MIT Libraries | DSpace@MIT

MIT Open Access Articles

Executing dynamic data-graph computations deterministically using chromatic scheduling

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Tim Kaler, William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson. 2014. Executing dynamic data-graph computations deterministically using chromatic scheduling. In Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures (SPAA '14). ACM, New York, NY, USA, 154-165.

As Published: http://dx.doi.org/10.1145/2612669.2612673

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/100928

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling

Tim Kaler William Hasenplaugh Tao B. Schardl Charles E. Leiserson

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139

ABSTRACT

A ***data-graph computation*** — popularized by such programming systems as Galois, Pregel, GraphLab, PowerGraph, and GraphChi — is an algorithm that performs local updates on the vertices of a graph. During each round of a data-graph computation, an ***update function*** atomically modifies the data associated with a vertex as a function of the vertex's prior data and that of adjacent vertices. A ***dynamic*** data-graph computation updates only an active subset of the vertices during a round, and those updates determine the set of active vertices for the next round.

This paper introduces PRISM, a chromatic-scheduling algorithm for executing dynamic data-graph computations. PRISM uses a vertex-coloring of the graph to coordinate updates performed in a round, precluding the need for mutual-exclusion locks or other nondeterministic data synchronization. A ***multibag*** data structure is used by PRISM to maintain a dynamic set of active vertices as an unordered set partitioned by color. We analyze PRISM using work-span analysis. Let $G = (V,E)$ be a degree-Δ graph colored with χ colors, and suppose that $Q \subseteq V$ is the set of active vertices in a round. Define $size(Q) = |Q| + \sum_{v \in Q} \deg(v)$, which is proportional to the space required to store the vertices of Q using a sparse-graph layout. We show that a P-processor execution of PRISM performs updates in Q using $O(\chi(\lg(Q/\chi) + \lg\Delta) + \lg P)$ span and $\Theta(size(Q) + \chi + P)$ work. These theoretical guarantees are matched by good empirical performance. We modified GraphLab to incorporate PRISM and studied seven application benchmarks on a 12-core multicore machine. PRISM executes the benchmarks 1.2–2.1 times faster than GraphLab's nondeterministic lock-based scheduler while providing deterministic behavior.

This paper also presents PRISM-R, a variation of PRISM that executes dynamic data-graph computations deterministically even when updates modify global variables with associative operations. PRISM-R satisfies the same theoretical bounds as PRISM, but its implementation is more involved, incorporating a ***multivector*** data structure to maintain an ordered set of vertices partitioned by color.

This research was supported in part by the National Science Foundation under Grants CNS-1017058, CCF-1162148, and CCF-1314547 and in part by grants from Intel Corporation and Foxconn Technology Group. Tao B. Schardl was supported in part by an NSF Graduate Research Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPAA'14, June 23–25, 2014, Prague, Czech Republic.
Copyright 2014 ACM 978-1-4503-2821-0/14/06 ...$15.00.
http://dx.doi.org/10.1145/2612669.2612697.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; E.1 [**Data Structures**]: *distributed data structures, graphs and networks*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*graph algorithms, sequencing and scheduling*

Keywords

Data-graph computations; multicore; multithreading; parallel programming; chromatic scheduling; determinism; scheduling; work stealing

1. INTRODUCTION

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

Intuitively, a data-graph computation is an algorithm that performs local updates on the vertices of a data graph. Several software systems have been implemented to support parallel data-graph computations, including Galois [63], Pregel [78], GraphLab [75, 76], PowerGraph [48], and GraphChi [64]. These systems often support "complex" data-graph computations, in which data can be associated with edges as well as vertices and updating a vertex v can modify any data associated with v, v's incident edges, and the vertices adjacent to v. For ease in discussing chromatic scheduling, however, we shall principally restrict ourselves to "simple" data-graph computations (which correspond to "edge-consistent" computations in GraphLab), although most of our results straightforwardly extend to more complex models. Indeed, six out of the seven GraphLab applications described in [74, 75] are simple data-graph computations.

Updates to vertices proceed in ***rounds***, where each vertex can be updated at most once per round. In a ***static*** data-graph computation, the ***activation set*** Q_r of vertices updated in a round r — the set of ***active*** vertices — is determined *a priori*. Often, a static data-graph computation updates every vertex in each round. Static data-graph computations include Gibbs sampling [41, 42], iterative graph coloring [30], and n-body problems such as the fluidanimate PARSEC benchmark [10].

We shall be interested in ***dynamic*** data-graph computations, where the activation set changes round by round. Dynamic data-graph computations include the Google PageRank algorithm [21],

loopy belief propagation [82, 87], coordinate descent [32], co-EM [84], alternating least-squares [54], singular-value decomposition [47], and matrix factorization [95].

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

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

The update $S = f(v)$ implicitly computes as a side effect a new value for the data associated with v as a function of the old data associated with v and v's neighbors. The update returns a set $S \subseteq \text{Adj}[v]$ of vertices that must be updated later in the computation. During a round r of the dynamic data-graph computation, each vertex $v \in Q_r$ is updated at most once, that is, Q_r is a set, not a multiset. For example, an update $f(v)$ might activate a neighbor u only if the value of v changes significantly.

The advantage of dynamic over static data-graph computations is that they avoid performing many unnecessary updates. Studies in the literature [75,76] show that dynamic execution can enhance the practical performance of many applications. We confirmed these findings by implementing static and dynamic versions of several data-graph computations. The results for a PageRank algorithm on a power-law graph of 1 million vertices and 10 million edges were typical. The static computation performed approximately 15 million updates, whereas the dynamic version performed less than half that number of updates.

A serial reference implementation

Before we address the issues involved in scheduling and executing dynamic data-graph computations in parallel, let us first hone our intuition with a serial algorithm for the problem. Figure 1 gives the pseudocode for SERIAL-DDGC. This algorithm schedules the updates of a data-graph computation by maintaining a FIFO queue Q of activated vertices that have yet to be updated. Sentinel values enqueued in Q on lines 4 and 9 demarcate the rounds of the computation such that the set of vertices in Q after the rth sentinel has been enqueued is the activation set Q_r for round r.

Given a data-graph $G = (V,E)$, an update function f, and an initial activation set Q_0, SERIAL-DDGC executes the data-graph computation $\langle G, f, Q_0 \rangle$ as follows. Lines 1–2 initialize Q to contain all vertices in Q_0. The **while** loop on lines 5–14 then repeatedly dequeues the next scheduled vertex $v \in Q$ on line 5 and executes the update $f(v)$ on line 11. Executing $f(v)$ produces a set S of activated vertices, and lines 12–14 check each vertex in S for membership in Q, enqueuing all vertices in S that are not already in Q.

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

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

The size of Q_r is asymptotically the space needed to store all the vertices in Q_r and their incident edges using a standard sparse-graph representation, such as compressed-sparse-rows (CSR) format [93]. For example, if $Q_0 = V$, we have $size(Q_0) = |V| + 2|E|$ by the handshaking lemma [29, p. 1172–3]. Let us make the reasonable assumption that the time to execute $f(v)$ serially is proportional to $\deg(v)$. If we implement the queue as a dynamic (resizable) table [29, Section 17.4], then line 14 executes in $\Theta(1)$ amortized time. All other operations in the **for** loop on lines 12–14 take $\Theta(1)$ time, and thus all vertices activated by executing $f(v)$ are examined in $\Theta(\deg(v))$ time. The total time spent updating the vertices in Q_r is therefore $\Theta(Q_r + \sum_{v \in Q_r} \deg(v)) = \Theta(size(Q_r))$, which is ***linear*** time: time proportional to the storage requirements for the vertices in Q_r and their incident edges.

```
SERIAL-DDGC(G, f, Q_0)
 1  for v ∈ Q_0
 2      ENQUEUE(Q, v)
 3  r = 0
 4  ENQUEUE(Q, NIL) // Sentinel NIL denotes the end of a round.
 5  while Q ≠ {NIL}
 6      v = DEQUEUE(Q)
 7      if v == NIL
 8          r += 1
 9          ENQUEUE(Q, NIL)
10      else
11          S = f(v)
12          for u ∈ S
13              if u ∉ Q
14                  ENQUEUE(Q, u)
```

Figure 1: Pseudocode for a serial algorithm to execute a data-graph computation $\langle G, f, Q_0 \rangle$. SERIAL-DDGC takes as input a data graph G and an update function f. The computation maintains a FIFO queue Q of activated vertices that have yet to be updated and sentinel values NIL, each of which demarcates the end of a round. An update $S = f(v)$ returns the set $S \subseteq \text{Adj}[v]$ of vertices activated by that update. Each vertex $u \in S$ is added to Q if it is not currently scheduled for a future update.

Parallelizing dynamic data-graph computations

The salient challenge in parallelizing data-graph computations is to deal effectively with races between updates, that is, logically parallel updates that read and write common data. A ***determinacy race*** [36] (also called a ***general race*** [83]) occurs when two logically parallel instructions access the same memory location and at least one of them writes to that location. Two updates in a data-graph computation ***conflict*** if executing them in parallel produces a determinacy race. A parallel scheduler must manage or avoid conflicting updates to execute a data-graph computation correctly and deterministically.

The standard approach to preventing races associates a mutual-exclusion lock with each vertex of the data graph to ensure that an update on a vertex v does not proceed until all locks on v and v's neighbors have been acquired. Although this locking strategy prevents races, it can incur substantial overhead from lock acquisition and contention, hurting application performance, especially when update functions are simple. Moreover, because runtime happenstance can determine the order in which two logically parallel updates acquire locks, the data-graph computation can act nondeterministically: different runs on the same inputs can produce different results. Without repeatability, parallel programming is arguably much harder [19,67]. Nondeterminism confounds debugging.

A known alternative to using locks is ***chromatic scheduling*** [1,9], which schedules a data-graph computation based on a coloring of the data-graph computation's ***conflict graph*** — a graph with an edge between two vertices if updating them in parallel would produce a race. For a simple data-graph computation, the conflict graph is simply the data graph itself. The idea behind chromatic scheduling is fairly simple. Chromatic scheduling begins by computing a ***(vertex) coloring*** of the conflict graph — an assignment of colors to the vertices such that no two adjacent vertices share the same color. Since no edge in the conflict graph connects two

Benchmark	|V|	|E|	χ	*GraphLab*	CILK+LOCKS	PRISM
PR/G	916,428	5,105,040	43	14.9	14.8	12.4
PR/L	4,847,570	68,475,400	333	217.1	227.9	172.3
ID/250	62,500	249,000	4	4.0	3.8	2.5
ID/1000	1,000,000	3,996,000	4	44.3	44.3	20.7
FBP/C1	87,831	265,204	2	13.7	7.4	7.6
FBP/C3	482,920	160,019	2	27.9	14.7	14.6
ALS/N	187,722	20,597,300	6	126.1	113.4	77.1

Figure 2: Comparison of dynamic data-graph schedulers on seven application benchmarks. All runtimes are in seconds and were calculated by taking the median 12-core execution time of 5 runs on an Intel Xeon X5650 with hyperthreading disabled. The runtime of PRISM includes the time used to color the input graph. PR/G and PR/L run a PageRank algorithm on the web-Google [72] and soc-LiveJournal [4] graphs, respectively. ID/250 and ID/1000 run an image denoise algorithm to remove Gaussian noise from 2D grayscale images of dimension 250 by 250 and 1000 by 1000. FBP/C1 and FBP/C3 perform belief propagation on a factor graph provided by the cora-1 and cora-3 datasets [79, 91]. ALS/N runs an alternating least squares algorithm on the NPIC-500 dataset [81].

vertices of the same color, updates on all vertices of a given color can execute in parallel without producing races. To execute a round of a data-graph computation, the set of activated vertices Q is partitioned into χ ***color sets*** — subsets of Q containing vertices of a single color. Updates are applied to vertices in Q by serially stepping through each color set and updating all vertices within a color set in parallel. The result of a data-graph computation executed using chromatic scheduling is equivalent to that of a slightly modified version of SERIAL-DDGC that starts each round (immediately before line 9 of Figure 1) by sorting the vertices within its queue by color.

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

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

Contributions

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

We analyze the performance of PRISM using work-span analysis [29, Ch. 27]. The ***work*** of a computation is intuitively the total number of instructions executed, and the ***span*** corresponds to the longest path of dependencies in the parallel program. We shall make the reasonable assumption that a single update $f(v)$ executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span.[1] Under this assumption, on a degree-Δ data graph G colored using χ colors, PRISM executes the updates on the vertices in the activation set Q_r of a round r on P processors in $O(size(Q_r) + \chi + P)$ work and $O(\chi(\lg(Q_r/\chi) + \lg\Delta) + \lg P)$ span.

Surprisingly, the "price of determinism" incurred by using chromatic scheduling instead of the more common locking strategy appears to be negative for real-world applications. As Figure 2 indicates, on seven application benchmarks, PRISM executes 1.2–2.1 times faster than GraphLab's comparable, but nondeterministic, locking strategy. This performance gap is not due solely to superior engineering or load balancing. A similar performance overhead is observed in a comparably engineered lock-based scheduling algorithm, CILK+LOCKS. PRISM outperforms CILK+LOCKS on all but one benchmark and is on average (geometric mean) 1.18 times faster.

PRISM behaves deterministically as long as every update is ***pure***: it modifies no data except for that associated with its target vertex. This assumption precludes the update function from modifying global variables to aggregate or collect values. To support this common use pattern, we describe an extension to PRISM, called PRISM-R, which executes dynamic data-graph computations deterministically even when updates modify global variables using associative operations. PRISM-R replaces each multibag PRISM uses with a "multivector," maintaining color sets whose contents are ordered deterministically. PRISM-R executes in the same theoretical bounds as PRISM, but its implementation is more involved.

Outline

The remainder of this paper is organized as follows. Section 2 reviews dynamic multithreading, the parallel programming model in which we describe and analyze our algorithms. Section 3 describes PRISM, the chromatic-scheduling algorithm for dynamic data-graph computations. Section 4 describes the multibag data structure PRISM uses to represent its color sets. Section 5 presents our theoretical analysis of PRISM. Section 6 describes a Cilk Plus [56] implementation of PRISM and presents empirical results measuring this implementation's performance on seven application benchmarks. Section 7 describes PRISM-R and its multivector data structure. Section 8 offers some concluding remarks.

2. BACKGROUND

We implemented the PRISM algorithm in Cilk Plus [56], a dynamic multithreading concurrency platform. This section provides background on the dag model of multithreading that embodies this and other similar concurrency platforms, including MIT Cilk [39], Cilk++ [70], Fortress [2], Habenero [6, 24], Hood [18], Java Fork/Join Framework [66], Task Parallel Library (TPL) [69], Threading Building Blocks (TBB) [88], and X10 [26]. We review

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

the Cilk model of multithreading, the notions of work and span, and the basic properties of the work-stealing runtime systems underlying these concurrency platforms. We briefly discuss worker-local storage, which PRISM's multibag data structure uses to achieve efficiency.

The Cilk model of multithreading

The Cilk model of multithreading [16, 17] is described in tutorial fashion in [29, Ch. 27]. The model views the executed computation resulting from running a parallel program as a ***computation dag*** in which each vertex denotes an instruction, and edges denote parallel control dependencies between instructions. To analyze the theoretical performance of a multithreaded program, such as PRISM, we assume that the program executes on an ***ideal parallel computer***, where each instruction executes in unit time, the computer has ample bandwidth to shared memory, and concurrent reads and writes incur no overheads due to contention.

We shall assume that algorithms for the dag model are expressed using the Cilk-like primitives [29, Ch. 27] **spawn**, **sync**, and **parallel for**. The keyword **spawn** when preceding a function call F allows F to execute in parallel with its ***continuation*** — the statement immediately after the spawn of F. The complement of **spawn** is the keyword **sync**, which acts as a local barrier and prevents statements after the **sync** from executing until all earlier spawned functions return. These keywords can be used to implement other convenient parallel control constructs, such as the **parallel for** loop, which allows all of its iterations to operate logically in parallel. The work of a **parallel for** loop with n iterations is the total number of instructions in all executed iterations. The span is $\Theta(\lg n)$ plus the maximum span of any loop iteration. The $\Theta(\lg n)$ span term comes from the fact that the runtime system executes the loop iterations using parallel divide-and-conquer, and thus fans out the iterations as a balanced binary tree in the dag.

Work-span analysis

Given a multithreaded program whose execution is modeled as a dag A, we can bound the P-processor running time $T_P(A)$ of the program using ***work-span analysis*** [29, Ch. 27]. Recall that the work $T_1(A)$ is the number of instructions in A, and that the span $T_\infty(A)$ is the length of a longest path in A. Greedy schedulers [20, 35, 49] can execute a deterministic program with work T_1 and span T_∞ on P processors in time T_P satisfying

$$\max\{T_1/P, T_\infty\} \le T_p \le T_1/P + T_\infty \,, \tag{1}$$

and a similar bound can be achieved by more practical "work-stealing" schedulers [16, 17]. The ***speedup*** of an algorithm on P processors is T_1/T_P, which Inequality (1) shows to be at most P in theory. The ***parallelism*** T_1/T_∞ is the greatest theoretical speedup possible for any number of processors.

Work-stealing runtime systems

Runtime systems underlying concurrency platforms that support the dag model of multithreading usually implement a ***work stealing*** scheduler [17, 23, 50], which operates as follows. When the runtime system starts up, it allocates as many operating-system threads, called ***workers***, as there are processors. Each worker keeps a ***ready queue*** of tasks that can operate in parallel with the task it is currently executing. Whenever the execution of code generates parallel work, the worker puts the excess work into the queue. Whenever it needs work, it fetches work from its queue. When a worker's ready queue runs out of tasks, however, the worker becomes a ***thief*** and "steals" work from another ***victim*** worker's queue. If an application exhibits sufficient parallelism compared to the actual number of workers/processors, one can prove mathematically that the computation executes with linear speedup.

```
PRISM(G, f, Q_0)
 1  χ = COLOR-GRAPH(G)
 2  r = 0
 3  Q = Q_0
 4  while Q ≠ ∅
 5     C = MB-COLLECT(Q)
 6     for C ∈ C
 7        parallel for v ∈ C
 8           active[v] = FALSE
 9           S = f(v)
10           parallel for u ∈ S
11              begin atomic
12              if active[u] == FALSE
13                 active[u] = TRUE
14                 MB-INSERT(Q, u, color[u])
15              end atomic
16     r = r + 1
```

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

Worker-local storage

Worker-local storage refers to memory that is private to a particular worker thread in a parallel computation. In this paper, in a P-processor execution of a parallel program, a variable x implemented using worker-local storage is stored as an array of P copies of x. A worker accesses its local copy of x using a runtime-provided worker identifier to index the array of worker-local copies of x. The Cilk Plus runtime system, for example, provides the `__cilkrts_get_worker_number()` API call, which returns an integer identifying the current worker. PRISM assumes the existence of a runtime-provided GET-WORKER-ID function that executes in $\Theta(1)$ time and returns an integer from 0 to $P-1$.

3. THE PRISM ALGORITHM

This section presents PRISM, a chromatic-scheduling algorithm for executing dynamic data-graph computations deterministically. We describe how PRISM differs from the serial algorithm in Section 1, including how it maintains activation sets that are partitioned by color using a multibag data structure.

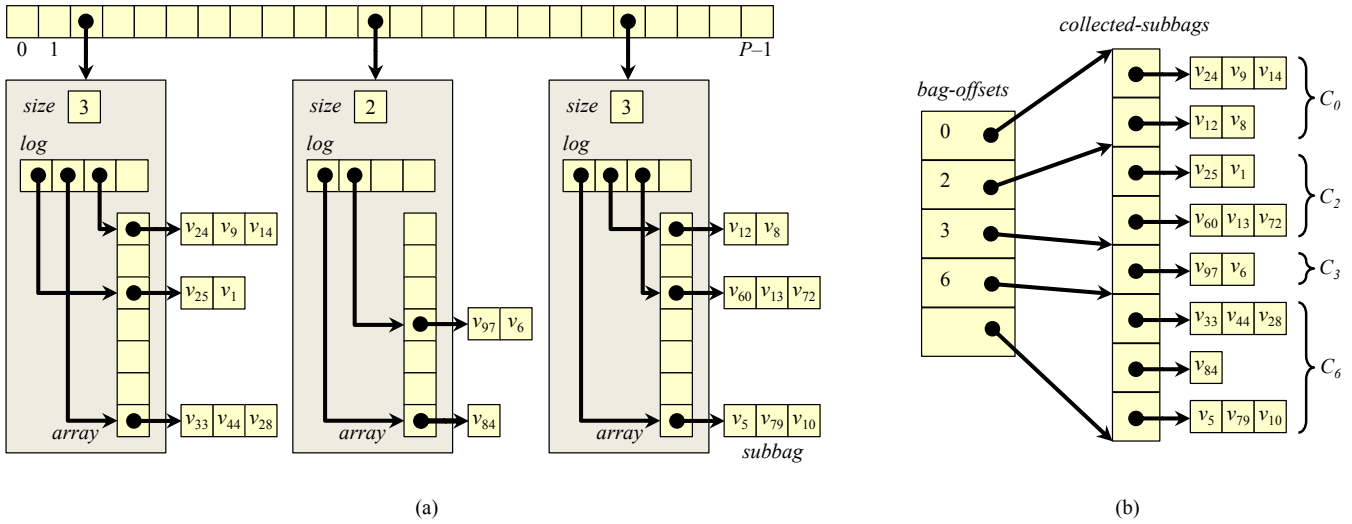
Figure 3 shows the psuedocode for PRISM, which differs from the SERIAL-DDGC routine from Figure 1 in two main ways: the use of a multibag data structure to implement Q, and the call to COLOR-GRAPH on line 1 to color the data graph.

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

- MB-INSERT(Q, v, k) inserts an element v into bag C_k in Q. A multibag supports parallel MB-INSERT operations.
- MB-COLLECT(Q) produces a collection $\mathscr{C}$ that represents a list of the nonempty bags in Q, emptying Q in the process.

PRISM stores a distinct color set in each bag of a multibag Q. Section 4 describes and analyzes the implementation of the multibag data structure.

PRISM calls COLOR-GRAPH on line 1 to color the given data graph $G = (V, E)$ and obtain the number χ of colors used. Although it is NP-complete to find either an ***optimal*** coloring of a graph [40] — a coloring that uses the smallest possible number of colors — or a $O(V^\varepsilon)$-approximation of the optimal col-



(a) (b)

Figure 4: A multibag data structure. **(a)** A multibag containing 19 elements distributed across 4 distinct bags. The contents of each bag are partitioned across the corresponding subbags in 3 nonempty worker-local SPA's. **(b)** The output of MB-COLLECT when executed on the multibag in **(a)**. Sets of subbags in *collected-subbags* are labeled with the bag C_k that their union represents.

oring [77], as Section 6 discusses, an optimal coloring is not necessary for PRISM to perform well in practice, as long as the data graph is colored with sufficiently few colors. Many parallel coloring algorithms exist that satisfy the needs of PRISM (see, for example, [3, 5, 45, 46, 52, 59, 61, 62, 73, 94]). In fact, if the data-graph computation performs sufficiently many updates, a $\Theta(V+E)$-work greedy coloring algorithm, such as that introduced by Welsh and Powell [96], can suffice as well. Our program implementation of PRISM uses a multicore variant of the Jones and Plassmann algorithm [59] that produces a deterministic $(\Delta+1)$-coloring of a degree-Δ graph $G=(V,E)$ in linear work and $O(\lg V + \lg\Delta \cdot \min\{\sqrt{E}, \Delta + \lg\Delta \lg V / \lg\lg V\})$ span [52].

Let us now see how PRISM uses chromatic scheduling to execute a dynamic data-graph computation $\langle G, f, Q_0\rangle$. After line 1 colors G, line 3 initializes the multibag Q with the initial activation set Q_0, and then the **while** loop on lines 4–16 executes the rounds of the data-graph computation. At the start of each round, line 5 collects the nonempty bags $\mathscr{C}$ from Q, which correspond to the nonempty color sets for the round. Lines 6–15 iterate through the color sets $C \in \mathscr{C}$ sequentially, and the **parallel for** loop on lines 7–15 processes the vertices of each C in parallel. For each vertex $v \in C$, line 9 performs the update $S = f(v)$, which returns a set S of activated vertices, and lines 10–15 insert into Q the vertices in S that are not currently active.

To ensure that an activated vertex is not added to Q multiple times in a round, PRISM maintains an array *active* of Boolean flags, where entry $active[v]$ indicates whether vertex v is currently active. Conceptually, flag $active[v]$ indicates whether $v \in Q$ in the modified version of SERIAL-DDGC that is analogous to PRISM. To process a vertex v, line 8 of PRISM sets $active[v]$ to FALSE, whereas SERIAL-DDGC removes v from Q. Lines 12 and 13 of PRISM set $active[u]$ to TRUE only if $active[u]$ was previously FALSE, whereas SERIAL-DDGC adds vertex u to Q only if $u \notin Q$. The **begin atomic** and **end atomic** statements on lines 11 and 15 ensure that $active[u]$ is read and set atomically, thereby preventing a data race from adding vertex u to PRISM's multibag Q multiple times. Although alternative strategy exist to avoid this atomicity check, our empirical studies indicate that this limited use of atomics seems to work well in practice.

4. THE MULTIBAG DATA STRUCTURE

This section presents the multibag data structure employed by PRISM. The multibag uses worker-local sparse accumulators [44] and an efficient parallel collection operation. We describe how the MB-INSERT and MB-COLLECT operations are implemented, and we analyze them using work-span analysis [29, Ch. 27]. When used in a P-processor execution of a parallel program, a multibag Q of χ bags storing n elements supports MB-INSERT in $\Theta(1)$ worst-case time and MB-COLLECT in $O(n+\chi+P)$ work and $O(\lg n + \chi + \lg P)$ span.

A ***sparse accumulator (SPA)*** [44] implements an array that supports lazy initialization of its elements. A SPA T contains a sparsely populated array $T.array$ of elements and a log $T.log$, which is a list of indices of initialized elements in $T.array$. To implement multibags, we shall only need the ability to create a SPA, access an arbitrary SPA element, or delete all elements from a SPA. For simplicity, we shall assume that an uninitialized array element in a SPA has a value of NIL. When an array element $T.array[i]$ is modified for the first time, the index i is appended to $T.log$. An appropriately designed SPA T storing $n = |T.log|$ elements admits the following performance properties:

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

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

Implementation of MB-INSERT *and* MB-COLLECT

The worker-local SPA's enable a multibag Q to support parallel MB-INSERT operations without creating races. Figure 5 shows

```
MB-INSERT(Q, v, k)
1  p = GET-WORKER-ID()
2  if Q[p].array[k] == NIL
3      APPEND(Q[p].log, k)
4      Q[p].array[k] = new subbag
5  APPEND(Q[p].array[k], v)
```

Figure 5: Pseudocode for the MB-INSERT multibag operation to insert an element v into bag C_k in multibag Q.

the pseudocode for MB-INSERT. When a worker p executes MB-INSERT(Q,v,k), it inserts element v into the subbag $C_{k,p}$ as follows. Line 1 calls GET-WORKER-ID to get worker p's identifier. Line 2 checks if subbag $C_{k,p}$ stored in $Q[p].array[k]$ is initialized, and if not, lines 3 and 4 initialize it. Line 5 inserts v into $Q[p].array[k]$.

Conceptually, the MB-COLLECT operation extracts the bags in Q to produce a compact representation of those bags that can be read efficiently. Figure 4(b) illustrates the compact representation of the elements of the multibag from Figure 4(a) that MB-COLLECT returns. This representation consists of a pair $\langle$*bag-offsets*, *collected-subbags*$\rangle$ of arrays that together resemble the representation of a graph in a CSR format. The *collected-subbags* array stores all of the subbags in Q sorted by their corresponding bag's index. The *bag-offsets* array stores indices in *collected-subbags* that denote the sets of subbags comprised by each bag. In particular, in this representation, the contents of bag C_k are stored in the subbags in *collected-subbags* between indices *bag-offsets*$[k]$ and *bag-offsets*$[k+1]$.

Figure 6 sketches how MB-COLLECT converts a multibag Q stored in worker-local SPA's into the representation illustrated in Figure 4(b). Steps 1 and 2 create an array *collected-subbags* of nonempty subbags from the worker-local SPA's in Q. Each subbag $C_{k,p}$ in *collected-subbags* is tagged with the integer index k of its corresponding bag $C_k \in Q$. Step 3 sorts *collected-subbags* by these index tags, and Step 4 creates the *bag-offsets* array. Step 5 removes all elements from Q, thereby emptying the multibag.

Analysis of multibags

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

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

PROOF. Consider each step of a call to MB-INSERT(Q,v,k). The GET-WORKER-ID procedure on line 1 obtains the executing worker's identifier p from the runtime system in $\Theta(1)$ time, and line 2 checks if the entry $Q[p].array[k]$ is empty in $\Theta(1)$ time. Suppose that $Q[p].log$ and each subbag in $Q[p].array$ are implemented as dynamic arrays that use a deamortized table-doubling scheme [22]. Lines 3–5 then take $\Theta(1)$ time each to append k to $Q[p].log$, create a new subbag in $Q[p].array[k]$, and append v to $Q[p].array[k]$. □

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

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

PROOF. We analyze each step of MB-COLLECT in turn. We shall use a helper procedure PREFIX-SUM(A), which computes the all-prefix sums of an array A of n integers in $\Theta(n)$ work and $\Theta(\lg n)$ span. (Blelloch [11] describes an appropriate implementation of PREFIX-SUM.) Step 1 replaces each entry in $Q[p].log$ in each worker-local SPA $Q[p]$ with the appropriate index-subbag pair $\langle k, C_{k,p}\rangle$ in parallel, which requires $\Theta(m+P)$ work and $\Theta(\lg m+\lg P)$ span. Step 2 gathers all index-subbag pairs into a single array. Suppose that each worker-local SPA $Q[p]$ is augmented with the size of $Q[p].log$, as Figure 4(a) illustrates. Executing PREFIX-SUM on these sizes and then copying the entries of $Q[p].log$ into *collected-subbags* in parallel therefore completes Step 2 in $\Theta(m+P)$ work and $\Theta(\lg m+\lg P)$ span. Step 3 can sort the *collected-subbags* array in $\Theta(m+\chi)$ work and $\Theta(\lg m+\chi)$ span using a variant of a parallel radix sort [15, 27, 98] as follows:

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

Figure 6: Pseudocode for the MB-COLLECT multibag operation. Calling MB-COLLECT on a multibag Q produces a pair of arrays *collected-subbags*, which contains all nonempty subbags in Q sorted by their associated bag's index, and *bag-offsets*, which associates sets of subbags in Q with their corresponding bag.

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

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

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

5. ANALYSIS OF PRISM

This section analyzes the performance of PRISM using work-span analysis [29, Ch. 27]. We derive bounds on the work and span of PRISM for any simple data-graph computation $\langle G, f, Q_0\rangle$. Recall that we make the reasonable assumptions that a single update $f(v)$

Graph	$\|V\|$	$\|E\|$	χ	CILK+LOCKS	PRISM	*Coloring*
cage15	5,154,860	94,044,700	17	36.9	35.5	12%
soc-LiveJournal1	4,847,570	68,475,400	333	36.8	21.7	12%
randLocalDim25	1,000,000	49,992,400	36	26.7	14.4	18%
randLocalDim4	1,000,000	41,817,000	47	19.5	12.5	14%
rmat2Million	2,097,120	39,912,600	72	22.5	16.6	12%
powerGraph2Million	2,000,000	29,108,100	15	12.1	9.8	13%
3dgrid5m	5,000,210	15,000,600	6	10.3	10.3	7%
2dgrid5m	4,999,700	9,999,390	4	17.7	8.9	4%
web-Google	916,428	5,105,040	43	3.9	2.4	8%
web-BerkStan	685,231	7,600,600	200	3.9	2.4	8%
web-Stanford	281,904	2,312,500	62	1.9	0.9	11%
web-NotreDame	325,729	1,469,680	154	1.1	0.8	12%

Figure 7: Performance of PRISM versus CILK+LOCKS when executing $10 \cdot |V|$ updates of the PageRank [21] data-graph computation on a suite of six real-world graphs and six synthetic graphs. Column "*Graph*," identifies the input graph, and columns $|V|$ and $|E|$ specify the number of vertices and edges in the graph, respectively. Column χ gives the number of colors PRISM used to color the graph. Columns "CILK+LOCKS" and "PRISM" present 12-core running times in seconds for the respective schedulers. Each running time is the median of 5 runs. Column "*Coloring*" gives the percentage of PRISM's running time spent coloring the graph.

executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span, and that the update only activates vertices in $\mathrm{Adj}[v]$.

THEOREM 3. *Suppose that* PRISM *colors a degree-Δ data graph $G = (V,E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0 \rangle$. Then, on P processors,* PRISM *executes updates on all vertices in the activation set Q_r for a round r using $O(size(Q_r) + \chi + P)$ work and $O(\chi(\lg(Q_r/\chi) + \lg\Delta) + \lg P)$ span.*

PROOF. Let us first analyze the work and span of one iteration of lines 6–15 in PRISM, which perform the updates on the vertices belonging to one color set $C \in Q_r$. Consider a vertex $v \in C$. Lines 8 and 9 execute in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span. For each vertex u in the set S of vertices activated by the update $f(v)$, Lemma 1 implies that lines 11–15 execute in $\Theta(1)$ total work. The **parallel for** loop on lines 10–15 therefore executes in $\Theta(S)$ work and $\Theta(\lg S)$ span. Because $|S| \leq \deg(v)$, the **parallel for** loop on lines 7–15 thus executes in $\Theta(size(C))$ work and $\Theta(\lg C + \max_{v \in C} \lg(\deg(v))) = O(\lg C + \lg\Delta)$ span.

By processing each of the χ color sets belonging to Q_r, lines 6–15 therefore executes in $\Theta(size(Q_r) + \chi)$ work and $O(\chi(\lg(Q_r/\chi) + \lg\Delta))$ span. Lemma 2 implies that line 5 executes MB-COLLECT in $O(Q_r + \chi + P)$ work and $O(\lg Q_r + \chi + \lg P)$ span. The theorem follows, because $|Q_r| \leq size(Q_r)$. □

6. EMPIRICAL EVALUATION

This section describes our empirical evaluation of PRISM. We implemented PRISM in Cilk Plus [56] and compared its performance to that of three other schedulers for executing data-graph computations. This section presents three studies of PRISM's performance. The first study, which compared PRISM to a nondeterministic locking strategy, indicates that the overhead of managing multibags is less than the cost of a locking protocol. The second study, which compared PRISM to a chromatic scheduler for static data-graph computations, shows that the overhead of maintaining activation sets dynamically is only about 20% more than using static activation sets. This study suggests that it is worthwhile to use dynamic data-graph computations instead of static ones even if only modest amounts of work can be saved by avoiding unnecessary updates. The third study shows the performance of PRISM is relatively insensitive to the number of colors used to color the data graph, as long as there is sufficient parallelism.

Experimental setup

We implemented PRISM and the multibag data structure in Cilk Plus [56], compiling with the Intel C++ compiler, version 13.1.1. Our source code and data are available from `http://supertech.csail.mit.edu`. To implement the GET-COLOR procedure, the PRISM implementation used a deterministic multicore coloring algorithm [52], which was also coded in Cilk Plus. For comparison, we engineered three other schedulers for executing dynamic data-graph computations in parallel:

- CILK+LOCKS uses a locking scheme to avoid executing conflicting updates in parallel. The locking scheme associates a shared-exclusive lock with each vertex in the graph. Prior to executing an update $f(v)$, vertex v's lock is acquired exclusively, and a shared lock is acquired for each $u \in \mathrm{Adj}[v]$. A global ordering of locks is used to avoid deadlock.
- CHROMATIC treats the dynamic data-graph computation as a static one — it updates every vertex in every round — and it uses chromatic scheduling to avoid executing conflicting updates in parallel.
- ROUND-ROBIN treats the dynamic data-graph computation as a static one and uses the locking strategy to coordinate updates that conflict.

These schedulers were implemented within a modified multicore version of GraphLab. Specifically, we modified GraphLab's engine to replace GraphLab's explicit management of threads with the Cilk Plus runtime. Using the GraphLab framework, we tested these schedulers on existing GraphLab applications with little to no alteration of the application code.

The benchmarks were run on Intel Xeon X5650 machines, each with 12 2.67-GHz processing cores (hyperthreading disabled); 49 GB of DRAM; two 12-MB L3-caches, each shared among 6 cores; and private L2- and L1-caches of sizes 128 KB and 32 KB, respectively.

Overheads for locking and for chromatic scheduling

We compared the overheads associated with coordinating conflicting updates of a dynamic data-graph computation using locks versus using chromatic scheduling. We evaluated these overheads by comparing the 12-core execution times for PRISM and CILK+LOCKS to execute the PageRank [21] data-graph computation on a suite of graphs. We used PageRank for this study because of its relatively cheap update function, which updates a vertex v by first scanning v's incoming edges to aggregate the data from among

Benchmark	χ	*# Updates*	ROUND-ROBIN	CHROMATIC	PRISM
PR/L	333	48,475,700	19.2	11.4	17.8
FBP/C3	2	16,001,900	12.4	8.8	9.5
ID/1000	4	10,000,000	13.6	13.3	14.8
PR/G	43	9,164,280	2.4	1.0	2.0
FBP/C1	2	8,783,100	6.6	4.6	4.7
ALS/N	6	1,877,220	75.2	36.7	38.2
ID/250	4	625,000	0.8	0.9	1.0

Figure 8: Performance of three schedulers on the seven application benchmarks from Figure 2, modified so that all vertices are activated in every round. Column "*# Updates*" specifies the number of updates performed in the data-graph computation. Columns "ROUND-ROBIN," "CHROMATIC," and "PRISM," list the 12-core running times in seconds for the respective schedulers to execute each benchmark. Each running time is the median of 5 runs.

v's neighbors, and then scanning v's outgoing edges to activate v's neighbors.

We executed the PageRank application on a suite of six synthetic and six real-world graphs. The six real-world graphs came from the Stanford Large Network Dataset Collection (SNAP) [71], and the University of Florida Sparse Matrix Collection [31]. The six synthetic graphs were generated using the "randLocal," "powerLaw," "gridGraph," and "rMatGraph" generators included in the Problem Based Benchmark Suite [90].

We observed that PRISM often performed slightly fewer rounds of updates than CILK+LOCKS when both were allowed to run until convergence. Wishing to isolate scheduling overheads, we controlled this variation by limiting the total number of updates the two algorithms executed on a graph to 10 times the number of vertices. The accuracy requirements for the PageRank application were selected to ensure that neither scheduler completed the computation in fewer than $10 \cdot |V|$ updates.

Figure 7 presents our empirical results for this study. Figure 7 shows that over the 12 benchmark graphs, PRISM executed between 1.0 and 2.1 times faster than CILK+LOCKS on PageRank, exhibiting a geometric mean speedup factor of 1.5. From Figure 7, moreover, we see that, on average, 10.9% of PRISM's total running time is spent coloring the data graph. This statistic suggests that the cost PRISM incurs to color the data graph is approximately equal to the cost of executing $|V|$ updates. PRISM colors the data-graph once to execute the data-graph computation, however, meaning that its cost can be amortized over all of the updates in the data-graph computation. In contrast, the locking scheme that CILK+LOCKS implements incurs overhead for every update. Before updating a vertex v, CILK+LOCKS acquires each lock associated with v and every vertex $u \in \text{Adj}[v]$. For simple data-graph computations whose update functions perform relatively little work, this step can account for a significant fraction of the time to execute an update.

Dynamic-scheduling overhead

To investigate the overhead of using multibags to maintain activation sets, we compared the 12-core running times of PRISM, CHROMATIC, and ROUND-ROBIN on the seven benchmark applications from Figure 2. For this study, we modified the benchmarks slightly for each scheduler in order to provide a fair comparison. First, because PRISM typically executes fewer updates than a scheduler for static data-graph computations does, we modified the update functions PRISM used for each application so that every update on a vertex v always activates all vertices $u \in \text{Adj}[v]$. This modification guarantees that PRISM executes the same set of updates each round as a ROUND-ROBIN and CHROMATIC. Additionally, we modified the update functions used by ROUND-ROBIN and CHROMATIC to remove any work that would be unnecessary in a statically scheduled computation.

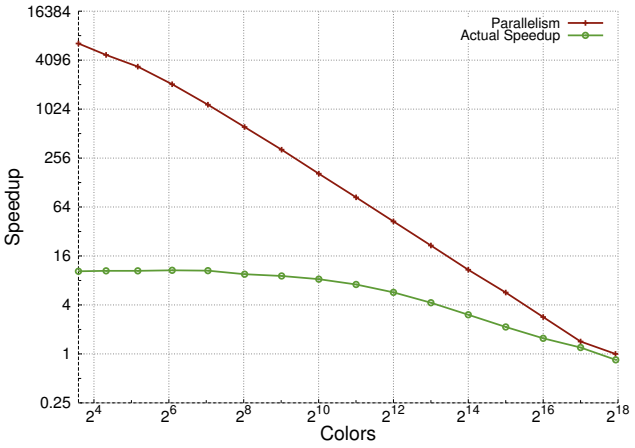


Figure 9: Scalability of PRISM on the image-denoise application as a function of χ, the number of colors used to color the data graph. The parallelism T_1/T_∞ is plotted together with the actual speedup T_1/T_{12} achieved on a 12-core execution. Parallelism values were measured using the Cilkview scalability analyzer [53], which measures the work and span of a Cilk program by counting instructions. Speedup on 12 cores was computed as the ratio of the 1-core and 12-core running times.

Figure 8 presents the results of these tests, which reveal the overhead PRISM incurs to maintain activation sets using a multibag. As Figure 8 shows, PRISM executed 1.0 to 2.0 times slower than CHROMATIC on the benchmarks with a geometric-mean slowdown of 1.2. PRISM nevertheless outperformed ROUND-ROBIN on all but the ID/250 and ID/1000 benchmarks due to ROUND-ROBIN's lock overhead. These results indicate that PRISM incurs relatively little overhead to maintain activation sets with multibags.

Scalability of PRISM

To study the parallel scalability of PRISM, we measured the parallelism T_1/T_∞ and the 12-core speedup T_1/T_{12} of PRISM executing the image-denoise application as we varied the number of colors used to color the application's data-graph. The image-denoise application performs belief propagation to remove Gaussian noise added to a gray-scale image. The data graph for the image-denoise application is a 2D grid in which each vertex represents a pixel, and there is an edge between any two adjacent pixels. PRISM typically colors this data-graph with just 4 colors. To perform this study, we artificially increased χ by repeatedly taking a random nonempty subset of the largest set of vertices with the same color and assigning those vertices a new color. Using this technique, we ran the image-denoise application on a 500-by-500 pixel input image for values of χ between 4 and $250,000$ — the last data point corresponding to a coloring that assigns all pixels distinct colors.

Figure 9 plots the results of these tests. As Figure 9 shows, although the parallelism of PRISM is inversely proportional to χ, PRISM's speedup on 12 cores is relatively insensitive to χ, as long as the parallelism is greater than 120. This result harmonizes with the rule of thumb that a program with parallelism above 10 times the number of workers ought to achieve near perfect linear speedup [28].

7. THE PRISM-R ALGORITHM

This section introduces PRISM-R, a chromatic-scheduling algorithm that executes a dynamic data-graph computation determinis-

```
FLATTEN(L, A, i)
1  A[i] = L
2  if L.left ≠ NIL
3      spawn FLATTEN(L.left, A, i − L.right.size − 1)
4  if L.right ≠ NIL
5      FLATTEN(L.right, A, i − 1)
6  sync
```

Figure 10: Pseudocode for the FLATTEN operation for a log tree. FLATTEN performs a post-order parallel traversal of a log tree to place its nodes into a contiguous array.

```
IDENTITY()
1  L = new log tree node
2  L.sublog = new vector
3  L.size = 1
4  L.left = NIL
5  L.right = NIL
6  return L

REDUCE(L_l, L_r)
1  L = IDENTITY()
2  L.size = L_l.size + L_r.size + 1
3  L.left = L_l
4  L.right = L_r
5  return L
```

Figure 11: Pseudocode for the IDENTITY and REDUCE log-tree reducer operations.

tically even when updates modify global variables using associative operations. The multivector data structure, which is a theoretical improvement to the multibag, is used by PRISM-R to maintain activation sets that are partitioned by color and ordered deterministically. We describe an extension of the model of simple data-graph computations that permits an update function to perform associative operations on global variables using a parallel reduction mechanism. In this extended model, PRISM-R executes dynamic data-graph computations deterministically while achieving the same work and span bounds as PRISM.

Data-graph computations that modify global variables

Several frameworks for executing data-graph computations allow updates to modify global variables in limited ways. Pregel aggregators [78], and GraphLab's sync mechanism [75], for example, both support data-graph computations in which an update can modify a global variable in a restricted manner. These mechanisms coordinate parallel modifications to a global variable using ***parallel reductions*** [12, 25, 57, 58, 60, 65, 80, 88], that is, they coordinate these modifications by applying them to local ***views*** (copies) of the variable and then ***reducing*** (combining) those copies together using a binary ***reduction operator***.

A ***reducer (hyperobject)*** [38, 68] is a general parallel reduction mechanism provided by Cilk Plus and other dialects of Cilk. A reducer is defined on an arbitrary data type T, called a ***view type***, by defining an IDENTITY operator and a binary REDUCE operator for views of type T. The IDENTITY operator creates a new view of the reducer. The binary REDUCE operator defines the reducer's reduction operator. A reducer is a particularly general reduction mechanism because it guarantees that, if its REDUCE operator is associative, then the final result in the global variable is deterministic: every parallel execution of the program produces the same result. Other parallel reduction mechanisms, including Pregel aggregators and GraphLab's sync mechanism, provide this guarantee only if the reduction operator is also commutative.

Although PRISM is implemented in Cilk Plus, PRISM does not produce a deterministic result if the updates modify global variables using a noncommutative reducer. The reason is that the order in which a multibag stores the vertices of an activation set depends on how the computation is scheduled. As a result, the order in which lines 7–15 of PRISM evaluate the vertices in a color set C can differ depending on scheduling. Therefore, if two updates on vertices in C modify the same reducer, then the relative order of these modifications can differ between runs of PRISM, even if a single worker happens to execute both updates.

PRISM-R extends PRISM to support data-graph computations that use reducers. Before presenting PRISM-R, we first describe the multivector data structure that is used by PRISM-R to maintain deterministically ordered color sets.

The multivector data structure

A ***multivector*** represents a list of χ ***vectors*** (ordered multisets). It supports two operations — MV-INSERT and MV-COLLECT — which are analogous to the multibag operations MB-INSERT and MB-COLLECT, respectively. We now sketch the design of the multivector data structure.

The multivector relies on properties of a work-stealing runtime system. Consider a parallel program modeled by a computation dag A in the Cilk model of multithreading. The ***serial execution order*** $R(A)$ of the program lists the vertices of A according to a depth-first traversal of A. A work-stealing scheduler partitions $R(A)$ into a sequence $R(A) = \langle t_0, t_1, \ldots, t_{M-1} \rangle$, where each ***trace*** $t_i \in R(A)$ is a contiguous subsequence of $R(A)$ executed by exactly one worker. A multivector represents each vector as a sequence of ***trace-local subvectors*** — subvectors that are modified within exactly one trace. The ordering properties of traces imply that concatenating a vector's trace-local subvectors in order produces a vector whose elements appear in the serial execution order. The multivector data structure assumes that a worker can query the runtime system to determine when it starts executing a new trace.

A multivector stores its nonempty trace-local subvectors in a ***log tree***, which represents an ordered multiset of elements and supports $\Theta(1)$-work append operations. A log tree is a binary tree in which each node L stores a dynamic array $L.sublog$. The ordered multiset that a log tree represents corresponds to a concatenation of the tree's dynamic arrays in a post-order tree traversal. Each log-tree node L is augmented with the size of its subtree $L.size$ counting the number of log-tree nodes in the subtree rooted at L. Using this augmentation, the operation FLATTEN$(L, A, L.size - 1)$ described in Figure 10 flattens a log tree rooted at L of n nodes and height h into a contiguous array A using $\Theta(n)$ work and $\Theta(h)$ span.

To handle parallel MV-INSERT operations, a multivector employs a ***log-tree reducer***, that is, a Cilk Plus reducer whose view type is a log tree. Figure 11 presents the pseudocode for the IDENTITY and REDUCE operations for the log-tree reducer. Notice that the log-tree reducer's REDUCE operation is logically associative, that is, for any three log-tree reducer views a, b, and c, the views produced by REDUCE(REDUCE$(a,b),c)$ and REDUCE$(a,$REDUCE$(b,c))$ represent the same ordered multiset.

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

Figure 12 sketches the MV-INSERT(Q, v, k) operation to insert element v into the vector $C_k \in Q$. MV-INSERT differs from MB-INSERT in two ways. First, when a new subvector is created and added to a SPA, lines 6–7 additionally append that subvector to $Q.log\text{-}reducer$, thereby maintaining the log-tree reducer. Second, lines 2–3 reset the contents of the SPA $Q[p]$ after worker p begins

```
MV-INSERT(Q, v, k)
1  p = GET-WORKER-ID()
2  if worker p began a new trace since last insert
3     reset Q[p]
4  if Q[p].array[k] == NIL
5     Q[p].array[k] = new subvector
6     L = GET-LOCAL-VIEW(Q.log-reducer)
7     APPEND(L.sublog, Q[p].array[k])
8  APPEND(Q[p].array[k], v)
```

Figure 12: Pseudocode for the MV-INSERT multivector operation to insert an element v into vector C_k in multivector Q.

executing a new trace, thereby ensuring that $Q[p]$ stores only trace-local subvectors.

Figure 13 sketches the MV-COLLECT operation, which returns a pair $\langle$*subvector-offsets*, *collected-subvectors*$\rangle$ analogous to the return value of MB-COLLECT. MV-COLLECT differs from MB-COLLECT primarily in that Step 1, which replaces Steps 1 and 2 in MB-COLLECT, flattens the log tree underlying Q.*log-reducer* to produce the unsorted array *collected-subvectors*. MV-COLLECT also requires that *collected-subvectors* be sorted using a stable sort on Step 2. The integer sort described in the proof of Lemma 2 for MB-COLLECT is a stable sort suitable for this purpose.

Analysis of multivector operations

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

LEMMA 4. *Executing* MV-INSERT *takes* $\Theta(1)$ *time in the worst case.*

PROOF. Resetting the SPA $Q[p]$ on line 3 can be done in $\Theta(1)$ worst-case time with an appropriate SPA implementation, and appending a new subvector to a log tree takes $\Theta(1)$ time. The theorem thus follows from the analysis of MB-INSERT in Lemma 1. □

Lemma 5 bounds the work and span of MV-COLLECT.

LEMMA 5. *Consider a computation A with span $T_\infty(A)$, and suppose that the contents of a multivector Q of χ vectors are distributed across m subvectors. Then a call to* MV-COLLECT(Q) *incurs $\Theta(m+\chi)$ work and $\Theta(\lg m+\chi+T_\infty(A))$ span.*

PROOF. Flattening the log-tree reducer in Step 1 is accomplished in two steps. First, the FLATTEN operation writes the nodes of the log tree to a contiguous array. FLATTEN has span proportional to the depth of the log tree, which is bounded by $O(T_\infty(A))$, since at most $O(T_\infty(A))$ reduction operations can occur along any path in A and REDUCE for log trees executes in $\Theta(1)$ work [38]. Second, using a parallel-prefix sum computation, the log entries associated with each node in the log tree can be packed into a contiguous array, incurring $\Theta(m)$ work and $\Theta(\lg m)$ span. Step 1 thus incurs $\Theta(m)$ work and $O(\lg m+T_\infty(A))$ span. The remaining steps of MV-COLLECT, which are analogous to those of MB-COLLECT and analyzed in Lemma 2, execute in $\Theta(\chi+\lg m)$ span. □

Deduplication

In addition to using a multivector in place of a multibag, PRISM-R differs from PRISM in how it ensures that the activation set for a given round contains each vertex at most once. Recall that PRISM uses atomic operations in lines 11–15 to determine whether to insert an activated vertex into its multibag. Although it is inconsequential in PRISM which update of a neighbor of a vertex caused the vertex to be added to the multibag, in PRISM-R, color sets must be ordered in a deterministic manner. To meet this requirement, PRISM-R assigns each vertex v a priority *priority*$[v]$, stores vertex-priority pairs in its multivector, (rather than just vertices), and replaces the Boolean array *active* in PRISM with a comparable array that stores priorities. For each vertex $u \in \text{Adj}[v]$ activated by update $f(v)$, the vertex-priority pair $\langle u, \textit{priority}[v]\rangle$ is inserted into the multivector, and a priority-write operation [89] is performed to set $\textit{active}[u] = \max\{\textit{active}[u], \textit{priority}[v]\}$ atomically. After executing MV-COLLECT in a round, PRISM-R performs a deduplication step, iterating over the vertex-priority pairs in parallel and deleting any pair $\langle v, p\rangle$ for which $p \neq \textit{active}[v]$.

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

Figure 13: Pseudocode for the MV-COLLECT multivector operation. Calling MV-COLLECT on a multivector Q produces a pair $\langle$*vector-offsets*, *collected-subvectors*$\rangle$ of arrays, where *collected-subvectors* contains all nonempty subvectors in Q sorted by their associated vector's color, and *vector-offsets* associates sets of subvectors in Q with their corresponding vector.

Analysis of PRISM-R

The next theorem shows that PRISM-R achieves the same theoretical bounds as PRISM.

THEOREM 6. *Let G be a degree-Δ data graph. Suppose that* PRISM-R *colors G using χ colors. Then* PRISM-R *executes updates on all vertices in the activation set Q_r for a round r of a simple data-graph computation $\langle G, f, Q_0\rangle$ in $O(\textit{size}(Q_r)+\chi)$ work and $O(\chi(\lg(Q_r/\chi)+\lg\Delta))$ span.*

PROOF. PRISM-R can perform a priority write to its *active* array with $\Theta(1)$ work, and it can remove duplicates from the output of MV-COLLECT in $O(\textit{size}(Q_r))$ work and $O(\lg(\textit{size}(Q_r))) = O(\lg Q_r + \lg\Delta)$ span. The theorem follows by applying Lemmas 4 and 5 appropriately to the analysis of PRISM in Theorem 3. □

8. CONCLUSION

Researchers over multiple decades have advocated that the difficulty of parallel programming can be greatly reduced by using some form of deterministic parallelism [7, 8, 13, 19, 33, 34, 36, 37, 43, 51, 55, 85, 86, 92, 97]. With a deterministic parallel program, the programmer observes no logical concurrency, that is, no nondeterminacy in the behavior of the program due to the relative and nondeterministic timing of communicating processes such as occurs when one process arrives at a lock before another. The semantics of a deterministic parallel program are therefore serial, and reasoning about such a program's correctness, at least in theory, is no harder than reasoning about the correctness of a serial program. Testing, debugging, and formal verification is simplified, because there is no need to consider all possible relative timings (interleavings) of operations on shared mutable state.

The behavior of PRISM corresponds to a variant of SERIAL-DDGC that sorts the activated vertices in its queue by color at the start of each round. Whether PRISM executes a given data

graph on 1 processor or many, it always behaves the same. With PRISM-R, this property holds even when the update function can perform reductions. Lock-based schedulers do not produce such a strong guarantee of determinism. Although updates in a round executed by a lock-based scheduler appear to execute according to some linear order, this order is nondeterministic due to races on the acquisition of locks.

Blelloch, Fineman, Gibbons, and Shun [14] argue that deterministic programs can be fast compared with nondeterministic programs, and they document many examples where the overhead for converting a nondeterministic program into a deterministic one is small. They even document a few cases where this "price of determinism" is *slightly* negative. To their list, we add the execution of dynamic data-graph computations as having a price of determinism which is *significantly* negative. We conjecture that research will uncover many more parallel applications that admit to efficient deterministic solutions.

9. ACKNOWLEDGMENTS

Thanks to Guy Blelloch of Carnegie Mellon University for sharing utility functions from his Problem Based Benchmark Suite [90]. Thanks to Aydın Buluç of Lawrence Berkeley Laboratory for helping us in our search for collections of large sparse graphs. Thanks to Uzi Vishkin of University of Maryland for helping us track down early work on parallel sorting. Thanks to Fredrik Kjolstad, Angelina Lee, and Justin Zhang of MIT CSAIL and Guy Blelloch, Julian Shun, and Harsha Vardhan Simhadri of Carnegie Mellon University for providing helpful discussions.

10. REFERENCES

[1] L. Adams and J. Ortega. A multi-color SOR method for parallel computation. In *ICPP*, 1982.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification Version 1.0*. Sun Microsystems, Inc., 2008.

[3] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 1986.

[4] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *ACM SIGKDD*, 2006.

[5] L. Barenboim and M. Elkin. Distributed $(\Delta+1)$-coloring in linear (in Δ) time. In *STOC*, 2009.

[6] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, et al. The Habanero multicore software research project. In *OOPSLA*, 2009.

[7] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[8] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[9] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., 1989.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[11] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

[12] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CS-92-103, Carnegie Mellon University, Pittsburgh, PA, 1992.

[13] G. E. Blelloch. Programming parallel algorithms. *CACM*, 1996.

[14] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of Principles and Practice of Parallel Programming*, pp. 181–192, 2012.

[15] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *SPAA*, 1991.

[16] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SICOMP*, 1998.

[17] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 1999.

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

[19] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First USENIX Conference on Hot Topics in Parallelism*, 2009.

[20] R. P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 1974.

[21] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Comput. Netw. ISDN Syst.*, 1998.

[22] A. Brodnik, S. Carlsson, E. Demaine, J. Ian Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *Algorithms and Data Structures*, volume 1663 of *LNCS*. Springer Berlin Heidelberg, 1999.

[23] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *ICFP*, 1981.

[24] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old x10. In *PPPJ*. ACM, 2011.

[25] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE TSE*, 2000.

[26] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[27] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *STOC*, 1986.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

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

[31] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM TOMS*, 2011.

[32] J. E. Dennis Jr. and T. Steihaug. On the successive projections approach to least-squares problems. *SIAM J. Numer. Anal.*, 1986.

[33] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[34] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A relaxed consistency deterministic computer. In *ASPLOS*, 2011.

[35] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEETC*, 1989.

[36] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA*, 1997.

[37] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[38] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA*, 2009.

[39] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, 1998.

[40] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1976.

[41] A. E. Gelfand and A. F. M. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 1990.

[42] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *PAMI*, 1984.

[43] P. B. Gibbons. A more practical PRAM model. In *SPAA*, 1989.

[44] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl*, 1992.

[45] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. In *SIAM J. Disc. Math*, 1987.

[46] M. Goldberg and T. Spencer. A new parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 1989.

[47] G. H. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *J. SIAM Numer. Anal.*, 1965.

[48] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[49] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 1966.

[50] R. H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Lisp and Functional Programming*, 1984.

[51] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 1985.

[52] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *SPAA*, 2014.

[53] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, 2010.

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

[55] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? Free will to choose. In *HPCA*, 2011.

[56] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[57] Intel Corporation. *Intel(R) Threading Building Blocks*, 2012. Available from `http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm`.

[58] K. E. Iverson. *A Programming Language*. John Wiley & Sons, 1962.

[59] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 1993.

[60] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[61] F. Kuhn. Weak graph colorings: distributed algorithms and applications. In *SPAA*, 2009.

[62] F. Kuhn and R. Wattenhofer. On the complexity of distributed graph coloring. In *PODC*, 2006.

[63] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[64] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *OSDI*. USENIX, 2012.

[65] C. Lasser and S. M. Omohundro. *The Essential *Lisp Manual, Release 1, Revision 3*. Thinking Machines Technical Report 86.15, Cambridge, MA, 1986.

[66] D. Lea. A Java fork/join framework. In *Conference on Java Grande*, 2000.

[67] E. A. Lee. The problem with threads. *IEEE Computer*, 2006.

[68] I.-T. A. Lee, A. Shafi, and C. E. Leiserson. Memory-mapping support for reducer hyperobjects. In *SPAA*, 2012.

[69] D. Leijen and J. Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007. Available from `http://msdn.microsoft.com/magazine/`.

[70] C. E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 2010.

[71] J. Leskovec. SNAP: Stanford network analysis platform. Available from `http://snap.stanford.edu/data/index.html`, 2013.

[72] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, 2008.

[73] N. Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 1992.

[74] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, pp. 716–727, 2012.

[75] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI*, 2010.

[76] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *PVLDB*, 2012.

[77] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *JACM*, 1994.

[78] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

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

[80] D. McCrady. Avoiding contention using combinable objects. Microsoft Developer Network blog post, Sept. 2008.

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

[82] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *UAI*, 1999.

[83] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1992.

[84] K. Nigam and R. Ghani. Analyzing the effectiveness and applicability of co-training. In *CIKM*, 2000.

[85] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[86] S. S. Patil. Closure properties of interconnections of determinate systems. In J. B. Dennis, editor, *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*. ACM, 1970.

[87] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[88] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[89] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *SPAA*, 2013.

[90] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012.

[91] P. Singla and P. Domingos. Entity resolution with markov logic. In *ICDM*, 2006.

[92] G. L. Steele Jr. Making asynchronous parallelism safe for the world. In *POPL*, 1990.

[93] J. Stoer, R. Bulirsch, R. H. Bartels, W. Gautschi, and C. Witzgall. *Introduction to Numerical Analysis*. Springer, New York, 2002.

[94] M. Szegedy and S. Vishwanathan. Locality based graph coloring. In *STOC*, 1993.

[95] A. M. Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1948.

[96] 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*, 1967.

[97] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[98] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Supercomputing*, 1991.