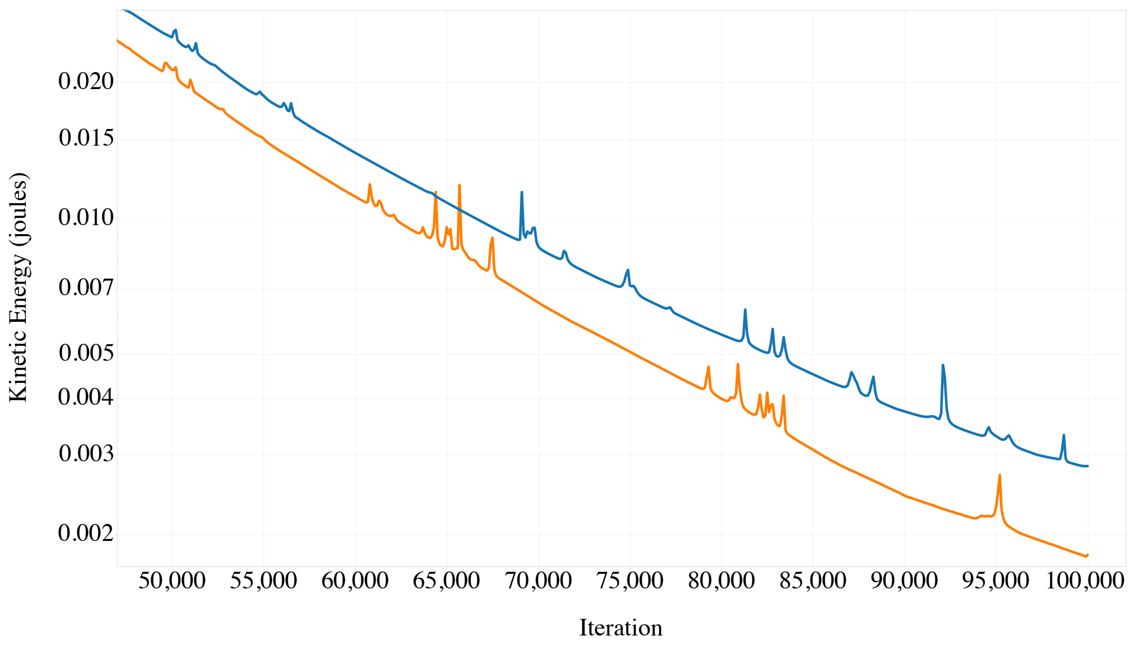

Figure 5-23: The kinetic energy, in joules, of the entire graph over the last 50,000 iterations of the Mass-Spring-Dashpot model, highlighting how the in-place version (i.e., in orange) of the model converges faster than the double-buffered version (i.e., in blue).

same result. The set V_A of anchored vertices are those that lie on a face of the axis-aligned minimum bounding box of G and are modeled as having infinite mass (i.e., $m_u = \infty \quad \forall u \in V_A$). All other vertices are modeled as having unit mass (i.e., $m_u = 1 \quad \forall u \in V \setminus V_A$). While there is no guarantee of convergence to a unique optimum state, we find that this anchoring strategy typically yields evidence of convergence, as we see in Figure 5-22, whereas if the graph were unmoored, it would float away and we would not be able to observe such convergence. Figure 5-23 also provides some motivation for the design of Laika: many in-place simulations of this type [170], and particularly iterative solvers of linear systems [188], converge faster than their double-buffered analogues. For instance, at roughly the halfway point of the simulation, the kinetic energy in the double-buffered implementation is 22% higher than the the in-place counterpart. This ratio grows until the end of the simulation when the kinetic energy in the double-buffered implementation is 57% higher than the in-place implementation. A typical strategy would be to iterate until, for example, the kinetic energy drops below a specific threshold, giving the in-place implementation an advantage in total iterations.

Since the parameters of the Mass-Spring-Dashpot model do not impact the running time of any of the schedulers for any fixed number of rounds, we merely desire a set of parameters that are stable and convergent across all runs. For simplicity, we choose the following parameters: spring stiffness $k = 1$, dashpot coefficient $c = 1$, timestep $\partial t = 0.1$. The rest length of each spring (i.e., edge) in the graph is set to the average initial length:

$$L = \frac{1}{|E|} \sum_{(u,w) \in E} \left\| \vec{p}_u - \vec{p}_w \right\| .$$

5.6 Experimental results

This section presents empirical data comparing the performance of Laika with four other data-graph computation schedulers using the Mass-Spring-Dashpot model, described in Section 5.5, as an example update function. The Mass-Spring-Dashpot model is comparatively simple, amortizing comparatively few instructions against the movement of much data, 2.22 instructions per byte, making it pessimal for Laika. We expect Laika to perform data-graph computations with more instructions per byte of application data at least as well as it performs in this section.

```
BSP($G$, $rounds$)
let $G = (V, E)$
for $i = 0$ to $rounds - 1$
    parallel for $v \in V$
        UPDATE($v$, $i$)
```

```
LAX($G$, $rounds$)
let $G = (V, E)$
for $i = 0$ to $rounds - 1$
    parallel for $v \in V$
        UPDATE($v$)
```

Figure 5-24: The Bulk-Synchronous Parallel (BSP) scheduling algorithm is a best-case scheduling algorithm in that all vertices are eligible to be updated at the beginning of each round and thus BSP incurs the minimum possible scheduling overhead. BSP uses double-buffering of application data to ensure that there are no data races when simultaneously updating neighboring vertices. By contrast, Lax updates all vertices simultaneously, in-place using a single copy of the application data, resulting in a nondeterministic parallel execution.

The experiments in this section demonstrate that Laika is 5.27 times faster when the vertices are reordered by a recursive space-filling curve (e.g., the Hilbert curve) than when they are ordered randomly. Laika also achieves a 10.89 times speedup on 12 cores. Overall, Laika is 44.27 times faster than the naive baseline serial implementation Lax in Figure 5-24, which is equivalent to Simit's current shared-memory implementation, effectively converting the problem of scheduling physical simulations from a traditionally memory-bound problem [7, 65, 70, 83, 91, 139, 162] to a compute-bound problem. LAIKA is approximately 80% as fast as an empirically-measured maximum speed for any scheduler on our test system and adds novel support for deterministic, in-place updates.

Scheduler implementations

We compare LAIKA to four other data-graph computation schedulers, described below. All schedulers were developed fairly, with a similar level of performance engineering. All schedulers were written in Cilk Plus [113] / C++ and compiled with the same compiler (`g++-5 v. 5.1.0`) and switches (full optimizations[18]), and tested on the same hardware [111], an Intel® Xeon® CPU X5650 with 12 processor cores running at 2.67GHz.

The five schedulers that we test in this section can be found online[19] and are described briefly here:

LAX — An in-place scheduler that updates all vertices in parallel, yielding a nondeterministic result with minimal scheduling overhead. Pseudocode can be found in Figure 5-24.

BSP — A double-buffered scheduler that updates all vertices in parallel, yielding a deterministic result with minimal scheduling overhead, but at the expense of double memory usage for application data. Pseudocode can be found in Figure 5-24.

LAIKA — An in-place, priority-dag scheduler that exploits the locality offered by reordering the vertices according to a recursive space-filling curve. Figure 5-13 gives a pictorial description of LAIKA and complete pseudocode can be found in Figures 5-14—5-17.

PRISM — An in-place scheduler that uses a vertex-coloring of the graph to parcel out independent sets of vertices that may be updated safely in parallel, as depicted in Figure 1-1.

JP — An in-place, priority-dag scheduler that uses a priority function to order the vertices as depicted in Figure 1-6. Pseudocode can be found in Figure 5-2.

Each scheduler has different requirements for the data included in the vertex data structure, presenting advantages or challenges, respectively, to the performance of each. In particular, each scheduler requires three data components at each vertex:

[18]All schedulers were compiled with the following compiler switches: `-fcilkplus -std=c++11 -O3 -Wall -m64 -march=native -mtune=native -pthread -ffast-math -fgcse-las`.

[19]The code is available at `https://github.com/obi1kenobi/laika` [97].

Scheduler	`data_t`	`sched_t`	`vertex_t`
Lax	32	0	48
BSP	64	0	80
Laika	32	32	80
Prism	32	0	48
JP	32	32	80

Table 5-25: Data structure memory usage in bytes for each scheduler and each component of the data at each vertex: `data_t`, `sched_t`, and `vertex_t`.

Graph	*Size*	$\|V\|$	$\|E\|$	$\|E\| / \|V\|$
dragon	0	106,140	1,200,470	11.31
	1	392,839	4,793,440	12.20
	2	1,637,007	22,054,694	13.47
	3	6,473,215	91,245,260	14.09
bunny	0	108,726	1,272,116	11.70
	1	416,929	5,496,990	13.18
	2	1,652,984	23,372,040	14.13
	3	6,202,402	89,551,696	14.43
cube	0	105,792	1,465,792	13.85
	1	397,165	5,652,760	14.23
	2	1,698,509	24,578,428	14.47
	3	6,363,260	92,992,976	14.61
rand	0	105,792	1,543,530	14.59
	1	397,165	5,904,290	14.86
	2	1,698,509	25,689,086	15.12
	3	6,363,260	97,937,358	15.39

Table 5-26: Sizes of each size of each graph topology used in our test suite. The dragon topology is pictured in Figure 5-1. The bunny and cube topologies are pictured in Figure 5-11. The rand topology is not pictured, but is a random cube graph, described in Section 5.2.

`data_t` — The `data_t` struct is defined by the application (e.g., Mass-Spring-Dashpot model) and is completely independent of the scheduler. An in-place scheduler has one copy per vertex and a double-buffered scheduler has two.

`sched_t` — The `sched_t` struct is defined by the scheduler and is completely independent of the application. It includes whatever per-vertex data is required by the scheduler to coordinate updates between workers.

`vertex_t` — The `vertex_t` struct includes the `sched_t` and `data_t` structs as payloads and also includes a pointer into the edge array and the number of neighbors, which are used by the UPDATE function to marshal data.

The size, in bytes, of each struct for each scheduler is given in Table 5-25, where the LAX and PRISM schedulers have notably smaller overall vertex data sizes than the rest.

Relevance of random cube graphs

Our goal is to provide a scheduler for physical simulations on graphs that matter in practice. Unfortunately, since practitioners typically use mesh graphs modeling complex physical objects rather than random cube graphs, our analysis in Section 5.4 is only insightful if typical graphs share essential properties with random cube graphs in practice. In order to test the empirical relevance of analysis on the random cube graph, we generated a graph suite of four sizes of four different graphs, summarized in Table 5-26. The graph topologies labeled *dragon*, *bunny*, and *cube* were generated using `TetGen v. 5.1` [181, 182], a tetrahedralizing mesh refinement engine. `TetGen` converts a 2D surface mesh (e.g., a surface tessellated by triangles, in this case supplied by the Stanford Computer Graphics Laboratory [1]) into a corresponding 3D mesh tessellated by non-overlapping tetrahedra. `TetGen` allows the user to specify the maximum edge length in the model. Through a binary search for each graph topology, we found the maximum edge length that leads to the desired number of vertices. We generated four sizes of each graph topology, where the number of vertices increases roughly by a multiple of 4 for each size. The graph topology labeled *rand* in Table 5-26 is a random cube graph generated by our own generator.[20]

We see from Figure 5-10 that typical graphs (i.e., those generated by `TetGen`: labeled dragon, bunny, and cube, respectively) have cache behavior which closely tracks that of the random cube graph, labeled rand. In particular, the rand and cube graphs, which occupy 100% of the unit cube have nearly identical cache miss rate curves, whereas the dragon and bunny graphs are shifted up, meaning they have a constant multiple more misses than the rand or cube graph at any given cache size, a consequence of Lemma 36. Furthermore, in Table 5-27, we see that measured serial and parallel runtimes, the metrics we ultimately care about, are quite consistent across each graph topology for all schedulers.

[20]The code is available at `https://github.com/obi1kenobi/laika` [97].

		Random (ρ_R)				Hilbert (ρ_H)				
Scheduler	*Graph*	T_1	T_{12}	T_S/T_1	T_1/T_{12}	T_1	T_{12}	T_S/T_1	T_1/T_{12}	T_S/T_{12}
Lax	dragon	89.4	9.99	1.00	8.95	19.9	1.70	4.50	11.72	52.68
	bunny	89.2	9.97	1.00	8.94	19.9	1.70	4.49	11.67	52.35
	cube	89.2	9.92	1.00	9.00	20.0	1.66	4.47	12.03	53.79
	rand	89.1	9.95	1.00	8.96	19.4	1.63	4.60	11.86	54.57
BSP	dragon	91.7	10.23	0.97	8.97	21.4	1.94	4.17	11.03	46.00
	bunny	91.6	10.19	0.97	8.99	21.5	1.95	4.16	10.99	45.67
	cube	91.5	10.22	0.97	8.96	21.0	1.89	4.25	11.12	47.28
	rand	91.5	10.20	0.97	8.97	20.7	1.85	4.30	11.21	48.18
Laika	dragon	93.8	10.66	0.95	8.79	22.4	2.02	4.00	11.07	44.23
	bunny	93.7	10.63	0.95	8.81	22.5	2.11	3.97	10.66	42.28
	cube	93.6	10.60	0.95	8.83	21.7	2.00	4.11	10.86	44.61
	rand	93.5	10.36	0.95	9.03	21.3	1.94	4.19	10.98	46.06
Prism	dragon	96.7	11.05	0.92	8.75	34.3	3.79	2.60	9.05	23.57
	bunny	96.3	11.02	0.93	8.74	34.3	3.83	2.60	8.96	23.29
	cube	96.4	11.01	0.93	8.75	32.9	3.59	2.71	9.16	24.87
	rand	96.4	11.03	0.92	8.74	34.2	3.77	2.61	9.07	23.66
JP	dragon	62.6	7.55	1.43	8.28	31.2	5.48	2.87	5.69	16.31
	bunny	62.4	7.56	1.43	8.25	32.2	6.64	2.77	4.84	13.43
	cube	62.3	7.43	1.43	8.39	28.2	6.60	3.17	4.27	13.52
	rand	58.4	6.83	1.53	8.55	28.0	6.34	3.18	4.41	14.05

Table 5-27: Runtimes of all data-graph computation schedulers across all graph topologies for the largest size (i.e., size 3) of each running the Mass-Spring-Dashpot model. The multi-column with heading "Random (ρ_R)" lists runtimes for graphs using randomly ordered vertices. In particular, T_S is the serial runtime of Lax, which serves as the baseline implementation. The multi-column with heading "Hilbert (ρ_H)" lists runtimes for graphs whose vertices are ordered by the Hilbert priority function. The columns labeled "T_k" list runtimes of the schedulers running with k workers. The columns labeled "T_S/T_1" list the speedup of the serial scheduler over the baseline. The columns labeled "T_1/T_{12}" list the parallel speedup of the scheduler when running on 12 cores.

Performance autopsy using event counters

Table 5-28 demonstrates the first bit of evidence that Laika successfully transformed the Mass-Spring-Dashpot model into a compute-bound problem. We can see that as the size of each graph increases from the smallest to the largest size, corresponding to a range of 29.8MB to 1.9GB, the random ordering (the multi-column under "Random (ρ_R)") timings

		Random (ρ_R)				Hilbert (ρ_H)				
Scheduler	*Size*	T_1	T_{12}	T_S/T_1	T_1/T_{12}	T_1	T_{12}	T_S/T_1	T_1/T_{12}	T_S/T_{12}
Lax	0	34.0	3.61	1.00	9.43	19.6	1.79	1.74	10.95	19.01
	1	54.1	5.64	1.00	9.58	19.8	1.75	2.72	11.33	30.86
	2	78.8	8.86	1.00	8.89	19.7	1.70	3.99	11.63	46.39
	3	89.2	9.96	1.00	8.96	19.8	1.67	4.51	11.82	53.34
BSP	0	34.6	3.96	0.98	8.73	21.1	2.01	1.62	10.45	16.90
	1	66.4	7.25	0.81	9.16	21.3	1.99	2.54	10.71	27.17
	2	85.0	9.59	0.93	8.86	21.2	1.94	3.71	10.95	40.64
	3	91.6	10.21	0.97	8.97	21.2	1.91	4.22	11.09	46.77
Laika	0	37.4	5.19	0.91	7.21	22.4	2.10	1.52	10.64	16.18
	1	75.0	7.90	0.72	9.50	24.0	2.19	2.25	10.95	24.65
	2	92.8	9.98	0.85	9.30	23.2	2.07	3.39	11.22	38.03
	3	93.6	10.56	0.95	8.87	22.0	2.02	4.07	10.89	44.27
Prism	0	37.0	4.48	0.92	8.27	25.3	3.16	1.35	7.99	10.76
	1	65.2	7.08	0.83	9.21	33.7	3.69	1.60	9.14	14.65
	2	87.2	10.27	0.90	8.48	34.1	3.95	2.31	8.63	19.96
	3	96.4	11.03	0.93	8.74	33.9	3.74	2.63	9.06	23.84
JP	0	37.9	6.77	0.90	5.60	29.0	5.47	1.17	5.30	6.23
	1	51.1	7.29	1.06	7.01	30.4	5.67	1.78	5.37	9.54
	2	58.4	7.99	1.35	7.31	30.2	5.97	2.61	5.06	13.20
	3	61.4	7.34	1.45	8.37	29.8	6.25	2.99	4.77	14.28

Table 5-28: Geometric mean of runtimes across all graph topologies for each scheduler and each size running the Mass-Spring-Dashpot model. The column headings are equivalent to those in Table 5-27.

for all schedulers increase dramatically. Whereas, when using the Hilbert priority function to order the vertices (the multi-column under "Hilbert (ρ_H)"), the timings are comparatively insensitive to graph size. Furthermore, we see in Table 5-29 that even when the graph fits entirely in the L1 cache, the number of instructions per cycle issued by the processor (i.e., 1.377 instructions / cycle) is only 8% higher than with a large graph (i.e., 1.272 instructions / cycle).

In order to get a better sense of what hardware limitations govern the performance of each scheduler, we measured several performance counters [115, Ch. 18] (e.g., the number of instruction, cycles, branch mispredictions, last level cache (LLC) misses, data translation

$\lvert V \rvert$	$\lvert E \rvert / \lvert V \rvert$	Graph Size	Fits in?	$\frac{\text{Instructions}}{\text{Cycle}}$
64	15.21	19.4 KB	L1 Cache	1.377
512	14.88	155.6 KB	L2 Cache	1.311
32,768	14.67	9.9 MB	L3 Cache	1.296
108,792	14.59	29.8 MB	dTLB	1.285
6,363,260	15.39	1.9 GB	Memory	1.272

Table 5-29: Instructions per cycle in a serial execution of the Lax data-graph computation scheduler as a function of graph size, for increasing sizes of random cube graphs. The column"$\lvert V \rvert$" shows the number of vertices in the graph and the average degree is listed under the heading "$\lvert E \rvert / \lvert V \rvert$". The column "Graph Size" shows the total memory space allocated for the vertex array and edge array. For reference, the smallest memory structure (e.g., L1 Cache, Memory, etc.) that can hold the entire is listed under the heading "Fits in?" Finally, the instruction throughput in instructions per cycle is given under the heading "Instructions / Cycle."

lookaside buffer (dTLB) misses, etc.) using the linux utility, `perf stat v. 3.13.11`. A summary of such measurements, using the largest size (i.e., size 3) of the "rand" graph as a test input, can be found in Tables 5-30 and 5-31.

We combine information from the performance counters for the LLC with measured runtimes to estimate memory bandwidth used in each configuration, listed under the heading "Gigabytes / Second" (GB/s) in Table 5-30. Since the Intel Xeon [111] test system can sustain 64GB/s peak memory bandwidth, we can see that in all scenarios the performance falls far short of the maximum.

The failure to saturate memory bandwidth using random ordering may be due to the fact that all schedulers suffer nearly 1 dTLB miss per edge, which has a high latency penalty [104, App. B]. Furthermore, all schedulers, except JP, incur roughly 2 LLC misses per edge, which may seem puzzling since the data portion of the `vertex_t` struct is only 32B and would fit on a single cache line. Such dTLB misses also frequently incur an additional LLC miss: roughly 500,000 pages are used to map the test graph (i.e., size-3 rand), which consumes roughly 4MB of memory space, much of which would be pushed out of the LLC by the actual graph state.[21] The fact that JP is faster than the other schedulers for both serial and parallel

[21]The dTLB is a small cache of translations between virtual memory and physical memory. When a dTLB miss occurs because e.g., a virtual address being loaded is not present, a "page walk" occurs. A page walk traverses a tree in memory, called the "page table," which organizes the mapping between virtual and physical memory. The leaves of this tree data structure reside in memory and are cached (e.g., in L3 cache) like any other memory state, and thus compete for cache space with the data in the application itself. Thus, a dTLB miss could cause extra LLC misses and latency in the process of retrieving the leaf of the page table.

Scheduler	*Workers*	Nanoseconds / Edge	Gigabytes / Second	LLC Misses / Edge	dTLB Misses / Edge
		ρ_R / ρ_H	ρ_R / ρ_H	ρ_R / ρ_H	ρ_R / ρ_H
Lax	1	89.48 / 19.75	1.49 / 0.62	1.91 / 0.015	0.79 / 9.81e-5
	12	9.98 / 1.69	13.42 / 8.03	1.92 / 0.035	0.82 / 1.39e-4
BSP	1	91.75 / 21.21	1.50 / 0.71	1.94 / 0.023	0.88 / 8.68e-5
	12	10.23 / 1.92	13.55 / 8.75	1.96 / 0.051	0.90 / 1.01e-4
Laika	1	94.47 / 22.17	1.50 / 0.88	2.01 / 0.092	0.93 / 7.10e-4
	12	10.57 / 2.03	13.37 / 9.09	2.00 / 0.078	0.96 / 1.17e-3
Prism	1	96.62 / 33.94	1.57 / 1.87	2.19 / 0.818	0.80 / 4.49e-4
	12	11.06 / 3.73	13.74 / 18.20	2.20 / 0.885	0.83 / 4.85e-4
JP	1	61.58 / 29.92	1.04 / 0.73	0.79 / 0.119	0.89 / 2.21e-4
	12	7.30 / 6.45	9.03 / 3.84	0.82 / 0.165	0.87 / 9.92e-4

Table 5-30: Memory-level hardware performance counter measurements of data-graph computation schedulers executing the Mass-Spring-Dashpot model on the size-3 rand graph. Each scheduler is measured both with 1 and 12 workers, given under the *Workers* heading. The subheadings ρ_R and ρ_H imply that the data for random ordering and Hilbert ordering, respectively, can be found on each side of the slash. The column "Nanoseconds / Edge" shows the nanoseconds required to execute the data-graph computation divided by the number of edges, as measured using the `clock_gettime` function in the `time.h` C++ header. The column "LLC Misses / Edge" shows the average number of measured LLC load, store, and prefetch misses in a single round, divided by the number of edges. The column "dTLB Misses / Edge" shows the average number of measured dTLB load and store misses in a single round, divided by the number of edges. The column "Gigabytes / Second" shows the ratio of the sum of the LLC load, store, and prefetch misses, in gigabytes, divided by time.

executions with random ordering corroborates the observation that it incurs comparatively few LLC misses. It would seem that JP traverses the graph in a way that exploits the locally-connected nature of the graph, even though the vertices are not explicitly ordered to expose it. JP also requires an atomic Dec-And-Fetch instruction per edge to coordinate updates in the priority dag (see Figure 5-2 for details), a significant overhead that limits JP's effectiveness relative to the other schedulers.

The schedulers also fail to saturate memory bandwidth using Hilbert ordering, but for a different reason: they use the cache more efficiently and are much faster (e.g., Laika is 5.23 times faster with Hilbert ordering than with random ordering). All schedulers incur many fewer LLC and dTLB misses per edge under Hilbert ordering, a consequence of the cache advantages detailed theoretically in Section 3.4. Prism is an outlier in that it still incurs

Scheduler	*Workers*	Nanoseconds / Edge	Instructions / Cycle	Instructions / Edge	Branch Misses / Edge
		ρ_R / ρ_H	ρ_R / ρ_H	ρ_R / ρ_H	ρ_R / ρ_H
LAX	1	89.48 / 19.75	0.28 / 1.25	65.7 / 65.7	0.101 / 0.096
	12	9.98 / 1.69	0.21 / 1.23	66.9 / 66.3	0.097 / 0.095
BSP	1	91.75 / 21.21	0.27 / 1.16	65.8 / 65.7	0.096 / 0.092
	12	10.23 / 1.92	0.21 / 1.09	66.8 / 66.0	0.096 / 0.093
LAIKA	1	94.47 / 22.17	0.28 / 1.14	71.4 / 68.1	0.167 / 0.100
	12	10.57 / 2.03	0.21 / 1.10	72.3 / 71.0	0.171 / 0.099
PRISM	1	96.62 / 33.94	0.26 / 0.73	66.1 / 65.9	0.095 / 0.094
	12	11.06 / 3.73	0.21 / 0.58	74.1 / 69.1	0.101 / 0.098
JP	1	61.58 / 29.92	0.47 / 1.01	77.6 / 80.1	0.282 / 0.381
	12	7.30 / 6.45	0.46 / 1.25	105.4 / 256.4	0.303 / 0.491

Table 5-31: Instruction-level hardware performance counter measurements of data-graph computation schedulers executing the Mass-Spring-Dashpot model on the size-3 rand graph. Each scheduler is measured both with 1 and 12 workers, given under the *Workers* heading. The subheadings ρ_R and ρ_H imply that the data for random ordering and Hilbert ordering, respectively, can be found on each side of the slash. The column "Nanoseconds / Edge" shows the nanoseconds required to execute the data-graph computation divided by the number of edges, as measured using the `clock_gettime` function in the `time.h` C++ header. The column "Instructions / Cycle" shows the measured number of x86 insructions divided by the number of measured cycles, as measure by `perf stat`, and "Instructions / Edge" and "Branch Misses / Edge" were both measured the same way.

nearly 1 LLC miss per edge: it reads the graph approximately once for every color used to color the graph, typically 3–5 colors for mesh graphs. For example, a vertex v of color c reads inputs of different colors, however the vertices in $N(v)$ cannot be updated until all of the vertices of color c have been updated, squandering the opportunity for temporal locality.

In Table 5-31 we confirm our expectations about the instruction-level behavior of each scheduler. For example, LAX, BSP, and PRISM all have instructions and branch misses per edge that are largely independent of the ordering, since each has ample parallelism and predictable allocation of work to each worker. By contrast, LAIKA has more instructions and branch misses per edge when it operates on randomly ordered vertices, which is outside its intended operating conditions. LAX, BSP, and LAIKA all lose efficiency in instructions per cycle with random ordering when going from 1 worker to 12. By contrast, with Hilbert ordering the efficiency in instructions per cycle is approximately the same between 1 and 12 workers, and 5–6 times higher than with random ordering.

5.7 Conclusion

We are optimistic that the vertex reordering and scheduling techniques will be widely adopted in the physical simulation community, given the provably good bounds we provide for mesh graphs and the empirically demonstrated fact that we make such problems compute-bound. However, we wonder if a more general scheduling algorithm may emerge out of this work on Laika. In particular, we rely on the Hilbert space-filling curve to order vertices such that relatively few edges cross between chunks (i.e., contiguous blocks of vertices). Could we not use a graph clustering algorithm [64, 122] on generic graphs with locality (e.g., power law graphs [44]) or on mesh graphs where we do not know their position in 3D space to form the chunks? Such a clustering may suffice in limiting the inter-chunk edge crossings and yield high performance on a broader class of input graphs. In particular, we note that JP incurs suspiciously few LLC misses on randomly ordered mesh graphs in Table 5-30. We leave as an open question why JP performs so well in this respect and if an ordering of vertices based on JP's traversal of the graph would suffice for ordering graphs where the Hilbert ordering is not possible.

Finally, we acknowledge that support for dynamically changing graphs (i.e., those whose structure changes over time) would be a valuable extension to Laika. For instance, it may be useful to dynamically refine a mesh graph in a data-dependent way. Our future work includes an investigation of what data-structures and algorithms would be required to support such graphs. Furthermore, while most physical simulations are static data-graph computations (i.e., they update every vertex every round) there may be efficiency gains in executing such physical simulations using data dependent subsets of vertices, e.g., using dynamic data-graph computations as in Table 3-3.

Chapter 6

Conclusion

"You can have a second computer once you have shown you know how to use the first one."

Paul Barham

In reflection on what I set out to accomplish in my PhD studies, I reviewed the Statement of Purpose that accompanied my application to MIT, which opened with the following paragraph:

> Determined to outclass a French magician, Wolfgang von Kempelen unveiled The Mechanical Turk in Vienna. Skeptics were invited to inspect the gears, cogs and other complications that were housed in the desk. The chessboard on top and the statue of the Turkish man which stood before it also defied inspection, but it did little to quell the incredulity of the audience. For several decades, this Automaton Chess Player defeated nobles and statesmen, including Benjamin Franklin, and its abilities seemed as attributable to magic as science. The claim that Napoleon responded to his ignominious defeat by invading Russia has never been confirmed, though it must have delighted the chess master / co-conspirator who surreptitiously controlled The Turk from within. In an ironic turnabout, the modern equivalent of his hoax might be that an ordinary person defeats a chess master with the help of an actual Automaton Chess Player concealed about their person. An iPhone would probably be sufficient; it has roughly the same computational power as Deep Blue, the supercomputer that defeated Garry Kasparov.
>
> — William Hasenplaugh, 2010

I was, and still am, awed by how the exponential march of technology has fundamentally changed society and accelerates discovery in the sciences. Much of this historical acceleration was due to increasing clock speeds: ENIAC [88] ran at 5KHz in 1946 and Intel's Haswell processor [114] ran at 4GHz in 2014, a mere 800,000 times faster. However, in the last decade, processor clock rates have stalled: processor performance now comes in the form of more cores, not faster cores. Thus, to make use of so many cores and keep the scientific revolution alive, one needs parallelism. But parallel programming is famously hard, so the parallel programming community has taken up the task of making it accessible to more people [15,16,21,28,62,63,71,72,82,100,110,161,164,185,200]. An overwhelming concensus of this effort is that determinism is essential to writing correct parallel programs. With determinism, one can reason about a parallel program in the same way that they reason about the equivalent serial program, which is already adequately difficult.

The goal in developing Prism and Laika was to provide tools that enable practitioners who are not parallel programming experts to solve problems in their respective domains with guarantees about performance and correctness. We have done that through a combination of theoretical analysis and empirical evaluation of performance engineering techniques. The next challenge is to evolve those systems, making them expressive enough to describe broader classes of problems that matter in the real world. I will be satisfied if my 2 year old daughter Ella is able to bend a cluster of 1,000,000 processors to her will for a future high school science project without worrying about determinacy races.[1]

[1]Ella, if you are reading this, no pressure.

Appendix A

The Cilk Model of Multithreading

All code in this thesis was implemented in Cilk Plus [112], a dynamic multithreading concurrency platform. This section provides background on the dag model of multithreading that embodies this and other similar concurrency platforms, including MIT Cilk [77], Cilk++ [135], Fortress [3], Habenero [12, 41], Hood [27], Java Fork/Join Framework [131], Task Parallel Library [134], Threading Building Blocks [169], and X10 [46]. We review the Cilk model of multithreading, the notions of work and span, and the basic properties of the work-stealing runtime systems underlying these concurrency platforms. We briefly discuss worker-local storage, which PRISM's multibag data structure uses to achieve efficiency.

The Cilk model of multithreading [25, 26] is described in tutorial fashion in [52, Ch. 27]. The model views the executed computation resulting from running a parallel program as a ***computation dag*** in which each vertex denotes an instruction, and edges denote parallel control dependencies between instructions. To analyze the theoretical performance of a multithreaded program, such as PRISM, we assume that the program executes on an ***ideal parallel computer***, where each instruction executes in unit time, the computer has ample bandwidth to shared memory, and concurrent reads and writes incur no overheads due to contention.

We assume that algorithms for the dag model are expressed using the Cilk-like primitives [52, Ch. 27] **spawn**, **sync**, and **parallel for**. The keyword **spawn** when preceding a function call F allows F to execute in parallel with its ***continuation*** — the statement immediately after the spawn of F. The complement of **spawn** is the keyword **sync**, which acts as a local barrier and prevents statements after the **sync** from executing until all earlier spawned functions return. These keywords can be used to implement other convenient parallel control constructs, such as the **parallel for** loop, which allows all of its iterations to operate logically in parallel. The work of a **parallel for** loop with n iterations is the total number of instructions in all executed iterations. The span is $\Theta(\lg n)$ plus the maximum span of any loop iteration. The $\Theta(\lg n)$ span term comes from the fact that the runtime system executes the loop iterations using parallel divide-and-conquer, and thus fans out the iterations as a balanced binary tree in the dag.

Work-span analysis

Given a multithreaded program whose execution is modeled as a dag A, we can bound the P-processor running time $T_P(A)$ of the program using ***work-span analysis*** [52, Ch. 27]. Recall that the work $T_1(A)$ is the number of instructions in A, and that the span $T_\infty(A)$ is the length of a longest path in A. Greedy schedulers [33, 69, 94] can execute a deterministic program with work T_1 and span T_∞ on P processors in time T_P satisfying

$$\max\{T_1/P, T_\infty\} \leq T_p \leq T_1/P + T_\infty \ , \tag{A.1}$$

and a similar bound can be achieved by more practical "work-stealing" schedulers [25, 26]. The ***speedup*** of an algorithm on P processors is T_1/T_P, which Inequality (A.1) shows to be at most P in theory. The ***parallelism*** T_1/T_∞ is the greatest theoretical speedup possible for any number of processors.

Work-stealing runtime systems

Runtime systems underlying concurrency platforms that support the dag model of multithreading usually implement a ***work stealing*** scheduler [26, 38, 99], which operates as follows. The runtime system initially allocates as many operating-system threads, called ***workers***, as there are processors. Each worker keeps a ***ready queue*** of tasks that can operate in parallel with the task it is currently executing. Whenever the execution of code generates parallel work, the worker puts the excess work into the queue. Whenever it needs work, it fetches work from its queue. When a worker's ready queue runs out of tasks, however, the worker becomes a ***thief*** and "steals" work from another ***victim*** worker's queue. If an application exhibits sufficient parallelism compared to the actual number of workers/processors, one can prove mathematically that the computation executes with linear speedup.

Worker-local storage

We refer to memory that is private to a particular worker thread as ***worker-local storage***. In a P-processor execution of a parallel program, a worker-local variable x can be implemented using a shared-memory array of length P. A worker accesses its local copy of x using a runtime-provided worker identifier to index the array of worker-local copies of x. The Cilk Plus runtime system, for example, provides the `__cilkrts_get_worker_number()` API call, which returns an integer identifying the current worker. Our implementation of PRISM assumes the existence of a runtime-provided GET-WORKER-ID function that executes in $\Theta(1)$ time and returns an integer from 0 to $P-1$. Other strategies for implementing worker-local storage exist that are comparable to the strategy outlined here.

Bibliography

[1] Stanford University Computer Graphics Laboratory. `graphics.stanford.edu/data/3Dscanrep`, August 2014.

[2] L. Adams and J. Ortega. A multi-color SOR method for parallel computation. In *International Conference on Parallel Processing*, pages 53–56, 1982.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification version 1.0. Technical report, Sun Microsystems, 2008.

[4] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin. A comparison of parallel graph coloring algorithms. Technical report, Syracuse University, 1995.

[5] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583, Dec. 1986.

[6] E. M. Arkin and E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Appl. Math.*, 18(1):1–8, Nov. 1987.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, EECS Department, University of California, Berkeley, December 2006.

[8] Associated Press. Soviet fires new satellite, carrying dog; half-ton sphere is reported 900 miles up, 1957.

[9] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[10] M. Bader. *Space-Filling Curves: An Introduction with Applications in Scientific Computing.* Springer Publishing Company, 2012.

[11] L. Barenboim and M. Elkin. Distributed (Δ+1)-coloring in linear (in Δ) time. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 111–120, New York, NY, USA, 2009. ACM.

[12] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşirlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero multicore software

research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 735–736, New York, NY, USA, 2009. ACM.

[13] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 61–70, New York, NY, USA, 2007. ACM.

[14] S. Berchtold, C. Böhm, B. Braunmüller, D. A. Keim, and H. Kriegel. Fast parallel similarity search in multimedia databases. *SIGMOD Rec.*, 26(2):1–12, June 1997.

[15] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. *SIGPLAN Not.*, 45(3):53–64, Mar. 2010.

[16] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. *SIGPLAN Not.*, 44(10):81–96, Oct. 2009.

[17] D. Bernoulli. *Hydrodynamica sive de viribus et motibus fluidorum commentarii.* Johann Reinhold Dulsecker, 1738.

[18] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[19] G. E. Blelloch. Prefix sums and their applications. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[20] G. E. Blelloch. NESL: A nested data-parallel language. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[21] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, Mar. 1996.

[22] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. *SIGPLAN Not.*, 47(8):181–192, Feb. 2012.

[23] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 308–317, New York, NY, USA, 2012. ACM.

[24] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 3–16, New York, NY, USA, 1991. ACM.

[25] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, Feb. 1998.

[26] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[27] R. D. Blumofe and D. Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, University of Texas at Austin, 1999.

[28] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[29] B. Bollobás. *Modern Graph Theory.* Springer, 1998.

[30] G. Boole. *A Treatise On The Calculus Of Finite Differences.* Macmillan and Company, London, England, 2nd edition, 1872.

[31] A. Brandt. Multi-level adaptive solutions to boundary value problems. *Mathematics of Computation*, 31(138):333–390, April 1977.

[32] D. Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, Apr. 1979.

[33] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, Apr. 1974.

[34] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Houston, TX, USA, 1992. UMI Order No. GAX92-34388.

[35] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, Apr. 1998.

[36] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, WADS '99, pages 37–48, London, UK, UK, 1999. Springer-Verlag.

[37] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 233–244, New York, NY, USA, 2009. ACM.

[38] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[39] A. R. Butz. Space filling curves and mathematical programming. *Information and Control*, 12(4):314 – 330, 1968.

[40] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Comput.*, 38(10-11):576–594, Oct. 2012.

[41] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[42] G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101, June 1982.

[43] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, Jan. 1981.

[44] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, 2004.

[45] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Trans. Softw. Eng.*, 26(3):197–211, Mar. 2000.

[46] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[47] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[48] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[49] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 206–219, New York, NY, USA, 1986. ACM.

[50] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983.

[51] M. E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pages 139–146, New York, NY, USA, 1963. ACM.

[52] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[53] R. Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bulletin of the American Mathematical Society*, 49(1):1–23, 1943.

[54] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, Oct. 1971.

[55] D. Cutting and M. Cafarella. Hadoop. `http://hadoop.apache.org`, 2005.

[56] G. B. Dantzig. Origins of the simplex method. In S. G. Nash, editor, *A History of Scientific Computing*, page 11. ACM, New York, NY, USA, 1990.

[57] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[58] A. De Morgan. *Formal Logic.* Taylor and Walton, London, 1847.

[59] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[60] J. E. Dennis, Jr. and T. Steihaug. On the successive projections approach to least-squares problems. *SIAM J. Numer. Anal.*, 23(4):717–733, Aug. 1986.

[61] P. Deuflhard and F. Bornemann. *Scientific Computing with Ordinary Differential Equations.* Springer, New York, 2002.

[62] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. *SIGPLAN Not.*, 44(3):85–96, Mar. 2009.

[63] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A relaxed consistency deterministic computer. *SIGPLAN Not.*, 47(4):67–78, Mar. 2011.

[64] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[65] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM, ACM.

[66] K. Diks. A fast parallel algorithm for six-colouring of planar graphs. In *Proceedings of the 12th Symposium on Mathematical Foundations of Computer Science 1986*, pages 273–282, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[67] G. L. Dirichlet. Sur une nouvelle méthode pour la détermination des intégrales multiples. *Journal de Mathématiques Pures et Appliquées*, 1839.

[68] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 174–183, New York, NY, USA, 1993. ACM.

[69] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, Mar. 1989.

[70] C. Feichtinger, J. Habich, H. Köstler, G. Hager, U. Rüde, and G. Wellein. A flexible patch-based lattice Boltzmann parallelization approach for heterogeneous GPU-CPU clusters. *Parallel Computing*, 37(9):536–549, September 2011.

[71] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 1–11, New York, NY, USA, 1997. ACM.

[72] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[73] M. Fischetti, S. Martello, and P. Toth. The fixed job schedule problem with spread-time constraints. *Oper. Res.*, 35(6):849–858, Nov. 1987.

[74] B. J. Ford. *Secret Weapons: Technology, Science, and the Race to Win World War II*. Osprey Publishing, 2011.

[75] G. C. Fox. *Numerical Algorithms for Modern Parallel Computer Architectures*. Springer-Verlag, 1988.

[76] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[77] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[78] S. H. Fuller and L. I. Millett. *The Future of Computing Performance: Game Over or Next Level?* National Academy Press, Washington, DC, USA, 2011.

[79] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[80] A. H. Gebremedhin and F. Manne. Scalable parallel graph coloring algorithms. *Concurrency – Practice and Experience*, 12(12):1131–1146, 2000.

[81] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen. ColPack: Software for graph coloring and related problems in scientific computing. *ACM Trans. Math. Softw.*, 40(1):1:1–1:31, Oct. 2013.

[82] P. B. Gibbons. A more practical PRAM model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 158–168, New York, NY, USA, 1989. ACM.

[83] H. Giefers, C. Plessl, and J. Förstner. Accelerating finite difference time domain simulations with reconfigurable dataflow computers. *SIGARCH Comput. Archit. News*, 41(5):65–70, December 2013.

[84] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in Matlab: Design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, Jan. 1992.

[85] R. K. Gjertsen, Jr., M. T. Jones, and P. E. Plassmann. Parallel heuristics for improved, balanced graph colorings. *J. Parallel Distrib. Comput.*, 37:171–186, sep 1996.

[86] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discret. Math.*, 1(4):434–446, Oct. 1988.

[87] M. Goldberg and T. Spencer. A new parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 18(2):419–427, Apr. 1989.

[88] H. H. Goldstine and A. Goldstine. The Electronic Numerical Integrator and Computer (ENIAC). *IEEE Ann. Hist. Comput.*, 18(1):10–16, March 1996.

[89] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, 2(2):205–224, 1965.

[90] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[91] N. Goodnight, C. Wolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[92] C. Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing*, 5(5):794–797, 1996.

[93] W. B. Gragg. On extrapolation algorithms for ordinary initial value problems. *SIAM Journal on Numerical Analysis*, 2(3):384–403, 1965.

[94] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[95] L. F. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* PhD thesis, Yale University, 1987.

[96] P. Gruevski, W. Hasenplaugh, C. E. Leiserson, and J. J. Thomas. Cache-efficient data-graph computations for physical simulations. Unpublished Manuscript, 2016.

[97] P. Gruevski, W. Hasenplaugh, and J. J. Thomas. Laika: A data-graph computation scheduler for physical simulations. `https://github.com/obi1kenobi/laika`, November 2015.

[98] G. Haase, M. Liebmann, and G. Plank. A Hilbert-order multiplication scheme for unstructured sparse matrices. *Int. J. Parallel Emerg. Distrib. Syst.*, 22(4):213–220, Jan. 2007.

[99] R. 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, New York, NY, USA, 1984. ACM.

[100] R. H. Halstead, Jr. MultiLisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[101] D. Harlacher, H. Klimach, S. Roller, C. Siebert, and F. Wolf. Dynamic load balancing for unstructured meshes on space-filling curves. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1661–1669, May 2012.

[102] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 166–177, New York, NY, USA, 2014. ACM.

[103] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 145–156, New York, NY, USA, 2010. ACM.

[104] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[105] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[106] D. Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. *Mathematische Annalen*, 1891.

[107] F. L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematical Physics*, 1927.

[108] R. Hooke. *Lectures De Potentia Restitutiva.* London, Printed for John Martyn, London, England, 1678.

[109] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 269–276, New York, NY, USA, 1999. ACM Press.

[110] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 333–334, Washington, DC, USA, 2011. IEEE Computer Society.

[111] Intel® Corporation. Intel® Xeon® Processor X5650. `http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI`, February 2010.

[112] Intel® Corporation. The Threading Building Blocks. `https://software.intel.com/en-us/intel-tbb`, 2012.

[113] Intel® Corporation. Intel® Cilk Plus. `https://software.intel.com/en-us/intel-cilk-plus`, 2013.

[114] Intel® Corporation. Intel® Core^{TM} i7-4790. `http://ark.intel.com/products/80806/Intel-Core-i7-4790-Processor-8M-Cache-up-to-4_00-GHz`, April 2014.

[115] Intel® Corporation. Intel® 64 and ia-32 architectures software developer's manual v3. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html#`, 2015.

[116] K. E. Iverson. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA, 1962.

[117] J. Jiang, M. Mitzenmacher, and J. Thaler. Parallel peeling algorithms. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 319–330, New York, NY, USA, 2014. ACM.

[118] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, May 1993.

[119] M. T. Jones and P. E. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Comput.*, 20(5):753–773, May 1994.

[120] W. Kahan. *Gauss-Seidel Methods of Solving Large Systems of Linear Equations.* PhD thesis, University of Toronto, 1958.

[121] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 154–165, New York, NY, USA, 2014. ACM.

[122] G. Karypis and V. Kumar. METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. Technical report, University of Minnesota, 1995.

[123] W. F. Kern and J. R. Bland. *Solid Mensuration.* John Wiley & Sons, Inc., 1954.

[124] F. Kjolstad, S. Kamil, J. Ragan-Kelley, D. I. W. Levin, S. Sueda, D. Chen, E. Vouga, D. M. Kaufman, G. Kanwar, W. Matusik, and S. Amarasinghe. Simit: A Language for Physical Simulation. Technical report, Massachusetts Institute of Technology, May 2015.

[125] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The High Performance Fortran Handbook.* MIT Press, Cambridge, MA, USA, 1994.

[126] F. Kuhn. Weak graph colorings: Distributed algorithms and applications. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 138–144, New York, NY, USA, 2009. ACM.

[127] F. Kuhn and R. Wattenhofer. On the complexity of distributed graph coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 7–15, New York, NY, USA, 2006. ACM.

[128] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *International Symposium on Performance Analysis of Systems and Software*, pages 65–76, 2009.

[129] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[130] C. Lasser and S. M. Omohundro. The essential Lisp manual. Technical report, Thinking Machines, Cambridge, MA USA, 1986.

[131] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

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

[133] I. A. Lee, A. Shafi, and C. E. Leiserson. Memory-mapping support for reducer hyperobjects. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 287–297, New York, NY, USA, 2012. ACM.

[134] D. Leijen and J. Hall. Optimize managed code for multi-core machines.

[135] C. E. Leiserson. The Cilk++ concurrency platform. *J. Supercomput.*, 51(3):244–257, Mar. 2010.

[136] J. Leskovec. SNAP: Stanford network analysis platform. `http://snap.stanford.edu/data/index.html`, 2013.

[137] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[138] V. Leung, E. Arkin, M. Bender, D. Bunde, J. Johnston, A. Lal, J. Mitchell, C. Phillips, and S. Seiden. Processor allocation on Cplant: achieving general processor locality using one-dimensional allocation strategies. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 296–304, 2002.

[139] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer. Beyond traditional microprocessors for geoscience high-performance computing applications. *IEEE Micro*, 31(2):41–49, March 2011.

[140] N. Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, Feb. 1992.

[141] L. Lovász, M. Saks, and W. T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Discrete Math.*, 75(1-3):319–325, Sept. 1989.

[142] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[143] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[144] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1055, Nov. 1986.

[145] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[146] D. Marx. Graph colouring problems and their applications in scheduling. *Periodica Polytechnica, Electrical Engineering*, 48(1-2):11–16, 2004.

[147] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.

[148] A. McCallum. Cora data set. `http://people.cs.umass.edu/mccallum/data.html`, 2012.

[149] D. McGrady. Avoiding contention using combinable objects, 2008.

[150] T. Mitchell. NPIC500 data set. `http://www.cs.cmu.edu/tom/10709_fall2009/NPIC500.pdf`, 2009.

[151] J. Mitchem. On various algorithms for estimating the chromatic number of a graph. *Comput. J.*, 19(2):182–183, 1976.

[152] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of Hilbert space-filling curve. Technical report, College Park, MD, USA, 1996.

[153] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, International Business Machines Co, 1966.

[154] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI'99, pages 467–475, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[155] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[156] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992.

[157] I. Newton. *Philosophiæ Naturalis Principia Mathematica.* Edmund Halley, London, England, 1687.

[158] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[159] D. Nguyen, A. Lenharth, and K. Pingali. Deterministic Galois: On-demand, portable and parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 499–512, New York, NY, USA, 2014. ACM.

[160] K. Nigam and R. Ghani. Analyzing the effectiveness and applicability of co-training. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, pages 86–93, New York, NY, USA, 2000. ACM.

[161] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. *SIGARCH Comput. Archit. News*, 37(1):97–108, Mar. 2009.

[162] K. Ono, S. Chiba, S. Inoue, and K. Minami. Performance improvement of iterative methods using a bit-representation technique for coefficient matrices. In *Proceedings of the 11th Annual Internation Meeting on High Performance Computing for Computational Science*, volume 11, July 2014.

[163] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

[164] S. S. Patil. Record of the project MAC conference on concurrent systems and parallel computation. chapter Closure Properties of Interconnections of Determinate Systems, pages 107–116. ACM, New York, NY, USA, 1970.

[165] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[166] J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Trans. Parallel Distrib. Syst.*, 7(3):288–300, Mar. 1996.

[167] S. Pinker. *The Sense of Style: The Thinking Person's Guide to Writing in the 21st Century*. Penguin, New York, NY, USA, 2014.

[168] H. Poincaré. *Les Méthodes Nouvelles de la Mécanique Céleste*. Gauthier-Villars et fils, Paris, France, 1892.

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

[170] G. Rokos, G. Peteinatos, G. Kouveli, G. Goumas, K. Kourtis, and N. Koziris. Solving the advection PDE on the Cell broadband engine. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[171] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[172] Y. Saad. *SPARSKIT: a basic tool kit for sparse matrix computations*, volume 2. Version, 1994.

[173] H. Sagan and J. Holbrook. *Space-Filling Curves*. Springer-Verlag New York, Inc., 1994.

[174] S. Salihoglu and J. Widom. GPS: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[175] A. E. Sariyüce, E. Saule, and Ü. V. Çataryürek. Improving graph coloring on distributed-memory parallel computers. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[176] R. Shelton. White Men Can't Jump. 20th Century Fox, 1992.

[177] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, Feb. 2013.

[178] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 152–163, New York, NY, USA, 2013. ACM.

[179] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM.

[180] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, pages 403–412, 2015.

[181] H. Si. TetGen. `wias-berlin.de/software/tetgen`, November 2013.

[182] H. Si. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2), January 2015.

[183] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 54–65, New York, NY, USA, 1993. ACM.

[184] P. Singla and P. Domingos. Entity resolution with Markov logic. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, pages 572–582, Washington, DC, USA, 2006. IEEE Computer Society.

[185] G. L. Steele, Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 218–231, New York, NY, USA, 1990. ACM.

[186] J. Stirling. *Methodus differentialis, sive tractatus de summation et interpolation serierum infinitarium.* J. Whiston & B. White, London, 1730.

[187] J. Stoer, R. Bulirsch, R. H. Bartels, W. Gautschi, and C. Witzgall. *Introduction to numerical analysis.* Texts in applied mathematics. Springer, New York, 2002.

[188] G. Strang. *Linear Algebra and Its Applications.* Wellesley-Cambridge Press, Wellesley, MA, 2009.

[189] G. Strang and G. J. Fix. *An Analysis of The Finite Element Method.* Prentice-Hall, Inc., 1973.

[190] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[191] M. Szegedy and S. Vishwanathan. Locality based graph coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 201–207, New York, NY, USA, 1993. ACM.

[192] S. Tirthapura, S. Seal, and S. Aluru. A formal analysis of space filling curves for parallel domain decomposition. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 505–512, Washington, DC, USA, 2006. IEEE Computer Society.

[193] A. M. Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[194] V. Turner, J. F. Gantz, D. Reinsel, and S. Minton. The digital universe of opportunities: Rich data and the increasing value of the Internet of Things. White paper, EMC^2, April 2014.

[195] M. Unger. *Michelangelo: A Life in Six Masterpieces.* Simon & Schuster, 2014.

[196] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 12–21, New York, NY, USA, 1993. ACM.

[197] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

[198] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[199] D. M. Young. *Iterative Solution of Large Linear Systems.* Academic Press, New York, 1971.

[200] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. *SIGARCH Comput. Archit. News*, 37(3):325–336, June 2009.

[201] A.-J. N. Yzelman and R. H. Bisseling. A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve. In M. Günther, A. Bartel, M. Brunk, S. Schöps, and M. Striebel, editors, *Progress in Industrial Mathematics at ECMI 2010*, volume 17 of *Mathematics in Industry*, pages 627–633. Springer Berlin Heidelberg, 2012.

[202] A.-J. N. Yzelman and D. Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):116–125, 2014.

[203] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 712–721, New York, NY, USA, 1991. ACM.

[204] K. Zhang, R. Chen, and H. Chen. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA, 2015. ACM.