Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling

Executing Dynamic Data-Graph Computations Deterministically Using Chromatic Scheduling

Tim Kaler, William Hasenplaugh, Tao B. Schardl, Charles E. Leiserson, MIT CSAIL

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 $\Theta(size(Q) + P)$ work.

These theoretical guarantees are matched by good empirical performance. In order to isolate the effect of the scheduling algorithm on 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 a deterministically ordered set of vertices partitioned by color. Despite its additional complexity, PRISM-R is only marginally slower than PRISM. On the seven application benchmarks studied, PRISM-R incurs a 7% geometric mean overhead relative to PRISM.

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*

Additional Key Words and Phrases: 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.

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.

DOI: http://dx.doi.org/10.1145/0000000.0000000

A data-graph computation is an algorithm that performs "local" updates on the vertices of a data graph, taking as input data associated with a vertex and its neighbors. Several software systems have been implemented to support parallel data-graph computations, including GraphLab [Low et al. 2010, 2012], Pregel [Malewicz et al. 2010], Galois [Nguyen et al. 2013, 2014], PowerGraph [Gonzalez et al. 2012], Ligra[1] [Shun and Blelloch 2013; Shun et al. 2015], and GraphChi [Kyrola et al. 2012]. These systems can support "complex" data-graph computations, in which data can be associated with edges as well as vertices and updating a vertex v can modify any data associated with v, v's incident edges, and the vertices adjacent to v. For 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 [Low et al. 2010, 2012] 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 [Geman and Geman 1984; Gelfand and Smith 1990], iterative graph coloring [Culberson 1992], and n-body problems such as the fluidanimate PARSEC benchmark [Bienia et al. 2008].

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 [Brin and Page 1998], loopy belief propagation [Murphy et al. 1999; Pearl 1988], coordinate descent [Dennis and Steihaug 1986], co-EM [Nigam and Ghani 2000], alternating least-squares [Hitchcock 1927], singular-value decomposition [Golub and Kahan 1965], and matrix factorization [Turing 1948].

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 $\mathrm{Adj}[v] = \{u \in V : (u,v) \in E\}$. The ***degree*** of v is thus $\deg(v) = |\mathrm{Adj}[v]|$, 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 an undirected graph with data associated with each vertex $v \in V$;
- $f : V \to 2^V$ is an ***update function***; and
- $Q_0 \subseteq V$ is the initial ***activation set***.

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

The advantage of dynamic over static data-graph computations is that they avoid performing many unnecessary updates. Studies in the literature [Low et al. 2010, 2012] 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 implementation. 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

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

```
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)
```

Fig. 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.

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 [Stoer et al. 2002]. For example, if $Q_0 = V$, we have $size(Q_0) = |V| + 2|E|$ by the handshaking lemma [Cormen et al. 2009, 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 [Cormen et al. 2009, Section 17.4], then line 14 executes in $\Theta(1)$ amortized time. Of course, a linked list would suffice to append operations in $\Theta(1)$ time, but would not allow for convenient subsequent parallel iteration over its elements. All other operations in the **for** loop on lines 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.

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*** [Feng and Leiserson 1997] (also called a ***general race*** [Netzer and Miller 1992]) 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 pro-

Benchmark	$\|V\|$	$\|E\|$	χ	RRLocks	Cilk+Locks	Prism	Prism-R
PR/G	916,428	5,105,040	43	15.5	14.3	9.7	12.6
PR/L	4,847,570	68,475,400	333	227.6	200.4	109.3	127.3
ID/2000	4,000,000	15,992,000	4	48.6	43.8	32.1	32.8
ID/4000	16,000,000	63,984,000	4	200.0	179.6	123.1	124.3
FBP/C1	87,831	265,204	2	8.7	8.9	6.9	7.0
FBP/C3	482,920	160,019	2	16.4	17.8	13.3	13.4
ALS/N	187,722	20,597,300	6	134.3	123.6	105.2	105.7

Fig. 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 runtimes of Prism and Prism-R include the time used to color the input graph. PR/G and PR/L run a PageRank algorithm on the web-Google [Leskovec et al. 2009] and soc-LiveJournal [Backstrom et al. 2006] graphs, respectively. ID/2000 and ID/4000 run an image denoise algorithm to remove Gaussian noise from 2D grayscale images of dimension 2000 by 2000 and 4000 by 4000. FBP/C1 and FBP/C3 perform belief propagation on a factor graph provided by the cora-1 and cora-3 datasets [Singla and Domingos 2006; McCallum 2012]. ALS/N runs an alternating least squares algorithm on the NPIC-500 dataset [Mitchell 2009].

duces 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 [Lee 2006; Bocchino et al. 2009]. Nondeterminism confounds debugging.

A known alternative to using locks is ***chromatic scheduling*** [Bertsekas and Tsitsiklis 1989; Adams and Ortega 1982; Low et al. 2012], which schedules a data-graph computation based on a coloring of the data-graph computation's ***conflict graph*** — a graph with an edge between two vertices if updating them in parallel would produce a race. For a simple data-graph computation, the conflict graph is simply the data graph itself with undirected edges. The idea behind chromatic scheduling is fairly simple. Chromatic scheduling begins by computing a ***(vertex) coloring*** of the conflict graph — an assignment of colors to the vertices such that no two adjacent vertices share the same color. Since no edge in the conflict graph connects two vertices of the same color, updates on all vertices of a given color can execute in parallel without producing races. To execute a round of a data-graph computation, the set of activated vertices Q is partitioned into χ ***color sets*** — subsets of Q containing vertices of a single color. Updates are applied to vertices in Q by serially stepping through each color set and updating all vertices within a color set in parallel. Indeed, the special case where the active set $Q == V$ is the entire graph (i.e., a static data-graph computation) can be executed using chromatic scheduling using Distributed GraphLab [Low et al. 2012]. 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. The data-graph computation is therefore executed deterministically, as long as a deterministic coloring algorithm is used to color the conflict graph. While chromatic scheduling potentially loses parallelism because colors are processed serially, we shall see that this concern does not appear to be an issue in practice.

To date, chromatic scheduling has been applied to static data-graph computations [Low et al. 2012], 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. By using the "multibag" and a deterministic coloring algorithm, PRISM guarantees to execute the data-graph computation deterministically.

We analyze the performance of PRISM using work-span analysis [Cormen et al. 2009, Ch. 27]. The ***work*** of a computation is the total number of instructions executed, and the ***span*** corresponds to the longest path of dependencies in the parallel program. We shall make the reasonable assumption that a single update $f(v)$ executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span.[2] 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 $O(size(Q_r)+P)$ work and $O(\chi(\lg(Q_r/\chi)+\lg\Delta)+\lg P)$ span.

The "price of determinism" incurred by using chromatic scheduling instead of the more common locking strategy appears to be negative for real-world applications. This discovery is perhaps surprising since it would seem to be strictly harder to guarantee that the computation behave deterministically than to allow for nondeterministic behaviors. Nevertheless, as Figure 2 indicates, on seven application benchmarks, PRISM executes 1.2–2.1 times faster than GraphLab's comparable, but nondeterministic, locking strategy, which we call RRLOCKS. This performance gap is not due solely to superior engineering or load balancing. A similar performance overhead is observed in a comparably engineered lock-based scheduling algorithm, CILK+LOCKS. PRISM outperforms CILK+LOCKS on each of the 7 application benchmarks and is on average (geometric mean) 1.4 times faster.

Our contribution is not a full-featured data-graph computation framework like GraphLab, Pregel, Galois, PowerGraph, Ligra, or GraphChi. Each of these systems is the result of countless hours of performance engineering and feature support. Instead, we provide a scheduling technique that could be adopted by any such framework to enable the deterministic execution of work-efficient, dynamic data-graph computations, which no existing framework currently supports[3] We use a modified shared-memory version of GraphLab in order to isolate the effect of our scheduling algorithms. Thus, the empirical comparisons in this paper are apples-to-apples comparisons of scheduling strategies, not competitive comparisons with other systems.

PRISM behaves deterministically as long as every update is ***pure***: it modifies no data except for that associated with its target vertex. This assumption precludes the update function from modifying global variables to aggregate or collect values. To support this common use pattern, we describe an extension to PRISM, called PRISM-R, which executes dynamic data-graph computations deterministically even when updates modify global variables using associative operations. PRISM-R replaces each multibag PRISM uses with a "multivector," maintaining color sets whose contents are ordered deterministically. PRISM-R executes in the same theoretical bounds as PRISM, but its implementation is more involved. Empirically PRISM-R is on average (geometric mean) only 1.07 times slower than PRISM and outperforms CILK+LOCKS on each of the seven application benchmarks.

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

[3]Deterministic Galois [Nguyen et al. 2014] has added support for deterministic execution of dynamic data-graph computations by recursively removing and executing independent sets of vertices. However, their algorithm is not work-efficient and, as a result, is much slower than the nondeterministic version.

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 [Intel 2013] implementation of PRISM and presents empirical results measuring this implementation's performance on seven application benchmarks. Section 7 describes PRISM-R which executes dynamic data-graph computations deterministically even when update functions modify global variables using associative operations. Section 8 describes and analyzes the multivector data structure PRISM-R uses to represent deterministically ordered color sets. Section 9 analyzes PRISM-R both theoretically, using work-span analysis, and empirically. Section 10 offers some concluding remarks.

2. BACKGROUND

We implemented the PRISM algorithm in Cilk Plus [Intel 2013], 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 [Frigo et al. 1998], Cilk++ [Leiserson 2010], Fortress [Allen et al. 2008], Habenero [Barik et al. 2009; Cavé et al. 2011], Hood [Blumofe and Papadopoulos 1999], Java Fork/Join Framework [Lea 2000], Task Parallel Library (TPL) [Leijen and Hall Leijen and Hall], Threading Building Blocks (TBB) [Reinders 2007], and X10 [Charles et al. 2005]. We review the dag 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 dag model of multithreading

The dag model of multithreading [Blumofe and Leiserson 1998, 1999] is described in tutorial fashion in [Cormen et al. 2009, 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 keywords [Cormen et al. 2009, 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.

An important property of this model is notion of the ***serial elision*** of a program. The serial elision is the serial program that results when the keywords **spawn** and **sync** are elided and the **parallel for** is replaced by an ordinary **for** . The model guarantees that the serial elision of a program always provides a correct implementation of the program. That is, the keywords indicate opportunities for parallelism, but they do not require parallel execution. In this sense, every program in this model has a ***serial semantics***.

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*** [Cormen et al. 2009, 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 [Brent 1974; Eager et al. 1989; Graham 1966] can execute a deterministic program with work T_1 and span T_∞ on P processors in time T_P satisfying

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

and a similar bound can be achieved by more practical "work-stealing" schedulers [Blumofe and Leiserson 1998, 1999]. 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 [Burton and Sleep 1981; Halstead 1984; Blumofe and Leiserson 1999], 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.

Worker-local storage

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

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 the multibag data structure.

Figure 3 shows the pseudocode 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 item v into bag C_k in Q. A multibag supports parallel MB-INSERT operations.
- MB-COLLECT(Q) produces a collection $\mathcal{C}$ that represents a list of the nonempty bags in Q, emptying Q in the process.

```
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              if CAS(active[u], FALSE, TRUE)
12                 MB-INSERT(Q, u, color[u])
13        r = r + 1

CAS(current, test, value)
14  begin atomic
15  if current == test
16     current = value
17     return TRUE
18  else
19     return FALSE
20  end atomic
```

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

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

PRISM calls COLOR-GRAPH on line 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 an ***optimal*** coloring of a graph [Garey et al. 1974] — a coloring that uses the smallest possible number of colors — an optimal coloring is not necessary for PRISM to perform well, as long as the data graph is colored deterministically, in parallel,[4] and with sufficiently few colors in practice. Many parallel coloring algorithms exist that satisfy the needs of PRISM (see, for example, [Alon et al. 1986; Linial 1992; Jones and Plassmann 1993; Goldberg et al. 1988; Hasenplaugh et al. 2014; Goldberg and Spencer 1989; Szegedy and Vishwanathan 1993; Kuhn and Wattenhofer 2006; Kuhn 2009; Barenboim and Elkin 2009]), however, our implementation of PRISM uses a multicore variant of the Jones and Plassmann algorithm [Jones and Plassmann 1993] that produces a deterministic $(\Delta+1)$-coloring of a degree-Δ graph $G = (V,E)$ in linear work and $O\left(\lg V + \lg\Delta \cdot \min\left\{\sqrt{E}, \Delta + \lg\Delta \lg V / \lg\lg V\right\}\right)$ span [Hasenplaugh et al. 2014].

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 4–13 executes the rounds of the data-graph computation. At the start of each round, line 5 collects the nonempty bags $\mathcal{C}$ from Q, which correspond to the nonempty color sets for the round. Lines 6–12 iterate through the color sets $C \in \mathcal{C}$ sequentially, and the **parallel for** loop on lines 7–12 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 10–12 insert into Q the vertices in S that have been activated.

Although a vertex u can be activated by multiple neighbors, it must only be updated at most once during a round. PRISM enforces this constraint[5] by using the atomic ***compare-and-swap*** operator [Herlihy and Shavit 2008, p. 480], which is available as a synchronization primitive on most machines and whose definition is given in lines 14–20. Lines 10–12 use the CAS primitive to activate each vertex $u \in S$ by atomically setting $active[u] =$ TRUE, and if $active[u]$ was previously FALSE, then calling MB-INSERT. Thus, each vertex is inserted into Q at most once during a round.

[4]If the data-graph computation performs sufficiently many updates, a serial $\Theta(V+E)$-work greedy coloring algorithm, such as that introduced by Welsh and Powell [Welsh and Powell 1967], can suffice as well, since the time to color the graph would be sufficiently amortized against the work performed.

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

Design considerations for the implementation of multibags

The theoretical performance of PRISM depends upon the properties of the multibag data structure. In particular, the multibag is carefully designed to ensure that PRISM is ***work-efficient*** — that is, it performs the same asymptotic work as the serial algorithm SERIAL-DDGC in Figure 1. Before examining the design of the multibag in Section 4, let us first explore why maintaining active color sets in PRISM in a work-efficient manner is tricky. Specifically, we shall consider two alternative strategies: bit vectors and an array of worker-local queues.

The bit-vector approach avoids the multibag altogether and simply manages activation sets using the bit vector *active* already used by PRISM. Recall that if $active[i]$ is TRUE, then the vertex $v_i \in V$ indexed by i is active. Suppose that *active* were the only data structure. To iterate over all activated vertices of color k, a **parallel for** could scan through *active*, updating the vertex v_i whenever $active[i]$ is TRUE and $color[i]$ is k. This scheme requires $\Omega(V\chi)$ work per round of the computation, where χ is the number of colors returned by COLOR-GRAPH in line 1 of Figure 3, since the entire bit vector must be scanned χ times each round. At the cost of additional preprocessing, *active* could be organized such that vertices of the same color are assigned contiguous indexes. Even with this optimization, however, scanning *active* requires $\Omega(V)$ work each round, which is not work-efficient for dynamic computations that activate only a sparse subset of the vertices each round.

An alternative strategy that one might consider is to represent the active color sets using an array of worker-local queues. A straightforward implementation of this approach, however, is also not work-efficient. For a dynamic data-graph computation using χ colors and P processors, a total of $P\chi$ worker-local queues would be needed to maintain the set of active vertices, and $\Omega(P\chi)$ work would be required to collect all nonempty queues. As we shall see in Section 4, however, by using a carefully designed data structure to manage worker-local queues, we can obtain a work-efficient data structure for maintaining color sets.

4. THE MULTIBAG DATA STRUCTURE

This section presents the multibag data structure employed by PRISM. The multibag uses worker-local sparse accumulators [Gilbert et al. 1992] 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 [Cormen et al. 2009, Ch. 27]. When used in a P-processor execution of a parallel program, a multibag Q of χ bags storing n elements supports MB-INSERT in $\Theta(1)$ worst-case time and MB-COLLECT in $O(n+\chi+P)$ work and $O(\lg n+\chi+\lg P)$ span. Such a multibag storing k elements uses $O(P\chi+k)$ space.

A ***sparse accumulator (SPA)*** [Gilbert et al. 1992] 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 k initialized elements of T takes $\Theta(k)$ work and $\Theta(\lg k)$ span.
- Emptying T takes $\Theta(1)$ work.

A multibag Q is an array of P worker-local SPA's, where P is the number of workers executing the program. We shall use p interchangeably to denote either a worker or that worker's unique identifier. Worker p's local SPA in Q is thus denoted by $Q[p]$. For a multibag Q of χ bags, each SPA $Q[p]$ contains an array $Q[p].array$ of size χ and a log $Q[p].log$. Figure 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 $C_{k,p}$.

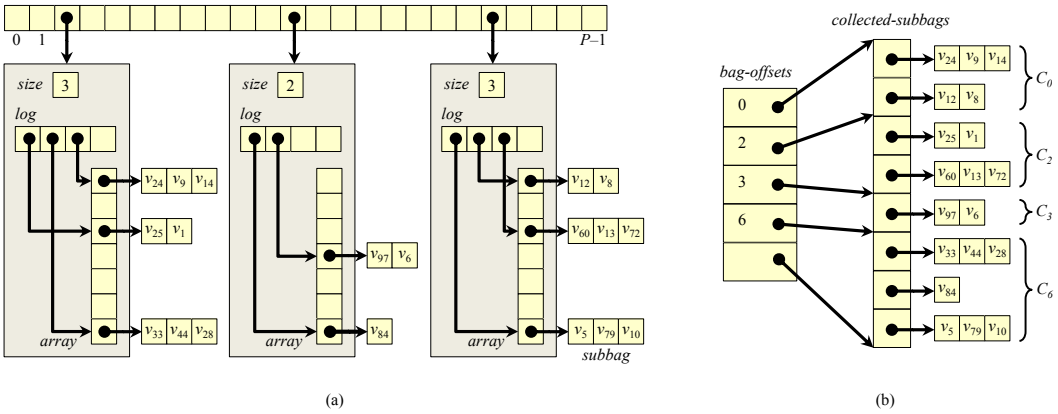

Fig. 4. A multibag data structure. **(a)** A multibag containing 19 elements distributed across 4 distinct bags: $\{C_0, C_2, C_3, C_6\}$, representing vertices of colors 0, 2, 3, and 6, respectively. Each worker keeps track of its portion of a particular bag, its ***subbag***, using a worker-local SPA,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

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)
```

Fig. 5. Pseudocode for the MB-INSERT multibag operation. MB-INSERT(Q, v, k) inserts the element v into the kth bag C_k of the multibag Q.

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 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 4.1. *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 [Brodnik et al. 1999]. 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 4.2. *In a P-processor parallel program execution, a call to* MB-COLLECT(Q) *on a multibag Q with χ bags whose contents are distributed across m distinct subbags executes in $O(m+\chi+P)$ work and $O(\lg m+\chi+\lg P)$ span.*

PROOF. We analyze each step of MB-COLLECT in turn. We shall use a helper procedure PREFIX-SUM(A), which computes the all-prefix sums of an array A of n integers in $\Theta(n)$ work and $\Theta(\lg n)$ span. (Blelloch [Blelloch 1990] 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 [Cole and Vishkin 1986; Blelloch et al. 1991; Zagha and Blelloch 1991] as follows:

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

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

Remark 4.3. Let Q be a multibag in a P-processor execution with m distinct subbags that represents bags whose indices lie in the range $[0,k]$. Then Q may be treated as a multibag representing k bags so that MB-COLLECT(Q) executes in $O(m+k+P)$ work and $O(\lg m+k+\lg P)$ span.

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

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

Fig. 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.

5. ANALYSIS OF PRISM

This section analyzes the performance of PRISM using work-span analysis [Cormen et al. 2009, Ch. 27]. We derive bounds on the work and span of PRISM for any simple data-graph computation $\langle G, f, Q_0\rangle$. Recall that we make the reasonable assumptions that a single update $f(v)$ executes in $\Theta(\deg(v))$ work and $\Theta(\lg(\deg(v)))$ span, and that the update only activates vertices in $\mathrm{Adj}[v]$. These work and span bounds can be used to characterize the data-graph computations on which PRISM achieves good parallel scalability. In particular, we show that on a data-graph on n vertices colored using χ colors that PRISM achieves good parallel speedup whenever the average work per round is much greater than $P\,\chi \lg n$

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

THEOREM 5.1. *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 $O(size(Q_r) + P)$ 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 6–12 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 4.1 implies that lines 11–12 execute in $\Theta(1)$ total work. The **parallel for** loop on lines 10–12 therefore executes in $\Theta(S)$ work and $\Theta(\lg S)$ span. Because $|S| \le \deg(v)$, the **parallel for** loop on lines 7–12 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 6–12 therefore executes in $\Theta(size(Q_r)+\chi)$ work and $O(\chi(\lg(Q_r/\chi)+\lg\Delta))$ span. Lemma 4.2 implies that line 5 executes MB-COLLECT in $O(Q_r + \chi_r + P)$ work and $O(\lg Q_r + \chi_r + \lg P)$ span where $\chi_r = \max_{v\in Q_r}\{color[v]\}$. Note that we take advantage here of the observation made in remark 4.3. The theorem follows since $|Q_r| + \chi_r \le size(Q_r) + 1$ □

Theoretical scalability of PRISM

Dynamic data-graph computations typically run for multiple rounds until a convergence criteria is met. We will now generalize Theorem 5.1 to prove work and span bounds for PRISM when executing a sequence of rounds.

THEOREM 5.2. *Suppose that* PRISM *colors a degree-Δ data graph $G = (V,E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0\rangle$ in r rounds applying updates to the activation sets $Q_0, Q_1, \ldots, Q_{r-1}$. Define the multiset $\mathcal{U} = \biguplus_{i=0}^{r-1} Q_i$ so that $|\mathcal{U}| = \sum_{i=0}^{r-1}|Q_i|$ and $size(\mathcal{U}) = \sum_{i=0}^{r-1} size(Q_i)$, where the symbol $\biguplus$ indicates a* ***multiset sum***.[6] *Then, on P processors,*

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

PRISM *executes the data-graph computation using* $O(size(\mathcal{U}) + rP)$ *work and* $O(r\,\chi(\lg((\mathcal{U}/r)/\chi) + \lg\Delta) + r\lg P)$ *span.*

PROOF. The work bound follows directly from Theorem 5.1 by taking the sum of work performed in each of the r rounds of PRISM. The total span of PRISM is equal to the sum of each round's span which by Theorem 5.1 is bounded by $\sum_{i=0}^{r-1}(\chi(\lg(Q_i/\chi) + \lg\Delta) + \lg P)$. Observing that $\sum_{i=0}^{r-1}\chi\lg(Q_i/\chi) \le r\,\chi\lg((\mathcal{U}/r)/\chi)$ completes the proof. □

Given Theorem 5.2 we can compute the parallelism of PRISM for a data-graph computation that applies a multiset $\mathcal{U}$ of updates over r rounds. The following corollary expresses the parallelism of PRISM in terms of the average size of the activation sets in a sequence of rounds.

COROLLARY 5.3. *Suppose* PRISM *executes a data-graph computation in r rounds during which it applies a multiset* $\mathcal{U}$ *of updates. Define the average number of updates per round* $U_{avg} = |\mathcal{U}|/r$ *and the average work per round* $W_{avg} = size(\mathcal{U})/r$. *Then* PRISM *has* $\Omega(W_{avg}/(\chi(\lg(U_{avg}/\chi) + \lg\Delta)))$ *parallelism.*

PROOF. Follows from Theorem 5.2 by computing the parallelism as the ratio of the work and span and then performing substitution. □

Corollary 5.3 implies that PRISM achieves near perfect linear parallel speedup on P processors for a graph of n vertices when the average work performed in each round $W_{avg} \gg P\,\chi\lg n$.

6. EMPIRICAL EVALUATION

This section explores the performance properties of PRISM from an empirical perspective. We describe three experiments designed to investigate the synchronization costs, dynamic-scheduling overheads, and scalability properties of PRISM. For the first experiment, on a suite of 12 benchmark graphs, PRISM executed between 1.0 and 2.1 times faster than a nondeterministic locking protocol on PageRank [Brin and Page 1998], exhibiting a geometric-mean speedup of a factor of 1.5, a substantial advantage in synchronization costs. The second experiment shows that the slowdown that PRISM incurs for dynamic scheduling using multibags, compared with static scheduling, is only about 1.16 when all vertices are activated in every round. This experiment shows that PRISM can be effective even for relatively densely activated graphs. The third experiment shows that PRISM scales well and is relatively insensitive to the number of colors needed to color the data graph, as long as there is sufficient parallelism.

Experimental setup

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

As a platform for our experiments, we implemented a new parallel execution engine within GraphLab [Low et al. 2010] that uses Intel Cilk Plus[7] [Intel 2013] to expose parallelism. The new execution engine and all of our scheduling algorithms were designed to be compatible with the original GraphLab API in order to facilitate a fair evaluation of the relative merits of different scheduling methodologies. In particular, to better understand the performance properties of PRISM, we developed four scheduling algorithms for comparison:

- SERIAL-DDGC is an implementation of the serial scheduling algorithm from Figure 1. SERIAL-DDGC provides a serial performance baseline for measuring the parallel speedup achieved by the other, more complex, scheduling algorithms for dynamic data-graph computations.

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

Graph	$\|V\|$	$\|E\|$	χ	CILK+LOCKS	PRISM	PRISM-R	*Coloring*
cage15	5,154,860	94,044,700	17	36.9	35.5	35.6	12%
liveJournal	4,847,570	68,475,400	333	36.8	21.7	22.3	12%
randLocalDim25	1,000,000	49,992,400	36	26.7	14.4	14.6	18%
randLocalDim4	1,000,000	41,817,000	47	19.5	12.5	13.7	14%
rmat2Million	2,097,120	39,912,600	72	22.5	16.6	16.8	12%
powerGraph2M	2,000,000	29,108,100	15	12.1	9.8	10.1	13%
3dgrid5m	5,000,210	15,000,600	6	10.3	10.3	10.4	7%
2dgrid5m	4,999,700	9,999,390	4	17.7	8.9	9.0	4%
web-Google	916,428	5,105,040	43	3.9	2.4	2.4	8%
web-BerkStan	685,231	7,600,600	200	3.9	2.4	2.7	8%
web-Stanford	281,904	2,312,500	62	1.9	0.9	1.0	11%
web-NotreDame	325,729	1,469,680	154	1.1	0.8	0.8	12%

Fig. 7. Performance of PRISM versus CILK+LOCKS when executing $10 \cdot |V|$ updates of the PageRank [Brin and Page 1998] 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," "PRISM," and "PRISM-R" present 12-core running times in seconds for each respective scheduler. Each running time is the median of 5 runs. Column "*Coloring*" gives the percentage of PRISM's running time spent coloring the graph. PRISM-R, discussed in Section 7, provides deterministic support for associative operations on global variables.

- CILK+LOCKS is a lock-based scheduling algorithm for dynamic data-graph computations. During each round, CILK+LOCKS updates only an active subset of the vertices in the graph. It uses a locking scheme to avoid executing conflicting updates in parallel. The locking scheme associates a shared-exclusive (i.e., reader-writer) lock [Courtois et al. 1971] 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 $u \in \text{Adj}[v]$. A global ordering of locks is used to avoid deadlock.
- RRLOCKS is the lock-based dynamic scheduling algorithm implemented by the round-robin ***sweep scheduler*** in the original shared-memory version of GraphLab. A bit vector *active* is used to represent the active set of vertices. During each round, RRLOCKS scans each vertex in the active set in a round-robin fashion, conditionally updating a vertex v_i if $active[i]$ is TRUE. To avoid races, a locking strategy is used to coordinate updates that conflict.
- RRCOLOR is a coloring-based dynamic scheduling algorithm that uses a bit vector *active* to represent the active set of vertices. Instead of using locks to coordinate conflicting updates, however, RRCOLOR uses a vertex-coloring of the graph. At the start of the computation, RRCOLOR partitions the vertices by color and stores them in static arrays. For a graph colored using χ colors, each round of the computation is divided into χ color steps. During the kth color step, RRCOLOR scans all color-k vertices and conditionally updates a color-kd vertex v_i if $active[i]$ is TRUE.

Overheads for locking and for chromatic scheduling

We compared the overheads associated with coordinating conflicting updates of a dynamic data-graph computation using locks versus using chromatic scheduling. We evaluated these overheads by comparing the 12-core execution times for PRISM and CILK+LOCKS to execute the PageRank [Brin and Page 1998] data-graph computation on a suite of graphs. We used PageRank for this study because of its comparatively cheap update function, which makes overheads due to scheduling more pronounced. PageRank updates a vertex v by first scanning v's incoming edges to aggregate the data from its incoming neighbors, and then by scanning v's outgoing edges to activate its outgoing neighbors.

We executed the PageRank application on a suite of six synthetic and six real-world graphs. The six real-world graphs came from the Stanford Large Network Dataset Collection (SNAP) [Leskovec 2013], and the University of Florida Sparse Matrix Collection [Davis and Hu 2011]. The six synthetic graphs were generated using the "randLocal," "powerLaw," "gridGraph," and "rMatGraph"

Benchmark	χ	*Updates*	RRLocks	RRColor	Prism	Prism-R
PR/L	333	48,475,700	35.25	14.5	17.7	18.4
ID/2000	4	40,000,000	63.15	50.1	59.2	59.9
FBP/C3	2	16,001,900	11.9	8.8	8.8	8.9
ID/1000	4	10,000,000	15.7	12.6	14.9	15.0
PR/G	43	9,164,280	3.1	1.3	2.1	2.2
FBP/C1	2	8,783,100	5.9	4.7	4.8	4.8
ALS/N	6	1,877,220	65.7	52.4	52.8	53.5

Fig. 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 "RRLocks," "RRColor," "Prism," and "Prism-R" list the 12-core running times in seconds for the respective schedulers to execute each benchmark. Each running time is the median of 5 runs. The Prism-R algorithm, which provides deterministic support for associative operations on global variables, will be discussed in Section 7.

generators included in the Problem Based Benchmark Suite [Shun et al. 2012]. We chose the graphs in this suite to be large enough to stress the memory system and thus make parallel speedup comparatively difficult. That is, given the random access inherent in data-graph computations, we expect most references to vertex data to come from DRAM, making DRAM bandwidth a scarce shared commodity. Since the span of Prism is superconstant, however, for a fixed number of workers, increasing the size of the graph only increases parallelism, making good parallel speedup comparatively easy. Thus, we have pessimistically chosen the graphs in the suite to be large enough to make DRAM bandwidth a shared bottleneck but not unduly larger.

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

Figure 7 presents the empirical results for this study. Figure 7 shows that over the 12 benchmark graphs, Prism executes between 1.0 and 2.1 times faster than Cilk+Locks on PageRank, exhibiting a geometric-mean speedup of a factor of 1.5. Moreover, from Figure 7 we see that an average of 10.9% of Prism's total running time is spent coloring the data graph, which is approximately equal to the cost of executing $|V|$ updates. Prism colors the data-graph once to execute the data-graph computation, however, meaning that its cost can be amortized over all of the updates in the data-graph computation. By contrast, the locking scheme implemented by Cilk+Locks incurs overhead for every update. Before updating a vertex v, Cilk+Locks acquires each lock associated with v and every vertex $u \in \text{Adj}[v]$. 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, RRColor, and RRLocks 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. In particular, because Prism typically executes fewer updates than a static data-graph computation scheduler, we modified the update functions for each application so that every update on a vertex v always activates all vertices $u \in \text{Adj}[v]$. This modification guarantees that Prism executes the same set of updates each round as RRLocks and RRColor, while still incurring the overhead that Prism requires in order to maintain a dynamic set of active vertices. Thus, we compare the worst case conditions for Prism with respect to scheduling overhead with the best case conditions for RRLocks and RRColor.

Figure 8 presents the results of these tests, revealing that the overhead Prism incurs to maintain its activation sets using a multibag. As can be seen from the figure, Prism is 1.0 to 1.6 times slower than RRColor on the benchmarks with a geometric-mean relative slowdown of 1.16. That is, for static data-graph computations, Prism incurs only an aggregate 16% slowdown through the use of

a multibag, as opposed to the simple array used by RRCOLOR, which suffices for static scheduling. The PRISM algorithm, which can also support *dynamic* activation sets efficiently, incurred minimal overhead for the multibag data structure. PRISM outperformed RRLOCKS on all benchmarks, achieving a geometric-mean speedup of 30% due to RRLOCKS's lock overhead. Thus, PRISM incurs relatively little overhead by maintaining activation sets with multibags.

The relative overhead of RRCOLOR and PRISM depends on the percentage of vertices active during a given round. As a typical example, RRCOLOR is approximately 1.09 times faster than PRISM on the image denoise benchmark when 80% of the vertices are active each round, but is 1.11 times slower when 5% or less of the vertices are active each round. As part of an effort to incorporate the PRISM scheduling paradigm into an existing data-graph computation framework (e.g., GraphLab, Pregel etc.), one might consider using a heuristic to switch between the use of a bitvector and a multibag depending on the density of the activation set. A simple heuristic such as a fixed threshold on the relative density of the activation set[8] (e.g., 10% of the vertices) would likely suffice to maintain activation sets with good performance: if fewer than 10% of vertices are active, use a multibag, otherwise use a bitvector.

Scalability of PRISM

To measure the scalability of PRISM, and CILK+LOCKS, we compared their 12-core runtimes to the serial reference implementation SERIAL-DDGC. Figure 9 shows the empirical 12-core speedups relative to SERIAL-DDGC of PRISM and CILK+LOCKS on seven application benchmarks. Data for PRISM-R is also included, which will be discussed in Section 9. In geometric mean, CILK+LOCKS achieved 5.73 times speedup, PRISM achieved 7.56 times speedup, and PRISM-R achieved 7.42 times speedup.

In order to study the effect of the number χ of colors used to color the application's data graph on the parallel scalability of PRISM, we measured the parallelism T_1/T_∞ and the 12-core speedup T_1/T_{12} of PRISM while executing the image-denoise application as we varied the number of colors used. The image-denoise application performs belief propagation to remove Gaussian noise added to a gray-scale image. The data graph for the image-denoise application is a two-dimensional grid in which each vertex represents a pixel, and there is an edge between any two adjacent pixels. The COLOR-GRAPH procedure invoked in line 1 of Figure 3 typically colors this data-graph with just 4 colors.

To perform this study, we artificially increased χ by repeatedly taking a random nonempty subset of the largest set of vertices with the same color and assigning a new color to those vertices. Using this technique, we ran the image-denoise application on a 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 10 plots the results of these tests. Although the parallelism of PRISM is inversely proportional to χ, PRISM's speedup on 12 cores is relatively insensitive to χ, as long as the parallelism is greater than about 120. This result is consistent with the rule of thumb that a program with at least $10P$ parallelism should achieve nearly perfect linear speedup on P processors [Cormen et al. 2009, p. 783].

7. THE PRISM-R ALGORITHM

This section introduces PRISM-R, a chromatic-scheduling algorithm that executes a dynamic data-graph computation deterministically even when updates modify global ***reducer variables*** using associative operations such as a reducer hyperobject [Frigo et al. 2009]. While the chromatic scheduling technique employed by PRISM ensures that there are no data races on the vertex data of the graph, the order in which updates are made to a reducer variable among vertices of a common color can yield a nondeterministic result to the final reducer variable value. The multivector data structure, which is a theoretical improvement to the multibag, is used by PRISM-R to maintain activation sets

[8] A similar heuristic was shown to be effective in the graph computation library Ligra [Shun and Blelloch 2013] for adaptively switching between "dense" and "sparse" representations of vertex subsets.

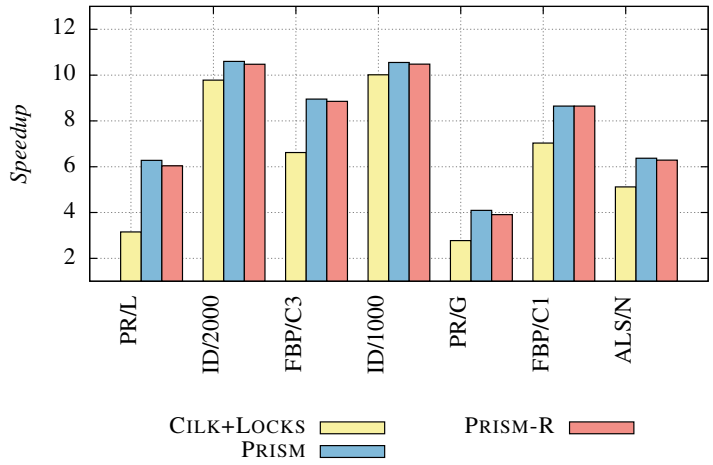

Fig. 9. Empirical speedup relative to SERIAL-DDGC on 12 processor cores. Shown are the empirical speedups T_s/T_{12} of CILK+LOCKS, PRISM, and PRISM-R, where T_s is the runtime of the serial scheduling algorithm SERIAL-DDGC and T_{12} is the runtime of the particular algorithm on 12 cores. The PRISM-R algorithm is discussed in Section 7.

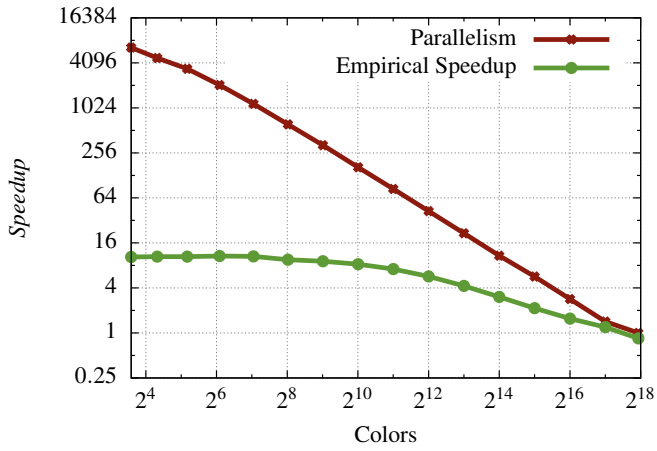

Fig. 10. Scalability of PRISM on the image-denoise application as a function of χ, the number of colors used to color the data graph. The parallelism T_1/T_∞ is plotted together with the empirical speedup T_1/T_{12} achieved on a 12-core execution. Parallelism values were measured using the Cilkview scalability analyzer [He et al. 2010].

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.

```
PRISM-R(G, f, Q_0)
 1  χ = COLOR-GRAPH(G)
 2  r = 0
 3  updates = 0
 4  Q = Q_0
 5  while Q ≠ ∅
 6     C = MV-COLLECT(Q)
 7     for C ∈ C
 8        parallel for i = 1,2,...,|C|
 9           ⟨v, p⟩ = C[i]
10           if p == priority[v]
11              rank[f(v)] = updates + i
12              priority[v] = ∞
13              S = f(v)
14              parallel for u ∈ S
15                 if PRIORITYWRITE(priority[u], rank[f(v)])
16                    MV-INSERT(Q, ⟨u, rank[f(v)]⟩, color[u])
17           updates = updates + |C|
18        r = r + 1

PRIORITYWRITE(current, value)
19  begin atomic
20  if current > value
21     current = value
22     return TRUE
23  else
24     return FALSE
25  end atomic
```

Fig. 11. Pseudocode for PRISM-R. The algorithm takes as input a data graph G, an update function f, and an initial activation set Q_0. COLOR-GRAPH colors a given graph and returns the number of colors it used. The procedures MV-COLLECT and MV-INSERT operate the multivector Q to maintain activation sets for PRISM-R. PRISM-R updates the value of *updates* after processing each color set and r after each round of the data-graph computation.

Data-graph computations with global reductions

Several frameworks for executing data-graph computations allow updates to modify global variables in limited ways. Pregel aggregators [Malewicz et al. 2010], and GraphLab's sync mechanism [Low et al. 2010], 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*** [Iverson 1962; Lasser and Omohundro 1986; Blelloch 1992; Chamberlain et al. 2000; Koelbel et al. 1994; Reinders 2007; Intel 2012; McGrady 2008], 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)*** [Frigo et al. 2009; Lee et al. 2012] 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 updates modify global variables using a noncommutative reducer. The reason for this is, in part, that the order of vertices within in a multibag depends on how the computation was scheduled among participating workers. As a result, the order in which lines 7–12 of PRISM in Figure 3 evaluates the vertices in a color set C is nondeterministic. If two updates on vertices in C modify the same reducer, then the relative order of these modifications can differ between runs of PRISM, even if a single worker happens to execute both updates.

PRISM-R

PRISM-R is an extension to PRISM that executes dynamic data-graph computations deterministically even when update functions are allowed to perform associative operations on global variables.

The semantics of PRISM-R mimic that of SERIAL-DDGC when its queue of active vertices is stable sorted by color at the start of each round. In this modified version of SERIAL-DDGC updates to active vertices of the same color are applied in increasing order of their insertion into the queue. PRISM-R guarantees that the result of associative reductions performed by update functions reflect this same order.

Figure 11 shows the pseudocode for PRISM-R which differs from PRISM in its use of alternate data structure to maintain partitioned activation sets and in its use of a priority deduplication strategy for avoiding multiple updates to the same vertex in a round.

A ***multivector*** is used by PRISM-R to represent a list of χ ***vectors*** (ordered multisets). It supports the operations MV-INSERT and MV-COLLECT, which are analogous to the multibag operations MB-INSERT and MB-COLLECT, respectively. Each vector maintained by a multivector has serial semantics, meaning that the order of elements within each vector is deterministic and equivalent to the insertion order in an execution of the serial elision of the parallel program. Section 8 describes and analyzes the implementation of the multivector data structure.

The serial semantics of the multivector are not alone sufficient to ensure that updates are ordered deterministically in an execution of the serial elision of the program. Consider, for example, a round of PRISM that updates the three vertices x, y, z in parallel. Suppose that y activates u and both x and z activate a common neighbor v. The atomic compare-and-swap operator used by PRISM on line 11 of Figure 3 ensures that x and z will not both insert v into the activation set, but which of the two succeeds is nondeterministic. Inserting these two activated vertices into a multivector would produce either the order u, v or v, u depending on whether x or z activated v.

To eliminate this source of nondeterminism, PRISM-R assigns each update $f(v)$ a unique integer $rank[f(v)]$ on line 11 of Figure 11 that orders updates applied during a round according to their execution order in an execution of the serial elision of PRISM-R. Instead of maintaining a bit vector denoting whether or not a vertex is active PRISM-R maintains an integer array *priority* of priorities. For each active vertex v the value $priority[v]$ is equal to the smallest rank of any update $f(u)$ that activated v in the previous round. The priority of a vertex v is reset on line 12 before applying $f(v)$ by setting $priority[v] = \infty$.

For each vertex $u \in \text{Adj}[v]$ activated by update $f(v)$, PRISM-R uses an atomic ***priority-write*** operator [Shun et al. 2013] to set $priority[u] = \min\{priority[u], rank[f(v)]\}$ and inserts the vertex-priority pair $\langle u, rank[f(v)]\rangle$ into the multivector if the priority write is successful on line 15. The color sets returned by MV-COLLECT on line 6 can contain multiple vertex-priority pairs for each active vertex. On lines 8–16 PRISM-R iterates over the vertex-priority pairs $\langle v, p\rangle$ in a color set and only applies the update $f(v)$ if $priority[v] == p$. Since $priority[v]$ is equal to the lowest ranked update that activated v, PRISM-R updates each active vertex exactly once during a round in the same order as a serial execution.

8. THE MULTIVECTOR DATA STRUCTURE

This section introduces the multivector data structure, which provides a theoretical improvement to the multibag. The multivector data structure maintains several vectors (dynamic arrays), each supporting a parallel append operation. Each vector has serial semantics, that is, the order of elements within any vector is equivalent to their insertion order in an execution of the serial elision of the Cilk parallel program. The multivector can be used in place of the multibag to provide a stronger encapsulation of nondeterminism in programs whose behavior depends on the ordering of elements in each set. This section assumes familiarity with the Cilk execution model [Frigo et al. 1998], as well as its implementation of reducers [Frigo et al. 2009].

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

```
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
```

Fig. 12. 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()
 7  L = new log-tree node
 8  L.sublog = new vector
 9  L.size = 1
10  L.left = NIL
11  L.right = NIL
12  return L
```

```
REDUCE(L_l,L_r)
13  L = IDENTITY()
14  L.size = L_l.size+L_r.size+1
15  L.left = L_l
16  L.right = L_r
17  return L
```

Fig. 13. Pseudocode for the IDENTITY and REDUCE log-tree reducer operations. The IDENTITY operation creates and returns a new log-tree node L. The REDUCE(L_l, L_r) operation concatenates a left log-tree node L_l with a right log-tree node L_r.

```
A(R)
1  LOG-INSERT(R,e_1)
2  spawn B(R)
3  LOG-INSERT(R,e_7)
4  sync
5  LOG-INSERT(R,e_8)

B(R)
6  LOG-INSERT(R,e_2)
7  spawn LOG-INSERT(R,e_3)
8  LOG-INSERT(R,e_4)
9  LOG-INSERT(R,e_5)
10 sync
11 LOG-INSERT(R,e_6)

LOG-INSERT(R,e)
12 L = GET-LOCAL-VIEW(R)
13 APPEND(L.subblog,e)
```

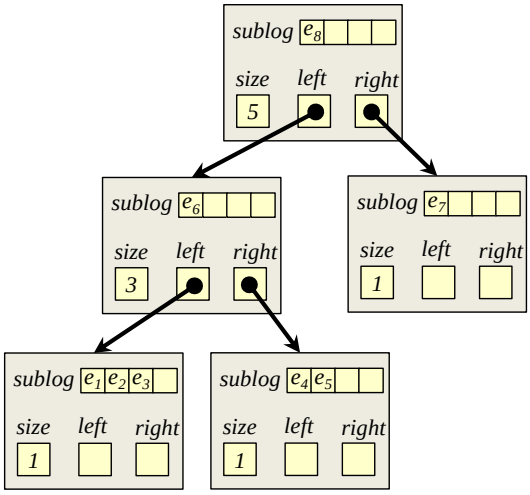

Fig. 14. The state of a log-tree reducer R after a work-stealing execution of A(R). Steals occur on line 2 of A and line 8 of B partitioning the execution into 5 traces. The ordered multiset $(e_1, e_2, \ldots, e_8)$ is represented by 5 trace-local sublogs ordered according to a post-order traversal of the log tree.

would be visited if an execution of the serial elision of the underlying Cilk program were executed, which corresponds to a left-to-right depth-first execution of the dag.

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

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

Fig. 15. 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.

The log-tree reducer

A multivector stores its nonempty trace-local subvectors in a ***log tree***, which represents an ordered multiset of elements and supports $\Theta(1)$-work append operations. A log tree is a binary tree in which each node L stores a dynamic array $L.$*sublog*. The ordered multiset that a log tree represents corresponds to a concatenation of the tree's dynamic arrays in a post-order tree traversal. Each log-tree node L is augmented with the size of its subtree $L.$*size* counting the number of log-tree nodes in the subtree rooted at L. Using this augmentation, the operation FLATTEN($L, A, L.size - 1$) described in Figure 12 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 13 presents the pseudocode for the IDENTITY and REDUCE operations for the log-tree reducer.

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

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

Figure 14 illustrates the state of a log-tree reducer R following the execution of a fork-join parallel function $A(R)$. Steals occur on line 2 of A and line 8 of B. The log-tree reducer partitions this execution of $A(R)$ into 5 traces each of which corresponds to one node in the tree. The first trace corresponds to the worker that begins the execution of $A(R)$ and each steal creates two additional traces: one corresponding to the stolen continuation of the spawned function, and another corresponding to the portion of the program following the associated **sync** statement.

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

Let us see how MV-INSERT and MV-COLLECT are implemented.

Figure 16 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 19–20 additionally append that subvector to $Q.$*log-reducer*, thereby maintaining the log-tree reducer. Second, lines 15–16 reset the contents of the SPA $Q[p]$ after worker p begins executing a new trace, thereby ensuring that $Q[p]$ stores only trace-local subvectors.

```
MV-INSERT(Q, v, k)
14  p = GET-WORKER-ID()
15  if worker p began a new trace since last insert
16      reset Q[p]
17  if Q[p].array[k] == NIL
18      Q[p].array[k] = newsubvector
19      L = GET-LOCAL-VIEW(Q.log-reducer)
20      APPEND(L.sublog, Q[p].array[k])
21  APPEND(Q[p].array[k], v)
```

Fig. 16. Pseudocode for the MV-INSERT multivector operation. MV-INSERT(Q,v,k) inserts an element v into the kth vector C_k maintained by the multivector Q.

Figure 15 sketches the MV-COLLECT operation, which returns a pair $\langle$*subvector-offsets*, *collected-subvectors*$\rangle$ analogous to the return value of MB-COLLECT. The procedure MV-COLLECT differs from MB-COLLECT primarily in that Step 1, which replaces Steps 1 and 2 in MB-COLLECT, flattens the log tree underlying Q.*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 4.2 for MB-COLLECT is a suitable stable sort 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 8.1. *Executing* MV-INSERT *takes* $\Theta(1)$ *time in the worst case.*

PROOF. Resetting the SPA $Q[p]$ on line 16 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 4.1. □

Lemma 8.2 bounds the work and span of MV-COLLECT.

LEMMA 8.2. *Consider a computation A with span $T_\infty(A)$, and suppose that the contents of a multivector Q of χ vectors are distributed across m subvectors. Then a call to* MV-COLLECT(Q) *incurs $\Theta(m+\chi)$ work and $\Theta(\lg m+\chi+T_\infty(A))$ span.*

PROOF. Flattening the log-tree reducer in Step 1 is accomplished in two steps. First, the FLATTEN operation writes the nodes of the log tree to a contiguous array. Execution of FLATTEN has span proportional to the depth of the log tree, which is bounded by $O(T_\infty(A))$, since at most $O(T_\infty(A))$ reduction operations can occur along any path in A, and REDUCE for log trees executes in $\Theta(1)$ work [Frigo et al. 2009]. 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 4.2, execute in $\Theta(\chi+\lg m)$ span. □

9. ANALYSIS AND EVALUATION OF PRISM-R

This section presents a theoretical work-span analysis of PRISM-R, demonstrating that its work and span are asymptotically equivalent to PRISM. This section also discusses PRISM-R's empirical performance relative to PRISM, which was evaluated in Section 6. In particular, PRISM-R is only 2-7% slower than PRISM, overall, while providing deterministic support for associative operations on global variables.

Work-span analysis of Prism-R

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

THEOREM 9.1. *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 $O(size(Q_r))$ work and $O(\chi(\lg(Q_r/\chi) + \lg\Delta))$ span.*

PROOF. PRISM-R can perform a priority write to its *active* array with $\Theta(1)$ work, and it can remove duplicates from the output of MV-COLLECT in $O(size(Q_r))$ work and $O(\lg(size(Q_r))) = O(\lg Q_r + \lg\Delta)$ span. The theorem follows by applying Lemmas 8.1 and 8.2 appropriately to the analysis of PRISM in Theorem 5.1. □

THEOREM 9.2. *Suppose that* PRISM-R *colors a degree-Δ data graph $G = (V,E)$ using χ colors, and then executes the data-graph computation $\langle G, f, Q_0 \rangle$ in r rounds applying updates to the activation sets $Q_0, Q_1, \ldots, Q_{r-1}$. Define the multiset $\mathcal{U} = \biguplus_{i=0}^{r-1} Q_i$ so that $|\mathcal{U}| = \sum_{i=0}^{r-1} |Q_i|$ and $size(\mathcal{U}) = \sum_{i=0}^{r-1} size(Q_i)$. Then* PRISM-R *executes the data-graph computation using $O(size(\mathcal{U}))$ work and $O(r \cdot \chi(\lg((\mathcal{U}/r)/\chi) + \lg\Delta))$ span.*

PROOF. By Theorem 9.1 PRISM-R executes a round of a data-graph computation using the same asymptotic work and span as PRISM. We mirror the arguments in Theorem 5.2 to bound the work and span of PRISM-R for a sequence of rounds. □

Given Theorem 9.2 we can compute the parallelism of PRISM-R for a data-graph computation that applies a multiset $\mathcal{U}$ of updates over r rounds. The following corollary expresses the parallelism of PRISM-R in terms of the average size of the activation sets in a sequence of rounds.

COROLLARY 9.3. *Suppose* PRISM-R *executes a data-graph computation in r rounds during which it applies a multiset $\mathcal{U}$ of updates. Define the average number of updates per round $U_{avg} = |\mathcal{U}|/r$ and the average work per round $W_{avg} = size(\mathcal{U})/r$. Then* PRISM-R *has $\Omega(W_{avg}/(\chi(\lg(U_{avg}/\chi) + \lg\Delta)))$ parallelism.*

PROOF. Follows from Theorem 9.2 by computing the parallelism as the ratio of the work and span and then performing substitution. □

***Empirical evaluation of* PRISM-R**

PRISM-R provides deterministic support for associative operations on global variables at the cost of additional complexity versus PRISM, specifically in the maintenance of activation sets. Nonetheless, PRISM-R guarantees the same asymptotic work and span as PRISM. Empirically, we find that PRISM-R suffers a geomean slowdown of only 2-7% versus PRISM in various scenarios. In particular, the 12-core performance for each dynamic data-graph computation application featured in Figure 2 demonstrate that for real-world applications PRISM-R is 7% slower in geometric mean than PRISM. in Figure 8 we see that PRISM-R is only 1.8% slower than PRISM for static versions of the applications featured in Figure 2 (i.e., all vertices are updated every round). Finally, in Figure 7 we present the 12-core performance of PRISM-R on PageRank [Brin and Page 1998] for a suite of six synthetic and six real-world graphs. In this case, PRISM-R is 3.5% slower in geometric mean than PRISM.

10. CONCLUSION

Researchers over multiple decades have soberly advised the rest of the community that the difficulty of parallel programming can be greatly reduced by using some form of deterministic parallelism [Patil 1970; Halstead 1985; Gibbons 1989; Steele 1990; Blelloch 1996; Feng and Leiserson

1997, 1999; Devietti et al. 2009, 2011; Hower et al. 2011; Bergan et al. 2010; Berger et al. 2009; Olszewski et al. 2009; Yu and Narayanasamy 2009; Bocchino et al. 2009]. With a deterministic parallel program, the programmer observes no logical concurrency, that is, no nondeterminacy in the behavior of the program due to the relative and nondeterministic timing of communicating processes (e.g., when two processes try to acquire a lock simultaneously). The semantics of a *deterministic* parallel program are therefore serial and reasoning about such a program's correctness is theoretically no harder than reasoning about the correctness of a serial program, which is already sufficiently hard for most people. Testing, debugging, and formal verification is simplified by determinism, because there is no need to consider all possible relative timings (i.e., interleavings) of operations on shared mutable state.

The behavior of PRISM corresponds to a variant of SERIAL-DDGC that sorts the activated vertices in its queue by color at the start of each round. Whether PRISM executes a given data graph on 1 processor or many, it always behaves the same way. With PRISM-R, this property holds even when the update function can perform reductions (e.g., associative operators on global variables). By contrast, lock-based schedulers provide no such a guarantee of determinism. Instead, updates in a round executed by a lock-based scheduler appear to execute according to some linear order, the so-called *sequential consistency* model employed by GraphLab [Low et al. 2010, 2012] and others. This order is nondeterministic due to races on the acquisition of locks.

Blelloch, Fineman, Gibbons, and Shun [Blelloch et al. 2012] 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.

11. ACKNOWLEDGMENTS

Thanks to Guy Blelloch of Carnegie Mellon University for sharing utility functions from his Problem Based Benchmark Suite [Shun et al. 2012]. 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.

REFERENCES

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

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. 2008. *The Fortress Language Specification Version 1.0*. Technical Report. Sun Microsystems.

Noga Alon, László Babai, and Alon Itai. 1986. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms* 7, 4 (Dec. 1986), 567–583. DOI:http://dx.doi.org/10.1016/0196-6774(86)90019-2

Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*. ACM, New York, NY, USA, 44–54. DOI:http://dx.doi.org/10.1145/1150402.1150412

Leonid Barenboim and Michael Elkin. 2009. Distributed (Δ+1)-coloring in Linear (in Δ) Time. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC '09)*. ACM, New York, NY, USA, 111–120. DOI:http://dx.doi.org/10.1145/1536414.1536432

Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşirlar, Yonghong Yan, Yisheng Zhao, and Vivek

Sarkar. 2009. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 735–736. DOI:http://dx.doi.org/10.1145/1639950.1639989

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. *SIGPLAN Not.* 45, 3 (March 2010), 53–64. DOI:http://dx.doi.org/10.1145/1735971.1736029

Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe Multithreaded Programming for C/C++. *SIGPLAN Not.* 44, 10 (Oct. 2009), 81–96. DOI:http://dx.doi.org/10.1145/1639949.1640096

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

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. DOI:http://dx.doi.org/10.1145/1454115.1454128

Guy E. Blelloch. 1990. *Prefix Sums and Their Applications*. Technical Report. Carnegie Mellon University, Pittsburgh, PA, USA.

Guy E. Blelloch. 1992. *NESL: A Nested Data-Parallel Language*. Technical Report. Carnegie Mellon University, Pittsburgh, PA, USA.

Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (March 1996), 85–97. DOI:http://dx.doi.org/10.1145/227234.227246

Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally Deterministic Parallel Algorithms Can Be Fast. *SIGPLAN Not.* 47, 8 (Feb. 2012), 181–192. DOI:http://dx.doi.org/10.1145/2370036.2145840

Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '91)*. ACM, New York, NY, USA, 3–16. DOI:http://dx.doi.org/10.1145/113379.113380

Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. Comput.* 27, 1 (Feb. 1998), 202–229. DOI:http://dx.doi.org/10.1137/S0097539793259471

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. DOI:http://dx.doi.org/10.1145/324133.324234

Robert D. Blumofe and Dionisios Papadopoulos. 1999. *Hood: A user-level threads library for multiprogrammed multiprocessors*. Technical Report. University of Texas at Austin.

Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel Programming Must Be Deterministic by Default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar'09)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1855591.1855595

Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206. DOI:http://dx.doi.org/10.1145/321812.321815

Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.* 30, 1-7 (April 1998), 107–117. DOI:http://dx.doi.org/10.1016/S0169-7552(98)00110-X

Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. 1999. Resizable Arrays in Optimal Time and Space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures (WADS '99)*. Springer-Verlag, London, UK, UK, 37–48. http://dl.acm.org/citation.cfm?id=645932.673194

F. Warren Burton and M. Ronan Sleep. 1981. Executing Functional Programs on a Virtual Tree of Processors. In *Proceedings of the 1981 Conference on Functional Programming*

Languages and Computer Architecture (FPCA '81). ACM, New York, NY, USA, 187–194. DOI: http://dx.doi.org/10.1145/800223.806778

Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. ACM, New York, NY, USA, 51–61. DOI: http://dx.doi.org/10.1145/2093157.2093165

Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. 2000. ZPL: A Machine Independent Programming Language for Parallel Computers. *IEEE Trans. Softw. Eng.* 26, 3 (March 2000), 197–211. DOI: http://dx.doi.org/10.1109/32.842947

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 519–538. DOI: http://dx.doi.org/10.1145/1094811.1094852

R Cole and U Vishkin. 1986. Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (STOC '86)*. ACM, New York, NY, USA, 206–219. DOI: http://dx.doi.org/10.1145/12130.12151

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

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

Joseph C. Culberson. 1992. *Iterated greedy graph coloring and the difficulty landscape*. Technical Report. University of Alberta.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. DOI: http://dx.doi.org/10.1145/2049662.2049663

J E Dennis, Jr. and Trond Steihaug. 1986. On the Successive Projections Approach to Least-squares Problems. *SIAM J. Numer. Anal.* 23, 4 (Aug. 1986), 717–733. DOI: http://dx.doi.org/10.1137/0723047

Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. *SIGPLAN Not.* 44, 3 (March 2009), 85–96. DOI: http://dx.doi.org/10.1145/1508284.1508255

Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: A Relaxed Consistency Deterministic Computer. *SIGPLAN Not.* 47, 4 (March 2011), 67–78. DOI: http://dx.doi.org/10.1145/2248487.1950376

D. L. Eager, J. Zahorjan, and E. D. Lozowska. 1989. Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. Comput.* 38, 3 (March 1989), 408–423. DOI: http://dx.doi.org/10.1109/12.21127

Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*. ACM, New York, NY, USA, 1–11. DOI: http://dx.doi.org/10.1145/258492.258493

M. Feng and C. E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.

Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and Other Cilk++ Hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 79–90. DOI: http://dx.doi.org/10.1145/1583991.1584017

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. *SIGPLAN Not.* 33, 5 (May 1998), 212–223. DOI: http://dx.doi.org/10.1145/277652.277725

M. R. Garey, D. S. Johnson, and L. Stockmeyer. 1974. Some Simplified NP-complete Problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing (STOC '74)*. ACM, New York, NY, USA, 47–63. DOI: http://dx.doi.org/10.1145/800119.803884

Alan E. Gelfand and Adrian F. M. Smith. 1990. Sampling-Based Approaches to Calculating Marginal Densities. *J. Amer. Statist. Assoc.* 85, 410 (June 1990), 398–409.

Stuart Geman and Donald Geman. 1984. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Trans. Pattern Anal. Mach. Intell.* 6, 6 (Nov. 1984), 721–741. DOI: http://dx.doi.org/10.1109/TPAMI.1984.4767596

P. B. Gibbons. 1989. A More Practical PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '89)*. ACM, New York, NY, USA, 158–168. DOI: http://dx.doi.org/10.1145/72935.72953

John R. Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse Matrices in Matlab: Design and Implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (Jan. 1992), 333–356. DOI: http://dx.doi.org/10.1137/0613024

Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. 1988. Parallel Symmetry-breaking in Sparse Graphs. *SIAM J. Discret. Math.* 1, 4 (Oct. 1988), 434–446. DOI: http://dx.doi.org/10.1137/0401044

Mark Goldberg and Thomas Spencer. 1989. A New Parallel Algorithm for the Maximal Independent Set Problem. *SIAM J. Comput.* 18, 2 (April 1989), 419–427. DOI: http://dx.doi.org/10.1137/0218029

G. Golub and W. Kahan. 1965. Calculating the Singular Values and Pseudo-Inverse of a Matrix. *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis* 2, 2 (1965), 205–224. DOI: http://dx.doi.org/10.1137/0702016

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883

R. L. Graham. 1966. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal* 45 (1966), 1563–1581.

Robert H. Halstead, Jr. 1984. Implementation of MultiLisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 9–17. DOI: http://dx.doi.org/10.1145/800055.802017

Robert H. Halstead, Jr. 1985. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538. DOI: http://dx.doi.org/10.1145/4472.4478

William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering Heuristics for Parallel Graph Coloring. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '14)*. ACM, New York, NY, USA, 166–177. DOI: http://dx.doi.org/10.1145/2612669.2612697

Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview Scalability Analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. ACM, New York, NY, USA, 145–156. DOI: http://dx.doi.org/10.1145/1810479.1810509

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

F. L. Hitchcock. 1927. The expression of a tensor or a polyadic as a sum of products. *J. Math. Phys.* (1927).

Derek R. Hower, Polina Dudnik, Mark D. Hill, and David A. Wood. 2011. Calvin: Deterministic or Not? Free Will to Choose. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 333–334. http://dl.acm.org/citation.cfm?id=2014698.2014870

Intel. 2012. The Threading Building Blocks. http://software.intel.com. (2012).

Intel. 2013. Intel Cilk Plus. http://software.intel.com. (2013).

Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA.

Mark T. Jones and Paul E. Plassmann. 1993. A Parallel Graph Coloring Heuristic. *SIAM J. Sci. Comput.* 14, 3 (May 1993), 654–669. DOI:http://dx.doi.org/10.1137/0914041

Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. 1994. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, USA.

Fabian Kuhn. 2009. Weak Graph Colorings: Distributed Algorithms and Applications. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 138–144. DOI:http://dx.doi.org/10.1145/1583991.1584032

Fabian Kuhn and Rogert Wattenhofer. 2006. On the Complexity of Distributed Graph Coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing (PODC '06)*. ACM, New York, NY, USA, 7–15. DOI:http://dx.doi.org/10.1145/1146381.1146387

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

Cliff Lasser and Steve M. Omohundro. 1986. *The Essential Lisp Manual*. Technical Report. Thinking Machines, Cambridge, MA USA.

Doug Lea. 2000. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande (JAVA '00)*. ACM, New York, NY, USA, 36–43. DOI:http://dx.doi.org/10.1145/337449.337465

Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (May 2006), 33–42. DOI:http://dx.doi.org/10.1109/MC.2006.180

I-Ting Angelina Lee, Aamir Shafi, and Charles E. Leiserson. 2012. Memory-mapping Support for Reducer Hyperobjects. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. ACM, New York, NY, USA, 287–297. DOI:http://dx.doi.org/10.1145/2312005.2312056

Optimize managed code for multi-core machines.

Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *J. Supercomput.* 51, 3 (March 2010), 244–257. DOI:http://dx.doi.org/10.1007/s11227-010-0405-3

J. Leskovec. 2013. SNAP: Stanford Network Analysis Platform. http://snap.stanford.edu/data/index.html. (2013).

Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

Nathan Linial. 1992. Locality in Distributed Graph Algorithms. *SIAM J. Comput.* 21, 1 (Feb. 1992), 193–201. DOI:http://dx.doi.org/10.1137/0221015

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727. DOI:http://dx.doi.org/10.14778/2212351.2212354

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. Catalina Island, California.

Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. DOI:http://dx.doi.org/10.1145/1807167.1807184

Andrew McCallum. 2012. Cora data set. http://people.cs.umass.edu/mccallum/data.html. (2012).

Avoiding contention using combinable objects, 2008.

Tom Mitchell. 2009. NPIC500 data set. http://www.cs.cmu.edu/tom/10709_fall2009/NPIC500.pdf.

(2009).

Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. 1999. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI'99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 467–475. http://dl.acm.org/citation.cfm?id=2073796.2073849

Robert H. B. Netzer and Barton P. Miller. 1992. What Are Race Conditions?: Some Issues and Formalizations. *ACM Lett. Program. Lang. Syst.* 1, 1 (March 1992), 74–88. DOI:http://dx.doi.org/10.1145/130616.130623

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 456–471. DOI:http://dx.doi.org/10.1145/2517349.2522739

Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, Portable and Parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 499–512. DOI:http://dx.doi.org/10.1145/2541940.2541964

Kamal Nigam and Rayid Ghani. 2000. Analyzing the Effectiveness and Applicability of Co-training. In *Proceedings of the Ninth International Conference on Information and Knowledge Management (CIKM '00)*. ACM, New York, NY, USA, 86–93. DOI:http://dx.doi.org/10.1145/354756.354805

Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. *SIGARCH Comput. Archit. News* 37, 1 (March 2009), 97–108. DOI:http://dx.doi.org/10.1145/2528521.1508256

Suhas S. Patil. 1970. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation. ACM, New York, NY, USA, Chapter Closure Properties of Interconnections of Determinate Systems, 107–116. DOI:http://dx.doi.org/10.1145/1344551.1344561

Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

James Reinders. 2007. *Intel Threading Building Blocks* (first ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (Feb. 2013), 135–146. DOI:http://dx.doi.org/10.1145/2517327.2442530

Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing Contention Through Priority Updates. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*. ACM, New York, NY, USA, 152–163. DOI:http://dx.doi.org/10.1145/2486159.2486189

Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. ACM, New York, NY, USA, 68–70. DOI:http://dx.doi.org/10.1145/2312005.2312018

Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*. 403–412.

Parag Singla and Pedro Domingos. 2006. Entity Resolution with Markov Logic. In *Proceedings of the Sixth International Conference on Data Mining (ICDM '06)*. IEEE Computer Society, Washington, DC, USA, 572–582. DOI:http://dx.doi.org/10.1109/ICDM.2006.65

Guy L. Steele, Jr. 1990. Making Asynchronous Parallelism Safe for the World. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. ACM, New York, NY, USA, 218–231. DOI:http://dx.doi.org/10.1145/96709.96731

Josef Stoer, Roland Bulirsch, Richard H. Bartels, Walter Gautschi, and Christoph Witzgall. 2002.

Introduction to numerical analysis. Springer, New York. http://opac.inria.fr/record=b1098819

Márió Szegedy and Sundar Vishwanathan. 1993. Locality Based Graph Coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing (STOC '93)*. ACM, New York, NY, USA, 201–207. DOI:http://dx.doi.org/10.1145/167088.167156

Alan M Turing. 1948. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics* 1, 1 (1948), 287–308.

D. J. A. Welsh and M. B. Powell. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Comput. J.* 10, 1 (1967), 85–86.

Jie Yu and Satish Narayanasamy. 2009. A Case for an Interleaving Constrained Shared-memory Multi-processor. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 325–336. DOI:http://dx.doi.org/10.1145/1555815.1555796

Marco Zagha and Guy E. Blelloch. 1991. Radix Sort for Vector Multiprocessors. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 712–721. DOI:http://dx.doi.org/10.1145/125826.126164