Appeared in Proc. ACM-SIAM Symp. on Discrete Algorithms (1995)

Chapter 1
External-Memory Graph Algorithms

Yi-Jen Chiang*† Michael T. Goodrich‡§ Edward F. Grove¶‖ Roberto Tamassia*†
Darren Erik Vengroff*†† Jeffrey Scott Vitter¶‖‡‡

Abstract

We present a collection of new techniques for designing and analyzing efficient external-memory algorithms for graph problems and illustrate how these techniques can be applied to a wide variety of specific problems. Our results include:

- *Proximate-neighboring.* We present a simple method for deriving external-memory lower bounds via reductions from a problem we call the "proximate neighbors" problem. We use this technique to derive non-trivial lower bounds for such problems as list ranking, expression tree evaluation, and connected components.

- *PRAM simulation.* We give methods for efficiently simulating PRAM computations in external memory, even for some cases in which the PRAM algorithm is not work-optimal. We apply this to derive a number of optimal (and simple) external-memory graph algorithms.

- *Time-forward processing.* We present a general technique for evaluating circuits (or "circuit-like" computations) in external memory. We also use this in a deterministic list ranking algorithm.

- *Deterministic 3-coloring of a cycle.* We give several optimal methods for 3-coloring a cycle, which can be used as a subroutine for finding large independent sets for list ranking. Our ideas go beyond a straightforward PRAM simulation, and may be of independent interest.

- *External depth-first search.* We discuss a method for performing depth first search and solving related problems efficiently in external memory. Our technique can be used in conjunction with ideas due to Ullman and Yannakakis in order to solve graph problems involving closed semi-ring computations even when their assumption that vertices fit in main memory does not hold.

Our techniques apply to a number of problems, including list ranking, which we discuss in detail, finding Euler tours, expression-tree evaluation, centroid decomposition of a tree, least-common ancestors, minimum spanning tree verification, connected and biconnected components, minimum spanning forest, ear decomposition, topological sorting, reachability, graph drawing, and visibility representation.

1 Introduction

Graph-theoretic problems arise in many large-scale computations, including those common in object-oriented and deductive databases, VLSI design and simulation programs, and geographic information systems. Often, these problems are too large to fit into main memory, so the input/output (I/O) between main memory and external memory (such as disks) becomes a significant bottleneck. In coming years we can expect the significance of the I/O bottleneck to increase to the point that we can ill afford to ignore it, since technological advances are increasing CPU speeds at an annual rate of 40–60% while disk transfer rates are only increasing by 7–10% annually [20].

Unfortunately, the overwhelming majority of the vast literature on graph algorithms ignores this bottleneck and simply assumes that data completely fits in main memory (as in the usual RAM model). Direct applications of the techniques used in these algorithms

*Department of Computer Science, Box 1910, Brown University, Providence, RI 02912–1910.
†Supported in part by the National Science Foundation, by the U.S. Army Research Office, and by the Advanced Research Projects Agency.
‡Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218–2694
§Supported in part by the National Science Foundation under grants CCR–9003299, IRI–9116843, and CCR–9300079.
¶Department of Computer Science, Box 90129, Duke University, Durham, NC 27708–0129.
‖Supported in part by the U.S. Army Research Office under grant DAAH04–93–G–0076.
††Supported in part by the U.S. Army Research Office under grant DAAL03–91–G–0035 and by the National Science Foundation under grant DMR–9217290.
‡‡Supported in part by the National Science Foundation under grant CCR–9007851 and by the U.S. Army Research Office under grants DAAL03–91–G–0035 and DAAH04–93–G–0076.

often do not yield efficient external-memory algorithms. Our goal is to present a collection of new techniques that take the I/O bottleneck into account and lead to the design and analysis of I/O-efficient graph algorithms.

1.1 The Computational Model. In contrast to solid state random-access memory, disks have extremely long access times. In order to amortize this access time over a large amount of data, typical disks read or write large blocks of contiguous data at once. An increasingly popular approach to further increase the throughput of I/O systems is to use a number of independent devices in parallel. In order to model the behavior of I/O systems, we use the following parameters:

$$\begin{aligned} N &= \text{\# of items in the problem instance} \\ M &= \text{\# of items that can fit into main memory} \\ B &= \text{\# of items per disk block} \\ D &= \text{\# of disks in the system} \end{aligned}$$

where $M < N$ and $1 \ll DB \leq M/2$. In this paper we deal with problems defined on graphs, so we also define

$$\begin{aligned} V &= \text{\# of vertices in the input graph} \\ E &= \text{\# of edges in the input graph.} \end{aligned}$$

Note that $N = V + E$. We assume that $E \geq V$. Typical values for workstations and file servers in production today are on the order of $10^6 \leq M \leq 10^8$, $B \approx 10^3$, and $1 \leq D < 100$. Problem instances can be in the range $10^{10} \leq N \leq 10^{12}$.

Our measure of performance for external-memory algorithms is the standard notion of I/O complexity for parallel disks [26]. We define an *input/output operation* (or simply *I/O* for short) to be the process of simultaneously reading or writing D blocks of data, one to or from each of the D disks. The total amount of data transferred in an I/O is thus DB items. The I/O complexity of an algorithm is simply the number of I/Os it performs. For example, reading all of the input data will take at least N/DB I/Os, since we can read at most DB items in a single I/O. We assume that our input is initially stored in the first N/DB blocks of each of the D disks. Whenever data is stored in sorted order, we assume that it is *striped*, meaning that the data blocks are ordered across the disks rather than within them. Formally, this means that if we number from zero, the ith block of the jth disk contains the $(iDB + jB)$th through the $(iDB + (j+1)B - 1)$st items.

Our algorithms make extensive use of two fundamental primitives, *scanning* and *sorting*. We therefore introduce the following shorthand notation to represent the I/O complexity of each of these primitives:

$$scan(x) = \frac{x}{DB},$$

which represents the number of I/Os needed to read x items striped across the disks, and

$$sort(x) = \frac{x}{DB} \log_{M/B} \frac{x}{B},$$

which is proportional to the optimal number of I/Os needed to sort x items striped across the disk [19].

1.2 Previous Work. Early work in external-memory algorithms for parallel disk systems concentrated largely on fundamental problems such as sorting, matrix multiplication, and FFT [1, 19, 26]. The main focus of this early work was therefore directed at problems that involved permutation at a basic level. Indeed, just the problem of implementing various classes of permutation has been a central theme in external-memory I/O research [1, 6, 7, 8, 26].

More recently, external-memory research has moved towards solving problems that are not as directly related to the permutation problem. For example Goodrich, Tsay, Vengroff, and Vitter study a number of problems in computational geometry [12]. Further results in this area have recently been obtained in [10, 27]. There has also been some work on selected graph problems, including the investigations by Ullman and Yannakakis [23] on problems involving transitive closure computations. This work, however, restricts its attention to problem instances where the set of vertices fits into main memory but the set of edges does not. Vishkin [25] uses PRAM simulation to facilitate prefetching for various problems, but without taking blocking issues into account. Also worth noting is recent work [11] on some graph traversal problems; this work primarily addresses the problem of storing graphs, however, not in performing specific computations on them. Related work [9] proposes a framework for studying memory management problems for maintaining connectivity information and paths on graphs. Other than these papers, we do not know of any previous work on I/O-efficient graph algorithms.

1.3 Our Results. In this paper we give a number of general techniques for solving a host of graph problems in external memory:

- *Proximate-neighboring.* We derive a non-trivial lower bound for a problem we call the "proximate neighbors" problem, which is a significantly-restricted form of permutation. We use this problem to derive non-trivial lower bounds for such problems as list ranking, expression tree evaluation, and connected components.

- *PRAM simulation.* We give methods for efficiently simulating PRAM computations in external memory. We also show by example that simulating certain non-optimal parallel algorithms can yield very simple, yet I/O-optimal, external-memory algorithms.

- *Time-forward processing*—a general technique for evaluating circuits (or "circuit-like" computations) in external memory. Our method involves the use of a number of interesting external-memory data structures, and yields an efficient external-memory algorithm for deterministic list ranking.

- *Deterministic 3-coloring of a cycle*—a problem central to list ranking and symmetry breaking in graph problems. Our methods for solving it go beyond simple PRAM simulation, and may be of independent interest. In particular, we give techniques to update scattered successor and predecessor colors as needed after re-coloring a group of nodes without sorting or scanning the entire list.

- *External depth-first search.* We discuss a method for performing depth first search and solving related problems efficiently in external memory and how it can be used, in conjunction with techniques due to Ullman and Yannakakis, to solve graph problems involving closed semi-ring computations even when their assumption that vertices fit in main memory does not hold.

We apply these techniques to some fundamental problems on lists, trees, and graphs, including list ranking, finding Euler tours, expression-tree evaluation, centroid decomposition of a tree, lowest-common ancestors, minimum spanning tree verification, connected and biconnected components, minimum spanning forest, ear decomposition, topological sorting, reachability, graph drawing, and visibility representation.

2 Lower Bounds: Linear Time vs. Permutation Time

In order to derive lower bounds for the number of I/Os required to solve a given problem it is often useful to look at the complexity of the problem in terms of the permutations that may have to be performed to solve it. In an ordinary RAM, any known permutation of N items can be produced in $O(N)$ time. In an N processor PRAM, it can be done in constant time. In both cases, the work is $O(N)$, which is no more than it would take us to examine all the input. In external memory, however, it is not generally possible to perform arbitrary permutations in a linear number ($O(scan(N))$) of I/Os. Instead, it is well-known that $\Theta(perm(N))$ I/Os are required in the worst case [1, 26] where

$$perm(N) = \min\left\{\frac{N}{D}, sort(N)\right\}.$$

When M or B is extremely small, $N/D = O(B \cdot scan(N))$ may be smaller than $sort(N)$. In the case where B and D are constants, the model is reduced to an ordinary RAM, and, as expected, permutation can be performed in linear time. However, for typical values in real I/O systems, the $sort(N)$ term is smaller than the N/D term. If we consider a machine with block size $B = 10^4$ and main memory size $M = 10^8$, for example, then $sort(N) < N/D$ as long as $N < 10^{40,004}$, which is so absurdly large that even the estimated number of protons in the universe is insignificant by comparison.

We can show that the lower bound $\Omega(perm(N))$ holds even in some important cases when we are not required to perform all $N!$ possible permutations:

LEMMA 2.1. *Let A be an algorithm capable of performing $(N!)^\alpha N^c$ different permutations on an input of size N, where $0 < \alpha \leq 1$ and c are constants. Then at least one of these permutations requires $\Theta(perm(N))$ I/Os.*

Proof Sketch. The proof is an adaptation and generalization of that given by Aggarwal and Vitter [1] for the special case $\alpha = 1$ and $c = 0$. □

In order to apply the lower bound of Lemma 2.1 to graph problems, we will first use it to prove a lower bound on the *proximate neighbors problem.* In later sections, we will show how to reduce the proximate neighbors problem to a number of graph problems. The proximate neighbors problem is defined as follows: Initially, we have N items in external memory, each with a key that is a positive integer $k \leq N/2$. Exactly two items have each possible key value k. The problem is to permute the items such that, for every k, both items with key value k are in the same block. We can now lower bound the number of permutations that an algorithm that solves the proximate neighbors problems is capable of producing.

LEMMA 2.2. *Solving the proximate neighbors problem requires $\Omega(perm(N))$ I/Os in the worst case.*

Proof Sketch. We define a block permutation to be an assignment of items to blocks. The order within blocks is unimportant. There are thus $N!/(B!)^{N/B}$ block permutations of N items. We show that to solve the proximate neighbors problem an algorithm must be capable of generating

$$\frac{N!}{2^{N/2}(B!)^{N/B}(N/2)!} = \Omega\left(\frac{\sqrt{N!}}{(B!)^{N/B}N^{1/4}}\right)$$

block permutations. Thus, using an additional $scan(N)$ I/Os to rearrange the items within each block, it could produce $\sqrt{N!}/(N^{1/4})$ permutations. The claim than follows from Lemma 2.1. □

Given the lower bound for the proximate neighbors problem, we immediately have lower bounds for a number of problems it can be reduced to.

COROLLARY 2.1. *The following problems all have an I/O lower bound of $\Omega(perm(N))$: list ranking, Euler tours, expression tree evaluation, centroid decomposition of a tree, and connected components in sparse graphs ($E = O(V)$).*

Proof Sketch. All these bounds are proven using input graphs with long chains of vertices. The ability to recognize the topology of these graphs to the extent required to solve the problems mentioned requires solving the proximate neighbors problem on pairs of consecutive vertices in these chains. □

Upper bounds of $O(sort(N))$ for these problems are shown in Sections 5 and 6, giving optimal results whenever $perm(N) = \Theta(sort(N))$. As was mentioned above, this covers all practical I/O systems. The key to designing algorithms to match the lower bound of Lemma 2.2 is the fact that comparison-based sorting can also be performed in $\Theta(sort(N))$ I/Os. This suggests that in order to optimally solve a problem covered by Lemma 2.1 we can use sorting as a subroutine. Note that this strategy does not work in the ordinary RAM model, where the sorting takes $\Omega(n \log n)$ time, while many problems requiring arbitrary permutations can be solved in linear time.

3 PRAM Simulation

In this section, we present some simple techniques for designing I/O efficient algorithms based on the simulation of parallel algorithms. The most interesting result appears in Section 3.2: In order to generate I/O-optimal algorithms we resort in most cases to simulating PRAM algorithms that are not work-optimal. The PRAM algorithms we simulate typically have geometrically decreasing numbers of active processors and very small constant factors in their running times. This makes them ideal for our purposes, since the I/O simulations do not need to simulate the inactive processors, and thus we get optimal and practical I/O algorithms.

We show in subsequent sections how to combine these techniques with more sophisticated strategies to design efficient external-memory algorithms for a number of graph problems. Related work on simulating PRAM computations in external memory was done by Cormen [6]. The use of PRAM simulation for prefetching, without the important consideration of blocking, is explored by Vishkin [25].

3.1 Generic Simulation of an $O(N)$ Space PRAM Algorithm. We begin by considering how to simulate a PRAM algorithm that uses N processors and $O(N)$ space. In order to simulate such a PRAM algorithm, we first consider how to simulate a single step. This is a simple process that can be done by sorting and scanning, as shown in the following lemma.

LEMMA 3.1. *Let A be a PRAM algorithm that uses N processors and $O(N)$ space. Then a single step of A can be simulated in $O(sort(N))$ I/Os.*

Proof Sketch. Without loss of generality, we assume that each PRAM step does not have indirect memory references, since they can be removed by expanding the step into $O(1)$ steps. To simulate the PRAM memory, we keep a task array of $O(N)$ on disk in $O(scan(N))$ blocks. In a single step, each PRAM processor reads $O(1)$ operands from memory, performs some computation, and then writes $O(1)$ results to memory. To provide the operands for the simulation, we sort a copy of the contents of the PRAM memory based on the indices of the processors for which they will be operands in this step. We then scan this copy and perform the computation for each processor being simulated, and write the results to the disk as we do so. Finally, we sort the results of the computation based on the memory addresses to which the PRAM processors would store them and then scan the list and a reserved copy of memory to merge the stored values back into the memory. The whole process uses $O(1)$ scans and $O(1)$ sorts, and thus takes $O(sort(N))$ I/Os. □

To simulate an entire algorithm, we merely have to simulate all of its steps.

THEOREM 3.1. *Let A be a PRAM algorithm that uses N processors and $O(N)$ space and runs in time T. Then A can be simulated in $O(T \cdot sort(N))$ I/Os.*

It is fairly straightforward to generalize this theorem to super-linear space algorithms. There are some important special cases when we can do much better than what would be implied by Theorem 3.1, however.

3.2 Reduced Work Simulation for Geometrically Decreasing Computations. Many simple PRAM algorithms can be designed so as to have a "geometrically decreasing size" property, in that after a constant number of steps, the number of active processors has decreased by a constant factor. Such algorithms are typically not work-optimal in the PRAM sense, since all processors, active or inactive, are counted when evaluating work complexity. When simulating a PRAM with I/O, however, inactive processors do not have to be simulated. This fact can be formalized as follows:

THEOREM 3.2. *Let A be a PRAM algorithm that solves a problem of size N by using N processors and $O(N)$ space, and that after each of $O(\log N)$ stages, each of time T, both the number of active processors and the number of memory cells that will ever be used again are reduced by a constant factor. Then A can be simulated in external memory in $O(T \cdot sort(N))$ I/O operations.*

Proof. The first stage consists of T steps, each of which can, by Lemma 3.1, be simulated in $O(T \cdot sort(N))$ I/Os. Thus, the recurrence

$$\mathcal{I}(N) = O(T \cdot sort(N)) + \mathcal{I}(\alpha N)$$

characterizes the number of I/Os needed to simulate the algorithm, which is $O(T \cdot sort(N))$. $\square$

4 Time-Forward Processing

In this section we discuss a technique for evaluating the function computed by a bounded fan-in boolean circuit whose description is stored in external memory. We assume that the labels of the nodes come from a total order $<$, such that for every edge (v, w) we have $v < w$. We call a circuit in such a representation *topologically sorted.*

Thinking of vertex v as being evaluated at "time" v motivates our calling such an evaluation *time-forward* processing of the circuit. The main issue in such an evaluation, of course, is to insure that when one evaluates a particular vertex one has the values of its inputs currently in main memory.

In Section 4.1 we introduce the concept of bucketing, which will prove to be of central importance in time-forward processing. In Section 4.2 we describe the construction of a tree on time. Finally, in Section 4.3 we demonstrate how bucketing can be used to navigate a tree on time in order to solve the circuit evaluation problem. Later, in Section 5.4, we demonstrate the use of time forward processing to find large independent sets for list ranking.

4.1 Bucketing. Divide and conquer is a classic technique that is useful in the design of I/O-efficient algorithms. When trying to minimize I/O, it is usually best to try to divide into as many subproblems as possible. The maximum number of subproblems is typically M/B because you want to use at least one block for each subproblem. It often turns out that $\sqrt{M/B}$ subproblems works better when you are trying to use parallel disks. If the subproblems are of equal size, a recursion depth of $O(\log_{M/B}(N/S))$ reduces to problems of size S. For example, to sort N numbers, it suffices to $O(\sqrt{M/B}\,)$ splitters, partition the input according to the splitters, recurse and concatenate the recursively sorted lists.

In order to divide up a problem, we maintain a set of buckets which support the following operations:

1. Allocate a new bucket.
2. Add one record to a bucket.
3. Empty a bucket, placing the records in a sequential list.

The order of the the records in the list from emptying a bucket is not required to be the order in which the records were added to the bucket. Once the input is divided into buckets, each bucket is a subproblem to be solved recursively.

Of course, the bucketing problem is easy if there is only one disk: we just allocate one block of memory to each bucket and flush it to disk when it gets full. In the presence of multiple disks, however, we must be sure to guarantee that each bucket is stored roughly evenly across the parallel disks; this is the fundamental problem addressed in [19].

An overview of a possible approach is to keep one block of main memory allocated to each bucket. When that block is filled up, we flush the contents to a buffer of D blocks, and when the buffer is full, we write at least half the blocks to disk in a single I/O. Let median(b) be the median value of the number of blocks from bucket b stored on each of the D disks. We keep the buckets balanced across disks by maintaining the invariant that for every bucket b the most blocks from b on any one disk is at most one more than median(b). For each bucket b, by definition of median, at least half the disks can be written to without violating theinvariant. Thus, any set of $\lceil D/2 \rceil$ blocks can be written to a set of $\lceil D/2 \rceil$ disks in a single I/O, maintaining the invariant. The most out-of-balance any bucket b can become is to have its blocks evenly distributed on about half the disks, with no blocks on the other half of the disks. Bucket b can then be read with at most about double the optimal number of I/Os. A bucket containing g items may thus be emptied using $O(\max\{1, scan(g)\})$ I/Os. All of the reads and writes effectively use at least half the bandwidth, except when emptying a bucket containing less than $DB/2$ items. The writes are striped for error correction purposes, but the reads are not, which is needed for optimality.

THEOREM 4.1. *A series of i insertions and e empty operations on $O(M/B)$ buckets can be be performed with $O(e + scan(i))$ I/Os.*

4.2 Building a Tree on Time. Let us return, then, to the circuit-evaluation problem. Recall that we are given a topologically ordered circuit with V vertices, and we wish to compute the values of the vertices in order. Intuitively, after calculating the value of a vertex v, we send the value "forward in time" to each future time step at which it will be needed.

We split memory into two pieces of size $M/2$, one for bucketing and one for holding values needed for an interval of time. We then break up time into intervals needing a total of at most $M/2$ inputs each. For example, for a fan-in 2 circuit, each interval is of the form $[1 + jM/4, (j+1)M/4]$. We make these intervals the leaves of a balanced tree T, which we call the "time tree," so that T has branching factor f and height h, where f is a parameter of our method and $h = O((\log \#\text{ intervals})/(\log(M/2B)))$, say $2\log_{M/2B}(4V/M)$. It will turn out that about fh buckets are required, yielding constraints $fh \leq M/2B$ and $f^h \geq \#$ intervals. For example, if we choose $f = \sqrt{M/2B}$, then we require that $\sqrt{M/2B} \geq 2\log_{M/B}(4V/M)$, which is satisfied assuming

$$(\sqrt{M/2B})^{\sqrt{M/2B}} \geq 4V/M \ . \tag{4.1}$$

This assumption does not depend on the number D of parallel disks. For typical machines, M/B is in the thousands, so this is not a restrictive assumption.

4.3 Moving into the Future. We can use the time tree constructed in the previous subsection to partition time. Let us say that vertex v lies in interval s. If we remove the path from s to the root of the time tree, the tree breaks up into $(f-1)h$ subtrees, whose leaves are all of the intervals except for s. We maintain a bucket for each of these subtrees. When the value of v is computed, for each edge (v, w), we send the value of v to a bucket representing the subtree containing w, or just keep it in memory if w lies in interval s.

When the current time crosses a interval boundary, the current interval s changes, and the path up to the root changes too. As a result, the subtrees induced by removing the path from s to the root change. Each vertex that is on the new path, but was not on the old path, corresponds to a subtree that is split. The bucket corresponding to the old subtree is emptied, and the values are added to the new buckets where they belong. Any particular value is involved in at most h splits. The total number of I/O operations is $O(h \cdot scan(E))$.

This approach works for a general class of problems. The main requirement is to specify, in advance, a partition of time into $O(N/M)$ intervals, each of which uses at most $M/2$ inputs. The internal nodes can be arbitrary functions. It is not necessary to know the exact time a value will be needed. It is sufficient be able to specify the destination interval. By keeping $c = O(1)$ intervals in memory simultaneously, it suffices to send a value to within $c-1$ intervals before the time it is needed. Summing up, then, we have the following:

THEOREM 4.2. *A topologically ordered circuit with N edges can be evaluated with $O(sort(N))$ I/Os if $\sqrt{M/2B}\log(M/2B) \geq 2\log(2N/M)$.*

5 List Ranking

In this section, we demonstrate how the lower bound techniques of Section 2 and the PRAM simulation techniques of Section 3 can be put together to produce an optimal external-memory algorithm.

The problem we consider is that of list ranking. We are given an N-node linked list L stored in external memory as an (unordered) sequence of nodes, each with a pointer *next* to the successor node in the list. Our goal is to determine, for each node v of L, the *rank* of v, which we denote $rank(v)$ and define as the number of links from v to the end of the list. We assume that there is a dummy node ∞ at the end of the list, and thus the rank of the last node in the list is 1. We present algorithms that use an optimal $\Theta(sort(N))$ I/O operations. The lower bound for the problem comes from Corollary 2.1.

5.1 An Algorithmic Framework for List Ranking. Our algorithmic framework is adapted from the work of Anderson and Miller [2]. It has also been used by Cole and Vishkin [5], who developed a deterministic version of Anderson and Miller's randomized algorithm.

Initially, we assign $rank(v) = 1$ for each node v in list L. This can be done in $O(scan(N))$ I/Os. We then proceed recursively. First, we produce an independent set of $\Theta(N)$ nodes. The details of how this independent set is produced are what separate our algorithms from one another. Once we have a large independent set S, we use $O(1)$ sorts and scans to *bridge* each node v in the set, as described in [2]. We then recursively solve the problem on the remaining nodes. Finally, we use $O(1)$ sorts and scans to re-integrate the nodes in S into the final solution.

In order to analyze the I/O-complexity of an algorithm of the type just described, we first note that once the independent set has been produced, the algorithm uses $O(sort(N))$ I/Os and solves a single recursive instance of the problem. If the independent set can also be found in $O(sort(N))$ I/Os, then the total number of I/Os done in the nonrecursive parts of the algorithm is also $O(sort(N))$.

Since $\Theta(N)$ nodes are bridged out before recursion, the size of the recursive problem we are left with is at most a constant fraction of the size of our original problem. Thus, according to Theorem 3.2, the I/O-complexity of our overall algorithm is $O(sort(N))$. All that remains is to demonstrate how an independent set of size $\Theta(N)$ can be produced in $O(sort(N))$ I/Os.

5.2 Randomized Independent Set Construction. The simplest way to produce a large independent set is a randomized approach based on that first proposed by Anderson and Miller [2]. We scan along the input, flipping a fair coin for each vertex v. We then make two copies of the input, sorting one by vertex and the other by successor. Scanning down these two sorted lists in step, we produce an independent set consisting of those vertices whose coins turned up heads but whose successors coins turned up tails. The expected size of the independent set generated this way is $(N-1)/4$.

5.3 Deterministic Independent Set Construction via 3-Coloring. Our first deterministic approach relies on the fact that the problem of finding an independent set of size $\Omega(N)$ in an N-node list L can be reduced to the problem of finding a 3-coloring of the list. We equate the independent set with the $\Omega(N)$ nodes colored by the most popular of the three colors.

In this section, we describe an external-memory algorithm for 3-coloring L that performs $O(sort(N))$ I/O operations. We make the simplifying assumption here (and also in the next section) that the block size B satisfies $B = O(N/\log^{(t)} N)$ for some fixed integer $t > 0$.[1] This assumption is clearly non-restrictive in practice. Furthermore, for simplicity, we restrict the discussion to the $D = 1$ case of one disk. The load balancing issues that arise with multiple disks are handled with balancing techniques akin to [18, 26].

The 3-coloring algorithm consists of three phases. Colors and node IDs are represented by integers.

1. In this phase we construct an initial N-coloring of L by assigning a distinct color in the range $[0, \cdots, N-1]$ to each node. This phase takes $O(scan(N))$ I/Os.

2. Recall that $B = O(N/\log^{(t)} N)$ for some fixed integer $t > 0$. In this phase we produce a $(\log^{(t+1)} N)$-coloring. We omit the details in this extended abstract. The method is based upon a non-trivial adaptation of the deterministic coin tossing technique of Cole and Vishkin [5]. The total number of I/Os performed in this phase is $O(t \cdot sort(N) + (\log^{(t+1)} N)^2)$.

3. In the final phase, for each $i = 3, .., \log^{(t+1)} N - 1$, we re-color the nodes with color i by assigning them a new color in the range $[0, 1, 2]$. This phase is performed iteratively in $O(\log^{(t+1)} N + sort(N_i))$ I/Os per iteration, where N_i is the number of vertices with color i (from the previous phase). We omit the details in this extended abstract. The total number of I/Os performed in this phase is

$$\sum_{i=0}^{\log^{(t+1)} N - 1} O(\log^{(t+1)} N + sort(N_i)) = O(sort(N) + (\log^{(t+1)} N)^2).$$

The overall time complexity of the 3-coloring algorithms is thus $O(t \cdot sort(N) + (\log^{(t+1)} N)^2)$. Since t is a constant and $B = O(N/\log^{(t)} N)$, we get the following time bound:

LEMMA 5.1. *The N nodes of a list L can be 3-colored with $O(sort(N))$ I/O operations.*

Recalling the algorithmic framework for list ranking of Section 5.1, we obtain the following result:

THEOREM 5.1. *The N nodes of a list L can be ranked with optimal $O(sort(N))$ I/O operations.*

5.4 Deterministic Independent Set Computation via Time-Forward Processing. We can use time-forward processing to construct an alternate proof of Lemma 5.1 for the case when M/B is not too small (which provides an alternate condition to the constraint on B not being too large). In this case we separate the edges of the cycle into forward edges $\{(a, b) \mid a < b\}$ and backward edges $\{(a, b) \mid a > b\}$. Each of these is a set of chains. We then color the forward edges with colors 0 and 1, coloring the first vertex on a chain 0, and then alternating. We color the backward edges with 2 and 1, starting each chain with 2. If a vertex is given two different colors (because it is the beginning or end of a chain in both sets) we color it 0 unless the two colors are 1 and 2, in which case we color it 2. This gives a 3-coloring of a N-vertex cycle in $O(sort(N))$ I/Os.

We can also use time-forward traversal to compute list ranking more directly than by removing independent sets—just calculate the ranks of the vertices along the chains in the forward and backward sets, and then instead of bridging over an independent set, bridge over entire chains. We give the details in the full version.

6 Additional Applications

In this section we show that the techniques presented in Sections 2–5 can be used to solve a variety of fundamental tree and graph problems. These results are summarized in Tables 1-3. We believe that many more problems are amenable to these techniques.

Now we briefly sketch our algorithms to the problems listed. Lower bounds are similar to Corollary 2.1.

[1] The notation $\log^{(k)} N$ is defined recursively as follows: $\log^{(1)} N = \log N$, and $\log^{(i+1)} N = \log\log^{(i)} N$, for $i \geq 1$.

For expression tree evaluation, we compute the depth of each vertex by Euler Tour and list ranking, and sort the vertices first by depth and then by key such that (i) deeper nodes precede higher nodes, and (ii) the children of each node are contiguous, and (iii) the order of the nodes on each level is consistent with the ordering of their parents. We then keep pointers to the next node to be computed and to the next input value needed. The two pointers move sequentially through the list of nodes, so all of the nodes in the tree can be computed with $O(scan(N))$ additional I/Os. Centroid decomposition of a tree can be performed similarly.

The least common ancestor problem can be reduced to the range minima problem using Euler Tour and list ranking [3]. We construct a search tree S with $O(N/B)$ leaves, each a block storing B data items. Tree S is a complete (M/B)-ary tree with $O(\log_{M/B}(N/B))$ levels, where each internal node v of S corresponds to the items in the subtree S_v rooted at v. Each internal node v stores two lists maintaining prefix and suffix minima of the items in the leaves of S_v, respectively, and a third list maintaining M/B items, each a minimum of the leaf items of the subtree rooted at a child of v. The K batched queries are performed by sorting them first, so that all queries can be performed by scanning S $O(1)$ times. If $K > N$ we process the queries in batches of N at a time.

For the minimum spanning tree (MST) verification problem, our technique is based on that of King [14]. We verify that a given tree T is an MST of a graph G by verifying that each edge (u, v) in G has weight at least as large as that of the heaviest edge on the path from u to v in T. First, using $O(sort(V))$ I/Os, we convert T into a balanced tree T' of size $O(V)$ such that the weight of the heaviest edge on the path from u to v in T' is equal to the weight of the heaviest edge on the path from u to v in T. We then compute the lowest common ancestor in T' of the endpoints of each edge of G. Using the technique described above to process the pairs V at a time, this takes $O((E/V)sort(V))$ I/Os. Finally, we construct tuples consisting of the edges of G, their weights and the lowest common ancestors of their endpoints, and, using the batch filtering technique of [12], we filter these tuples through T', V at a time. This batch filtering takes $O((E/V)sort(V))$ I/Os. When a tuple hits the lowest common ancestor of the endpoints of its edge, it splits into two queries, one continuing on towards each of its endpoints. If, during subsequent filtering, a query passes through an edge whose weight is less than its own, the algorithm can stop immediately and report that T is not an MST of G. If this never happens, then T is an MST.

For connected components and minimum spanning forest, our algorithm is based on that of Chin *et al.* [4]. Each iteration performs a constant number of sorts on current edges and one list ranking to reduce the number of vertices by a constant factor. After $O(\log(V/M))$ iterations we fit the remaining M vertices to the main memory and solve the problem easily. For biconnected components, we adapt the PRAM algorithm of Tarjan and Vishkin [22], which requires generating an arbitrary spanning tree, evaluating an expression tree, and computing connected components of a newly created graph. For ear decomposition, we modify the PRAM algorithm of Maon *et al.* [17], which requires generating an arbitrary spanning tree, performing batched lowest common ancestor queries, and evaluating an expression tree. Note that all these problems can be solved within the bound of computing minimum spanning forest. Our randomized algorithm reduces this latter bound by decreasing in each iteration the numbers of *both* edges and vertices by a constant factor, using an external-memory variation of the random sampling technique by [13, 15] and the previously mentioned minimum spanning tree verification method.

Planar *st*-graphs were first introduced by Lempel, Even, and Cederbaum [16], and have a variety of applications in Computational Geometry, motion planning, and VLSI layout. We obtain the given upper bounds by modifying the PRAM algorithms of Tamassia and Vitter [21], and applying the list ranking and the PRAM simulation techniques.

7 Depth First Search and Closed Semi-Ring Computation

Many algorithms for problems on directed graphs are easily solved in main memory by depth first search (DFS). We analyze the performance of sequential DFS, modifying the algorithm to reprocess the graph when the number of visited vertices exceeds $\Theta(M)$. We present a graph with V vertices and E edges by three arrays. There is a size-E array A containing the edges, sorted by source. Size V arrays $Start[i]$ and $Stop[i]$ denote the range of the adjacency list of i. Vertex i points to vertices $\{A[j] \mid Start[i] \leq j \leq Stop[i]\}$.

DFS maintains a stack of vertices corresponding to the path from the root to the current vertex in the DFS tree. The pop and push operations needed for a stack are easily implemented optimally in I/Os. For each current vertex, examine the incident edges in the order given on the adjacency list. When a vertex is first encountered, it is added to a search structure, put on the stack, and made the current vertex. Each edge read is discarded. When an adjacency list is exhausted, pop the stack and retreat the path one vertex.

Problem	Notes	Lower Bound	Upper Bound
Euler Tour		$\Omega(sort(N))$	$O(sort(N))$
Expression Tree Evaluation	Bounded Degree Operators	$\Omega(sort(N))$	$O(sort(N))$
Centroid Decomposition		$\Omega(sort(N))$	$O(sort(N))$
Least Common Ancestor	K Queries		$O((1+K/N)sort(N))$

Table 1: I/O-efficient algorithms for problems on trees. The problem size is $N = V = E + 1$.

Problem	Notes	Lower Bound	Upper Bound
Minimum Spanning Tree Verification			$O((E/V)sort(V))$
Connected Components, Biconnected Components, Minimum Spanning Forest, and Ear Decomposition			$O(\min\{sort(V^2), \log(V/M)\cdot sort(E)\})$
	Sparse graphs ($E = O(V)$) closed under edge contraction	$\Omega(sort(V))$	$O(sort(V))$
	Randomized, with probability $1-\exp(-E/\log^{O(1)} E)$		$O((E/V)sort(V))$

Table 2: I/O-efficient algorithms for problems on undirected graphs.

The only problem arises when the search structure holding visited vertices exceeds the memory available. When that happens, we make a pass through all of the edges, discarding all edges that point to vertices already visited, and compacting so that all of the edges in each adjacency list are consecutive. Then we empty out the search structure and continue.

The algorithm must perform $O(1)$ I/Os every time a vertex is made the current vertex. This can only happen $2V$ times, since each such I/O is due to a pop or to a push. The total additional number of I/Os due to reading edge lists is $O(scan(E) + V)$. The search structure fills up memory at most $O(V/M)$ times. Each time the search structure is emptied, $O(scan(E))$ I/Os are performed.

THEOREM 7.1. *Let G be a directed graph containing V vertices and E edges in which the edges are given in a list that is sorted by source. DFS can be performed on G with $O((1 + V/M)scan(E) + V)$ I/Os.*

COROLLARY 7.1. *Let G be a directed graph containing V vertices and E edges in which the edges are given in a list that is sorted by source. Then one can compute the strongly connected components of G and perform a topological sorting on the strongly connected components using $O((1 + V/M)scan(E) + V)$ I/Os.*

Ullman and Yannakakis have recently presented external-memory techniques for computing the transitive closure of a directed graph [23]. They solve this problem using $O(dfs(V, E) + scan(V^2\sqrt{E/M}))$ I/Os, where $dfs(V, E)$ is the number of I/Os needed to perform DFS on the input graph in order to find strongly connected components and topologically sort it. They assume that $V < M$, and under that assumption, they give an $O(scan(E) + V)$ algorithm for DFS. In their other routines, small modifications to the algorithms allow for full blocking even when $V > M$. Our DFS algorithm works for the general case when $V > M$, and its I/O complexity is always less than the $scan(V^2\sqrt{E/M})$ term in complexity of transitive closure. Thus, we get the following corollary to Corollary 7.1 and the work of Ullman and Yannakakis:

COROLLARY 7.2. *The transitive closure of a graph can be computed in $O(scan(V^2\sqrt{E/M}))$ I/Os.*

8 Conclusions

We have presented a number of techniques for designing and analyzing external-memory algorithms for graph theoretic problems and showed a number of applications for them. Our techniques, particularly proximate neighbors problem lower bounding, derivation of I/O-optimal algorithms from non-optimal PRAM algorithms, and time-forward processing, are general enough that they are likely to be of value in other domains as well. Applications to memory hierarchies and parallel memory hierarchies will be discussed in the full paper.

Although we did not specifically discuss them, the constants hidden in the big-oh notation tend to be small for algorithms based on our techniques. For example, randomized list ranking can be done using 3 sorts per recursive level, which leads to an overall I/O complexity roughly 12 times that required to sort the original input a single time. An implementation along these

Problem	Notes	Lower Bound	Upper Bound
Reachability	K queries		$O((1+K/V)sort(V))$
Topological Sorting		$\Omega(sort(V))$	$O(sort(V))$
Drawing and, Visibility Representation	$2V-5$ bends $O(V^2)$ area	$\Omega(sort(V))$	$O(sort(V))$

Table 3: I/O-efficient algorithms for problems on planar st-graphs. Note that $E = O(V)$ for these graphs.

lines has been written using an alpha version of TPIE, a transparent parallel I/O environment designed to facilitate the implementation of I/O efficient algorithms from a variety of domains [24]. We expect to implement additional algorithms using TPIE and publish empirical results regarding their efficiency in the near future.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Info. Proc. Letters*, 33(5):269–273, 1990.

[3] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. Technical report, Institue for Advanced Computer Studies, Univ. of Maryland, College Park, 1990.

[4] F. Y. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Comm. of the ACM*, 25(9):659–665, 1982.

[5] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal list-ranking. *Information and Control*, 70(1):32–53, 1986.

[6] T. H. Cormen. *Virtual Memory for Data Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[7] T. H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, Jan./Feb. 1993.

[8] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Dept. of Computer Science, July 1994.

[9] E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. In *Proc. Int. Symp. on Algorithms and Comp.*, 1993.

[10] P. G. Franciosa and M. Talamo. Orders, implicit k-sets representation and fast halfplane searching. In *Proc. Workshop on Orders, Algorithms and Applications (ORDAL'94)*, pages 117–127, 1994.

[11] M. T. Goodrich, M. H. Nodine, and J. S. Vitter. Blocking for external graph searching. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Sys.*, pages 222–232, 1993.

[12] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.

[13] D. R. Karger. Global min-cuts in RNC and other ramifications of a simple mincut algorithm. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 21–30, 1993.

[14] V. King. A simpler minimum spanning tree verification algorithm, 1994.

[15] P. Klein and R. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proc. ACM Symp. on Theory of Computing*, 1994.

[16] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs, Int. Symp. (Rome, 1966)*, pages 215–232. Gordon and Breach, New York, 1967.

[17] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search and st-numbering in graphs. *Theoretical Computer Science*, 47(3):277–296, 1986.

[18] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, June 1993.

[19] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, Jan. 1993.

[20] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Comp.*, 27(3):17–28, Mar. 1994.

[21] R. Tamassia and J. S. Vitter. Optimal cooperative search in fractional cascaded data structures. In *Proc. 2nd ACM Symosium on Parallel Algorithms and Architectures*, pages 307–316, 1990.

[22] R. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM J. Computing*, 14(4):862–874, 1985.

[23] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3:331–360, 1991.

[24] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, July 1994.

[25] U. Vishkin. Personal communication, 1992.

[26] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2), 1994.

[27] B. Zhu. Further computational geometry in secondary memory. In *Proc. Int. Symp. on Algorithms and Computation*, 1994.