Crosslinking in Parallel

Hari S. Asuri *†

Technical Report 92-93

September, 1992

ABSTRACT

A crosslink is a double link established between the two entries of an edge in an adjacency list representation of a graph. Crosslinks play important roles in several parallel algorithms as they provide constant time access between the two entries of an edge; the existence of crosslinks is usually assumed. We consider the problem of establishing crosslinks in a crosslink-less adjacency list for graphs that belong to a class of graphs called the *linearly contractible* graphs, and show that cross-links can be established optimally in $O(\log n \log^* n)$ time using a CREW PRAM and optimally in $O(\log n)$ time using a CRCW PRAM for such graphs.

* Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92717.

† The support of the National Science Foundation under Grant CCR 89-12063 is gratefully acknowledged.

Crosslinking in Parallel

Hari S. Asuri[†]
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

1. Introduction

The use of an adjacency list representation for a graph (particularly if it is sparse *), saves space, amongst other advantages. In parallel graph algorithms (for sparse and dense graphs) an adjacency list representation helps to quickly access successive neighbors of a vertex from within its adjacency list. The adjacency list representation of an undirected graph contains two copies of an edge, say $e = (v, w)$, with one entry each in the adjacency lists of vertices v and w. Since both the entries represent the same edge, computations done on one copy may have to be reported to the other copy so that incorrect or needless computations on the second copy can be avoided. If the graph is represented using an adjacency matrix structure, accessing the entry for the second copy is trivial. However, with an adjacency list representation, such an information transfer may require more than $O(1)$ time unless there is an easy way to access one copy from the other. *Crosslinks* provide such easy access.

A crosslink is just a pointer stored with an entry of an edge to the other copy of an edge. We will hereafter call the two copies *partners* of each other. Crosslinks provide constant time access between partners. Crosslinks have been used in sequential algorithms (for example, see [CNS81]) to achieve better running times. Several parallel algorithms use crosslinks to achieve optimality, and in some parallel algorithms existence of crosslinks helps in avoiding concurrent memory access. Fundamental algorithms such as the computation of Euler tours ([TV85]) use the crosslinks to achieve fast running times. Many parallel algorithms take as input an adjacency list representation of a graph and assume the existence of crosslinks in the input. In this paper, we consider the problem of establishing crosslinks in an adjacency list input that does not already have crosslinks. We consider the class of *linearly contractible graphs* ([Hage90]) and show that for such graphs, crosslinked adjacency lists can be constructed optimally in $O(\log n \log^* n)$ time using a CREW PRAM and optimally in $O(\log n)$ time using a CRCW PRAM.

---

† The support of the National Science Foundation under Grant CCR 89-12063 is gratefully acknowledged.

* We will say that a graph is *sparse* if the number of edges is linear in the number of nodes of the graph.

2. Preliminaries

Let $G = (V, E)$ be an undirected graph with $|V| = n$ and $|E| = m$. The vertices are numbered from 1 to n. Following Hagerup ([Hage90]), we say a class $\mathcal{G}$ of graphs is *linearly contractible* if for all graphs $G = (V, E)$ in $\mathcal{G}$,

a) $|E| = O(|V|)$, and

b) every minor of G is also in $\mathcal{G}$.

In particular, if (a) can be replaced by $|E| \leq c|V|$, we will say $\mathcal{G}$ is *linearly contractible with parameter c.*

Hereafter, we will only consider linearly contractible classes of graphs. Notice that planar graphs and bounded genus graphs in general are examples of classes of linearly contractible graphs.

We will assume that the input is a set of doubly-linked adjacency lists, one for each vertex $v \in V$, where an adjacency list for v contains one entry for each one of the edges incident on v, stored in no particular order. We assume that the numbers of both endpoints of an edge are stored in its entry in an adjacency list. (If not, the number of a vertex can be propagated through the entries of its adjacency list very easily.) Since the entries in an adjacency list are assumed to be stored in no particular order, for an edge $e = (v, w)$, finding the partner of the entry w in v's adjacency list may require a sequential search for an entry for v through the entries in w's adjacency list. This can be avoided if crosslinks can be established.

Sequentially, it is quite easy to establish crosslinks. We can in fact do it in linear time using bucketsort. We first form a list of records, with one record for each edge entry, where a record for an edge entry (v, w) will have the smaller of the two numbers v and w first, followed by the larger one. Now, we bucketsort the list of $2n$ entries first on the smaller number. Then, we bucketsort the resulting list on the second number. In the resulting sorted list, all partners are placed next to each other and therefore crosslinks can be established easily. This procedure takes $O(n)$ time.

In parallel however, we do not know of an integer sorting algorithm that sorts deterministically using linear work (none was known at the time of writing this paper). The best known deterministic integer sorting method uses $O(n \log\log n)$ work ([BDHPRS89]). Randomized parallel algorithms that do linear work are known ([Hage91] and [RR89]); however, these algorithms do not apply in our case as they consider more restricted input cases. A second option is to use $O(n^2)$ space to establish crosslinks, and this method takes only $O(1)$ time using $O(n)$ processors. For sparse graphs, the second method still uses $O(n^2)$ space, and one would prefer to eliminate the use of extra space. In this paper, we give optimal deterministic algorithms to establish crosslinks in certain sparse graphs, namely, linearly contractible graphs. We remark that in case of bounded degree graphs, the problem can be solved fairly trivially. In Section 3 below, we give an algorithm to establish crosslinks, that takes $O(\log n \log^* n)$ time using $O(n/\log n \log^* n)$ processors on a CREW PRAM. In Section 5, we give a CRCW PRAM algorithm that takes $O(\log n)$ time using $O(n/\log n)$ processors. Neither of these two algorithms uses more than $O(n)$ extra space.

3. A Parallel Algorithm for Establishing Crosslinks using a CREW PRAM

Our method is based on an alternative sequential method for establishing crosslinks in an adjacency list representation. Algorithm 1 establishes crosslinks sequentially.

Algorithm 1: Sequential Algorithm for Crosslinks

Input: An adjacency list representation $\mathcal{A}$ of a graph G and a parameter c

1. For each entry w in the adjacency list of each remaining vertex v do

   1.1 scan through the first $4c$ elements in the adjacency list of w and find the entry for v; (If w's adjacency list has less than $4c$ elements, scan through all of them.)

   1.2 establish crosslinks between the two entries;

   1.3 remove both entries from the corresponding lists;

   endfor;

2. Remove all vertices which have empty adjacency lists;
3. If there are no more vertices remaining, go to 4; else, go to 1;
4. Replace the removed entries in their respective adjacency lists;

Theorem 1: Algorithm 1 establishes crosslinks in the adjacency list $\mathcal{A}$ of $G \in \mathcal{G}$ (where $\mathcal{G}$ represents a linearly contractible class of graphs) in $O(n)$ time.

Proof: In step 1, we scan through each entry in each one of the adjacency lists and we establish crosslinks whenever we find a short list to search in. Assuming that we will have a short list to search in every time we execute step 1, at least one of the lists is emptied each time and hence, after a number of executions of step 1, $\mathcal{A}$ becomes empty. Since the entries that are removed are those for which crosslinks are established, when the algorithm stops, all crosslinks will have been established.

Let $G_0 = G, G_1, G_2, \ldots, G_k$ be the sequence of graphs produced during successive iterations of step 1, where G_k represents the empty graph. Let $G_i = (V_i, E_i)$, for $0 \leq i \leq k$. Since every G_i is a subgraph of G, $|E_i| \leq c|V_i|$. Hence, at least $|V_i|/2$ vertices of G_i must each have degree at most $4c$. In other words, during each iteration of step 1, we are not only guaranteed to find vertices with small degree, but the number of such vertices is at least one half of the number of vertices remaining in the graph during that iteration. Therefore, $|V_i| \leq n/2^i$ and $|E_i| \leq cn/2^i$. Hence, it suffices to perform $O(\log n)$ iterations of step 1 to process the entire graph. The number of operations performed during iteration i is $O(|V_i|)$ and hence, using a simple recurrence relation, we can see that the time taken by Algorithm 1 is $O(n)$. ■

The first version of the parallel algorithm that we propose is a simple parallelization of Algorithm 1. We assume that each entry in $\mathcal{A}$ has been allocated a processor. We then execute step 1 in parallel, i.e., processor $p_{v,w}$ allocated to entry $e = (v, w)$ on v's list searches through w's adjacency list for the entry $e' = (w, v)$. We let $p_{v,w}$ search through no more than the first $4c$ entries of w's adjacency list. If $p_{v,w}$ finds e' within the first $4c$ entries, then the crosslinks between e and e' are immediately established. If not, $p_{v,w}$ will wait to establish the crosslinks during a later iteration. Since several processors may need to read through a list simultaneously, we allow concurrent reading. After establishing crosslinks

between e and e', $p_{v,w}$ marks the two entries for deletion. The time spent by any processor $p_{v,w}$ is $O(1)$ as only a constant number of entries are searched in w's list. After an execution of step 1, a list compaction is performed on the entire set of adjacency lists (by combining them into a single long list) and the marked items are spliced out. List compaction can be done in $O(\log n)$ time. Step 1 is then iterated until there are no more vertices left to process. After $O(\log n)$ iterations, all crosslinks would have been established (from the proof of theorem 1) and we just replace the removed elements in the reverse order in which we removed them back in the adjacency lists by retracing the compaction steps. Alternatively, we can maintain two copies of the adjacency list representation (say, $\mathcal{A}$ and $\mathcal{B}$), and during the algorithm, we can establish crosslinks in $\mathcal{B}$ and splice out elements from $\mathcal{A}$. By doing this, we do not have to place the removed elements back in $\mathcal{A}$ (i.e., we can eliminate step 4), as at the end, $\mathcal{B}$ will have the adjacency list with the crosslinks.

The problem with the above method is that it under-utilizes too many processors and takes too much time. As given, the number of processors used is $O(n)$ and the time taken is $O(\log^2 n)$ because, though we spend constant time in the searching part of step 1, we spend $O(\log n)$ time for the compaction during each execution of step 1 and there are $O(\log n)$ iterations of step 1. We note, however, that since the size of the graph decreases by at least a half after each execution of step 1, and since the number of operations performed during each iteration of step 1 is linear in the size of the graph, the total number of operations performed is $O(n)$. After the first iteration, the number of idle processors is more than one half and this number increases with every iteration. We wish to reduce the time consumed as well as the number of processors used in the parallel algorithm.

There are two major issues to be addressed while trying to improve the parallel algorithm mentioned above:

(1) After each iteration of step 1, at least a constant fraction of entries are marked. Deleting all of them will require a list compaction which will require $O(\log n)$ time. Hence, we have to delete only some of the marked entries each time. Also, to ensure that we perform only $O(\log n)$ iterations of step 1, we must make sure that we delete at least a constant fraction of the marked entries each time.

(2) If we use $o(n)$ processors, then we should address the problem of reallocating the processors to the remaining entries such that the load on the processors is balanced.

We first handle the problem of selecting at least a constant fraction of the marked entries for deletion by finding a large subset of the marked entries that can be independently deleted. When two successive entries in an adjacency list have to be deleted at the same time, since the entries are allocated to different processors, there is a need for conflict resolution. To enable conflict-free, constant time deletion, we select, from the set D of entries marked for deletion, a set I of entries that are nonadjacent. The splicing out process now takes only $O(1)$ time (this is because for any two consecutive marked entries, only one of them is deleted and there are no conflicts among processors). Further, we find a large I such that $|I| = \Omega(|D|)$. Hence, we can still splice out a constant fraction of the total number of entries in $\mathcal{A}$ during each iteration of step 1, and the total number of operations performed during the entire algorithm is still $O(n)$. There are two ways to compute a large set I of nonadjacent entries:

(a) we can use deterministic coin tossing ([CV86] and find an $O(1)$ ruling set using $O(\log^* n)$ time and $O(n)$ processors, or

(b) we can use list ranking to rank consecutive entries and choose the odd (even) numbered entries using $O(\log n)$ time and $O(n/\log n)$ processors.

Using $O(n)$ processors and the first method to compute the set I, we get a parallel algorithm that runs in $O(\log n \log^* n)$ time. Although the processor-time product would then be $O(n \log n \log^* n)$, the total number of operations performed is still $O(n)$ as the size of the graph still decreases by a constant factor during each iteration.

4. Reducing the Processors and Reallocation

We reduce the processors to $O(n/\log n \log^* n)$ while maintaining the running time at $O(\log n \log^* n)$ by employing Cole and Vishkin's technique of accelerating cascades (see [CV86], [CV86a] and [CV86b]). Our technique is quite similar to one used in an earlier paper by Chrobak and Eppstein ([CE91]). We compute the crosslinks in three phases. In the first two phases, we have fewer processors than the entries in the list. After the first two phases, the number of processors equals the number of entries and we use the algorithm described in the previous section during the third phase. The third phase, therefore, takes $O(\log n \log^* n)$ time. During the first two phases, each processor is allocated a set of entries to process and each processor sequentially processes these items. For every iteration during phases I and II, the size of the graph decreases by a constant factor. To balance the load on the processors, we periodically perform processor reallocation. We describe the first two phases below.

At the start of phase I, we have $O(n/\log n \log^* n)$ processors and $O(n)$ entries. Hence each processor is allocated $O(\log n \log^* n)$ entries. The entries are first allocated to the processors as follows: we first combine all the adjacency lists into a single long list L (we merely link the end of A_v to the front of A_{v+1}, for $1 < v < n$). We list rank L and place the entries in L in an array C in positions indexed by their ranks in L. We divide the entries in C into groups of $O(\log n \log^* n)$ consecutive entries and allocate the i-th such group to processor p_i. During the course of the algorithm, during reallocation, we simply compact C (to get rid of the deleted entries) while maintaining the relative order of the remaining entries and allocate groups of consecutive entries to processors. We note that by doing this, we ensure that there are at most two entries per processor p_i with a neighbor in its adjacency list allocated to a different processor. This is important for resolving conflicts, as we will see shortly.

We run $O(\log^* n)$ iterations of the algorithm during phase I. After each iteration, we reallocate the processors among the remaining entries. The total time involved in phase I is $O(\log n \log^* n)$. Table 1 describes the various parameters involved during phase I. During each iteration in phase I, each processor performs some sequential work (i.e., simply processes the entries allocated to it sequentially). During this sequential processing stage, any entry that needs to be spliced out can be spliced out in constant time provided the neighbors of the entry are both allocated to the same processor as the entry itself. Otherwise, there may be a need for symmetry breaking, which is achieved (after the sequential stage) by a routine that

finds a large set of nonadjacent entries among the remaining marked entries; this routine can be executed in $O(\log n)$ time (using method (b) mentioned in the previous section) during each iteration in phase I because there are at most $O(1)$ entries per processor that may require conflict resolution. The sequential work decreases exponentially with the number of iterations as the number of entries allocated to each processor decreases exponentially with the number of iterations, and the rest of the work involves $O(\log n)$ time during each iteration. When summed over the $O(\log^* n)$ iterations, the time for the sequential part converges to $O(\log n \log^* n)$ and hence the total time (not including the time for processor reallocation) amounts to $O(\log n \log^* n)$.

Iteration	#Entries	#Entries/Processor	Processing Time	Reallocation Time
1	$c_1 n$	$c_2 \log n \log^* n$	$c_3 \log n \log^* n +$ $c_4 \log n$	$c_5 \log n \log^* n +$ $c_6 \log n$
2	$c_1 n/k$	$(c_2 \log n \log^* n)/k$	$(c_3 \log n \log^* n)/k +$ $c_4 \log n$	$(c_5 \log n \log^* n)/k +$ $c_6 \log n$
o o o				
i+1	$c_1 n/k^i$	$(c_2 \log n \log^* n)/k^i$	$(c_3 \log n \log^* n)/k^i +$ $c_4 \log n$	$(c_5 \log n \log^* n)/k^i +$ $c_6 \log n$
o o o				
After $O(\log^* n)$ iterations	$c_1' n/\log^* n$	$c_2' \log n$	$c_3' \log n \log^* n$ (total time)	$c_5' \log n \log^* n$ (total time)

Table 1 : Shows allocation of entries to processors and the times involved during phase I.
$c_1, c_2, c_3, c_4, c_5, c_6, k, c_1', c_2', c_3'$, and c_5' are constants.
The values shown are all upper bounds.

Processor reallocation is done by compacting the array C (mentioned two paragraphs ago) using a simple prefix sum computation. The prefix sum computation involves a sequential computation part (i.e., where each processors performs some sequential work on its allocated entries), followed by a parallel computation part. The time for the sequential

part decreases with the size of the graph and hence sums up to $O(\log n \log^* n)$ and the parallel computation part takes $O(\log n)$ time each time. During phase I, processor reallocation time is done after every iteration and takes $O(\log n \log^* n)$ total time.

At the start of phase II, the number of entries remaining is $O(n/\log^* n)$. During each iteration, every processor works on the entries allocated to it sequentially and the time for this sequential part decreases with the size of the graph. During the sequential part, each processor establishes crosslinks, wherever possible, in its allocated entries and deletes entries that can be deleted without conflicts. After the sequential part, as in phase I, a large set of nonadjacent entries among the deletable entries with conflicts is found (this time, using method (a) mentioned in the previous section). Since the number of entries in conflict is $O(1)$ per processor, the time spent in finding the large set of nonadjacent entries is $O(\log^* n)$ per iteration. We run $O(\log \log n)$ iterations of the algorithm during phase II. During each iteration, the graph size decreases by a constant factor and hence after $O(\log \log n)$ iterations, the number of entries decreases from $O(n/\log^* n)$ to $O(n/\log n \log^* n)$ equaling the number of processors.

Processor reallocation is a little difficult in phase II as compared to phase I. We cannot perform the prefix computation during each of the $O(\log \log n)$ iterations of phase II because this will result in a total time of $O(\log n \log \log n)$ which is beyond our bound of $O(\log n \log^* n)$. Hence, we perform only $O(\log^* n)$ processor reallocations at carefully chosen intervals such that the total time used is still $O(\log n \log^* n)$. Table 2 shows the parameters involved in the first few iterations of phase II and suggests our idea. We explain the details briefly now.

During the first iteration of phase II, we reallocate the entries among the processors. We will call this the 0-th reallocation. After the first iteration of phase II, we have $O(n/k' \log^* n)$ remaining entries, for some constant $k' > 1$. We now run b iterations of the algorithm without performing any processor reallocations, where b is a constant such that $1 < b < k'$. Then, we compact C and perform processor reallocation. The number of remaining entries at this time is $O(n/k'^b \log^* n)$. The next reallocation is performed after b^b iterations of the algorithm run without any processor reallocation. In general, the $(i+1)$-th reallocation is performed b^{t_i} iterations after the ith reallocation, where t_i is a tower of b's with a height equal to i. (If b is not an integer, then we perform $\lfloor (b^{t_i}) \rfloor$ iterations.) During each iteration, the number of unmarked entries decreases by a factor of k'. Hence, after the i-th reallocation, there will be $O(n/k'^{t_i} \log^* n)$ items left in the adjacency lists. It is easy to see that after $O(\log^* n)$ reallocations (i.e., when the height of the tower t_i is $O(\log^* n)$), the number of entries left equals the number of processors. Each reallocation takes $O(\log n)$ time and hence the total time for reallocations is $O(\log n \log^* n)$.

The number of entries sequentially processed per iteration by each processor between the i-th and $(i+1)$-th reallocations is $O(\log n/k'^{t_i})$, which is also the time required to process these entries. The time required to find the large set of nonadjacent entries among those with conflicts is $O(\log^* n)$ per iteration. However, b^{t_i} iterations are performed during this interval without any reallocation and therefore, the time involved between the i-th and $(i+1)$-th reallocations is $O(b^{t_i} \log n/k'^{t_i} + b^{t_i} \log^* n)$. Since we perform $O(\log^* n)$ reallocations in phase II (actually, we perform the reallocations until $k'^{t_i} = O(\log n)$)

and because $b < k'$, the time taken by the part of the algorithm including the time for reallocations is $O(\log n \log^* n)$.

Iteration	#Entries	#Entries per Processor	Processing Time	Reallocation Time
1	$d_1 n/\log^* n$	$d_2 \log n$	$d_3 \log n +$ $d_4 \log^* n$	$d_5 \log n +$ $d_6 \log n$
2	$d_1 n/(k' \log^* n)$	$(d_2 \log n)/k'$	$d_3 \log n/k' +$ $d_4 \log^* n$	–
o o o				
$b+1$	$d_1 n/(k' \log^* n)$	$(d_2 \log n)/k'$	$d_3 \log n/k' +$ $d_4 \log^* n$	$d_5 \log n/k' +$ $d_6 \log n$
$b+2$	$d_1 n/(k'^b \log^* n)$	$(d_2 \log n)/k'^b$	$d_3 \log n/k'^b +$ $d_4 \log^* n$	–
o o o				
b^b+b+1	$d_1 n/(k'^b \log^* n)$	$(d_2 \log n)/k'^b$	$d_3 \log n/k'^b +$ $d_4 \log^* n$	$d_5 \log n/k'^b +$ $d_6 \log n$
b^b+b+2	$d_1 n/(k'^{b^b} \log^* n)$	$(d_2 \log n)/k'^{b^b}$	$d_3 \log n/k'^{b^b} +$ $d_4 \log^* n$	$d_5 \log n/k'^{b^b} +$ $d_6 \log n$
o o o				

Table 2 shows what happens during a few iterations of phase II. d_1, d_2, d_3, d_4, d_5, and d_6 are constants. $b, k > 1$ are constants and $b < k$.

5. A Faster CRCW PRAM Algorithm

In the previous algorithm, the graph size decreases by a constant factor after each iteration and hence $O(\log n)$ iterations are sufficient to completely process the graph. However, processing within each iteration takes more than constant time and hence the total time turns out to be $O(\log n \log^* n)$. If we are able to perform the computations within each iteration in $O(1)$ time and if we can reallocate the processors quickly, then we can get a better running time. Since prefix sum computations can be performed in sublogarithmic time ([CV86] and [RR89]) using a CRCW PRAM, the processor reallocation part seems possible. We show here how the main algorithm can be performed so that the total time used is $O(\log n)$ and how this can be combined with a prefix sum computation method using a CRCW PRAM for processor reallocation so that we can perform crosslinking in $\mathcal{A}$ in $O(\log n)$ time optimally.

After an iteration in the main algorithm, the following happen:

(1) For a constant fraction of the vertices, we would have established crosslinks for all the entries in their adjacency lists and we do not have to process these vertices again for the rest of the algorithm if we can remove all their entries immediately.

(2) For a constant fraction of the rest of the entries in $\mathcal{A}$, we would have established crosslinks and marked the entries for deletion from their respective adjacency lists.

In the sequential algorithm, we removed all the marked entries immediately, and we were then left with a smaller graph on which we ran the algorithm again. In parallel, removing the marked entries of the type mentioned in (1) above is easy but removing all the marked entries of the type mentioned in (2) above at once takes $O(\log n)$ time and in the previous parallel algorithm, we found a large subset of such marked entries such that the chosen subset can be deleted at once. This reduced the running time of our algorithm from $O(\log^2 n)$ to $O(\log n \log^* n)$, but still cost us a factor of $O(\log^* n)$ over the preferred $O(\log n)$ running time.

Call a nonempty adjacency list $\mathcal{A}[v]$ *active*, if the number of unmarked entries on that list is $\leq 4c$. Since $G \in \mathcal{G}$, and since all the subgraphs produced during the algorithm are subgraphs of G, a large fraction of the nonempty adjacency lists remaining in the graph are active during any iteration. A list that was not active during one iteration may become active during the next iteration because of the marking (or deletion) of a sufficient number of its entries. To be able to process the entire graph in $O(\log n)$ iterations, we should make sure that the number of active lists during an iteration is more than a constant fraction of the number of remaining nonempty adjacency lists. As for those lists that don't become active in the current iteration, we can delay the deletion process for some or all of their entries until these lists are about to become active. Since a list that is about to become active will have less than $4c$ unmarked entries in it (no matter what the original number of entries), we can determine, in $O(1)$ time, if a list is about to be active or not. To do this, we allocate an array of size $4c + 1$ to each adjacency list. Let us consider the case when we have allocated one processor per entry. Then, each processor allocated to an unmarked entry attempts to write into these $4c + 1$ locations, one location at a time. Conflicts are resolved arbitrarily. Processors that fail to write in a location will attempt to write in the next location, until all locations have been written in. Hence, after $4c + 1$ attempts, if all

the locations have been filled, the size of the list must be larger than $4c$ and hence the list is not active. If no more than $4c$ locations are filled, then the list becomes active. Using $O(n)$ processors, this procedure takes just $O(1)$ time and hence using $O(n/\log n)$ processors, it takes $O(\log n)$ time provided entries are allocated among the processors properly. We use Algorithm 2 to solve the cross-linking problem.

Algorithm 2: Parallel CRCW Algorithm for Crosslinks Using $O(n/\log n)$ Processors
Input: An adjacency list representation $\mathcal{A}$ of a graph G and a parameter c
Main:

1. List rank the set of all adjacency lists treating it as a single long list; copy the set of adjacency lists into an array C such that an entry with rank i is placed into location $C[i]$;
2. Allocate the entries in the array C evenly to the $O(n/\log n)$ processors such that each processor gets a set of consecutive entries in the adjacency lists /* if the number of entries is less than the number of processors, we allocate one processor per entry*/
3. For each processor p do

   For each entry $e = (v, w)$ allocated to p do /* w is an entry in the adjacency list of v */

   3.1 if the adjacency list of w has size $\leq 4c$ then scan through the list and find the entry for v;

   3.2 establish crosslinks between the two entries;

   3.3 mark both the entries for removal;

   endfor;
4. call Subroutine Cleanup to remove marked entries and compact the array C;
5. If there are no more lists to consider, place back the deleted entries in the exact reverse order in which they were deleted; else go to step 2;

end Main;

Subroutine Cleanup:

for each remaining vertex v let L_v be an array of size $4c+1$ attached to v's adjacency list;

1. for each processor p do

   for each entry (v, w) allocated to the processor do If the entry is marked, and if there are no conflicts, delete the entry; else if the entry is not marked, repeatedly attempt to write entry into the next available location in L_v until a successful write is achieved or all $4c+1$ locations of L_v have been written into;
2. for all vertices v where L_v has at least one unwritten location do:

   2.1 sort the entries in L_v according to their rank in the original adjacency list of v;

   2.2 From the adjacency list of v, splice out any entries between consecutive entries in L_v;
3. using a prefix sum computation algorithm, compact the array C to contain just the undeleted entries; /* This step is not necessary if the number of entries is no more than the number of processors */

end Subroutine Cleanup;

Theorem 2: Algorithm 2 establishes crosslinks in the adjacency lists $\mathcal{A}$ of a *linearly contractible* graph G in $O(\log n)$ time using $O(n/\log n)$ processors.

Proof: We first discuss Subroutine Cleanup. As mentioned earlier, in constant time (with $O(n)$ processors), we can determine if a list is about to become active in the next iteration or not. In Subroutine Cleanup, the entries in L_v are filled up arbitrarily. We sort these entries (step 2.1) according to their original ranks. It is now an easy task to remove any marked

entries in the adjacency list of v that are between consecutive entries in the sorted list L_v. As for the lists that do not become active in the next iteration, the number of unmarked entries is larger than $4c$. In such a list, we let each processor associated with the list, as it traverses through the list of its allocated entries, splice out every entry that can be deleted without a conflict. The marked entries that cause conflicts are those that are consecutive in an adjacency list and are allocated to different processors. Because we allocate a set of consecutive entries to each processor, there are only $O(n/\log n)$ such entries. We just don't delete any of these entries and leave them for later deletion. This will mean that processors will have to perform extra work during each iteration by scanning through these entries as well, but there will only be $O(1)$ extra work each time, per processor.

Processor reallocation is performed by compacting the array C of entries in the adjacency lists to just contain the entries not deleted from $\mathcal{A}$. We mark the deleted entries with a 0 in C and the remaining entries with a 1. A prefix sum computation performed on these 0's and 1's will be sufficient to compact C. Prefix sum computation can be performed optimally (on an array consisting of $O(n)$ integers of $O(\log n)$ bits each) in $O(\log n/\log\log n)$ time on a CRCW PRAM [CV86]. We perform the prefix sum computation in two stages: in the first (sequential) stage, each processor scans through its allocated entries in array C and computes their prefix sums. We then calculate the prefix sums of the $O(n/\log n)$ sums calculated from the first stage using the sublogarithmic algorithm [CV86]. After the first stage, we have $O(n/\log n)$ elements for which we need to calculate prefix sums and we have $O(n/\log n)$ processors, and hence the algorithm due to Cole and Vishkin can be easily applied. We note that the processor reallocation needs to be performed only during the first $O(\log\log n)$ iterations of the algorithm as the number of entries remaining during subsequent iterations will be $O(n/\log n)$.

Time: The number of entries in $\mathcal{A}$ to begin with is $O(n)$. During each iteration, at least half the remaining unmarked entries are marked. We will consider the time by considering the algorithm in two stages. The first stage is when processor reallocation is performed, i.e., when the number of remaining entries is more than the number of available processors. The duration of the first stage is $O(\log\log n)$ iterations as during each iteration, at least half the remaining entries are marked and almost all the marked entries (excepting $O(1)$ entries per processor) are deleted. The second stage is when no processor reallocation is required.

During the first stage, in iteration i, each processor is allocated $O(\frac{1}{2^i}\log n)$ entries and step 3 of Main takes $O(\frac{1}{2^i}\log n)$ time. In Subroutine Cleanup, steps 1 and 2 take $O(\frac{1}{2^i}\log n)$ time (note that all the L_v's together have $O(\frac{n}{2^i})$ entries). The prefix sum computation is performed in two stages, the first stage of which is sequential (for each processor) and takes $O(\frac{1}{2^i}\log n)$ time and second stage takes $O(\log n/\log\log n)$ time. The total time, therefore, is:

$$\sum_{i=1}^{O(\log\log n)} (O(\frac{1}{2^i}\log n) + O(\log n/\log\log n))$$

$$= (\sum_{i=1}^{O(\log\log n)} O(\frac{1}{2^i}\log n)) + O(\log n) = O(\log n)$$

During the second stage, step 3 of Main and steps 1 and 2 of Subroutine Cleanup take $O(1)$ time during every iteration. This is because each processor is now allocated $O(1)$ entries. Hence, the overall time of Algorithm 2 is $O(\log n)$. ■

The algorithm for prefix computation due to Cole and Vishkin is rather complicated and is not very practical. There is another algorithm due to Rajasekharan and Reif ([RR89]) that can perform prefix sums optimally using a CRCW PRAM in $O(\log n/\log\log\log n)$ time. This is a simpler algorithm and is quite practical. The only problem with using this algorithm directly is that since we have to perform prefix sums during the first $O(\log\log n)$ iterations of the algorithm, our algorithm will no longer be optimal. However, by using a technique similar to the one used in the optimal $O(\log n \log^* n)$ time CREW algorithm described earlier, we can perform the prefix sums at only some selected intervals calculated just like in the earlier case and achieve optimality.

6. Conclusion

We have presented optimal parallel algorithms for establishing crosslinks between the two copies of each edge in the adjacency list representation of linearly contractible classes of graphs. The first algorithm takes $O(\log n \log^* n)$ time using $O(n/\log n \log^* n)$ processors on a CREW PRAM. The second algorithm takes $O(\log n)$ time using $O(n/\log n)$ processors on a CRCW PRAM.

We remark that for bounded degree graphs, crosslinks can be established in $O(1)$ time using $O(n)$ processors and hence in $O(\log n)$ time using $O(n/\log n)$ processors by a simple algorithm, where we perform just one iteration of the sequential algorithm for the linearly contractible class of graphs presented in this paper. (Note that neither one of the class of linearly contractible graphs and the class of bounded degree graphs is contained in the other.)

Acknowledgements

I would like to thank Prof. George Lueker very much for the immense help he provided with this paper. Without him, this paper would not have been possible. I would also like to thank Prof. David Eppstein and Prof. Marek Chrobak for useful suggestions and hints.

References

[BDHPRS89] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, and S. Saxena. Improved Deterministic Parallel Integer Sorting. *Information and Computation*, 94, pages 29–47, 1991.

[Bren74] R. P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *J. Assoc. Comput. Mach.* 21, pages 201–206, 1974.

[CNS81] N. Chiba, T. Nishizeki, and N. Saito. A Linear 5-Coloring Algorithm of Planar Graphs. *J. Algorithms* 2, 4, pages 317–327, 1981.

[CE91] M. Chrobak and D. Eppstein. Planar Orientations with Low Out-degree and Compaction of Adjacency Matrices. *Theoretical Computer Science*, 86:243-266,1991.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70, pages 32–53, 1986.

[CV86a] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17, pages 128–142, 1988.

[CV86b] R. Cole and U. Vishkin. Approximate parallel scheduling .2. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92, pages 1–47, 1991.

[CV86] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81, pages 334–352, 1989.

[GPS87] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel Symmetry-breaking in Sparse Graphs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 315–324, 1987.

[Hage90] T. Hagerup. Optimal Parallel Algorithms on Planar Graphs. *Information and Computation*, 84, pages 71–96, 1990.

[Hage91] T. Hagerup. Constant-Time Parallel Integer Sorting. *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 299–306, 1991.

[HCD87] T. Hagerup, M. Chrobak, and K. Diks. Optimal 5-coloring of Planar Graphs. *SIAM Journal on Computing*, 18, pages 288–300, 1989.

[LF80] R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *J. Assoc. Comput. Mach.* 27, pages 831–838, 1980.

[RR89] S. Rajasekaran and J. H. Reif. Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms. *SIAM Journal on Computing*, 18, pages 594–607, 1989.

[TV85] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM Journal on Computing* 14, 4, pages 862–874, 1985.