Graph Structure Management in Parallel Symbolic Systems*

Michael Fisher, Robert Johnson
Department of Computing
Manchester Metropolitan University
Manchester M1 5GD, U.K.
EMAIL: {M.Fisher,R.Johnson}@doc.mmu.ac.uk

John Keane
Department of Computation
UMIST
Manchester M60 1QD, U.K.
EMAIL: jak@co.umist.ac.uk

Abstract

We discuss the efficient exploitation of parallelism in systems where a common data structure is accessed, using message-passing, by a set of parallel processes. In particular, we provide a framework in which such systems can be abstractly represented and present a range of definitions in terms of constraints upon the available parallelism. In addition to considering a particular class of problems (based on common graph structures), we study purely *control* parallelism. Within these restrictions, we consider applications in automated deduction and garbage collection, and contrast the constraints on parallel activity within such systems.

1 Introduction

In this paper we study the general problem of maintaining the integrity of a shared data-structure, which is being manipulated by a set of parallel processes. In particular, we discuss the conflict between maximal exploitation of parallelism and the requirement that the data-structure remain consistent across all its copies. We restrict ourselves to the case where the shared data-structure is controlled solely by a single process containing that structure. In such a system, parallel processes that read parts of the data-structure, and carry out some computation dependent upon those parts, may find that the structures they hold are out-of-date with respect to the centrally shared structure. This is termed the problem of *coherence* of a structure across a parallel system.

A high level of coherence (termed *strong coherence*) can be achieved if any process that requests a portion of the data structure is notified of any changes to that portion as soon as they happen. Unfortunately, although such a high level of coherence will ensure that no computations are carried out based upon incorrect or out-of-date information, this may severely restrict possible parallelism. Additionally, if a data structure is widely distributed, a strongly coherent approach may incur unacceptable data transmission overheads as a result of the regular updates that are required.

A very low level of coherence (termed *weak coherence*) can be obtained by processes only checking that the portions of the data structure that they initially requested are still valid at the end of their local computation. If they find that these structures have been altered, then they must recompute their results based upon the new information. While such a scheme allows a high level of potential (independent) parallelism, it may be inefficient, especially when the shared structure is frequently modified. Consequently, there is a trade-off between parallelism and coherence that affects overall efficiency.

We restrict our discussion to data structures that are directed graphs. Many fundamental problems are based upon a basic representation in terms of directed graphs. The solutions not only involve searching the graphs for particular paths, but often incorporate the construction of the graphs themselves. Our arguments may equally be applied to simpler data structures, such as DAGs, trees, etc. However, it is the manipulation of directed graphs that appears to present the most difficult problems, and so we consider such structures as the 'worst cases'.

2 Abstract Model

Our model is a set of parallel processes communicating via message-passing. Within this set of processes, certain ones contain (and maintain) data structures that are required and/or manipulated by other processes. As the interface to such processes is through serialised message-passing, these processes can provide serious bottlenecks in the system, especially if they are accessed regularly. Although, in general, there may be more than one shared data-structure within a set of processes, we will restrict our study to the problem of one shared data-structure within a system. Given such a system, other processes can be defined, for example:

- processes that read part of the data-structure and use this as the basis of some computation;
- processes that attempt to add new element(s) to the structure;
- processes that attempt to delete single elements of the data-structure;

*Work partially supported by UK SERC grant GR/J48979.

- processes that attempt to delete larger sections of the data-structure (sometimes all of it!).

In order to define our model more formally, we will consider the basic graph structures, process types and communication mechanisms in the following sections.

2.1 Graph Structures

A graph is a tuple $\langle V, E, L\rangle$, where V is a set of vertices, E is a set of edges of the form $V \times V$, and L is a mapping from vertices to strings. A directed graph (digraph) is a graph where each edge is an ordered pair. We treat vertices as atomic entities, but may associate a label with a vertex. Initially, we consider the label to be simply a string, but in particular example systems (see §4), we label the vertices with more complex data. Edges are unlabelled. Graph structures need not be connected, i.e., the graph may contain isolated subgraphs.

2.2 Processes

The systems we consider consist of a set of asynchronously executing parallel processes, each of which communicates through message-passing using a simple protocol (see below). These processes are self contained (i.e., we can consider them as 'objects') in that they have access to no storage except their own. However, they can request data from other processes.

2.3 Communication

As stated above, the general model we use is based upon a set of processes that wish to manipulate a common graph structure, but with the actual structure being under the control of a single *manager* process. Processes may interrogate or affect this structure by sending messages to the manager process. To simplify the discussion, we consider only the following messages that can be sent to the manager process.

read_graph() — returns the graph in its entirety.

read_subgraph($[n_1, \ldots, n_j]$) — returns a portion of the graph containing all vertices that are reachable from any of n_1 to n_j, together with any edges between these vertices.

create_vertices(*num*) — returns a set of *num* new vertices, and adds them to the graph.

create_edges($[\langle n_1, n'_1\rangle, \ldots, \langle n_j, n'_j\rangle]$) — adds a set of edges to the graph.

create_labels($[n_1 \to str_1, \ldots, n_j \to str_j]$) — adds labels to vertices n_1 to n_j.

delete_vertices($[n_1, \ldots, n_j]$) — deletes n_1 to n_j (and any edges between them) from graph.

delete_edges($[\langle n_1, n'_1\rangle, \ldots, \langle n_j, n'_j\rangle]$) — deletes a set of edges.

N.B., no *delete_labels*(...) message is required as a vertex can simply be labelled with an empty string if a null label is required.

The above messages, sent to the manager process, can be considered as atomic from the perspective of the other parallel processes.

2.4 Process Types

Given the general protocol based upon the restricted set of messages that can be sent to the manager process, we can define a range of prototypical processes that manipulate the graph structure. Each of these has a simple abstract description yet is used, in a more concrete form, within the applications described in §4. We outline below the particular processes that are required within our analysis:

`manager` — a process that contains the central representation of the graph data structure, controls access to it and orchestrates the requests for change within the graph.

`constructor` — a type of process that, given certain vertices from the graph, will attempt to construct additional vertices and label them. We can subcategorise these processes into those that only add one new vertex ($\texttt{constructor}_1$) and those that add n vertices, where $n > 1$ ($\texttt{constructor}_n$).

`linker` — a type of process similar to a `constructor`, but which adds new edges rather than vertices. Again, these can be categorised into $\texttt{linker}_1$ and $\texttt{linker}_n$.

`checker` — a type of process that simply examines a set of vertices to check whether or not they satisfy certain criteria. Again, $\texttt{checker}_1$ just attempts to recognise a single vertex, while $\texttt{checker}_n$ attempts to recognise a larger sub-graph.

`reducer` — a type of process that removes a sub-graph from the common graph structure (it does this by sending the appropriate message to the manager). Again, $\texttt{reducer}_1$ deletes a single vertex and its associated edges, while $\texttt{reducer}_n$ deletes a larger sub-graph.

While this list is not meant to be exhaustive, it provides the basic processes that will be discussed in later sections. Each of these processes is, in turn, built up from operations defined in §2.3.

Having defined the abstract model, we now consider the problem of ensuring the consistency of the shared graph structure, whilst maintaining a reasonable level of independent parallel activity.

3 Constraints on Graph Manipulation

The definitions we provide here will be formalised in terms of temporal logic [2]. This provides a simple, yet expressive, notation for characterising constraints upon the dynamic message-passing activity of the system. The simple fragment of temporal logic we utilise consists of classical first-order logic extended with three temporal operators: $\bigcirc$, $\Diamond$, and $\Box$. Here we consider time to consist of a discrete series of moments and so the meaning of these operators can be informally stated as follows:

- $\bigcirc A$ is true now if A is true at the *next* moment in time.
- $\Diamond A$ is true now if A is true at *some* moment in the future.
- $\Box A$ is true now if A is true at *all* future moments.

We first consider communication properties related to message-passing between objects. Note that, for simplicity, we categorise all messages as either *read* or *change* requests and introduce the predicate *holds* where $holds(i, G)$ is true if G is the graph structure currently held within the process i. Note also that M is the particular graph manager process, $read(i)$ represents a read request by process i, and $change(i, c)$ represents a request by process i to carry out change c on the graph.

Definition 1 [Read Consistency] *Whenever a read is requested, then by the next moment in time the correct graph is supplied to the sender.*

$$\forall i.\ \forall g.\ [read(i) \wedge holds(M, g)] \Rightarrow \bigcirc holds(i, g).$$

Definition 2 [Change Message Recording] *If a change is requested then the change will eventually be recorded within the manager process.*

$$\forall i.\ \forall c.\ change(i, c) \Rightarrow \Diamond change(M, c).$$

In both these definitions, we are assuming that no significant delays occur during message-passing. If we wish to model such delays, the communication model can easily be extended in this way. Also, *change* requests are sent to the manager process, but we leave undefined the way in which the changes are implemented as this is a matter for the internal consistency algorithm of the manager process. In addition, the action of the manager recording a change does not necessarily imply that the change is made in the graph, since particular changes may be prevented by some preceding change requests.

Definition 3 [Weak Coherence] *If a change is made to the central graph structure, then any process will see the change at its next read.*

Note that, for weak coherence, no additional defining formula is required as Definition 1 is strong enough to enforce weak coherence.

Definition 4 [Strong Coherence] *If a change is made to the central graph structure, then every process will see the change at the next moment in time.*

$$\forall c.\ (change(M, c) \wedge holds(M, c)) \Rightarrow (\forall i. \bigcirc\ holds(i, c)).$$

Here, '$change(M, c) \wedge holds(M, c)$' represents a change that has been requested *and* acted upon.

In our model, we define strong coherence of a graph to indicate that processes that have a copy of the graph structure will always see the most up-to-date state. Weak coherency indicates that processes will have a snapshot that *may not* correspond to the most recent updates of the graph. Thus, each process may potentially see a different version of the graph. At various points, *coherency establishing* operations (i.e the *read* operation) will be performed, when each copy of the graph is made consistent. Between each of these synchronisation points each process proceeds to work on its version of the graph.

Strong coherence restrains parallelism but ensures that no process ever sees an out-of-date state. Weakly coherent models allow more liberated parallel activity at the expense of possibly out-of-date views.

Definition 5 [Weak Fairness] *If a process requests a change infinitely often, then eventually that request will be acted upon by the manager process.*

$$\forall i.\ \forall c.\ \Box\Diamond change(i, c) \Rightarrow \Diamond holds(M, c).$$

This final definition is important in avoiding livelock (see later) and ensures that a request, if continually repeated, will eventually be acted upon.

4 Applications

This section shows how the trade-off between coherency and parallelism, as defined in §3, occurs in practical situations. In particular, we utilise the process categorisation described in §2 in studying applications from theorem-proving and garbage-collection, both of which conform to the model of a shared graph structure being manipulated by parallel processes.

4.1 Parallel Temporal Theorem-Proving

CACTTuS (Communicating Agents for Concurrent Temporal Tableaux Systems) [3] is a parallel automated theorem prover for propositional temporal logic (PTL) [2]. It has been constructed using the basic process object types given in §2.4 and effectively consists of the expansion and reduction of a graph structure by a set of parallel processes.

4.1.1 The Soundness, Completeness and Coherency of the PTL Tableau Method

The procedure for deciding satisfiability for a propositional temporal formula is to construct a graph structure representing all possible models of the formula and then to delete vertices under certain conditions. If the whole graph is deleted, then the formula has no models and, hence, is unsatisfiable.

Wolper's original adaptation [5] of the tableau method for temporal logic suggests that a directed graph may be constructed by applying graph generation rules to the formulae labelling each vertex to derive its successor vertex/vertices. This process continues until no leaf vertex is labelled with any unused formulae, at which point graph reduction may commence. For a vertex to be removed from the graph it must either have had all successor vertices removed, be labelled by contradictory formulae, or must satisfy a particular complex condition on the temporal formulae labelling both the vertex and it's successors[1].

A weakly coherent view is more likely to affect completeness than soundness in the case of theorem proving. For instance, if a process' occupation is the deleting of childless vertices from a graph, and that process is aware that a vertex has successors but it is not made aware of their deletion by another process, then completeness may be lost. It is more usual, however, for the described weak coherency problem to simply delay the operation since the process' view of the graph would most likely be updated with the requisite information eventually. A more probable case is the loss of completeness as a result of livelock. If a vertex is repeatedly added by a constructor and removed by a reducer then there is a possibility that the computation will not terminate. We avoid this by explicitly enforcing constraints on the permitted graph reduction operations.

4.1.2 Adapting Tableaux For Parallel Computation

In order for us to derive a parallel adaptation of the tableau method for temporal logic, it is first necessary to identify the fundamental processes that constitute the sequential system. The following list of processes, together with their particular process types, reflect the activities undertaken by the sequential theorem prover. Recall that, in the list below, $\texttt{reducer}_1$ reduces the graph by one vertex at a time whereas $\texttt{reducer}_n$ can delete a whole subgraph at a time.

- *GC* — Graph construction:
  `constructor + linker`
- *LR* — Liberalised reduction: $\texttt{reducer}_1$
- *SR* — Strict reduction: $\texttt{reducer}_1$
- *CC* — Cycle checking: `checker`
- *CR* — Cycle reduction: $\texttt{reducer}_n$
- *DC* — Duplicate checking: `checker`
- *IR* — Inconsistency reduction: $\texttt{reducer}_1$

Note that we have decomposed the graph reduction operations into the separate processes *LR, SR, CR* and *IR*. *LR* occurs when no successors may be derived from a vertex, i.e. all formulae in the label are used or atomic, but *GC* has not yet completed. *SR* is as *LR* but post *GC*. *CR* denotes the deletion of vertices failing to satisfy their temporal constraints while *IR* refers to the deletion of vertices containing contradictory formulae. Note also that we do not explicitly refer to a `manager` process in our discussion. We assume that such a process is running continually in order to oversee control of the data structure at all times.

[1]This is not be described here — for more details, see [5].

4.1.3 Livelock

Both livelock and deadlock will affect the completeness of a proof procedure. While livelock tends to result in an infinitely cycling sequence of operations on the data structure, deadlock results in a halting of all operations on the structure since processes mutually suspend. For instance process p_i may suspend pending a write operation from process p_j which has itself suspended awaiting an action from p_i. However, livelock can be identified by analysing the data structure over time, i.e. without considering the internal behavioural motivations of the processes themselves. Therefore, we concern ourselves primarily with the problem of livelock.

Livelock occurs when processes still continue to execute, but are continuously performing the same cycle of operations. For example, consider a constructor process that adds successor vertices to terminal vertices within the graph, together with a deleter process that deletes terminal vertices. The constructor might add vertex n, and the deleter process might delete it straight away. The constructor process now attempts to find a terminal vertex to expand, finds the same one it chose before and constructs n again. The deleter again removes n, and so on. We consider three possible ways of either avoiding, or coping with this problem and categorise them in terms of their relation to coherency issues.

Explicit Synchronisation (Strong Coherency): Here, we may opt to enforce synchronisation at specific points. For instance, the problem of livelock described above would be avoided given a regime that suspends all reduction processes until construction is completed. The major drawback to this option is that the available parallelism is significantly reduced. In addition the completed graph may exceed the available storage. However, if reduction is permitted to proceed in parallel with construction the maximum size of storage required may be markedly lower.

Lookup Tables (Strong Coherency): If we maintain a table containing the details of all vertices that have ever been produced, it will not be necessary to rebuild portions of the graph that have appeared previously. So, we may impose constraints on the addition of vertices to the graph on the basis that they may only be added if they have never been added previously. While this option avoids livelock and increases parallelism it suffers from an increased demand on storage. Again this is based on strong coherence.

Speculative Parallelism (Strong Coherency): Weak coherency permits us to liberally utilise speculative parallelism. If we allow all processes to continue their computations, and only check for coherency on their termination, it is likely that we will liberate maximal parallelism. The major problem is that if information is out of date a great deal of wasted computation is likely.

At first these approaches appear to offer a direct trade-off between levels of parallelism and coherency. As is evident in the case of lookup tables, the amount of available storage is also a factor. The appropriation of storage to support `checker` operations, in particular, permits further parallelism to be extracted from a strongly coherent approach.

4.2 Garbage Collection

In garbage collection the relaxation of strong coherence can have a very different effect to that experienced in theorem-proving. Previously, we showed that weakening the coherence of an agent's view of a data structure can affect completeness. Here we show how it can also affect the soundness of a system.

Programs in functional languages are often represented as graphs, and their model of computation termed *graph rewriting*. Garbage collection occurs when free space falls below some pre-determined level and involves determining which vertices of the graph are *in-use* and treating all other vertices as garbage, and thus enabling their re-use. The graph of concern here is the in-use vertices, the out-of-use vertices (unconnected components), and a free list of (unused) vertices.

Mark-scan garbage collection works as follows: the graph of in-use vertices is marked, and the scan phase returns any unmarked vertices, i.e. garbage, to some free list. Attempts have been made to allow the graph rewriting process to proceed in parallel as much as is possible with garbage collection, e.g. [1].

4.2.1 Process Types

The graph `rewriter` process involves a number of the process types mentioned in §2.4: `constructor`, `linker` and `reducer`. It may reduce part of the graph (i.e. make some vertices out-of-use) (`reducer`), add new vertices to the graph (`constructor`), or re-write (re-configure) the existing graph, for instance by establishing new arcs to existing vertices (`linker`).

The garbage collector has two separate, sequential phases: `mark` and `scan` respectively.

$$\texttt{garbage-collector} \stackrel{def}{=} \texttt{marker ; scanner}$$

1. In the `marker`, each in-use vertex is marked by following through from all root vertices. Arc reversal occurs when child vertices are rewritten, thus there are potentially any number of root vertices [1].

   The `marker` corresponds to a `checker` process.

2. In the `scanner`, the store is scanned in a linear fashion, all unmarked vertices (i.e. out-of-use free list vertices) are included in a new free list.

   The `scanner` corresponds to a `reducer` process.

4.2.2 Allowing Parallelism

We consider in this section how the `rewriter` and the `garbage-collector` can run in parallel.

A *weakly coherent* approach would be to take a snapshot of the graph at some point in time, mark the in-use vertices and then pass this marked snapshot onto the `scanner`. However, a snapshot is insufficient for the correct operation of garbage collection. In particular, if the `rewriter` creates a new in-use vertex which is allocated from the free list, *after* the snapshot is taken, then the `scanner` will regard this new in-use vertex as garbage since when the snapshot was taken the vertex was part of the free list. Treating in-use vertices as garbage leads to erroneous results.

Thus, we require the graph to be strongly coherent. If we allow the `rewriter` to proceed in parallel with garbage collection on the *same* graph structure, then we have a form of strong coherence, i.e. a single instance of the graph.

Consider the case where the `rewriter` is working on a part of the graph already visited by the `marker`. There are three actions of the `rewriter` which lose strong coherency of the shared graph:

Case 1: The `rewriter` constructs a new vertex (allocated from the free list) to be the child of an already marked vertex.

The `marker` will not visit this new in-use vertex, and thus the `scanner` will treat it as garbage.

Case 2: The `rewriter` reconfigures the graph such that an unmarked (previously in-use) vertex becomes the child of a marked vertex.

The effect of this would be that an in-use vertex (the child) will not be visited by the `marker`, and thus will be treated as garbage by the `scanner`.

Case 3: The `rewriter` reconfigures the graph such that a vertex previously marked as in-use now becomes out-of-use.

The consequence is that not all vertices that are garbage at the *time* the `scanner` executes will be collected.

Cases 1 and 2 allow the possibility of treating in-use vertices as garbage and thus producing erroneous results. Case 3 means garbage collection will be invoked more often. To allow the `rewriter` to proceed in parallel with garbage collection there must be coordination between the processes. To handle Case 1, the `rewriter`, during the execution of either the `marker` or the `scanner`, must mark all new vertices allocated from the free list as in-use. This involves relatively simple cooperation between the `rewriter` and the `marker` and `scanner` respectively. Case 2 is more complicated as, to ensure that all such child vertices are marked, the `rewriter` must spawn a *mark_task* on the child if the following hold [1]:

$$\left[\begin{array}{ccc} state_{new\_parent} = marked & \wedge & state_{new\_child} = unmarked \\ & \vee & \\ state_{new\_parent} = marked & \wedge & state_{old\_parent} = unmarked \end{array}\right]$$
$$\Rightarrow \quad spawn(mark\_task(new\_child))$$

4.2.3 Parallel Processes

The behaviour of the system (to handle Cases 1 and 2) is as follows:

$$\texttt{P = rewriter}_1\texttt{;(rewriter}_2\|\texttt{marker);}$$
$$\texttt{(rewriter}_3\|\texttt{scanner);P}$$

- $\texttt{rewriter}_1$: the normal graph rewriter that does not operate in parallel with any other process.
- $\texttt{rewriter}_2$: $\texttt{rewriter}_1$ extended by cooperating with the `marker`.

  $\texttt{rewriter}_2$ deals with both Cases 1 and 2.
- $\texttt{rewriter}_3$: $\texttt{rewriter}_1$, extended by cooperating with the `scanner`.

  $\texttt{rewriter}_3$ is able to deal with Case 1.

The `rewriter` cooperates less with the `scanner`. All in-use vertices will already have been marked. Therefore, cooperation occurs only in the marking of newly created vertices as in-use.

Case 3 can be handled by the `rewriter` cooperating even further so that when it reconfigures the graph, any vertex that has been marked as in-use and is now out-of-use is unmarked. Extra coordination of this sort will slow the `rewriter`. The trade-off is reduced frequency of garbage collection.

5 Conclusions

We have provided an initial investigation of the management of a shared data structure and the (often conflicting) requirements of coherence and parallelism, discussing various possible solutions to these problems. We have also considered the influence of coherence on other general problems, notably that of livelock. We presented a set of access primitives as a means to describe the various management operations. In particular, we have shown that different applications can be defined within our framework.

There seems to have been little work on the questions of coherence versus parallelism relating to shared data structures, even the the problem that we face is at a meta-level to existing work such as the traversal algorithms discussed by Kronsjö [4].

Although the more widely distributed data structures lend themselves to weakly coherent approaches, since the inter-vertex transmission delays required by strongly coherent systems tend to restrict processing efficiency, there are inherent problems that must be accounted for when relaxing coherency.

Finally, the logical representation of both constraints on graph manipulation actions, and on the primitives themselves, give us the opportunity to apply standard temporal theorem-proving techniques in order to perform verification. Thus, part of our future work is to be able to, for example, verify the coherence of the system.

References

[1] M. Derbyshire, Mark Scan Garbage Collection on a Distributed Architecture, *Journal of Lisp and Symbolic Computation* 3, pp. 135-170, 1990.

[2] E. Emerson, *Temporal and modal logic*. In van Leeuwen, J. (ed), Handbook of Theoretical Computer Science, pp. 996-1072. Elsevier Science Publishers, 1990.

[3] R. Johnson, A Blackboard Approach To Parallel Temporal Tableaux, *Proc. AIMSA*, pp. 183-193, World Scientific, 1994.

[4] L. Kronsjö, *Computational Complexity of Sequential and Parallel Algorithms*, Wiley, 1985.

[5] P. Wolper, The Tableau Method For Temporal Logic: An Overview, *Logique et Analyse* Vols 110-111 June-Sept 1985.