Distributed Task and Memory Management

Paul Hudak

Yale University

Department of Computer Science

New Haven, CT

Abstract

A model of *distributed graph reduction* is described that has features common to many distributed computing systems: a program (represented as a graph) is partitioned and dynamically distributed among an arbitrary number of processing elements having only local store, and computation takes place as *tasks* are propagated between vertices in the graph. Specific problems are addressed that are inherent in a computing model of this sort, including garbage collection, detecting deadlock, deleting tasks, and the dynamic prioritization of tasks. By characterizing these problems in terms of graph connectivity, a *decentralized graph-marking algorithm* is shown to provide an effective solution. This algorithm is unique in that it allows marking a distributed graph whose connectivity is continually changing.

1. Introduction

Graph reduction holds promise as a simple, effective model on which to base highly-parallel computation. The idea is simple enough, indeed emerges quite naturally from the reduction rules of the λ-calculus, where the absence of side-effects guarantees that only a program's "data dependencies" prescribe an ordering on the creation of values. Graph reduction becomes especially attractive when modeling the *combinatory calculus*, since all occurrences of bound variables are removed, obviating the need for the centralized *environment* implied by β-reduction. The resulting strategy has a completely homogeneous, "diffused" nature -- there is no centralized structure that could become a bottleneck, and no extraneous control-flow dependencies are introduced as in most conventional models.

Because of these beguiling properties, we have been exploring a model of *distributed graph reduction*, in which the computation graph is partitioned into subgraphs that are distributed among autonomous processing elements having only local store and communicating via messages. In this paper we address specific problems in this model concerning the management of the distributed graph and its subsequent mutations, including garbage collection, finding deadlocked regions of the computation, finding and deleting tasks determined to be no longer relevant to the computation, and the dynamic prioritization of tasks. All of these problems can be characterized in terms of *reachability properties* of the distributed graph, so it is not surprising that our solutions rely on versions of a *decentralized graph-marking algorithm* that executes *concurrently with the graph reduction process*; that is, simultaneously with mutations to the graph.

Although the problems we address are couched in terms of graph reduction, variations of them are common among many distributed computing systems. For example, the dependencies between sub-processes in most distributed computations can generally be represented as a directed graph, where the spawning or completion of sub-processes corresponds to mutations to the graph, and the "irrelevant tasks" that we describe in Section 3.2 are not unlike "orphans" [9]. We feel that the solutions presented here are applicable to a variety of distributed computing models.

2. Distributed Graph Reduction

Graph reduction involves representing a program as a directed graph (called the **computation graph**) whose vertices are labeled with primitive operators and values, and whose edges reflect data dependencies between operators and values. Program execution is accomplished via transmutations (called *reductions*) to the graph. This includes not only relabeling vertices with their ultimate value (eventually labeling the root of the graph with the result of the overall computation), but also changing the connectivity of the graph, which can result in the implicit deletion of some vertices as well as the explicit creation of new ones. (New vertices, for example, are added as the result of a function invocation.) The graph thus expands and contracts as the computation progresses.

The most *conservative* implementation of graph reduction is sequential execution on a conventional machine,[1] whereas the most *radical* implementation would approach that of dataflow researchers, in which each vertex is assigned to a single processing element. We are interested in highly-parallel implementations, but wish to avoid the high communication overhead inherent in the fine-grained dataflow approach. In our model the computation graph is divided into a number of subgraphs (called **partitions**), each of which is assigned to an autonomous **processing element** (PE) that is responsible for the reductions in that portion of the graph. The model is thus more akin to conventional distributed computing models.

2.1. A Formal Model

The mechanistic details by which the distributed graph reductions are carried out are straightforward, although tedious. A formal description might follow the lead of Kennaway and Sleep [8] in which each primitive operation is specified in terms of a version of CCS, Milner's calculus of communicating systems; or the approach that we discuss in [5, 6] of specifying each reduction in terms of a set of primitive graph mutations. In this paper we are concerned with only two particular aspects of the reduction process: (1) the connectivity of the computation graph, and (2) the propagation of work. To this end, we present the following abstract model:

A **task** is the smallest unit of work, and a **process** is a collection of tasks that accomplish some particular goal (the primary process being the **reduction process**, responsible for program execution). Tasks propagate between adjacent vertices in the graph (and therefore may cross partition boundaries), and are responsible for primitive operations such as data manipulation, primitive graph mutations, and the spawning of additional tasks to continue the computation. An unexecuted task **t** is represented as a pair **<s,d>**, where **s** (the **source**) is the vertex from which the task is spawned, and **d** (the **destination**) is the vertex to which work is propagated.[2] In this sense an unexecuted task may be viewed simply as a *message* from one vertex to another. (There are times when there is no need to remember the source vertex, in which case we represent the task as "**<-,d>**".) We consider task execution as an *atomic operation*.

The **value** of a vertex refers to its unique ultimate value computed by the reduction process. There is a distinguished vertex called the **root** at which the reduction process is initiated, and whose value represents the result of the overall computation. As the evaluation proceeds, the tasks in the reduction process modify the computation graph to reflect the local status of the evaluation. In particular, certain sets of outgoing edges are kept current for each vertex **v**,[3] and are represented by:

1. **args(v)**, the set of vertices reflecting the original data dependencies in the program. That is, the value of each element of args(**v**) may be needed directly to compute the value of **v**.
2. **req-args(v)** $\subseteq$ **args(v)**, the set of vertices whose values have been requested by **v**. That is, a task has been spawned on each element of req-args(**v**) that should eventually return a value to **v**.
3. **requested(v)**, the set of vertices that have requested **v**'s value, but to which **v** has not yet replied.

With this model it is not hard to imagine how a computation takes place. For example, consider a vertex **v** that represents a strict function[4] requiring the values of vertices $\mathbf{d_1}$, $\mathbf{d_2} \in$ **args(v)**. The execution of a task **<s,v>** in quest of **v**'s value would result in adding **s** to

---

[1] The ordering of operations being governed by the particular reduction strategy being used, the two standards being *normal order* and *applicative order* reduction.

[2] A task normally contains other information that does not concern us here, such as the task type, an atomic value, etc.

[3] These edges are simply the ones that would be involved in any graph reduction scheme, except that in sequential graph reduction some of them are implicit in the use of an auxilliary structure such as a stack to traverse the graph [12].

[4] A function $f(x_1,x_2,\ldots,x_n)$ is *strict* in x_i if $f(x_1,\ldots,x_{i-1},\perp,x_{i+1},\ldots,x_n) = \perp$ for all values of x_j, $j \neq i$.

requested(**v**) and spawning tasks $\langle \mathbf{v},\mathbf{d}_1 \rangle$ and $\langle \mathbf{v},\mathbf{d}_2 \rangle$. Eventually the tasks $\langle \mathbf{d}_1,\mathbf{v} \rangle$ and $\langle \mathbf{d}_2,\mathbf{v} \rangle$ "return" to **v** with the appropriate values, in which case a result is computed and tasks $\langle \mathbf{v},\mathbf{s}_1 \rangle$ are spawned for each $\mathbf{s}_1 \in$ requested(**v**). Note that this scenario reflects a *demand-driven evaluation strategy*, which is the most general since it supports normal-order reduction.

2.2. Reachability Properties

The abstract graph reduction model that we have presented is an over-simplification of what might occur in practice, but is sufficient to characterize several properties that are germane to our discussion. We first define two forms of reachability in the computation graph, one based on data dependence, the other on task propagation. Let $\mathcal{V}$ denote the finite set of all vertices. We write $\mathbf{x} \rightarrow \mathbf{y}$ (read "x *directly depends* on y") if **x** is a vertex whose value might directly depend on the value of **y**; that is:

$$\mathbf{x} \rightarrow \mathbf{y} \quad \Leftrightarrow \quad \mathbf{y} \in \text{args}(\mathbf{x}).$$

We also define the relation $\overset{*}{\rightarrow}$ as the reflexive, transitive version of $\rightarrow$.

Similarly, we write $\mathbf{x} \mapsto \mathbf{y}$ if task activity might propagate directly from vertex **x** to vertex **y**. In our model this can happen if either **x** returns a value to **y**, or **x** requests the value corresponding to **y**. Since task activity has already propagated to vertices in req-args(**x**), we have:

$$\mathbf{x} \mapsto \mathbf{y} \quad \Leftrightarrow \quad \mathbf{y} \in \text{requested}(\mathbf{x}) \vee \mathbf{y} \in (\text{args}(\mathbf{x}) - \text{req-args}(\mathbf{x})).$$

where "–" denotes set difference. As above, $\overset{*}{\mapsto}$ is defined as the reflexive, transitive version of $\mapsto$.

Finally, we define the following important sets of vertices:

- $\mathcal{R} \equiv \{ \mathrm{v} \in \mathcal{V} \mid \mathrm{root} \overset{*}{\rightarrow} \mathrm{v} \}$
- $\mathcal{T} \equiv \{ \mathrm{v} \in \mathcal{V} \mid (\exists \langle \mathrm{s},\mathrm{d} \rangle)\ (\mathrm{d} \overset{*}{\mapsto} \mathrm{v} \vee \mathrm{s} \overset{*}{\mapsto} \mathrm{v}) \}$
- $\mathcal{F} \equiv$ a known set of **free vertices**.

Thus $\mathcal{R}$ is the set of vertices reachable from the root, $\mathcal{T}$ is the set of vertices to which task activity might propagate, and $\mathcal{F}$ (which is disjoint from $\mathcal{R}$ and $\mathcal{T}$) is analogous to the "free-list" in conventional list-processing systems. Initially (i.e., prior to program execution), there is only one task, $\langle$-,**root**$\rangle$ (which has not yet executed), and for all $\mathbf{v} \in \mathcal{V}$, requested(**v**) = req-args(**v**) = $\emptyset$. This in turn implies that $\mathcal{R} = \mathcal{T}$ and $\mathcal{F} = \mathcal{V} - \mathcal{R}$. As we shall demonstrate shortly, all of these relationships change as the computation proceeds.

3. Static Characterization of Task and Graph Management

Although we have not given the details of the reduction process, it is still possible to describe certain global properties that are consistent with the semantics of the computation, and that are sufficient for our discussion.

3.1. Garbage and Deadlocked Vertices

Any system that performs graph reduction in the style so far described will need a mechanism to handle the allocation of new vertices and, more importantly, the reclamation of vertices no longer being used. New vertices are acquired from the set $\mathcal{F}$ of free vertices. In terms of reachability, the set $\mathcal{R}$ is precisely the set of vertices involved in the computation. Thus the set $\mathcal{GAR}$ of "garbage vertices" is expressed by:

Property 1: $\mathcal{GAR} = \mathcal{V} - \mathcal{R} - \mathcal{F}$.

(where set difference in left associative). Note that $\mathcal{GAR}$ and $\mathcal{T}$ are not necessarily disjoint; more will be said about this in Section 3.2.

Another important issue is the possibility of **deadlock**, which in graph reduction corresponds to an expression whose ultimate value is *undefined* (generally denoted "$\perp$"), and is manifested by a subgraph in which task activity has ceased, yet the subgraph's value is being awaited by some other vertex. For example, the definition $\mathbf{x} = \mathbf{x} + \mathbf{1}$ corresponds to the graph shown in Figure 3-1. Deadlock results when this expression is evaluated, since $\mathbf{x} \in$ args(**x**) and thus **x** will await its own value before trying to compute that same value.

(In this and subsequent figures, a vertex is shown as a circle, and a task $\langle \mathbf{s},\mathbf{d} \rangle$ by a triangle with a solid arc to **d**, and a dashed one to **s**. A solid arc from a vertex **x** to a vertex **y** denotes the presence of **y** in **args(x)**, and an asterisk beside such an arc denotes that **y** is in **req-args(x)**. A dashed arc from **x** to **y** implies that **y** is in **requested(x)**.)

A mechanism is needed to discover deadlocked regions of the computation graph,[5] but static solutions (i.e., compile-time analyses [7, 13]) are not general enough. In our model a vertex **v** has **deadlocked** if it is reachable from the root (i.e., via the relation $\overset{*}{\rightarrow}$) but not from any task (i.e., via $\overset{*}{\mapsto}$), because this implies that the root depends on **v**'s value yet no task can ever propagate there to compute it. This leads us to the characterization of the set $\mathcal{DL}$ of deadlocked vertices:

Property 2: $\mathcal{DL} = \mathcal{R} - \mathcal{T}$.

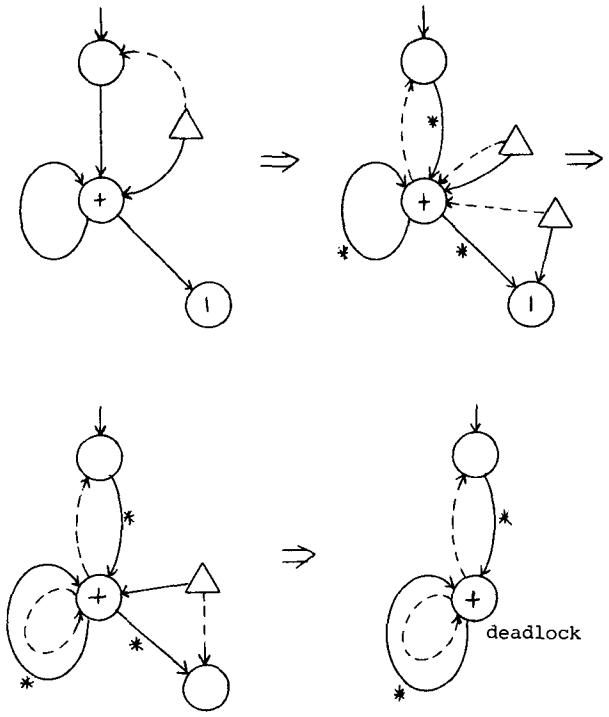

Figure 3-1: Deadlocked Computation

3.2. Task Propagation Properties

A key aspect of our graph reduction strategy is the inclusion of mechanisms to "eagerly" invoke computations whose ultimate value might not be needed. Such a strategy may result from a desire for increased parallelism,[6] or from the use of certain non-deterministic operators. In any case we differentiate such **eager tasks** from **vital tasks** (those whose results are known to be needed). The inclusion of eager tasks seems innocuous enough, until one considers that:

1. Eager tasks "compete" with vital ones, so it is reasonable to assign a higher *priority* to the latter, especially when resources are limited.
2. It may be discovered subsequently that the value of an eagerly invoked subcomputation *is* needed, in which case all of the resulting eager tasks that comprise the subcomputation are now vital, warranting an upgrade in priority.
3. Worse, one may discover that an eagerly invoked subcomputation is *not* needed -- the tasks comprising such a subcomputation are called **irrelevant tasks**. These tasks behave no differently than any others, and may distribute through the system generating an arbitrarily large (and irrelevant) parallel workload; indeed, the subcomputation may be non-terminating.
4. If shared subexpressions are allowed the changes in a task's priority are even more subtle. In fact, a *fourth* type of task arises when an eager task's destination is dereferenced from the vertex that initially spawned the eager computation, but is still accessible from some other vertex that has not requested its value yet. We refer to this as a **reserve task**.

Thus an initially eager task may expand into a highly parallel workload of many other tasks, each of which is subject to change as the computation proceeds, especially when shared subexpressions are allowed. Figure 3-2 shows examples of the development of each task type. (Arcs whose value has been demanded for eager or vital computation are shown marked with "***e**" or "***v**", respectively.) This example shows the evaluation of the nonsensical expression "if p then d else c, where p = if true then (a+1) else (a+b+c)". Note that the lower "if" has eagerly requested its conditional branches, but subsequently finds that its predicate is true, which "implicitly" changes the task types.

We wish to be able to find all occurrences of the four task types mentioned. To adequately characterize the myriad of possibilities we must slightly refine our model. Specifically, for each vertex **x** we wish to keep track of how elements in **args(x)** were requested. Define the *disjoint* sets **req-args$_e$(x)** and **req-args$_v$(x)** to be the set of vertices "eagerly requested" and "vitally requested," respectively, by vertex **x**, and define **req-args$_r$(x)** to be the remaining elements in **args(x)**. It is then natural to define the sets $\mathcal{R}_v$ and $\mathcal{R}_r$ just as $\mathcal{R}$, except that reachability is only considered through **req-args$_v$** and **req-args$_r$**, respectively.

[5]It should be pointed out that in reduction over continuous functions, if any region of the graph were to deadlock, then the entire graph would eventually deadlock, but only if there were no non-terminating subcomputations. Furthermore, it may be desirable to introduce a predicate **is-bottom** to facilitate recovery from deadlocked subcomputations. Such a non-monotonic function may introduce semantic irregularities in that, for example, least fixed-points are not guaranteed and non-deterministic behavior can result. Nevertheless, the use of such "pseudo-functions" is likely, especially in a multi-user environment where one would not expect the entire system to deadlock just because one user's program has deadlocked!

[6]Studies have shown that most programs simply do not have enough *strict parallelism* to effect an overall speed-up of more than a factor of 5 or so. One way to increase the *effective parallelism* is to "eagerly" compute some values (resources permitting) that ultimately might not be needed.

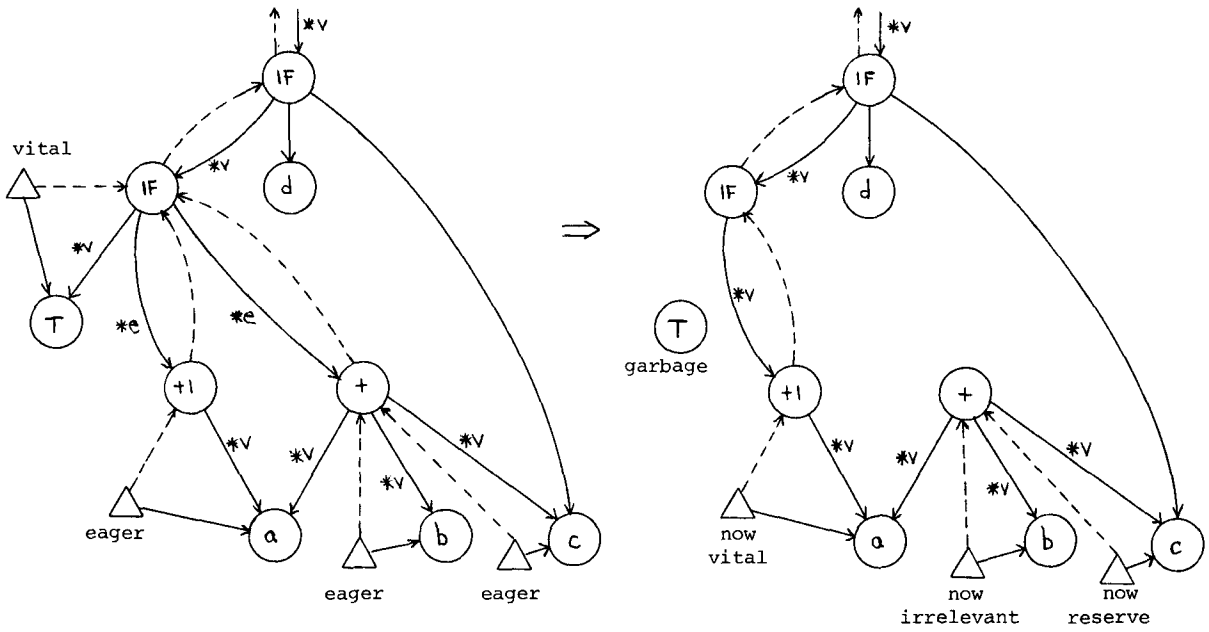

Figure 3-2: Vital, eager, irrelevant, and reserve tasks.

We would also like to define the set of vertices that have been eagerly requested. Note, however, that a vertex **v** that has been eagerly requested may itself vitally request some other node **w**; yet relative to the overall computation **w** should only be considered eagerly requested. Thus $\mathcal{R}_e$ is defined as the set of vertices reachable from the root via **req-args**$_v$ and *at least one* arc involving **req-args**$_e$. Note that because of shared subexpressions, $\mathcal{R}_e$, $\mathcal{R}_v$, and $\mathcal{R}_r$ are not necessarily disjoint.

With the above definitions we then arrive at a refined version of $\mathcal{DL}$:

Property 2': $\mathcal{DL}_v = \mathcal{R}_v - \mathcal{T}$.

as well as definitions for vital, irrelevant, and reserve tasks:

Property 3:

The set of *vital tasks* is
$\mathcal{VIT} = \{ \langle s,d \rangle \mid d \in \mathcal{R}_v \}$.

Property 4:

The set of *eager tasks* is
$\mathcal{EAG} = \{ \langle s,d \rangle \mid d \in (\mathcal{R}_e - \mathcal{R}_v) \}$.

Property 5:

The set of *reserve tasks* is
$\mathcal{RES} = \{ \langle s,d \rangle \mid d \in (\mathcal{R}_r - \mathcal{R}_e - \mathcal{R}_v) \}$.

As part of the normal reduction process, a vertex **x** that has eagerly requested the value of vertex **y** "dereferences" **y** if its value is determined to be irrelevant. This is done not only by removing the reference to **y** from **req-args**$_e$**(x)**, but also removing **x** from **requested(y)**. Once this is done, any task whose destination is not reachable from the root must be irrelevant, leading us to our final property:

Property 6:

The set of *irrelevant tasks* is
$\mathcal{IRR} = \{ \langle s,d \rangle \mid d \in (\mathcal{V} - \mathcal{R} - \mathcal{F}) = \mathcal{GAR} \}$.

The reader may wish to verify that the tasks shown in Figure 3-2 are properly characterized by Properties 3 through 6. The key idea to note here is that the complex interactions between shared subexpressions are handled quite naturally in our model by considering only reachability properties of the computation graph. A Venn diagram that summarizes these relationships is shown in Figure 3-3.

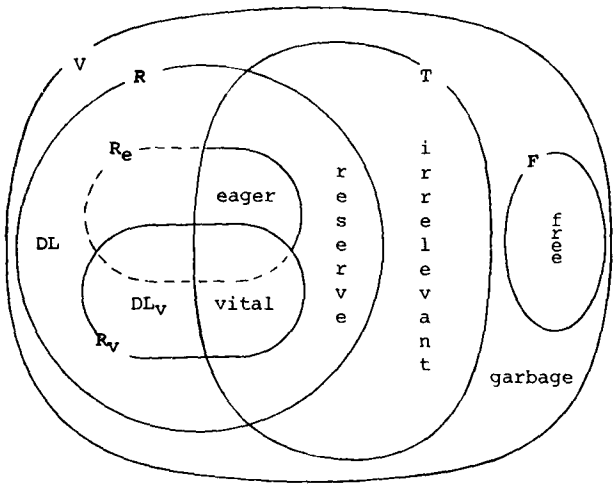

Figure 3-3: Summary of Reachability Characteristics

4. Decentralized Graph-Marking

Now that the model is complete, our problem can be summarized as one of finding all garbage and deadlocked vertices, as well as all vital, eager, irrelevant, and reserve tasks. Once found, the appropriate action may be taken, such as adding elements of $\mathcal{GAR}$ to $\mathcal{F}$, expunging irrelevant tasks, and so on.

It is clear that a simple static graph-marking algorithm could be used to mark vertices in such a way as to provide the information implied by Properties 1-6. This would require, however, that the computation be halted while marking takes place. Furthermore, most marking algorithms are sequential and use a centralized control. What we desire instead is a graph-marking algorithm that is:

1. highly parallel and easily extensible,
2. decentralized, not relying on any centralized data or control, and
3. able to execute concurrently with the graph reduction process; that is, simultaneously with mutations to the graph.

Our strategy is to use such an algorithm to mark the computation graph (a *mark phase*), and then perform any adjustments deemed necessary such as adding garbage nodes to the free list and deleting irrelevant tasks (a *restructuring phase*). This cycle is repeated endlessly, and the algorithms are written in such a way that vertices incorrectly identified in one cycle are properly identified in the next.

Note that this strategy is similar to that used in "conventional" parallel garbage collectors for *sequential* computers [1, 4, 11]. However, the lack of an effective marking algorithm for *distributed* computation has prevented extension of the technique to highly-parallel systems, and has prompted alternative schemes such as *reference counting* to accomplish garbage collection in a distributed environment. Reference counting has particular deficiencies that make it unsuitable for our purposes, such as the inability to reclaim *self-referencing structures*, and the inability to perform the tracing necessary to identify task types.

Our purpose in this paper is to describe a technique for *finding* the vertices and tasks of interest. We therefore concentrate solely upon the mark phase, leaving the specifications of the restructuring phase as something to be tailored to a particular system (see [5]).

4.1. The Basic Algorithm

A simplified form of the algorithm will be described first, which will then be modified to accomplish our particular goals. Intuitively, the algorithm works by dynamically building a *spanning tree* on the computation graph that we call the **marking tree**. Marking is initiated by spawning a **mark task** at the root, which propagates additional mark tasks to the appropriate children of the root. These tasks continually propagate in a "forward" direction through the graph, until either a leaf is found or an already marked vertex is encountered. In either case a **return task** is then spawned that works its way "backward" through the graph. When all of the mark tasks spawned from a vertex **v** have "returned" in this manner, a return task is recursively spawned on **v**'s parent in the marking tree. To implement this, a *count* is maintained for each vertex **v** of the number of mark tasks that have been spawned from it but have not "returned" (called **mt-cnt(v)**), in addition to a *pointer* to the vertex's parent in the marking tree (called **mt-par(v)**).

We now have two sets of tasks -- those involved in the reduction process, and those involved in the marking process. Both types can be represented using the notation "**<s,d>**", but although we are not concerned with the details of the reduction tasks, we are very much concerned with the internal behavior of the tasks performing marking. To specify this behavior we use an Algol-like syntax in which tasks are defined in a way similar to procedures except that the keyword "task-procedure" is used and we occasionally use ≪...≫ as an abbreviation for begin...end.

Figure 4-1 shows the specifications for the mark and return tasks (**mark1** and **return1**). The lexically first argument in a task's parameter-list corresponds to the destination **d** in the **<s,d>** representation. An instance of a task may either be **spawned** or **executed**. The statement **spawn f(x)** causes an instance of the task **f** to be created and spawned for execution. 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)** creates an instance of the task **f** for immediate execution. Execution continues with the statement following **execute f(x)** only after the task has finished executing. Finally, as mentioned earlier, we assume that task execution is *atomic* with respect to the vertices that it manipulates. That is, no task will see the "partial results" of some other task's efforts. For simplicity we choose not to provide the details of an appropriate mutual exclusion mechanism.

In the specifications of Figure 4-1 we assume an arbitrary graph with a root **root**. Each vertex **v** has a set of children denoted **children(v)**. A vertex **v** can be in one of three states:[7]

1. **unmarked**, meaning that no task of the form **mark1(v,p)** has executed.
2. **transient**, meaning that a task of form **mark1(v,p)** *has* executed, but that the new tasks (of form **mark1(c,v)**, **c∈children(v)**) spawned from **v** have not all "returned" (and therefore **mt-cnt(v)** > 0).

```
task-procedure mark1(v,par);
    if unmarked(v)
    then begin touch(v);
               mt-par(v) := par;
               for each c ∈ children(v)
               do << spawn mark1(c,v);
                     increment(mt-cnt(v)) >>;
               if mt-cnt(v) = 0
               then << mark(v);
                       spawn return1(par) >>
         end
    else spawn return1(par);

task-procedure return1(v);
    if v = rootpar then done := true
    else begin decrement(mt-cnt(v));
               if mt-cnt(v) = 0
               then << mark(v);
                       spawn return1(mt-par(v)) >>
         end;
```

Figure 4-1: Simplified Marking Algorithm

[7]These states are similar to Dijkstra's *white*, *gray*, and *black* cells [4], but are subtlely different due to the distributed system context.

3. **marked**, meaning that marking is complete for vertex **v**; that is, all of the newly spawned **mark1** tasks have returned (and therefore **mt-cnt(v)** = 0).

The predicates **unmarked**, **transient** and **marked** return true if a vertex is unmarked, transient, or marked, respectively, and the operations **unmark**, **touch**, and **mark** make a node unmarked, transient, and marked, respectively.

Initially the mt-cnt of all vertices is zero. Marking is started by spawning the task **mark1(root,rootpar)**, where **rootpar** is a dummy node that **return1** uses to detect termination. Marking is complete when the special flag **done** is true.

4.2. Cooperating Mutator Primitives

So far it should be clear that our algorithm is highly parallel (via the spawning of mark tasks) and decentralized (via the use of the marking tree, imbedded in the computation graph, to coordinate marking instead of a centralized structure such as a stack). These same techniques allow us to conduct marking concurrently with graph mutations, since doing so requires the reduction process to *cooperate* with the marking process, and can be done by *splicing extra marking activity into the marking tree* to guarantee that none of the invariants on which the marking process depends are invalidated. This feature makes our algorithm fundamentally different from, for example, the distributed algorithms proposed by Chandy and Misra [2, 3, 10], in which the graph is assumed to be static.

The cooperation demanded of the reduction process depends on the particular graph mutations used as well as their interaction. For example, suppose we allow the three primitive mutations shown graphically on the right side of Figure 4-2, which are sufficient to accomplish normal-order graph reduction [6]. The standard argument for why cooperation is needed proceeds as follows:

Suppose we have a graph **a** → **b** → **c**, and the marking process has just spawned a **mark** task from **a** to **b**. Next a series of mutations occur, connecting **a** to **c** (via **add-reference(a,b,c)**) and disconnecting **c** from **b** (via **delete-reference(b,c)**) and thus leaving **b** ← **a** → **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 number of vertices accessible from **c** that need to be marked

as well. A similar argument can be made for why cooperation is needed of the **expand-node** primitive, which splices an arbitrary subgraph (obtained from the free-list) below some vertex.

It is beyond the scope of this paper to discuss the details of the subtle cooperation required of the reduction process (the interested reader may find a suitable discussion in [5, 6]). Suffice it to say that it is sufficient for the reduction process to maintain the following invariants, which we take as axioms:

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

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

The task specifications for the resulting "cooperating mutator primitives" are shown on the left side of Figure 4-2; as with the marking tasks, we assume their execution to atomic. In these specifications the operation **connect(a,b)** adds **b** to **children(a)**, and **disconnect(a,b)** removes **b** from **children(a)**. Note that the **add-reference** primitive is only defined for three *adjacent* vertices. Also, given a vertex **v** in the computation graph, and an arbitrary subgraph **g** taken from the free list, **splice-in-subgraph(v,g)** has the effect of splicing in **g** below vertex **v**. That is, the elements of **children(v)** become references to vertices in **g**, and some vertices in **g** are made to point to elements of the original **children(v)**.

```
task-procedure delete-reference(a,b);
      disconnect(a,b);

task-procedure add-reference(a,b,c);
% where b∈children(a) and c∈children(b)
begin if transient(a) and unmarked(b)
      then << spawn mark1(c,a);
              increment(mt-cnt(a)) >>
      else if marked(a) and transient(b)
           then << execute mark1(c,b);
                   increment(mt-cnt(b)) >>;
      connect(a,c)
end;

task-procedure expand-node(a,g);
begin if marked(a) then mark(g)
      else unmark(g);
      splice-in-subgraph(a,g);
      if transient(a)
      then for each x ∈ children(a)
           do << spawn mark1(x,a);
                 increment(mt-cnt(a)) >>
end;
```

5. Using the Algorithm: Dynamic Solutions to Task and Graph Management

Returning now to our original problem, we wish to adapt our basic algorithm to properly identify the vertices defined by Properties 1 through 6. Having one marking process for each property would impose quite an overhead, but fortunately it is possible to define just *two* processes to accomplish our goals, one marking from the root, the other from tasks. Our strategy is to identify the vertices in $\mathcal{R}_v$, $\mathcal{R}_e$, and $\mathcal{R}_r$, by a marking process called $\mathbf{M}_{\mathcal{R}}$, and those in $\mathcal{T}$ by one called $\mathbf{M}_{\mathcal{T}}$.

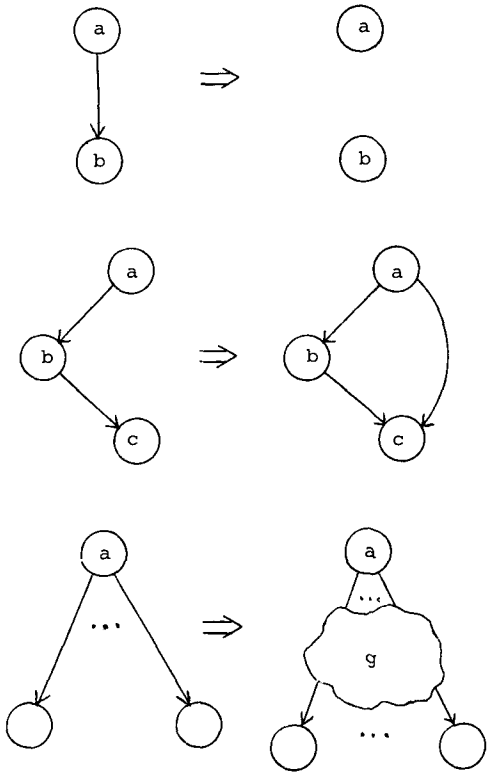

Figure 4-2: Cooperating Mutator Primitives

5.1. Marking From the Root: $M_{\mathcal{R}}$

As in the basic algorithm described above, the process $M_{\mathcal{R}}$ marks from the root; however we need to distinguish the paths through which a node is traced. Our strategy is to augment each node with a **prior**-ity field to indicate whether it is an element of $\mathcal{R}_v$, $\mathcal{R}_e$, or $\mathcal{R}_r$; we use the integers 3, 2, and 1, respectively, for this purpose.

Figure 5-1 shows the new mark task (**mark2**); the return task is identical to **return1** defined earlier. The resulting algorithm is primarily the same as before, but note that if a vertex **x** is being marked with priority **n**, then a new mark task spawned on a child **c** should carry priority **min(n,request-type(c,x))**. For example, if **x** has been marked with priority 2 (meaning that its value was "eagerly" requested) then a child **c**∈req-args$_v$(**x**) should also be marked with priority 2, even though *relative to* **x** its value was "vitally" requested. Furthermore, if a vertex **x** has been marked with priority **n**, and subsequently an attempt is made to mark it with priority **m** > **n**, then the higher priority should prevail. Implementing this requires re-marking **x** (as well as certain of its children) with the higher priority. The specifications shown for **mark2** handle both of these cases.

```
task-procedure mark2(v,par,prior);
    if unmarked(v)
    then modify(v,par,prior)
    else if prior ≤ prior(v)
         then spawn return1(par)
         else << if transient(v) then
                    spawn return1(mt-par(v));
                 modify(v,par,prior) >>;

procedure modify(v,par,prior);
    begin touch(v);
          mt-par(v) := par;
          prior(v) := prior;
          for each c ∈ args(v)
          do << spawn mark2(c,v,
                    min(prior,request-type(c,v)));
                increment(mt-cnt(v)) >>;
          if mt-cnt(v) = 0
          then << mark(v);
                  spawn return1(par) >>
    end;

function request-type(c,v);
    if c ∈ req-args_v(v) then 3
    else if c ∈ req-args_e(v) then 2
         else 1;
```

Figure 5-1: New Mark Task for Process $M_{\mathcal{R}}$

As before, the mt-cnt of all vertices is initially zero. The overall marking process $M_{\mathcal{R}}$ is shown in Figure 5-2. It simply spawns a **mark2** task from the root and waits for **done** to become true. Note that the initial priority is 3, since we assume that the value of the root is essential to the overall computation.

```
procedure M_R();
    begin spawn mark2(root,rootpar,3);
          wait until done
    end;
```

Figure 5-2: Task Marking Process $M_{\mathcal{T}}$

5.2. Marking From Tasks: $M_{\mathcal{T}}$

In order to find the vertices in $\mathcal{T}$, it is necessary to have a handle on the set of all tasks, which so far we have left unspecified. Suppose there are n PE's, and each maintains a list **taskpool(i)** of all reduction tasks whose destination resides on that PE (not an unreasonable assumption). For convenience we then define a special auxiliary vertex **taskroot**$_i$ on each PE such that args(**taskroot**$_i$) = { **v** | **v** is the source or destination of some task in taskpool(i) }. Then define one more vertex, **troot**, such that args(**troot**) = {**taskroot**$_i$, i=1,...,n}. Note that all vertices reachable from a task are now reachable from **troot**.

In accordance with the definition of $\mathcal{T}$, we then define **mark3** as shown in Figure 5-3. Note that **mark3** is identical to **mark1** except that vertices are traced through requested(**v**) ∪ (args(**v**) − req-args(**v**)). The overall marking algorithm $M_{\mathcal{T}}$ simply spawns a **mark3** task on **troot**, which has the effect of marking all vertices reachable from the tasks. (We assume that rootpar, done, mt-cnt, mt-par, and the marking bits used by $M_{\mathcal{T}}$ are distinct from those used by $M_{\mathcal{R}}$).

It should be pointed out that in this simplistic model we have not taken into account the fact that tasks may be in transit between PE's. A solution to this problem is presented in [5].

```
procedure M_T();
    begin spawn mark3(troot,rootpar);
          wait until done
    end;

task-procedure mark3(v,par);
    if unmarked(v)
    then begin touch(v);
               mt-par(v) := par;
               for each c ∈ C
               do << spawn mark3(c,v);
                     increment(mt-cnt(v)) >>;
               if mt-cnt(v) = 0
               then << mark(v);
                       spawn return1(par) >>
         end
    else spawn return1(par);

where C = requested(v) ∪ (args(v) − req-args(v))
```

Figure 5-3: Task Marking Process $\mathbf{M}_{\mathcal{T}}$

5.3. Cooperation

One can specify the cooperation required of the reduction process by ensuring that each primitive mutation does not interfere with the goals of $\mathbf{M}_{\mathcal{R}}$ and $\mathbf{M}_{\mathcal{T}}$. This was done in Section 4.2 for the **delete-reference**, **add-reference**, and **expand-node** primitives when executing with the simplified marking process presented there. The situation is no different with $\mathbf{M}_{\mathcal{R}}$ and $\mathbf{M}_{\mathcal{T}}$, where the primitive mutations must be identified and collectively designed to ensure non-interference. However, the situation with $\mathbf{M}_{\mathcal{R}}$ has been complicated somewhat, because a vertex may be marked with one of three different priorities. If the priority changes while marking a vertex **v** (i.e., while **v** is transient), one may choose to try re-marking the descendants of **v**, or simply to wait until the next marking cycle. By choosing the latter option the specifications for the three primitives mentioned earlier do not change much; the details are left to the reader.

Identifying the actual primitive mutations used requires knowing the details of the reduction process, which we have avoided giving. Furthermore, the mutations of interest vary; each mutation need only cooperate with the marking process(es) that it could affect. For example, a mutation that changes an element of requested(**v**) need not cooperate with $\mathbf{M}_{\mathcal{R}}$, since $\mathbf{M}_{\mathcal{R}}$ only traces through args(**v**). We will provide formal specifications in the next section of the necessary invariants needed for correct behavior of $\mathbf{M}_{\mathcal{R}}$ and $\mathbf{M}_{\mathcal{T}}$.

5.4. Correctness

In this section we present informal proofs for the correctness of our algorithms with respect to Properties 1-6.

5.4.1. Preliminaries

The sets of vertices marked by $\mathbf{M}_{\mathcal{R}}$ and $\mathbf{M}_{\mathcal{T}}$ are defined by:

1. $\mathcal{R}' \equiv \{ \mathbf{v} \mid \text{marked}_{\mathcal{R}}(\mathbf{v}) \}$
2. $\mathcal{R}'_v \equiv \{ \mathbf{v} \mid \text{marked}_{\mathcal{R}}(\mathbf{v}) \wedge \text{prior}(\mathbf{v}) = 3 \}$
3. $\mathcal{R}'_e \equiv \{ \mathbf{v} \mid \text{marked}_{\mathcal{R}}(\mathbf{v}) \wedge \text{prior}(\mathbf{v}) = 2 \}$
4. $\mathcal{R}'_r \equiv \{ \mathbf{v} \mid \text{marked}_{\mathcal{R}}(\mathbf{v}) \wedge \text{prior}(\mathbf{v}) = 1 \}$
5. $\mathcal{T}' \equiv \{ \mathbf{v} \mid \text{marked}_{\mathcal{T}}(\mathbf{v}) \}$

These sets, as well as others that we have defined, are not fixed; they change as the computation progresses. For a time-dependent set **S** we write **S**(t) for its value at time t. This implies, for example, that $\mathcal{GAR}(t) = \mathcal{V} - \mathcal{R}(t) - \mathcal{F}(t)$. To properly detect deadlocked vertices it is necessary that $\mathbf{M}_{\mathcal{T}}$ execute before $\mathbf{M}_{\mathcal{R}}$ (the reason for this will become more apparent in the proof of Theorem 2). Define t_a, t_b, and t_c to be the times when (a) $\mathbf{M}_{\mathcal{T}}$ begins, (b) $\mathbf{M}_{\mathcal{T}}$ ends and $\mathbf{M}_{\mathcal{R}}$ begins, and (c) $\mathbf{M}_{\mathcal{R}}$ ends, respectively.

To prove proper behavior of $\mathbf{M}_{\mathcal{R}}$ and $\mathbf{M}_{\mathcal{T}}$ we must state some axioms that capture the semantics of the evaluation during the marking phase. For $t_2 \geq t_1$ these **reduction axioms** are:

1. $\mathcal{R}(t_2) \subseteq \mathcal{R}(t_1) \cup \mathcal{F}(t_1)$
2. $\mathcal{T}(t_2) \subseteq \mathcal{T}(t_1) \cup \mathcal{F}(t_1)$
3. $\mathcal{GAR}(t_1) \subseteq \mathcal{GAR}(t_2)$
4. $\mathcal{DL}_v(t_1) \subseteq \mathcal{DL}_v(t_2)$

The first two axioms are statements of the relationship between $\mathcal{F}$ and the reduction process; that is, $\mathcal{R}$ and $\mathcal{T}$ expand only by acquiring nodes from $\mathcal{F}$. Axioms 3 and 4 ensure the integrity of garbage and deadlocked nodes; that is, once a vertex is garbage or deadlocked, it remains that way. From this we can conclude that $\mathcal{F}$ is strictly decreasing.

We also need to say something about the behavior of the various task types, especially vital tasks, whose results are needed for proper execution. The next two axioms capture the behavior that we desire. Let $r = \text{req-args}_v(\mathbf{x})$, $s = \text{req-args}_v(\mathbf{y})$, and $t_2 \geq t_1$:

5. $\mathbf{x} \in s(t_1) \wedge \mathbf{x} \notin \mathcal{T}(t_1) \Rightarrow \mathbf{x} \in s(t_2)$
6. $\mathbf{x} \in s(t_1) \Rightarrow (r(t_2) \neq \emptyset \Rightarrow \mathbf{x} \in s(t_2))$

Axiom 5 amounts to saying that if the value of a "vitally requested" vertex is never computed, then it remains vitally requested forever. Axiom 6 further states that a vertex cannot return a value unless it receives values from all nodes on whose value it depends.

We also need a set of invariants that the reduction process must maintain to guarantee the absence of interference with the marking process. Given a mark task **t** = **mark(c,v)**, let returned(**t**) denote that the return task eventually spawned as a result of executing **t** has executed. Then the following **marking invariants** are valid for both $\mathbf{M}_{\mathcal{R}}$ and $\mathbf{M}_{\mathcal{T}}$:

1. transient(**v**) ⇒
   ∀**c**∈children(**v**), ∃mark(**c**,**w**)
2. marked(**v**) ⇒
   ¬(∃**w**∈children(**v**) ∧ unmarked(**w**))
3. mt-cnt(**v**) = |{task | task = mark(**c**,**v**) ∧
   c∈children(**v**) ∧ ¬returned(task) }|

These invariants are simply formal versions of the ones mentioned in Section 4.2; indeed it is not hard to show that they are satisfied by the primitive graph mutations of Figure 4-2. Together with the reduction axioms they form a set of reasonable constraints on the reduction process.

Lemma 1: ($\mathbf{M}_{\mathcal{R}}$ safety)

$\mathcal{GAR}(t_b) \cap \mathcal{R}'(t) = \emptyset$, $t \geq t_c$.
That is, $\mathbf{M}_{\mathcal{R}}$ does not mark any vertex that is garbage prior to the start of marking.

Proof: By inspection of **mark2**, the only vertices that $\mathbf{M}_{\mathcal{R}}$ marks are those found starting at the root and tracing through a vertex's **args** set. At time t_b no element of $\mathcal{GAR}$ is reachable in this way (since $\mathcal{GAR} = \mathcal{V} - \mathcal{R} - \mathcal{F}$). Furthermore, by reduction axiom 1, new vertices added to $\mathcal{R}$ are obtained only from $\mathcal{F}$, which is disjoint from $\mathcal{GAR}$. Thus $\mathbf{M}_{\mathcal{R}}$ cannot mark any vertices in $\mathcal{GAR}(t_b)$, establishing the lemma at time t_c. Since axiom 2 also applies after t_c, no cooperating mutator primitive can mark nodes in $\mathcal{GAR}(t_b)$ either, establishing the lemma. □

Corollary: $\mathcal{R}'(t) \subseteq \mathcal{R}(t_b) \cup \mathcal{F}(t_b)$, $t \geq t_c$.

Lemma 2: ($\mathbf{M}_{\mathcal{R}}$ liveness)

$\mathcal{R}(t) \subseteq \mathcal{R}'(t)$, $t \geq t_c$.
That is, after marking is complete, all vertices reachable from the root are marked.

Proof: First define M to be the set of vertices comprising the marking-tree, and M_i to be those nodes in M at depth i; that is:

- M_0 = { root }
- M_i = { v | mt-par(v)=w for w∈M_{i-1} }, i≥1.
- $M = M_0 \cup M_1 \cup \ldots \cup M_\infty$.

The proof then proceeds as follows:

1. If $t \geq t_c$ then **done** must be true, mt-cnt(**root**) = 0, and the root is marked. So (∀**v**∈M_0), marked(**v**) and returned(**v**).
2. If each vertex **v** in M_i is marked, then mt-cnt(**v**) = 0 and by marking invariant 2 each vertex **w** in M_{i+1} must have spawned a **return1** task on its parent, and so must also be marked.
3. Using (1) as a basis and (2) as an induction rule, it then follows that (∀**v**∈M) marked(**v**).
4. Now consider **c** ∈ children(**root**). If **c** ∈ M, then by (3) it must be marked. If it is not in M, then it cannot be unmarked (because of marking invariant 3), nor can it be transient (since that would imply it is in M). Thus it must be marked.
5. From (4) a simple inductive argument follows, proving that all elements of $\mathcal{R}$ are marked. □

Corollary: $\mathcal{GAR}'(t) \subseteq \mathcal{GAR}(t)$, $t \geq t_c$,
where $\mathcal{GAR}'(t) = \mathcal{V} - \mathcal{R}'(t) - \mathcal{F}(t)$.

Lemma 3: ($\mathbf{M}_{\mathcal{T}}$ safety)

$(\mathcal{V} - \mathcal{T}(t_a) - \mathcal{F}(t_a)) \cap \mathcal{T}'(t) = \emptyset$, $t \geq t_b$.
That is, no vertices not reachable from some task prior to marking are marked by $\mathbf{M}_{\mathcal{T}}$ unless they are new nodes from the free list.

Proof: By an argument similar to that of Lemma 1. □

Corollary: $\mathcal{T}'(t) \subseteq \mathcal{T}(t_a) \cup \mathcal{F}(t_a)$, $t \geq t_b$

Lemma 4: ($\mathbf{M}_{\mathcal{T}}$ liveness)

$\mathcal{T}(t) \subseteq \mathcal{T}'(t)$, $t \geq t_b$.
That is, after marking is complete, all vertices reachable from some task are marked.

Proof: By an argument similar to that of Lemma 2. □

5.4.2. Identifying Garbage and Deadlocked Vertices

From the previous lemmas we can now prove our main results:

Theorem 1:

$\mathcal{GAR}(t_b) \subseteq \mathcal{GAR}'(t) \subseteq \mathcal{GAR}(t)$, $t \geq t_c$,
where $\mathcal{GAR}'(t) = \mathcal{V} - \mathcal{R}'(t) - \mathcal{F}(t)$.

which says that all of the vertices that are garbage prior to marking are found by $\mathbf{M}_{\mathcal{R}}$, and no vertices are erroneously identified as garbage.

Proof: (a) By reduction axiom 3 $\mathcal{GAR}(t_b) \subseteq \mathcal{GAR}(t)$, or $\mathcal{R}(t) \cup \mathcal{F}(t) \subseteq \mathcal{R}(t_b) \cup \mathcal{F}(t_b)$ which implies $\mathcal{F}(t) \subseteq \mathcal{R}(t_b) \cup \mathcal{F}(t_b)$. Furthermore, by the corollary to Lemma 1 we know $\mathcal{R}'(t) \subseteq \mathcal{R}(t_b) \cup \mathcal{F}(t_b)$. Therefore $\mathcal{R}'(t) \cup \mathcal{F}(t) \subseteq \mathcal{R}(t_b) \cup \mathcal{F}(t_b)$, which implies $\mathcal{V} - \mathcal{R}(t_b) - \mathcal{F}(t_b) \subseteq \mathcal{V} - \mathcal{R}'(t) - \mathcal{F}(t)$, thus establishing the left-hand containment of the theorem. (b) The right-hand containment is established directly by the corollary to Lemma 2. □

Although the next theorem looks similar to Theorem 1, it implications (and consequently its proof) are quite different:

Theorem 2:

$\mathcal{DL}_v(t_a) \subseteq \mathcal{DL}'_v(t) \subseteq \mathcal{DL}_v(t)$, $t \geq t_c$,
where $\mathcal{DL}'_v(t) = \mathcal{R}'_v(t) - \mathcal{T}'(t)$.

which says that all deadlocked nodes prior to marking are found by the marking process, and none are erroneously categorized as being deadlocked.

Proof: (a) The left-hand containment: Consider $v \in \mathcal{DL}_v(t_a) = \mathcal{R}_v(t_a) - \mathcal{T}(t_a)$. Clearly $v \notin \mathcal{T}(t_a)$ and $v \notin \mathcal{F}(t_a)$. Then by Lemma 3 $v \notin \mathcal{T}'(t)$. To show that $v \in \mathcal{R}'_v(t)$, consider a sequence of vertices at time t_a from the root to v linked through req-args$_v$ (which must exist since $v \in \mathcal{DL}_v(t_a)$). By reduction axioms 5 and 6 this sequence must remain intact throughout marking (since $v \notin \mathcal{T}(t_a)$), and so clearly v will be "vitally marked." (b) The right-hand containment: Consider $v \in \mathcal{DL}'_v(t)$; we want to show that $v \in \mathcal{R}_v(t) - \mathcal{T}(t)$. Since $v \notin \mathcal{T}'(t)$, by Lemma 4 we know $v \notin \mathcal{T}(t)$. To show $v \in \mathcal{R}_v(t)$, note first that $v \notin \mathcal{T}(t_b)$. At the moment v is subsequently vitally marked by $\mathbf{M}_{\mathcal{R}}$, there must exist a vertex w such that $v \in$ req-args$_v(w)$ and w is also vitally marked. Then working backwards using reduction axioms 5 and 6, there must exist a sequence of nodes from the root to v linked through req-args$_v$, and this sequence must persist. Therefore $v \in \mathcal{R}_v(t)$. Note that this part of the proof is the only part that requires $\mathbf{M}_{\mathcal{T}}$ to execute before $\mathbf{M}_{\mathcal{R}}$. □

5.4.3. Identifying Task Types

Identifying irrelevant tasks may be stated as a corollary to Theorem 1:

Corollary 1:

$(\mathcal{IRR}(t_a) \cap \mathcal{IRR}(t)) \subseteq \mathcal{IRR}'(t) \subseteq \mathcal{IRR}(t)$,
$t \geq t_b$.

Proof: Directly from Theorem 1 and the definition of $\mathcal{IRR}$; but note that irrelevant tasks before marking might not exist after marking, and therefore a restriction is made on the left-hand containment. □

Although identifying irrelevant tasks is straightforward, it is difficult to state appropriate theorems that characterize the correctness of $\mathbf{M}_{\mathcal{R}}$ and $\mathbf{M}_{\mathcal{T}}$ in identifying the other task types. The reason for this is that vertices may pass freely between the sets $\mathcal{R}_v$, $\mathcal{R}_e$, and $\mathcal{R}_r$ as the evaluation unfolds! We *can* say that if a vertex is *continuously* reachable through n paths, then it will be marked by whichever path results in the highest priority. This result is trivial, and will not be stated formally.

6. Remarks

It should be obvious that marking process $\mathbf{M}_{\mathcal{T}}$ is only used to find deadlocked vertices; all other chores are handled by $\mathbf{M}_{\mathcal{R}}$. In a system where deadlock is of no concern, $\mathbf{M}_{\mathcal{T}}$ may be eliminated altogether. In the systems that concern us, deadlock is a possibility, but it is not imperative to discover it quickly (indeed, a deadlocked system generally does no harm, it just never does any good!). Since $\mathbf{M}_{\mathcal{T}}$ must execute before $\mathbf{M}_{\mathcal{R}}$ to properly detect deadlocked nodes (thus reducing the throughput of the overall process), our approach is to execute $\mathbf{M}_{\mathcal{T}}$ only occasionally.

Although we have not discussed the mechanisms by which the tasks are made atomic, we can at least discuss the contention for vertices that might exist between $\mathbf{M}_{\mathcal{R}}$, $\mathbf{M}_{\mathcal{T}}$, and the reduction process. The key point to be made is that none of the marking tasks ever nest the locking of vertices, and none need to leave a vertex locked while marking is continued on some other PE (this precludes the possibility of a resource deadlock between marking tasks competing for the same vertices). The marking processes' interference with the reduction process is thus minimal. In fact, since the marking tasks can be made to execute in a bounded amount of time once the required vertices are accessed, it is conceivable to design a system in which the reduction task observes a simple busy-waiting protocol to gain access to a vertex locked by a marking task.

The algorithms as presented incur a high space overhead, in that each vertex requires space for mt-cnt, mt-par, and marking bits, and each PE requires room for a task-pool. In [6] we discuss techniques to reduce this space drastically. In particular, it is possible to combine all of the mt-cnt's and mt-par's into just two words on each PE. In the fine-grained systems that concern us, this is an important optimization, although in systems where the object granularity is large, the current approach may be acceptable.

7. Summary

We have described a model of distributed graph reduction in which a computation graph is partitioned and distributed among an arbitrary number of autonomous processing elements having only local store and communicating by spawning tasks among them. Within this model several aspects of distributed graph reduction have been identified, in particular the dynamic characteristics of the reduction process and how it affects the status of both vertices and tasks. We have further shown how a novel decentralized parallel graph-marking algorithm can be used as an effective mechanism to manage the tasks and graph proper. Although our solutions are couched within distributed graph reduction, we feel that they are applicable to a broader class of distributed computing models.

8. Acknowledgements

Thanks to Mike Fischer and Jerry Leichter for valuable comments on an earlier draft of the manuscript, and to the Yale Department of Computer Science for providing the excellent environment in which to nurture my research.

References

[1] Baker, H.G. Jr.
List processing in real time on a serial computer.
CACM 21(4):280-294, April, 1978.

[2] Chandy, K.M., Misra, J.
Distributed computation on graphs: shortest path algorithms.
Commun. ACM 25(11):833-837, November, 1982.

[3] Chandy, K.M, Haas, L.M., and Misra, J.
Distributed Deadlock Detection.
ACM Trans. Prog. Lang. Sys. 1(2):144-156, May, 1983.

[4] 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.

[5] Hudak, P. and Keller, R.M.
Garbage collection and task deletion in distributed applicative processing systems.
In Park et al. (editors), *Sym. on Lisp and Functional Prog.*, pages 168-178. ACM, August, 1982.

[6] Hudak, P.
Distributed Graph Marking.
Research Report 268, Yale University, January, 1983.

[7] Jhon, C.S., Keller, R.
Analysis of unbounded token-flow graphs and realization of unbounded token-flow graph by bounded token-flow graphs.
AMPS Technical Memorandum 8, University of Utah, April, 1982.

[8] Kennaway, J.R., and Sleep, M.R.
Expressions as Processes.
In *Sym. on Lisp and Functional Prog.*, pages 21-28. ACM, August, 1982.

[9] Liskov, B. and Scheifler, R.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
In *9th Sym. on Prin. of Prog. Lan.*, pages 7-19. ACM, January, 1982.

[10] Misra, J., Chandy, K.M.
A distributed graph algorithm: knot detection.
ACM Trans. Program. Lang. Syst. 4(4):678-686, October, 1982.

[11] Steele, G.L. Jr.
Multiprocessing compactifying garbage collection.
CACM 18(9):491-500, September, 1975.

[12] Turner, D.A.
A new implementation technique for applicative languages.
Software-Practice and Experience 9:31-49, 1979.

[13] Wadge, W.W.
An extensional treatment of dataflow deadlock.
In Kahn, G. (editor), *Semantics of Concurrent Computation*, pages 285-299. Springer-Verlag, 1979.