Parallel Algorithms I

Tight Bounds on Parallel List Marking

Sandeep N. Bhatt[1], Gianfranco Bilardi[2], Kieran T. Herley[3], Geppino Pucci[2], Abhiram G. Ranade[4]

[1] Bell Communications Research, Morristown NJ 07960, USA
[2] Dip. di Elettronica e Informatica, Univ. di Padova, Padova 35131, Italy
[3] Dept. of Computer Science, Univ. College Cork, Cork, Ireland
[4] Computer Science Div., Univ. of California at Berkeley, Berkeley CA 94720, USA

Abstract. The list marking problem involves marking the nodes of an ℓ-node linked list stored in the memory of a (p, n)-PRAM, when only the location of the head of the list is initially known. Under the assumption that memory cells containing list nodes bear no distinctive tags distinguishing them from other cells, we establish an $\Omega(\min\{\ell, n/p\})$ randomized lower bound for ℓ-node lists and present a deterministic algorithm whose running time is within a logarithmic additive term of this bound. In the case where list cells are tagged in a way that differentiates them from other cells, we establish a tight $\Theta\left(\min\left\{\ell, \ell/p + \sqrt{(n/p)\log n}\right\}\right)$ bound for randomized algorithms.

1 Introduction

Linked structures are widely used in non numerical as well as sparse numerical computations. Therefore, it is important to ascertain whether parallelism can be exploited to process such structures effectively.

In this paper, we focus on lists, possibly the simplest type of linked structures, and on a very basic operation, which we call *marking*, consisting of writing a given value in each node of a given list. The essence of marking is that each node in the list has to be affected and no other. This feature is common to several basic list operations such as searching an element or ranking all nodes (determining their distance from the head). Marking itself is used in important practical applications, such as garbage collection, for identifying active structures in a large memory heap.

The complexity of parallel list operations crucially depends on the list representation, and is often affected by features that are irrelevant to sequential complexity. When managing lists in parallel, a favourable case arises if the the growth process affords keeping all list nodes in a compact region of memory. Specifically, the list could be represented as an array of ℓ records, each record corresponding to a list node, with a field storing the array index of its successor. Indeed, most list-based parallel algorithms in the literature (e.g., searching and ranking [2]) do assume such compact representation.

In other scenarios, unfortunately, list nodes become naturally scattered throughout a portion of memory whose size is much larger than the length of the list.

This case arises when a sequence of concatenations and splittings is performed on a set of lists.

We also distinguish between tagged and untagged lists, a tagged list being one where each node contains a *tag* that uniquely identifies the list. Tags can be maintained with small overhead if lists are modified only by insertion and deletion of nodes. However, the overhead is not negligible if other operations, such as concatenation and splitting, are allowed.

We investigate the extent to which parallelism, randomization, and tagging can be profitably exploited to improve upon sequential performance when lists are scattered throughout the memory. Specifically, we develop deterministic and randomized upper and lower bounds for marking a (tagged or untagged) list of ℓ nodes stored in the memory of a p-processor PRAM with n memory cells, when only the location of the head of the list is initially known.

1.1 Related Work and New Results

A restricted version of the list marking problem was introduced and analyzed by Luccio and Pagli in [7]. Under the assumption that list elements are distinguishable from non-list elements by inspection, the authors prove a deterministic $\Omega(min\{\ell, n/p\})$ lower bound and provide a tight upper bound when $p = O(\ell/\log \ell)$ and $n = O(\ell \log \ell)$. In this paper we improve and generalize these results in the following directions:

1. In the case where list elements are indistinguishable from non-list elements, we prove that an $\Omega(\min\{\ell, n/p\})$ lower bound also holds for randomized algorithms. Moreover, we give a deterministic algorithm optimal to within a logarithmic additive term, therefore showing that randomization can not be exploited in any significant way in this setting.
2. In the case where list elements are tagged in a way that makes them recognizable by inspection, we establish a tight $\Theta\left(\min\left\{\ell, \ell/p + \sqrt{(n/p)\log n}\right\}\right)$ bound for randomized algorithms, showing that, for a wide range of list lengths, considerable speedups can be attained by means of randomization.

1.2 Preliminaries

We will assume that each memory cell has the same format and contains a memory address which will be interpreted as a pointer (called the *successor pointer*) to another cell, a *tag field*, capable of holding a distinctive symbol, a *data field*, and a small constant amount of additional space, called *scratch space*. The head of the list, denoted by h, occupies cell 0 and its data field contains some arbitrary symbol which we will refer to as the *signature* of the list. Finally, each node points to the next node in the list, and the pointer field of the last node r contains the address of cell 0, which we will interpret as a *nil* pointer.

We identify two variants of the problem. The list is *untagged* if list nodes bear no distinctive mark or symbol that renders them instantly identifiable as such. The list is *tagged* if each list node bears a distinctive symbol in its tag field which no non-list node bears, thus allowing list nodes to be identified by inspection. In both cases, the goal of the list marking problem is to copy the signature into the data field of every node in the list; the data fields of all other nodes should remain unchanged. A node is said to be *marked* once its data field bears the appropriate signature.

Since each memory cell contains a successor pointer, the entire memory can be interpreted as a directed graph G of n nodes. Each node must have outdegree zero or one, but a node (including list nodes) may have indegree zero (*leaves*), one (*unary nodes*), or higher. Such a graph is known as a *pseudoforest*. (Such structures feature in some connected component algorithms, for example [5].) Within the pseudoforest, the chain of list nodes forms a directed path in a structure T that we may interpret as a tree, the edges of which are oriented from child to parent. The node h is a leaf of T and the list nodes are the ancestors of h in T located along the directed path from h to r, the root of T. In fact, G consists of T plus a collection of node-disjoint components each of which is either a tree or one or more trees joined by a cycle connecting their roots. In this setting, the objective is to mark all those nodes in G that are ancestors of h in T.

The algorithms presented here all assume the ARBITRARY CRCW variant of the PRAM model of shared-memory computation [5]. Thus concurrent reads and writes are permitted. Whenever a number of processors attempt to write simultaneously to a cell, one of them, chosen arbitrarily, succeeds, while the others fail. For convenience, we will refer to a PRAM with p-processors and n-cells of memory as a (p, n)-PRAM and will assume throughout that $p \leq n$. We will also assume that each processor has a private area of $O(1)$ storage for workspace.

In Section 2 we investigate the problem of marking untagged lists and present a randomized lower bound and an almost matching deterministic upper bound. Section 3 deals with the tagged case. Section 4 offers some concluding remarks.

2 Marking Untagged Lists

In this section we determine the complexity of the untagged variant of the list marking problem. In Subsection 2.1, we prove that any randomized *Las Vegas* algorithm for the problem requires $\Omega(\min\{\ell, n/p\})$ time with high probability. In Subsection 2.2, we give a deterministic algorithm whose running time is within an additive logarithmic term of the lower bound, thereby proving that randomization can not be conveniently exploited in this case.

2.1 A Randomized Lower Bound

The intuition behind the lower bound is that a list element becomes distinguishable from a non-list element only when every element along the directed path from the head of the list to that element has been marked. Therefore random probes of the memory cells will not speed-up the computation in any significant way. This argument is formalized in the following theorem.

Theorem 1. *Suppose that with probability* $1 - o(1)$ *a randomized parallel algorithm on a* (p,n)*-PRAM marks every element of a list of length* ℓ *within* t *time steps. Then* $t = \Omega(\min\{\ell, n/p\})$.

Proof. Observe that a randomized algorithm can be seen as one chosen uniformly at random from a set $\mathcal{D}$ of deterministic algorithms (each deterministic algorithm being characterized by the outcome of a sequence of random choices). In order to prove our lower bound for the untagged case, we construct a set of inputs with the property that *every* algorithm in $\mathcal{D}$ fails to mark the list in less than the time prescribed by the lower bound for a constant fraction of the inputs. From this, it immediately follows that, for some input in the set, a constant fraction among all the deterministic algorithms fail to mark the list in the prescribed time. Therefore, the failure probability of a randomly chosen algorithm, on that particular input, is bounded below by a positive constant.

We will assume that ℓ is fixed and will restrict our attention to the following set of inputs. The contents of the memory are organized as a circular list of length n. The target list of length ℓ is stored as a contiguous sublist, and the address of the head and tail of the target list is given as input to the algorithm. There are $n!$ different inputs, corresponding to the $(n-1)!$ different circular lists of length n and the choice for the address of the head of the target list. This formulation of the problem is essentially equivalent to that presented in the introduction, but more convenient in the current context.

Without loss of generality, suppose that $t \le \frac{n}{2p}$. At any time step, the algorithm probes a set of at most p memory cells. At step $i \le t$, we can think of the nodes on the circular list to be grouped into *sublists.* A sublist consists of a maximal set of adjacent probed nodes terminated by an unprobed node. According to this definition, we initially have n sublists, each consisting of a single distinct unprobed node. When a node v is probed, its pointer to the next element v' in the list becomes known, and their corresponding sublists merge. We will refer to the sublist containing the head of the target list as the *principal* sublist. This sublist will contain a prefix of the target list that grows in length as the algorithm executes. Notice that each step of the algorithm causes up to p merges and hence at least $n - pi$ sublists remain after i steps.

At the beginning of the i^{th} step, suppose that the nodes are partitioned into a total of k_i sublists of various lengths (including the principal sublist), and let n_j^i denote the number of nonprincipal sublists of length j. Note that $\sum_{j=1}^{n} n_i^j = k_i - 1$. For convenience, we assume that each step consists of a first substep during which $p-1$ arbitrary cells are probed, followed by a second substep when the tail of the principal sublist is probed, which has the effect of grafting a single sublist onto the end of the principal sublist. Clearly, conforming to this discipline will not alter the running time of an algorithm by more than a constant factor. The merges provoked by the $p-1$ probes of the first substep yield k_{i+1} nonprincipal sublists. With the possible exception of the single sublist that will be grafted onto the principal sublist during the second substep, there are at most n_j^{i+1} sublists of length j. One of these sublists is grafted onto the principal sublist during this step, and because all input lists are equally likely, each of these sublists is equally likely to be chosen. Thus, the expected value of δ_i, the increase in the length of the principal sublist during step i, is bounded as follows

$$E[\delta_i] \le \sum_{j=1}^{n} \frac{jn_j^{i+1}}{k_{i+1}} + \frac{n}{k_{i+1}} \le \frac{2n}{n-pi} \le 4 ,$$

since $k_{i+1} \ge n - pi \ge n/2$. (Note the sum accounts for all but one of the nonprincipal sublists; the term n/k_{i+1} accounts for the contribution of the remaining one.)

Therefore, only constant progress is made, on average, at each step on the prefix of the list, and the proof follows.

Note that the above bound is obtained under the optimistic assumption that a node belonging to the target list is marked as soon as all the other nodes between the head of the target list and that node are probed, even though the algorithm, by the time it touches the corresponding memory cells, may not have sufficient information to determine that these cells actually contain list elements.

2.2 A Deterministic Upper Bound

We begin by outlining a relatively simple but slightly inefficient deterministic algorithm for the untagged list marking problem. We then provide a fast technique to transform the input instance into an equivalent, smaller one so that the running time of the algorithm on the new instance is within the desired bound. The "shrinking" process is accomplished by deleting nodes and rearranging edges of the pseudoforest underlying the original input instance.

Consider an untagged list of ℓ nodes stored in the memory of a (p, n)-PRAM, and let h be the (given) distinguished pointer to the head of the list. As we observed in Section 1.2, the nodes to be marked are the ancestors of h in a tree T whose root is r, the tail of the list. Suppose that we are given the preorder and postorder number of every node in T. Then, a particular node x is an ancestor of h if and only if $preorder(x) \le preorder(h)$ and $postorder(x) \ge postorder(h)$. Although the Euler-tour techniques of Tarjan and Vishkin[8] can be used to efficiently compute the preorder and postorder numberings in trees, these may not be applied immediately in the present context. Firstly, the presence of other components in the pseudoforest may complicate matters, and secondly the techniques rely on an adjacency list representation for trees.

We circumvent the first difficulty as follows. Using a straightforward pointer-jumping technique, each node in T can identify the root r in $O(\log n)$ time per node or $O((n/p)\log n)$ time overall. Nodes not in T executing the same algorithm will "converge" on some node other than r and will hence be clearly identifiable as not belonging to T. All such nodes will remain dormant for the remainder of the algorithm.

Having eliminated all nodes not in T, we may now construct an adjacency list representation for T. Label the i^{th} cell C_i with the pair $< s, i >$ where s is the address of its parent in T. By sorting the cells lexicographically using Cole's algorithm [1], the children of each node occupy adjacent positions, and so they may be easily linked together in an adjacency list for that node. (These linking pointers are distinct from the successor pointers for the nodes and are stored in the scratch storage associated with the nodes in question.) The cells are then resorted with respect to their original addresses in order to reconstruct the original structure of T and to attach adjacency lists to the appropriate tree nodes. The implementation details are straightforward. The sorting steps dominate the running time and so the adjacency list representation for T can be constructed in $O((n/p)\log n)$ time.

Given the adjacency list representation for T, the preorder and postorder numberings can be computed in $O((n/p)\log n)$ time. The identification and marking of the ancestors of h can be completed within the same time bounds. Interleaving this $O((n/p)\log n)$ algorithm with the obvious $O(\ell)$ sequential algorithm, we obtain the following result.

Proposition 2. *An untagged list of length ℓ in the memory of an (p, n)-PRAM can be marked deterministically in $O(\min\{\ell, (n/p)\log n\})$ time.*

We now proceed to extend Proposition 2 to provide a more work-efficient algorithm. Our algorithm will consist of a sequence of *pruning* steps applied to the pseudoforest G. Each pruning step will be applied to the result of the previous one and each will reduce the size of the problem (the number of nodes in the pseudoforest) by a constant factor by deleting some nodes and rearranging pointers among the active (undeleted) nodes. After a certain number of stages, the original pseudoforest G will have been reduced to one G' of significantly smaller size. Because of its smaller size, it is faster to identify and mark the ancestors of h in G' than in G using the techniques of Proposition 2. Moreover, the ancestors of h in G' are among the ancestors of h in G and so the marking of G' can be extended to mark all the ancestors of h in G.

The following lemma provides the basis for pruning steps.

Lemma 3. *There is a constant $\mu < 1$ such that for any pseudoforest $H = (V, E)$ containing h, there is a subset $W \subseteq V$ consisting entirely of leaves and unary nodes, but not containing h, such that :*

(i) $|W| \geq \mu|V| - 1$;
(ii) *No pair of nodes in W are neighbours in H.*

Moreover, set W can be computed in $O(|V|/p + \log n/\log\log n)$ time.

Proof. Let $H = (V, E)$ be a pseudoforest. We show that there exists an independent subset W of V consisting entirely of leaves and unary nodes that has size at least $\mu|V| - 1$.

The nodes eligible for inclusion in W are the nodes in H of indegree at most one apart from h. This set induces a set of maximal node-disjoint linear chains of eligible nodes. Apply the following operation to each such chain. If the chain is of length one or two, select the first node; if the chain length has length three or more select a subset of the nodes such that (i) no two adjacent nodes are selected, and (ii) the maximum number of consecutive unselected nodes on the chain is two. The union of the selected nodes for the various chains forms the desired set W.

The identification of nodes in H of indegree at most one is straightforward and requires only constant work per node. The selection of nodes belonging to chains of length three or greater is instead accomplished by applying the 2-ruling algorithm of Cole and Vishkin [2] and selecting the nodes in the ruling. (The latter algorithm is formulated in terms of circular lists, but this is not an essential restriction.) This latter step can be completed in $O(|V|/p + \log n / \log\log n)$ time.

Consider a chain of length k. If $k \leq 2$ then, clearly at least $k/2$ of the nodes on the chain are selected. If $k \geq 3$, then each pair of selected nodes is separated by at most two unselected nodes. Allowing for the possibility that the first two nodes on a chain might be unselected, we see that the number of selected nodes is at least $\lceil (k-2)/3 \rceil \geq (1/9)k$.

If we let s denote the number of nodes in H of indegree at most one (including h), it is clear that $s \geq (|V|+1)/2$. However the $s-1$ eligible nodes are arranged into chains of length one, two, or greater, and so we are guaranteed that at least $(1/9)(s-1) \geq (1/18)|V| - 1$ nodes are selected.

Notice that it is easy to delete a node w in W from H by redirecting the edge incident on w (if any) to point to w's parent. (Recall that w has at most one child.) By the above lemma, the graph H' thus obtained contains significantly fewer nodes than H (at most $(1-\mu)|V|+1$). It is also easy to verify that for every pair of nodes x and y in H', x is an ancestor of y in H' if and only if x is an ancestor of y in H. Thus, while smaller in size, the graph H' retains some of the ancestor-descendent information of the original graph H.

Before we describe the implementation of the pruning step in greater detail, we must introduce some auxiliary data structures employed by the algorithm, the role of which will become clear in due course.

Each processor maintains a private stack that is empty before the first pruning. When a processor deletes a node during a pruning step, it pushes that node onto its stack. This facilitates the reconstruction of the graph at a later stage in the algorithm. Each processor also has a private list called its work list. Collectively the p work lists hold all the active nodes in the graph. Between pruning steps, nodes are redistributed among work lists to ensure that each processor's work list contains an equal number of items. A processor is responsible for performing whatever operations are required for the nodes on its work list during a pruning step.

It should be emphasized that the only space overhead for these stacks and work lists is $O(1)$ per processor for a header pointer: the objects in these structures are nodes linked by pointers. These linking pointers are distinct from the successor pointers of the nodes in question and are represented within the scratch space of the nodes.

The pruning step applied to $H = (V, E)$ may be described as follows.

1. Identify the set W.
2. Perform the following step for each active node x in W: mark x deleted, label the node with the current time and the name of its lone child, and push the node onto the local stack. For each active node c whose parent x is in W do the following: redirect c's successor pointer to point to x's parent, or to *nil* if x has no parent. (Each processor is responsible for the nodes on its own work list.)
3. Update the work lists.

From Lemma 3 it follows that Step 1 is completed in $O(|V|/p + \log n/\log\log n)$ time. Assuming for the moment that every processor list holds $O(|V|/p)$ items at the start of the pruning step, it is easy to see that Step 2 requires $O(|V|/p)$ time. To update the work lists in Step 3, each processor scans through its own list removing the deleted nodes and counting the active nodes. Let a denote the total number of active nodes. Using a straightforward combination of parallel prefix [3] and routine pointer manipulations, it is possible to redistribute the active nodes among the work lists so that each processor's list receives at most $\lceil a/p \rceil$ in $O(a/p + \log n/\log\log n)$ time. Thus, each stage can be completed in $O(|V|/p + \log n/\log\log n)$ time.

The following recurrence provides an upper bound governing the number of active nodes remaining active after the i^{th} pruning step:

$$a_i = \begin{cases} (1-\mu)a_{i-1} + 1, & \text{for } i > 0 \ , \\ n, & \text{for } i = 0 \ . \end{cases}$$

Thus, $a_i \le (1-\mu)^i n + \sum_{j=0}^{i-1}(1-\mu)^j$, which is bounded above by $\mu^{-1}n/\log n$ for $i \ge \lceil \log\log n/\log(1-\mu)^{-1} \rceil$. Selecting k to be this latter quantity, we see that the number of active nodes can be reduced to at most $\mu^{-1}n/\log n$ in

$$O\left(\sum_{i=0}^{k-1} \frac{(1-\mu)^i n + \mu^{-1}}{p} + k\log n/\log\log n\right) = O\left(n/p + \log n\right)$$

time.

In conclusion, the overall algorithm is as follows.

1. Apply k stages to G to produce G' of size at most $\mu^{-1}n/\log n$.
2. Apply the algorithm of Proposition 2 to mark all nodes in G' that are ancestors of h.
3. Reincorporate the nodes deleted during Step 1 in the reverse order to which they were deleted. In other words, first undo the deletions of the k^{th} pruning, then those of the $(k-1)^{st}$ pruning, and so on. For each reinserted node, mark it if its child is marked.

We have already noted that Step 1 runs in $O(n/p + \log n)$ time, and since Step 3 is similar, it too has the same time bound. By Proposition 2, Step 2 requires $O(((\mu^{-1}n/\log n)/p)\log n) = O(n/p + \log n)$ time.

By using the facts that an ancestor of a node x in G' is also an ancestor of x in G, and that only leaves and unary nodes are deleted, it is not difficult to see that all of the nodes marked by this algorithm are ancestors of h in G. On the other hand, suppose that some node along the directed path from h to r is not marked by the algorithm, and let x be the first such node. This node must have been deleted during one of the pruning operations in Step 1, otherwise it would have been marked during Step 2. Suppose that c was the lone child of x at the time that x was deleted. By assumption, the node c is marked by the algorithm, and so when x is reinserted during Step 3, it too would be marked.

The main result of this section is summarized in the following theorem:

Theorem 4. *An untagged list of length ℓ stored in the memory of a (p, n)-PRAM may be marked deterministically in $O(\min\{\ell, n/p + \log n\})$ time.*

3 Marking Tagged Lists

Recall that in a tagged list each node carries a special symbol in its tag field so that it can be distinguished from non-list elements by inspection. Although deterministic list-marking algorithms cannot exploit this property to improve upon the worst-case performance of Theorem 4 [7], we sketch a simple but optimal randomized strategy which takes advantage of these tags.

The randomized algorithm is quite simple and proceeds in two stages. In the first stage, each processor randomly accesses q memory locations and retains the addresses of those locations that contain list elements. Successful probes partition the original list into chains of nodes whose heads are marked and randomly distributed among the processors. Our intuition is that the qp random probes in the first stage will select list elements so that g, the length of longest chain, is sufficiently small. Once the list has been "chopped" this way, in second stage we invoke a standard randomized search algorithm [6] that marks all the list nodes while balancing the load among the processors in time $O(\ell/p + g)$, with high probability. Note that this idea is not new: Greene and Knuth [4] have analyzed it in the context of graph traversal, and Ullman and Yannakakis [9] have used it for searching graphs.

By interleaving the above strategy with the straightforward $O(\ell)$ sequential algorithm we can see that the list can be marked in $O(\min\{\ell, \ell/p + q + g\})$ time, with high probability. The following lemma illustrates the tradeoff between the parameters q and g, the two quantities that determine the running time of the algorithm.

Proposition 5. *Suppose that a tagged list of length ℓ is stored in the memory of a (p, n)-PRAM, into which t probes are made at random. Let random variable X denote the length of a longest contiguous subsequence of unprobed list elements. Then $Pr(X > g) < \ell e^{-tg/n}$.*

Proof. The probability that no probes are made within a fixed subsequence of length $g \leq \ell$ equals $(1 - g/n)^t < e^{-tg/n}$, and there are at most $\ell - g + 1$ such subsequences.

With p processors making a total of pq probes, we have $Pr(X > g) < \ell e^{-pqg/n}$. Setting $q = g = \sqrt{k(n/p)\log n}$ where k is an arbitrary positive integer, we can see that

$$Pr(X > \sqrt{k(n/p)\log n}) < \ell n^{-k} \leq n^{-(k-1)} \ .$$

Thus our algorithm runs in $O(\min\left\{\ell, \ell/p + \sqrt{(n/p)\log n}\right\})$ time, with high probability.

Next, we establish a lower bound that is within a constant factor of this upper bound.

Theorem 6. *For any randomized list marking algorithm, if the probability that it terminates in time t is at least $1 - n^{-k}, k > 0$, then*

$$t = \Omega(\ell/p + \min\left\{\ell, \sqrt{k(n/p)\log n}\right\}) \ ,$$

where ℓ is the length of the input list stored in a memory of size n.

Proof. First note that ℓ/p is a trivial work-based lower bound for the problem. Next, assume that in i steps the first i list elements are marked, for all $1 \leq i \leq \ell$. This assumption does not weaken the argument for the lower bound.

Let W_j be the event that each of the first $j+1$ list elements is marked within the first j steps. Also, let C_t be the event that every list element has been marked within t or fewer steps. Assuming for the moment that $t < \ell$, we can see $C_t \subseteq W_t$, which means that $Pr(C_t) \leq Pr(W_t)$ and, therefore $Pr(\overline{C_t}) \geq Pr(\overline{W_t})$.

Now, the probability that the $(i+1)st$ list element was touched by a random probe within the first i steps does not exceed pi/n. Hence, $Pr(W_i|\overline{W_{i-1}}) \leq pi/n$, and consequently, $Pr(\overline{W_i}|\overline{W_{i-1}}) \geq 1 - pi/n$. Combining this with the observation that $Pr(\overline{W_1}) = 1 - \frac{p}{n}$, we can see that

$$\begin{aligned} Pr(\overline{W_t}) &\geq \prod_{i=1}^{t}(1 - pi/n) \\ &\geq (1 - pt/n)^t \\ &= e^{\Omega(-pt^2/n)}, \end{aligned}$$

for $t > 1$.

Thus, if $Pr(\overline{C_t}) \leq n^{-k}$, it must be the case that $n^{-k} = e^{\Omega(-pt^2/n)}$, hence

$$t = \Omega(\sqrt{k(n/p)\log n}) \ ,$$

and the theorem follows.

4 Conclusions

The results of this paper are summarized in the following table.

Deterministic and Rand. Untagged	$O(\min\{\ell, n/p + \log n\})$ $\Omega(\min\{\ell, n/p\})$
Rand. Tagged	$\Theta\left(\min\left\{\ell, \ell/p + \sqrt{(n/p)\log n}\right\}\right)$

The table shows that in all cases, speed-ups over sequential performance can be obtained only for a number of processors larger than a certain threshold p_0. Namely, $p_0 = \Theta(n/\ell)$ and $\log n = o(\ell)$ in the deterministic untagged case, and $p_0 = \Theta(n \log n/\ell^2)$ for the randomized tagged case. Moreover, in the untagged case, the deterministic upper bound and the randomized lower bound match except for $p = \Omega(n/\log n)$, therefore randomization can not be exploited in any significant way. In the case of tagged lists, however, for ℓ in the range $\sqrt{(n/p)\log n} \leq \ell \leq n/p + \log n$ the speedup attainable by exploiting randomization can be considerable.

Finally, preliminary investigations indicate that aspects of the above behaviour remain when extending the algorithms for marking to other basic operations and/or to broader classes of linked structures.

Acknowledgments

This research was supported, in part, by the Istituto Trentino di Cultura through the Leonardo Fibonacci Institute, in Trento, Italy. Further research support for G. Bilardi and G. Pucci by MURST and CNR of Italy, and by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (project GEPPCOM); for K.T. Herley by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (project GEPPCOM).

The authors wish to thank the referees for their thoughtful reading of the paper and their many suggestions, which resulted in improvements of the manuscript.

References

1. R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.
2. R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.
3. R. Cole and U. Vishkin. Faster optimal prefix sums and list ranking. *Information and Computation*, 81(3):344–352, 1989.
4. D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkauser, Boston MA, 1982.
5. J. JaJa. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading MA, 1992.
6. R. M. Karp and Y. Zhang. A randomized parallel branch and bound procedure. In *Proceedings of the* 20th *Annual ACM Symposium on Theory of Computing*, pages 290–300, May 1988.

7. F. Luccio and L. Pagli. A model of sequential computation with pipelined access to memory. *Mathematical Systems Theory*, 26:343–356, 1993.
8. R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic time. *SIAM Journal on Computing*, 14(4):862–874, 1985.
9. J. D. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. In *Proceedings of the* 2^{nd} *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 200–209, July 1990.