

Distributed Graph Marking

Paul Hudak

JANUARY, 1983

RESEARCH REPORT #268

Copyright © 1983 Hudak

Work supported in part by the National Science Foundation
under Grant No. MCS-8106177.

Table of Contents

1 Introduction	1
2 Basic Concepts and Terminology	4
3 Parallel Graph Mutations	7
4 The Marking-Tree.	8
4.1 The First Algorithm (Scheme A)	9
4.2 Parallel Mutations.	10
4.3 Remarks	14
4.4 Performance	14
4.5 Correctness Proof	16
5 Using a Stack to Coordinate Local Marking	19
5.1 The Second Algorithm (Scheme B)	19
5.2 Parallel Mutations.	21
5.3 Remarks	22
5.4 Performance	24
6 Copying Nodes to Coordinate Local Marking	24
7 "Uncooperative" Graph Mutations	25
8 Conclusions.	28
9 Acknowledgements	29
I Examples of Functional Graph Mutations	30
II The Third Algorithm (Scheme C)	31

List of Figures

Figure 3-1:	Functional Mutator Primitives	8
Figure 4-1:	Scheme A: relying exclusively on the marking-tree	11
Figure 4-2:	Cooperation by the add-reference primitive (Scheme A)	12
Figure 4-3:	Cooperating Mutator Primitives for Scheme A	13
Figure 5-1:	Scheme B: Local marking with a stack	20
Figure 5-2:	Cooperation by the add-reference primitive (Scheme B)	22
Figure 5-3:	Cooperating Mutator Primitives for Scheme B	23
Figure 7-1:	Examples of "uncooperative mutations"	25
Figure 7-2:	Difficulty with the new add-reference primitive (Scheme A)	26
Figure 7-3:	A solution to the uncooperative mutation	27
Figure 10-1:	$S f g x \Rightarrow f x (g x)$	30
Figure 10-2:	$K x y \Rightarrow x$	30
Figure 10-3:	$Y f \Rightarrow f (Y f)$	31
Figure 11-1:	Scheme C: Local marking using a "copying collector" strategy	33
Figure 11-2:	Cooperation by the add-reference primitive (Scheme C)	34
Figure 11-3:	Cooperating Mutator Primitives for Scheme C	35

Distributed Graph Marking

Paul Hudak

YALE UNIVERSITY

Abstract

Three new algorithms are presented for marking a distributed directed graph, each demonstrating the feasibility of a system-wide, decentralized marking process. The algorithms are couched within a distributed processing model in which a graph is arbitrarily partitioned and distributed among the local stores of any number of autonomous processing elements that communicate only by messages. A class of functional graph mutations are shown to be able to execute concurrently with the algorithms without destroying their distributed nature, as long as certain invariants are maintained during execution. A class of mutations is also identified that precludes concurrent execution in a distributed manner. Two of the marking algorithms are especially practical and useful, one using a stack to coordinate local marking, the other behaving locally like a copying garbage collector. In all cases a tree structure known as the **marking-tree** is used to coordinate inter-processor marking, to allow cooperation by the mutator, and to detect termination. Applications include standard garbage collection, as well as more dynamic processes such as run-time deadlock detection and irrelevant task deletion.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems -- *distributed applications, network operating systems*; D.1.3 [Programming Techniques]: Concurrent Programming; D.4.4 [Operating Systems]: Communications Management -- *message sending, network communication*; G.2.2 [Discrete Mathematics]: Graph Theory -- *graph algorithms, network problems*

General Terms: algorithms, languages

Additional Key Words and Phrases: distributed algorithms, garbage collection, graph reduction

1. Introduction

Many computations can be modeled as the **transmutation** of a directed graph (which we will refer to as the **computation graph**). This is not only true for many LISP implementations, but also for data-flow and functional programming systems, in which the computation reflects a **graph reduction semantics**; that is, a program is represented as an expression graph whose nodes are incrementally overwritten with their ultimate value, eventually replacing the root of the graph with the result of the overall computation. A common need among such systems is the ability to search through the computation graph, visiting each node exactly once to perform a particular operation. We refer to this process as **marking**, recognizing that the operation

performed on a particular node may be quite simple, such as with garbage collection, or arbitrarily complex.

The homogeneous nature of graph reduction makes it an attractive avenue for highly parallel computation, since the graph may be partitioned and distributed among an arbitrary number of processing elements, each performing part of the overall reduction [14, 16, 22, 30]. The highly parallel computation that results introduces additional problems that make the ability to effectively mark the computation graph even more useful, especially if it can be accomplished *in concert with* the main computation (i.e., in such a way as to allow the transmutation process to continue while the marking takes place). For example, eagerly invoked computations that subsequently become irrelevant may require termination ("irrelevant task deletion" [2, 12, 15]), deadlocked portions of the computation graph may exist ("run-time deadlock detection" [6, 15]), and tasks may require migration or prioritization ("load-balancing" [16]).

Despite the usefulness of a distributed marking process, no such algorithms have appeared in the literature -- in fact, the apparent intractability of the problem has prompted a remarkable number of alternative solutions. Most of these alternatives have been motivated from a need for garbage collection, and have included *reference counting* techniques, as well as strategies that subdivide the graph space into small regions within which separate garbage collections take place. There are various reasons why each of these alternatives is inferior in some way to a system-wide, effectively distributed scheme, and this has motivated our current work. A distributed scheme is also quite interesting theoretically, in that it provides an intriguing view of the complex interactions that exist between parallel processes.

We present in this paper three new algorithms that effectively distribute the graph marking process. The first is used primarily to introduce the concept of a **marking-tree**, a key mechanism used by our methods. The second algorithm could be classified as a distributed version of the mark phase of a conventional *mark-sweep garbage collector*, and the third could be viewed as a distributed version of a *copying collector*. The schemes exhibit the following behavior:

- They accomplish a single, system-wide marking of the computation graph from a given root.
- They are effectively distributed, having no centralized data or control.

- They allow a certain class of graph mutations to be executed concurrently with them.
- Although mutually exclusive access to nodes in the graph is required, the mechanisms to accomplish this are simple and do not result in excessive contention.

Related Work

Sequential graph marking techniques are imbedded within every garbage collection system of the "tracing" variety. We assume the reader to be familiar with conventional techniques such as described in [17]. Starting with Steele's paper in 1975, a flurry of articles appeared proposing several versions of "parallel", "real-time", or "on-the-fly" garbage collectors [3, 8, 13, 18, 19, 21, 25, 29, 32]. The motivation behind these schemes is to conduct garbage collection simultaneously with the main computation, so as to avoid the annoying pause in execution characterized by conventional collectors. This is accomplished by having the collector either execute in *parallel* with the mutator, or by *interleaving* its operations with those of the mutator. Steele also briefly discusses an extension of his work to a multiprocessing environment (with shared memory). Dijkstra et al. [8] provide a correctness proof for their algorithm, as have others [10, 11, 18, 19].

Other than Steele's work, little has been done to implement garbage collection in multiprocessor or distributed processing systems. Lamport discusses a version of Dijkstra's algorithm in which synchronization is forced between multiprocessors using a shared memory [20]. Almes introduces a multiprocessor scheme [1], but again relies on a centralized data structure to coordinate the collection. Bishop proposes dividing the memory into regions and performing collections separately within each region -- reference counting is then used *between* regions [4]. Reference counting has also been proposed in [16] and [26]. All of these schemes either rely on a centralized structure to coordinate marking (thus introducing a bottleneck), or use some form of reference counting (precluding the collection of circular structures). We demonstrated the first effectively distributed algorithm for garbage collection in [15] -- the distributed marking strategy described there is further discussed and extended in this paper, where we present two alternative algorithms as well as additional insight into the nature of the problem.

In a non-garbage-collection setting, there have appeared several distributed graph algorithms using a model in which a network of processors is isomorphic to a given graph (one node per processor, one arc per communication channel). Chang [5] describes various algorithms using this

model, and demonstrates the usefulness of graph algorithms for deadlock detection [6]. Dijkstra and Scholten [9] describe a technique for detecting the termination of "diffusing computations," and Misra and Chandy [23, 24] use Dijkstra's technique to describe distributed algorithms for finding knots and shortest paths in a graph. We have generalized this work by decoupling the processor network from the graph structure, using a model in which the graph is arbitrarily partitioned and distributed among any number of processors. Furthermore, we are interested in dynamic graph structures whose connectivity constantly changes during execution of the marking algorithms that we propose.

The remainder of this paper is organized as follows. The next section establishes a model of distributed computation and other concepts needed for our discussion. Section 3 presents the concept of parallel graph mutations. In Sections 4, 5, and 6 we present our three marking algorithms in turn, establishing the concept of a marking-tree and its adaptability to various environments. The algorithms are first presented alone, then discussed in relation to parallel mutations. In Section 7 we discuss the general nature of parallel graph mutations, and reveal a class of mutations for which no distributed cooperation can be found. Section 8 concludes with a comparison of the schemes and a discussion of problems and related issues.

2. Basic Concepts and Terminology

We wish to express our algorithms in as general a setting as possible. The model that we use represents a class of distributed processing systems in which:

1. There are an arbitrary number **processing elements** or **PE's**.
2. Each PE has only **local store** which is used to hold a portion of the computation graph. (Collectively the local stores may be viewed as a single heap upon which the computation graph is distributed.)
3. There is a **global addressing** scheme whereby one node in the graph may reference any other in the system.
4. The PE's are interconnected by an **arbitrary communication network**.
5. **Communication** between PE's occurs by **spawning tasks** from one processor to another. A task is the smallest specifiable unit of processor activity.

We do not wish to restrict any more than necessary the internal architecture of a PE, in particular the task scheduling mechanism. For the moment we assume only that tasks spawned for execution on a particular PE reside in a "task pool", and unless otherwise stated we assume

unlimited concurrent execution of any tasks within this pool. The correctness proof that we provide for Scheme A, in fact, places no restriction on the order of task execution; highly-parallel execution within the PE's themselves is not precluded.

The computation graph (or just "graph") is the graph to be marked, and a node is an element of that graph. We will use variables as *names* for nodes, but for convenience we often write "node *n*" to indicate "the node referenced by *n*"; context should make the meaning clear.

Tasks are specified in terms of primitive constructs for manipulating the computation graph and spawning or executing other tasks -- these constructs are described below using an Algol-like syntax. The specification of a task looks like that for a procedure, except that the keyword **task-procedure** is used. The lexically first argument in a task's parameter-list shall be used to determine the PE upon which the task is intended to execute. An instance of a task may either be **spawned** or **executed**. The statement **spawn *f(x,y)*** causes an instance of the task *f* to be added to the task pool of the PE containing node *x*. No waiting is done for the completion of the task; rather, execution continues immediately with the next statement. On the other hand, the statement **execute *f(x,y)*** creates an instance of the task *f* for immediate execution. Execution continues with the statement following the **execute** statement only after the task has finished executing.

For reasons that will become apparent later, it is necessary for tasks to be able to gain exclusive access to a node. For this purpose we adopt a simple locking protocol in which only one task may lock a node at a time; the effect of a task *t* executing the statement **while-locking *n* do *S*** is:

```

if node n is unlocked
then lock n, execute S, and unlock n
else unlock all nodes locked by t
and re-spawn t.

```

With this interpretation deadlock is not possible, since when the construct is nested, the entire task is re-spawned and all outer nodes unlocked if an inner node is found locked. In the special case where task execution is nested (i.e., where one task **executes** another), all nodes are unlocked as above, but only the outer-most task is re-spawned.

At times it will be necessary to have more control over locking than that provided by

while-locking. For this purpose the statement **lock n** attempts to lock node **n**, returning true if successful, false otherwise. Similarly, **unlock n** is the converse operation, and always succeeds. Note that if **while-locking n** fails to lock **n**, the task executing it is re-spawned, whereas **lock n** would simply return false. If the current task is **t** then a *non-nested* version of **while-locking n** do **S** is:

```

    if lock n
    then << S; unlock n >>
    else spawn t

```

where **<<...>>** is an abbreviation for **begin...end**, and which we will often use for clarity and conciseness.

Despite the need for these locking mechanisms, we will make a concerted effort to minimize their use. For example, one of our implicit goals is that any task that we create for marking has the property that no node (or other structure) is left locked while marking is continued on some other PE. This is consistent with our desire for an effectively distributed algorithm.

We assume that the computation graph has a unique node called the **root** from which all active nodes may be reached. Let **proc(n)** denote the PE containing node **n**. The following functions are used to manipulate the graph:

- **children(n)** returns the set of nodes that are descendants of **n** in the computation graph.
- **connect(a,b)** adds a reference to **b** in **children(a)**. Similarly, **disconnect(a,b)** removes a reference to **b** from **children(a)**.
- **replace-child(n,old,new)** has the effect of removing **old** from **children(n)** (if it is there) and replacing it with **new**.
- **select-child(n)** returns an arbitrary element from **children(n)**, and nil if **children(n) = ∅**.
- Given a node **n** in the computation graph, and an arbitrary subgraph **g** that is *not* part of the graph, **splice-in(n,g)** has the effect of splicing in **g** below node **n**. That is, the elements of **children(n)** become references to nodes in **g**, and some nodes in **g** are made to point to elements of the original **children(n)**.
- **unmarked(n)**, **transient(n)**, and **marked(n)** return true if node **n** is unmarked, transient, or marked, respectively, and false otherwise (the meaning of these terms will be explained later). Similarly, **clear(n)**, **touch(n)**, and **mark(n)** make node **n** unmarked, transient, and marked, respectively.

We say that a node **n** is *reachable* if there exists a sequence of nodes **<root, n₁, n₂, ..., n_x, n>** such that **n₁ ∈ children(root)**, ... **n_i ∈ children(n_{i-1})**, ... , **n ∈ children(n_x)**.

3. Parallel Graph Mutations

A mutator is any process that changes the connectivity of the computation graph. In our model the transmutation of the graph is distributed, and each PE may be viewed as an autonomous mutator.

Given any marking process, an important question to ask is what type of graph mutations, if any, may execute concurrently with it? Dijkstra and others have asked this question in the context of a *sequential* garbage collector executing in parallel with a *sequential* mutator. Their work has shown that the mutations must be executed *cooperatively* with the collector, so as not to invalidate the marking process. Our model is one of *highly-parallel* marking executing in parallel with *highly-parallel* mutations, so it is not surprising that we have found similar results.

The approach we have taken is to determine a small set of primitive graph mutations that can simulate a given class of computations on a graph. We consider first the **functional mutator primitives** shown graphically in Figure 3-1, which are collectively sufficient to perform graph reductions for the evaluation of any functional program. This rather strong statement can be supported by noting that any functional program may be translated into a **combinator expression** (as described in [31]), and that the graph reduction associated with each of the common combinators (as well as the reductions associated with standard primitive functions such as arithmetic operators) can be simulated with our set of primitive mutations. We demonstrate this for the combinators **S**, **K**, and **Y** in the Appendix.

There are three mutations in the set: **delete-reference** removes an arc from the graph, **add-reference** adds an arc between the first and third node in a sequence of three nodes, and **expand-node** adds new nodes to the graph by splicing in an arbitrary subgraph that is not already part of the graph. For simplicity we assume that the nodes **a**, **b**, and **c** in Figure 3-1b are distinct, recognizing that **a=b**, **b=c**, or **a=c** are special cases that are easily handled.¹ All of the mutations are viewed as *indivisible operations*, requiring the use of our locking constructs in the formal specifications of their behavior to be presented later.

In the following three sections we introduce the three marking algorithms in turn. In each

¹For example, we should allow a node to add a reference to itself.

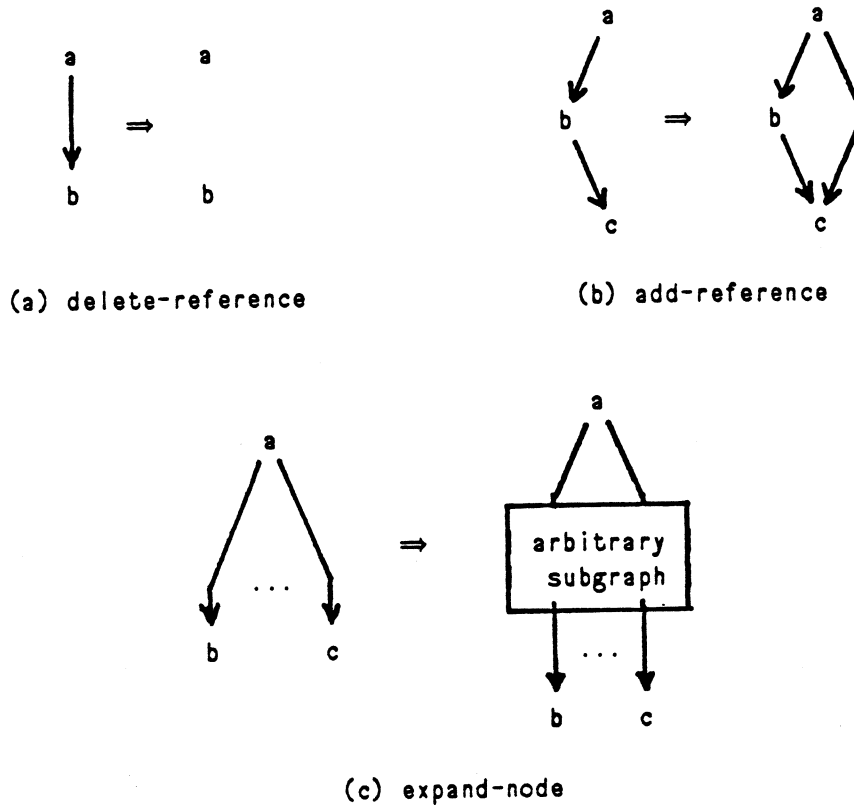


Figure 3-1: Functional Mutator Primitives

section we first concentrate on marking as a single distributed process, and then discuss the feasibility of parallel execution with the set of functional mutator primitives just described.

4. The Marking-Tree

By way of introduction, consider for a moment a stack implementation of a sequential marking process: The root is marked and pushed onto the stack, then nodes are repetitively removed from the stack for the purpose of propagating marking to each of their children in the graph. The stack is used primarily as a *place-holder* for nodes that have been marked but whose descendants have not yet been examined, and the eventual *emptiness* of the stack indicates that marking is complete. In most of the "on-the-fly" garbage collectors, the stack (or other linear structure) serves another role, that of a *shared data structure* to allow the mutator to cooperate with the marking process so as to not interfere with it.

A linear structure such as a stack could naively be used in a distributed system either as a

centralized structure or somehow distributed. In either case problems arise, since a centralized stack (such as used in [1]) causes an intolerable communications bottleneck, and it is difficult to ascertain the emptiness of a distributed stack, since a single PE's portion of the stack may become empty at one point, only to have more nodes added subsequently.

The key idea behind the algorithms to be presented is the use of a *tree* structure to coordinate the marking process. This tree is referred to as the **marking-tree**, and may be viewed roughly as a *spanning tree* of the reachable nodes of the computation graph. Just as a linear stack reflects the nature of sequential marking, the marking-tree reflects the parallel nature of distributed marking. Just as a sequential mutator may cooperate by adding nodes to a stack (the stack's emptiness meaning sequential marking is complete), a distributed mutator may add branches to the marking-tree (the tree's total collapse indicating distributed marking is complete).

4.1. The First Algorithm (Scheme A)

The first algorithm that we present (Scheme A), relies exclusively on the marking-tree to coordinate marking. There are only two types of tasks involved in the entire scheme. A **mark-A** task is used to initiate marking on the root, which has the effect of propagating marking "forward" through the graph by spawning more such mark tasks on descendant nodes. For every **mark-A** task spawned there is eventually one **return-A** task spawned that propagates "backward" in the graph; one of these eventually reaches the root indicating that marking is complete. (The suffix **-A** is to distinguish these tasks from similar ones used in Schemes B and C, which are explained later.)

To best understand Scheme A it is helpful to be able to refer to the "mark state" of a node; that is, its relationship to **mark-A** and **return-A** tasks:

1. An **unmarked** node is one to which marking has not yet propagated (although a **mark-A** task may be pending with it as target). Initially, all nodes are unmarked.
2. A **transient** node is one to which marking has propagated, and from which a **mark-A** task has been spawned on each of its children.
3. A **marked** node is one of two types:
 - a. A previously transient node with the added characteristic that all of the **mark-A** tasks spawned from it have "returned" (via a **return-A** task), and that it has recursively spawned a **return-A** task on its parent in the marking-tree.

- b. A node added to the graph by the **expand-node** primitive, in a way to be described shortly.

Eventually all reachable nodes become marked.

Dijkstra [8] refers to a node's mark state by attributing a "color" to each node, where white, gray, and black nodes correspond roughly to our unmarked, transient, and marked nodes, respectively. Our meanings are subtly different, however, due to the distributed system context; indeed, the meanings will be different for each of the marking algorithms that we propose.

A **return-A** task may be spawned as soon as all marking has completed "below" a node. To facilitate determining when and where to spawn the **return-A** task, each node is augmented with two fields, one containing a pointer to its parent in the marking-tree (**mt-par**), and one to maintain a *count* of the number of active **mark-A** tasks that have been spawned on its children (**mt-cnt**). Intuitively, once a node **n** has been marked, then when **mt-cnt(n)** reaches zero, it is safe to spawn a **return-A** task on **mt-par(n)**.

The formal specifications for **mark-A** and **return-A** are shown in Figure 4-1. We assume that each PE continually executes tasks from its task pool in arbitrary order and with unlimited concurrency (including, of course, the special case of completely sequential execution). Initially all nodes are unmarked and their **mt-cnt** is zero. Marking is started by spawning the task **mark-A(root,rootpar)**, where **rootpar** is a dummy node that **return-A** uses to detect termination; that is, eventually the task **return-A(rootpar)** will be spawned, which will set the global flag **done** to true, indicating that marking is complete.

4.2. Parallel Mutations

Consider now the functional mutator primitives of Figure 3-1 as an arbitrary number of them execute concurrently with Scheme A. To see that cooperation is needed at all, consider the following dilemma: Suppose we have a graph $a \rightarrow b \rightarrow c$, and the marking process has just spawned a **mark-A** task from **a** to **b**. Next a series of mutations occur, connecting **a** to **c** and disconnecting **c** from **b** (leaving $b \leftarrow a \rightarrow c$). At this point **c** is only accessible from **a**, but since marking has already propagated beyond **a**, **c** will never get marked. The mutator needs to cooperate with the marking process in such situations, but it is not sufficient for the mutator to simply *mark c* at the time it adds the reference from **a** to **c**, since there may be an arbitrary

```

task-procedure mark-A(n,par);
  while-locking n
  do if unmarked(n)
    then begin touch(n);
          mt-par(n) := par;
          for each x ∈ children(n)
            do << spawn mark-A(x,n);
              increment(mt-cnt(n)) >>;
          if mt-cnt(n) = 0
            then << mark(n);
              spawn return-A(par) >>
          end
        else spawn return-A(par);

task-procedure return-A(n);
  while-locking n
  do if n = rootpar then done := true
    else begin decrement(mt-cnt(n));
          if mt-cnt(n) = 0
            then << mark(n);
              spawn return-A(mt-par(n)) >>
          end;

```

Figure 4-1: Scheme A: relying exclusively on the marking-tree

number of nodes accessible from c that need to be marked as well. There must be some way to *splice in* extra marking activity; it will be shown that the marking-tree provides a convenient way to do this.

Interference by the mutator can be stated in terms of violations of the marking state of nodes. By studying the definitions of marking states given earlier, we see that there are several properties that must be maintained, but there are two in particular that the mutator primitives of Figure 3-1 could violate. Specifically:

1. For each transient node, there must be at least one **mark-A** task spawned on each of its children (and the **mt-cnt** must reflect this).
2. A marked node may never point to an unmarked node.

The first invariant preserves the meaning of a transient node, ensuring the integrity of a **mark-A** task. Similarly, the second invariant preserves the meaning of a marked node; that is, if all of the **mark-A** tasks spawned from a node have “returned”, then none of that node’s children could be unmarked; they must be “at least transient.”

		mark state of node a		
		unmarked	transient	marked
mark state of node b	unmarked	no	coop	impos- sible
		coop	w / a	
	transient	no	no	coop
		coop	coop	w / b
	marked	no	no	no
		coop	coop	coop

Figure 4-2: Cooperation by the add-reference primitive (Scheme A)

Now consider the mutator primitives in turn. The **delete-reference** primitive cannot violate either of the invariants, so no cooperation is needed. However, the **add-reference** primitive can violate both of them. To see this, consider all combinations of mark states of nodes **a** and **b**, just as the reference is being added from **a** to **c**; the nine combinations are shown in Figure 4-2. If node **a** is unmarked, no violations are possible (and thus no cooperation is needed) since marking has not propagated to **a** yet. If node **b** is marked, still no cooperation is needed, since the second invariant tells us that node **c** is either transient or marked. This same invariant discounts the situation where **a** is marked and **b** is unmarked, so that leaves three other combinations. If **a** and **b** are both transient, again there is no problem, since according to the first invariant, there is at least one mark task spawned on **c**. Finally:

1. If **a** is transient and **b** is unmarked, then the first invariant may be violated. This can be prevented by *spawning* the task `mark-A(c,a)` (and incrementing `mt-cnt(a)`).
2. If **a** is marked and **b** is transient, then the second invariant may be violated. This can be prevented by *executing* the task `mark-A(c,b)` (and incrementing `mt-cnt(b)`).

Note that in the second case it is necessary to *execute* the mark task so that **c** indeed becomes transient or marked. The first case only requires *spawning* the task to maintain the invariant.

The resulting “cooperating mutator primitives” may be expressed textually as shown in Figure 4-3 by using the **while-locking** construct to effect indivisibility. The function `select-child` is

used to enforce the constraints on the graph needed by **delete-reference** and **add-reference**.²

Note the cooperative action taken by the **add-reference** task, as outlined above. Also, recall from the definition of **while-locking** that if **c** is already locked when **execute mark-A(c,b)** is reached, then the outer-most task is re-spawned, which in this case is the original **add-reference** task.

```

task-procedure delete-reference-A(a);
  while-locking a do
    << b := select-child(a);
      disconnect(a,b) >>;

task-procedure add-reference-A(a);
  while-locking a do
    << b := select-child(a);
      while-locking b do
        begin c := select-child(b);
          if transient(a) and unmarked(b)
            then << spawn mark-A(c,a);
              increment(mt-cnt(a)) >>
          else if marked(a) and transient(b)
            then << execute mark-A(c,b);
              increment(mt-cnt(b)) >>;
          connect(a,c)
        end >>;

task-procedure expand-node-A(a,g);
  while-locking a do
    begin if marked(a) then mark(g)
      else clear(g);
    splice-in-subgraph(a,g);
    if transient(a)
    then for each x ∈ children(a)
      do << spawn mark-A(x,a);
        increment(mt-cnt(a)) >>;
    end;

```

Figure 4-3: Cooperating Mutator Primitives for Scheme A

A similar solution takes care of the cooperation needed by the **expand-node** primitive, and is

²Normally the semantics of a particular graph-reduction strategy would select the nodes involved.

also shown in Figure 4-3. Here the functions **mark** and **clear** have been extended over subgraphs; **g** is first marked or cleared depending on the mark state of **a**. This implies that just after **g** is spliced into the computation graph, the second invariant is still valid. However, if **a** is transient the first invariant might have been violated, but is easily corrected by spawning mark tasks on **a**'s children, as shown. Finally, we note that the subgraph **g** may itself be distributed; it does not have to reside solely on **proc(a)**.

4.3. Remarks

The specifications for the mutator primitives as presented do not consider the fact that the nodes involved in a mutation may reside on different PE's. This is easily handled by extending the behavior of the **while-locking** construct so as to work properly across processor boundaries. A more detailed specification would involve re-specifying the primitives using the **lock** and **unlock** constructs, and subdividing the tasks into a collection of **sub-tasks**, each performing a portion of the mutation on one PE. The specifications as given here are still logically correct, and since the details of an alternative specification are not central to the main thesis, they are not included (the interested reader may find them in [14]).

Also, the highly-parallel nature of our model of computation requires that locking be used to prevent the mutator primitives from interfering with each other (i.e., to maintain their indivisibility). We suspect that these same locking mechanisms are crucial to effecting the cooperation necessary to accomplish parallel marking. Previous work with "on-the-fly" garbage collectors avoided locking by assuming the mutations to be indivisible operations of a sequential computer, something that we are unable to do in our highly-parallel distributed model.

Finally, although the ideas were developed independently, the marking-tree could be viewed as a mechanism for detecting the termination of a *diffusing computation* as identified by Dijkstra [9], the diffusing computation being the marking process itself.

4.4. Performance

Scheme A is useful in demonstrating how the marking-tree can be used to coordinate the overall marking process. The scheme is fairly straightforward, and relatively efficient, in that each arc in the computation graph is only traversed twice, once by a **mark-A** task and once by a

return-A task. In terms of inter-processor communication, it is obviously superior to the use of a centralized stack. Since locking is used, there exists the potential problem of excessive contention with a parallel mutator, but Scheme A does quite well in this regard in that the mark and return tasks never nest the locking of nodes, and never leave a node locked while marking is continued elsewhere.

There is, however, a significant space overhead: the space for the marking-tree itself (embedded within the computation graph), and space for the task pool.³ Consider first the extra space required of each node in the computation graph. The mark state of a node can be determined from the status of **mt-cnt** and **mt-par**, and a single bit is required to realize locking. In most systems the number of children of a node is bounded, and mutations can only add a fixed number of branches from a node in the marking-tree -- two bits for **mt-cnt** is typically sufficient. On the other hand, a full pointer is generally required for **mt-par**. Consequently the extra space for the marking-tree required of each node is on the order of one full word plus about three bits.

A less tangible form of overhead is the room required for the task pool⁴ (we may assume for analysis purposes that the mutator tasks are managed in a separate pool). Each entry requires one bit to distinguish between the two task types, plus room for the arguments (a **mark-A** task requires two arguments, a **return-A** task one). The *total* number of tasks spawned is twice the number of arcs traversed in the graph. The required size of the pool, however, depends on the maximum number of tasks that could exist *simultaneously*, which unfortunately depends on statistical properties of the computation graph. The worst case is quite bad, and occurs in the following situation. Suppose the maximum number of children of a node were n . Construct an n -ary tree of depth k , and then modify each leaf so that it has n pointers to the root. The task pool of the PE containing the root could then have n^{k+1} entries.

A refinement of Scheme A that improves the worst-case is to modify the spawning mechanism so that before a PE adds a task **mark-A(x,y)** to its pool, it first checks to see if a task of the

³These overheads might still be tolerable for some applications.

⁴Which presumably would be implemented as a queue, stack, linked list, or some other appropriate structure.

form $\text{mark-A}(x,z)$ is already there.⁵ If so, it would ignore the new task and spawn $\text{return}(y)$. The worst-case number of entries in the task pool is then the number of nodes in the computation graph resident on that PE; this situation is analogous to the number of pushes required of a stack in a conventional garbage collector.

4.5. Correctness Proof

In [14] we give a detailed correctness proof for Scheme A operating in parallel with a "random mutator", in which an arbitrary number of functional mutator primitives are continuously spawned to random spots in the computation graph. The proof is accomplished independently of a particular task scheduling strategy. It is shown that marking will not only occur correctly, but that it is guaranteed to terminate given a fair scheduling policy in a finite system. We use a form of Owicki's axiomatic proof technique for parallel programs [27] that we have extended to include the while-locking construct and task spawning mechanisms, in conjunction with simple axioms of temporal logic [28]. This combination of proof techniques allows us to prove the invariance of certain properties concerning the mark states of nodes, in addition to certain liveness properties dealing with the global behavior of the marking process. The proof is long and tedious, and is therefore presented in summary form below.

We use $\text{spawned}(t)$ to assert that task t has been spawned, and $\text{completed}(t)$ to assert that task t has successfully locked all desired nodes and has completed execution. Tasks are referred to by specifying a form and quantifying over some set of nodes (as in " $(\forall x) \text{completed}(\text{return-A}(x))$ "). The form " $\text{return-A}(\text{mt-par}(x))$ " refers to the *unique* return-A task that is eventually spawned as a result of the mark-A task that initially touched x (assuming that x was initially unmarked in the computation graph).

$\square P$ (read "always P ") means that P is always true, and $\diamond P$ (read "eventually P ") means that P is true now or will become true at some time in the future. The assertion $P \mapsto Q$ is read " P leads to Q " and is equivalent to $\square(P \Rightarrow \diamond Q)$. The formal specification of the fair scheduling axiom is:

⁵Implementing the pool as a priority queue whose elements are ordered by the first argument of the mark task is one possible implementation for this.

fair scheduling: $(\forall t) \text{spawned}(t) \mapsto \text{completed}(t)$.

Definitions:

rootpar	a special node such that $\text{children}(\text{rootpar}) = \{\text{root}\}$ and $\neg \exists x \mid \text{rootpar} \in \text{children}(x)$.
CG	the finite set of nodes reachable from the root (the <u>computation graph</u>).
FL	the finite set of free nodes from which expand-node-A acquires nodes (the <u>free list</u>).
GAR	the finite set of nodes not in CG or FL at the moment marking begins (<u>garbage</u>).
MT_i, i > 0	the set of nodes x such that there exists a sequence of nodes $\langle \text{rootpar}, x_1, \dots, x_{i-1}, x \rangle$ such that $\text{rootpar} = \text{mt-par}(x_1)$, $x_k = \text{mt-par}(x_{k+1})$, $x_{i-1} = \text{mt-par}(x)$ (or $\text{rootpar} = \text{mt-par}(x)$ if $i=1$).
MT	$\text{MT}_1 \cup \text{MT}_2 \cup \dots \cup \text{MT}_\infty$ (the <u>marking tree</u>).

Note that **CG** and **FL** are dynamic structures (they change as mutations are performed to the computation graph), whereas **GAR** is fixed.

The initial conditions are:

- $(\forall n) \text{mt-cnt}(n) = 0 \wedge \text{unmarked}(n)$
- $\text{spawned}(\text{mark-A}(\text{root}, \text{rootpar}))$
- there exists a "generator" that continuously spawns new **delete-reference-A**, **add-reference-A**, and **expand-node-A** tasks on random nodes in **CG**.

Note that once the generator spawns a task on a node n , it is possible that n will become unreachable from the root; that is, it will no longer be a member of **CG**. Such tasks are called **irrelevant**, and we wish to show correct behavior of the marking process even in their presence.

The detailed proof involves establishing a set of *program invariants* that are always true, and a set of *resource invariants* that establish properties of a node that must be true whenever the node is unlocked. The proofs are conducted for each of the tasks in isolation, and are then shown to be *interference-free*. We finally prove two theorems:

Theorem 1: Liveness

◇ done. That is, marking eventually terminates.

Proof:

(1) By the initial conditions and fair scheduling we know $\diamond \text{completed}(\text{mark-A}(\text{root}, \text{rootpar}))$.

(2) Now consider a task $\text{mark-A}(x, y)$ at the moment node x is locked: (a) If x is already marked or transient, a $\text{return-A}(y)$ task is immediately spawned. Otherwise, (b) new mark-A tasks are spawned on each child of x , and the number of such tasks that have not "returned" is retained in $\text{mt-cnt}(a)$. (It is necessary to prove the invariance of this over execution of all mutator tasks.)

(3) The "call-graph" of the tasks in (2b) must form a finite tree whose nodes comprise precisely the marking-tree. The leaves of this tree either have no children or all the mark-A tasks spawned from it are of form (2a) (i.e., they encountered already-marked nodes). By fair scheduling then, each leaf's mt-cnt eventually drops to zero, spawning a return-A task on its parent.

(4) From (3) a recursive argument follows, showing that the root must also eventually spawn a return-A task on its parent; that is, $\diamond \text{spawned}(\text{return-A}(\text{rootpar}))$ from which we derive $\diamond \text{done}$.

□

Theorem 2: Safety

$\text{done} \Rightarrow ((\forall x \in \text{CG}) \text{marked}(x)) \text{ and } (\forall x \in \text{GAR}) \text{unmarked}(x)$.

That is, once done is true, all nodes in the computation graph must be marked, and all nodes unreachable from the root prior to marking are still unmarked.

Proof:

(1) The only task that could set done to true is $\text{return-A}(\text{rootpar})$, and since root is the only child of rootpar , then this task must be $\text{return-A}(\text{mt-par}(\text{root}))$, meaning that the root is marked. Thus $(\forall x \in \text{MT}_0) \text{completed}(\text{return-A}(\text{mt-par}(x))) \wedge \text{marked}(x)$ since $\text{MT}_0 = \{\text{root}\}$.

(2) If for each node x in MT_i , $\text{return-A}(\text{mt-par}(x))$ has been spawned, then clearly for each node y in MT_{i+1} , y must be marked and $\text{return-A}(\text{mt-par}(y))$ must have been spawned and completed.

(3) Using (1) as a basis and (2) as an induction rule, it then follows that $(\forall x \in \text{MT}) \text{marked}(x) \wedge \text{completed}(\text{return-A}(\text{mt-par}(x)))$.

(4) Now consider $n \in \text{children}(\text{root})$. If $n \in \text{MT}$, then by (3) it must be marked. If it is not in MT , then it cannot be unmarked (since a marked node can never point to an unmarked node) nor can it be transient (since that would imply it is in MT). Thus it must be marked. An inductive argument follows, proving that all elements of CG are marked.

(5) To prove that all nodes in GAR are white, we simply note that there is no way for any of the tasks to even access such nodes, so their mark state could not change. Since they are initially unmarked, they must still be unmarked.

□

5. Using a Stack to Coordinate Local Marking

Our second marking technique is a refinement of Scheme A based on the following observation: The primary purpose of the marking-tree is to coordinate the overall marking process -- therefore why not use it only as an inter-PE data structure, and use a *stack* to coordinate local marking internal to each PE? Indeed this idea works, as explained below.

5.1. The Second Algorithm (Scheme B)

We again wish to present the most general solution, and therefore do not restrict a PE's task scheduling mechanism. Our only assumption is that each PE asynchronously executes a local stack-based marking process *local-mark* in parallel with all other task activity. Within a given PE, the mutator tasks as well as the tasks responsible for marking cooperate with *local-mark* through four shared variables:

- A *stack* on which *push*, *pop*, *top*, and *stack-empty* operate in the obvious way.
- *local-root*, a pointer to a local node that serves as the only local node in the global marking-tree. Its value is *nil* only when the stack is empty and all *mark-B* tasks spawned from the PE have "returned".
- *mt-count*, the number of mark tasks spawned from the PE.
- *mt-parent*, a pointer to the parent of *local-root* in the marking-tree.

Since these data structures are shared (locally), we define the functions *push*, *pop*, *increment*, and *decrement* to be indivisible operations. Furthermore, we introduce a binary semaphore *mutex* on each PE for which the primitive functions *P* and *V* operate in the conventional way. Let "[S]" be shorthand for " $\ll P(\text{mutex}); S; V(\text{mutex}) \gg$ ".

The new mark and return tasks (called *mark-B* and *return-B*, respectively) as well as the procedure *local-mark* are shown in Figure 5-1. Marking is initiated by spawning the task *mark-B*(*root*,*rootpar*). A task *mark-B*(*n*,*par*) behaves similar to a *mark-A* task, except that once having touched node *n*, it is pushed onto the local stack for further marking. Also, if there is already a *local-root* node on the current PE, *return-B*(*par*) is immediately spawned. On the other hand, if *local-root* is *nil*, then it takes on the value of *n*. Note the mutual exclusion constraints placed on the manipulations of *local-root*; this is to prevent interference by some other *mark-B* task. *return-B* simply decrements *mt-count*, or sets *done* if marking is complete.

```

task-procedure mark-B(n,par);
  while-locking n
  do if unmarked(n)
    then begin touch(n);
             push(n);
             [ if local-root = nil
               then << mt-parent := par;
                    local-root := n >>
               else spawn return-B(par) ]
            end
    else spawn return-B(par);

task-procedure return-B(n);
  if n = rootpar then done := true
  else decrement(mt-count);

procedure local-mark();
while true do
begin while not stack-empty() and lock top()
  do << n := pop();
     for each x ∈ children(n)
     do if local(x)
        then if lock x
             then if unmarked(x)
                  then << touch(x); push(x) >>
                  else no-op()
             else << spawn mark-B(x,n);
                    increment(mt-count) >>
        else << spawn mark-B(x,n);
              increment(mt-count) >>;
     mark(n);
     unlock n >>;
  [ if stack-empty() and mt-count = 0 and local-root ≠ nil
    then << spawn return-B(mt-parent);
          local-root := nil >> ]
end;

```

Figure 5-1: Scheme B: Local marking with a stack

The procedure `local-mark` continually tries to propagate marking from nodes removed from the stack. The top node is first locked (if possible), and its children are examined in turn. If a child is non-local or cannot be locked, a `mark-B` task is spawned on it. Otherwise, if it is successfully locked and is unmarked, then it is touched and pushed onto the stack. After each attempt to trace nodes removed from the stack, `local-mark` checks to see if local marking is complete (indicated by an empty stack and `mt-count = 0`). If so a `return-B` task is spawned on `mt-parent` if it has not been done already (`local-root` is set to `nil` to indicate that this has been done).

5.2. Parallel Mutations

As with Scheme A, we assume unlimited concurrent execution of the functional mutator primitives. Correct marking behavior can again be achieved if proper cooperation is observed, although for the moment we restrict the `expand-node` primitive so that the new subgraph resides on the same PE as the node below which it is being spliced. We shall return to the more general case in Section 5.3.

First define an unmarked node as before, a transient node as one that resides on a local stack, and a marked node as one whose children are either marked or have a mark task pending on each of them. The latter definition subsumes previously transient nodes as well as new nodes added by `expand-node`. We assume that a proper set of invariants to be maintained by the mutators reflects the preservation of a node's mark state, and thus exists implicitly given the above definitions.

Figure 5-2 shows the nine combinations of mark states for nodes `a` and `b` for the `add-reference` primitive as it would execute concurrently with Scheme B; note that *all* combinations are possible. In the case where `a` is marked and `b` is unmarked, marking must still be active on the processor containing node `a`, since `b` being unmarked implies that at least one of the `mark-B` tasks spawned from `a` has not yet executed, so the `mt-count` of `proc(a)` is greater than zero. This allows cooperation with `a` even though it is marked, but if we are to rely on this property then its validity must be maintained. Hence if `a` is marked and `b` is transient, we must *execute* the task `mark-B(c,b)` to avoid the possibility of a marked-unmarked arc for which marking might not be active on the PE containing the marked node.

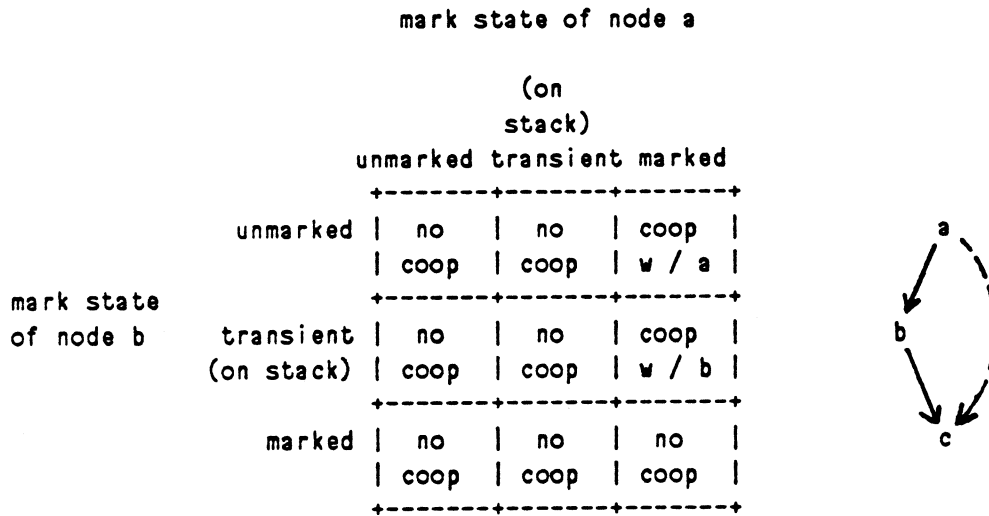


Figure 5-2: Cooperation by the add-reference primitive (Scheme B)

Figure 5-3 shows the resulting cooperating mutator primitives for Scheme B, where `mt-count(proc(n))` denotes the `mt-count` on the PE containing node `n`. `delete-reference` again requires no cooperation, and the cooperation required of `expand-node` is now trivial.

5.3. Remarks

In the quest for the most general solution we have not limited the degree of concurrency within a PE. This has resulted in the need for mutual exclusion constraints on the shared variables that underpin the local marking process. A more conventional (and perhaps more realistic) implementation might incorporate sequential processors for each PE, or even a "dual processor" arrangement in which one is devoted to marking, the other to the main computation. In either case, the mutual exclusion constraints would be greatly simplified.

Recall that the `expand-node` primitive was restricted as to where the new subgraph was allocated. If we were to remove this restriction while still using the specification of `expand-node` shown in Figure 5-3, then a situation could arise in which a node `n` in the new subgraph that is allocated on a different PE could point to an unmarked node, yet `proc(n)` has

```

task-procedure delete-reference-B(a);
  while-locking a do
    << b:= select-child(a);
      disconnect(a,b) >>;

task-procedure add-reference-B(a);
  while-locking a do
    << b := select-child(a);
      while-locking b do
        begin c := select-child(b);
          if marked(a) and unmarked(b)
            then << spawn mark-B(c,a);
              increment(mt-count(proc(a))) >>
          else if marked(a) and transient(b)
            then << execute mark-B(c,b);
              increment(mt-count(proc(b))) >>;
          connect(a,c)
        end >>;

task-procedure expand-node-B(a,g);
  while-locking a do
    << if marked(a) then mark(g)
      else clear(g);
      splice-in-subgraph(a,g) >>;

```

Figure 5-3: Cooperating Mutator Primitives for Scheme B

no local marking activity.⁶ This would invalidate the property that **add-reference** depends on for proper cooperation. There is an alternative specification of **expand-node** that avoids this problem, based on the following strategy: Consider **expand-node(a,g)**. If **a** is unmarked, clear **g** and proceed with the mutation. If it is marked but there is no marking activity on **proc(a)**, then mark **g** and proceed, since none of **a**'s children could be unmarked. Otherwise, clear **g**, perform the mutation, and then spawn mark tasks on all of **a**'s children using **proc(a)** as a base. The details of this more complex strategy are left to the reader.

⁶For example, consider $a \rightarrow b \rightarrow c$, where all three nodes reside on separate PE's, **a** is marked, and **b** and **c** are unmarked. Suppose **expand-node-B(a)** results in $a \rightarrow \text{new} \rightarrow b \rightarrow c$ where **new** is on a fourth PE (thus **new** is marked). Now if an **add-reference** mutation were to occur using the three-node sequence **new-b-c**, there might not be any marking activity on the participant PE's.

5.4. Performance

In terms of contention for resources, the tasks responsible for marking in Scheme B meet our earlier goal of never leaving a node or other structure locked while marking is continued on some other PE. Given the same statistical properties of a computation graph, Scheme B exhibits improved performance over Scheme A in the following ways:

1. No extra room is required for each node in the graph, other than two bits to encode mark state and one for locking.
2. The only marking-tree structure on a PE is the set of values *local-root*, *mt-count*, and *mt-parent*, a total of three full words per PE.
3. Entries in the marking pool are primarily due to arcs in the graph that cross processor boundaries -- almost all local arcs cause entries to the stack instead. Tasks are also spawned if *local-mark()* fails to lock a child of a node popped from the stack -- how often this occurs depends on statistical properties of the graph and how active the mutating process is. Finally, note that the second argument of a *mark-B* task is only used to determine the *processor* to which a *return-B* task is spawned, and can therefore be shortened to just that processor name.
4. The entries on the stack are much smaller than a corresponding entry in the task pool; that is, each entry is smaller than a full word, since they are all local pointers. Also, these entries only appear once on the stack, as opposed to the two entries in the pool (one for *mark-B* and one for *return-B*).

Scheme B is an efficient, practical algorithm for distributed graph marking. Its primary shortcomings are the overhead for management of stack space, which is no more difficult than the analogous problem encountered in any other stack-based marking algorithm, and the management of the task pool, which must be dealt with anyway to realize the highly-parallel nature of the mutator.

6. Copying Nodes to Coordinate Local Marking

Our final algorithm (Scheme C) is similar to Scheme B in that each PE executes a local marking process in parallel with all task activity, and a global marking-tree is used to coordinate inter-PE marking. It is different in that the local marking process is a variation of a *copying garbage collector* rather than a stack-based marking algorithm. A nice feature of the resulting scheme is that memory is compacted; indeed the scheme becomes a full-blown garbage collector.

Although garbage collection and memory compaction are not our chief concerns, the popularity of copying collectors makes Scheme C a worthwhile development and demonstrates the utility of the marking-tree. The specifications for Scheme C are relegated to Appendix II, and the reader

who is interested in a copying-style marking process is encouraged to read that section, although no new information is added there that is crucial to the remainder of our discussion.

The degree of contention for shared resources, as well as the task pool space requirements of Scheme C, are essentially the same as for Scheme B. The mark state of a node is determined by whether or not the node has been "copied" and its relationship to the pointer that sweeps over copied nodes. The chief difference in space, of course, is that Scheme C requires room for both fromspace and tospace, thus twice the normal amount.

Scheme C is a practical distributed marking strategy, especially if used in situations in which a conventional copying collector would appear attractive (with the advent of large address-space architectures and efficient paging mechanisms, copying collectors have become fairly popular in modern LISP implementations). The scheme is also interesting in that it accomplishes a *true parallel compaction of memory*, which no other algorithm heretofore has accomplished. This includes the scheme proposed by Steele [29], where during compaction the mutator may create new nodes that place "holes" in the free area, and the scheme proposed by Baker [3], in which the executions of the mutator and collector are interleaved rather than executed in parallel.

7. "Uncooperative" Graph Mutations

Are there mutations for which no degree of cooperation can guarantee proper marking? To answer this first requires defining what is meant by "cooperation", for clearly the mutator could simply *wait until marking is complete*, and then perform the mutation. Disallowing cooperation of this sort (i.e., where the mutator waits for further action by the marking process), there are still mutations for which defining an effective form of distributed cooperation appears intractable.

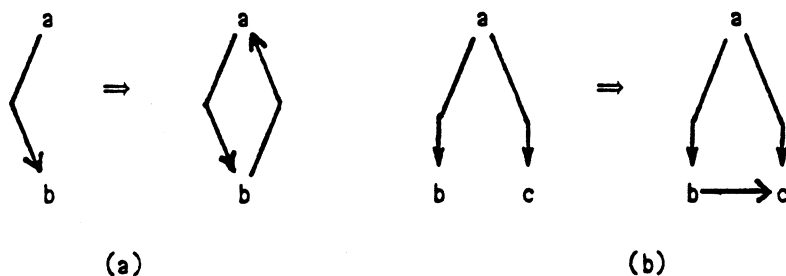


Figure 7-1: Examples of "uncooperative mutations"

For example, consider the simple mutation shown in Figure 7-1a as it executes concurrently with Scheme A. If **a** were unmarked and **b** marked, inserting an arc from **b** to **a** could violate the second invariant established in Section 4.2. This itself is not catastrophic, except that in such a situation there is no guarantee that *any* marking activity exists on the PE's containing nodes **a** and **b**, thus precluding any form of cooperation!

Another example of an "uncooperative" mutation is the one shown in Figure 7-1b. To see that this mutation cannot always cooperate with Scheme A, consider once again all combinations of the mark states of nodes **a**, **b**, and **c**, as shown in Figure 7-2. Note that the invariant violations due to the **a/b** mark state combinations unmarked/transient and transient/marked can be prevented by similar measures as before. However, the combination unmarked/marked (indicated by a question mark) appears to be intractable given our current marking strategies. In this situation adding the arc from **b** to **c** *may* violate the second invariant; that is, **c** may be unmarked. As above, no cooperation can remedy this since there is no known marking activity on the participant PE's.

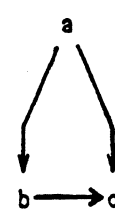
		mark state of node a			
		unmarked	transient	marked	
mark state of node b	unmarked	no coop	no coop	impos- sible	
	transient	coop w / b	no coop	no coop	
	marked	? 	coop w / a	no coop	

Figure 7-2: Difficulty with the new add-reference primitive (Scheme A)

Even though the above mutations appear unnecessary for evaluating functional programs, they are (at least in a graph-theoretic sense) more general than the previous set of primitives. This can be seen by noting that the original set of primitives is not capable of the mutation of Figure 7-1b, whereas the latter mutation together with **delete-reference** and **expand-node** can

generate all mutations created by the original set (the mutation of Figure 7-1a is a special case of Figure 7-1b, so it is sufficient to restrict our attention to the latter).

In previous parallel marking strategies for sequential computers (i.e., those proposed by Dijkstra and others), these "uncooperative" mutations do not arise because there is a centralized data-structure (such as a stack) that coordinates *all* marking; cooperation is easily induced by modifying this structure accordingly. There *is* a solution to our dilemma analogous to solutions in sequential systems, but it does not fit nicely into our distributed model of computation. The solution takes advantage of the fact that there is *one* source of marking activity that is always known: the marking activity at the root; that is, if marking is still active, the root must still be transient. To effect cooperation for the mutation of Figure 7-1b in the case where node *a* is unmarked and node *b* is marked, we simply execute a **mark-A** task on node *c* from the root! The full specifications for the resulting primitive are shown in Figure 7-3.

```

task-procedure new-add-reference(a,b,c)
  while-locking a do
    while-locking b do
      begin if unmarked(a) and transient(b)
        then << spawn mark-A(c,b);
              increment(mt-cnt(b)) >>
        else if transient(a) and marked(b)
          then << execute mark-A(c,a);
                increment(mt-cnt(a)) >>
        else if unmarked(a) and marked(b)
          then while-locking root do
              << execute mark-A(c,root);
                increment(mt-cnt(root)) >>;
      connect(b,c)
    end;
end;

```

Figure 7-3: A solution to the uncooperative mutation

This solution is not entirely satisfactory since, as mentioned, it is not effectively distributed in that *all* occurrences of this situation result in a **mark-A** task being spawned on a single, global node known to all (i.e., the root). This may be tolerable if the use of this primitive were minimized, or if it can be shown that the particularly troublesome mark state combination is rare.

An interesting alternative solution has been proposed by Jerry Leichter at Yale University. Observe first that if no delete-reference mutations were to occur, there would be no need for the other mutator primitives to exercise cooperation, except for those that add new nodes to the graph. Define a new mutation **kill-reference(a,b)** that does not actually delete the reference from **a** to **b**, but merely flags it as "killed". Then modify the mark tasks such that when a killed arc is traversed, it is also deleted (hence the marking process cooperates with the mutator, instead of the other way around). This strategy provides a distributed solution for our uncooperative mutations, but has two disadvantages. First, some nodes get marked that are actually known not to be part of the computation graph; and second, extensibility is impaired in that **children(n)** may increase arbitrarily as references from **n** are alternatively added and deleted.

8. Conclusions

Scheme A is a simple, effective algorithm for distributed graph marking. It is useful in demonstrating the nature of the marking-tree, but suffers from two inefficiencies; namely, each node in the graph requires space for marking-tree data, and the task pool space requirements are extreme. Schemes B and C improve on these inefficiencies through the use of a stack and dual address space, respectively, to coordinate local marking, while using the marking-tree to coordinate the overall marking process. Both schemes eliminate the marking-tree data stored in each node, and drastically reduce the space required for the task pool. The space for the pool becomes directly proportional to the number of inter-processor references, and will not be extreme if locality of reference is a dominant feature of programs (as empirical studies [7] seem to indicate), and if the graph distribution strategy reflects this locality. Scheme C has the advantage of compacting memory, but at the expense of using a dual address space.

All three algorithms may execute concurrently with a certain class of distributed graph mutations without affecting the distributed nature of the marking process or the mutations themselves. Such a set of graph mutations must at least have the following property:

Given a marking strategy M and a set of primitive mutations $P = \{p_0, p_1, \dots, p_n\}$ define system S to be the parallel execution of $M \cup P$. If it is possible for a mutation p_i to reach a state from which S would result in improper marking, then it must be possible for p_i to locate a local source of marking with which to invoke a proper form of cooperation.

Note that the mutations are considered collectively, since their interaction is crucial. Using this property as a guideline, a slightly broader class of mutations may be executed with Scheme A than with Schemes B or C. The reason for this is primarily that a transient node under Scheme A is closer to being "fully marked" than a transient node under Schemes B or C; that is, under Scheme A a transient node has not only been "touched", but also **mark-A** tasks have been spawned on each of its children. As a result, more cooperation is generally required; compare **expand-node-A** to **expand-node-B** or **expand-node-C**. On the other hand, there are more available sources of marking with which the mutations may cooperate. Recall that because of this we did not restrict **expand-node-A** as we did **expand-node-B** and **expand-node-C** -- the latter tasks are thus weaker in functionality than the former.

A final point worth noting is that none of our algorithms place any restrictions on the order of task execution. Furthermore, despite the need for locking mechanisms, none of the tasks responsible for marking leave a node or local variable locked on one PE while marking is continued elsewhere; indeed, none of them ever *nest* the locking of nodes (this is not true of the mutator tasks, some of which require nesting simply to ensure their own proper behavior). Finally, although the local marking processes of Schemes B and C need to nest the locking of local nodes only, they never leave one node locked while waiting for another to become unlocked. All of these factors contribute to our algorithms' distributed nature.

9. Acknowledgements

I am indebted to Robert Keller for initially inspiring this research, and to Jerry Leichter for his critical review of the work. Also thanks to Cathy Van Dyke for many helpful comments on earlier drafts of the manuscript. This work was begun at the University of Utah under NSF support and a fellowship from the University Research Committee, and is currently being continued under support of the Yale Department of Computer Science.

Appendix

I. Examples of Functional Graph Mutations

Figures 10-1, 10-2, and 10-3 show how three common combinator reductions can be simulated with the set of functional mutator primitives shown in Figure 3-1.

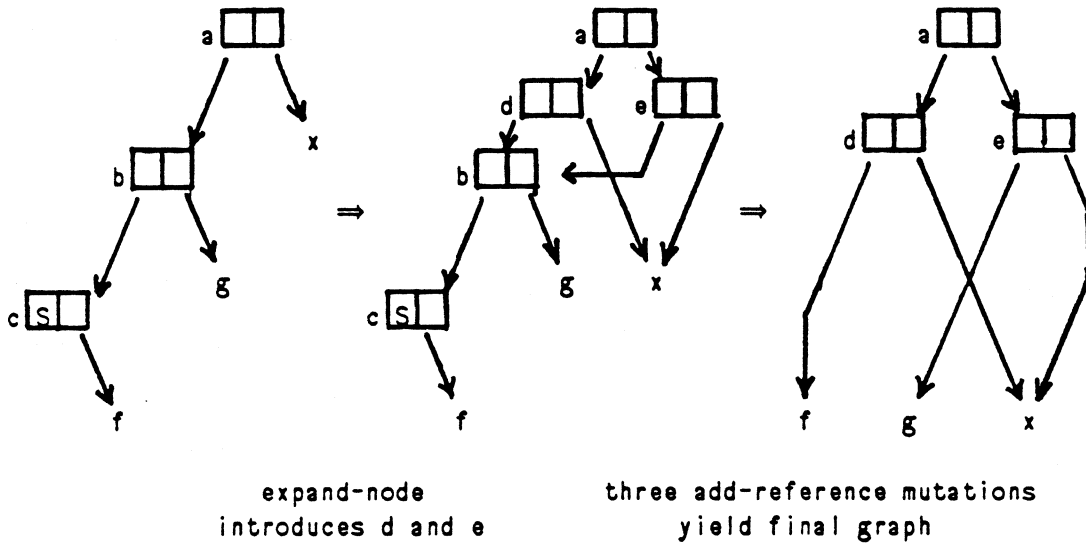


Figure 10-1: $S f g x \Rightarrow f x (g x)$

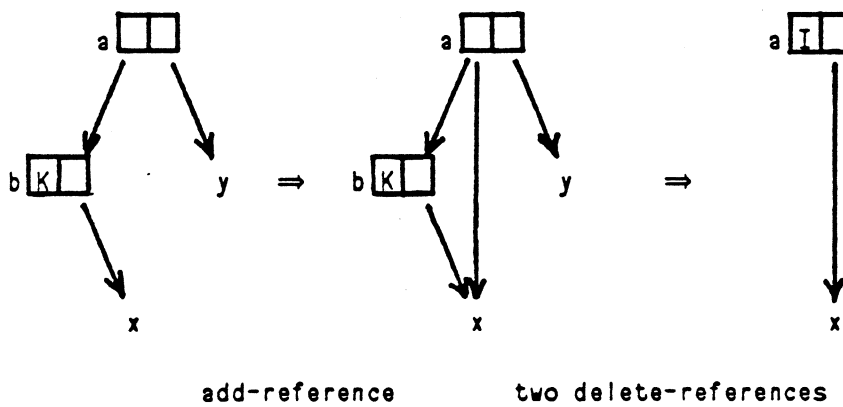
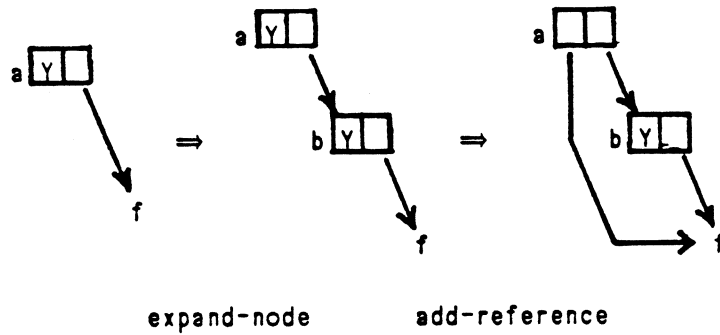


Figure 10-2: $K x y \Rightarrow x$

Figure 10-3: $Y f \Rightarrow f (Y f)$

II. The Third Algorithm (Scheme C)

As with a conventional copying collector, memory in each PE is divided into two contiguous regions; the *fromspace* and *tospace*. Let $from_{left}$, $from_{right}$, to_{left} and to_{right} be pointers to the boundary cells for these spaces, such that $from_{left} < from_{right}$ and $to_{left} < to_{right}$. The *tospace* has three pointers associated with it: $fs-left$ points to the first free node available from the left boundary of *tospace* (initially $fs-left = to_{left}$), and $fs-right$ is the analogous pointer to the first free node available from the right side of *tospace* (initially $fs-right = to_{right}$). Thus $fs-left$ and $fs-right$ form the boundaries of the *free-space*; if $fs-left = fs-right$ there are no free nodes available. The third pointer is $cnode$, and references the current node in *tospace* which is being traced by the local copying process. Initially $cnode = to_{left}$.

The value of $unmarked(n)$ is true if n is in *fromspace*, and false otherwise. We assume that a node's presence in *fromspace* or *tospace* can be determined by looking at the address of the node, so this predicate can be evaluated without looking at the node itself. Given that node n is in *fromspace*, $copy(n)$ returns a new node in *tospace* that is identical to n . New nodes are allocated by $copy$ starting at $fs-left$, which is then updated accordingly. Also, a forward pointer to the new location is left in the old location. The function $moved(n)$ returns true if node n (which must be in *fromspace*) has been copied into *tospace* (and false otherwise), and $forward(n)$ returns the reference to the new node.

The functions $allocate-from-right-side-of-tospace$ and $allocate-in-fromspace$ are used by the $expand-node$ primitive to access free nodes with which to build the new subgraph (we

assume that an error handler is invoked if there is no free-space available). To avoid interference between various tasks accessing shared variables, the above two functions as well as `copy`, `increment-by-2` (which adds two to its argument) and `decrement` are defined as indivisible operations. As with scheme B, we also assume a binary semaphore `mutex` exists on each PE.

Figure 11-1 shows the specifications for Scheme C. Each PE executes the procedure `local-copy`, which is analogous to `local-mark` in Scheme B, and the local pointers `local-root`, `mt-count`, and `mt-parent` serve precisely the same roles as in Scheme B. Marking is started in the normal way, by spawning a `mark-C` task on the root.

A complication to our new strategy is that once a node is copied into tospace, the old references to it (i.e., pointers to fromspace) must be updated. Thus for each `mark-C` task that is spawned, not only does one `return-C` task eventually get spawned, but also one `update` task. We wish that all `update` tasks complete before marking terminates, so each of them spawns a *second* `return-C` task, which is accounted for by incrementing `mt-count` by two whenever a `mark-C` task is spawned. With this in mind, compare `mark-C` to `mark-B`; nodes are copied into tospace instead of pushing them on the stack, and the old reference is updated as described above.

The procedure `local-copy` is analogous to a conventional copying collector, and otherwise has a strong resemblance to `local-mark`. As nodes are copied into tospace, the pointer `fs-left` increases. The pointer `cnode` is used to sweep over the copied nodes since their children have not been traced; `cnode` is in essence trying to "catch up" to `fs-left`. Once the node reference by `cnode` is successfully locked, its children are examined in turn:

1. If the child is not local, then a `mark-C` task is spawned on it and `mt-count` is incremented by two.
2. If the child is local but has already been moved, then the reference to it in `cnode` is simply updated with the forward pointer.
3. If the child is local, has not been moved yet (i.e., is unmarked), and is successfully locked, then it is copied into tospace.
4. If the child is local, has not been moved, but is already locked, then it is treated as a non-local node as in (1) above.

`local-copy` also continuously checks to see if local marking activity has ceased, in which case it behaves precisely as `local-mark` in the same situation.

```

task-procedure mark-C(n,par);
  while-locking n
  do if moved(n)
    then << spawn update(par,n,forward(n));
          spawn return-C(par) >>
    else begin spawn update(par,n,copy(n));
              [ if local-root = nil
                then << local-root := n;
                      mt-parent := par >>
                else spawn return-C(par) ]
    end;

task-procedure update(n,oldchild,newchild)
  while-locking n
  do << replace-child(n,oldchild,newchild)
      spawn return-C(n) >>;

task-procedure return-C(n);
  if n = rootpar then done := true
  else decrement(mt-count);

procedure local-copy();
while true do
begin while cnode ≠ fs-left and lock cnode
  do << for each x ∈ children(cnode)
      do if local(x)
        then if moved(x)
              then replace-child(cnode,x,forward(x))
              else if lock x
                   then << replace-child(cnode,x,copy(x));
                           unlock x >>
                   else << spawn mark-C(x,cnode);
                           increment-by-2(mt-count) >>
        else << spawn mark-C(x,cnode);
              increment-by-2(mt-count) >>;
      unlock cnode;
      increment(cnode) >>;
  [ if cnode = fs-left and mt-count = 0 and local-root ≠ nil
    then << spawn return-C(mt-parent);
          local-root := nil >> ]
end;

```

Figure 11-1: Scheme C: Local marking using a “copying collector” strategy

Parallel Mutations

As with the prior marking schemes, we assume unlimited concurrent execution of the functional mutator primitives. Correct marking behavior can again be achieved if proper cooperation is observed.

First define a node n as unmarked if it resides in fromspace, transient if it has been moved to tospace but has not been traced (i.e., $\text{cnode} \leq n < \text{fs-left}$), and marked if it is in tospace and has either been traced ($n < \text{cnode}$) or was allocated from the right end of the free-space ($n > \text{fs-right}$). An additional requirement for a marked node is that we guarantee that all of its references to fromspace get updated properly.

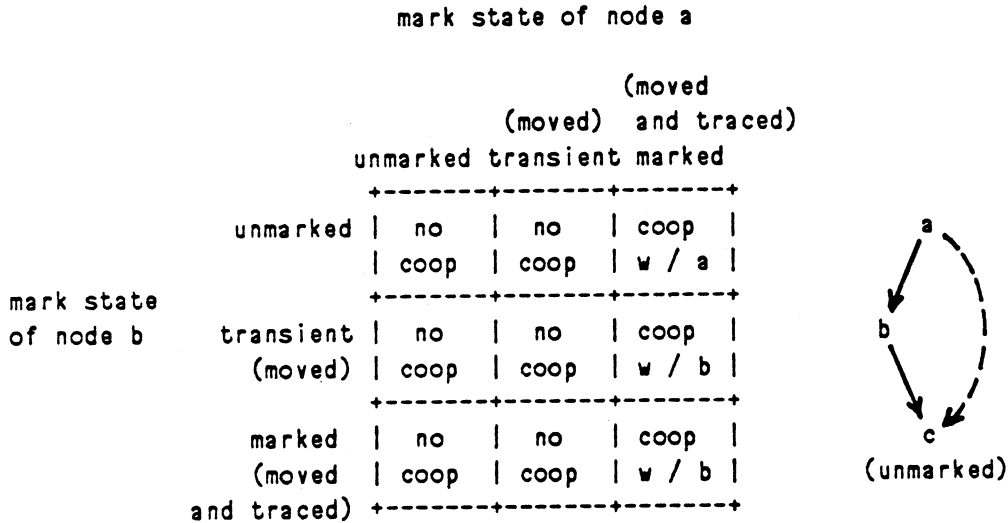


Figure 11-2: Cooperation by the add-reference primitive (Scheme C)

Figure 11-2 shows the nine mark state combinations for the **add-reference** primitive. Cooperation is only needed when **a** is marked and **c** is unmarked (remember that **c**'s being unmarked can be determined without locking it). Either of two situations can then arise:

1. If **b** is unmarked, then marking must still be active on **proc(a)**; spawning **mark-C(c,a)** and incrementing the appropriate **mt-count** is sufficient cooperation.
2. If **b** is transient or marked, both **a**'s and **b**'s reference to **c** must be updated. This can be accomplished by first executing **mark-C(c,b)** (since marking must still be active on **proc(b)**), updating the appropriate **mt-count**, and updating the value of **c** with the pointer to the new node.

The reader should convince him/herself that this cooperation is sufficient to maintain the meaning of a node's mark state as defined above. Figure 11-3 shows the specifications for the three new cooperating mutator primitives -- note again that **delete-reference** requires no cooperation.

```

task-procedure delete-reference-C(a);
  while-locking a do
    << b := select-child(a);
      disconnect(a,b) >>;

task-procedure add-reference-C(a);
  while-locking a do
    << b := select-child(a);
      while-locking b do
        begin c := select-child(b);
          if marked(a) and unmarked(c)
            then if unmarked(b)
              then << spawn mark-C(c,a);
                increment-by-2(mt-count(proc(a))) >>
              else << execute mark-C(c,b);
                increment-by-2(mt-count(proc(b)));
                c := forward(c) >>;
          connect(a,c)
        end >>;

task-procedure expand-node-C(a,g);
  while-locking a do
    << if marked(a) then allocate-from-right-side-of-tospace(g)
      else allocate-in-fromspace(g);
      splice-in-subgraph(a,g) >>;

```

Figure 11-3: Cooperating Mutator Primitives for Scheme C

We restrict **expand-node** as was done for Scheme B; the more general unrestricted version has a solution similar to that outlined in Section 5.2. **expand-node** is specified in Figure 11-3 in such a way as to allocate space for **g** from the *right* side of the free-space if **a** is marked (since there is no need for **cnode** to scan it), and in *fromspace* if **a** is unmarked or transient. The latter strategy is a simple way to effect cooperation; that is, the new graph **g** will subsequently get moved to *tospace*, but *only if it should*. Indeed, if **a** is unmarked, the **expand-node** mutation may be an *irrelevant task*, meaning that **g** will not get copied, which is what is desired.

References

- [1] Almes, G.T.
Garbage Collection in an Object-Oriented System.
PhD thesis, Carnegie-Mellon University, June, 1980.
- [2] Baker, H.G. and Hewitt, C.
The incremental garbage collection of processes.
AI Working Paper 149, Mass. Institute of Technology, July, 1977.
- [3] Baker, H.G. Jr.
List processing in real time on a serial computer.
CACM 21(4):280-294, April, 1978.
- [4] Bishop, P.
Computer Systems with a Very Large Address Space and Garbage Collection.
PhD thesis, Laboratory for Computer Science, Mass. Institute of Technology, May, 1977.
- [5] Chang, E.
PhD thesis, University of Waterloo, , .
- [6] Chang, E.
Decentralized deadlock detection in distributed systems.
Technical Report, University of Victoria, Victoria, B.C., Canada, 1978.
- [7] Clark, D.W.
An empirical study of list structure in LISP.
CACM 20(2):78-87, February, 1977.
- [8] Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S. and Steffens, E.F.M.
On-the-fly garbage collection: an exercise in cooperation.
CACM 21(11):966-975, November, 1978.
- [9] Dijkstra, E.W., Scholten, C.S.
Termination detection for diffusing computations.
Inf. Proc. Letters 11(1):1-4, August, 1980.
- [10] Francez, N.
An application of a method for the analysis of cyclic programs.
IEEE Trans. on Software Engineering 4(5):371-377, September, 1978.
- [11] David Gries.
An exercise in proving parallel programs correct.
CACM 20(12):921-930, December, 1977.
- [12] Grit, D.H. and Page, R.L.
Deleting irrelevant tasks in an expression oriented multiprocessor system.
ACM Transactions on Programming Languages and Systems 3(1):49-59, January, 1981.

- [13] Hibino, Y.
A practical parallel garbage collection algorithm and its implementation.
Sigarch Newsletter 8(3):113-120, May, 1980.
- [14] Hudak, P.
Object and Task Reclamation in Distributed Applicative Processing Systems.
PhD thesis, University of Utah, July, 1982.
- [15] Hudak, P. and Keller, R.M.
Garbage collection and task deletion in distributed applicative processing systems.
In Park et al. (editors), *Symposium on Lisp and Functional Programming*, pages
168-178. ACM, August, 1982.
- [16] Keller, R.M., Lindstrom, G., and Patil, S.
A loosely-coupled applicative multi-processing system.
In *AFIPS*, pages 613-622. AFIPS, June, 1979.
- [17] Knuth, D.E.
The Art of Computer Programming.
Addison-Wesley Publishing Company, Reading, Mass., 1973.
- [18] Kung, H.T. and Song, W.
An efficient parallel garbage collection system and its correctness proof.
Technical Report, Department of Computer Science, Carnegie-Mellon Univ., September,
1977.
- [19] Lamport, L.
On-the-fly garbage collection: once more with rigor.
Technical Report, CA-7508-1611, Massachusetts Computer Associates, Inc., August, 1975.
- [20] Lamport, L.
Garbage collection with multiple processors: an exercise in parallelism.
Technical Report, CA-7602-2511, Massachusetts Computer Associates, Inc., February,
1976.
- [21] Lieberman, H. and Hewitt, C.
A real time garbage collector that can recover temporary storage quickly.
AI Memo 569, Mass. Institute of Technology, Cambridge, Mass., April, 1980.
- [22] Mago, G.A.
A network of microprocessors to execute reduction languages, Part I.
International Journal of Computer and Information Sciences 8(5):349-385, March, 1979
revised.
- [23] Chandy, K.M., Misra, J.
Distributed computation on graphs: shortest path algorithms.
Commun. ACM 25(11):833-837, November, 1982.
- [24] Misra, J., Chandy, K.M.
A distributed graph algorithm: knot detection.
ACM Trans. Program. Lang. Syst. 4(4):678-686, October, 1982.

- [25] Muller, K.C.
On The Feasibility Of Concurrent Garbage Collection.
PhD thesis, Tech. Hogeschool Delft, March, 1976.
- [26] Nori, A.K.
A storage reclamation scheme for applicative multiprocessor system.
Master's thesis, Department of Computer Science, University of Utah, December, 1979.
- [27] Owicki, S.
Axiomatic Proof Techniques For Parallel Programs.
PhD thesis, Cornell University, Department of Computer Science TR 75-251, July, 1975.
- [28] Owicki, S. and Lamport, L.
Proving liveness properties of concurrent programs.
Technical Report 57 (S&L 1), Stanford Univ./SRI International, October, 1980.
- [29] Steele, G.L. Jr.
Multiprocessing compactifying garbage collection.
CACM 18(9):491-500, September, 1975.
- [30] Treleaven, P.C., and Mole, L.F.
A multi-processor reduction machine for user-defined reduction languages.
In *Proc. of the 7th Int. Symposium on Computer Architecture*, pages 121-130. IEEE,
May, 1980.
- [31] Turner, D.A.
A new implementation technique for applicative languages.
Software-Practice and Experience 9:31-49, 1979.
- [32] Wadler, P.L.
Analysis of an algorithm for real time garbage collection.
CACM 19(9):495-508, September, 1976.