Chromatic Scheduling of Dynamic Data-Graph Computations

by

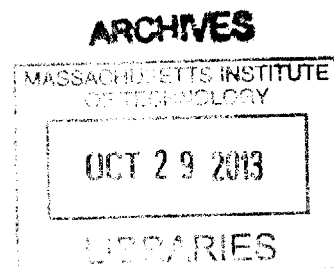
Tim Kaler

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY



May 2013
[June 2013]

Copyright 2013 Tim Kaler. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author ....
Department of Electrical Engineering and Computer Science
May 24, 2013

Certified by ....
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Dennis M. Freeman
Chairman, Department Committee on Graduate Students

Chromatic Scheduling of Dynamic Data-Graph Computations

by

Tim Kaler

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Data-graph computations are a parallel-programming model popularized by programming systems such as Pregel, GraphLab, PowerGraph, and GraphChi. A fundamental issue in parallelizing data-graph computations is the avoidance of races between computation occuring on overlapping regions of the graph. Common solutions such as locking protocols and bulk-synchronous execution often sacrifice performance, update atomicity, or determinism. A known alternative is ***chromatic scheduling*** which uses a vertex coloring of the ***conflict graph*** to divide data-graph updates into sets which may be parallelized without races. To date, however, only static data-graph computations, which do not schedule updates at runtime, have employed chromatic scheduling.

I introduce PRISM, a work-efficient scheduling algorithm for dynamic data-graph computations that uses chromatic scheduling. For a collection of four application benchmarks on a modern multicore machine, chromatic scheduling approximately doubles the performance of the lock-based GraphLab implementation, and triples the performance of GraphChi's update execution phase when enforcing determinism.

Chromatic scheduling motivates the development of efficient deterministic parallel coloring algorithms. New analysis of the Jones-Plassmann message-passing algorithm shows that only $O(\Delta + \ln \Delta \ln V / \ln \ln V)$ rounds are needed to color a graph $G = (V, E)$ with max vertex degree Δ, generalizing previous results for bounded degree graphs. A new ***log-degree ordering*** heuristic is described which can reduce the number of colors used in practice, while only increasing the number of rounds by a logrithmic factor. An efficient implementation for the shared-memory setting is described and analyzed using the CRQW contention model, showing that this algorithm performs $\Theta(V + E)$ work and has expected span $O(\Delta \ln \Delta + \ln^2 \Delta \ln V / \ln \ln V)$. Benchmarks on a set of real world graphs show that, in practice, these parallel algorithms achieve modest speedup over optimized serial code (around 4× on a 12-core machine).

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgements

Charles E. Leiserson, my advisor, has provided ~~damned~~ good advice and guidance which has influenced the contents of this thesis and my decision to remain in academia. His writing related suggestions have been especially helpful and, at times, amusing.

My coauthors William Hasenplaugh and Tao B. Schardl have made significant contributions to the content of this thesis. In particular, the main results on parallel graph coloring (chapters 6-9) would not have been possible without their work.

The Supertech research group as a whole has provided a variety of resources. It has provided an engaged audience to listen to my presentations and provide valuable feedback. It's provided a community effective researchers to serve as role models and provide examples of high quality research. Finally, its provided wonderful collaborators who have contributed their time and energy to help improve my research.

Thanks!

Contents

1 Introduction 9

I Chromatic Scheduling 17

2 Data-graph Computations 19

3 Analysis of Parallel Data-Graph Computations 25

4 Dynamic Chromatic Scheduling 29

5 Performance Evaluation of Prism 37

II Deterministic Parallel Coloring 41

6 Graph Coloring 43

7 Jones-Plassmann Coloring Algorithm 47

8 Log Degree Ordering Heuristic 53

9 Multicore Implementation of Jones-Plassmann 57

10 Conclusion 65

Chapter 1

Introduction

Graphs provide a natural representation for many systems from physics, artificial intelligence, and scientific computing. Physical systems can often be decomposed into a finite number of elements whose interactions induce a graph. Probabilistic graphical models in AI are often used to represent the dependency structure of a set of random variables. Sparse matrices may be interpreted as a graph for the purpose of performing scientific computing. Generally, these representations are data-graphs which have data associated with their vertices and edges.

A data-graph computation is an algorithm implemented as a sequence of local ***updates*** on a data-graph. Each of these updates operates on the data at a vertex, its incident edges, and its neighboring vertices, providing new data values for the vertex. A data-graph computation typically proceeds in ***rounds*** in which a set of vertices in the graph are updated. Data-graph computations may be either ***static***, where all vertices are updated in a round, or ***dynamic***, where the set of vertices for the next round is determined during the execution of the current round. For example, a dynamic data-graph computation may schedule an update at a vertex only if the value of its neighbors have changed by some threshold. Dynamic data-graph computations avoid unnecessary work, which has been shown to improve the practical performance of many applications [45, 47]. Static data-graph computations include Gibbs sampling [18,19], iterative graph coloring [14], and n-body problems such as the fluidanimate PARSEC benchmark [7]. Dynamic data-graph computations include the

Google PageRank algorithm [10], loopy belief propagation [52,56], coordinate descent [15], co-EM [54], alternating least-squares [35], singular value decomposition [26], and matrix factorization [59].

Parallelizing data-graph computations

The prevalence of data-graph computations in World Wide Web analysis, machine learning, numerical computation, and other areas has inspired several systems for parallelizing such applications, such as Pregel [49], GraphLab [45, 47], PowerGraph [27], and GraphChi [41]. These systems use three common methods to parallelize data-graph computations: bulk-synchronous updates, lock-synchronized updates, and chromatic scheduling. These methods vary in the types of data-graph computations they can express and on the semantic guarantees they provide the programmer. These differences stem mostly from each method's strategy for resolving race conditions between parallel updates.

Before discussing these three methods, let us review the two types of races that can affect a data-graph computation. A ***determinacy race*** [16] (also called a ***general race*** [53]) occurs if two parallel updates access the same memory location and one of those accesses is a write. A ***data race*** [53] exists if there is a determinacy race between two updates which hold no common mutual-exclusion locks. Data races can cause the result of a parallel data-graph computation to fail to correspond to any sequential application of updates, and a determinacy race can cause nondeterminism. A parallel data-graph computation guarantees ***update atomicity*** if the final result of the computation corresponds to some sequential application of updates. It is ***deterministic*** if all executions are guaranteed to produce the same result. Any determinacy race can cause a program to become nondeterministic, but only data-races can compromise update atomicity.

It is desirable for parallel data-graph computations to be both deterministic and guarantee update atomicity. Determinism is valuable because it makes the execution of a data-graph computation repeatable. Repeatability allows data-graph algorithms to be debugged more effectively and makes it easier to implement fault tolerance

mechanisms, such as replicated state machines, when working with distributed systems. Update atomicity is valuable because it allows for the expression of a wider class of data-graph algorithms which require updates to be applied in some sequential order in order to guarantee correctness or good convergence properties.

Bulk-synchronous The bulk-synchronous method executes a data-graph computation in parallel as a sequence of rounds. During each round, the bulk-synchronous method avoids nondeterminism by maintaining two copies of the data-graph: one read-only and the other write-only. Updates applied during the round only modify the write-only copy of the data-graph. After all updates in the round have been applied, the changes to the write-only data-graph arc propagated to the read-only copy, and the next round is ready to execute.

The bulk-synchronous method has been used to parallelize both static and dynamic data-graph computations. Pregel [49], for example, utilizes the bulk-synchronous method and also supports dynamic scheduling. Row 1 of Figure 1-1 notes that the bulk-synchronous method provides determinism, but not atomicity. For any pair of updates executed in parallel, neither see the effect of applying the other, whereas the second update would see the effect of the first if they were applied in some sequential order.

Lock-synchronized The lock-synchronized method executes a set of updates in parallel without violating update atomicity by using a locking protocol to guarantee that any two updates which access the same memory location hold a shared lock. Atomicity is provided by acquiring all necessary locks before applying the update function and then releasing them afterwards. Deadlock is avoided by requiring that each update function acquire its necessary locks in an order based upon a total ordering of all locks.

Locks can be used to parallelize both static and dynamic data-graph computations. GraphLab and PowerGraph, for example, utilize a locking protocol to guarantee update atomicity while supporting dynamic scheduling. Row 2 of Figure 1-1 notes

Method	Atomicity	Determinism
1. Bulk-synchronous	No	Yes
2. Lock-synchronized	Yes	No
3. Chromatic scheduling	Yes	Yes

Figure 1-1: Comparison of three methods used in practice to parallelize data-graph computations. Atomicity refers to the property that the data-graph computation corresponds to some sequential application of local updates. Determinism refers to the property that the computation is guaranteed to produce the same result for all possible executions.

that this method provides atomicity, but not determinism. The use of locks provides atomicity by eliminating all data-races in the program. Using locks, however, does not necessarily resolve determinacy races that can cause a data-graph computation to be nondeterministic. Indeed, races to acquire locks may result in updates executing in a different order in multiple executions.

Chromatic scheduling Chromatic scheduling [1,6] can be used to parallelize data-graph computations while guaranteeing both update atomicity and determinism. A chromatic scheduler executes updates in parallel based on the computation's ***conflict graph***—a graph which contains an edge between two vertices if updating them in parallel could violate the atomicity of either update. For many data-graph algorithms, the conflict graph is simply the undirected version of the data-graph. The conflict graph is used to assign each vertex a color such that no two vertices of the same color share an edge in the conflict graph. A chromatic scheduler sequences through the colors, applying updates to all vertices of the same color in parallel. Since the number of colors used in chromatic scheduling is inversely proportional to the parallelism, it is desirable to color the conflict graph using a few colors as possible.

Row 3 of Figure 1-1 notes that chromatic scheduling is deterministic and provides update atomicity. Chromatic scheduling provides both of these guarantees because vertices of the same color form an independent set in the conflict graph, and hence they can be updated in parallel without violating the atomicity of any update. To date, however, only static data-graph computations have employed chromatic scheduling. GraphLab, for example, supports chromatic scheduling for static data-graph computations. It does not support chromatic scheduling of dynamic data-

Version	T_1 *(s)*	T_{12} *(s)*	*Total updates*
Bulk-synchronous PageRank	37.26	6.32	18,691,620
Chromatically scheduled PageRank	12.54	2.24	7,347,401

Figure 1-2: Performance comparison of the Jacobi and Gauss-Seidel (chromatic scheduling) algorithms for computing PageRank. These data result from running the described algorithm on a "power-law" graph with 1 million vertices and 10 million edges. All tests were run on an Intel Xeon 2.67 GHz 12-core machine with 48 GB of memory.

graph computations. Indeed, the GraphLab team [46] notes, "While the chromatic engine satisfies the distributed GraphLab abstraction ..., it does not provide sufficient scheduling flexibility for many interesting applications. In addition, it presupposes the availability of a graph coloring, which may not always be readily available."

Price of atomicity and determinism

The guarantees provided by the bulk-synchronous and lock-synchronized methods come at the expense of performance. The bulk-synchronous method can require up to twice as much memory to store modifications to the data-graph during a round. In addition, the bulk-synchronous method's lack of update atomicity can impact the convergence speed of some algorithms. For example, Figure 1-2 shows that the chromatically scheduled PageRank converges approximately $2\times$ faster than when implemented using the bulk-synchronous method. The lock-synchronized method provides atomicity, but requires the use of a locking protocol which introduces a significant synchronization overhead. A comparison of the *Not Atomic* and *Locks* columns of Figure 1-3 reveals that the use of locks can have a significant impact on performance. For example, turning on GraphLab's locking protocol increased the runtime of the PageRank benchmark from 3.7 seconds to 5.2 seconds.

The performance cost of obtaining determinism using the bulk-synchronous method and atomicity using lock-synchronized updates suggests that the overhead of performing chromatic scheduling should be even higher.

This thesis shows, however, that chromatic scheduling often performs better than the bulk-synchronous and lock-synchronized methods, even though these methods provide weaker guarantees. A comparison of the *Locks* and PRISM columns of Figure 1-3 shows that on a set of four benchmarks, chromatic scheduling improves the

Benchmark	*Not Atomic*	*Locks*	PRISM *(this thesis)*
1. Loopy belief propagation	2.6	2.8	1.5
2. PageRank	3.7	5.2	3.7
3. Alternating least squares	83.8	138.6	73.2
4. Alternating least squares with sparsity	108.1	142.8	93.3

Figure 1-3: Comparison of the relative performance of using locks and chromatic scheduling to guarantee update atomicity for four dynamic data-graph computations. All measurements are in seconds. *Not Atomic* is GraphLab with its locking protocol turned off so that it does not guarantee atomicity. *Locks* is GraphLab using its locking protocol to guarantee atomicity. PRISM is GraphLab using a new parallel engine and scheduling algorithm to support dynamic chromatic scheduling. The runtime for PRISM includes the time required to color the graph at runtime using a serial greedy coloring algorithm. Benchmarks were run on an Intel Xeon X5650 with a total of 12 2.67-GHz processing cores (hyperthreading disabled), 49 GB of DRAM, two 12-MB L3-caches each shared between 6 cores, and private L2- and L1-caches with 128 KB and 32 KB, respectively.

performance of GraphLab by a factor of 1.5 to 2. When the conflict graph can be colored using few colors, the cost of providing atomicity and determinism using chromatic scheduling is small: the time required to generate a coloring of the computation's conflict graph.

Deterministic parallel coloring

The semantic and performance advantages of PRISM motivate the development of efficient deterministic parallel coloring algorithms. Although vertex coloring is NP-complete [17], linear-time "greedy" heuristics exist which achieve at most $\Delta+1$ colors, where Δ is the maximum degree of nay vertex in the conflict graph. In practice, these linear-time heuristics tend to produce even fewer colors.

The simplicity of the greedy coloring algorithm, however, makes it challenging to develop a parallel algorithm that can achieve speedup relative to optimized serial code. For example, speculative coloring algorithms [8], commonly used in practice, are not suitable for PRISM. They produce a nondeterministic coloring, which eliminates one of the semantic advantages of chromatic scheduling. In addition, speculative coloring algorithms do not support the use of vertex-ordering heuristics which are commonly used in conjunction with greedy coloring algorithms to reduce the size of the coloring in practice.

We shall show, however, that the Jones-Plassmann coloring algorithm [37] admits

a work-efficient multicore implementation that achieves good speedup in practice. Although the algorithm is randomized, it is deterministic when the randomness is produced using a pseudorandom number generator. Pseudorandomness suffices to reap many of the benefits of deterministic programs—debuggability and reproducibility. Additionally, we shall show that the Jones-Plassmann algorithm is amenable to vertex-ordering heuristics that produce better colorings without sacrificing significant parallelism.

Summary of contributions

This thesis includes the following contributions which were developed jointly with William Hasenplaugh, Charles E. Leiserson, and Tao B. Schardl [30], all of the MIT Computer Science and Artificial Intelligence Laboratory.

- Design and analysis of PRISM, a work-efficient parallel algorithm for executing dynamic data-graph computations using chromatic scheduling.
- Implementation and performance evaluation of chromatic scheduling in the shared and external memory setting by modifying two existing data-graph computation libraries: GraphLab and GraphChi.
- Analysis of the Jones-Plassmann parallel greedy vertex-coloring algorithm for arbitrary degree graphs. We prove that the algorithm colors a graph with maximum degree Δ in $O(\Delta + \log \Delta \log V / \log \log V)$ rounds in the message passing model.
- A new work-efficient implementation of the Jones-Plassmann greedy coloring algorithm for the multi-core setting with $O(\log \Delta)$ contention and $O(\Delta \log \Delta + \log^2 \log V / \log \log V)$ span. A performance comparison with an optimized serial greedy coloring algorithm shows that the parallel algorithm is practical: it achieves modest speedup over the optimized serial code even when run on a small number of cores.
- A new vertex-ordering heuristic called ***log-degree ordering*** which can, in practice, decrease the number of colors used by the Jones-Plassmann coloring algorithm in exchange for a modest (logarithmic) increase in span. We prove

that the Jones-Plassmann algorithm runs in $O(\Delta + \log^2 \log V / \log\log V)$ rounds when using the log-degree ordering heuristic.

Organization of thesis

This thesis is broken into two parts.

Part I explores how chromatic scheduling can be used to efficiently parallelize data-graph computations. Chapter 2 provides a precise description of data-graph computations, demonstrating the importance of update atomicity for the performance of certain data-graph computations. Chapter 3 describes a simple static chromatic scheduling algorithm and analyzes its parallel performance using work/span analysis. Chapter 4 describes and analyzes PRISM, a chromatic scheduling algorithm for dynamic data-graph computations. Chapter 5 evaluates the performance of PRISM when implemented within two data-graph computation frameworks: GraphLab and GraphChi.

Part II describes an efficient parallel coloring algorithm that can be used by PRISM to perform chromatic scheduling. Chapter 6 introduces graph coloring and provides context for our contributions. Chapter 7 introduces the Jones-Plassmann parallel greedy coloring algorithm, and analyzes it's performance for arbitrary degree graphs. Chapter 9 describes a work-efficient implementation of the Jones-Plassmann algorithm for a modern multicore machine which achieves $3--5\times$ speedup over optimized serial code. Chapter 8 introduces the log-degree vertex-ordering heuristic and shows that in practice it can significantly reduce the number of colors while only decreasing the parallelism of Jones-Plassmann by a logarithmic factor.

The thesis ends with Chapter 10, which provides some concluding remarks.

Part I

Chromatic Scheduling

Chapter 2

Data-graph Computations

This chapter provides a formal introduction to data-graph computations and demonstrates the performance benefits of both dynamic scheduling and update atomicity. PageRank provides an illustrative example as an application whose performance benefits significantly from both dynamic scheduling and atomic updates.

Defining data-graph computations

Data-graph computations are a convenient way of specifying many parallel algorithms and applications. A ***data graph*** is an undirected graph $G = (V, E)$ with data associated with vertices and edges. The data-graph computation applies a user-defined ***update function*** f to a vertex $v \in V$ to ***update*** values in v's ***neighborhood*** — incident edges and adjacent vertices. The updates are typically partitioned into a series of sequentially executed ***rounds*** such that each vertex v is updated at most once per round. We characterize a data-graph computation as ***static*** if every vertex in V is updated in every round, and as ***dynamic*** otherwise. The computation typically continues either for a fixed number of rounds or, more usually, until a convergence criterion is met.

The updates to two distinct vertices $u, v \in V$ are said to ***conflict*** if executing $f(u)$ and $f(v)$ in parallel results in a ***determinacy race*** [16]—$f(u)$ and $f(v)$ access a common memory location, and at least one of them writes to that location. The ***conflict graph*** for a data-graph computation is the graph $G_c = (V, C_f)$ which con-

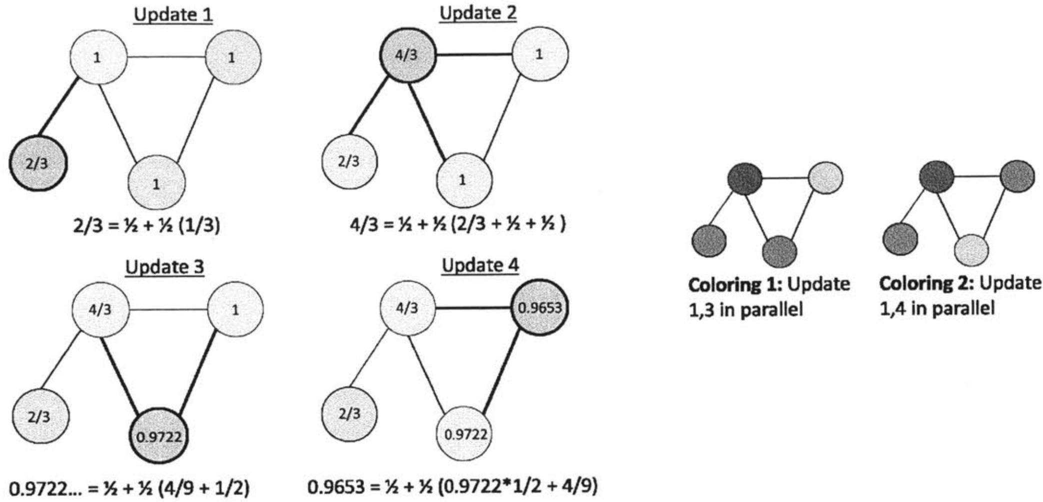


Figure 2-1: Illustration of one computation round of PageRank with damping factor $d = 1/2$ on a 4 node undirected graph. The update function computes a new PageRank for v as a function of its neighbors. $Pr(v) = (1-d) + d\sum_i N_i/D_i$ where N_i and D_i are the PageRank and degree of the ith neighbor of v. The graph colorings 1 and 2 illustrate the two possible parallelizations of this data-graph computation.

tains an edge $(u, v) \in C_f$ if the updates $f(u)$ and $f(v)$ conflict. Within a round, many non-conflicting updates can be safely applied in parallel.

Formulating PageRank as a data-graph computation

The PageRank algorithm provides an illustrative example of a data-graph computation which benefits from both dynamic scheduling and update atomicity. The PageRank algorithm can be formulated as a data-graph computation in which there is a vertex for each webpage, and an edge (u, v) when site u links to site v. The vertex data, in this case, contains a single number representing the associated webpage's PageRank which is iteratively updated until convergence. The data-graph computation's update function is applied to each vertex modifying its PageRank according to the formula $PR(v) = (1-d) + d\sum_i N_i/D_i$ where $d \in (0, 1)$ is a damping factor, N_i is the PageRank of v's ith neighbor, and D_i is the ith neighbor's degree.

The conflict graph for PageRank can be determined by considering the sets of updates which read and write shared data. The update function at vertex v reads the PageRank of all neighboring nodes in order to update it's own PageRank. Figure 2-1 highlights the regions of the graph which are read and written by the PageRank

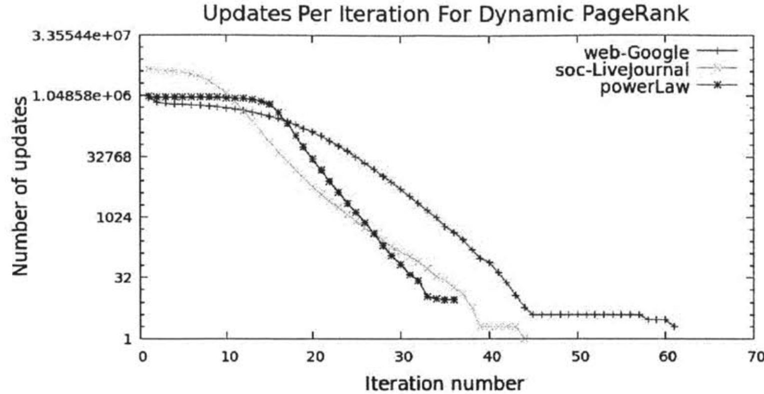


Figure 2-2: Number of updates performed per iteration for a dynamically scheduled PageRank application run on the non-synthetic web-Google and soc-LiveJournal graphs, and a synthetic power law graph of 1 million vertices and 10 million edges.

application during one computation round. Note that there is a read/write conflict between any two update functions operating on adjacent vertices. Therefore, the conflict graph for PageRank is simply the undirected version of the data-graph.

A coloring of the conflict graph reveals sets of updates which can be executed in parallel. For the example in Figure 2-1 there are two possible colorings which correspond to running updates 1 and 4 in parallel or updates 1 and 3 in parallel.

Static versus dynamic scheduling

PageRank can be implemented as either a static or dynamic data-graph computation. A static version of PageRank iteratively applies it's update function update function to all vertices in the graph for a fixed number of iterations or until some global termination condition is satisfied. This version of PageRank may perform unnecessary work, however, because it may update vertices whose neighbor's values have not changed—in which case the update would have no effect. A dynamic version of PageRank, however, can determine at runtime whether it is worthwhile to apply an update at a given vertex. For example, the update function for GraphLab's dynamic PageRank algorithm in Figure 2-3 schedules an update at a vertex only if it has a neighbor whose PageRank changed by some threshold.

Dynamic scheduling can significantly improve the performance of the PageRank application. Figure 2-2 plots the number of updates performed each iteration for the

```
void pagerank_update(gl_types::iscope &scope,
                     gl_types::icallback &scheduler) {
  // Get the data associated with the vertex
  vertex_data& vdata = scope.vertex_data();

  // Sum the incoming weights; start by adding the
  // contribution from a self-link.
  double sum = vdata.value*vdata.self_weight;
  const gl_types::edge_list& in_edges = scope.in_edge_ids();

  for (int i = 0; i < in_edges.size(); i++) {
    graphlab::edge_id_t eid = in_edges[i];
    // Get the neighobr vertex value
    const vertex_data& neighbor_vdata =
      scope.const_neighbor_vertex_data(scope.source(eid));
    double neighbor_value = neighbor_vdata.value;

    // Get the edge data for the neighbor
    edge_data& edata = scope.edge_data(eid);
    // Compute the contribution of the neighbor
    double contribution = edata.weight * neighbor_value;

    // Add the contribution to the sum
    sum += contribution;

    // Remember this value as last read from the neighbor
    edata.old_source_value = neighbor_value;
  }

  // compute the jumpweight
  sum = random_reset_prob/scope.num_vertices() +
    (1-random_reset_prob)*sum;
  vdata.value = sum;
  const gl_types::edge_list& out_edges = scope.out_edge_ids();

  // Schedule the neighbors as needed
  for (int i = 0; i < out_edges.size(); i++) {
    graphlab::edge_id_t eid = out_edges[i];
    edge_data& outedgedata = scope.edge_data(eid);

    // Compute edge-specific residual by comparing the new value of this
    // vertex to the previous value seen by the neighbor vertex.
    double residual =
      outedgedata.weight *
      std::fabs(outedgedata.old_source_value - vdata.value);

    // If the neighbor changed sufficiently add to scheduler.
    if(residual > termination_bound) {
      gl_types::update_task task(scope.target(eid), pagerank_update);
      scheduler.add_task(task, residual);
    }
  }
}
```

Figure 2-3: The update function for the PageRank application in GraphLab.

dynamic PageRank application on three different graphs. For each graph the number of updates performed each iteration decreases rapidly. By performing fewer updates, the dynamic version of PageRank runs faster than the static variant. On a power law graph of a million vertices and 10 million edges the static version of PageRank performs approximately 15 million updates whereas the dynamic version performs approximately 7 million updates. This translates into a reduction in total runtime. The static version runs in 3.39 seconds and the dynamic version runs in 2.24 seconds.

Effect of atomicity on convergence

Update atomicity also improves the performance of the PageRank application by improving its rate of convergence. Strategies for parallelizing data-graph computations that do not provide update atomicity may perform additional work due to slower convergence rates. The bulk-synchronous method, for example, is a method of parallelizing data-graph computations that does not provide update atomicity. The PageRank data-graph computation can be implemented correctly in the bulk-synchronous by maintaining two copies of each webpage's PageRank: a read copy that was computed the previous round, and a write copy which will store the updated value for the current round. This approach, however, sacrifices the atomicity of the PageRank update function. The effect on convergence rates causes the bulk-synchronous version of PageRank to be up to $2\times$ slower than the version with atomic updates.

This phenomenon applies more generally to iterative solvers for linear systems which are known to have better convergence properties when each iteration utilizes the most "up to date" values when updating a variable. The bulk-synchronous version of PageRank corresponds to the Jacobi method for iteratively solving a linear system, and the atomic update version corresponds to the Gauss-Seidel approach. Although the Jacobi algorithm has more parallelism, the Gauss-Seidel algorithm is known to converge up to 2x faster [38]. Indeed, Figure 2-4 demonstrates that the Jacobi version of both the static and dynamic versions of PageRank performs more than twice as many updates as the Gauss-Seidel variant.

Version	T_1 *(s)*	T_{12} *(s)*	*Total updates*
1. Static Jacobi	42.02	9.71	33,500,033
2. Static Gauss-Seidel	16.26	3.39	15,000,015
3. Dynamic Jacobi	37.26	6.32	18,691,620
4. Dynamic Gauss-Seidel	12.54	2.24	7,347,401

Figure 2-4: Performance comparison of the Jacobi and Gauss-Seidel (chromatic scheduling) algorithms for computing PageRank using both static and dynamic scheduling. These data result from running the described algorithm on a "power-law" graph with 1 million vertices and 10 million edges. All tests were run on an Intel Xeon 2.67 GHz 12-core machine with 48 GB of memory.

Dynamic data-graph computations with atomic updates

The PageRank example motivates the development of techniques to parallelize dynamic data-graph computations that guarantee update atomicity. Figure 2-4 illustrates four variants of PageRank with and without dynamic scheduling and update atomicity. The Gauss-Seidel versions (rows 2 and 4) of PageRank which requires update atomicity is approximately 3x faster than the Jacobi versions (rows 1 and 3) of PageRank when using either static or dynamic scheduling. Dynamic scheduling provides another 2x reduction in total updates. A comparison of rows 2 and 4 of Figure 2-4 shows that the dynamically scheduled Gauss-Seidel version of PageRanks runs approximately 30% faster than the static variant. The remainder of Part I of this thesis develops a scheduling algorithm which uses chromatic scheduling to efficiently parallelize dynamic data-graph computations while guaranteeing update atomicity.

Chapter 3

Analysis of Parallel Data-Graph Computations

In this chapter, we describe how to analyze the parallel performance of data-graph computations using work/span analysis. In particular, we analyze the performance of CHROMATIC, a chromatic scheduling algorithm for static data-graph computations. We will see that a parallel data-graph computation using CHROMATIC on a graph $G = (V, E)$ with a coloring of size χ performs $\Theta(V+E)$ work in $O(\chi(\log(V/\chi)+\log\Delta))$ span when the update function $f(v)$ performs $O(\deg(v))$ work in $O(\log\deg(v))$ span. This implies that CHROMATIC achieves near linear speedup when the number of processors $P \gg \Omega((V + E)/(\chi(\log(V/\chi) + \log\Delta)))$.

Work/span analysis

Work/span analysis [13, Ch. 27] is a technique for analyzing the theoretical performance of parallel algorithms. The performance of a parallel algorithm depends on the total number of operations it performs and on the length of the longest path in the program's "computation dag". The ***work*** T_1 of a parallel program is its total running time on a single processor. The ***span*** T_∞ is its running time on an infinite number of processors, assuming an ideal scheduler with no overhead.

The work and span of an algorithm can be used to accurately predict its parallel speedup when a "greedy scheduler" is used to assign work to processors. A ***greedy***

scheduler is a scheduler which will assign as much work as possible during each timestep. Using a greedy scheduler, a program with work T_1 and span T_∞ can always be executed in time T_P on P processors, where $\max\{T_1/P, T_\infty\} \leq T_P \leq T_1/P + T_\infty$. The ***parallelism*** T_1/T_∞ is the greatest speedup T_1/T_P possible for any number P of processors. When the parallelism of an algorithm is much larger than P, then the greedy scheduler bounds on T_P guarantee near linear speedup.

A static chromatic scheduling algorithm

Work/span analysis can be applied to theoretically analyze the parallel performance of data-graph computations performed using CHROMATIC— a chromatic scheduling algorithm for static data-graph computations.

The CHROMATIC algorithm executes a round of a data-graph computation via a sequence of subrounds that each apply updates to a monochromatic set of vertices. The CHROMATIC algorithm begins by dividing the set of vertices into χ sets $A_1, \ldots, A_\chi$ where χ is the number of colors used to color the graph. This step can be performed in $O(\Theta(V))$ work and $O(\Theta(\log V))$ span with a parallel integer-sort of the vertex set using each vertex's color as its key [21]. To execute a round of the data-graph computation CHROMATIC updates each set $A_0, \ldots, A_\chi$ in series. The updates within a given set A_i may be executed in parallel since the updates to two vertices of the same color cannot be adjacent in the conflict graph. Figure 3-1 provides pseudo-code for CHROMATIC that executes computation rounds iteratively until a termination condition is satisfied.

In general, the work and span of CHROMATIC during a particular computation round depends on the work and span of the update function. It is common, however, for update functions for data-graph computations to have similar parallel structure. In particular, we refer to an update function $f(v)$ as a ***standard update function*** if it can be broken up into $\deg(v)$ updates $f_1(v), \ldots, f_{\deg(v)}(v)$ which each perform $O(1)$ work and may be evaluated in parallel. A standard update function can be parallelized in $O(\deg(v))$ work and $O(\log \deg(v))$ span by performing a parallel loop over the index set $i = 1, \ldots, \deg(v)$, evaluating $f_i(v)$ in the body of the loop.

```
CHROMATIC(V, f, χ)
1  sort the elements of V by color into A_0, A_1, ..., A_{χ-1}.
2  repeat
3      done = TRUE
4      for k = 0 to χ - 1
5          parallel for i = 0 to |A_k| - 1 reducing done
6              done ∧= f(A_k[i])
7  until done
```

Figure 3-1: Parallel pseudocode for CHROMATIC, a chromatic scheduler for static data-graph computations. CHROMATIC takes as input the vertex set V, the update function f, and the number of colors χ used to color V. The **reducing** clause in the **parallel for** on line 5 indicates that all the operations on the variable *done* should be combined with a parallel tree. The update function $f(v)$ returns TRUE if a termination condition has been satisfied at v. CHROMATIC terminates once the termination condition has been satisfied by all $v \in V$.

It turns out, however, that the exact distribution of vertices amongst the χ color sets has a limited impact on the parallel performance of CHROMATIC. The following theorem proves bounds for the work and span of CHROMATIC for standard update functions. It shows that CHROMATIC has parallelism that is inversely proportional to the number of colors χ used to color the graph.

Theorem 1 CHROMATIC *executes a round of a data-graph computation that applies a standard update function to every vertex in V in $\Theta(V+E)$ work and $O(\chi(\log(V/\chi)+\log\Delta))$ span.*

PROOF. The total work in one round of CHROMATIC is $T_1 = \sum_{i=1,\ldots,\chi}\sum_{v\in A_i}\deg(v)$ when f is a standard update function. Since the color sets A_i are nonintersecting and their union is V, the total work is equal to the sum of all vertex degrees in the graph. The handshaking lemma implies, therefore, that the work is $T_1 = O(V+E)$.

The span of one round of CHROMATIC is $T_\infty = \sum_{i=1,\ldots,\chi}\log A_i + max_{v\in A_i}\{\log\deg(v)\}$. The $\log A_i$ term in the span is due to the parallel loop on line 5 over the index set $1,\ldots,|A_i|$. The second term $max_{v\in A_i}\{\log\deg(v)\}$ is due to the evaluation of the update function in the body of the parallel loop.

The expression for the span can be simplified to remove its dependence on the distribution of vertices amongst the color sets $A_1,\ldots,A_\chi$. The contribution of the

first span term $\sum_{i=1,\ldots,\chi} \log A_i$ can be simplified by noting that the sum of a pair of logarithms $\log(a - \epsilon) + \log(a + \epsilon) = \log(a^2 - \epsilon^2)$ is maximized when $\epsilon = 0$. The maximum value of $\sum_{i=1,\ldots,\chi} \log A_i$, therefore, occurs when each set A_i contains an equal number of vertices. This allows us to bound the contribution of the first span term by $\chi \log(V/\chi)$. The second term can be simplified by using the maximum degree of the graph Δ to bound the term $max_{v \in A_i}\{\log \deg(v)\}$ by $\log \Delta$. An upper bound on the span of a round of CHROMATIC, therefore, is $T_\infty = O(\chi(\log(V/\chi) + \log \Delta))$.
□

The analysis of CHROMATIC for general update functions closely matches the proof of Theorem 1, but requires more cumbersome notation. Let $T_1(f(v))$ and $T_\infty(f(v))$ be the work and span of the update $f(v)$. Then one can define the work and span of f applied to a set A_i of vertices in parallel to be $T_1(A_i) = \sum_{v \in A_i} T_1(f(v))$ and $T_\infty(A_i) = \max_{v \in A_i} T_\infty(f(v))$. The following corollary uses this notation to state the general bound on the work and span of a round of CHROMATIC.

Corollary 2 *A round of* CHROMATIC *applying a general update function f to every vertex $v \in V$ in work $T_1 = \Theta(T_1(f(V)))$ and span $T_\infty = O(\chi \log(V/\chi) + \chi T_\infty(f(V)))$.*

Chapter 4

Dynamic Chromatic Scheduling

In this chapter, we describe PRISM, a work-efficient algorithm for the chromatic scheduling of dynamic data-graph computations. Using the techniques developed and applied in Chapter 3 when analyzing CHROMATIC, we analyze the parallel performance of PRISM. We will first derive bounds on the work and span of PRISM for standard update functions for which $f(v)$ performs $O(\deg(v))$ work in $O(\log \deg(v))$ span. In this case, for a graph $G = (V, E)$ with max vertex degree Δ, PRISM updates a set Q of vertices with E_Q incident edges in $\Theta(Q + E_Q)$ work and $O(\chi(\log(Q/\chi) + \log \Delta))$ span. This analysis can be generalized to provide bounds on the work and span of PRISM for arbitrary update functions. We will see that PRISM performs as well asymptotically as CHROMATIC in the case in which $Q = V$.

Complexity of dynamic scheduling

The primary challenge in implementing a provably good dynamic scheduling algorithm is guaranteeing that the work to update a set of vertices $Q \subset V$ depends only on the size of Q. This guarantee is important because, in practice, Q may be much smaller than V. A given round of a data-graph computation may, for example, update only a single perturbed vertex. It is straightforward to devise dynamic chromatic scheduling algorithms that fail to provide this guarantee. For example, the static chromatic scheduling algorithm CHROMATIC can be modified to check a *scheduled bit* associated with each vertex before executing an update. The scheduled bit

for a vertex is set to TRUE when an update should be applied to it in the next round. This modified version of CHROMATIC is not work-efficient for two reasons. The first reason is that $\Omega(V)$ work is required to check whether an update is scheduled at each vertex. The second reason is due to possible write-contention on the "scheduled bit" which can increase the work of a parallel algorithm on a multicore machine by requiring a processor to wait additional timesteps to obtain exclusive write access to a memory location. This chapter describes PRISM— a provably good dynamic chromatic scheduling algorithm.

Analyzing memory contention

Work/span analysis does not usually consider the effect of memory contention. In fact, the effect of memory contention can be safely ignored as long as the number of parallel writes which contend for any given memory location is bounded by a constant — since a constant delay in memory operations will not change the asymptotic work and span of an algorithm. Unbounded contention, however, can have a large impact on a parallel algorithm's performance when it is implemented on a multicore machine. Unbounded contention can arise in dynamic data-graph computations when scheduling an update at a vertex in the next computation round. For example, if a single byte is used to indicate whether a vertex has been scheduled, then up to Δ contention may occur if multiple neighbors of that vertex attempt to schedule its update at the same time.

All but one of the parallel phases of PRISM have bounded contention. The one exception is the procedure used to deduplicate sets of scheduled vertices. For this phase, we shall analyze contention in the CRQW model [21, 22], where concurrent reads to a variable can be accomplished in constant time, but concurrent writes to the same memory location are serviced in FIFO order by the memory system. In the CRQW model, the deduplication of scheduled vertices can be accomplished by invoking the randomized integer-sorting algorithm from [21] which sorts n nonnegative integers less than n using $\Theta(n)$ work and $\Theta(\lg n)$ span in expectation.

```
PRISM(V, f, Q, χ)
 1  parallel for v ∈ Q
 2      INSERT(A_{v.color}[wid], v)
 3  repeat
 4      done = TRUE
 5      for k = 0 to χ − 1
 6          X = COLLECT-SET(A_k)
 7          done ∧= (X == ∅)
 8          parallel for i = 0 to |X| − 1
 9              if i == |X| − 1 or X[i] ≠ X[i + 1] // V_k = V_k ∪ {X[i]}
10                  S = f(X[i])
11                  parallel for u ∈ S
12                      INSERT(A_{u.color}[wid], u)
13  until done
```

Figure 4-1: Pseudocode for PRISM, a chromatic scheduler for dynamic data-graph computations. PRISM takes as input the vertex set V, the update function f, an initial set Q of vertices to process, and the number of colors χ used to color V. Each update $f(v)$ returns a subset of the vertices neighboring v to process in a subsequent color step. The variable *wid* stores the ID of the current worker, which is used to access worker-local storage.

```
COLLECT-SET(A)
1  X = GATHER(A)
2  fail = TRUE
3  while fail
4      Randomly choose h : V → {0, 1, ..., |X|^3 − 1}
5      RADIX-SORT-BY-HASH(X, h)
6      fail = FALSE
7      parallel for i = 0 to |X − 1| reducing fail
8          fail ∨= (h(X[i] == h(X[i + 1]) ∧ (X[i] ≠ X[i + 1])
9  return X
```

Figure 4-2: Pseudocode for COLLECT-SET, which collects the activations in the P worker-local arrays $A_k[0], \ldots, A_k[P-1]$ to produce a version of the activation set A_k where all duplicate vertices are stored adjacent to each other. GATHER is assumed to empty the vectors storing A as a side effect.

Design and analysis of PRISM

Figure 4-1 gives the pseudocode of PRISM. Conceptually, PRISM operates much like CHROMATIC, except that, rather than process vertices in a static array in color step k, PRISM maintains an ***activation set*** A_k for each color k—a set of activations for vertices of color k that occurred since color step k in the previous round. After PRISM loads the initial set of active vertices Q into the activation sets (on lines 1–2),

PRISM executes the rounds of the data-graph computation by looping over the colors $0, \ldots, \chi - 1$ (on lines 5–12) until it finds no vertices to update in a round. Within a single color step k, PRISM executes all updates $f(v)$ on distinct activated vertices v in A_k (on lines 8–12). PRISM stores an activation set A_k using worker-local storage. For a P-processor execution, PRISM allocates P vectors $A_k[0], \ldots, A_k[P-1]$, one for each worker. Each worker-local set $A_k[P_i]$ is implemented as a vector that uses incremental resizing to support INSERT operations in $\Theta(1)$ worst-case time. Each update returns a set S of activations (line 10), and then PRISM distributes the set of activations to its activation sets in parallel (on lines 11–12). The use of activation sets avoids the need to perform $\Omega(V)$ checks each round to determine which vertices should be updated.

By using worker-local storage, no contention occurs when inserting a vertex into an activation set (on line 12). However, it is possible for a vertex to appear in the activation sets of multiple workers because a vertex may be activated by any of its neighbors. A given worker can ensure that no duplicate vertices appear in its local activation set by performing lookups in a worker local hash table that contains the set of previously inserted vertex ids. Preventing a vertex from appearing in multiple activation sets could be accomplished by having all workers utilize a shared data-structure supporting atomic lookups of a vertex's activated state. For example, an array of V bytes could be used to avoid duplicates by having each worker perform a compare and swap on the ith byte in the array before adding the vertex with id i to its activated set. This approach, however, requires a data synchronization operation for every activation. Furthermore, when the number of processors is large there could be significant contention on the byte array as multiple processors attempt to write to the same cache line.

The alternate strategy used by PRISM is to allow a vertex to appear in multiple activation lists, but prevent it from being updated more than once. If all of the worker's activation lists are merged and sorted based on the vertex id, then a duplicate vertex can be identified by checking whether its predecessor in the activation list shares its vertex id. To process the distinct vertices in activation set A_k, PRISM calls COLLECT-SET, depicted in Figure 4-2, to collect the activations in the worker-local

activation sets for A_k and sort them into a single array X in which, for each distinct vertex v in A_k, all copies of v are adjacent in X. The appendix contains a proof that, with probability $1 - 2/|A_k|$, COLLECT-SET operates in $\Theta(A_k)$ expected work and $O(\lg n + \lg P)$ expected span in the CRQW contention model.

To bound the work, span, and contention of PRISM, we first analyze the work, span, and contention of COLLECT-SET.

Lemma 3 *With probability $1 - 2/n$, the function* COLLECT-SET *collects the set of n activated vertices A_k of color k from all P worker-local arrays and groups duplicates together in expected $\Theta(n)$ work and expected $O(\lg n + \lg P)$ span in the CRQW contention model.*

PROOF. Let us first analyze one iteration of COLLECT-SET. First, GATHER on line 1 collects the contents of the P worker-local copies of A_k into a single array X in $\Theta(n)$ work, $\Theta(\lg n + \lg P)$ span, and no contention. To do this, GATHER first performs a parallel prefix-sum over the sizes of the worker-local arrays $A_k[0], \ldots, A_k[P-1]$ to determine a location in X for each worker-local array, followed by P parallel copies to copy the worker-local arrays into X in parallel. Next, line 4 chooses a random hash function $h : V \to [|X|^3]$ from the vertices V to $3 \lg n$-bit numbers, and line 5 calls RADIX-SORT-BY-HASH to sort X by h. RADIX-SORT-BY-HASH may be implemented with 3 passes of the stable linear-work sorting algorithm from [21], in which case RADIX-SORT-BY-HASH executes in expected $\Theta(n)$ work and expected $\Theta(\lg n)$ span with probability $1 - 1/n$ in the CRQW contention model. Finally, lines 7–8 checks the sorted X to verify that all adjacent elements in X are either duplicates or have distinct hashes, requiring $\Theta(n)$ work, $\Theta(\lg n)$ span, and bounded contention to compute.

We now verify that, with probability $1 - 1/n$, a constant number of iterations suffices. The probability that one iteration of lines 3–8 fails is precisely the probability that any two distinct vertices u and v in X hash to the same value. If h is pairwise independent, then for any two distinct vertices u and v in X, we have $\Pr\{h(u) = h(v)\} = 1/n^3$. By a union bound, the probability of a collision between any

two vertices in X is at most $\sum_{u,v \in X} \Pr\{h(u) = h(v)\} \leq 1/n$. COLLECT-SET thus runs in the stated work, span, and contention bounds with probability $(1-1/n)^2 \geq 1-2/n$. □

We now bound the expected work and expected span to compute a single color step, i.e. to perform one iteration of lines 5–12 of Figure 4-1.

Theorem 4 *Consider the execution of color step k of* PRISM *with P workers. Let $E_k = \sum_{v \in V_k} \deg(v)$. With probability $1 - 2/|A_k|$, color step k of* PRISM *executes with expected work $T_1 = \Theta(A_k + E_k)$ and expected span $T_\infty = O(\lg A_k + \lg P + \lg \Delta)$ in the CRQW contention, where Δ is the degree of G.*

PROOF. We analyze the code in Figure 4-1. By Lemma 3, with probability $1 - 2/|A_k|$, the call to COLLECT-SET on line 6 performs expected $\Theta(A_k)$ work, expected $\Theta(\lg A_k + \lg P)$ span, and bounded contention. The result of line 6 is the array X, a version of A_k where duplicate vertices are stored next to each other. The loop over X on lines 8–12 touches every element of X a constant number of times in $\Theta(A_k)$ work, $\Theta(\lg A_k)$ span, and no contention. Next, line 10 calls f on each unique element in X once, incurring E_k work and $\lg \Delta$ span.

Each update $f(X[i])$ activates some set S of vertices for future color steps, which lines 11–12 insert into the worker-local arrays in parallel. Because each worker-local array is implemented as an incrementally resizing vector, each insertion takes $\Theta(1)$ worst-case time. Moreover, because the number of vertices activated by some $f(X[i])$ is bounded by the work of $f(X[i])$, lines 11–12 execute in $\Theta(\deg(X[i]))$ work and $\Theta(\lg \deg(X[i])) = O(\lg \Delta)$ span per update $f(X[i])$. Summing the work and span over all vertices in V_k completes the proof. □

From Theorem 4, we conclude that, with high probability, PRISM achieves at worst the same theoretical bounds as a chromatic scheduler for static data-graph computations. Consequently, PRISM is work-efficient, incurring no additional overhead (with high probability) over a chromatic scheduler for static data-graph computations. The following corollary formalizes this observation.

Corollary 5 *Consider a data-graph computation to apply the update function f to a graph $G = (V, E)$ with degree Δ. Let P be the number of workers executing the data-graph computation. Suppose that, in one round of the data-graph computation, the set $V' \subseteq V$ of vertices are updated, where $|V'| > P$, and let A denote the activation set of V', the union of A_k over all color steps k in the round. Then, with probability $1-2/|A|$, the expected work and span of* PRISM *match the work and span of* CHROMATIC.

PROOF. For didactic simplicity, we assume that an update $f(v)$ executes in work $\Theta(\deg(v))$ and span $\Theta(\lg \deg(v))$. A chromatic scheduler for static data-graph computations can execute a round of this data graph computation in $\Theta(V + E)$ work and $O(\chi \lg(V/\chi) + \chi \lg \Delta)$ span with no contention. Let E' be the set of outgoing edges from all vertices in V'. By Theorem 4, with probability $1 - 2/|A|$, PRISM executes all updates vertices V' in a round in expected work $\Theta(A + V' + E')$, expected span $O(\chi \lg(A/\chi) + \chi \lg P + \chi \lg \Delta)$, and bounded contention. If $V' = V$, then because the f must perform work to activate a vertex $v \in V'$, we have $|A| \leq |E'|$. Finally, because $|A| \leq |E| \leq |V|^2$, we have $\chi \lg(A/\chi) = O(\chi \lg(V/\chi))$. Hence, the expected work and expected span bounds for PRISM are bounded above by the work and span of CHROMATIC. □

Chapter 5

Performance Evaluation of Prism

In this chapter, we evaluate the performance of PRISM by implementing it in GraphLab and GraphChi. We will see that PRISM improves the performance of GraphLab by a factor between 1.5 and 2 on a set of four benchmarks and that PRISM improves the performance of GraphChi's execute update phase by a factor between 2 and 3 when it guarantees determinism. The impact of coloring size on the parallelism and runtime of chromatically scheduled data-graph computations is also analyzed empirically.

Chromatic scheduling in GraphLab

We implemented PRISM in GraphLab v1.0 to add support for chromatic scheduling of dynamic data graph computations. The scheduler interface was modified to expose the method *getActivationSet()* which returns update tasks on a mono-chromatic set of vertices. Each activation set is implemented using worker local storage as a dynamic array. We evaluated PRISM in GraphLab by comparing its performance on a set of four benchmarks to the "edge consistency" locking protocol that provides the same data consistency guarantees as chromatic scheduling by acquiring locks on updated vertices and their incident edges.

The GraphLab applications for loopy belief propagation, PageRank, and alternating least squares were used to compare the relative overheads of chromatic scheduling and lock-based synchronization. Loopy belief propagation was run on a subset of the Cora dataset of computer science paper citations [50, 58] which has an MRF of

Benchmark	*Seconds*			*# Updates* (10^6)		
	Sweep-VC	*Sweep-EC*	PRISM-*EC*	*Sweep-VC*	*Sweep-EC*	PRISM-*EC*
Loopy belief propagation	2.6	2.8	1.5 (0.1)	2.19	2.22	2.20
PageRank	3.7	5.2	3.7 (1.3)	10.21	10.18	9.75
Alternating least squares	83.8	138.6	73.2 (1.0)	4.17	3.99	3.85
Alternating least squares with sparsity	108.1	142.8	93.3 (1.0)	4.03	3.88	3.85

Figure 5-1: Performance comparison of PRISM (PRISM-EC), GraphLab's edge consistency locking protocol (Sweep-EC), and GraphLab's vertex consistency locking protocol (Sweep-VC). All benchmarks were run using 12 cores. GraphLab was run using its "sweep" scheduler which provides scheduling semantics equivalent to PRISM. The time to color the graph serially is included in the PRISM-EC runtime, and also provided separately in parenthesis.

160K vertices, and 480K edges. PageRank was run on the web-Google dataset of 87K websites and 5M links [42]. Two versions of alternating least squares, one of which enforces a sparsity constraint on computed factors, was run on the NPIC500 dataset consisting of 88K noun phrases, 99K contexts, and 20M occurrences [51].

All benchmarks were run on an Intel Xeon X5650 with a total of 12 2.67-GHz processing cores (hyperthreading disabled), 49GB of DRAM, two 12-MB L3-caches each shared between 6 cores, and private L2- and L1-caches with 128 KB and 32 KB, respectively.

The benchmark results in Figure 5-1 show that PRISM is approximately 1.5-2 times faster than GraphLab's edge consistency locking protocol. Furthermore, PRISM is often faster than GraphLab's much weaker vertex consistency locking protocol, which only serializes updates occurring at the same vertex. The latter observation can be attributed to our use of the Intel Cilk Plus runtime system, and a lower per task scheduling overhead. This suggests that PRISM provides the properties of data consistency and determinism at low cost. The performance comparison becomes even more favorable when we consider that PRISM's runtime includes the time required to color the graph using a serial greedy coloring algorithm. The time required to color the graph can be reduced through the use of parallel graph coloring algorithms, and may not be necessary at all if a coloring has been previously computed.

The scalability of PRISM was measured by comparing its speedup to GraphLab's dynamic scheduler with and without its edge consistency locking protocol. Figure 5-2 compares the speedup achieved on a PageRank benchmark for these three programs relative to the fastest serial program. The speedup curve for PRISM is noticeably

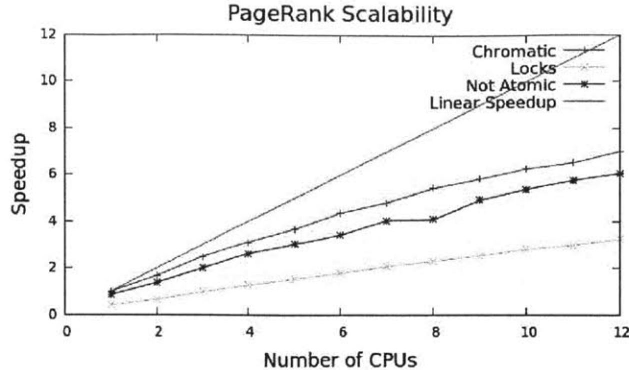


Figure 5-2: Speedup plot of Not Atomic, Locks, and Chromatic on 1-12 cores for the PageRank benchmark run on a random power law graph of 1 million vertices and 10 million edges. Speedup is measured relative to the fastest serial program. Not Atomic refers to GraphLab with its locking protocol turned off, Locks refers to GraphLab with its edge consistency locking protocol used to provide atomicity, and Chromatic refers to the version of GraphLab utilizing PRISM to provide atomicity using chromatic scheduling.

Dataset	*GraphChi (execute updates)*	PRISM *(execute updates)*	*GraphChi (total time)*	PRISM *(total time)*
cit-Patents	2.38	1.93	22.24	21.98
Mediawiki	33.13	11.86	239.72	209.87

Figure 5-3: Benchmark results for 4 iterations of PageRank in GraphChi. All measurements are in seconds. Performance comparison of two ways to guarantee determinism in GraphChi. Serialize Conflicts provides determinism by serializing all conflicting updates within a round. Chromatic provides determinism by using PRISM. The runtime for Chromatic includes the time required to color the graph at runtime.

steeper than the speedup curve for Locks demonstrating that PRISM not only outperforms Locks in terms of raw performance, but can also exhibit superior parallel scalability.

Chromatic scheduling in GraphChi

GraphChi [41] is an extension of GraphLab to the external memory setting which can provide both determinism and atomicity. It divides a graph into shards stored on disk, and applies updates to each shard in series. GraphChi can guarantee atomicity and determinism by serializing all updates which conflict with another update in the same shard. This approach often sacrifices a large amount of parallelism. For example, on a PageRank application 30% of all updates were serialized on the cit-Patents dataset and 70% were serialized on the Mediawiki dataset.

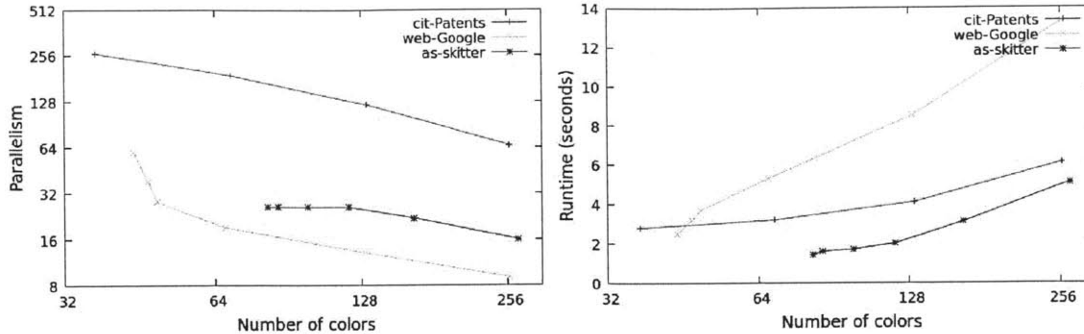


Figure 5-4: Effect of the number of colors used on parallelism and runtime of PageRank.

To perform chromatic scheduling in GraphChi, the sub-graph associated with each of its shards is colored at runtime. Once a shard is colored, the color assignment is stored on disk with the other vertex data. Figure 5-3 demonstrates that chromatic scheduling can improve the performance of GraphChi's execute update phase by a factor of 3 when using GraphChi's deterministic engine.

Parallelism as a function of the graph coloring

The parallelism present in a data-graph computation when using PRISM can depend on size of the graph coloring. To explore this effect, we used the Cilkview scalability analyzer [31] to analyze the parallelism of PRISM on the PageRank application when the size of the coloring is varied. To obtain larger graph colorings, we modified the greedy coloring algorithm to select the smallest available color greater than r, where r is chosen uniformly at random from the range $[0, R)$ for each vertex. Larger colorings were then obtained by coloring the graph for $R = 16, 32, \ldots, 256$. The results in Figure 5-4 indicate that the size of the graph coloring can have an impact on the scalability of data-graph computations when chromatic scheduling is used. When parallelism is scarce larger colorings can also translate into an increase in runtime.

Part II

Deterministic Parallel Coloring

Chapter 6

Graph Coloring

Chromatic scheduling utilizes a vertex coloring of the conflict graph in order to identify sets of nonconflicting updates that can be applied in parallel. In the next four chapters, we explore how such a vertex coloring can be generated in parallel on a multicore machine.

This chapter provides an brief overview of related work on vertex-coloring algorithms. We review the classic serial greedy coloring algorithm and discuss a common variant, the Welsh-Powell algorithm, which utilizes a vertex-ordering heuristic to reduce the number of colors used in practice. We will then review previous work on parallel greedy coloring algorithms including the Jones-Plassmann coloring algorithm which will be analyzed in greater detail in Chapter 7.

Greedy graph coloring

Optimally coloring the vertices of a graph is a known NP-complete problem [17]. Although polynomial-time approximation algorithms are known [61], linear-work greedy algorithms are often used in practice due to their practical efficiency — typically they perform only one or two passes over the data.

The ***lexicographically first*** [12], [28, Appendix A] greedy coloring of a graph $G = (V, E)$ can be computed by processing each vertex in an order defined by a permutation π. A processed vertex v is assigned a color greedily by picking the lowest numbered color that has not been assigned to any of v's neighbors. For a

graph with max vertex degree Δ, this algorithm runs in $\Theta(V + E)$ time and uses at most $\Delta + 1$ colors. Figure 6-1 gives pseudocode for this greedy algorithm based on an arbitrary permutation π of V.

Vertex ordering heuristics

Practitioners often use heuristics to pick an ordering of the vertices that may allow GREEDY to use fewer colors. The ***degree-ordering*** heuristic [3,23,37], for example, orders the vertices by decreasing degree so that larger degree vertices are colored first. The degree-ordering heuristic seems to allow GREEDY to use fewer colors in practice. The classic Welsh-Powell greedy coloring algorithm [60], in fact, can be viewed as an execution of GREEDY utilizing the vertex-ordering heuristic.

One disadvantage of the degree-ordering heuristic, however, is that it is vulnerable to adversarial input graphs which force GREEDY to produce a coloring with $\Delta + 1$ colors. For instance, although the "crown" graph on $|V|$ vertices is 2-colorable, GREEDY produces a $|V|/2$ coloring using degree ordering [36]. The ***random-ordering*** heuristic which utilizes a random permutation of the vertices can use fewer colorings in practice for certain adversarial graphs.

Parallel graph coloring

Luby explored parallel randomized greedy coloring [48] and inspired many papers [4,24,25,43] on the topic of parallel coloring, including methods for derandomizing. Several deterministic parallel coloring algorithms [5,39,40], based on the algebraic construction of Linial [44] have been shown to be theoretically fast. Unfortunately, none of these algorithms is work efficient, that is, uses as little asymptotic work as the greedy algorithm.

Given a random permutation π on the vertices of a graph G, Jones and Plassmann demonstrated that a parallel implementation of GREEDY runs deterministically in expected $O(\ln V / \ln \ln V)$ rounds in the message-passing model [11,20], assuming that the graph has bounded degree. Of course, the assumption of bounded degree also circumvents the issue of contention, which is a real issue when implementing

```
GREEDY(G = (V, E), π)
1  i = 0
2  while there are uncolored vertices
3       let v be the ith vertex in V by the ordering π
4       v.color = lowest numbered color not taken by any u ∈ Adj[v]
5       i = i + 1
```

Figure 6-1: The serial greedy graph-coloring algorithm in pseudocode based on the Welsh-Powell [60] coloring algorithm. GREEDY colors all vertices of a graph $G = (V, E)$ in the order dictated by the permutation π.

graph algorithms on modern multicore machines. We shall both analyze the Jones-Plassmann algorithm for large degree and demonstrate that it can be made to operate work efficiently and with small span using the CRQW contention model.

Chapter 7

Jones-Plassmann Coloring Algorithm

This chapter examines JP, the Jones-Plassmann parallel greedy coloring algorithm [37]. Given a random permutation π on the vertices of a graph $G = (V, E)$ of bounded degree, Jones and Plassmann demonstrated that JP runs deterministically and work-efficiently in expected $O(\ln V / \ln \ln V)$ rounds in the message passing model [11, 20]. We extend this analysis to arbitrary degree graphs showing that JP runs in expected $O(\Delta + \ln \Delta \ln V / \ln \ln V)$ rounds in the message passing model.

Induced priority dag

The Jones-Plassmann algorithm colors a graph according to a "priority dag" that is induced by an ordering π of the vertices. We say that a ***priority dag*** $G_\pi = (V, E_\pi)$ is ***induced*** by π on the undirected graph $G = (V, E)$ when G_π and G share the same vertices and there is a directed edge (u, v) in E_π for some $(u, v) \in E$ if $\pi(u) > \pi(v)$. We call $\pi(v)$ the ***priority*** of vertex v.

When selecting a color for a vertex $v \in V$, GREEDY only considers colored neighbors, which are precisely those that come before v in the permutation π, or equivalently $Pred[v]$. So, once every predecessor $u \in Pred[v]$ has been colored, v is also free to be colored. Jones and Plassmann [37] exploited this observation to produce an asynchronous parallel coloring algorithm, referred to here as JP, which produces the same deterministic output as GREEDY given the same inputs. JP operates by first finding the priority dag $G_\pi = (V, E_\pi)$ induced on a graph $G = (V, E)$ by permutation

π. Then, a message travels along the directed edge $(u, v) \in E_\pi$ only after vertex u has been successfully colored, notifying vertex v. Once v has received $|Pred[v]|$ messages, then v may be colored and send all successors of v similar messages.

JP($G = (V, E), \pi$)
1 **parallel for** all $v \in V$
2 $\quad$ $Pred[v] = \{u \in Adj[v] \mid \pi(u) > \pi(v)\}$
3 **parallel for** all $v \in V \mid Pred[v] == \emptyset$
4 $\quad$ COLOR(v)

COLOR(v)
1 $v.color$ = lowest available color
2 **parallel for** all $v_{succ} \in Succ[v]$
3 $\quad$ **if** v is last of v_{succ}'s predecessors
4 $\quad\quad$ COLOR(v_{succ})

Figure 7-1: JP implements the parallel greedy graph coloring algorithm of Jones and Plassmann [37] using the recursive method COLOR. JP produces the same identical coloring of a graph $G = (V, E)$ given an arbitrary permutation on the vertices π as GREEDY given the same inputs.

Jones and Plassmann showed that $O(\ln V / \ln \ln V)$ expected rounds in the message-passing model suffices to greedily color a graph $G = (V, E)$ of bounded degree $\Delta = \Theta(1)$ given a random vertex permutation π. We will show that, in fact, JP needs only $O(\Delta + \ln \Delta \ln V / \ln \ln V)$ expected rounds to color any general graph G with degree Δ given random permutation π.

Bounding the longest directed path in a random-induced priority dag

We show that the length of the longest directed path in the priority dag G_π induced on a graph G by a random permutation π is likely to be bounded and consequently that JP runs in the same number of rounds in the message-passing model. We bound the longest directed path in G_π by first demonstrating that any particular sufficiently long path in G is unlikely to be a directed path in G_π. We then use the union bound to show that no such path is likely to exist in G_π. However, first we will prove a useful lemma.

Lemma 6 *For $\alpha, \beta > 1$, define the function $g(\alpha, \beta)$ as*

$$g(\alpha, \beta) = e^2 \frac{\ln \alpha}{\ln \beta} \ln\left(e \frac{\beta \ln \alpha}{\alpha \ln \beta} \right) .$$

For all $\beta \geq e^2$, $\alpha \geq 2$ and $\beta \geq \alpha$, we have $g(\alpha, \beta) \geq 1$.

PROOF. We consider the cases when $\alpha \geq e^2$ and when $\alpha < e^2$ separately.

First, we consider the case when $\alpha > e^2$. The partial derivative of $g(\alpha, \beta)$ with respect to β, $\partial g(\alpha, \beta)/\partial \beta$, is

$$\begin{aligned} &= e^2 \frac{\ln \alpha}{\beta \ln^2 \beta} \ln\left(\frac{\alpha}{e^2} \frac{\ln \beta}{\ln \alpha} \right) \\ &\geq 0 , \end{aligned}$$

since $\alpha \ln \beta / e^2 \ln \alpha \geq 1$ when $\alpha \geq e^2$ and $\beta \geq \alpha$. Thus, $g(\alpha, \beta)$ is a nondecreasing function in its second argument when $\alpha \geq e^2$ and $\beta \geq \alpha$. Since we have

$$\begin{aligned} g(\alpha, \alpha) &= e^2 \frac{\ln \alpha}{\ln \alpha} \ln\left(e \frac{\alpha \ln \alpha}{\alpha \ln \alpha} \right) \\ &\geq 1 , \end{aligned}$$

it follows that $g(\alpha, \beta) \geq 1$ for $\alpha \geq e^2$ and $\beta \geq \alpha$.

Next, we consider the case where $e^2 > \alpha \geq 2$. We make use of the fact that $2\beta / e \ln \beta > \sqrt{\beta}$ for all $\beta > e^2$ to bound

$$\begin{aligned} g(\alpha, \beta) &= e^2 \frac{\ln \alpha}{\ln \beta} \ln\left(e \frac{\beta \ln \alpha}{\alpha \ln \beta} \right) \\ &\geq \frac{e^2 \ln 2}{\ln \beta} \ln\left(\frac{2}{e} \frac{\beta}{\ln \beta} \right) \\ &\geq \frac{e^2 \ln 2}{\ln \beta} \ln\left(\sqrt{\beta} \right) \\ &\geq \frac{e^2 \ln 2}{2} \frac{\ln \beta}{\ln \beta} \\ &\geq 1 . \end{aligned}$$

□

Theorem 7 *Let $G = (V, E)$ be an undirected graph with $n = |V|$ vertices and degree Δ. Let G_π be a priority dag induced on G by a random permutation π of V such that all vertex orderings are equally likely. For any constant $\epsilon > 0$ and sufficiently large n, there exists a directed path with $e^2(1+\epsilon)(\Delta + \ln\Delta \ln n / \ln\ln n)$ vertices in G_π with probability at most $n^{-\epsilon}$.*

PROOF. This proof begins by demonstrating in the same manner as Jones and Plassmann [37] that the probability of a k-vertex path in G_π is at most $n(e\Delta/k)^k$. Let $p = \langle v_1, v_2, \ldots, v_k\rangle$ be a k-vertex path in G. For p to be a path in G_π, we must have that $\pi(v_1) < \pi(v_2) < \cdots < \pi(v_k)$. Of the $k!$ permutations of the vertices in p, exactly one is so ordered, and thus the probability that p is a path in G_π is at most $1/k!$. We now count the number of k-vertex paths in G. There are n choices for v_1 and then at most Δ choices for each v_i given the path $\langle v_1, v_2, \ldots, v_{i-1}\rangle$, for $i = 1, 2, \ldots, k$. Thus, the total number of k-vertex paths is at most $n\Delta^k$. By the union bound, the probability that a k-vertex path exists in G_π is at most $n\Delta^k/k! \le n(e\Delta/k)^k$ which follows from Stirling's approximation [13, p. 57].

For the choice $k = e^2(1+\epsilon)(\Delta + \ln\Delta \ln n/\ln\ln n)$, we now bound the probability that a k-vertex path exists in G_π. We consider the cases when $\Delta < \ln n$ and $\ln n \le \Delta$ separately.

For the case when $\Delta < \ln n$, we assume that $\Delta \ge 2$ since the theorem is trivially true when $\Delta \in \{0, 1\}$. We apply Lemma 6 with $\alpha = \Delta$ and $\beta = \ln n$ and diminish the magnitude of the negative exponent to conclude

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

For the case $\Delta \geq \ln n$, we use the facts that $k \geq (1+\epsilon)\ln n$ and $k \geq e^2\Delta$, whence we have

$$\begin{aligned} n(e\Delta/k)^k &\leq n(1/e)^k \\ &\leq ne^{-(1+\epsilon)\ln n} \\ &= n^{-\epsilon} . \end{aligned}$$

□

Corollary 8 *Let $G = (V, E)$ be an undirected graph with degree Δ, and let G_π be a priority dag induced on G by a random permutation π of V. Then, the expected length of the longest directed path in G_π is $O(\Delta + \ln \Delta \ln V / \ln \ln V)$.* □

Lemma 9 *Let $G = (V, E)$ be a graph with degree Δ. For any random permutation π of V, the number of rounds in the message-passing model for Algorithm* JP *to vertex-color G is equal to the length of the longest directed path in the priority dag G_π induced on G by π.*

PROOF. This proof is paraphrased from [37]. Algorithm JP requires at least as many rounds to color the graph G as the length k of the longest directed path p in the priority graph G_π. This is clear, since the processing of each vertex in p is dependent on its predecessor.

Furthermore, Algorithm JP requires at most k rounds to color G. Suppose not. Then there must be a $k+1$-vertex path $p' = \langle v_1, v_2, \ldots, v_{k+1} \rangle$ in G_π. There must be some vertex $v' \in p'$ such that no predecessor of v' is processed in the round immediately preceding v', otherwise, there would be a $k+1$-vertex path in G_π. However, Algorithm JP colors every vertex v in the round immediately following the round of the last predecessor of v, which is a contradiction. □

Corollary 10 *Let $G = (V, E)$ be a graph with degree Δ. For any random permutation π of V, the expected number of rounds in the message-passing model for Algorithm* JP *to vertex-color G is $O(\Delta + \ln \Delta \ln V / \ln \ln V)$.* □

Chapter 8

Log Degree Ordering Heuristic

This chapter introduces the log-degree ordering heuristic for greedy coloring algorithms. We prove that the log-degree ordering heuristic can be used by JP while increasing the span by only a logarithmic factor. The effectiveness of the ordering heuristic is evaluated empirically on a set of large real world graphs by comparing the number of colors obtained when using different vertex-ordering heuristics.

Practitioners use ordering heuristics to reduce the number of colors that greedy coloring algorithms use [2, 9]. Ordering heuristics permute the vertices of a graph to help greedy coloring algorithms find colorings using a small number of colors. Although for every graph $G = (V, E)$ there exists a permutation of the vertices $\pi^\star$ such that GREEDY$(G = (V, E), \pi^\star)$ produces a coloring using the minimum number of colors, finding such a $\pi^\star$ is well known to be NP-hard [29]. Several ordering heuristics have been developed with much success on real-world graphs, however. A particularly popular ordering heuristic is the ***degree-ordering*** heuristic [3,23,37], in which vertices are ordered in decreasing order of degree. More precisely, the degree-ordering heuristic permutes the graph G by π such that, for two vertices $u, v \in V$, we have $\pi(v) < \pi(u)$ if $\deg(v) < \deg(u)$. Because it is deterministic, the degree-ordering heuristic is vulnerable to adversarial input graphs which force GREEDY to produce a coloring with $\Delta + 1$ colors. For instance, although the "crown" graph on $|V|$ vertices is 2-colorable, GREEDY produces a $|V|/2$ coloring using degree ordering [36].

We introduce the log-degree ordering heuristic, which exhibits nearly the same

asymptotic span as random ordering with MULTICORE-JP and generates colorings with approximately as few colors as degree ordering on real-world graphs, while maintaining robustness in the face of adversarial graphs. For the log-degree heuristic priority function π, $\pi(v)$ is less than $\pi(u)$ for two vertices $u, v \in V$ if $\lceil \log(\deg(v)) \rceil < \lceil \log(\deg(u)) \rceil$ or $\text{RAND}(v) < \text{RAND}(u)$ and $\lceil \log(\deg(v)) \rceil = \lceil \log(\deg(u)) \rceil$, for some random permutation function RAND.

Theorem 11 *Using the log-degree heuristic priority function π, Algorithm* MULTICORE-JP *colors all vertices of a graph $G = (V, E)$ with degree Δ in expected $O(\Delta \ln \Delta + \ln^3\Delta \ln V / \ln \ln V)$ span.*

PROOF. Let $V_i \subseteq V$ be the set of vertices such that $\lceil \lg|\deg(v)| \rceil = i$ for all $v \in V_i$. The log-degree heuristic guarantees that for every vertex $v \in V_i$ and $u \in V_j$, we have $\pi(v) < \pi(u)$ when $i < j$. Thus, the longest directed path p in the priority dag G_π induced on G by π must traverse some subset of vertices in $V_{\lceil \lg \Delta \rceil}$ then $V_{\lceil \lg \Delta \rceil - 1}$ and so on down to V_1. The log-degree heuristic also guarantees that for two vertices $v_1, v_2 \in V_i$, the priorities $\pi(v_1)$ and $\pi(v_2)$ are uniformly random. It then follows from Corollary 8 that the expected length of the longest path through V_i is $O(2^i + i \ln V / \ln \ln V)$. By linearity of expectation, it follows that the expected length of the longest path p through $V = \{V_1, V_2, \ldots, V_{\lceil \lg \Delta \rceil}\}$ is

$$\begin{aligned} \mathrm{E}[|p|] &= \sum_{i=1}^{\lceil \lg \Delta \rceil} O\left(2^i + i \frac{\ln V}{\ln \ln V}\right) \\ &= O\left(\Delta + \ln^2\Delta \frac{\ln V}{\ln \ln V}\right). \end{aligned}$$

Finally, the theorem follows by Lemma 13. □

Figure 8-1 illustrates the effect of different ordering heuristics on the depth of the priority dag G_π and on the number of colors used by MULTICORE-JP. For some graphs, such as *web-Google* and *cit-Patents* the use of vertex ordering heuristics appears to have little beneficial impact on the depth of the priority dag. For *soc-LiveJournal* and *com-orkut*, however, the input ordering induces a priority dag that

is about double the depth of that induced by the random ordering. This highlights the importance of incoporating randomness into the vertex orering to handle adversarial graphs. The log-degree ordering heuristic consistently induces a priority dag of similar depth to a random order. Furthermore, the number of colors used with the log-degree ordering is comparable to that of the degree ordering heuristic. These results demonstrate that the log-degree ordering is an appealing compromise between the random and degree ordering heuristics. The log-degree ordering provides theoretical guarantees on the parallelism of JP while producing colorings of similar quality to the degree-ordering heuristic.

Graph Info			***# Colors***		***Depth of*** G_π	
	Name		*IO*	*DO*	*IO*	*DO*
V	*E*	Δ	*RO*	*LDO*	*RO*	*LDO*
	soc-LiveJournal1		*352*	*324*	*76*	*43*
4,847,571	85,702,474	20,333	*330*	*327*	*41*	*40*
	web-Google		*44*	*45*	*24*	*29*
916,428	8,644,102	6,332	*44*	*44*	*30*	*28*
	com-orkut		*175*	*87*	*98*	*47*
3,072,627	234,370,166	33,313	*129*	*99*	*41*	*45*
	cit-Patents		*17*	*14*	*19*	*45*
16,518,949	33,037,900	793	*20*	*15*	*40*	*44*
	as-skitter		*103*	*71*	*53*	*42*
11,095,299	22,190,604	35,455	*83*	*71*	*34*	*39*

Figure 8-1: In practice, the number of colors needed by a greedy coloring algorithm can be reduced by applying vertex ordering heuristics. We compare the performance of MULTICORE-JP to an optimized serial algorithm under four different ordering heuristics: Input order (IO), Random order (RO), Degree order (DO), and Log-degree order (LDO).

Chapter 9

Multicore Implementation of Jones-Plassmann

This chapter extends the Jones-Plassman coloring algorithm to the shared-memory setting. While JP is fast in the message-passing model, there are two challenges which must be overcome to implement it efficiently on a modern shared-memory machine. The first challenge arises on line 1 of COLOR, when a vertex is to be colored, in finding the lowest available color quickly, efficiently, and in parallel. The second challenge arises in line 3 of COLOR when multiple workers contend for shared-memory locations in an attempt to discover whether a vertex is ready to be colored. We address both of these challenges and present an implementation MULTICORE-JP which colors all vertices of a graph $G = (V, E)$ given a random permutation π of V in expected $O(\Delta \ln \Delta + \ln^2 \Delta \ln V / \ln \ln V)$ span and $\Theta(V + E)$ work in the CRQW contention model. Empirical results show that the multicore version of JP achieves 3-5$\times$ speedup on 12 cores over the optimized serial code when both use the log-degree ordering heuristic.

Finding the smallest available color

A simple method for coloring a vertex v scans its neighbors in parallel and marks the ith entry of a byte-array of size $\deg(v)$ if the ith color has been assigned to a neighbor. The byte-array can then be scanned to identify the smallest available

color. This approach, however, can suffer from write contention if multiple processors attempt to write to the same memory location in the byte array.

In MULTICORE-JP, an alternative approach which suffers from less memory contention is used to identify the smallest available color. In line 1 of MULTICORE-COLOR we assign $v.color$ the lowest numbered color currently not taken by any vertex $u \in Pred[v]$. The procedure GET-COLOR solves the problem of quickly and efficiently assigning the color using a parallel integer sort [21] and a parallel MIN reduction [13, Ch.27]. $colors[i]$ and $candidates[i]$ correspond to C_i and S_i, respectively, in Lemma 12.

Lemma 12 *To assign the lowest available color to a vertex with degree* Δ *requires* $\Theta(\Delta)$ *work and* $\Theta(\log \Delta)$ *span in the CRQW contention model.*

PROOF. The lowest available color must lie in the interval $[0, \Delta]$ since there are at most Δ neighboring vertices which can take at most Δ of the $\Delta + 1$ possible colors. Let C_i be the minimum of $\Delta + 1$ and the color of the ith predecessor in ascending sorted order. If C is not already sorted, it may be sorted using the method of Gibbons *et al.* in $O(\Delta)$ work and $O(\log \Delta)$ span in the CRQW contention model [21]. If C_0 does not equal 0, then it is available and we can select it. Otherwise, we proceed.

We define the ith available color S_i as C_i+1 if $C_{i+1} > C_i$ and $\Delta+1$ otherwise, which represents a *gap* between the colors taken by the predecessors. Each available color can be computed in parallel in $\Theta(\Delta)$ work and $\Theta(\ln \Delta)$ span and the lowest available color is the least among them. Finally, we find the minimum color in parallel using a binary tree, collecting the S array pairwise with a MIN reduction [13, Ch.27] taking $\Theta(\Delta)$ work and $\Theta(\log \Delta)$ span. □

Assigning responsibility for vertex coloring

The second implementation challenge occurs in line 3 of MULTICORE-COLOR(v) where the caller detects whether or not all predecessors $Pred[v_{succ}]$ of a vertex $v_{succ} \in Succ[v]$ have been colored and, crucially, whether or not the caller of IS-LAST-TO-ARRIVE(v_{succ}) is responsible for coloring v_{succ}. For correctness, exactly one call to the method

IS-LAST-TO-ARRIVE(v) must return TRUE for each vertex $v \in V$, which implies that the method must appear to take place atomically [34, Ch.2]. A naive way to implement IS-LAST-TO-ARRIVE is to use of an atomic counter $v.counter$ for each vertex $v \in V$, which is initialized to $|Pred[v]|$. Then, $v.counter$ may be decremented via an atomic FETCH-AND-DECREMENT($v.counter$) call upon each call of IS-LAST-TO-ARRIVE(v). If the return value of FETCH-AND-DECREMENT($v.counter$) is 1, then the caller is indeed the last to arrive and is responsible for coloring vertex v. This implementation has two drawbacks. First, it requires the use of an atomic read-modify-write instruction—and therefore a strictly more powerful machine [33]—since it is not possible to provide mutual exclusion to an arbitrary number of threads with $\Theta(1)$ state without such an instruction [34]. Second, an atomic counter occupying a single location in memory creates the opportunity for memory contention, as each worker calling IS-LAST-TO-ARRIVE(v) must obtain exclusive ownership [55] over the memory location containing $v.counter$ prior to executing FETCH-AND-DECREMENT($v.counter$), which creates an $O(\Delta)$ delay due to memory contention.

```
MULTICORE-JP(G = (V, E), π)
1  parallel for all v ∈ V
2      INIT-VERTEX(v)
3  parallel for all v ∈ V | Pred[v] == ∅
4      MULTICORE-COLOR(v)
```

```
MULTICORE-COLOR(v)
1  v.color = GET-COLOR(v)
2  parallel for all v_succ ∈ Succ[v]
3      if IS-LAST-TO-ARRIVE(v_succ, v)
4          MULTICORE-COLOR(v_succ)
```

Figure 9-1: MULTICORE-JP is an implementation of the basic recursive structure of Algorithm JP adapted for a modern shared-memory multicore computer. That is, algorithms that comprehend memory contention and computations local to each vertex are developed to make methods IS-LAST-TO-ARRIVE and GET-COLOR, respectively, fast and efficient. INIT-VERTEX creates the binary tournament structures which are used by IS-LAST-TO-ARRIVE.

We implement IS-LAST-TO-ARRIVE(v) using a ***tournament***, which quickly and efficiently decides if the caller should color vertex $v \in V$. Figure 9-2 depicts a tourna-

```
IS-LAST-TO-ARRIVE(v, v_pred)
1  i = v.leaf-index[v_pred]
2  while i is not root of tournament tree
3      i = i.parent
4      if I am first to this node
5          return FALSE
6  return TRUE
```

```
GET-COLOR(v)
1  if Pred[v] == ∅ return 0
2  colors = PARALLEL-INTEGER-SORT(Pred[v])
3  if colors[0] > 0 return 0
4  parallel for i = 0 to |Pred[v]| - 2
5      if colors[i + 1] > colors[i] + 1
6          candidates[i] = colors[i] + 1
7      else
8          candidates[i] = |Pred[v]| + 1
9  return PARALLEL-FIND-MIN(candidates)
```

ment, where the dark circles denote a *loss* and an arrow denotes a *win*. A tournament resembles a binary tournament barrier [32] except that for our purposes only one worker needs to know that all predecessors of a vertex have been colored, whereas a barrier forces *all* participating workers to wait. We define the *winner* of a node in the tournament as the last to arrive. Any worker which finds that it is *first* to arrive loses and is free to do other work, satisfied that some other predecessor will be responsible for coloring the vertex. The winner moves toward the root of the tree, competing at each node until it loses or wins the root node, in which case it is responsible for coloring the vertex. We determine the ordering of the two workers at a node in the tournament using Peterson's Algorithm for two-thread mutual exclusion [57], which does not require the use of an atomic read-modify-write instruction. The single shared variable in Peterson's Algorithm is the only memory location in the tournament that is written concurrently and by exactly two workers. Furthermore, the critical section of each node in the tournament is held for $\Theta(1)$ time yielding bounded contention.

In line 1 of IS-LAST-TO-ARRIVE(v, v_{pred}) local variable i is assigned the value $v.\mathit{leaf\text{-}index}[v_{pred}]$, an assignment which maps each predecessor of v to a unique leaf in the tournament tree. During the initialization phase in line 2 of MULTICORE-JP,

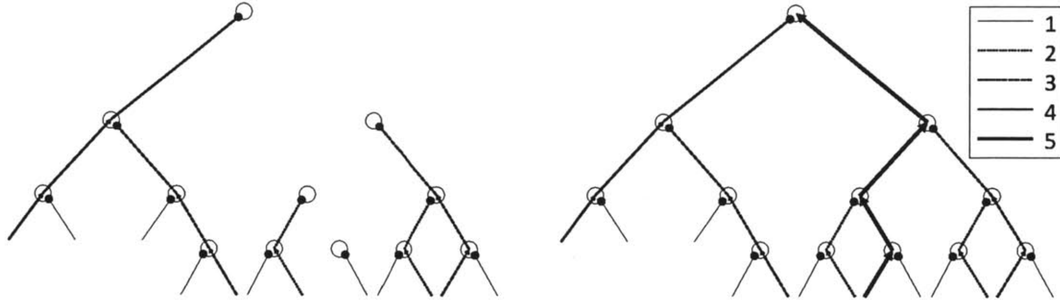


Figure 9-2: A tournament before and after the last predecessor calls IS-LAST-TO-ARRIVE. The legend specifies the order in which each predecessor arrived. Dark circles indicate that the predecessor arrived first at the node in question, whereas arrows indicate that the predecessor arrived last and proceeded on toward the root.

the mapping $v.\mathit{leaf\text{-}index}[u]$ is defined for each vertex $v \in V$ and $u \in \mathit{Pred}[v]$ via a hash-table which is constructed in $\Theta(\mathit{Pred}[v])$ work and span.

Theorem 13 *Given any arbitrary permutation π of vertices V, if Algorithm* JP *requires $O(R)$ rounds to color a graph $G = (V, E)$ with degree Δ in the message-passing model then Algorithm* MULTICORE-JP *has $O(R \ln \Delta)$ span in the CRQW contention model.*

PROOF. By Lemma 9 the number of rounds in the message-passing model required by JP to color the graph G equals the length of the longest directed path through the priority dag G_π induced on G by π. There are at most Δ participants in the tournament of any particular vertex. A tournament is a balanced binary tree, thus the number of steps through any such tournament is $O(\log \Delta)$, each of which takes $\Theta(1)$ time. The winner of the tournament for a vertex v assigns a color using the method GET-COLOR in $O(\ln \Delta)$ span by Lemma 12. Thus, there is at most $O(\ln \Delta)$ delay through any vertex and Algorithm MULTICORE-JP colors G with permutation π with $O(R \ln \Delta)$ span in the CRQW contention model. □

Corollary 14 *Given a random permutation π of the vertices in a graph $G = (V, E)$, Algorithm* MULTICORE-JP *colors all vertices in expected $O(\Delta \ln \Delta + \ln^2 \Delta \ln V / \ln \ln V)$*

span in the CRQW contention model. □

Theorem 15 *The expected work required for* MULTICORE-JP *to color all vertices of a graph $G = (V, E)$ is $\Theta(V + E)$ and thus is work-efficient.*

PROOF. Consider the tournament tree for a vertex v with $Pred[v]$ participants and exactly $2(Pred[v] - 1)$ internal edges (it is a complete binary tree). As exactly one winner follows each edge in the tournament, the work associated with it is $\Theta(Pred[v])$. As well, the coloring of vertex v requires $\Theta(Pred[v])$ work by Lemma 12. Every edge in G participates in exactly one tournament, thus the total work in all tournaments and vertex colorings is $\Theta(E)$. Because every vertex in G is processed exactly once, it follows that the expected total work required for MULTICORE-JP to color every vertex of a graph G is $\Theta(V + E)$. □

Empirical evaluation

The performance of our multicore implementation of the Jones-Plassmann algorithm was evaluated on a set of five benchmark graphs and four vertex ordering heuristics. The algorithm was compared against a serial greedy coloring algorithm that utilized the same ordering heuristics. The vertex ordering is provided to each algorithm in a convenient format: the serial code is provided a sorted list of vertex identifiers, and MULTICORE-JP is provided a mapping of vertex id to a priority. Providing the serial code a presorted list is, in a sense, giving it an advantage — since in practice it may be necessary for it to perform a sort. This experimental decision, however, allows the results to be more easily interpreted.

Both implementations were optimized to obtain the best possible performance on our experimental machine. In particular, the GET-COLOR routine for both MULTICORE-JP and GREEDY was optimized to track the assignment of colors to neighbors using a 64-bit bit-vector, which produces a color in $\Theta(1)$ time if the color is in $\{0, \ldots, 63\}$. In addition, MULTICORE-JP makes use of software prefetching to parallelize access to the successors in line 2 of MULTICORE-COLOR. Finally, each leaf of the tournament structure in IS-LAST-TO-ARRIVE resolves $\Theta(1)$ predecessors, allowing us to merely

hash the vertex number to a leaf rather than store the mapping in *v.leaf-index*. This optimization reduces overhead while still bounding the contention. Figure 8-1 suggests that the log-degree ordering heuristic indeed delivers a number of colors comparable to degree ordering without compromising performance relative to random ordering.

Graph Info			*Input (IO)*		*Random (RO)*		*Degree (DO)*		*Log-degree (LDO)*	
Name			T_s	T_1	T_s	T_1	T_s	T_1	T_s	T_1
V	E	Δ	T_s/T_{12}	T_1/T_{12}	T_s/T_{12}	T_1/T_{12}	T_s/T_{12}	T_1/T_{12}	T_s/T_{12}	T_1/T_{12}
soc-LiveJournal1			0.89	2.79	1.72	2.9	1.3	2.46	1.71	2.5
4,847,571	85,702,474	20,333	1.8	5.61	3.51	5.92	3.29	6.23	4.35	6.35
web-Google			0.11	0.27	0.17	0.29	0.14	0.25	0.17	0.25
916,428	8,644,102	6,332	1.7	4.21	4.27	7.07	3.81	6.94	4.64	6.89
com-orkut			2.32	9.19	3.28	10.09	2.89	8.27	3.25	8.62
3,072,627	234,370,166	33,313	1.18	4.67	1.96	6.03	2.03	5.83	2.3	6.11
cit-Patents			0.61	1.57	1.78	1.73	0.85	1.64	1.65	1.66
16,518,949	33,037,900	793	2.58	6.65	7.68	7.48	3.79	7.37	7.17	7.22
as-skitter			0.22	0.74	0.88	0.77	0.29	0.64	0.71	0.64
11,095,299	22,190,604	35,455	1.5	5.09	6.57	5.74	2.28	5.1	5.65	5.13

Figure 9-3: Empirical comparison of an optimized serial greedy coloring algorithm with MULTICORE-JP. All units are in seconds.

The results of this evaluation are summarized in Figure 9-3. The time T_s provides the runtime of an optimized serial greedy coloring algorithm, T_1 provides the runtime of MULTICORE-JP algorithm when run on a single core, and T_{12} provides the runtime of MULTICORE-JP when run on 12 cores. For the random, degree, and log-degree orderings MULTICORE-JP achieves 3-5$\times$ speedup on 12 cores over the optimized serial code. It appears to perform a bit worse on the input ordering — obtaining only 1.5-3$\times$ speedup. This difference is likely due to the serial code's more predictable memory access pattern when using the input order.

Chapter 10

Conclusion

In this thesis, I have shown how chromatic scheduling and deterministic parallel coloring algorithms can be used to implement dynamic data-graph computations that provide update atomicity and guarantee determinism. The latter benfit of determinism can be obtained at low cost through the use of the MULTICORE-JP algorithm. These techniques, therefore, can be used to improve both the performance and semantic guarantees of existing systems including GraphLab and GraphChi.

Several avenues remain to be explored more deeply in future work. There are opportunities to further improve the performance of data-graph computations in the multicore setting. It is likely, for example, that data-graph computations would benefit a scheduling algorithm that attempts to use each processors local cache optimally. Another interesting area for future work is the use of chromatic scheduling for dynamic data-graph computations in distributed systems. Scaling data-graph computations to a cluster environment requires that one implement fault tolerance mechanisms to maintain correctness when individual nodes fail. The algorithms presented in this thesis may be utilized to guarantee that each node in such a cluster is deterministic — allowing for simpler fault tolerance strategies. A replicated state machine, for example, could be used for each node in the cluster. Alternatively, logs could be maintained for each node recording its communication with other nodes in the cluster — allowing a node to be "restored" by simply playing back the communication logs to a new machine.

Bibliography

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

[2] Md. Mostofa Ali Patwary, Assefaw H. Gebremedhin, and Alex Pothen. New multithreaded ordering and coloring algorithms for multicore architectures. In Proceedings of the 17th international conference on Parallel processing - Volume Part II, Euro-Par'11, pages 250–262, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[4] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. J. Algorithms, 7:567–583, December 1986.

[5] Leonid Barenboim and Michael Elkin. Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. In 41st Annual ACM Symposium on Theory of Computing, STOC '09, pages 111–120, New York, NY, USA, 2009. ACM.

[6] Dimitri P. Bertsekas and John N. Tsitsiklis. Parallel and Distributed Computation: Numerical Methods. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In

Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, October 2008.

[8] Doruk Bozdağ, Assefaw H Gebremedhin, Fredrik Manne, Erik G Boman, and Umit V Catalyurek. A framework for scalable greedy coloring on distributed-memory parallel computers. Journal of Parallel and Distributed Computing, 68(4):515–535, 2008.

[9] Doruk Bozdağ, Assefaw H. Gebremedhin, Fredrik Manne, Erik G. Boman, and Umit Catalyurek. A framework for scalable greedy coloring on distributed memory parallel computers. Journal of Parallel and Distributed Computing, 68(4):515–535, 2008.

[10] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. Comput. Netw. ISDN Syst., 30(1-7):107–117, April 1998.

[11] Richard Cole and Ofer Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures, pages 169–178, Santa Fe, New Mexico, June 1989.

[12] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. Inf. Control, 64:2–22, March 1985.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, third edition, 2009.

[14] Joseph C. Culberson. Iterated greedy graph coloring and the difficulty landscape. Technical report, University of Alberta, 1992.

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

[16] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), pages 1–11, June 1997.

[17] Michael R. Garey and David S. Johnson. Computers and Intractability. W.H. Freeman and Company, 1979.

[18] Alan E. Gelfand and Adrian F. M. Smith. Sampling-based approaches to calculating marginal densities. Journal of the American Statistical Association, 85(410):398–409, 1990.

[19] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. Pattern Analysis and Machine Intelligence, IEEE Transactions on, PAMI-6(6):721 –741, November 1984.

[20] Phillip B. Gibbons. A more practical PRAM model. In ACM Symposium on Parallel Algorithms and Architectures, pages 158–168, June 1989.

[21] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. Efficient low-contention parallel algorithms. In Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures, SPAA '94, pages 236–247, New York, NY, USA, 1994. ACM.

[22] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 638–648, Arlington, Virginia, January 1994.

[23] Robert K. Gjertsen Jr., Mark T. Jones, and Paul E. Plassmann. Parallel heuristics for improved, balanced graph colorings. Journal of Parallel and Distributed Computing, 37:171–186, 1996.

[24] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. In SIAM J. Disc. Math, pages 315–324, 1987.

[25] Mark Goldberg and Thomas Spencer. A new parallel algorithm for the maximal independent set problem. SIAM Journal on Computing, 18:419–427, April 1989.

[26] Gene H. Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. Journal of the Society of Industrial and Applied Mathematics Series B: Numerical Analysis, 1964.

[27] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In USENIX Conference on Operating Systems Design and Implementation, OSDI, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[28] Raymond Greenlaw, James H. Hoover, and Walter L. Ruzzo. Limits to Parallel Computation: P-Completeness Theory. Oxford University Press, USA, April 1995.

[29] Frank Harary. Graph Theory. Addison-Wesley, 1972.

[30] William Hasenplaugh, Tim Kaler, Charles E Leiserson, Schardl, and Tao B. Chromatic scheduling of dynamic data-graph computations, including parallel coloring of arbitrary-degree graphs. Test, 2013.

[31] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The cilkview scalability analyzer. In SPAA, pages 145–156, 2010.

[32] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. Int. J. Parallel Program., 17(1):1–17, February 1988.

[33] M. Herlihy. Wait-free synchronizatoin. ACM Transactions on Programming Languages and Systems, 13(1):124–149, January 1991.

[34] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[35] F. L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. Journal of Mathematical Physics, 1927.

[36] David S. Johnson. Worst case behavior of graph coloring algorithms. In Proceedings of the 5th Southeastern Conference on Combinatorics, Graph Theory, and Computing, Winnipeg, Ontario, Canada, 1974.

[37] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. SIAM Journal on Scientific Computing, 14(3):654–669, May 1993.

[38] D. P. Koester, S. Ranka, and G. C. Fox. A parallel Gauss-Seidel algorithm for sparse power system matrices. In SuperComputing '94, pages 184–193, 1994.

[39] Fabian 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, 2009.

[40] Fabian Kuhn and Roger Wattenhofer. On the complexity of distributed graph coloring. In PODC, pages 7–15, 2006.

[41] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: large-scale graph computation on just a pc. In USENIX Conference on Operating Systems Design and Implementation, OSDI, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[42] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. CoRR, abs/0810.1355, 2008.

[43] Nathan Linial. Distributive graph algorithms global solutions from local data. In Proceedings of the 28th Annual Symposium on Foundations of Computer Science, SFCS '87, pages 331–335, Washington, DC, USA, 1987. IEEE Computer Society.

[44] Nathan Linial. Locality in distributed graph algorithms. SIAM J. Comput., 21(1):193–201, February 1992.

[45] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: a framework for machine

learning and data mining in the cloud. Proceedings of the VLDB Endowment, 5(8):716–727, April 2012.

[46] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow., 5(8):716–727, April 2012.

[47] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In Conference on Uncertainty in Artificial Intelligence (UAI), July 2010.

[48] Michael Luby. A simple parallel algorithm for the maximal independent set problem. SIAM Journal on Computing, 15(4):1036–1053, 1986.

[49] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz 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.

[50] Andrew McCallum. Cora data set. Available from `http://people.cs.umass.edu/~mccallum/data.html`.

[51] Tom Mitchell. NPIC500 data set. Available from `http://www.cs.cmu.edu/~tom/10709_fall09/NPIC500.pdf`, 2009.

[52] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In Proceedings of Uncertainty in AI, pages 467–475, 1999.

[53] Robert H. B. Netzer and Barton P. Miller. What are race conditions? ACM Letters on Programming Languages and Systems, 1(1):74–88, March 1992.

[54] Kamal Nigam and Rayid Ghani. Analyzing the effectiveness and applicability of co-training. In Proceedings of the 9th International Conference on Information and Knowledge Management, CIKM '00, pages 86–93, 2000.

[55] Mark S. Papamarcos and Janak 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.

[56] Judea Pearl. Probabilistic reasoning in intelligent systems: networks of plausible inference. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[57] G. L. Peterson. Myths about the mutual exclusion problem. Information Processing Letters, 12(3):115–116, June 1981.

[58] Parag Singla and Pedro Domingos. Entity resolution with markov logic. In In ICDM, pages 572–582. IEEE Computer Society Press, 2006.

[59] A. M. TURING. Rounding-off errors in matrix processes. The Quarterly Journal of Mechanics and Applied Mathematics, 1(1):287–308, 1948.

[60] 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.

[61] David P. Williamson and David B. Shmoys. The Design of Approximation Algorithms. Cambridge University Press, New York, 2011.