Simple, Fast and Scalable Parallel Algorithms for Shared Memory (Thesis Proposal)

Julian Shun

Computer Science Department
Carnegie Mellon University
jshun@cs.cmu.edu

Abstract

To ease the transition into the multicore/manycore era, shared-memory programming must be made more natural and accessible to the community. Furthermore, shared-memory algorithms need to be fast and scalable in order to quickly process large data. In this proposed thesis we will study techniques for simplifying parallel programming and allowing users to easily write efficient and scalable algorithms. Our work will consist of (1) designing a benchmark suite which allows for head-to-head comparisons of parallel languages and architectures for given problems, (2) developing methods for writing deterministic parallel programs to simplify programming and debugging, (3) developing a useful primitive for reducing memory contention on shared-memory multicore machines, (4) building a simple framework for implementing efficient large-scale shared-memory graph algorithms and (5) designing efficient and scalable string processing algorithms. We describe how each portion of our work fits into our overall goal of simplifying the task of writing fast and scalable parallel programs.

Thesis Committee

Guy Blelloch (Chair)
Christos Faloutsos
Jeremy Fineman (Georgetown)
Phillip Gibbons
Charles Leiserson (MIT)
Gary Miller

1 Introduction

The performance of single processor machines has reached a peak due to physical limitations of processor hardware. As a result, many have turned to parallelism to deliver higher performance. Broadly speaking, parallel computing is the study of using multiple resources to perform tasks. Perhaps the most familiar example of parallelism is the presence of multiple cores on personal computers today (in fact almost any personal computer has multiple cores nowadays). Even cellular phones have started to adopt the multicore/manycore trend (for example the iPhone 5 from Apple has a dual-core central processing unit (CPU) and tri-core graphical processing unit (GPU)). In addition to multicore technology, parallelism has presented itself in distributed systems (multiple machines working together), symmetric multiprocessing, field programmable gate arrays (FPGAs), etc. The fastest supercomputers in the world all take advantage of parallelism.

Although parallelism has many advantages, parallel programming is notoriously difficult. In this thesis we study techniques for simplifying parallel programming, and show that these techniques can also lead to parallel algorithms which are fast and scalable. The topics studied are listed below:

1. We describe the Problem Based Benchmark Suite, a set of benchmarks defined only in terms of problem specifications. It contains a wide variety of real-world problems, and can be used for comparing parallel programs written using different algorithms, programming languages and/or architectures.

2. We discuss techniques for writing internally deterministic parallel programs to simplify programming and debugging. We also prove complexity bounds for some of these algorithms.

3. We explore the effect of memory contention on shared memory machines, and propose the priority update operation for reducing contention in parallel algorithms.

4. We develop Ligra, a simple lightweight framework for writing graph algorithms on shared memory machines. We demonstrate that programs written in this framework are simple, efficient and outperform programs written in existing graph processing frameworks.

5. We design simple parallel algorithms for two problems on strings—suffix tree construction and Lempel-Ziv factorization. We show that these algorithms have good theoretical guarantees. Experimentally, our algorithms achieve good scalability and outperform the best sequential algorithms for the same task on a modest number of processors.

Thesis statement: With appropriate tools, parallel algorithms can be simple to write, fast on modern shared-memory multicore machines and scalable to large data.

2 Preliminaries

In this thesis, we use the parallel random access machine (PRAM) model for analyzing algorithms [77]. We use the exclusive-read exclusive-write (EREW), concurrent-read exclusive-write (CREW) and concurrent-read concurrent-write (CRCW) versions of the PRAM in our various algorithms, and will specify which one we use when we use them. For CRCW, we assume both the arbitrary and priority write versions, where a priority write here means that the minimum (or maximum) value written concurrently is recorded. Our results are stated in the work-depth model where work is equal to the number of operations (equivalently the product of the time and processors) and depth is equal to the number of time steps.

3 Problem Based Benchmark Suite (PBBS)

We are developing the Problem Based Benchmark Suite (PBBS) which is a set of benchmarks for non-numerical applications defined only in terms of problem specifications [132]. The motivation for developing this benchmark suite is that for any particular problem, there are many different types of parallel architectures, programming approaches and algorithms that can be used to solve the problem, but it is difficult to obtain a head-to-head comparison of the different solutions. Unlike most existing benchmarks, which are based on specific code, the problem-based benchmarks are defined in terms of the problem specifications—a concrete description of valid inputs and corresponding valid outputs, along with some specific inputs. Any algorithms, programming methodologies, specific programming languages, or machines can be used to solve the problems. The benchmark suite is designed to compare the benefits and shortcomings of different algorithmic and programming approaches, and to serve as a dynamically improving set of educational examples of how to parallelize applications. The nature of PBBS will encourage the community to submit open-source solutions that will be judged by not only its performance but also the quality of the code: its elegance, readability, extensibility, modularity, scalability, correctness guarantees, and the ability to formally analyze performance. Many of these measures are hard to quantify and ultimately the judgment will be in the eye of the reader. Thus, the main outcome should be the code itself (and its performance numbers).

Our benchmark problems are selected to have reasonably simple efficient solutions (all of our base implementations use fewer than 500 lines of code), but represent realistic real-world problems covering a wide class of domains and potential solution approaches. These consist of many well-known problems that are already de facto standards for benchmarking, such as sorting, nearest-neighbor searching, breadth-first search, Delaunay triangulation and ray tracing, as well as many others. In the suite, each benchmark consists of (1) the problem specification including specific input and output file formats, (2) input generators and specific input instances, (3) code for checking the correctness of output for the given input, (4) scripts for running tests, (5) a reasonably efficient sequential base implementation for the problem, and (6) a reasonably efficient parallel (multicore) base implementation for the problem.

3.1 Related Work

Many benchmark suites have been designed and are currently being used for different purposes, but none match our goals for a problem-based suite. There are several broad-based performance-based suites such as SPEC, WorldBench, V8, and DaCapo [16]; and domain-specific benchmarks such as BioBench [4], the San Diego vision benchmarks [138], MediaBench [90] (multimedia), SATLIB [71] (satisfiability), MineBench [109] (data mining), and the TPC benchmarks (databases). Except for SATLIB and the TPC benchmarks, these are code-based benchmarks. The TPC and SATLIB are problem based, but for specific domains.

For parallel machines, there have also been many benchmarks developed. Broad-based performance benchmarks include Splash-2 [142], PARSEC [15], and STAMP [29], which are designed for shared memory machines. Other benchmarks cover a more general class of machines but are meant to measure particular machine characteristics, such as the HPC Challenge Benchmarks [98] that put an emphasis on measuring communication throughput. There are benchmarks aimed at particular languages, such as the Java Grande Benchmark Suite [134]. There are also some domain-specific parallel benchmarks such as ALPBench [92] (multimedia) and BioParallel [78]. All these benchmarks are code-based. The Berkeley "dwarfs" define a set of 13 parallel computational patterns [6]. While sharing some of the same high-level goals as ours (e.g., evaluate parallel programming mod-

Basic Building Blocks	Scan, Integer Sort, Comparison Sort, Remove Duplicates, Dictionary, Sparse matrix-vector multiply
Graph Algorithms	Breadth First Search, Spanning Forest, Minimum Spanning Forest, Maximal Independent Set, Maximal Matching, Graph Separators
Computational Geometry	Quad/Oct Tree, Delaunay Triangulation, Convex Hull, k-Nearest Neighbors
Text Processing	Tokenize, Suffix Array
Computational Biology	Multiple sequence alignment, Phylogenetic tree, N-body
Data Mining	Build Index, Edit Distance Graph
Graphics	Ray Casting, Micropolygon Rendering
Machine Learning	Sparse SVM, K-means, Gibbs Sampling in Graphical Models

Table 1: A preliminary set of 28 problem-based benchmarks covering a reasonably broad set of non-numerical applications.

els), their benchmarks are in terms of patterns, not problems. The Galois benchmarks [117] are defined in terms of particular algorithmic approaches but are not problem based.

In terms of being defined with regards to a problem specification, perhaps the closest benchmarks to PBBS are the NAS benchmarks [7]. In the original form (NPB 1), these consisted of a set of eight problem-based benchmarks where one of the main goals was architecture neutrality. Indeed, several different programming styles (vector code, message passing, data parallel) were used to code the benchmarks on different machines. These benchmarks, however, did not focus on code quality and because vendors were not required to release their codes, some of the solutions were extremely messy. Also, the NAS benchmarks were focused on numerical computing.

Finally, there have also been various attempts to compare programming languages by defining a set of benchmarks. Probably the one that captures the broadest set of languages is the Computer Language Benchmarks Game [52], which compares over 25 programming languages on a set of 12 micro benchmarks. Benchmarks results are reported in terms of performance and size of the `gzip`-compressed source file (comments and redundant whitespace removed). The benchmarks, however, only consider small inputs—for example, their "n-body" benchmark consists of 5 bodies. Also, the benchmarks require that the program use the "same algorithm" as specified—returning the same result is not sufficient.

3.2 Benchmark Problems and Current Status

We selected benchmark problems with the following goals in mind. First, the set of problems should have a wide coverage from state-of-the-art real-world applications. Second, the problem must have a well-defined way to validate output correctness or quality. Third, the problem should have efficient solutions that can be implemented in a reasonably small program. Finally, the inputs to these problem should be scalable. Table 1 summarizes a set of problem-based benchmarks categorized by application domain or type of data. These 28 benchmarks represent our current list of what we believe would make a good mix of problems, though the list is flexible.

We require the program to output the result to a file in a particular format. We provide test code that checks correctness and outputs any quality criteria (e.g. the size of a graph cut). The time for input and output is not included in the running time or code length—for some benchmarks it could dominate the cost.

It is important to have at least one base implementation of each benchmark so that results can be compared and as a proof of concept that the benchmarks fit within our parameters (e.g., have reasonably simple and efficient solutions). We are currently developing two base implementations for each benchmark, one serial and one parallel. Our parallel implementations are designed for multicores and use only parallel loops, nested fork-join, and compare-and-swap operations and are currently implemented in Intel Cilk Plus [74]. We have implemented an initial set of base implementations for some of the benchmarks and have made initial timings. For various sets of inputs, our parallel implementations are competitive with the serial implementations, and achieve speedups ranging from 12 to 32 on a 40-core machine. All code is available on the benchmark webpage: `http://www.cs.cmu.edu/~pbbs`.

3.3 Future Work

In some situations, one may want to take an existing serial implementation and make it parallel without changing the code by too much. We plan on extending the current benchmark suite to include code difference from certain baseline implementations as one of the measurements of quality. Some problems have no clear specification, for example, tracking problems arising in computer vision. In such cases, the input/output behavior can be defined by a serial implementation, and parallel implementations should return results that are of the same quality as that of the serial implementation. Currently we do not have these types of problems in PBBS, but we plan on identifying problems of this type and adding them to PBBS. Finally we intend to explore more enhanced solutions to the existing PBBS benchmark problems.

We plan on fully documenting all of the code in PBBS and increasing the number of problems and test cases. As an extension of our previous work [132], we intend to write a paper explaining each of the benchmarks in detail and comparing with solutions from other benchmarks if available.

4 Determinism

One of the key challenges of parallel programming is dealing with nondeterminism. For many computational problems, there is no inherent nondeterminism in the problem statement, and indeed a serial program would be deterministic—the nondeterminism arises solely due to the parallel program and/or due to the parallel machine and its runtime environment. The challenges of nondeterminism have been recognized and studied for decades [114, 64, 53, 135]. More recently, there has been a surge of advocacy for and research in determinism, seeking to remove sources of nondeterminism via specially-designed hardware mechanisms [43, 44, 72], runtime systems and compilers [11, 13, 111, 144], operating systems [12], and programming languages/frameworks [26, 95].

While there seems to be a growing consensus that determinism is important, there is disagreement as to what degree of determinism is desired. Some examples of determinism include external determinism, internal determinism and functional determinism. There are trade-offs among the various options, with stronger forms of determinism often viewed as better for reasoning and debugging but worse for performance and perhaps programmability.

4.1 Internal Determinism

We advocate a form of *internal determinism* as providing a sweet spot for a class of nested-parallel (i.e., nested fork-join) computations in which there is no inherent nondeterminism in the problem [17]. An execution of a nested-parallel program defines a dependence DAG (directed acyclic graph) that represents every operation executed by the computation (the nodes) along with the control dependencies among them (the edges). These dependencies represent ordering within sequential code sequences, dependencies from a fork operation to its children, and dependencies from the end of such children to the join point of the forking parent. We refer to this DAG when annotated with the operations performed at each node (including arguments and return values, if any) as the *trace*. Informally, a program/algorithm is *internally deterministic* if for any input there is a *unique* trace. This definition depends on the level of abstraction of the operations in the trace. At the most primitive level the operations could represent individual machine instructions, but more generally, and as used in this work, it is any abstraction level at which the implementation is hidden from the programmer. Internal determinism does not imply a fixed schedule since any schedule that is consistent with the DAG is valid. Note that internal determinism is stronger than external determinism, which only requires the output to be deterministic given the same input.

Internal determinism has many benefits. In addition to leading to external determinism [114] it implies a sequential semantics—i.e., considering any sequential traversal of the dependence DAG is sufficient for analyzing the correctness of the code. This in turn leads to many advantages including ease of reasoning about the code, ease of verifying correctness, ease of debugging, ease of defining invariants, ease of defining good coverage for testing, and ease of formally, informally and experimentally reasoning about performance [43, 44, 72, 13, 111, 144, 12, 26, 11]. Two primary concerns for internal determinism, however, are that it may restrict programmers to a style that (1) is complicated to program, and (2) leads to slower, less scalable programs than less restrictive forms of determinism. Indeed, prior work advocating less restrictive forms of determinism has cited these concerns, particularly the latter concern [66].

In this thesis we will address these two concerns by studying a set of benchmark problems and showing that for this broad set of problems, there are fast and scalable internally deterministic solutions, and furthermore these algorithms are natural to reason about and not complicated to code. Our approach for achieving internal determinism for these benchmarks is to use nested parallel programs in which concurrent operations to shared states are required to commute [135, 140] in their semantics and be linearizable [69] in their implementation. Many of the algorithms that we implement use standard algorithmic techniques based on nested data parallelism (e.g. divide-and-conquer, map, reduce, and scan), where the shared states across concurrent operations are read-only. However, a key aspect to several of our algorithms is the use of non-trivial commutative operations on shared states, detailed in the following section.

4.2 Commutative Building Blocks

In this section, we define some useful higher-level operations that we use as commutative operations in many of our algorithms. They are all defined over abstract data types supporting a fixed set of operations. We also have non-blocking linearizable implementations of each operation. These implementations do not commute at the level of single memory instructions and hence the abstraction is important.

Priority update. Our most basic data type is a memory cell that holds a value and supports a priority update and a read. The priority update on a cell x, denoted by $x.\texttt{pUpdate}(v)$ updates x to be the maximum of the old value of x and a new value v. It does not return any value. $x.\texttt{read}()$ is just a standard read of the cell x returning its value.

Any two priority updates $x.\texttt{pUpdate}(v_1)$ and $x.\texttt{pUpdate}(v_2)$ commute, because (i) there are no return values, and (ii) the final value of x is the maximum among its original value, v_1, and v_2, regardless of which order these operations execute. A priority update and a read do not commute since the priority update can change the value at the location.

Dynamic map. The purpose of our dynamic map is to incrementally insert keyed elements and, once finished inserting, to return an array containing a pseudorandom permutation of these elements, omitting duplicates. A dynamic map M supports insertions and an `elements`() call, which returns an arbitrary, but deterministic, permutation of all the elements in the map M. The map removes duplicate keys on inserts, where duplicate keys are discarded deterministically.

In our implementation, insertions commute which each other, however insertions do not commute with the `elements`() operation. Hence, when we use dynamic maps we make sure that insertions are not called logically in parallel with `elements`().

Disjoint sets. Our spanning forest algorithms rely on a structure for maintaining a collection of disjoint sets corresponding to connected components. Each set is associated with a unique element acting as the identifier for the set. A disjoint-set data type supports two operations: a find and a link. For an instance F, the F.`find`(x) operation returns the set identifier for the set containing x. The F.`link`(S, x) operation requires that S be a set identifier and the set containing x be disjoint from the set S. It logically unions the set S with the set containing x such that the identifier for the resulting unioned set is the identifier of the set containing x. Here, x and S denote references or pointers to elements in the sets.

In our implementation, `find` operations commute with each other, as they cause no semantic modifications. Two `link` operations commute with each other as long as they do not share the same first argument. A link operation `link`(S_1, x_1) and a find operation `find`(x_2) only commute if $x_2 \notin S_1$. In our algorithms that use disjoint sets, `find` is never called in parallel with `link`.

4.3 Deterministic Reservations

Several of our algorithms (maximal independent set, maximal matching, spanning forest, Delaunay triangulation, and Delaunay refinement) are based on greedy sequential algorithms that process elements (e.g., vertices or edges) in linear order. We implement these algorithms using our technique of *deterministic reservations*, which preserves internal determinism. Deterministic reservations are speculative executions on a sequential loop that iterates over the elements in the greedy order.

The generic greedy algorithm for deterministic reservations works as follows. Given a sequence of iterates (e.g., the integers from 0 to $n - 1$), it proceeds in rounds until no iterates remain. Each round takes any prefix of the remaining unprocessed iterates, and consists of two phases that are each parallel loops over the prefix, followed by some bookkeeping to update the sequence of remaining iterates. The first phase executes a *reserve* component on each iterate, using a priority update (`pUpdate`) with the iterate priority, in order to reserve access to data that might interfere (involve non-commuting or non-linearizable operations) with other iterates. The second phase executes a *commit* component on each iterate, checking to see if the reservations succeeded (i.e. whether value stored is equal the iterate priority), and if the required reservations succeed then the iterate is processed, otherwise it is not. Typically updates to shared state (at the abstraction level available to the programmer) are only made if successful. After running the commit phase, the processed iterates are removed.

We note that the generic approach can select any prefix size including a single iterate or all the iterates. There is a trade off, however between the two extremes. If too many iterates are selected for the prefix, then many iterates can fail. This not only requires repeated effort for processing those iterates, but can also cause high contention on the reservation slots. On the other hand if too few iterates are selected then there might be insufficient parallelism. Clearly the amount of contention depends on the specific algorithms and likely also on the input data.

4.3.1 Related Work

Various studies have suggested both compiler [123, 119] and runtime techniques [136, 66] to automate the process of simulating in parallel the sequential execution of such a loop. These approaches rely on recognizing at compile and/or run time when operations in the loop iterates commute and allowing parallel execution when they do. Often the programmer can specify what operations commute. We are reasonably sure that the compiler-only techniques would not work for our benchmark problems because the conflicts are highly data dependent and any conservative estimates allowing for all possible conflicts would serialize the loop. We note that our technique does not require any compiler extensions. The runtime techniques typically rely on approaches similar to software transactional memory: the implementation executes the iterations in parallel or out of order but only commits any updates after determining that there are no conflicts with earlier iterations. As with software transactions, the software approach is expensive, especially if required to maintain strict sequential order. In fact in practice the suggested approaches typically relax the total order constraint by requiring only a partial order [119], potentially leading to nondeterminism. A second problem with the software approach is that it makes it very hard for the algorithm designer to analyze efficiency—it is possible that subtle differences in the under-the-hood conflict resolution could radically change which iterates can run in parallel.

4.4 Experiments

The benchmark problems we use include many well-known problems such as sorting, minimum spanning tree, breadth-first search, Delaunay triangulation, and maximal independent set, which are all part of the Problem Based Benchmark Suite (see Section 3). Our experiments show that our internally deterministic algorithms achieve good speedup and good performance even relative to prior nondeterministic and externally deterministic solutions, implying that the performance penalty of internal determinism is quite low. We achieve speedups of up to 31.6 on 32 cores and almost all of our speedups are above 16. Compared to good sequential implementations, our implementations are at most 2 times slower on a single core. All of our implementations are quite concise (20–500 lines of code) and we believe that they are natural to reason about, thereby addressing the concern that internally deterministic algorithms are hard to program. We believe that this combination of performance and understandability provides significant evidence that internal determinism is a sweet spot for a broad range of computational problems.

4.5 Theory of Deterministic Algorithms

Using deterministic reservations, we obtain parallel algorithms that return the same solution as a simple greedy sequential algorithm would return. In this section, we study the theoretical bounds of the maximal independent set and maximal matching algorithms implemented using deterministic reservations.

4.5.1 Definitions

We use $G(V, E)$ to denote a graph with vertex set V and edge set E. We use n and m to refer to the number of vertices and edges, respectively.

The *maximal independent set* (MIS) problem is given an undirected graph $G = (V, E)$ to return a subset $U \subseteq V$ such that no vertices in U are neighbors of each other (independent set), and all vertices in $V \setminus U$ have a neighbor in U (maximal). MIS is a fundamental problem in parallel algorithms with many applications [96]. For example, if the vertices represent tasks and each edge represents the constraint that two tasks cannot run in parallel, MIS finds a maximal set of tasks to run in parallel.

Parallel algorithms for the problem have been well studied. Luby's randomized algorithm [96], for example, runs in $O(\log n)$ time on $O(m)$ processors of an CRCW PRAM. The problem, however, is that on a modest number of processors it is very hard for these parallel algorithms to outperform the very simple and fast sequential greedy algorithm. Furthermore the parallel algorithms give different results than the sequential algorithm. This can be undesirable in a context where one wants to choose between the algorithms based on platform but wants deterministic answers.

A related problem is the maximal matching problem. The *maximal matching* (MM) problem is given an undirected graph $G = (V, E)$ to return a subset $E' \subseteq E$ such that no edges in E' share an endpoint, and all edges in $E \setminus E'$ have a neighboring edge in E'. There are also polylogarithmic depth and linear work algorithms for the MM problem [76, 75], but they do not return the same result as the sequential greedy algorithm.

4.5.2 Results

We show that, perhaps surprisingly, a trivial parallelization of the sequential greedy algorithm is in fact highly parallel (polylogarithmic time) when the order of vertices is randomized [19]. In particular, processing each vertex as in the sequential algorithm as soon as it has no more neighbors earlier in the ordering gives a parallel linear work algorithm. The MIS returned by the sequential greedy algorithm, and hence also its parallelization, is referred to as the *lexicographically first* MIS [34]. In a general undirected graph and an arbitrary order, the problem of finding a lexicographically first MIS is P-complete [34, 60], meaning that it is unlikely that any efficient low-depth parallel algorithm exists for this problem. Moreover, it is even P-complete to approximate the size of the lexicographically first MIS [60]. Our results show that for any graph and for the vast majority of orders the lexicographically first MIS has $O(\log^2 n)$ depth.

The MM of G can be solved by finding an MIS of its line graph (the graph representing adjacencies of edges in G), but the line graph can be asymptotically larger than G. Instead, the efficient (linear time) sequential greedy algorithm goes through the edges in an arbitrary order adding an edge if no adjacent edge has already been added. As with MIS this is naturally parallelized by adding in parallel all edges that have no earlier neighboring edges. Our results on MIS directly imply that this algorithm has $O(\log^2 m)$ depth depth for random edge orderings. We show how to make the both our MIS and MM algorithms run in linear work, while incurring a logarithmic factor in the depth bound.

4.6 Future Work

4.6.1 Theory

We found that the deterministic reservations-based spanning tree and minimum spanning tree implementations also perform very well experimentally but we have not yet been able to prove any good theoretical bounds (polylogarithmic depth) for them. The techniques we used for MIS and MM do not directly apply due to the different dependences among elements. Our next goal is to come up with good complexity bounds for our spanning forest and minimum spanning forest algorithms. We have made a first step towards this goal by showing a polylogarithmic depth bound for a deterministic reservations-based algorithm for list contraction.

4.6.2 Race Detectors for Commutativity Analysis

There has been previous work on data races and determinacy races in nested parallel programs (e.g. Cilk) [10, 31, 50, 51], and has recently been extended to a more general class of async/finish programs [122, 121]. These works include theoretical analyses of the serial and parallel race detectors, and experimental results. Previous race detectors only handle races at the level of read and write operations on memory locations. We plan to generalize the detection schemes to the object level, and allow the user to provide specifications on what constitutes a "race". There can be more than two types of operations on each object, which makes designing the detection protocols more challenging. One application of a more general "race" detector is to detect violation of commutativity specifications when using our commutative building blocks (described in Section 4.2). A separate goal is to design the race detector to run quickly in parallel.

4.6.3 Concurrent Hash Tables

We plan to further study the concurrent hash table (dynamic map) discussed in Section 4.2 [23]. We will prove the correctness of the various operations supported by the hash table. We will also experimentally analyze the efficiency of the various operations under various loads, and compare our performance to previous implementations of concurrent hash tables [68, 89].

We plan on using the hash table in various applications to simplify their implementations and perhaps improve their performance. Additionally, the hash table preserves internal determinism when used in an application. One example is to use hash tables to simplify and improve the efficiency of parallel breadth-first search. In parallel breadth-first search, one can use a hash table to store the frontier of vertices. This deterministically eliminates duplicates, which not only results in a deterministic breadth-first search tree, but also obviates the need to separately remove duplicates which could possibly lead to more efficient code. In Delaunay triangulation, we currently represent the triangles as a collection of pointers, which is complicated to maintain. Using a dynamic map to insert and delete triangles would dramatically simplify the code. Our graph-partitioning code is based on recursive graph contraction, and after each contraction, we must combine the weights of duplicate edges between components. This process can be implemented relatively easily by using a hash table to keep only a single edge between two components, adding the weights of duplicate edges to this single edge on insertion. The performance of connectivity algorithms based on graph contraction could possibly be improved by using a hash table to store the edges.

5 Memory Contention

Memory contention can be a serious performance bottleneck in concurrent programs on shared-memory multicore architectures. Having all threads write to a small set of shared locations, for example, can lead to orders of magnitude loss in performance relative to all threads writing to distinct locations, or even relative to a single thread doing all the writes. Shared write access, however, can be very useful in parallel algorithms, concurrent data structures, and protocols for communicating among threads.

Some of our internally deterministic algorithms operate on shared state and are thus potential victims of memory contention. These algorithms operate on shared state through priority updates, discussed in Section 4.2. We will further investigate the priority update operation, both experimentally and theoretically.

We distinguish between sharing and contention. By ***sharing*** we mean operations that share the same memory location (or possibly other resource)—for example, a set of instructions reading a single location. By ***contention*** we mean some form of sequential access to a resource that causes a bottleneck. Contention can be a major source of performance problems on parallel systems while sharing need not be. A key motivation for the priority update operation is to reduce contention under a high degree of sharing.

5.1 Related Work

The problem with contention for shared memory access has been recognized since at least the early 80s and the work on the NYU Ultracomputer [59, 58]. To avoid or reduce the problem, the researchers suggested that hardware for combining requests be added to the memory system. Although this was a nice idea, as far as we know no machine since the IBM RP3 (a follow up on the Ultracomputer) and the Connection Machine II supported combining in hardware. Besides the cost of implementing the combining techniques, the problem is that there are many possible operations to use in combining and none or no known combination are universal, making it hard to decide which ones to support. Furthermore, operations such as the compare-and-swap cannot be combined. More recent work has suggested that combining be made more flexible so that it can be programmed and used at the granularity of cache lines [46, 20]. However, no current machines supports hardware combining.

To avoid the need for hardware combining researchers have suggested techniques for software combining [128, 40, 49]. The idea of most of these approaches is to simulate in some way using software what hardware combining would do. However, such combining can have a high overhead due to various conditional checks. To avoid such overheads more recent work has suggested simply sequentially combining from a buffer that has one location per core (or few cores) [67]. This approach, however, does not scale.

Other work on avoiding contention involves recognizing that certain simple protocols can significantly reduce the number of concurrent updates to a shared location [125, 103], but the operations are insufficient for many applications.

5.2 Preliminary Results

We implemented the priority update operation in software using the compare-and-swap primitive, and experimentally showed that even in cases of high memory contention (for example, a vertex in a graph concurrently written to by many neighboring vertices), our algorithms using priority update do not suffer much in performance, whereas algorithms using plain writes do [131]. This is due to properties of our implementation of priority update, which allows most of the writers to avoid doing any writes (which would happen every time in the case of plain writes).

5.3 Future Work

We plan on proving theoretical properties of the priority update operation [130]. Also we will perform a more extensive experimental analysis of the priority update operation and compare its performance with that of fetch-and-add, compare-and-swap, test-and-set and plain writes. In addition, we will explore the performance of priority update in applications on inputs causing high sharing. We plan to develop more theoretical results of the priority update operation. Inspired by the work of Gibbons et. al. [54, 55], we are interested in developing a theoretical model for parallel programs which can accurately capture the contention in computations and identify when using priority updates will be beneficial.

6 Graph Algorithms

6.1 The Ligra Framework

We present Ligra, a framework for implementing parallel graph algorithms on shared-memory multicore machines. Recently several packages have been developed for processing large graphs on parallel machines including the parallel Boost graph library (PBGL) [61], Pregel [99], PEGASUS [80], GraphLab [93, 94], PowerGraph [56], the Knowledge Discovery Toolkit [27, 97], GPS [126], Giraph [1], and Grace [118]. Motivated by the need to process very large graphs, most of these systems (with the exception of the original GraphLab) have been designed to work on distributed memory parallel machines. Although shared memory machines cannot scale to the same size as distributed memory clusters, current commodity single unit servers can easily fit graphs with well over a hundred billion edges in memory, large enough for any of the graphs reported in the papers of the aforementioned packages. Furthermore, commodity shared-memory servers are quite reliable, often running for up to months or years without a failure.

Ligra is a lightweight interface for graph algorithms for shared-memory that is particularly well-suited for graph traversal problems. Such problems visit possibly small subsets of the vertices on each step. The interface is lightweight in that other than constructors and size query functions it supplies only two functions, one for mapping over vertices and the other for mapping over edges. The implementation is simple (a few hundred lines of code), fast, uses only a modest amount of memory, and scales to large graphs.

6.1.1 Interface

Ligra supports two data types, one representing a graph $G = (V, E)$ with vertices V and edges E, and another for representing subsets of the vertices V, which we refer to as *vertexSubset*s. Other than constructors and size queries, the interface supplies only two functions, one for mapping over vertices (*vertexMap*) and the other for mapping over edges (*edgeMap*). Since a vertexSubset is a subset of V, the vertexMap can be used to map over any subset of the original vertices, and hence its utility in traversal algorithms—or more generally in any algorithm in which only (possibly small) subsets of the graph are processed on each round. The edgeMap also processes a subset of the edges, which is specified using a vertexSubset to indicate the valid sources, and a Boolean function to indicate the valid targets of each edge. Abstractly, a vertexSubset is simply a set of integer labels for the included vertices and the vertexMap simply applies the user supplied function to each integer. It is up to the user to maintain any vertex based data. The implementation switches between a sparse and dense representation of the integers depending on the size of the vertexSubset. In our interface, multiple vertexSubsets can be maintained and furthermore, a vertexSubset can be used for multiple graphs with different edge sets, as long as the number of vertices in the graphs are the same.

6.1.2 Example: Breadth-First Search

With this interface a breadth-first search (BFS), for example, can be implemented as shown in Figure 2. This version of BFS uses a Parents array (initialized all to -1, except for the root r where Parents$[r] = r$) in which each vertex reachable from r will point to its parent in a BFS tree.

```
 1: Parents = $\{-1, \ldots, -1\}$                                   ▷ initialized to all -1's
 2:
 3: procedure UPDATE($s$, $d$)
 4:     return (CAS(&Parents[$d$], $-1$ , $s$ ))
 5:
 6: procedure COND($i$)
 7:     return (Parents[$i$] == $-1$)
 8:
 9: procedure BFS($G$, $r$)                                           ▷ $r$ is the root
10:     Parents[$r$] = $r$
11:     Frontier = $\{r\}$                       ▷ vertexSubset initialized to contain only $r$
12:     while (SIZE(Frontier) $\neq 0$) do
13:         Frontier = EDGEMAP($G$, Frontier, UPDATE, COND)
```

Figure 1: Pseudocode for Breadth-First Search in our framework. The compare-and-swap function CAS(*loc*,*oldV*,*newV*) atomically checks if the value at location *loc* is equal to *oldV* and if so it updates *loc* with *newV* and returns *true*. Otherwise it leaves *loc* unmodified and returns *false*.

As with standard parallel versions of BFS [132, 91], on each step i (starting at 0) the algorithm maintains a frontier of all vertices reachable from the root r in i steps. Initially a vertexSubset containing just the root vertex is created to represent the frontier (line 11). Using edgeMap, each step checks the neighbors of the frontier to see which have not been visited, updates those to point to their parent in the frontier, and adds them to the next frontier (line 13). The user supplied function UPDATE (lines 3–4) atomically checks to see if a vertex has been visited using a compare-and-swap (CAS) and returns true if not previously visited (Parents[i] == -1). The COND function (lines 6–7) tells edgeMap to consider only target vertices which have not been visited (here, this is not needed for correctness, but is used for efficiency). The edgeMap function returns a new vertex set containing the target vertices for which UPDATE returns true, i.e., all the vertices in the next frontier (line 13). The BFS completes when the frontier is empty and hence no more unvisited vertices are reachable.

6.1.3 Experiments

Ligra achieves close to the same efficiency (time and space) as the most efficient BFS implementation for shared-memory [8, 9], and we apply it to many other applications including betweenness centrality, graph radii estimation, graph connectivity, PageRank and single-source shortest paths. Our solutions get quite good speedups over sequential implementations (up to 39 fold on 40 cores in our experiments), and compared to the distributed memory systems mentioned above, our system is over an order of magnitude faster on a per-core basis for the benchmarks we could compare with, and often faster even on absolute terms to the largest systems run, which sometimes have two orders of magnitude more cores. We test our implementations on some of the largest real-world graphs available, such as the Yahoo Web graph (1.4 billion vertices and 6.6 billion edges) [143] and the Twitter graph (41.7 million vertices and 1.47 billion edges) [87]. These were the largest real-world graphs reported in the papers of the other graph processing systems.

6.2 Future Work

6.2.1 GPUs

There has been recent work on designing breadth-first search algorithms for graphics processing units (GPUs) [104, 70]. One direction for future work is to extend Ligra to the GPU setting. This would involve addressing issues in GPU computing, such as thread divergence, shared memory bank conflicts, and how to maximize usage of the memory bandwidth. A framework for graph algorithms for shared-memory using GPUs (possibly as part of an accelerated processing unit) could potentially deliver even better performance.

6.2.2 External Memory Graph Algorithms

Another direction for future work is to incorporate techniques into Ligra to handle graphs which do not fit in memory, for example as done in GraphChi [88].

We are interested in proving theoretical properties of disk-based graph algorithms written within the GraphChi framework, and comparing their performance with previous disk-based algorithms for the same problem. There has been a decent amount of research on the theory of external memory algorithms [32, 2, 102, 41], and we would like to design GraphChi-based algorithms that match the previously obtained bounds. We will also design algorithms that are experimentally competitive with the disk-based algorithms that have been previously implemented [41, 3, 33, 25].

6.2.3 Cache-Oblivious Triangle Listing

We are interested in developing a cache-oblivious algorithm for triangle listing, which lists all of the triangles (3-cycles) in a graph, which has many applications in data mining [127]. This problem is harder than that of triangle counting, which only requires a total count of the number of triangles in a graph. Chu and Cheng [33] develop a disk-based algorithm for triangle listing, however their algorithms makes assumes that the degrees of the vertices are not too large and that the number of vertices fit in memory. We would like to improve upon their work and extend it to the cache-oblivious setting by using the techniques of [21, 18].

6.2.4 Other Graph Algorithms

We are interested in further exploring graph problems arising in data mining. For example, it would be interesting to prove guarantees (theoretically or empirically) of our parallel algorithm for graph radii approximation [129] based on multiple breadth-first searches, and compare it to the only other parallel algorithm for the same task by Kang et. al. [79].

7 String Algorithms

We present parallel algorithms for two string processing problems—suffix tree construction and Lempel-Ziv factorization. Our algorithms are simple and our experiments show that they are fast and scale to large data.

7.1 Suffix Trees

Suffix trees support constant-time searches in strings and also efficiently support many other operations on strings, such as longest common substring, maximal repeats, longest repeated substrings, and longest palindrome, among many others [62]. As such it is one of the most important data structures for string processing. For example, it is used in several bioinformatic applications, such as REPuter [86], MUMmer [39], OASIS [101] and Trellis+ [115, 116].

7.1.1 Preliminaries

Given a string s of length n over an ordered alphabet Σ, the *suffix array*, SA, represents the n suffixes of s in lexicographically sorted order. To be precise, $SA[i] = j$ if and only if the suffix starting at the j'th position in s appears in the i'th position in the suffix-sorted order. A *Patricia tree* [107] (or compacted trie) of a set of strings S is a modified trie in which (1) edges can be labeled with a sequence of characters instead of a single character, (2) no node has a single child, and (3) every string in S corresponds to concatenation of labels for a path from the root to a leaf. Given a string s of length n, the *suffix tree* for s stores the n suffixes of s in a Patricia tree.

In addition to supporting searches in s for any string $t \in \Sigma^*$ in $O(|t|)$ expected time (worst case time for constant sized alphabets), suffix trees efficiently support many other operations on strings, such as longest common substring, maximal repeats, longest repeated substrings, and longest palindrome, among many others [62].

We assume an integer alphabet $\Sigma \subseteq [1, \ldots, n]$ where n is the total number of characters. We require that the Patricia tree and suffix tree support the following queries on a node in constant expected time: finding the child edge based on the first character of the edge, finding the first child, finding the next and previous sibling in the character order, and finding the parent. If the alphabet is constant sized all these operations can easily be implemented in constant worst-case time.

A *Cartesian tree* [139] on a sequence of elements taken from a total order is a binary tree that satisfies two properties: (1) heap order on values, i.e. a node has an equal or lesser value than any of its descendants, and (2) an inorder traversal of the tree defines the sequence order. If elements in the sequence are distinct then the tree is unique, otherwise it might not be. When elements are not distinct we refer to a connected component of equal value nodes in a Cartesian tree as a *cluster*. A *multiway Cartesian tree* is derived from a Cartesian tree by contracting each cluster into a single node while maintaining the order of the children. A multiway Cartesian tree of a sequence is always unique.

Let LCP(s_1,s_2) be the length of the longest common prefix of s_1 and s_2. Given a sorted sequence of strings $S = [s_1, \ldots, s_n]$, if the string lengths are interleaved with the length of their longest common prefixes (i.e. $[|s_1|$, $LCP(s_1, s_2)$, $|s_2|$,..., $LCP(s_{n-1}, s_n)$,$|s_n|]$) the corresponding multiway Cartesian tree has the structure of the Patricia tree for S. The Patricia tree can be generated by adding strings to the edges, which is easy to do—e.g. for a node with value $v = LCP(s_i, s_{i+1})$ and parent with value v' the edge corresponds to the substring $s_i[v' + 1, \ldots, v]$. As a special case, interleaving a suffix array with its LCPs and generating the multiway Cartesian tree gives the suffix tree structure. In summary, beyond some trivial operations, generating a multiway Cartesian tree is sufficient for converting a suffix array and its corresponding LCPs to a suffix tree.

7.1.2 Related Work

Both suffix trees and a linear time algorithm for constructing them were introduced by Weiner [141]. Since then various similar constructions have been described [100, 137] and there have been many implementations of these algorithms. Although originally designed for fixed-sized alphabets with deterministic linear time, Weiner's algorithm can work on an alphabet $[1, \ldots, n]$ in linear expected time simply by using hashing to access the children of a node.

The algorithm of Weiner and its derivatives are all incremental and inherently sequential. The first parallel algorithm for suffix trees was given by Apostolico et. al. [5] and was based on a quite different doubling approach. For a parameter $0 < \epsilon \leq 1$ the algorithm runs in $O(\frac{1}{\epsilon}\log n)$ time, $O(\frac{n}{\epsilon}\log n)$ work and $O(n^{1+\epsilon})$ space on the CRCW PRAM for arbitrary alphabets. Although reasonably simple, this algorithm is likely not practical since it is not work efficient and uses superlinear memory (by a polynomial factor). The parallel construction of suffix trees was later improved to linear work and space by Hariharan [65], with an algorithm taking $O(\log^4 n)$ time on the CREW PRAM, and then by Farach and Muthukrishnan to $O(\log n)$ time using a randomized CRCW PRAM [48] (high-probability bounds). These later results are for a constant-sized alphabet, are "considerably non-trivial", and do not seem to be amenable to efficient implementations.

One way to construct a suffix tree is to first generate a suffix array (an array of pointers to the lexicographically sorted suffixes), and then convert it to a suffix tree. For binary alphabets and given the length of the longest common prefix (LCP) between adjacent entries this conversion can be done sequentially by generating a Cartesian tree in linear time and space. The approach can be generalized to arbitrary alphabets using multiway Cartesian trees without much difficulty. Using suffix arrays is attractive since in recent years there has been considerable theoretical and practical advances in the generation of suffix arrays (see e.g. [120]). The interest is partly due to their need in the widely used Burrows-Wheeler compression algorithm [28], and also as a more space-efficient alternative to suffix trees. As such there have been dozens of papers on efficient implementations of suffix arrays. Among these Karkkainen and Sanders have developed a quite simple and efficient parallel algorithm for suffix arrays [81] that can also generate LCPs.

The story with generating Cartesian trees in parallel is less satisfactory. Berkman et. al [14] describe a parallel algorithm for the all nearest smaller values (ANSV) problem, which can be directly used to generate a binary Cartesian tree for fixed sized alphabets. However, it cannot directly be used for non-constant sized alphabets, and the algorithm is very complicated. Iliopoulos and Rytter [73] present two much simpler algorithms for generating suffix trees from suffix arrays, one based on merging and one based on a variant of the ANSV problem that allows for multiway Cartesian trees. However they both require $O(n \log n)$ work.

7.1.3 Results

We develop a linear work, linear space, and $O(\log^2 n)$ time algorithm for generating multiway Cartesian trees on the CREW PRAM [22]. In conjunction with the parallel suffix array algorithm of [81], this gives a rather simple suffix tree algorithm requiring linear work and $O(\log^2 n)$ time algorithm for constant-sized alphabets, or $O(n^\epsilon)$ time ($0 < \epsilon \leq 1$) integer alphabets on the CRCW PRAM.

We have implemented a version of our algorithm for shared-memory. First, we compare our Cartesian tree algorithm with a simple stack-based sequential implementation. On one core our algorithm is about 3.5x slower, but achieves about 47x speedup on 40 cores with hyper-threading (2 threads per core). We also analyze the algorithm when used as part of code to generate a suffix tree from the original string. We compared our parallel algorithm with the best sequential suffix tree algorithm [85] available online. For a variety of real-world and artificial input strings, our algorithm achieves 11–24 times speedup on 40 cores and is up to 31 times faster than the sequential algorithm. We also show experimentally that searching and computing certain properties of repeated substrings is efficient using our suffix tree [24].

7.2 Lempel-Ziv Compression

Lempel-Ziv-77 (LZ77) is a lossless dynamic compression method that has been popular due to its simplicity and computational efficiency. It is used in the DEFLATE algorithm, which is used in software packages such as gzip and PKZIP among others. It has also been used in algorithms for detecting maximal repetitions in strings [84, 63]. The LZ77 algorithm consists of a compression stage, which computes the Lempel-Ziv factorization (LZ-factorization) of the input string, and a decompression stage, which recovers the original string from the compressed string.

7.2.1 Preliminaries

The LZ-factorization of a string $\mathsf{S}[0,\ldots,\mathsf{n}-1]$ is $\mathsf{S} = \omega_0\omega_1\ldots\omega_{\mathsf{m}-1}$, where $m \leq n$ and for each $0 \leq i < m$, ω_i (called the ith ***factor*** of the string) is either a single character which does not appear in $\omega_0\ldots\omega_{i-1}$ or is the longest prefix of $\omega_i\ldots\omega_{m-1}$ that also appears starting at a position to the left of ω_i in S.

7.2.2 Related Work

The LZ-factorization can be computed sequentially [124] in linear time with a suffix tree [100], and decompression can be done sequentially in linear time with a scan. The first parallel algorithms for LZ-factorization were described independently by Noar [108] and Crochemore and Rytter [38]. Their algorithms require $O(\log n)$ time and $O(n \log n)$ work, which make them not work-efficient. Farach and Muthukrishnan [47] give the first work-optimal algorithms for both LZ-factorization and decompression, each requiring $O(\log n)$ expected time, and they make use of a parallel suffix tree algorithm [48].

There has been much research done in designing practical sequential algorithms for computing LZ-factorization. Recently researchers have proposed the use of suffix arrays instead of suffix trees to obtain faster and more space-efficient algorithms for LZ-factorization [37, 35, 30, 36, 110, 82, 57]. Since suffix arrays can be computed in linear time [81], these LZ-factorization algorithms also able to run in linear time. The aforementioned sequential algorithms have been shown to perform well in practice.

To the best of our knowledge, the only parallel implementations of LZ-factorization described in the literature are those of Klein and Wiseman (using CPUs) [83] and Ozsoy and Swany (using GPUs) [112]. Both implementations involve splitting the input string among processors and having each processor independently compute the factorization of its substring. Because in these implementations the processors do not necessarily have access to the entire input string, they do not always compute the same LZ-factorization as would be computed sequentially, and thus can produce larger compressed files. Furthermore, the corresponding papers [83, 112] do not provide any complexity bounds on work and time. Previous work on parallel algorithms for computing the same LZ-factorization as would be computed sequentially do not include any implementations or experiments [47, 108, 38]. The linear-work algorithm of Farach and Muthukrishnan [47] is quite complicated and not amenable to a practical implementation.

7.2.3 Results

We present a simple linear-work parallel algorithm for LZ-factorization and practical implementations of it [133]. Our algorithm computes the same factorization as would be computed sequentially. The algorithm is based on parallel suffix arrays [81], finding all nearest smaller values [14], and uses simple parallel routines such as prefix sums and leaffix operations [77]. Theoretically, our algorithm requires $O(n)$ work and $O(\log^2 n)$ time (randomized) on the CRCW PRAM due to the use of suffix arrays, so does not achieve the $O(\log n)$ time bound of Farach and Muthukrishnan [47], but lends itself to a practical implementation. We show experimentally that on 40 cores with hyper-threading we achieve speedups between 10 and 22 compared to running the algorithm on a single thread, and outperforms good sequential implementations with just 2 or more threads.

7.3 Future Work

We plan on submitting a journal version of our paper on parallel suffix tree construction [22], which will contain new and updated experiments on larger data, and a more detailed description of our simpler algorithm for computing all nearest smaller values.

Recently there has been work on implementing compression algorithms on the GPU [113]. We would like to extend this work by implementing suffix trees and Lempel-Ziv factorization on the GPU. Suffix arrays and the longest common prefix arrays, which are components of our algorithms, have recently been implemented on the GPU by Deo and Keely [42], using integer sorting and scan techniques developed by Merrill and Grimshaw [105, 106].

There has been recent work on developing an adaptation of Lempel-Ziv-78 compression to allow for fast random access support on the compressed representation of the data [45]. We would like to see whether there is an analogous algorithm for LZ77.

8 Schedule

- Spring 2013: Submit journal version of suffix tree paper [22], and continue work on concurrent hash tables and reducing memory contention.
- Summer 2013: Figure out bounds for the deterministic spanning tree algorithm, work on external graph algorithms, and look at parallel I/O-efficient and cache-oblivious algorithms for triangle listing.
- Fall 2013: Look at race detectors for commutative building blocks, and extend the Problem Based Benchmark Suite.
- After 2013: Look at other graph and string problems, study GPUs and look at GPU algorithms, and do more research on determinism.

References

[1] Apache incubator giraph. `http://uncubator.apache.org/giraph`, 2012.

[2] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[3] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory BFS implementation. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 2007.

[4] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. BioBench: A benchmark suite of bioinformatics applications. In *IEEE ISPASS*, 2005.

[5] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algomthmica*, 3:347–365, 1988.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, UC Berkeley, 2006.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings Supercomputing '91*, 1991.

[8] S. Beamer, K. Asanovic, and D. Patterson. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for Graph500. *Technical Report, UC Berkeley*, 2011.

[9] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *SC*, 2012.

[10] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures*, 2004.

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

[12] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Usenix OSDI*, 2010.

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

[14] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14:371–380, 1993.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[16] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM OOPSLA*, 2006.

[17] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic algorithms can be fast. In *Proceedings of Principles and Practice of Parallel Programming*, PPoPP, 2012.

[18] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Symposium on Parallelism in Algorithms and Architectures*, June 2011.

[19] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *SPAA*, 2012.

[20] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Combinable memory-block transactions. In *SPAA*, pages 23–34, 2008.

[21] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low-depth cache oblivious algorithms. In *ACM SPAA*, 2010.

[22] G. E. Blelloch and J. Shun. A simple parallel cartesian tree algorithm and its application to suffix tree construction. In *ALENEX*, 2011.

[23] G. E. Blelloch and J. Shun. A concurrent history-independent hash table and its applications. 2013. In submission.

[24] G. E. Blelloch and J. Shun. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. 2013. Journal version in submission.

[25] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and I/O efficient set covering algorithms. In *SPAA*, 2012.

[26] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Usenix HotPar*, 2009.

[27] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 2011.

[28] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, HP Labs, 1994.

[29] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[30] G. Chen, S. Puglisi, and W. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1, 2008.

[31] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *ACM SPAA*, 1998.

[32] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. of the ACM-SIAM symposium on Discrete algorithms*, 1995.

[33] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011.

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

[35] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2), Apr. 2008.

[36] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń. Combinatorial algorithms. chapter LPF Computation Revisited. 2009.

[37] M. Crochemore, L. Ilie, and W. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *Data Compression Conference, 2008*, 2008.

[38] M. Crochemore and W. Rytter. Efficient parallel algorithms to test square-freeness and factorize strings. *Inf. Process. Lett.*, 38(2), Apr. 1991.

[39] A. Delcher, A. Phillippy, J. Carlton, and S. Salzberg. Fast algorithms for large-scale genome alignment and comparision. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[40] G. Della-Libera and N. Shavit. Reactive diffracting trees. *J. Parallel Distrib. Comput.*, 60(7), 2000.

[41] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. *Exploring New Frontiers of Theoretical Informatics*, pages 195–208, 2004.

[42] M. Deo and S. Keely. Parallel suffix array and least common prefix for the GPU. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013.

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

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

[45] A. Dutta, R. Levi, D. Ron, and R. Rubinfield. A simple online competitive adaptation of Lempel-Ziv compression with efficient random access support. In *DCC*, 2013.

[46] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker. Active memory operations. In *Proc. 21st ACM Int'l Conf. on Supercomputing*, 2007.

[47] M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression (extended abstract). In *Proceedings of the ACM symposium on Parallel algorithms and architectures*, 1995.

[48] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *ICALP*, 1996.

[49] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *PPoPP*, 2012.

[50] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1997.

[51] J. T. Fineman and C. E. Leiserson. Race detectors for Cilk and Cilk++ programs. *Encyclopedia of Parallel Computing*, pages 1706–1719, 2011.

[52] B. Fulgham. The computer language benchmarks game. http://shootout.alioth.debian.org/, 2009.

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

[54] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. *J. Comput. Syst. Sci.*, 53(3), 1996.

[55] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. Comput.*, 28(2), 1998.

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

[57] K. Goto and H. Bannai. Simpler and faster Lempel Ziv factorization. In *DCC*, 2013.

[58] A. Gottlieb, R. Grishman, C. P. Kruskal, C. P. Mcauliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer—designing an MIMD parallel computer. *IEEE Trans. Comput.*, C-32(2):75–89, 1983.

[59] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.*, 5(2), 1983.

[60] R. Greenlaw, J. H. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, USA, Apr. 1995.

[61] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[62] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[63] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4), Dec. 2004.

[64] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4), 1985.

[65] R. Hariharan. Optimal parallel suffix tree construction. In *Proceedings of the ACM Symposium on Theory of Computing*, 1994.

[66] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *ACM PPoPP*, 2011.

[67] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[68] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *DISC*, 2008.

[69] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[70] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *PACT*, pages 78–88, 2011.

[71] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. P. Gent, H. v. Maaren, and T. Walsh, editors, *SAT 2000.* IOS Press, 2000.

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

[73] C. Iliopoulos and W. Rytter. On parallel transformations of suffix arrays into suffix trees. In *AWOCA*, 2004.

[74] Intel. Cilk++ programming language, 2010. `http://software.intel.com/en-us/articles/intel-cilk`.

[75] A. Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22:77–80, February 1986.

[76] A. Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22:57–60, February 1986.

[77] J. Jaja. *Introduction to Parallel Algorithms.* Addison-Wesley Professional, 1992.

[78] A. Jaleel, M. Mattina, and B. Jacob. Last-level cache (LLC) performance of data-mining workloads on a CMP—A case study of parallel bioinformatics workloads. In *IEEE HPCA*, 2006.

[79] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. In *TKDD*, 2011.

[80] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011.

[81] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. In *ICALP*, 2003.

[82] D. Kempa and S. J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *ALENEX*, 2013.

[83] S. T. Klein and Y. Wiseman. Parallel Lempel Ziv coding. *Discrete Appl. Math.*, 146(2), 2005.

[84] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1999.

[85] S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice Experience*, 1999.

[86] S. Kurtz and C. Schleiermacher. Reputer: Fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, 1999.

[87] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

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

[89] D. Lea. Hash table util.concurrent.concurrenthashmap in java.util.concurrent the Java Concurrency Package.

[90] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM MICRO*, 1997.

[91] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, 2010.

[92] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *IEEE IISWC*, 2005.

[93] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[94] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[95] L. Lu and M. L. Scott. Toward a formal semantic framework for deterministic parallel programming. In *Proceedings of the 25th international conference on Distributed computing*, 2011.

[96] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 1986.

[97] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *SIAM Conference on Data Mining*, 2012.

[98] P. Luszczek, D. Bailey, J. Dongarra, J. Kepner, R. Lucas, R. Rabenseifner, and D. Takahashi. The HPC challenge (HPCC) benchmark suite. In *ACM/IEEE SC06 Conference Tutorial*, 2006.

[99] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.

[100] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[101] C. Meek, J. M. Patel, and S. Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *VLDB*, 2003.

[102] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. *ESA*, 2002.

[103] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. *SIGPLAN Not.*, 26(4), Apr. 1991.

[104] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012.

[105] D. Merrill and A. S. Grimshaw. Parallel scan for stream architectures. *Technical Report, Department of Computer Science, University of Virginia*, 2009.

[106] D. Merrill and A. S. Grimshaw. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 2011.

[107] D. R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[108] M. Naor. String matching with preprocessing of text and pattern. In *ICALP*, 1991.

[109] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. N. Choudhary. MineBench: A benchmark suite for data mining workloads. In *IEEE IISWC*, 2006.

[110] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *Proceedings of the 22nd annual conference on Combinatorial pattern matching*, CPM'11.

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

[112] A. Ozsoy and M. Swany. CULZSS: LZSS lossless data compression on CUDA. In *Cluster Computing (CLUSTER)*, 2011.

[113] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. Parallel lossless data compression on the GPU. In *Innovative Parallel Computing*, 2012.

[114] S. S. Patil. Closure properties of interconnections of determinate systems. In J. B. Dennis, editor, *Record of the Project MAC conference on concurrent systems and parallel computation.* ACM, 1970.

[115] B. Phoophakdee and M. Zaki. Trellis+: An effective approach for indexing genome-scale sequences using suffix trees. In *Pacific Symposium on Biocomputing*, 2008.

[116] B. Phoophakdee and M. J. Zaki. Genome-scale disk-based suffix tree indexing. In *Proceedings of the ACM SIGMOD international conference on Management of data*, 2007.

[117] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, 2011.

[118] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX ATC*, 2012.

[119] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *ACM PLDI*, 2011.

[120] S. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.

[121] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *Proceedings of the First international conference on Runtime verification*, 2010.

[122] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012.

[123] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM TOPLAS*, 19(6), 1997.

[124] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. ACM*, 28(1), Jan. 1981.

[125] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *ISCA*, pages 340–347, 1984.

[126] S. Salihoglu and J. Widom. GPS: A graph processing system. Technical Report InfoLab 1039, Stanford University, 2012.

[127] T. Schank. *Algorithmic Aspects of Triangle-Based Network Analysis.* PhD thesis, Universität Karlsruhe, 2007.

[128] N. Shavit and A. Zemach. Combining funnels: a dynamic approach to software combining. *J. Parallel Distrib. Comput.*, 60(11), Nov. 2000.

[129] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared-memory. In *PPoPP*, 2013.

[130] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. 2013. Under Submission.

[131] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates (poster paper). In *PPoPP*, 2013.

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

[133] J. Shun and F. Zhao. Practical parallel Lempel-Ziv factorization. In *DCC*, 2013.

[134] L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel Java Grande benchmark suite. In *ACM/IEEE SC2001*, 2001.

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

[136] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ACM ISCA*, 2000.

[137] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[138] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego vision benchmark suite. In *IEEE IISWC*, 2009.

[139] J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.

[140] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Computers*, 37(12), 1988.

[141] P. Weiner. Linear pattern matching algorithm. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[142] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[143] Altavista web page hyperlink connectivity graph. `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`, 2012.

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