Graph Algorithms for Multicores with Multilevel Caches

Brandon Blakeley

May 17, 2009

Adviser: Professor Vijaya Ramachandran

Abstract

Historically, the primary model of computation employed in the design and analysis of algorithms has been the sequential RAM model. However, recent developments in computer architecture have reduced the efficacy of the sequential RAM model for algorithmic development. In response, theoretical computer scientists have developed models of computation which better reflect these modern architectures. In this project, we consider a variety of graph problems on parallel, cache-efficient, and multicore models of computation. We introduce each model by defining the analysis of algorithms on these models. Then, for each model, we present current results for the problems of prefix sums, list ranking, various tree problems, connected components, and minimum spanning tree. Finally, we present our novel results, which include the multicore oblivious extension of current results on a private cache multicore model to a more general multilevel multicore model.

Contents

1 Introduction

Historically, the primary model of computation employed in the design and analysis of algorithms has been the sequential RAM model. However, recent developments in computer architecture have reduced the efficacy of the sequential RAM model for algorithmic development. In response, theoretical computer scientists have developed models of computation which better reflect these recent architectural developments. In this thesis, we consider primarily the problems of computing the prefix sums, of ranking a list, and some related graph problems in the parallel, cache efficient, and multicore models of computation.

1.1 Problems Considered

We start by defining the problems under consideration in this thesis.

Consider a sequence $(x_1, x_2, \ldots, x_n)$ of n elements taken from a set G with a binary operation denoted by $+$. We define the i^{th} partial sum s_i of such a sequence to be $s_i = x_1 + x_2 + \cdots + x_i$. The **prefix sums** of such a sequence, originally posed by Ladner and Fischer [20], is the n partial sums of the sequence. There is a trivial sequential algorithm which recursively computes s_i using the fact that $s_i = s_{i-1} + x_i$ for $2 \leq i \leq n$ and hence takes $\mathcal{O}(n)$ time.

Consider a linked list L of n nodes. We define the rank $r(i)$ of a node i to be its distance from the end of the list. The **list ranking** problem, originally posed by Wyllie in [22], is to determine the ranks of every node in a list. The linked list L is typically represented by a successor array S where $S(i)$ contains a pointer to the next node following node i in L. We additionally assume that $S(t) = t$ for the last node t in L and for no node i does $S(i) = s$, for some node s, the first node. Note that $r(t) = 0$ and that $r(1) = n$. The challenge in solving this problem results from the fact that consecutive nodes are not necessarily in adjacent memory addresses. Again, there is a trivial sequential algorithm which recursively computes $r(i) = r(S(i)) + 1$ for $i \neq t$ and hence takes $\mathcal{O}(n)$ time.

We will sometimes use the language of graph theory to describe a linked list L on n nodes as graph $G = (V, E)$ where $V = \{1, 2, \ldots, n\}$ and $E = \{(i, S(i)) | 1 \leq i \leq n\}$. Additionally, there is a weighted version of this problem where each edge has an associated weight and the rank of a node is defined to be the sum of the weights of all edges along the path from that node to the last node.

Additionally, we consider a number of problems on trees. Let $T = (V, E)$ be a tree. Let $T' = (V, E')$ be the directed graph constructed by replacing every edge $(u, v) \in E$ with the two directed arcs (u, v) and (v, u); that is, $E' = \{(u, v), (v, u) | (u, v) \in E\}$. We define an Euler circuit on T to be a directed circuit that traverses each arc exactly once. Because the indegree of each vertex equals its outdegree, every graph constructed this way will have such an Euler circuit. We call the Euler circuit of T' the Euler tour of T. The Euler tour technique can be used for the optimal computation of many problems on trees; in a sense, it is a highly parallel alternative to depth first search. Given a tree and a designated vertex $r \in V$, a function $p : V \to V$ **roots** a tree at r if for each node $v \neq r$, $p(v)$ is the next node on the unique path from v to r. Given a tree T rooted at r, the **preorder** of a node is the order in which that node is visited in a depth first search of T. The **depth** of a vertex v is the distance between v and r. The subtree size at v is the number of nodes below v in T;

that is, the number of nodes whose path to r contains v. For each of these values we define an associated problem for which we must compute that value for each vertex.

Finally, we consider two more problems on graphs. Given a graph $G = (V, E)$, two vertices u and v are said to be connected if there is a path from u to v in G. The problem of computing the **connected components** of a graph is to compute a partition $D : V \to V$ such that $D(u) = D(v)$ if and only if u is connected to v for each $u, v \in V$.

We say that a subset $T \subseteq E$ of the edges of a connected graph $G = (V, E)$ is a spanning tree if (V, T) is a tree and every pair of vertices is connected through T. If G has an associated weight function $w : E \to \mathbb{R}$, then we define the weight $w(T)$ of a spanning tree to be the sum of the weights of each edge in T. We define a tree to be a **minimum spanning tree** if it is a spanning tree with weight at most that of any other spanning tree.

1.2 Overview

In the next three sections, we review the parallel, cache-efficient, and multicore models of computation, define the analysis of algorithms on these models, and then describe algorithms solving the previously defined problems. For the parallel model, we first describe the optimal prefix-sums algorithm and then three list-ranking algorithms, leading up to the optimal algorithm. We then describe how the Euler tour technique can be applied to solve tree problems optimally. We end this section by describing the best known parallel algorithms for the connected components and minimum spanning tree problems. For the cache-efficient model, we describe the list-ranking algorithm employing a cache-oblivious queue, and note that the sequential algorithm for prefix-sums is already cache-efficient. We then describe previous results which adapt parallel algorithms solving graph problems to the cache efficient context. Finally, we review two multicore models of computation (private cache and multilevel multicore) and describe current results for these models. Also of interest are the cache oblivious and multicore oblvious algorithms which achieve good time and cache performance without explicitly referencing any machine parameters.

In the following section, we present our novel results. In particular, we show that the PRAM prefix-sums algorithm (with a reasonable scheduler) is also multicore efficient (and so, multicore oblivious). Next, we extend the list ranking and tree problems results of Arge et al. [3] from the private external memory multicore model to the multilevel multicore model. Finally, we present algorithms solving the connected components and minimum spanning tree problems on the multilevel multicore model. One of our main contributions is to obtain multicore oblivious algorithms for these graph problems.

In the next section, we remark briefly on the challenges that remain in creating an optimal multicore algorithm for list ranking, and what kind of limitations we might expect in eliminating certain assumptions relating multicore parameters.

2 Parallel Algorithms

2.1 Parallel Models

Algorithm design has historically centered around the sequential model with unit access time to memory. However, certain niche and scientific computing applications have introduced

parallelism as a tool for faster computation. In response, theoretical computer scientists have developed the PRAM (parallel random access memory) model [19, 18] as a compromise between simplicity and realism for modeling parallel computation.

In the PRAM model, we have multiple processors which share access to a main memory and we are interested in the number of parallel steps required to solve a problem instance as a function of the size of the instance and the number of processors. However, there is the possibility of simultaneous access of several processors to the same memory location; variations on the PRAM model address this issue. The **exclusive read exclusive write** (EREW) PRAM forbids simultaneous access to the same memory location. The **concurrent read exclusive write** (CREW) PRAM allows concurrent reads but not concurrent writes. The **concurrent read concurrent write** (CRCW) PRAM allows both concurrent reads and writes; again, variations on this model address how write conflicts are resolved. It is worth noting that, while the CRCW is strictly more powerful than the CREW and the CREW is strictly more powerful than the EREW, these variants do not differ substantially in their computational speed.

2.2 Performance of Parallel Algorithms

Our analysis of parallel algorithms will consider a number of dimensions. Let $\mathcal{Q}$ be a problem which runs in time $T(n)$ with $P(n)$ processors on a PRAM model for an instance of size n. Then we define $C(n) = T(n)P(n)$ to be the **cost** of the parallel algorithm. Furthermore, we define the **work** performed by a parallel algorithm to be the total number of operations used. If we denote by $T^*(n)$ the inherent sequential time complexity of $\mathcal{Q}$, then a sequential algorithm whose running time is $\mathcal{O}(T^*(n))$ is called time optimal. A parallel algorithm to solve $\mathcal{Q}$ is said to be **optimal** if the work $W(n)$ of the algorithm satisfies $\Theta(T^*(n))$. Furthermore, an optimal parallel algorithm is said to be **work-time optimal** if its time requirement cannot be improved by any other optimal parallel algorithm.

2.3 Prefix Sums

The following algorithm, due to Ladner and Fischer [20], employs the balanced binary tree technique to compute prefix sums in $\mathcal{O}(n/p)$ time with $p \leq n/\log n$ processors on an EREW PRAM.

Recursive Prefix Sums
if $n = 1$ **then** $s_1 := x_1$; **return**
pfor $i = 1, \ldots, n/2$
  $y_i := x_{2i-1} + x_{2i}$
Recursively compute the prefix sums $z_1, z_2, \ldots, z_{n/2}$ of $(y_1, y_2, \ldots, y_{n/2})$.
pfor $1 \leq i \leq n$
  if $i = 1$ **then** $s_i := x_1$
  else if i is even **then** $s_i := z_{i/2}$
  else $s_i := z_{(i-1)/2} + x_i$
end

The correctness of this algorithm can be proven by induction on k, where the size of the input $n = 2^k$, though we omit the details of the proof as they are primarily arithmetic.

We now analyze the complexity of this algorithm. The base case clearly takes time and work $\mathcal{O}(1)$, and every nonrecursive step reduces the input size by half and takes time $\mathcal{O}(n/p)$ where $p \leq n$ using work $\mathcal{O}(n)$. Thus, the time complexity of this algorithm satisfies $T(n) = T(n/2) + \mathcal{O}(n/p)$ for $p \leq n$ and $T(p) = \mathcal{O}(\log p)$ while the work required by this algorithm satisfies $W(n) = W(n/2) + \mathcal{O}(n)$ where $W(1) = \mathcal{O}(1)$. The solutions to these recurrences are therefore $T(n) = \mathcal{O}(n/p) + \mathcal{O}(\log p)$ and $W(n) = \mathcal{O}(n)$. This algorithm is therefore optimal, and whenever $n/\log n \leq p \leq n$, this algorithm takes time $\mathcal{O}(\log n)$, which matches a known lower bound.

Finally, we remark that another approach to parallel computation is to consider the computation DAG (directed acyclic graph). A computation DAG is a model of the execution of an algorithm: nodes represent subtasks for a computation, while directed edges represent dependencies among subtasks. Here, the computation DAG for prefix-sums is a balanced binary tree (as previously remarked), where internal nodes represent the sum of their children. We can thus view this algorithm as continually computing the sums of any $2p$ children of greatest depth. This provides an alternate proof of the time complexity as $\mathcal{O}(N/p + \log p)$.

2.4 List Ranking

While the list ranking problem has a trivial sequential algorithm, designing an optimal parallel algorithm is more challenging. We will thus describe the well known optimal algorithm incrementally, describing the ideas which will be used for the accelerated cascading technique.

2.4.1 A Simple Algorithm

The following parallel algorithm, due to Wyllie [22], employs the pointer jumping technique to compute list rankings in $\mathcal{O}(n \log n/p)$ time with $p \leq n$ processors on an CREW PRAM.

List Ranking by Pointer Jumping
pfor $i = 1, \ldots, n$
  $\sigma(i) := S(i)$.
  if $S(i) = i$ **then** $\Delta(i) := 0$
    else $\Delta(i) := 1$.
for $\lceil \log n \rceil$ iterations **repeat**
  pfor $i = 1, \ldots, n$
    $\Delta(i) := \Delta(i) + \Delta(S(i))$
    $S(i) := S(S(i))$
output $\Delta(i)$

To complexity of this algorithm is clear. To argue correctness, let $r(i)$ denote the *rank* of the element in location i. Correctness follows from the invariant that at the start of iteration l, we have that $S(i) = \sigma^{2^{l-1}}(i)$ and $\Delta(i) = r(i) - r(S(i))$. Thus, after $\lceil \log n \rceil$ iterations, $S(i) = \sigma^n(i)$ and so $r(S(i)) = 0$, implying that $\Delta(i) = r(i)$.

2.4.2 A Linear Work Algorithm

Note that the work performed by this algorithm is $\Theta(n \log n)$. Because there exists a linear-time sequential algorithm, this parallel algorithm is not optimal. However, this algorithm can be made optimal if we could somehow in time $\mathcal{O}(n/p)$ and linear work decrease the size of the list to $\mathcal{O}(n/\log n)$. In fact, we can do so by recursively constructing a reduced list by contracting the nodes of an *independent set* of size at least n/c for some $c > 1$, using ideas due to Cole and Vishkin [14]. We first describe their algorithm abstractly, and then discuss how to construct an independent set using nearly constant time and linear work.

List Ranking by Contraction
(1) Create reduced list by contracting an independent set.
(2) Recursively solve the list ranking problem for the reduced list.
(3) Extend the solution to the contracted nodes.

They conclude the recursion by solving the problem in time $\mathcal{O}(n/p)$ using the List Ranking with Pointer Jumping algorithm once the contracted list is of size $n/\log n$. They show how to construct in time $\mathcal{O}(n/p)$ an independent set of size n with $c > 1$, and so can recursively contract the list to length at most $n/\log n$ using $\mathcal{O}(\log \log n)$ contractions. They can extend the solution to each of the n' contracted nodes i in parallel by setting $\Delta(i) = \Delta(S(i)) + 1$; this uses $\mathcal{O}(n'/p)$ time with $p \leq n'/\log n'$. So the total running time is $\mathcal{O}(n \log \log n/p)$.

We now describe an algorithm from [13] to determine an independent set without randomization in almost constant time. We find an independent set by k-coloring (with integers) the nodes of the linked list and selecting all nodes u such that the color of u is less than the colors of both its predecessor and its successor. Note that, between any two consecutive local minima u and v is an increasing sequence followed by a decreasing sequence; therefore, any two consecutive local minima have at most $2k - 3$ nodes between them, and so this independent set is of size $\Omega(n/k)$. Specifically, we show how to construct a 4-coloring c for this directed path, obtaining an independent set of size cn for some $0 < c < 1$.

We begin by initializing in parallel $c(l) = l$ for all l. Next, we iteratively decrease the number of colors by finding a new coloring c' with the following algorithm.

Iterative Coloring
pfor $i = 1, \ldots, n$
  Set k_i to the least significant bit position at which $c(i)$ and $c(S(i))$ disagree.
  Set $c'(i) := 2k_i + c(i)_{k_i}$

First, note that because c is a coloring, every $c(i)$ disagrees with $c(S(i))$ at some k. Next, suppose for contradiction that $c'(i) = c'(j)$ for some i, j where $S(i) = j$. Then for the uniquely determined k, l in the algorithm, $c'(i) = 2k + c(i)_k$ and $c'(j) = 2l + c(j)_l$. Because $c'(i) = c'(j)$, we have that $k = l$ (since the two additive terms influence disjoint bit positions). This implies that $c(i)_k = c(j)_k$, contradicting the definition of k. Therefore, $c'(i) \neq c'(j)$ whenever $S(i) = j$. So c' is a valid coloring.

We now establish a bound on the number of colors in c' as a function of the number in c. Let $t \geq 4$ be the number of bits used to represent each of the colors in c. Then the

least significant bit in which two colors disagree is at most $k = t$. So every color in c' is at most $2t + 1$. Thus, each color in c' can be represented with at most $\lceil \log_2 t \rceil + 1$ bits. Therefore, if the number of colors q in c satisfies $2^{t-1} < q \leq 2^t$, the the number of colors in c' is $2^{\lceil \log_2 t \rceil + 1} = \mathcal{O}(t) = \mathcal{O}(\log q)$. Hence, the number of colors decreases exponentially with each iteration; that is, $|c'| \in \mathcal{O}(\log |c|)$.

Beginning with the initial coloring $c(l) = l$ of n colors, after iteration i the number of colors in our coloring is $\mathcal{O}(\log^{(i)} n)$. Therefore, after $\mathcal{O}(\log^* n)$ iterations, our coloring will use at most 8 colors.

Hence, by contracting out as previously described $\mathcal{O}(\log \log n)$ times the nodes in the independent set determined by $\mathcal{O}(\log^* n)$ applications of ITERATIVE COLORING, we can solve the list ranking problem in time $\mathcal{O}(\frac{n}{p} \log \log n \log^* n)$ for $p \leq n \log^* n / \log n$ using the LIST RANKING BY CONTRACTION technique. However, as we have already seen an algorithm which uses $\mathcal{O}(\log n)$ time, this algorithm is not work-time optimal.

2.4.3 An Optimal Logarithmic Time Algorithm

The shortcoming of the algorithm presented in the previous section comes from the expensive contraction operations. In this section, we present a method, due to Anderson and Miller [2], to shrink the list of n nodes to $\mathcal{O}(n/\log n)$ nodes without the need to recursively contract out nodes in the list.

At a high level, their strategy is to divide the array S into $n/\log n$ contiguous subarrays $\{B_i\}$, called *blocks*, each containing $\log n$ items. An index $p(i)$ points to a node in B_i, denoted by $N(p(i))$. Initially, $p(i)$ points to the first node in each block B_i. They label each initial $N(p(i))$ as *active* and the rest as *inactive*. They call an active node *isolated* if neither its predecessor nor successor is active.

Their goal is to contract a suitably sized independent set. The contraction procedure consists of $\mathcal{O}(\log n)$ stages, each of which runs in $\mathcal{O}(n/p \log n)$ time using $\mathcal{O}(n/\log n)$ operations.

In the first stage, they remove each isolated node $N(p(i))$ from the list. Now, each active node is part of a sublist. We show how to break each sublist into short *chains* in $\mathcal{O}(1)$ time using a linear number of operations. They apply ITERATIVE COLORING to the remaining $\mathcal{O}(n/\log n)$ active nodes, resulting in a valid coloring with $\mathcal{O}(\log \log n)$ colors. Each node whose color is a local minimum is designated as a *ruler*, while the rest are labeled as subjects. Thus, the rulers partition the sublists into chains of length $\mathcal{O}(\log \log n)$. They conclude the first stage by incrementing the pointer of each subject node and each removed node, labeling these corresponding new nodes as active.

In the general case, they begin with at most $p \leq n/\log n$ pointers $p(i)$, each pointing to an element in block B_i. Each node $N(p(i))$ is labeled either active or a ruler. For each ruler $N(p(i))$, the next subject given by the original successor relation is removed and labeled removed. If the removed node was its last subject, $N(p(i))$ is labeled active. They then proceed as in the first stage by removing isolated nodes and identifying rulers and subjects again, concluding by advancing the pointers of the removed and subject nodes and labeling the new nodes as active.

The following describes the operation of processor P_i on block B_i at any general iteration.

Parallel Contraction
(1) If $N(p(i))$ is a ruler, remove the next subject. Label $N(p(i))$ as active if this was the last subject in its chain
(2) If $N(p(i))$ is active and isolated, remove $N(p(i))$
(3) If $N(p(i))$ is active, recolor $N(p(i))$ twice and label either subject or ruler.
(4) If $N(p(i))$ was removed or labeled subject, increment $p(i)$. If $p(i)$ leaves B_i, exit. Otherwise, label $N(p(i))$ as active.

Observe that each pointer is updated at most $\mathcal{O}(\log n)$ times. Thus, assuming we can complete each iteration in $\mathcal{O}(1)$ time, each of the p processors finishes in time $\mathcal{O}(n/p)$ and so our total work is $\mathcal{O}(n)$, which is optimal. All that will remain to show is that at termination the number of remaining nodes is $\mathcal{O}(n/\log n)$. We first verify that each step can be completed in $\mathcal{O}(1)$ time.

In steps (1) and (2), we only check the labels and change the labels of a constant number of nodes, which requires $O(1)$ time. In step (4), we additionally only increment a pointer value. Some care must be taken to describe step (3). Specifically, *all* active nodes are recolored. Note that we assign a pointer to each processor, and every active node has a pointer to it. Thus, each processor can apply in parallel the coloring function $c'(i) := 2k_i + c(i)_{k_i}$ where k_i is the least significant bit at which i and $S(i)$ disagree in $O(1)$ time, and this is done twice. Furthermore, we can decide in $O(1)$ time whether $N(p(i))$ is a ruler by comparing its color with that of its predecessor and that of its successor. Thus, each processor in step (3) takes $O(1)$ time per iteration.

We now prove that the contracted list has size at most $n/\log n$ after $\mathcal{O}(\log n)$ iterations. Because nodes in each block are processed at different rates, our analysis depends on a scheme which accounts for the removal of nodes in an amortized sense.

Let $q = 1/\log\log n$ and assign each of the $0 \le i < \log n$ nodes in a block $(1-q)^i$ where i is the distance of the node from the top of the block. Thus, the total weight of each block is $\Sigma_{0\le i<\log n}(1-q)^i < 1/q$ and the total weight of all items is thus less than $n/q\log n$. We will thus show that after $5\log n$ iterations, the weight of the remaining elements is at most $(n/\log n)(1-q)^{\log n}$, which implies that at most $n/\log n$ elements remain as the smallest weight of any element is $(1-q)^{\log n-1}$.

Lemma 2.1. *After each iteration, the total weight of the elements that have not been removed is reduced by at least a factor of at least* $(1-\frac{q}{4})$.

Proof. We distribute the total remaining weight in the list as follows. With each block B_i, we associate the elements remaining in B_i and also the subjects of $N(p(i))$ if $N(p(i))$ is a ruler. Note that at any stage, each remaining node is associated with exactly one block.

We have three cases to consider.

1. *An active node is removed.* If the node removed is the i^{th} node, then the weight of the block is reduced from $\Sigma_{i\le j<\log n}(1-q)^j$ to $\Sigma_{i+1\le j<\log n}(1-q)^j$, which is less than $(1-\frac{q}{4})\Sigma_{i\le j<\log n}(1-q)^j$.
2. *An active node is labeled as a subject.* We consider the chain containing this node and all associated blocks. Will will account for half of the weight when the node becomes

a subject and the other half when the subject is removed. Suppose the weight of the ruler is $(1-q)^{i_1}$ and the weights of the subjects are $(1-q)^{i_j}$ with $2 \leq j \leq k$ for some k. Furthermore, we assume that i_1 is the largest weight (because otherwise we could reassign the weights from the start). The total weight Q of all blocks associated with this chain is $\Sigma_{j=1}^{k}\Sigma_{i_j \leq l < \log n}(1-q)^l$, and the labeling of this node as subject reduces the weight to $Q - \frac{1}{2}\Sigma_{j=2}^{k}(1-q)^{i_j} \leq (1-\frac{q}{4})Q$.

3. *The current node is labeled ruler and removes a subject.* Let the ruler have weight $(1-q)^{i_1}$ and its subjects have weights $(1-q)^{i_j}$ where $2 \leq j \leq k$ for some k. Again, assume the element of heaviest weight (say, i_2) is removed. Then the total weight of the block and subjects of the ruler is $Q = \Sigma_{i_1 \leq l < \log n}(1-q)^l + \frac{1}{2}\Sigma_{2 \leq j \leq k}(1-q)^{i_j} \leq (1-\frac{q}{4})Q$.

Thus, after each iteration, the total weight is reduced by a factor of at least $(1-\frac{q}{4})$. □

Therefore, after $5\log n$ iterations, the total weight is at most $(n/q\log n)(1-\frac{q}{4})^{5\log n}$, which, for sufficiently large n, is less than $(n/\log n)(1-q)^{\log n}$. As the smallest weight is $(1-q)^{\log n - 1}$, there are $\mathcal{O}(n/\log n)$ elements after $\mathcal{O}(\log n)$ iterations.

As each of the $\mathcal{O}(\log n)$ iterations runs in time $\mathcal{O}(n/p \log n)$, the total running time of this routine is $\mathcal{O}(n/p)$, with linear work.

2.5 Euler Tours and Tree Problems

In this section, we first explain how to efficiently obtain such an Euler tour in parallel, and then describe various optimal algorithms for tree problems using the Euler tour technique.

2.5.1 Euler Tours

We specify an Euler tour of $T = (V, E')$ by defining a successor function S mapping each arc to the next arc along the circuit. One way to do this, described by Tarjan and Vishkin [21], is to fix a cyclic ordering on V and the sucessor of each arc (u, v) is (v, w) where w is the node following u in the cyclic ordering of the adjacency of v. We show that this defines an Euler tour (instead of a set of arc-disjoint cycles) by induction on $|V| = n$. The base case of $n = 2$ is trivial. Next, let $n > 2$ and consider some leaf node u adjacent to v. Then, by definition of our function S, the successor of (v, u) is (u, v). We can thus remove u from the graph and our inductive hypothesis guarantees that the successor function S defines an Euler tour on the resulting tree. Thus, the full range of S defines an Euler tour on T'.

Algorithmically constructing this function is straightforward. For each node v, we assume that the ordering on the set of nodes adjacent to v is simply the order in which these nodes appear in the adjacency list of v. Then for each edge (u_i, v), we can identify the successor $(v, u_{i'})$ as $u_{i'}$ follows u_i in the adjacency list, except when u_i is the last node in the adjacency list of v. We can fix this by making the adjacency list circular. Thus, for each node u_i in a given adjacency list of v, we can define the successor $(v, u_{i'})$ of (u, v) in constant time. Thus, as we have $2n-2$ such edges to define, we can construct this function in time $\mathcal{O}(n/p)$ for any $p \leq n$.

We now show how, by list ranking an Euler tour with cleverly defined edge weights, we can solve particular tree problems.

2.5.2 Rooting a Tree

Given a tree $T = (V, E)$ and a vertex $r \in V$, we can root a tree at r as follows. We construct an Euler tour S starting at r, but breaking the edge from the last node u on the adjacency list of r and setting $S((u, r)) = u$. We then compute the list ranking of the list of arcs defined by S. Finally, for each arc (x, y), we set $p(y) = x$ whenever the ranking of (x, y) is smaller than that of (y, x). Correctness of this algorithm follows from the observation that the Euler tour starting at r follows a depth first search of T; thus, an arcs along the path towards the root will have smaller rankings than those along paths away from the root. That the time complexity of this algorithm is $\mathcal{O}(n/p + \log n)$ follows from our previous observations of complexities of the subroutines of this algorithm, as well as the observation that the weights of the rankings of the $n-1$ pairs of edges can be compared in time $\mathcal{O}(n/p)$.

2.5.3 Traversal Numbering

Given a tree $T = (V, E)$ rooted at $r \in V$ by $p : V \to V$, we can compute the preorder traversal d in which each node appears in time $\mathcal{O}(n/p+\log n)$ as follows. First, we construct an Euler tour S of T' as previously described. We then assign the weights $w((p(v), v)) = 1$ and $w((v, p(v))) = 0$ to these arcs. Then the preordering d of a vertex v equals the ranking of $(v, p(v))$, and $p(r) = 0$. With small variations, we can also compute the inorder traversal and the postorder traversal of T.

2.5.4 Vertex Depth

Given a tree $T = (V, E)$ rooted at $r \in V$ by $p : V \to V$, we can compute the depth d of each vertex in time $\mathcal{O}(n/p + \log n)$ using the same approach as before, except by defining $w((p(v), v)) = 1$ and $w((v, p(v))) = -1$. Then the depth d of a node v equals the rank of $(p(v), v)$.

2.5.5 Subtree Size

Using a similar approach, but with $w((p(v), v)) = 0$ and $w((v, p(v))) = 1$, the size of the subtrees rooted at each vertex v equals the difference between the ranking of $(v, p(v))$ and $(p(v), v)$ since each arc between these two arcs is in the subtree rooted at v.

2.6 Connected Components

In this section, we describe an algorithm as presented by Hirschberg, Chandra, and Sarwate [17] for connected components, assuming the input is provided as an adjacency list. We recursively solve the problem as follows. For each node, we select the edge incident to the smallest ordered vertex. The set of edges induces a forest on the graph; in each tree, some edge appears twice. We arbitrarily select one of the endpoints as the root of this tree, and connect all roots to a 'superroot' s. We then compute the depth first search ordering on this tree and note that nodes in the same tree will have adjacent numberings. Using this fact, we can replace all edges (u, v) with edges incident to the leaders of u and v and recursively solve the problem. As each iteration decreases the number of nodes by at least half, we

will solve the base case after $\mathcal{O}(\log n)$ iterations. Thus, this problem can be solved in time $\mathcal{O}((m+n)\log n/p)$ for any $p \leq n+m$.

2.7 Minimum Spanning Tree

We can adapt the previous algorithm for connected components to find a minimum spanning tree. For each vertex v, instead of selecting the vertex of smallest address, we select a vertex v' across the edge of least weight incident to v, as long as the address of v' is less than that of v. Thus, by the same analysis, the total time complexity of this algorithm is $\mathcal{O}((m+n)\log n/p)$.

3 Cache-Efficient Algorithms

A shortcoming of the sequential model is the assumption of unit access time to memory. In reality, processor speeds are increasing at a rate much greater than that of data access speeds. For example, main memory access is about one million times faster than disk access. Consequently, often in applications with large data sets, a significant fraction of the running time is the memory access time.

Initially, external memory algorithms were developed for certain applications with large datasets, such as sorting checks by account numbers for banks [1]. More recently, however, with the development of cache hierarchies, the benefit of these algorithms extends to more mainstream applications and architectures, as L_1 caches behave essentially the same as external memories.

3.1 Performance of Cache-Efficient Algorithms

In the cache efficient model, we have a memory hierarchy consisting of two levels– an internal memory (or *cache*) of size M and an arbitrarily large external memory (or *disk*)– between which we exchange blocks of size B in unit time. The efficiency of an algorithm is measured in terms of the number of such exchanges (called *memory transfers*).

The cache efficient model serves well to model memory transfers between two adjacent cache levels. However, with multiple cache levels, algorithm design tuned to cache parameters becomes cumbersome and rarely portable. A result due to Frigo et al. [15] states that, if we design good cache efficient algorithms without reference to the particular parameters of memory hierarchy, then this *cache-oblivious* algorithm will perform well on any multilevel memory hierarchy.

There are two lower bounds for primitive operations in cache efficient algorithms. First, the *linear* or *scanning* bound $\Omega(\frac{N}{B})$ represents the number of memory transfers needed to just read N contiguous elements from memory. This scanning bound is trivial to prove, as we can read at most B new words with one cache miss, and so to read N words, we will cause at least N/B cache misses. Second, the *sorting* bound $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B})$ represents the number of memory transfers needed to sort N elements. The sort bound is presented by Aggarwal and Vitter [1].

3.2 Cache-Oblivious Prefix Sums

In the cache efficient model, computing the prefix sums optimally is trivial, however we include it in this thesis for completeness and for a simple example of I/O analysis. The algorithm operates exactly the same as the sequential algorithm and so that remains to analyze is its I/O complexity. This computation proceeds by transferring in one block of $\mathcal{O}(B)$ summands into the cache, computing the first B partial sums, and writing these partial sums to memory. These summands are no longer used in the computation, and so can be evicted later; we need only maintain a current summation of all summands added so far, and this requires only constant space. Thus, each contiguous block of $\mathcal{O}(B)$ summands contributes 2 memory transfers. This process continues until all partial sums have been written, and so this algorithm executes $\mathcal{O}(N/B)$ memory transfers. Additionally, note that, because the sequential algorithm does not reference the cache parameters, this algorithm is cache-oblivious.

3.3 Cache-Oblivious List Ranking

The cache-oblivious algorithm due to Arge et al.[4] for list ranking is inspired by those for parallel models. At a high level, the algorithm behaves essentially the same: an independent set of $\Theta(N)$ nodes is found in with $\mathcal{O}(\text{sort}(N))$ cache misses, these nodes are contracted out, and the remaining list is recursively ranked. This requires a total of $T(n) = T(N/c) + \mathcal{O}(\text{sort}(N)) = \mathcal{O}(\text{sort}(N))$ I/Os. One essential difference is in the construction of an independent set.

One way of finding an independent set by 4-coloring a list, due to Arge et al. [4], is as follows. Call an edge $(i, S(i))$ a *forward* edge if i appears before $S(i)$ in the unordered sequence of nodes; that is, $i < S(i)$. Otherwise call it a *backwards* edge. This definition naturally separates sequences of node into forward chains and backwards chains. Call a node i a *head node* if it has the least index in its forward chain; that is, $i < S(i)$ and $i < P(i)$. Similarly define tail nodes. Note that every node will be in one chain, and head or tail nodes will be in both a forward chain and a backward chain. They color these chains as follows. They color the head of each forward chain red, and then color the succeeding nodes alternatingly blue and red. Similarly, they color the heads of each backward chain red, and then color the succeeding nodes alternatingly blue and red. At this point, every node is colored with one color validly except for the head and tail nodes. This can be fixed by coloring every head node green and every tail node yellow.

Following Arge et al. [4], we describe how to efficiently implement this algorithm cache-obliviously assuming that we have a cache-oblivious priority queue which supports each INSERT, DELETEMIN, and DELETE operation in $\mathcal{O}(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers and $\mathcal{O}(\log N)$ amortized computation time. Arge et al. [4] describe how to implement such a priority queue. We detail how to cache-obliviously color the forward lists, as the backward lists can be colored similarly.

CACHE-OBLIVIOUS COLORING
(1) Create a cache oblivious priority queue Q
(2) INSERT each head node with color red
(3) While Q is nonempty

(3.1) Let i be the index and c be the color of the ExtractMin operation
(3.2) Color node i with color c
(3.3) If $S(i)$ is in this forward chain, Insert $S(i)$ with the color opposite c.

At a high level, they use the priority queue to ensure that nodes are colored (and thus, accessed) from least to highest memory address, so as to require only $\mathcal{O}(\text{scan}(N))$ I/Os. The correctness of this algorithm can be proven with the following loop invariant: Let F denote all nodes in a forward chain. After i ExtractMin operations, the i nodes in F of least index are properly 2-colored and each element of Q not a head node has associated color opposite that of its predecessor.

We now analyze complexity. We assume we have a predecessor array P as it can be constructed with $\mathcal{O}(\text{sort}(N))$ I/Os. We can then identify the head nodes by testing if both $i < S(i)$ and $i < P(i)$ with $\mathcal{O}(\text{scan}(N))$ I/Os. We perform at most N of each of the Insert and ExtractMin operations, and so the queue operations require $\mathcal{O}(\text{sort}(N))$ I/Os. Finally, because we color nodes from least memory address to greatest, these operations require $\mathcal{O}(\text{scan}(N))$ I/Os. Thus, we can color the forward lists with $\mathcal{O}(\text{scan}(N))$ I/Os.

Finally, they achieve a complete coloring of the list by performing a similar 2-coloring on the backward chains and a straightforward coloring of the head and tail nodes as previously described.

So, we can color a linked list of N nodes in time $\mathcal{O}(\text{sort}(N))$. In conjunction with the contraction technique, we can solve the list ranking problem cache-obliviously in $\mathcal{O}(\text{sort}(N))$ memory transfers.

3.4 Euler Tours and Tree Problems

We now show how Arge et al. [4] cache obliviously construct the Euler Tour defined in section 2.5 by adapting the corresponding parallel algorithm. First, for each vertex v, they construct a list of every incoming edge to v and sort it according to the cyclic order. Then, if (u, v) appears just before (u, w) in this sorted list, the successor of the incoming edge (u, v) is the outgoing edge (v, w). Therefore, they can can compute all successors in a scan of each of these lists. Thus, by taking the set of all pairs of edges and their successors, they can compute an Euler Tour of a tree by list ranking that set and then sorting the edges by rank. Because each subroutine of this algorithm has $\mathcal{O}(\text{sort}(N))$ cache complexity, and each subroutine is called only a constant number of times, the total cache complexity of this algorithm is $\mathcal{O}(\text{sort}(N))$.

As described in the previous section, by assigning certain weights to these edges and by comparing the rankings of the edges, we can compute a variety of tree problems, all within the $\mathcal{O}(\text{sort}(N))$ cache complexity.

3.5 Connected Components and Minimum Spanning Tree

We now describe a cache oblivious minimum spanning tree algorithm presented by Arge et al. [4]. At a high level, we contract a minimum weight edges of each vertex and recursively solve the problem on resulting super-vertices. At the beginning of each stage, we sort the edges by weight and for each vertex, we select a minimum weight edge to be part of our MST. The subgraph induced by the edges selected is then a forest. For each connected

component (tree) in this induced subgraph, some edge appears twice (namely, the minimum weight edge in the tree); we designate an endpoint of this vertex as the leader of its connected component. We connect each leader to a special vertex s and compute depth first search of the resulting tree rooted at s. Nodes within the same tree will then have consecutive numbers and so with a constant number of sorts and scans we can mark every vertex with its leader. Finally, we replace every edge between u and v with an edge between the leaders of u and v. The number of vertices has been reduce by at least half, using a constant number of sorts and scans. Thus, we can compute a minimum spanning tree (or, more generally, a minimum spanning forest) cache obliviously with $\mathcal{O}(\text{sort}(E) \cdot \log V)$ cache complexity.

4 Multicore Algorithms

As increases in processor speeds decelerate, multicore architectures have been introduced to restore the performance improvements to its historical orders of magnitude. Consequently, a variety of theoretical models [6, 5] have been presented to fully realize these advances in computer architecture. In this section, we discuss multicore models and we describe a number of graph algorithm results from [5] for the private cache model.

4.1 History of Theoretical Frameworks for Multicores

Alongside the introduction of chip multiprocessors (or multicores) has been the development of theoretical frameworks to maximally exploit these emerging architectures. A variety of papers [11, 6, 12, 16, 7, 10] have begun to realize this goal. Initial approaches focused on the development of schedulers with provably good performance [6]. Chowdhury and Ramachandran [10] introduced the evaluation of parallel algorithms with both private and shared caches by extending the Gaussian Elimination Paradigm framework to have distributed cache efficiency. Subsequent papers have introduced a variety of multicore models. Blelloch, Chowdhury et al. [6] describe a multicore cache model with both per-processor private caches and a large shared cache. Arge et al. [5] introduced a parallel external memory model which generalizes the PRAM model by augmenting each processor with a private cache. Chowdhury and Ramachandran [11] present a number of divide and conquer algorithms for this framework. In recent work, Chowdhury, Ramachandran, and Silvestri introduced a multicore with multilevel cache hierarchy model as well as a multicore obliviousness framework [12].

4.2 Performance of Multicore Algorithms

The model of computation we will use in this section is the *private cache* multicore model. In this model, each of the p processors has a private cache of size M and all share a common arbitrarily large external memory. The complexity analysis of an algorithm considers the number of parallel memory cache misses, parallel computation time, and space requirements. More formally, the cache complexity of an algorithm is the maximum number of cache misses at any processor. This model is similar to, but more general than the *parallel external memory* (PEM) model presented by Arge et al. in [5]. In particular, the private cache multicore model implements parallelism through the standard parallel constructs of *fork*

and *join* and parallel loop constructs, mirroring modern architectures, whereas the PEM model resembles the PRAM model with a global clock but with a single level of private caches. In this section, however, we will describe an algorithm for the PEM model before considering the private cache multicore model.

As introduced by Chowdhury, Ramachandran, and Silvestri [12], an oblivious algorithm is one which does not explicitly reference any machine parameters: the number of processors, the number of cache levels or their cache or block size. These algorithms often offer an appealing compromise between speed and portability. As multicores become increasingly prevalent, the expansive number of permutations on architectural specifications (number of processors, block transfer sizes, cache sizes, memory topology, among others) will make implementing multicore aware algorithms increasingly and perhaps prohibitively cumbersome. In a later section, we will present multicore oblivious variations on these algorithms and compare the complexity of these algorithms with that of their multicore aware counterparts.

4.3 Private Cache Multicore List Ranking

At a high level, Arge et al. [3] adapt the known PRAM algorithms and cache-efficient algorithms for list ranking to create a multicore efficient algorithm for list ranking with private caches. They first produce an independent set S of size $\Theta(N)$ and *contract out* of the list the members of this independent set. They then solve this problem recursively on the remaining nodes before extending the solution to the nodes in S.

The nonrecursive steps require a constant number of scans and sorts and operations with cache complexity $\mathcal{O}(\text{sort}_p(N))$. Thus, if an independent set S of size N/c for some $c > 1$ can be found in $\mathcal{O}(\text{sort}_p(N))$ cache misses, then the cache complexity of the list ranking algorithm satisfies $\mathcal{Q}(N,p) = \mathcal{Q}(N/c,p) + \mathcal{O}(\text{sort}_p(N)) = \mathcal{O}(\text{sort}_p(N))$. However, the only known PEM sorting algorithm in this context is optimal for only up to N/B^2 processors. Thus, they need in addition an optimal algorithm for problems of size less than pB^2. They solve the problem recursively in different ways depending on the size of the problem.

When $N > pB^2$, they use the optimal algorithm previously described with cache complexity of $\mathcal{O}(\text{sort}_p(N))$. When the problem size is $pB \leq N \leq pB^2$, they reduce the number of processors proportionally to the problem size while maintaining the cache complexity of sorting at $\mathcal{O}(B \log_{M/B} N/B)$. Finally, when the problem size is $N < pB$, they revert to the PRAM list ranking algorithm which runs in $\mathcal{O}(\frac{N}{p} + \log N)$ parallel time; when $N < pB$, this reduces to $\mathcal{O}(B + \log pB) = \mathcal{O}(B + \log p)$ parallel time. Even if at every time step they incur a cache miss, this results in cache complexity of $\mathcal{O}(B + \log p)$. We claim that this approach obtains the following recurrence relation for the cache complexity (the proof will follow shortly).

Problem Size	Cache Complexity	Algorithm
$N \geq pB^2$	$\mathcal{Q}(N/c,p) + \mathcal{O}(\frac{N}{pB} \log_{M/B} \frac{N}{B} + \log p \log\log N + (\log\log N)^2)$	Recursive
$pB < N < pB^2$	$\mathcal{Q}(N/c,p/c) + \mathcal{O}(B \log_{M/B} \frac{N}{B} + \log p + \log\log pB)$	Processor Scaling
$N \leq pB$	$\mathcal{O}(B + \log p)$	PRAM

The solution to this recurrence is $\mathcal{O}((B \log B + \frac{N}{pB}) \log_{M/B} \frac{N}{B} + (\log p + \log\log N)(\log \frac{N}{pB} \log\log N))$, which under a certain set of assumptions reduces to $\mathcal{O}(\text{sort}_p(N))$.

As usual, the heart of the problem is efficiently finding an independent set. The method described in [3] involves constructing a 2-ruling set from a $\log \log n$-coloring as follows. At a high level, they group nodes by color, and in iteration i select all nodes of color i which are not adjacent to nodes already selected. For the moment, we exclude the details of exactly how wise sorting makes this algorithm cache efficient, but will later describe our the details in the cache complexity analysis.

Multicore Ruling Set
(1) Apply Iterative Coloring twice to $\log \log n$-color the nodes and sort by color.
(2) Associate with each node the ID and color of its predecessor and successor.
(3) For each group G_i of nodes colored i from $1 \leq i \leq \log \log n$, do:
  (3.1) Collect the nodes without duplicates into a contiguous array T_i.
  (3.2) Add a duplicate of the successor of each node in T_i to its color group.
  (3.3) Add a duplicate of the predecessor of each node in T_i to its color group.
(4) Return R, the concatenation of all T_i

The correctness of this algorithm follows from the following claims which show that the set $R = \bigcup T_i$ constructed satisfies the definition of a 2-ruling set.

Lemma 4.1 ([3])**.** *The set R is an independent set.*

Proof. In each iteration, we add only nodes of the same color and so by the definition of a coloring, none of these nodes are adjacent. Across each iteration, we explicitly exclude the successor and predecessor of nodes already added to R. So R is an independent set. □

Lemma 4.2 ([3])**.** *Every element not in R has a neighbor in R.*

Proof. This algorithm excludes a vertex precisely when one of its neighbors has been selected for the ruling set. □

From Lemma 4.3, we conclude that every sequence of consecutive unselected items has at most two items; otherwise, a middle element would not have a neighbor in the ruling set. So R is in fact a 2-ruling set. As we have established correctness, we now analyze complexity. Recall first that we assume that $\log N \in \mathcal{O}(B)$ and $M = B^{\mathcal{O}(1)}$

Lemma 4.3 ([3])**.** *The cache complexity of* Multicore Ruling Set *is* $\mathcal{O}(\frac{N}{pB} \log_{M/B} \frac{N}{B})$.

Proof. Steps (1) and (2) take some constant number of sorts and scans of the whole list for a total of $\mathcal{O}(\text{sort}_p(N))$ cache misses. We now consider the complexity of each round of step (3), denoting by N_i the size of group G_i. To identify the items with duplicates present, they sort the nodes by their original numbering and scan across this list, comparing adjacent nodes. They compute the addresses to write the contiguous array by a prefix sums with cache complexity $\mathcal{O}(N_i/pB + \log p)$. So line (3.1) has a cache complexity of $\mathcal{O}(\text{sort}_p(N_i) + \log p)$. Between lines (1) and (2), for each node, this algorithm has computed the ID and color for both the predecessor and successor of a given node with a constant number of sorts, and have this information associated with that particular node with only a constant space overhead. Thus, in line (3.2), this algorithm will sort the array T_i first by color and then by ID. This algorithm then appends the duplicates to the end of the appropriate group by

computing a prefix sums on the array of duplicates and on the sizes of the groups. Note that, in the worst case, this algorithm must write some number of duplicates to each of the $\log \log N$ groups.

Thus, each round takes $\mathcal{O}(\text{sort}_p(N_i) + \log p + (\log \log N))$ cache misses. There are at most two duplicates of each node (that is, $\Sigma_i N_i \leq 3N$) and so each node participates in at most some constant number of sorts. Therefore, the sum cache complexity of step (3) is

$$\sum_{i=1}^{\log \log N} \mathcal{O}(\text{sort}_p(N_i) + \log p + \log \log N) = \mathcal{O}(\text{sort}_p(N) + \log p \cdot \log \log N + (\log \log N)^2)$$

We now want to argue that $\text{sort}_p(N)$ is the dominating term in this complexity. Because $N \geq pB^2$ and $M = B^{\mathcal{O}(1)}$, we have that $\frac{B \log N}{\log B} = \mathcal{O}(B \log_{M/B} \frac{N}{B}) = \mathcal{O}(\text{sort}_p(N))$.

□

Thus, their primary result follows.

Theorem 4.4. *A linked list of size N can be ranked in the PEM model with cache complexity $\mathcal{O}(sort_p(N))$ using up to $p = \frac{N}{B^2 \log B}$ processors, assuming $\log N = \mathcal{O}(B)$ and $M = B^{\mathcal{O}(1)}$.*

As a concluding remark, we note that, by executing Iterated Coloring additional times, the latter assumption can be reduced to $\log^{(k)} N = \mathcal{O}(B)$ for any constant k.

4.4 Euler Tours and Tree Problems

All that is required to construct an Euler tour and solve various tree problems is an algorithm to solve list ranking. The details were discussed in the previous two eponymous sections and so we do not revisit them in this section, except to mention that it can be done with the expected complexity bounds.

4.5 Connected Components and Minimum Spanning Tree

We now describe the algorithm due for connected components and minimum spanning tree. This algorithm is an adaptation of the single-processor external memory algorithm of Chiang et al. [8], which in turn is based on the PRAM algorithm of Chin et al. [9]. This multicore algorithm follows the same strategy as the external memory algorithm, but instead using the appropriate multicore subroutines and concluding the recursion with a PRAM optimal algorithm. As we have previously shown, the list ranking and tree contraction multicore algorithms are optimal for $p \leq \frac{N}{B^2 \log B}$. Therefore, when the size of the subproblem decreases to $\mathcal{O}(pB^2 \log B)$ vertices, we scale down the number of processors in proportion to the number of vertices. Finally, when the number of remaining vertices has decreased to $\mathcal{O}(pB)$, we use the PRAM optimal algorithm with time complexity $\mathcal{O}(\frac{n+m}{p} + \log p)$. Thus, the cache complexity of this algorithm is defined by the following recurrence:

$$C(n, m, p) = \begin{cases} C(n/2, m, p) + \mathcal{O}(\frac{n+m}{pB} \log_{M/B} \frac{n+m}{B}) & n \geq pB^2 \log B \\ C(n/2, m, p/2) + \mathcal{O}(B \log B \log_{M/B} \frac{n+m}{B}) & pB < n < pB^2 \log B \\ \mathcal{O}(\frac{n+m}{p} + \log p) & n \leq pB \end{cases}$$

This recurrence solves to $C(n,m,p) = \mathcal{O}((\frac{n}{pB} + \frac{m}{pB}\log\frac{n}{pB} + B\log^2 B)\log_{M/B}\frac{n+m}{B})$. Thus, assuming $p \leq \frac{n}{B^2\log^2 B}$, we have that $C(n,m) = \mathcal{O}(\text{sort}_p(n) + \text{sort}_p(m)\log\frac{n}{pB})$.

5 Our Novel Results for the Multilevel Multicore Model

In this section, we present our novel results. We will discuss the multilevel multicore model using the following notation. We let h be the number of distinct levels in the cache hierarchy. For $1 \leq i \leq h$, M_i denotes the size of the caches to level i, B_i denotes the size of the block transfer at level i, and q_i denotes the number of level i caches (or, equivalently, the number of distinct groups of processors sharing a cache). When any of these variables appear without a subscript, we implicitly intend it to be subscripted with a 1. We extend the tall cache assumption to entail $M_i \geq B_i^2$. We also assume that $M_i \geq p_{i-1}M_{i-1}$. Essentially, this assumption states that lower cache levels have at least as much bandwidth and storage as higher levels; otherwise, these lower levels could not accomodate the higher levels.

5.1 Prefix Sums

We observe that, with a wise scheduler, the optimal PRAM algorithm for prefix sums can be made cache efficient for a multicore model with an arbitrary cache hierarchy. Essentially, we tell the scheduler to assign contiguous elements to contiguous processors; that is, those which share caches. This scheduler and proof of complexity can be found in [12].

We now analyze the time complexity of this algorithm. We claim that the time complexity of this algorithm is defined by the following function:

$$T(N) = \begin{cases} \mathcal{O}(N/p) + T(N/2) & N \geq 2p \\ \mathcal{O}(1) + T(N/2) & 1 < N < 2p \\ \mathcal{O}(1) & N = 1 \end{cases}$$

To solve this recurrence, we consider two separate recurrences whose sum is an upper bound of $T(N)$. The first recurrence is defined by $T'(N) = \mathcal{O}(N/p) + T(N/2)$ when $N > 1$ and $\mathcal{O}(1)$ otherwise; by the master theorem, this solves to $\mathcal{O}(N/p)$. The second recurrence is defined by $T''(N) = \mathcal{O}(1) + T(N/2)$ for $N > 1$ and $\mathcal{O}(1)$ otherwise; by the master theorem, this solves to $\mathcal{O}(\log N)$. As every term in $T(n)$ is accounted for in $T'(N)$ or $T''(N)$, we have that $T(N) \leq T'(N) + T''(N) = \mathcal{O}(N/p + \log N)$.

We now analyze the cache complexity of the nonrecursive steps in the algorithm, given a caching policy which distributes. When $N > 2p$, an optimal caching policy would request the first C_1 summands from each contiguous block of N/p summands. After all the pairs have been summed, the next set of C_1 summands is requested, and so on. An optimal caching policy would follow a similar approach after the recursive step as well. Thus, the cache complexity of this algorithm is defined by the following recurrence:

$$C(N,p) = \begin{cases} \mathcal{O}(N/pB) + C(N/2,p) & N \geq 2p \\ \mathcal{O}(1) + C(N/2,p/2) & 1 < N < 2p \\ \mathcal{O}(1) & N \leq 1 \end{cases}$$

Again, to solve this recurrence, we consider two separate recurrences whose sum is an upper bound of $C(N,p)$. The first recurrence is defined by $C'(N) = \mathcal{O}(N/pB) + C(N/2,p)$

whenever $N > 1$ and $\mathcal{O}(1)$ otherwise; again, by the master theorem, this solves to $\mathcal{O}(N/pB)$. The second recurrence is defined by $C''(N) = \mathcal{O}(1)+C(N/2)$ whenever $N > 1$ and $C''(N) = \mathcal{O}(1)$ otherwise; by the master theorem, this solves to $\mathcal{O}(\log N)$. As every term in $C(N, p)$ is accounted for in $C'(N, p)$ or $C''(2p)$, it follows that $C(N, p) \leq C'(N, p) + C''(2p) = \mathcal{O}(N/pB + \log p)$.

Finally, we note that this L_1 cache complexity analysis easily extends to caches in an arbitrary hierarchy, as described by Chowdhury, Ramachandran, and Silvestri [12], by replacing M with M_i, the L_i cache size, by replacing B with B_i, the size of the block transfers to level i, and by replacing p with p_i, the number of processors sharing an L_i cache. Thus, the L_i cache complexity of this algorithm is $\mathcal{O}(N/p_iB + \log p_i)$, which is optimal. Additionally, it follows that, because this algorithm is multicore-oblivious (that is, it does not reference the parameters of the multicore) and because it is cache efficient for a particular cache level, this algorithm is efficient for any particular cache level.

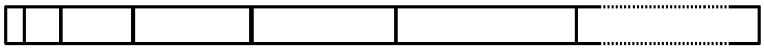
5.2 List Ranking

In the previous section, we described list ranking algorithms efficient for the private external multicore model given certain assumptions. In this section, we extend these results to the multilevel multicore model. First, we present an improvement to the list ranking algorithm as stated. In the algorithm due to Arge et al. [5], while writing the duplicates to separate groups, each processor could write to up to $\log \log N$ different groups; thus, step (3.2) of each iteration of the loop incurred $\mathcal{O}(\mathrm{sort}_p(N_i)+\log p+\log \log N)$ cache misses. We claim that this can be done with $\mathcal{O}(\mathrm{sort}_p(N_i) + \log p + \log \log N/p)$ cache complexity with a novel scheduler (their paper does not describe any scheduler for this routine). This results in optimal $\mathcal{O}(\mathrm{sort}_p(N))$ cache complexity under the assumptions that $\log p \log \log N = \mathcal{O}(\mathrm{sort}_p(N))$, $N > M$, and the tall cache assumption $M = \Omega(B^2)$, instead of the nonstandard assumptions of $M = B^{\mathcal{O}(1)}$ and $\log N = \mathcal{O}(B)$. In fact, we could derive the same result with $N = \Omega(B^{1+\epsilon})$ for any $\epsilon > 0$ instead of $N > M$ and $M = \Omega(B^2)$.

5.2.1 Disjoint Blocks Writing

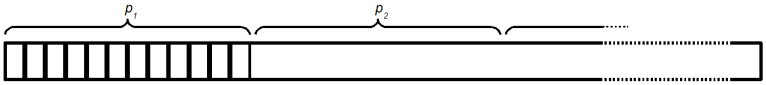
We begin by stating the problem abstractly and motivate this problem as nontrivial before presenting our solution and applying it to the list ranking problem. The input to the Disjoint Blocks Writing problem is a set of G tags, each with an associated G memory addresses sufficiently distant, and N items in contiguous memory, each with an associated tag from G. Note that the N items are not necessarily grouped by tag number. The required output is all of the items of the same tag written contiguously starting at the corresponding memory address. This problem is of theoretical interest also because it is trivial in both the parallel and the cache efficient contexts, but nontrivial in the multicore context.

We now motivate the need for a nontrivial scheduler by showing the cases when two naïve schedulers are unsatisfactory. At a given time step t, let $G(t)$ denote the number of contiguous groups, let $N_i(t)$ denote the number of unwritten elements in group i, and let $N(t)$ denote the total number of elements not yet written. First, consider the scheduler where we assign an equal number of processors to each group, and whenever a group finishes, its processors are reassigned to the lexicographically next group. Consider the instance where the size of group i is $\frac{G}{p}(2i-1)$ and $N > 2p$, as depicted in the following illustration.



Then it can be proven by induction that between time steps t and $t+1$, all groups $i \leq t$ have been written, and group $t+1$ has $\frac{G}{p}(t+1)$ elements to be written. So the processors assigned to the first group will be assigned to write some elements of every group, and so will have cache complexity $\Omega(G)$.

Second, consider the scheduler which assigns the i^{th} chunk of elements to be written to processor i. Consider the instance where the first $G-1$ groups have size 1 and the last group has size $(G-1)(p-1)$, as depicted in the following illustration.



The first processor will be assigned the first $G-1$ elements, each in separate groups, and so will have cache complexity $\Omega(G)$.

We now describe our scheduler, which assigns processors according to two separate cases. In both cases, we first sort the groups by tag and then sort the groups from smallest to greatest size. Whenever $p < G$, we assign any available processor to the group of smallest size which currently has no processor assigned to it. Thus, whenever a processor completes its group, all processors which were assigned since it was reassigned have finished their group as well (because these groups were of smaller size), and so each time a processor is reassigned, p groups have been completed. So, during this phase, no processor is assigned to more than $\frac{G}{p}$ groups. Next, whenever $p \geq G$, we assign $\lceil \frac{N'_i}{N'} \rceil p$ processors to group i. So we assign each processor to one group, and a total of $\Sigma_i(\frac{N'_i}{N'}+1)p = p+G$ processors are scheduled; however, because $G \leq p$, we will need no more than $2p$ processors. We can thus assign each of the p processors to complete the load of two processors, and so each processor is assigned to at most $2 = \mathcal{O}(\frac{G}{p})$ groups. Finally, it is clear that under this schedule, except for the last time step per group it is assigned to, each processor writes B elements, and so the total cache complexity of this schedule is $\mathcal{O}(\text{sort}_p(N') + \frac{N'+BG}{pB} + \frac{G}{p}) = \mathcal{O}(\text{sort}_p(N) + \frac{G}{p})$.

We now apply this result to augment the Multicore Ruling Set algorithm. In our particular case of one iteration of step (3), $N' = N_i$ and $G = \mathcal{O}(\log\log N)$, and so the cache complexities of step (3.2) and (3.3) are both $\mathcal{O}(\text{sort}_p(N_i) + \log p + (\log\log N)/p)$. Thus, the cache complexity of the the entire ruling set algorithm is $\mathcal{O}(\text{sort}_p(N) + \log p \log\log N + (\log\log N)^2/p)$. We now show that $(\log\log N)^2/p = \mathcal{O}(\text{sort}_p(N))$. By the tall cache assumption that $M = \Omega(B^2)$ and the assumption that $N > M$, we have that $\sqrt{N} = \frac{N}{\sqrt{N}} = \mathcal{O}(\frac{N}{\sqrt{M}}) = \mathcal{O}(\frac{N}{B})$. Because $(\log\log N)^2 = \mathcal{O}(\sqrt{N})$, we have that $(\log\log N)^2/p = \mathcal{O}(\text{sort}_p(N))$. Thus, this complexity reduces to $\mathcal{O}(\text{sort}_p(N) + \log p \log\log N)$, and so our scheduler improves the asymptotic complexity of this algorithm whenever $(\log\log N)^2$ would have been the dominating term in the complexity.

We now want to show that $\text{sort}_p(N)$ is the dominating term in this complexity whenever $\log N = \mathcal{O}(B)$ and $N \geq pM$. We thus have that $\log_{N/pB} \frac{N}{B} \leq \log_{M/B} \frac{N}{B}$. Also, $\log_{N/pB} \frac{N}{B} = 1 + \log_{N/pB} p$ and $\log \frac{N}{pB} = \frac{\log p}{\log_{N/pB} p}$. So if $\log \frac{N}{pB} \log\log N = \mathcal{O}(\frac{N}{pB})$, then $\log p \log\log N = \mathcal{O}(\text{sort}_p(N))$. Because $N \geq pB^2$ and by our assumption that $\log N = \mathcal{O}(B)$, we have that

$\log N = \mathcal{O}(B) = \mathcal{O}(\frac{N}{pB})$ and thus that $\log\log N = \mathcal{O}(\frac{N}{pB})$. Therefore, $\frac{\log p}{\log_{N/pB} p} \log\log N = \mathcal{O}((\log \frac{N}{pB})^2) = \mathcal{O}(\frac{N}{pB})$ and so we conclude that $\log p \log\log N = \mathcal{O}(\text{sort}_p(N))$.

The analysis of the list ranking algorithm given our analysis of this independent set algorithm follows in the same manner as before. We now highlight the conditions on the multicore parameters for which the two analyses presented reduce to the $\mathcal{O}(\text{sort}_p(N))$ lower bound. In this analysis, we have assumed that $N \geq pM$, $N \geq B^2 \log B$ and $\log^{(k)} N = \mathcal{O}(B)$ for any constant k. The analysis in [5] assumes that $N \geq pB^2 \log B$, $M = B^{\mathcal{O}(1)}$, and $\log^{(k)} B = \mathcal{O}(N)$ for any constant k.

5.3 Multicore Oblivious List Ranking

In this section, we assume we have a multicore oblivious sorting algorithm with time complexity $\mathcal{O}(N/p+\log p)$, cache complexity $\mathcal{O}(\text{sort}_p(N))$, and that the greatest term dominates in a series geometrically decreasing in N down to some $g(N)$ for a given p. We now adapt the previous algorithm to create a multicore oblivious algorithm. The algorithm follows the design of the List Ranking by Contraction Algorithm of section 2.4.2, but we find an independent set using the Multicore Ruling Set algorithm.

At a high level, beginning with a problem initially of size N_0, we identify an independent set using the Multicore Ruling Set algorithm, contract out the nodes, and recursively solve until the problem size decreases to at most a certain size $g(N_0)$. We then solve the subproblem of size $g(N_0)$ with the optimal PRAM list ranking algorithm, noting that the number of parallel time steps is an upper bound for the number of parallel cache misses. We thus derive the following recurrence relation defining the time complexity of this algorithm:

$$T(N,p) = \begin{cases} \mathcal{O}(\frac{N}{p} + \log p) + T(N/c, p) & g(N_0) \leq N \\ \mathcal{O}(\frac{N}{p} + \log p) & N < g(N_0) \end{cases}$$

This recurrence relation solves to $\mathcal{O}(\frac{N_0}{p} + \log p \log \frac{N_0}{g(N_0)})$. Next, we derive the following recurrence relation defining the cache complexity of this algorithm:

$$C(N,p) = \begin{cases} \mathcal{O}(\text{sort}_p(N) + \log p \log\log N + (\log\log N)^2) + C(N/c, p) & g(N_0) \leq N \\ \mathcal{O}(\frac{N}{p} + \log p) & N < g(N_0) \end{cases}$$

Similarly, this recurrence relation solves to $\mathcal{O}(\text{sort}_p(N_0)+(\log p \log\log N_0+(\log\log N_0)^2) \log \frac{N_0}{g(N_0)}+ \frac{g(N_0)}{p})$.Furthermore, the L_i cache complexity of this algorithm can be similarly derived with the appropriately subscripted parameters.

5.4 Euler Tours and Tree Problems

We can directly adapt many the parallel algorithms for tree problems to be multicore oblivious by using the previously described multicore oblivious list ranking algorithm. As previously described, all that is required is to define an Euler tour within the required complexity bounds. Instead of making the adjacency list circular, we simply ensure that the same processor is assigned to writing the successor of arc from the first node and to the last node in the adjacency list. Assigning the successor of each edge in the same manner

as the parallel algorithm is straightforward. This can thus be accomplished with the same complexity as list ranking. Furthermore, given a tree, we can computing a rooting, a traversal numbering, the depths of the vertices, and the sizes of the subtrees within the same complexity bounds.

5.5 Connected Components and Minimum Spanning Tree

Similarly, we can directly adapt the parallel algorithms for connected components and minimum spanning tree to be multicore oblivious by using the previously described multicore oblivious algorithms. Recall the steps in the connectivity graph algorithms. First, for each vertex, an edge of minimum weight is selected. Then, the roots of the forest induced by these edges are identified by selecting an endpoint of the edges which appear twice. A depth first traversal numbering of the nodes in these trees then associates each node with its root. Every edge is then replaced with an edge between its endpoints corresponding roots, and the problem is recursively solved. However, whenever the problem reaches size less than $g(N_0)$, we revert to the PRAM optimal algorithm for the corresponding connectivity problem. We thus derive the following recurrence relation defining the time complexity of this algorithm, where we denote by N the size of the input (i.e., $N = n + m$):

$$T(n,m,p) = \begin{cases} \mathcal{O}(\frac{N}{p} + \log p \log \frac{N}{g(N)} + \frac{g(N_0)}{p}) + T(n/2,m,p) & g(N_0) \leq N \\ \mathcal{O}(\frac{g(N_0)}{p} + \log p)) & \text{otherwise} \end{cases}$$

This recurrence relation solves to $\mathcal{O}((\frac{N_0}{p} + \log p \log \frac{N_0}{g(N_0)}) \log \frac{N_0}{g(N_0)} + \frac{g(N_0)}{p})$. Next, we derive the following recurrence relation defining the cache complexity of this algorithm:

$$C(n,m,p) = \begin{cases} \mathcal{O}(\text{sort}_p(N) + (\log p \log\log N + (\log\log N)^2) \log \frac{N}{g(N_0)} + \frac{g(N_0)}{p}) + C(n/2,m,p) & g(N_0) \leq N \\ \mathcal{O}(\frac{N}{p} \log g(N)) & \text{otherwise} \end{cases}$$

Similarly, this recurrence relation solves to $\mathcal{O}((\text{sort}_p(N_0)+(\log p \log\log N_0+(\log\log N_0)^2) \log \frac{N_0}{g(N_0)}+ \frac{g(N_0)}{p}) \log \frac{N_0}{g(N_0)} + \frac{g(N_0)}{p} \log g(N_0))$

5.6 Shared Caches

In presenting these multicore oblivious algorithms, we have analyzed only the private cache complexity explicitly. However, note that, other than sorting, all operations occur on disjoint sets of data of constant size. Thus, by using the coarse grained contiguous scheduler described by Chowdhury, Ramachandran and Silvestri [12], these steps achieve the scanning bound for cache complexity at any level. Thus, the cache complexity at each level is dominated by the cache complexity of sorting or of the PRAM algorithm.

6 Remarks

At present, Arge et al. require that $N \geq pB^2$ and we require that $\log p \log\log N = \mathcal{O}(\text{sort}_p(N))$. Ideally, we would eliminate this nonstandard assumption and design an algo-

rithm which is efficient for up to $p \leq N/\log N$ processors. In this section, we discuss what we might expect of this algorithm, and some of the substantial challenges this imposes.

6.1 Maintaining Work-Time Optimality

The first consideration in developing a multicore efficient algorithm is its time complexity. Ideally, we expect such list ranking algorithms to run in time $\mathcal{O}(N/p + \log p)$. The most simple parallel algorithm which achieves this time bound is the well known algorithm presented by Anderson and Miller [2]. We show that, in the worst case, this algorithm incurs unacceptably many cache misses in both shared and private caches. Additionally, we cannot overcome this bound by a simple preprocessing permutation, and it seems unlikely that a constant number of permutations through the execution of the algorithm could overcome these bounds. Thus, it seems we first need to develop a more sophisticated optimally parallel algorithm for list ranking; that is, to avoid needing nonstandard assumptions, we would need completely novel parallel techniques.

6.1.1 Shared Cache

As we process the blocks at different rates, we are unable to predict a priori which pointers will increment next, and thus our memory access could correspond to random access. We now present a (worst case) instance of the Parallel Contraction algorithm and analyze its cache complexity. In summary, each adjacent node is B memory addresses apart, except for the last in any block, which points to the first node in the next block. More formally, in a given block B_i where 1 is the index of the first node, we have that

$$S(i) = \begin{cases} \log n & \text{if } x = \log n - 1 \\ i + B & \text{if } i + B < \log n \\ i + B - \log n & \text{otherwise} \end{cases}$$

In the case where $B < \log N$ and $M < BN/\log N < N$, both the successor and predecessor of every node $N(p(i))$ are in different memory blocks than $N(p(i))$. Thus, the number of distinct memory blocks needed for each parallel step is $3N/\log N$. So every parallel step incurs $\Theta(N/\log N)$ cache misses. As the algorithm runs for $\Omega(\log N)$ steps, Parallel Contraction incurs $\Theta(N/\log N)\Theta(\log N) = \Theta(N)$ cache misses.

6.1.2 Private Cache

In the Parallel Contraction algorithm, nodes are explicitly partitioned to processors, but dynamic data about other processors' nodes is required. In particular, each iteration requires access the successor of $N(p(i))$, either to remove the next subject or to check if its isolated or to recolor itself. Furthermore, this successor changes with each iteration. We now present a (worst case) instance of the Parallel Contraction algorithm and analyze its cache complexity. In summary, every pair of blocks has a node in one whose successor is in the other. More formally, where i' denotes the distance from i to the top of the block and b denotes the block i is in,

$$S(i) = b + i' \log n - i' \pmod{N}$$

In the case where $M < \log N/B$, the successor of any given node $N(p(i))$ is in a different memory block. Thus, because each iteration accesses the successor of $N(p(i))$, each iteration of Parallel Contraction incurs a cache miss for each pointer $p(i)$. So, after $\Omega(\log N)$ iterations, we have incurred $\Theta(\log N)\Theta(N/p \log N) = \Theta(N/p)$ cache misses.

7 Conclusion

In this thesis, we have considered the problem of computing prefix sums, the problem of ranking a linked list, and some related graph problems on the parallel, cache efficient, and multicore models of computation. We have presented an optimal and multicore oblivious prefix sums algorithm on the multilevel multicore model. We also describe a scheduler for a part of a known list ranking algorithm which enables us to eliminate a term from the cache complexity of the algorithm. Finally, we have presented a multicore oblivious algorithms for solving list ranking, constructing an Euler tour, and related tree and connectivity problems.

References

[1] A. Aggarwal and J. S. Vitter. The I/O complexity of sorting and related problems. In *14th International Colloquium on Automata, languages and programming*, pages 467–478, London, UK, 1987. Springer-Verlag.

[2] Richard J. Anderson and Gary L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–869, 1991.

[3] L. Arge, M.T. Goodrich, and N. Sitchinava. Parallel external memory graph algorithms. 2009.

[4] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 268–276, New York, NY, USA, 2002. ACM.

[5] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 197–206, New York, NY, USA, 2008. ACM.

[6] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[7] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244, New York, NY, USA, 2004. ACM.

[8] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[9] Francis Y. Chin, John Lam, and I-Ngo Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 25(9):659–665, 1982.

[10] Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 71–80, New York, NY, USA, 2007. ACM.

[11] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 207–216, New York, NY, USA, 2008. ACM.

[12] Rezaul Alam Chowdhury, Vijaya Ramachandran, and Francesco Silvestri. Oblivious algorithms for multicore, network, and petascale computing. 2009.

[13] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control*, 70(1):32–53, 1986.

[14] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Inf. Comput.*, 81(3):334–352, 1989.

[15] Matteo Frigo, Charles E. Leiserson, Harald Prokop, Sridhar Ramachandran, and Z W(l. Cache-oblivious algorithms. extended abstract submitted for publication. In *In Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–397. IEEE Computer Society Press, 1999.

[16] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 271–280, New York, NY, USA, 2006. ACM.

[17] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, 1979.

[18] Joseph Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley Professional, March 1992.

[19] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. pages 869–941, 1990.

[20] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[21] R.E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14:862–874, 1984.

[22] James C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Cornell University, Ithaca, NY, USA, 1979.