Parallel External Memory Graph Algorithms

Lars Arge
MADALGO
University of Aarhus
Aarhus, Denmark
large@madalgo.au.dk

Michael T. Goodrich
University of California – Irvine
Irvine, CA USA
goodrich@ics.uci.edu

Nodari Sitchinava
MADALGO
University of Aarhus
Aarhus, Denmark
nodari@madalgo.au.dk

Abstract—In this paper, we study parallel I/O efficient graph algorithms in the Parallel External Memory (PEM) model, one of the private-cache chip multiprocessor (CMP) models. We study the fundamental problem of list ranking which leads to efficient solutions to problems on trees, such as computing lowest common ancestors, tree contraction and expression tree evaluation. We also study the problems of computing the connected and biconnected components of a graph, minimum spanning tree of a connected graph and ear decomposition of a biconnected graph. All our solutions on a P-processor PEM model provide an optimal speedup of $\Theta(P)$ in parallel I/O complexity and parallel computation time, compared to the single-processor external memory counterparts.

I. Introduction

With the advent of multicore architectures, and the realization that processors are not increasing in speed the way they used to, there is an increased realization that parallelism is the primary remaining method for achieving orders of magnitude improvements in performance, through the use of chip-level multiprocessors (CMPs) [18], [21].

Recently Arge *et al.* [3] introduced the *parallel external memory (PEM)* model to model the modern CMP architectures. The PEM model is a parallelization of the sequential external memory (EM) model of Aggarwal and Vitter [1]. In this model each of the P processors contains a private cache of size M and all the processors share a common "external" memory (see Figure 1). The processors are not assumed to be organized in any particular network structure; hence, their only communication channel is through the common shared memory. Memory is subdivided into blocks of B contiguous memory cells, and, in any step, each processor can read or write one such block of memory cells. Access to the blocks is based on concurrent-read, exclusive-write (CREW) conflict resolution protocol, but can be adapted to have any other reasonable protocol, such as concurrent-read, concurrent-write (CRCW) or exclusive-read, exclusive-write (EREW). The complexity measures of the model are the number of parallel I/Os, parallel computation time and the total space required by an algorithm.
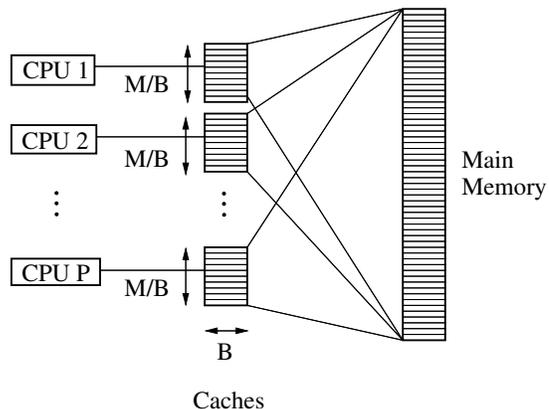
Figure 1.   The PEM model.

In this paper, we are interested in the design of efficient graph algorithms in the CREW PEM model. From an algorithmic standpoint, the main challenge is to exploit spatial locality of data while maintaining maximum concurrency.

In contrast, existing shared memory parallel (e.g. PRAM) algorithms access data in too random a pattern to utilize spatial locality. At the same time the existing external-memory algorithms are either too sequential [8] or rely on message passing and routing networks to communicate between processors [15], [16]. Thus, we need new approaches and techniques for the design of efficient PEM algorithms.

A. Prior Related Work.

With the chip multiprocessors becoming widely available, there has been some work in designing appropriate models for the CMPs. Bender *et al.* [4] proposed a concurrent cache-oblivious model, i.e. algorithms are oblivious to the cache parameters M and B. The authors presented and analyzed a search tree data structure.

Blelloch *et al.* [6] (building on the work of Chen *et al.* [7]) proposed *multicore-cache model* – a concurrent cache-oblivious model which consists of a two-level memory hierarchy: a per-processor private (L_1) cache and a larger (L_2) cache which is shared among all processors. The authors consider thread scheduling algorithms for a wide range of problems in the new model. However, their

analysis is limited to the hierarchical divide-and-conquer problems and a moderate level of parallelism. Chowdhury and Ramachandran [9] consider cache-complexity in both private- and shared-cache models for matrix-based computations, including all-pairs shortest paths algorithm of Floyd-Warshall. They also consider parallel dynamic programming algorithms in private-, shared- and multicore-cache models [10].

In contrast to the cache-oblivious multicore-cache model, Arge *et al.* [3] proposed a *cache-aware* parallel external memory (PEM) model. The authors provide several basic parallel techniques and solutions to fundamental combinatorial problems, such as prefix sums and sorting.

The cache-oblivious models are more attractive from the algorithm design point of view because, once designed, the algorithms do not depend on the particular values of the cache parameters M and B. However, just as in the single-processor case, it is more manageable to prove tighter bounds in the cache-aware PEM model while requiring fewer and/or weaker assumptions.

From algorithm engineering aspect, Cong and Bader [14] develop and analyze several practical techniques for efficient implementation of graph algorithms on CMPs.

B. Our Results.

In this paper we are interested in continuing exploration of the PEM model and provide a number of efficient graph algorithms.

Section III presents the main contribution of the paper – a solution to the well-known *weighted list ranking* problem [2], [13]. List ranking has traditionally been the linchpin in the design of parallel and EM graph algorithms. The I/O complexity of our algorithm on a list of size N is $\Theta(\text{sort}_P(N)) = \Theta\left(\frac{N}{PB}\log_{M/B}\frac{N}{B}\right)$ – time it takes to sort an array of N items on a P-processor PEM machine. This is a speedup of $\Theta(P)$ over the optimal single-processor EM algorithm, which takes $\Theta(\text{sort}(N)) = \Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ I/Os.

We show that while we can achieve optimal $\Theta(\text{sort}_P(N))$ I/O complexity via direct simulation of the *randomized* PRAM list ranking algorithm (Section III-A), the situation is not as simple for the deterministic case. Chiang *et al.* [8] prove that the difficulty arises from the random access pattern of following pointers in the list. Thus, to achieve efficient memory access during simulation of the PRAM algorithm we are forced to sort the whole list in each of the $\mathcal{O}(\log^* N)$ rounds of the recursive algorithm. This results in an extra $\log^* N$ factor in the overall I/O complexity.

However, we achieve the optimal $\Theta(\text{sort}_P(N))$ I/O complexity by developing the new *delayed pointer processing (DPP)* technique – a technique for lazy batched pointer processing. The DPP technique is described in Section III-B.

Solution to the list ranking problem opens opportunities for the use of the Euler tour technique of Tarjan and Vishkin [23] to solve various graph problems. The PEM solutions to the graph problems are presented in Section IV. All our solutions exhibit optimal $\Theta(P)$ speedup in the parallel I/O and computational complexities over their single-processor EM counterparts, while maintaining linear overall space complexity.

II. POINTERS AND EXTERNAL MEMORY

To solve the weighted list ranking problem we will make an extensive use of the PEM sorting algorithm of Arge *et al.* [3]. Thus, throughout this paper we will make the same assumption on the cache size M relative to the block size B, namely that $M = B^{\mathcal{O}(1)}$. Also, since the sorting algorithm requires that the number of processors be at most N/B^2, for ease of exposition we denote the maximum allowable number of processor by $p^* = N/B^2$.

While a list can be ranked in linear time in the RAM model, Chiang *et al.* [8] proved that list ranking and, consequently, many problems on graphs require $\Omega(perm(N))$[1] I/Os in the (single-processor) external memory model. In the next section we will show an upper bound of $\mathcal{O}(\text{sort}_P(N)) = \mathcal{O}(1/P \cdot \text{sort}(N))$ I/Os to rank a list.

To process lists I/O efficiently, the lists must be stored in contiguous memory. In particular, we view each list element as a record with fields stored in a contiguous array. Each record contains fields for successor and predecessor pointers, as well as any other auxiliary data required during an algorithm. Each list element is identified by a unique identifier, which is stored in one of the fields. A good choice for such an identifier is the record's original index in the array. The successor (predecessor) fields store the values of the identifiers of the corresponding elements in the list. For convenience, we refer to the values stored in the successor (predecessor) fields as the *successor (predecessor) pointers*.

All our algorithms require the linked lists to be doubly-linked. The following lemma states that even if the input list is singly-linked, we can construct its doubly-linked equivalent by performing a constant number of sorting rounds.

Lemma 2.1: A singly-linked list can be converted into a doubly-linked list in the PEM model in $\mathcal{O}(\text{sort}_P(N))$ I/Os.

Proof: Sort two copies of the array of records representing the linked list: one by the items' identifiers and the other by the items' successor pointers. For each record in the ith position of the first sorted array its predecessor in the list is located in the ith position of the second sorted array. Thus, by scanning the two sorted arrays in parallel with each processor scanning up to $\lceil N/P \rceil$ records of each array, store the identifiers of the predecessors of each item of the list with that item. Thus, the total I/O complexity of computing

---

[1]$perm(N) = \min\{N, \text{sort}(N)\}$ is the number of I/Os required to perform an arbitrary permutation of N elements, which for most realistic values of N and B is $\text{sort}(N)$.

the predecessor pointers is $\mathcal{O}(\text{sort}_P(N) + \text{scan}_P(N)) = \mathcal{O}(\text{sort}_P(N))$ I/Os.[2] ∎

In light of Lemma 2.1, throughout the paper we assume that lists are doubly-linked and each item stores the identifiers of both of its immediate neighbors.

During the list ranking algorithm we will often utilize operations on list elements which require access to the elements' immediate neighbors. We can generalize Lemma 2.1 to show that such operations can be applied in parallel to all items of the list in $\mathcal{O}(\text{sort}_P(N))$ I/O complexity in the PEM model.

Lemma 2.2: An operation on an item of a linked list which requires access only to the item and its immediate neighbors can be applied to all N items of a linked list in $\mathcal{O}(\text{sort}_P(N))$ I/O complexity on the P-processor PEM model.

Proof: Sort the linked list three times: by the items' identifiers, the successor pointers and the predecessor pointers. For each item in the ith position of the first sorted list the values of the neighbors will be located in the ith position of the other two lists. Scan the three lists simultaneously applying the operation to the items of the first list, with the items of the other two lists serving as operands. If the scan is performed in parallel with each processor scanning $\mathcal{O}(N/P)$ items of each list, the total I/O complexity is $\mathcal{O}(\text{sort}_P(N) + \text{scan}_P(N)) = \mathcal{O}(\text{sort}_P(N))$ I/Os. ∎

III. LIST RANKING

In this section we describe a PEM solution to the weighted list ranking problem. Given a linked list with weights on the edges, the *rank* of a node is defined as the sum of the weights on the edges from the head of the list to the node. The goal of the list ranking problem is to determine the ranks of every node in the list.

We adapt the PRAM algorithmic framework of Anderson and Miller [2] for list ranking, which is also used in the EM algorithm of Chiang *et al.* [8]. The pseudo-code for the PEM list ranking algorithm is presented in Algorithm 1.

We start by initializing the ranks of each list item to 0. Next, we find an independent set $\mathcal{S}$ of size $\Theta(N)$, bridge out the elements of the independent set from the list (updating the edge weights and item ranks in the process), recursively solve the weighted list ranking problem on the remaining items and, finally, reintegrate the elements of the independent set back into the list (see Figure 2 for an example). At the base of the recursion, when the list contains less than $PB\log^{(t)} N$ elements[3] (for some constant $t > 0$ defined by the independent set construction algorithm), we rank the list using any $\mathcal{O}(\log n)$ PRAM list ranking algorithm, for example Wyllie's pointer hopping algorithm [24].

---

[2] $\text{scan}_P(N) = \mathcal{O}(N/PB)$ is the I/O complexity of scanning an array of N items in parallel with each processor scanning $O(N/P)$ elements.

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

Algorithm 1 PEM list ranking algorithm.

1: INITIALIZE_RANKS($\mathcal{L}, 0$)
2: RANK($\mathcal{L}, P, |\mathcal{L}|$)

3: **procedure** RANK($\mathcal{L}, P, N$)
4:     **if** $|\mathcal{L}| \leq PB\log^{(t)} N$ **then** PRAM_RANK($\mathcal{L}, P$)
5:   ▷ t is some constant
6:     **else**
7:         $\mathcal{S} \leftarrow$ INDEP_SET($\mathcal{L}, P$)
8:         **for each** $v \in \mathcal{S}$ **in parallel do:**
9:             BRIDGE_OUT($v, \mathcal{L}$)
10:        $\mathcal{L} \leftarrow \mathcal{L}\backslash\mathcal{S}$; RANK($\mathcal{L}, P, N$)
11:        **for each** $v \in \mathcal{S}$ **in parallel do:**
12:            REINTEGRATE($v, \mathcal{L}$)
13:        $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{S}$
14:     **end if**
15: **end procedure**

(a) The members of the independent set are shaded.

(b) The members of the independent set are bridged out.

(c) After the recursive call returns.

(d) The members of the independent set are reintegrated.
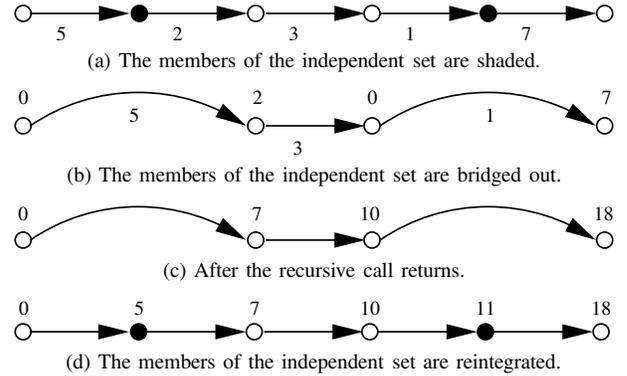
Figure 2. An example of running the list ranking algorithm. The node ranks are listed above the nodes, the edge weights are listed below the edges.

The seemingly strange choice for the base case of the recursion arises from the analysis of the algorithm. In particular, if we can perform every operation of the algorithm in $\mathcal{O}(\text{sort}_P(N))$ I/Os, then the I/O complexity of the list ranking algorithm is defined by the recurrence $Q(n, p) = Q(n/c, p) + \mathcal{O}(\text{sort}_P(n))$, where c is the constant defining the size of the independent set. Indeed, the initialization of the ranks can be achieved via parallel scan of the elements of the list, totaling $\mathcal{O}(N/PB) = \mathcal{O}(\text{sort}_P(N))$ I/Os. The bridging out and reintegration operations also require $\mathcal{O}(\text{sort}_P(N))$ I/O complexity (see Appendix A for details). However, as we will see in Section III-B, we can construct the independent set deterministically in the sorting I/O complexity only for up to $P \leq \frac{N}{B^2\log^{(t)} N}$ processors, which dictates a lower bound on the size of the list that can be ranked using the PEM algorithm before reverting to a PRAM algorithm.

We defer the detailed analysis of the list ranking algorithm to Section III-C, for after the presentation of the PEM

algorithm for independent set construction in the next two subsection.

A. Randomized Independent Set Construction

The simplest parallel solution to find an independent set of a list is an adaptation of the random mate approach originally proposed by Anderson and Miller [2]. The idea is to flip an independent fair coin for each item of the list and select the independent set consisting of the items whose coin flip turned up heads, but whose successor's coin flip turned up tails. The expected size of an independent set constructed in such a manner will be $(N-1)/4$. The coin flipping is independent of the values of the neighbors and can be conducted via a simple scan of the items. An item's membership to the independent set depends only on its own and its successor's coin flip value and by Lemma 2.2 can be accomplished in $\mathcal{O}(\text{sort}_P(N))$ I/O complexity.

B. Deterministic Independent Set Construction

The deterministic approach to finding a large independent set is via finding a 2-ruling set of the list, which satisfies our requirement for the size to be $\Theta(N)$.

Definition 3.1: An *r-ruling set* is a subset of the linked list such that it is an independent set and the number of consecutive unselected items is at most r. The items of the ruling set are called *rulers*, each of which *rules* up to r of the unselected items which follow the ruler in the list.

We adapt the deterministic coin tossing algorithm of Cole and Vishkin [13] for finding 2-ruling set, which also defines coloring of the list.

1) Deterministic coin tossing [13]: We assign each element of the list a tag. These tags will determine the colors of the nodes. The tags can be arbitrary, with the only constraint that the tags of two neighboring elements are distinct. Initially, we set the tag of each element to be the element's unique identifier.

A round of deterministic coin tossing, defines the color of each element v in the list as $2i+b$, where i is the index of the least significant bit of where the binary representations of v's tag differs from the tag of $succ(v)$, and b is the value of the ith bit of v's tag. For example, given list link (u,v) with $tag(u) = 21_{10} = 10101_2$ and $tag(v) = 57_{10} = 111001_2$, then $color(u) = 2 \cdot 2 + 1 = 5$ because the two tags differ in the 2nd bit, and the 2nd bit of $tag(u)$ has value 1.

The coloring via deterministic coin tossing immediately implies a solution to finding $\mathcal{O}(\log N)$-ruling set. In particular, the first item in the list is defined to be a *ruler*. For the rest of the members of the list, an item is defined to be a ruler if it is not an immediate neighbor of the first item in the list and its color value is smaller than the color values of both of its immediate neighbors in the list, i.e., its color value is a local minima (see Figure 3 for an example). Cole and Vishkin [13] prove that this definition of rulers defines an $\mathcal{O}(\log N)$-ruling set.
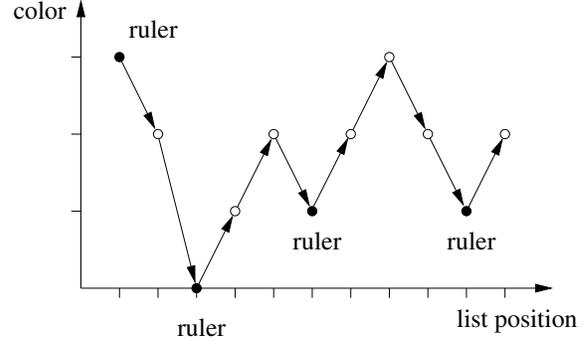
Figure 3. Definition of rulers given colors after deterministic coin tossing.

Lemma 3.2: A round of deterministic coin tossing and, consequently, $\mathcal{O}(\log N)$-coloring/ruling set of a linked list can be computed in the PEM model in $\mathcal{O}(\text{sort}_P(N))$ I/O complexity.

Proof: The process of computing the colors of the elements via deterministic coin tossing requires access to each element's tag and the tag of the item's successor. Computing $\mathcal{O}(\log N)$-ruling set requires access to each item's color, and colors of the item's successor and predecessor. Thus, by Lemma 2.2 all these operations can be accomplished in the PEM model in $\mathcal{O}(\text{sort}_P(N))$ I/O complexity. ∎

2) Finding 2-ruling set in $\mathcal{O}(\text{sort}_P(N) \cdot \log^* N)$ I/Os: Note that originally there are N distinct tag values and a single round of deterministic coin tossing assigns a color to each element for up to $\mathcal{O}(\log N)$ distinct colors. Now, if we update the tags of each element to be that element's newly computed color and perform another round of deterministic coin tossing, we will obtain $\mathcal{O}(\log \log N)$-coloring of the list, and, consequently, compute $\mathcal{O}(\log \log N)$-ruling set of the list. Note that during the $\mathcal{O}(\log N)$-coloring of the list, no two neighbors are assigned the same color and, therefore, the newly computed colors can be used as new tags. By iterating this process, we obtain a solution to finding a 2-ruling set as follows.

Initially, we set the tag of each element to be the element's identifier. We run deterministic coin tossing t (to be defined later) times. This provides us with the r-coloring and, consequently, $2r$-ruling set[4] of the list, where $r = \mathcal{O}(\log^{(t)} N)$. We let the colors of the rulers be 0 and group the items by their colors into $r+1$ groups $G_0, \ldots, G_r$, where index of the group indicates the color of the elements belonging to it.

We next proceed in $r+1$ rounds processing one group in each round using all P processors, starting with G_0 – the group containing the rulers of the $2r$-ruling set. In each round, we identify the (remaining) items of the current group as the 2-rulers and delete the immediate neighbors of the

[4]To be precise, r-coloring provides a $(2r-3)$-ruling set, but for the sake of simplicity, $2r$ is a good upper bound.

Algorithm 2 Simple PEM algorithm for computing 2-ruling set.

1: **procedure** RULING_SET($\mathcal{L}$)
2:     **for** $i = 0$ to t **do**: DET_COIN_TOSS($\mathcal{L}$)   ▷ Compute $2r$-ruling set of $\mathcal{L}$, $r = \mathcal{O}(\log^{(t)} N)$
3:     **for** $\forall v_R \in rulers(\mathcal{L})$ **in parallel do**: $color(v_R) \leftarrow 0$ ▷ Set rulers' color to 0
4:     Sort $\mathcal{L}$ by colors, grouping items into contiguous groups $G_0, \ldots G_r$ by their colors.
5:     **for** $i = 0$ to r **do**
6:         Identify items of G_i as the 2-rulers.
7:         **for each** $v \in G_i$ **in parallel do**: DELETE($succ(v)$)
8:         **for each** $v \in G_i$ **in parallel do**: DELETE($pred(v)$)
9:     **end for**
10: **end procedure**

Algorithm 3 PEM algorithm for computing 2-ruling set via delayed pointer processing (DPP).

1: **procedure** DPP_RULING_SET($\mathcal{L}$)
2:     **for** $i = 0$ to t **do**: DET_COIN_TOSS($\mathcal{L}$)   ▷ Compute $2r$-ruling set of $\mathcal{L}$, $r = \mathcal{O}(\log^{(t)} N)$
3:     **for** $\forall v_R \in rulers(\mathcal{L})$ **in parallel do**: $color(v_R) \leftarrow 0$ ▷ Set rulers' color to 0
4:     **for** $\forall v \in \mathcal{L}$ **in parallel do**: store copies of $succ(v)$ and $pred(v)$ with v.
5:     Sort $\mathcal{L}$ by colors, grouping items into contiguous groups $G_0, \ldots G_r$ by their colors.
6:     **for** $i = 0$ to r **do**
7:         **if** $i > 0$ **then**   ▷ Eliminate items marked for exclusion
8:             SORT(G_i); Scan G_i and remove all items that have duplicates.
9:         **end if**

10:         Identify the remaining items of G_i as the 2-rulers and compact G_i.   ▷ Add the rulers

11:         Sort G_i by colors of successors.   ▷ Mark successors for exclusion
12:         **for each** $v \in G_i$ **in parallel do**: APPEND($succ(v), G_{color(succ(v))}$)

13:         Sort G_i by colors of predecessors.   ▷ Mark predecessors for exclusion
14:         **for each** $v \in G_i$ **in parallel do**: APPEND($pred(v), G_{color(pred(v))}$)
15:     **end for**
16: **end procedure**

---

newly identified 2-rulers. The pseudo-code is presented in Algorithm 2.

Let's analyze the I/O complexity of this approach. Lemma 3.2 dictates that t rounds of deterministic coin tossing take $\mathcal{O}(t \cdot \text{sort}_P(N))$ I/Os. Grouping the items by colors can be accomplished in $\mathcal{O}(\text{sort}_P(N))$ I/Os by sorting the list by items' color values. Identification of the 2-rulers in Line 6 requires a simple parallel scan of elements of G_i. However, eliminating the neighbors of the newly identified 2-rulers, by Lemma 2.2 requires sorting the *entire* list in each of the $r + 1$ rounds. Thus, the total I/O complexity of the algorithm adds up to $\mathcal{O}(t \cdot \text{sort}_P(N) + (r + 1) \cdot \text{sort}_P(N))$. And if we set $t = \log^* N$, we obtain the 2-ruling set in $\mathcal{O}(\text{sort}_P(N) \cdot \log^* N)$.

Note that by running deterministic coin tossing $\mathcal{O}(\log^*(N))$ rounds we can reduce the number of colors to 4 without the need for batched parallel neighbor deletion of lines 5 through 9. This provides us with a 5-ruling set. The 5-ruling set is still of size $\Theta(N)$, and,

therefore, it suffices for use in the list ranking algorithm. However, the batched parallel deletion is of independent interest and will help us reduce the I/O complexity of finding a ruling set to $\mathcal{O}(\text{sort}_P(N))$ in the next subsection.

3) Computing 2-ruling set in $\mathcal{O}(\text{sort}_P(N))$ I/Os via Delayed Pointer Processing: While the previous algorithm is simple and elegant, we can improve the I/O complexity of finding the 2-ruling set to a constant number of sorting rounds. We achieve it via *lazy* batched parallel deletion of the neighbors. We call the technique *delayed pointer processing (DPP)*.

The pseudo-code for computing 2-ruling set via DPP is presented in Algorithm 3. As before, we compute r-coloring and, consequently, $2r$-ruling set ($r = \mathcal{O}(\log^{(t)} N)$) by running deterministic coin tossing t times and group items by their colors into $r + 1$ contiguous groups. However, this time we pick t to be an arbitrary positive constant and before processing each group, we store with every node a
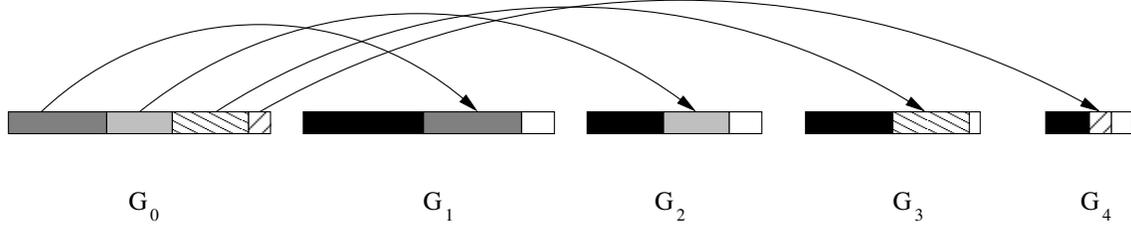
Figure 4. Group G_0 is sorted using the color of each element's successor as the comparison key. Then for every pair of elements $a = G_0[i], b = G_0[j]$ and their successors $succ(a) \in G_k, succ(b) \in G_l$, if $i < j$, then $k \leq l$.

copy of its two immediate neighbors.[5]

Next, we proceed in r rounds processing each color group G_i one at a time, starting with the group containing the rulers of the $2r$-ruling set as before.

The key idea in *delayed pointer processing* is to delay accessing the neighbors of the newly identified rulers as long as possible. In particular, consider newly identified ruler v and its neighbor w, which belongs to group $G_{color(w)}$. Instead of immediately accessing w and deleting it from the list, we append a copy w' of w at the end of $G_{color(w)}$. When it is the turn of $G_{color(w)}$ to be processed, a simple sort of only the elements of the group (rather than the whole list) will place w and w' in contiguous space. Thus, w and its duplicate can be identified with a simple parallel scan of the sorted $G_{color(w)}$. The presence of the duplicate will indicate that w should be removed from the list and not be one of the 2-rulers.

The key to success of the DPP approach is storing the copies of the neighbors with each node before processing each group. In particular, the copies provide each node with the colors of its two neighbors, and $G_{color(w)}$ can be determined locally without the need to follow the pointers to the neighbors. Sorting the elements of each group by using their neighbors' colors as the comparison keys allows us to append the duplicates I/O efficiently. In particular, all nodes whose neighbors have the same color will be placed in contiguous memory and, therefore, the copying can be performed via a simple scan.

Appending the duplicates requires extra space in each group which must be preallocated during group creation. Note that each item has at most two neighbors, hence, each item will acquire at most two duplicates. Therefore, for each group we need to allocate space equal to three times the number of elements of a particular color.

Finally, we need to determine the addresses within $G_{color(w)}$ where to copy the duplicates. In particular, multiple processors might be writing to the same group and

to avoid writing conflicts, the processors must determine the destination addresses before starting the copying process. Sorting the elements by the neighbors' colors, places items in memory in such a way that each processor will copy duplicates to a consecutive sequence of groups (see Figure 4). Thus, a simple run of segmented prefix sums on the number of duplicates that each processor writes to a particular group will uniquely identify the destination addresses for all duplicates.

Let's analyze the correctness and the I/O complexity of the above algorithm.

Lemma 3.3: The selected set is indeed a 2-ruling set.

Proof: No two neighbors are selected in the same round, because each group contains items with the same color. Across the rounds, the algorithm explicitly excludes the successors and predecessors of the items that have already been selected. Thus, the selected items constitute an independent set. Any unselected element has a selected neighbor because the only reason a vertex is excluded is because it has a selected neighbor; i.e., there are at most 2 consecutive unselected elements of the list. ∎

Lemma 3.4: If $P \leq p^*/\log^{(t)} N$ and $M = B^{\mathcal{O}(1)}$, then the I/O complexity of the 2-ruling set algorithm is $\mathcal{O}(\text{sort}_P(N))$.

Proof: By Lemma 2.2 it takes $\mathcal{O}(t \cdot \text{sort}_P(N))$ I/Os to generate the r-coloring. Storing the copies of the neighbors and grouping the items by colors takes another $\mathcal{O}(1)$ rounds of sorting. Let's say group G_i contains N_i items.

Let's analyze the I/O complexity of processing each group G_i. As we mentioned before, each item v has at most two neighbors in the linked list and, therefore, each group will at most triple in size during the duplicate addition. Thus, the duplicate removal takes a sort and a scan of $\mathcal{O}(N_i)$ items. Collecting the remaining items to be in the 2-ruling set and packing them into contiguous space for future sorting takes a scan and a run of prefix sums $\mathcal{O}(\text{scan}_P(N_i) + \log P)$ I/Os. Finally, we need to sort the newly identified members of the 2-ruling set ($\mathcal{O}(\text{sort}_P(N_i))$ I/Os), compute the addresses where the duplicates need to be written (a run of segmented prefix sums - $\mathcal{O}(\text{scan}_P(N_i) + \log P)$ I/Os) and copy the duplicates. Note, that each processor writes at most $2N_i/P$ duplicates, and while the sorting step puts all duplicates with

---

[5]Since each node has only two immediate neighbors, storing copies of the neighbors increases the space requirement only by a constant factor. However, if space usage is a concern, we can store only the comparison keys and the colors of the two neighbors, instead of the full copies of the nodes.

the same destination group contiguously, a processor might have to write to up to r different groups. Thus, the total I/O complexity of writing the duplicates is $\mathcal{O}(\mathrm{sort}_P(N_i) + \mathrm{scan}_P(N_i) + \log P + r) = \mathcal{O}(\mathrm{sort}_P(N_i) + \log P + \log^{(t)} N)$ I/Os.

The only difficulty that might arise is from the fact that some group might be smaller than the required minimum size of PB^2 for the PEM sorting. This is easily rectified via *processor scaling* – reducing the number of processors involved in sorting proportionally with the problem size. This observation provides us with $\Omega(B \log_{M/B}(N/B))$ lower bound required to sort a group. Therefore, the total I/O complexity of processing all rounds is

$$\sum_{i=1}^{\log^{(t)} N} \mathcal{O}(\mathrm{sort}_P(N_i) + \log P + \log^{(t)} N) =$$
$$\mathcal{O}\left(\mathrm{sort}_P(N) + B \log_{M/B}(N/B) \cdot \log^{(t)} N + \right.$$
$$\left. \log P \cdot \log^{(t)} N + (\log^{(t)} N)^2\right)$$

Since $M = B^{\mathcal{O}(1)}$, $B/\log(M/B) = \Omega(1)$ and, therefore, $\log^{(t)} N = \mathcal{O}\left(B \log_{\frac{M}{B}} \frac{N}{B}\right)$. And since $P \le p^*/\log^{(t)} N = \frac{N}{B^2 \log^{(t)} N}$, $\mathrm{sort}_P(N)$ is the dominating term.

Combining this with the I/O complexity of the initial r-coloring step and noting that t is a constant, we conclude that the total I/O complexity of finding the 2-ruling set is $\mathcal{O}(\mathrm{sort}_P(N))$. ∎

C. Analysis of the List Ranking Algorithm

With the solution to the independent set construction we can analyze the I/O complexity of the recursive list ranking algorithm described at the beginning of Section III.

Theorem 3.5: If $P \le \frac{p^*}{\log B \cdot \log^{(t)} N}$ and $M = B^{\mathcal{O}(1)}$, a linked list of size N can be ranked in the PEM model in $\mathcal{O}(\mathrm{sort}_P(N))$ I/Os.

Proof: As mentioned before, all steps of the PEM list ranking algorithm (Algorithm 1) except for constructing the independent set take $\mathcal{O}(\mathrm{sort}_P(N))$ I/O complexity for up to $p^* = N/B^2$ processors. The independent set construction algorithm exhibits optimal I/O complexity of $\mathcal{O}(\mathrm{sort}_P(N))$ for up to $\frac{p^*}{\log^{(t)} N} = \frac{N}{B^2 \log^{(t)} N}$ processors. For the case when the problem size is $PB \log^{(t)} N \le n < PB^2 \log^{(t)} N$, we reduce the number of processors proportionally to the problem size (processor scaling), to maintain the I/O complexity of finding an independent set at $\mathcal{O}(B \log^{(t)} N \cdot \log_{M/B} n/B)$ I/Os. Finally, we stop the recursive list ranking algorithm whenever the problem size becomes smaller than $PB \log^{(t)} N$ and revert to the PRAM list ranking algorithm. The PRAM list ranking algorithm with P processors on input of size $PB \log^{(t)} N$ runs in $\mathcal{O}(B \log^{(t)} N + \log PB)$ parallel time. Since $P \le \frac{N}{B^2}$ and $M = B^{\mathcal{O}(1)}$, we observe that $\mathcal{O}(\log PB) = O(\log \frac{M}{B} \cdot \log_{M/B} \frac{N}{B}) = \mathcal{O}(\log B \cdot \log_{M/B} \frac{N}{B})$. And even if we charge a full block transfer

for each PRAM memory access, the parallel I/O complexity of this step is $\mathcal{O}(B \log^{(t)} N + \log B \cdot \log_{M/B} N/B)$.

Thus, the total I/O complexity of the list ranking algorithm is defined by the recurrence:

$$Q(n,p) = \begin{cases} Q(n/c, p) + \mathcal{O}\left(\dfrac{n}{pB} \log_{M/B} \dfrac{n}{B}\right) \\ \qquad\qquad \text{if } n \ge PB^2 \log^{(t)} N; \\ Q(n/c, p/c) + \mathcal{O}\left(B \log^{(t)} N \cdot \log_{M/B} \dfrac{n}{B}\right) \\ \qquad\qquad \text{if } PB \log^{(t)} N \le n < PB^2 \log^{(t)} N; \\ \mathcal{O}\left(B \log^{(t)} N + \log B \cdot \log_{M/B} \dfrac{N}{B}\right) \\ \qquad\qquad \text{if } n < PB \log^{(t)} N. \end{cases}$$

The solution to this recursion is $Q(N,P) = \mathcal{O}\left(\left(\frac{N}{PB} + B \log B \cdot \log^{(t)} N + \log B\right) \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$. And as long as $P \le \frac{N}{B^2 \log B \cdot \log^{(t)} N}$, $B \log B \cdot \log^{(t)} N = \mathcal{O}\left(\frac{N}{PB}\right)$. Therefore, $Q(N,P) = \mathcal{O}\left(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B}\right) = \mathcal{O}(\mathrm{sort}_P(N))$. ∎

IV. APPLICATIONS

The *Euler tour technique* was introduced by Tarjan and Vishkin [23] to solve various graph problems in the PRAM model. We follow the same framework to solve graph problems in the PEM model.

A. Euler Tour and Basic Tree Problems

Given a tree T and a special vertex r of T, an *Euler tour* of T is defined as a traversal of T that starts and ends at r and visits every edge exactly twice: once in each direction.

Theorem 4.1: In the P-processor PEM model an Euler tour of a tree can be built in $\mathcal{O}(\mathrm{sort}_P(N))$ I/O complexity using up to N/B^2 processors.

Proof: The single-processor EM solution of Chiang *et al.* [8] for building the Euler tour involves scanning the undirected edges of the tree, replacing each one with two directed edges, sorting the new edges lexicographically and, finally, scanning them assigning appropriate successor for each edge in the Euler tour order. By performing the sorting and scanning using the corresponding PEM versions of each subroutine, we can build the Euler tour in the PEM model in $\mathcal{O}(\mathrm{sort}_P(N))$ I/O complexity. The only constraint on the maximum number of processors is imposed by the sorting algorithm, which requires $P \le N/B^2$. ∎

Theorem 4.2: The following problems can be solved in $\mathcal{O}(\mathrm{sort}_P(N))$ I/O complexity using up to $\frac{N}{B^2 \log B \cdot \log^{(t)} N}$ processors in the P-processor PEM model: rooting a tree, determining preorder and postorder numbering of the vertices, the depth of each vertex and the sizes of the subtrees rooted at each vertex.

Proof: The single-processor EM solution for each of the problems involves building an Euler tour, running weighted list ranking with the appropriate edge weights and a constant number of sorts and scans. With the PEM solution to

building the Euler tour, all these problems can also be solved in the PEM model by utilizing the appropriate PEM versions of each subroutine, each of which takes at most $\mathcal{O}(\text{sort}_P(N))$ I/Os using up $\frac{N}{B^2 \log B \cdot \log^{(t)} N}$ processors. ∎

B. Tree Contraction and Expression Tree Evaluation

Trees play an important role in designing efficient parallel algorithms. Contracting a tree in parallel has been extensively studied and efficient solutions have been proposed in the PRAM model [17], [20]. One of the applications of tree contraction is expression evaluation, which we will use in the next section to answer *lowest common ancestor* queries efficiently.

In the PEM model, we adapt the optimal deterministic solution in the EREW PRAM model due to Gazit *et al.* [17]. The PRAM solution decomposes the tree into subtrees. Each subtree, which is called an m-*bridge*, is of size at most $m+1$. An important property of the m-bridges is that each one contains at most two *attachments*, i.e. vertices where the m-bridge is connected to the rest of the tree. Moreover, at most one of these attachments is a leaf of the m-bridge, thus, each m-bridge can be processed and contracted into a single vertex independently of other parts of the tree.

The m-bridges are easily identified by constructing an Euler tour of the tree and performing list ranking of the tour.

The PRAM algorithm sets $m = 2N/P$ and decomposes the tree into P m-bridges. Each of the P processors in parallel contracts an m-bridge into a single vertex, hence, reducing the size of the tree from N to P. Finally, the P processors contract the tree of P vertices into a single vertex in $\mathcal{O}(\log P)$ time.

Theorem 4.3: In the P-processor PEM model tree contraction and, consequently, expression tree evaluation can be solved in $\mathcal{O}(\text{sort}_P(N))$ I/O complexity using up to $\frac{N}{B^2 \log B \cdot \log^{(t)} N}$ processors.

Proof: The PEM Euler tour construction and list ranking algorithms allow us to decompose the tree into m-bridges, for $m = 2N/P$. The properties of the m-bridges imply that each one can be laid out in $\mathcal{O}(m/B) = \mathcal{O}(N/PB)$ blocks of memory by sorting the nodes of the tree in the postorder. Each processor concurrently contracts the m-bridge using the sequential EM tree contraction algorithm of Chiang *et al.* [8]. Finally, the resulting P-vertex tree is contracted into a single vertex in $\mathcal{O}(\log P)$ I/Os by running the n-processor $\mathcal{O}(\log n)$ EREW PRAM tree contraction algorithm and charging a full I/O for each PRAM memory access. The I/O complexity of every subroutine (including the last one) is dominated by the sorting complexity of the list ranking algorithm. The theorem follows. ∎

C. Batched Lowest Common Ancestors

The *lowest common ancestor (LCA)* of two vertices u and v of a rooted tree T is defined as the lowest node in T that has both u and v as descendants.

The single-processor external memory solution to the LCA problem is an adaptation of the PRAM solution of Berkman and Vishkin [5]. The EM solution [8] reduces the LCA problem to the range-minima problem, constructs a complete (M/B)-ary search tree with $\mathcal{O}(\log_{M/B}(N/B))$ levels and maintains at each internal node prefix and suffix minima of the leaves of subtrees rooted at those nodes, as well as a list of M/B minima of the subtrees rooted at each child of the node. Batched LCA queries are performed by first sorting the items, so that all of the queries can be answered by scanning the search tree a constant number of times.

Theorem 4.4: In the P-processor PEM model, K batched LCA queries in a tree of size N can be answered in $\mathcal{O}((1 + K/N) \cdot \text{sort}_P(N))$ I/O complexity for up to $\frac{N}{B^2 \log B \cdot \log^{(t)} N}$ processors.

Proof: To adapt the single-processor EM solution to the batched LCA problem in the PEM model, we set the branching factor of the search tree at $d = \min\{\sqrt{N/P}, M/B\}$ as in the sorting algorithm of Arge *et al.* [3]. This changes the depth of the tree to $\log_d P$. However, as it has been shown in [3], $\log_d P = \mathcal{O}(\log_{M/B} N/B)$ if $M = B^{\mathcal{O}(1)}$ and $P \le N/B^2$. By resorting to the PEM algorithms for sorting, scanning and expression evaluation (for computing the prefix and suffix minima) we can answer K batched LCA queries in the PEM model in $\mathcal{O}((1 + K/N) \cdot \text{sort}_P(N))$ I/O complexity. ∎

D. Connected and Biconnected Components, Ear Decomposition and Minimum Spanning Tree

In this section we show that several graph connectivity problems on undirected graphs $G = (V, E)$ can be solved in the PEM model with $\Theta(P)$ speedup in the I/O complexity over the corresponding single-processor EM algorithms, using up to $P \le \frac{|V|+|E|}{B^2 \log B \cdot \log^{(t)} N}$ processors.

Consider the single-processor external memory algorithm [8] for finding connected components of an undirected graph. For each vertex $v \in V$ the algorithm considers the set of edges $\{v, w_v\} \in E$, where w_v is the neighbor of v with the smallest identifier. The subgraph H of G induced by these edges is a forest and the connected components of H are identified via construction of the Euler tours for individual trees and a run of list ranking. Each connected component is then contracted into individual super-vertices and any singleton super-vertices are removed from further consideration. The algorithm proceeds recursively identifying the connected components of the resulting graph. Finally, when the recursive call returns, the singleton vertices are reintroduced and the super-vertices are expanded into the original subgraph H.

Theorem 4.5: Given an undirected graph $G = (V, E)$, connected components of the graph can be computed in the P-processor PEM model in $\mathcal{O}(\text{sort}_P(|V|) +$

$\text{sort}_P(|E|)\log(|V|/PB))$ I/Os using up to $\frac{|V|+|E|}{B^2\log B\cdot\log^{(t)}N}$ processors.

Proof: In the PEM model the single-processor EM algorithm is implemented using the PEM versions of the sorting, scanning and list ranking procedures. The only difference in the algorithm lies in the base case of the recursion. In particular, the PEM algorithm terminates the recursion and reverts to the $\mathcal{O}(\log n)$ PRAM connectivity algorithm of Shiloach and Vishkin [22] as soon as the size of the graph reaches $\mathcal{O}(PB\log^{(t)}N)$ vertices.

The PEM list ranking and tree contraction algorithms require at most $\frac{N}{B^2\log B\cdot\log^{(t)}N}$ processors to maintain the optimal $\mathcal{O}(\text{sort}_P(N))$ I/O complexity. Thus, once the size of the graph reaches $\mathcal{O}(PB^2\log B\cdot\log^{(t)}N)$ vertices, the I/O complexity of list ranking is maintained at $\mathcal{O}(B\log B\cdot\log^{(t)}N\cdot\log_{M/B}(n+m)/B)$ I/Os via processor scaling. The total I/O complexity of the algorithm is, therefore, defined by the recurrence:

$$Q(n,m,p)=$$
$$\begin{cases} Q(\frac{n}{2},m,p)+\mathcal{O}\left(\frac{n+m}{pB}\log_{M/B}\frac{n+m}{B}\right)\\ \qquad\qquad\text{if } n\geq PB^2\log B\cdot\log^{(t)}N;\\ Q(\frac{n}{2},m,\frac{p}{2})+\mathcal{O}\left(B\log B\cdot\log^{(t)}N\cdot\log_{\frac{M}{B}}\frac{n+m}{B}\right)\\ \qquad\qquad\text{if } PB\log^{(t)}N\leq n< PB^2\log B\cdot\log^{(t)}N;\\ \mathcal{O}\left(B\log B\cdot\log^{(t)}N+\log B\cdot\log_{M/B}\frac{n+m}{B}\right)\\ \qquad\qquad\text{if } n< PB\cdot\log^{(t)}N. \end{cases}$$

The solution to the recurrence is $Q(n,m,P)=\mathcal{O}\left(\left(\frac{n}{PB}+\frac{m}{PB}\log\frac{n}{PB}+B\log^2 B\cdot\log^{(t)}N\right)\log_{\frac{M}{B}}\frac{n+m}{B}\right)$. And as long as $P\leq\frac{n+m}{B^2\log B\cdot\log^{(t)}N}$, $B\log^2 B\cdot\log^{(t)}N=\mathcal{O}\left(\frac{m}{PB}\log\frac{n}{PB}\right)$. Therefore,

$$\begin{aligned} Q(n,m) &= \mathcal{O}\left(\left(\frac{n}{PB}+\frac{m}{PB}\log\frac{n}{PB}\right)\log_{M/B}\frac{n+m}{B}\right)\\ &= \mathcal{O}\left(\text{sort}_P(n)+\text{sort}_P(m)\log\frac{n}{PB}\right) \end{aligned}$$
∎

Theorem 4.6: Given an undirected connected graphs $G=(V,E)$ the following problems can be solved in $\mathcal{O}(\text{sort}_P(|V|)+\text{sort}_P(|E|)\log(|V|/pB))$ I/Os in the P-processor PEM model using up to $p\leq\frac{|V|+|E|}{B^2\log B\cdot\log^{(t)}N}$ processors: finding a minimum spanning tree, biconnected components, and ear decomposition (if the graph is biconnected).

Proof: The solution to finding connected components is easily augmented to solve the *minimum spanning tree (MST)* problem. In particular, when building the subgraph H, instead of picking the neighbor w_v with the smallest identifier, the algorithm picks the one incident on the edge with the smallest edge weight (ties are broken lexicographically using the identifier of the edge's endpoints as the secondary criteria). Then the MST contains all the edges of H plus the smallest-weight edge between the connected components found during the recursive calls.

The biconnected components algorithm of Tarjan and Vishkin [23] requires generating an arbitrary spanning tree, evaluating an expression tree, and computing connected components of the newly generated graph. The algorithm for ear decomposition of a graph of Maon *et al.* [19] requires generating an arbitrary spanning tree, performing batched lowest common ancestor queries and evaluating an expression tree. Using the appropriate PEM subroutines results in the PEM solutions for each of these problems. ∎

Note that, while the number of vertices is reduced by a factor of 2 between iterations in the above algorithms, no good upper bound can be given on the rate of reduction of edges for general graphs. However, for the family of graphs which are sparse (i.e., $|E|=\mathcal{O}(|V|)$) and which are closed under contraction, the rate of decrease of the edges between iterations matches that of the vertices. Thus, we obtain the following result for sparse graphs:

Theorem 4.7: For the family of sparse graphs $G=(V,E)$ closed under contraction, connected components, minimum spanning tree (if G is connected), bi-connected components and ear decomposition (if G is bi-connected) can be computed in $\mathcal{O}(\text{sort}_P(|V|))$ I/Os in the P-processor PEM model using up to $\frac{|V|+|E|}{B^2\log B\cdot\log^{(t)}N}$ processors. ∎

V. Conclusions

In this paper we extend on the work of Arge *et al.* [3] and study several fundamental graph problems in the PEM model. Our solutions provide an optimal speedup of $\Theta(P)$ in parallel I/O complexity and parallel computation time compared to the single-processor external memory counterparts when using up to $P\leq\frac{N}{B^2\log B\cdot\log^{(t)}N}$ processors.

Our solution to the list ranking, connected components and minimum spanning tree problems do not rely on the cache parameters M and B. In fact, since the initial release of the draft of this paper, Chowdhury *et al.* [11], [12] adapted our results to the multicore-cache model, a cache-oblivious parallel model. On the other hand, the reliance on the cache parameters is inherent to our solutions to the tree contraction, batched lowest common ancestor, and, consequently, to the solutions for finding biconnected components and ear decomposition in graphs. Thus, finding solutions to these problems in parallel cache-oblivious models remains an open question.

REFERENCES

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

[2] R. Anderson and G. Miller. Deterministic parallel list ranking. In *Proc. 3rd Aegean Workshop Computing*, volume 319 of *Lecture Notes Comput. Sci.*, pages 81–90. Springer-Verlag, 1988.

[3] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proc 20th ACM SPAA '08*, pages 197–206, 2008.

[4] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th ACM SPAA*, pages 228–237, 2005.

[5] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

[6] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proc. 19th ACM-SIAM Sympos. Discrete Algorithms*, 2008.

[7] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proc. 19th ACM SPAA*, pages 105–115, 2007.

[8] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 139–149, 1995.

[9] R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *Proc. 19th ACM SPAA*, pages 71–80, 2007.

[10] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. 20th ACM SPAA*, pages 207–216, 2008.

[11] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. Technical Report TR-09-19, University of Texas at Austin, Department of Computer Science, July 2009.

[12] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, 2010. To Appear.

[13] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal list ranking. *Inform. Control*, 1:153–174, 1986.

[14] G. Cong and D. A. Bader. Techniques for designing efficient parallel graph algorithms for SMPs and multicore processors. In *ISPA*, pages 137–147, 2007.

[15] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Algorithmica*, 36:97–122, 2003.

[16] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Bulk-synchronous parallel algorithms for external memory model. *Theory of Computing Systems*, 35:567–597, 2002.

[17] H. Gazit, G. L. Miller, and S.-H. Teng. Optimal tree contraction in an EREW model. In *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156, New York, 1988. Plenum Press.

[18] D. Geer. Chip Makers Turn to Multicore Processors. *IEEE Computer*, 38(5):11–13, 2005.

[19] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoret. Comput. Sci.*, 47:277–298, 1986.

[20] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. In *Proc. 26th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 478–489, 1985.

[21] J. Rattner. Multi-Core to the Masses. *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 3–3, 2005.

[22] Y. Shiloach and U. Vishkin. An O(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[23] R. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14:862–874, 1985.

[24] J. C. Wyllie. The complexity of parallel computation, 1979. Ph.D. thesis, Department of Computer Science, Cornell University.

APPENDIX

In this appendix we describe the details of bridging out and reintegrating the nodes of the independent set.

The bridging out and reintegration operations are presented in Algorithms 4 and 5, respectively. In particular, let v be a member of the independent set and x and y be the elements immediately preceding and, respectively, succeeding v in the list. The *bridging out* of v from the list consists of incrementing y's rank by the rank of v and the weight of edge (v, y), removing node v and edges (x, v) and (y, v) from the list and adding edges (x, y) and (y, x) to the list with the same weights as the weight of edge (x, v). We keep the pointers from v to its neighbors to be used during the reintegration of v.

Once all items of the independent set are bridged out, we still need to delete them from the array representations of the list, save them in a separate array (for the reintegration step) and pack the array representation into contiguous memory. These steps can be performed by marking the items as deleted, sorting the array using these marks as the comparison keys (this results in the marked elements being packed contiguously at the end of the array) and moving the

Figure 5. An example of applying BRIDGE_OUT operation on a list vertex v.

Algorithm 4 The operation of bridging out an element from a doubly-linked list

1: **procedure** BRIDGE_OUT$(v, \mathcal{L})$
2:     $x \leftarrow pred_\mathcal{L}(v); y \leftarrow succ_\mathcal{L}(v);$
3:                                  $\triangleright$ Update rankings
4:     $rank_\mathcal{L}(y) \leftarrow rank_\mathcal{L}(v) + rank_\mathcal{L}(y) + w_\mathcal{L}(v,y)$
5:                                    $\triangleright$ Update pointers
6:     $succ_\mathcal{L}(x) \leftarrow y; pred_\mathcal{L}(y) \leftarrow x;$
7:     $w_\mathcal{L}(x,y) \leftarrow w_\mathcal{L}(x,v)$
8: **end procedure**

"marked" items to a different array. Thus, deletion can be accomplished via a constant number of sorts and scans.

The *reintegration* of node v back into the list consists of removing the newly created edges (x, y) and (y, x), reinstating edges (x, v) and (y, v) with their original weights and setting the rank of v to be the sum of the newly computed rank of x and the weight of edge (x, v). Examples of application of these two operations are illustrated in Figures 5 and 6.

When copying the members of the independent set back to the array representation of the list, it is sufficient to simply append them to the end of the array and sort the array by the identifiers. Hence, copying the items back into the list can also be accomplished via a constant number of sorts and scans.

Figure 6. An example of applying REINTEGRATE operation on a list vertex v.

Algorithm 5 The operation of reintegrating an element back into a doubly-linked list

1: **procedure** REINTEGRATE$(v, \mathcal{L})$
2:                       $\triangleright$ Identified original neighbors
3:     $y \leftarrow succ_\mathcal{L}(v); x \leftarrow pred_\mathcal{L}(y)$
4:                           $\triangleright$ Restore original pointers
5:     $pred_\mathcal{L}(v) \leftarrow x; succ_\mathcal{L}(x) \leftarrow v$
6:     $w_\mathcal{L}(x,v) \leftarrow w_\mathcal{L}(x,y); pred_\mathcal{L}(y) \leftarrow v;$
7:                                $\triangleright$ Update rankings
8:     $rank_\mathcal{L}(v) \leftarrow rank_\mathcal{L}(v) + rank_\mathcal{L}(y) + w_\mathcal{L}(x,v)$
9: **end procedure**

Theorem A.1: The bridging out and reintegration of all elements of an independent set $\mathcal{S}$ in a linked list can be performed in the P-processor CREW PEM model in $\mathcal{O}(\text{sort}_P(N))$ I/Os.

Proof: The bridging out and reintegration operations require access only to the neighboring elements and, by Lemma 2.2, can be applied to all elements of $\mathcal{S}$ in $\mathcal{O}(\text{sort}_P(N))$ I/O complexity. And since $\mathcal{S}$ is an independent set, no concurrent writes are required when updating the pointers. As mentioned before, the deletion and reinsertion of the elements from the array representation of the list also takes a constant number of sorts and scans. The theorem follows. ■