A Hybrid Multiple Processor Garbage Collection Algorithm

I. A. Newman, R. P. Stallard and M. C. Woodward

Department of Computer Studies, University of Technology, Loughborough, Leicestershire LE11 3TU

Dynamic storage allocation schemes continue to grow in popularity. The problem of automatically reclaiming space that is no longer used is the principal drawback of such memory management schemes. Several previous papers have described algorithms for the reclaim of unused space to be carried out on separate Garbage Collection processors. This paper gives an overview of existing algorithms and proposes a new combined approach that overcomes some of the problems with these algorithms. Results are presented for several algorithms executed on a four-processor system that suggest that the proposed combined approach offers an improved performance when executed on shared-memory multiprocessors.

Received September 1985

I. INTRODUCTION

When objects in memory are created and deleted dynamically, and are referenced by pointers rather than fixed addresses, it is often necessary to identify and reclaim released space so that it can be re-used. This process, which is commonly referred to as 'Garbage Collection', can be approached in one of two ways. If each object has a count of references to it then the space occupied by that object can be released when the count becomes zero. This method, termed 'Reference Count', will not release cyclic garbage structures, unless special checks are made when such structures are created.1 The other approach, termed 'Mark-Scan' or just 'Marking', traces the whole structure from certain fixed nodes called 'roots' and then reclaims the nodes that were not reached by the trace. It is only the latter approach that is discussed in this paper.

With a conventional single processor system the list processor operates until space is exhausted and then a 'Marking' process is invoked to reclaim space and make it available for further processing. The interest in parallel processing is based on the hope that offloading the task of garbage collection to other processors will both reduce the chance of an embarrassing pause while garbage collection is carried out, and remove some of the overhead of storage reclamation from the list processor. Several algorithms5,6 have been proposed which time-slice a garbage collection process alongside normal list processing on a single processor, but these algorithms do not have to tackle the conflicts which may arise when a garbage collector is run on a separate asynchronous processor. Moreover, these algorithms may require up to double the dataspace of the conventional garbage collector.

It should be stressed that a parallel solution does not ensure the list processor will never have to wait for reclaimed space to become available, it can only reduce the possibility of this happening. The probability of having to wait for space is determined by the rate of demand for new nodes, the size of the free list and the speed and frequency of garbage collection. The dynamic behaviour of the list processor and garbage collector system has been analysed in a recent paper by Hickey and Cohen,4 the present paper studies improvements to the space and time performance of the parallel garbage collector.

The execution time of the parallel marking algorithm must, due to the overhead of multiprocessing, be greater than the normal single-processor solution. The two objectives for a good parallel garbage collection scheme should therefore be: the minimization of interference by the collector on the list processor; and the speed with which garbage becomes available for re-use. The absolute efficiency (processing overhead) of the garbage collector is less important in a parallel processing system as the elapsed time for the list processor no longer includes the collector execution time.

This paper describes garbage-collection algorithms in terms of several asynchronous processes all of which can access a large shared data structure and which can also access local and shared workspaces. The algorithms could, however, be adapted for different types of parallel computer architecture.

The problems associated with some of the existing parallel algorithms are discussed, namely the Minsky–Knuth–Steele–Müller–Wadler algorithm (referred to as 'Stacking')7,8,9 the Woodward algorithm (referred to as 'Chaining')7 and the Dijkstra–Lamport algorithm.8 A new hybrid marking algorithm is described that solves some of these problems. Results are presented from an implementation of this algorithm on the Neptune four-processor system at Loughborough University.10

2. TERMINOLOGY

This paper uses the same terminology as earlier papers on multiple-processor garbage-collection algorithms.7,8,11 The processes that carry out the marking of the nodes are termed 'markers' and processes that carry out the actual accesses and updates to the nodes 'mutators'.

The node space is assumed to consist of N nodes of the same size, of which A nodes are accessible to the mutators. Those remaining (N−A) are the garbage nodes. Nodes which are already on the free list (i.e. available for allocation) are regarded as accessible to the mutator. Each node has a number of pointers and some data values. Each pointer can reference any node (including itself) or have a special value 'nil' to indicate that no node is referenced. The average number of non 'nil' successors of a node is designated by 's' (a normal LISP system has two pointers per node and 's' will be at most two). All the accessible nodes are reachable from fixed 'roots' that can be referenced by both mutators and
markers. Arbitrary structures can be built up with the nodes, including cyclic structures.

The unused nodes that are available for allocation by the mutator are held on the free list. The free list nodes are assumed in this paper to be marked in the same manner as all other mutator accessible nodes but this may be avoided. Nodes which have been identified as garbage are added to the free list by the 'collector'.

A structure is termed interconnected if any of the nodes can be reached by two or more different paths from the roots.

In addition to the pointers and data values each node must also have a field to indicate the mark status of the node. For some of the algorithms only a single bit is required, indicating whether it has been marked. For others there are intermediate states that must be identified, and the field contains two or more bits defining the 'colour' of the node. The normal colour coding scheme is as follows

White: nodes that have not been marked; garbage nodes remain white.

Grey: nodes that are reachable but still have to be used in the marking process; these will become black before marking has finished.

Black: nodes that are reachable and need no further processing by the marker.

2.1. Multiple process conflicts

When there is both a marker and a mutator active on the same nodespace there is potential conflict, in the same way that readers and writers can conflict when accessing a shared data structure. A process (mutator or marker) can no longer assume that the pointers or values in the nodespace will remain unchanged between successive accesses by that process.

Two areas of conflict will arise: over accesses to the nodespace and over accesses to other shared data structures. If standard techniques of locking the whole structure were used to ensure that only one process updates the shared data at any time then the access to the nodespace would be impractically slow. Access to the nodespace is therefore usually limited to only one mutator or, if the nodespace can be subdivided into logically distinct areas, to one mutator per area.

There can be no direct conflict between markers and mutators over garbage nodes since these nodes are, by definition, inaccessible to the mutators. Markers cannot conflict with other markers either, since they only set the mark field and never reset it, although they can duplicate work if two or more markers access the same node at the same time. However, the mutator and marker can conflict if the mutator moves unmarked structures on to structures that the marker believes it has marked. To avoid this conflict the mutator must inform the marker it has moved nodes that may otherwise escape marking. The various marking algorithms differ in the way this information is passed as well as in the algorithm for tracing structures.

Both mutator and collector can access the free list, the mutator by taking new nodes and adding them to data structures and the collector by adding new reclaimed nodes. These producer-consumer conflicts can be solved relatively easily, mutual exclusion being necessary only when the list contains a single node.

All the references to nodes that are used by the mutator (for instance a private stack) must also be accessible to the marker, or else these nodes could escape marking and so be reclaimed as garbage.

2.2 Stacking algorithm

This method, developed and refined by several workers, uses a stack to hold nodes whose successors have not yet been marked. It progressively traverses the structure in depth-first order, placing any unmarked successors on the stack, fetching the next node off the stack and marking its successors. It is clear that in this way it will mark all the nodes in a static structure (no active mutator) even if the structure contains loops. It visits each node only once, and so is optimal in this regard.

The algorithm is a straightforward extension of the normal sequential marking algorithm. Various proposals have been made to improve the performance of the basic sequential method both in terms of space requirement and execution time. Some of these are appropriate for use by a parallel algorithm and are described in a later section.

The Stacking algorithm requires the largest amount of stack space to mark a single long list of nodes from one root, with each node in the chain having the next node in the chain as one successor with $s - 1$ other successors which are terminal nodes (Fig. 1). The algorithm could therefore need to stack $(s-1)$ nodes at each step and there will be $A/s$ steps leading to a requirement of $A(s-1)/s$ nodes.

![Figure 1. Worst case structure for Stacking algorithm.](https://academic.oup.com/comjnl/article-abstract/30/2/119/404272/032151042272/06March2019)

If the system only allows a small number of successors then the storage requirement is considerably reduced. For normal LISP systems $s$ is at most two and the requirement is only $A/2$.

The use of more than one marker process for this algorithm was first suggested by Steele. The simplest multiple-marker implementation is for all the markers to add and remove nodes from a shared stack. Because these stack accesses involve two operations, updating the stack pointer and accessing data at the top of the stack, a lock must be used to prevent markers interleaving these operations. Using the lock, however, causes the stack to become a bottleneck because it is frequently accessed by the markers, and the processes will therefore spend a

120 THE COMPUTER JOURNAL, VOL. 30, NO. 2, 1987
significant amount of time waiting for access to the stack. This is exacerbated by the small amount of marker processing between stack accesses.

If the structure is interconnected there is the possibility of two or more markers adding the same node to the stack, so that the space requirement for the implementation described can have a worst-case value, if there are \( m \) markers, of

\[
\frac{A(s-1)m}{s}
\]

This increase in shared stack size can be avoided if the marker tests the mark bit while the shared stack lock is held. Only if the mark bit has not been set is the node added to the stack. The algorithm will then only require the same total stack size as for a single marker.

Another possible algorithm for multiple markers is for all the markers to use a local stack as well as a shared stack, but using the local stack whenever possible.\(^{14}\) The shared stack would only be accessed if a local stack becomes full or empty. This idea forms the basis of the new algorithm.

The potential mutator–marker conflict for this algorithm is solved if the mutator, when a new link from a marked to unmarked node is made, marks the new successor and adds it to the shared stack.

2.3. Chaining algorithm

Instead of using a shared stack to distribute work, this method uses a shared subroot list.\(^{7}\) Each marker takes a node from this subroot list and marks the whole of the structure accessible from that node. These nodes are termed ‘subroots’ and the nodes accessible from each subroot a ‘subtree’. (The subtree is marked by repeated depth-first tracing from the current subroot. It is therefore effectively marked from the bottom of the structure upwards.) The subroot list size must be large enough to hold all the fixed roots (the initial members of the subroot list), but otherwise the space requirement for the marker is flexible, it can be arbitrarily small. Markers remove a node from the subroot list and commence marking the subtree emanating from it. Periodically (when a node with two or more unmarked successors is found), the marker checks the subroot list. If the number of nodes on the list has fallen below some arbitrary size then the marker will add one or more (but not all of) the unmarked successors of the current node to the subroot list. An extra colour code ‘yellow’ is used to distinguish nodes that are on the subroot list and should not therefore be marked if reached from another subtree. A yellow node is effectively a black node as far as other markers are concerned.

The principal advantages of this algorithm are that there is no longer a need for a single shared stack and that singly linked lists are marked rapidly.

Unfortunately this method can loop indefinitely if certain types of cyclic structure are found in the subtree. A count of the number of nodes traced could enable the algorithm to detect such a loop and break out of it, but this method would be easily outperformed by other marking algorithms on these particular kinds of structure.

Another disadvantage of this method is that nodes may be visited several times during marking, particularly in the case of highly branched structures. Furthermore, instead of the one-mark bit used by ‘Stacking’) it needs a two-bit colour field on each node.

In order to avoid mutator–marker conflicts the mutator must add, if it makes a link from a black to a white node, the white node to the subroot list. If the mutator finds the subroot list full it may have to wait until a marker removes a new subroot from the list. Although this may happen very infrequently, the time to add a link between two nodes can have as worst case the marking time for the whole structure. The system must therefore tolerate a possible delay of this length.

2.4. Dijkstra–Lamport algorithm

When the idea of multiprocessing garbage collection was first studied more importance was placed on the proof that a solution could not cause conflicts between the mutator and marker rather than on finding an optimal solution. The original algorithm was described for a single marker by Dijkstra\(^{6}\) and extended for multiple markers by Lamport.\(^{9}\) A colour code is used on each node to indicate the status of marking.

Each marker is given a disjoint subset of the nodes to mark and repeatedly scans through its subset of nodes until it finds a grey node. The successors of the grey node are shaded (set grey if they are white) and the node itself coloured black. The marker is then reset to the start of its subset of nodes and the scan is repeated until all the markers fail to find a grey node during a complete scan.

Because this algorithm makes repeated scans through already marked and garbage nodes there are a large number of node visits, normally of order \( N^2 \).

In the original algorithm each time a grey node is found the marker resets itself to the start of its nodespace. A recent report and paper\(^{11,14}\) showed how a simple modification substantially reduces the number of node visits for most kinds of structure. This new algorithm has a comparable performance to ‘Chaining’ for structures with a large number of successors. Since the method scans through the physical structure rather than tracing the logical structure there is a strong dependence between the number of node visits and the physical ordering of nodes. Performance will be best when pointers are directed in the same direction as the marker scan; this dependency could be beneficially exploited in some systems.

For this algorithm the mutator–marker conflict can be solved if the mutator checks the colour of nodes whenever it makes a new link. If the link is from a black to a white node then it must shade the white node and reset all the markers as if one of them had found a grey node. The markers will then scan through all the nodes to find the new grey node.

2.5. Summary of existing methods

Each of these algorithms has its drawbacks. The main problem with Stacking is the large stack space requirement;\(^{2}\) with Chaining it is the inability to mark some types of cyclic structure; and Dijkstra–Lamport suffers from a possibly very long execution time.

Algorithms that scan the logical nodespace rather than the physical nodespace have the advantage that the number of node visits will depend on ‘A’ rather than ‘N’. But the best marking algorithm of this type, the
Stacking algorithm, has the drawback that a large workspace is required to hold the path from the root to the current position in the tree while the structure is traced.

An attractive compromise is for an algorithm to be tailored to be space- and time-efficient for the majority of types of structure that are ever likely to be marked and less efficient for other structures. The average marking time of the algorithm would therefore be reduced.

2.6. Hybrid algorithm

The following objectives were in mind when formulating the new algorithm. (A) Good multiple marker performance. (B) Rapid and predictable marking times. (C) Independence between the marker’s workspace size and the total number of nodes or the type of structure. (D) Small average overhead on mutator operations imposed by garbage collector activity and also a low maximum overhead. (E) Ability to mark arbitrary structures (including cyclic structures). It can be seen that none of the algorithms meets all these objectives but a hybrid of the methods could possibly do so.

Accordingly, the new algorithm uses a local stack to mark a subtree whose root is passed via a shared subroot list. When the subtree has been marked (the stack is empty) the marker can fetch a new subroot from the list. So in some senses it resembles ‘Chaining’, with the use of ‘Stacking’ to mark each subtree. Because there is only one visit to each node in the subtree there is no need for the colour yellow and the colour grey. Nodes are either ‘black’ when they have been marked and may be on the stack or subroot list, or ‘white’ when unmarked.

If the local stacks are the same size as for the ‘Stacking’ algorithm, then the stacks will always be large enough and the nodes will be visited only once. But if the markers never attempt to mark the same node then the individual stack sizes can be reduced by a factor equal to the number of markers. When a marker runs out of private stack space it could pass the current node’s successors via the shared subroot list to the other markers. However, the markers would only be guaranteed to mark different nodes if the structure is not interconnected. For the case where the structure is interconnected there is the danger of a node being marked by more than one marker (and subsequently the subtree referenced by that node). This effect cannot be cured in the same way as when a shared stack was used, because the markers work autonomously with private stacks. One possible solution would be for an indivisible ‘test and set’ instruction to be used to mark the nodes; if the test found the node already marked then the node need not be added to the private stack as there must be at least one other path to that node.

There is also the refill policy of the subroot list to be considered. In the ‘Chaining’ algorithm this was done by refilling with all but one of the new subroot’s successors, provided there was still room for one subroot to be added by each mutator. This method may lead to markers being starved of work if it happened that a marker was working on a very large subtree. An alternative refilling policy would be to check the current subroot list each time a node with more than one successor is marked, and refill the list if it is shorter than a certain length. This would give a better distribution of work, but with the extra cost of performing the checks.

Clearly such an algorithm meets objectives (A, B and E), but the space requirement objective C, is not satisfied as the total size may still be too large for systems with very large N (perhaps several million or more).

The last objective (D) is also difficult to satisfy. When the mutator changes a marked section of the list structure it has to pass information to the marker. If a shared workspace is used there is always the danger of that workspace becoming full. The mutator may therefore be delayed until the remainder of the structure has been marked. The probability of a mutator generating new subroot entries may be very small in practice and the risk of waiting for long periods can be reduced by using a larger shared workspace. The alternative is for the mutator to mark the nodes themselves in some way and force the marker to scan through all physical nodes, as in the Dijkstra–Lamport method. This would lead to a possible extra scan and so increase the marking time substantially. In a combined approach the mutator would normally use the subroot list, but if it found the list full it could set a shared boolean flag to indicate that it has resorted to marking nodes. The marker would then only need to perform an extra scan if it found the flag to be set. This extra marker scan would then be extremely rare and the average marking time will not be significantly increased.

2.7. Space requirement

A major problem with the above algorithm is the large workspace requirement for each of the markers. Moreover, it is extremely unlikely that all this space will ever be used. In fact for the structures used for the evaluation of the hybrid algorithm the largest stack usage was for a highly interconnected structure (described in the results section), and in this case only 15% of the available stack space was needed. It is therefore worth investigating methods of reducing the requirement.

The techniques used for stack space reduction by the sequential stacking algorithm can be applied to the private stack of the hybrid algorithm. These cannot guarantee that the stack will never overflow, but can reduce the possibility to a tolerable level. There are two main methods; one avoids stacking nodes unnecessarily and the other compacts the stack when it becomes full by removing entries that no longer need to be held. These approaches only give a large benefit for non-interconnected structures. For simplicity neither of these methods has been included in the algorithm implementation given below.

The hybrid algorithm described above does not have the same space requirement as for the original Stacking algorithm. The use of the subroot list as a stack ‘overspill’ has to be taken into account. For a single marker the subroot list refilling policy will leave a certain proportion of the subroot list free (the effect of mutator-added subroots is ignored). So, when the stack becomes full, this free space can be used. For the worst case structure for Stacking this will mean that the structure can be processed in chunks of the stack size and the rest passed via the subroot list. Any finite size of stack will therefore be sufficient to mark an indefinitely sized structure of this type.
When either the structure is interconnected or an active mutator is present, the space requirement is more complicated to calculate. An interconnected structure can lead to subtrees referenced by the subroot list or the stack being marked by tracing other paths to these nodes. Even though compaction techniques can be applied to both the stack and the subroot list to ease this problem, complicated, interconnected structures and non-deterministic, dynamic behaviour by the mutators and markers could increase the space required up to the same limit as the original stacking algorithm.

2.8. The modified Dijkstra–Lamport as a fallback algorithm

Objective D cannot be met for the hybrid algorithm as it stands. The only alternative is to use it with a smaller stack size and then carry out a special ‘fallback’ action if both the stack and subroot list ever become full. This fallback routine must mark the remainder of the structure with a fixed-size private workspace, and so must use the mark fields on the nodes to carry out the marking. Both the Dijkstra–Lamport and the Chaining algorithms use this method of structure tracing.

Dijkstra–Lamport makes extra unnecessary node visits as it scans repeatedly through the physical nodespace and cannot be restricted to mark a particular substructure. The Chaining method, however, will mark a particular substructure and generally make fewer node visits than Dijkstra–Lamport, but unfortunately it will not mark some types of cyclic structure. As the ability to mark an arbitrary structure was considered an important objective a modified Dijkstra–Lamport method is the only suitable type of fallback algorithm.

Most of the node visits in Dijkstra–Lamport are made when searching for the next grey node to process. The order of processing grey nodes is completely arbitrary, and so the method can be improved if the algorithm can make a good guess as to where the next grey node lies. One improvement, described in a recent paper, allows the markers to scan forward from the previous grey node, rather than resetting to the start of the nodespace, thus probably processing several grey nodes in each scan. A further improvement can be obtained if a marker looks for a grey successor of the node it has just processed. In this way a singly linked list can be marked by visiting each node in the list only once. However, this modification results in markers tracing through structures in other markers’ node subsets, and it is therefore possible for them to mark the same node. The multiple marker performance is expected to be poor because of this duplication of work.

There must still be a final scan of all the nodes to ensure there are no more grey nodes left, and so the minimum number of node visits to mark the whole of the structure using the modified algorithm would be $A + N$. The maximum number of node visits would be

$$N\left(\frac{A}{2} + 1\right) + \frac{A}{2}$$

(where $[N]$ is the smallest integer greater than $N$) for the worst-case structure of a single binary tree with each node having a single terminal node and the rest of the tree as successors (this is the same worst-case structure as for Stacking). This figure arises because the marker needs to scan through all $N$ nodes to mark each level of the structure, and on average will need to scan through half of the accessible nodes to find it (assuming that at each level the algorithm chooses the single terminal node as the next subtree to trace from).

As will be seen in Section 3, this algorithm gives a roughly uniform performance for all types of structure.

The mutator–marker conflict is solved in exactly the same way as for the original Dijkstra–Lamport method.

2.9 Hybrid algorithm with reduced workspace requirement

By using the fallback algorithm described above when both the private stack and the subroot list are full, the two remaining objectives (C and D) can be met. The subroot list and stack size can now be set to any fixed size, it is no longer dependent on $N$. The reduction in marker stack space requirement has cost an extra bit on each node as there are now three possible mark status values instead of two. If the choice of the coding of the colours white, grey and black is made carefully then the node colouring operations need only be single-bit operations, as the unused two-bit combination can also be used to represent black.

If the fallback routine is ever called there will be an increase in marking times and the maximum possible marking time will be significantly increased. The chances of calling the fallback algorithm can be reduced by using compaction methods on both the stack and the subroot list. An advantage of using the fallback algorithm is that there need not be the test and set instruction that ensures a node is not stacked by more than one marker, because there is no longer a problem if a stack does become full.

There is a high fixed overhead of at least $N$ node visits whenever the fallback algorithm is called. For a subtree with a uniform height slightly greater than the available workspace size a large number of calls to the fallback algorithm would be generated. A solution to this problem, adopted in this implementation, is for the fallback algorithm to mark all of the remaining substructure held on the stack rather than just the substructure the marker could not put on the subroot list. This approach will generally increase the number of node visits made by each call to the fallback algorithm, but should ensure fewer calls to the routine.

Using a fallback algorithm must not cause conflicts between markers using the two different types of marking algorithm. Only if the structure is interconnected will markers attempt to mark the same nodes. Furthermore, a node must become either black or grey (it can never be set white) and in either case the markers will both attempt to mark its successors. There is therefore only duplication of effort; nodes cannot escape being marked.

The mutator–marker conflict for this combined algorithm is the same as for the hybrid without the fallback routine. It must check if a new black-to-white node link has been made, and if so mark and add the white node to the subroot list. However, if the mutator finds the subroot list full it can now mark the node grey and signal a marker (by setting a shared boolean flag) to run the fallback algorithm to find this node. Moreover, when this happens it is likely that a marker will already be running or be about to run the fallback routine, and
so the overhead will be small. The maximum mutator operation time will now be a small fixed value and will be independent of the marking time; it no longer needs to wait for the marker if the subroot list is full. Objective D is therefore satisfied. The modifications to the algorithm cannot, of course, ensure that the mutator will never have to wait for free space to be provided by the markers, but this possibility is governed principally by the mutator activity and the nodespace size rather than by the efficiency of the markers.

This new combined scheme is now a very complex algorithm as it tries to avoid all the problems with existing methods. The extra complexity introduced by the fallback routine should not be too severe as with a careful choice of marker workspace sizes the fallback algorithm will be used so infrequently that its effect on marking time can be neglected.

3. RESULTS

An implementation of the combined algorithm is shown in Fig. 2, and results obtained on the Neptune multiple processor system\(^\text{16}\) are given in Table 1. The implementation is given in an informal Pascal notation, leaving some of the implementation-specific details undefined, for example the order of visiting successors and the format of the nodes. For the structures and the workspace sizes used (subroot list size 64 node pointers and stack size 100 node pointers) the fallback algorithm was not called. A special structure was designed, and used to check for reliable operation under the combined use of fallback and hybrid algorithms.

The structures were of the following form.

**Linear List Dense (LLD)**
- 5 roots each having a single list of 97 nodes

**Linear List Sparse (LLS)**
- 7 roots each with a list of only 5 nodes

**Curtain Dense (CD)**
- 7 roots from each of which emanated 6 lists of 12 nodes

**Curtain Sparse (CS)**
- 4 roots each with 4 lists of 4 nodes

**Interconnected Dense (IND)**
- 5 roots with 96 nodes arranged as a list with each node having pointers to the subsequent 5 nodes in the list

**Interconnected Sparse (INS)**
- 5 roots with 15 nodes with pointers to the subsequent 4 nodes

**Random branched (RND1)**

A non-interconnected branched structure of irregular shape, each node having on average 2 successor nodes (similar to a binary tree). Has 4 roots with trees containing 100 nodes

**Random interconnected (RND2)**

Similar to RND1 but with 50 extra interconnecting links between parts of the structure but not producing any cycle

**Random interconnected (RND3)**

Similar to RND2 except that 100 rather than 50 extra links were added

**Random branched (RND4)**

A smaller version of RND1 of 7 roots each with 47 nodes

<table>
<thead>
<tr>
<th>Table 1. Performance of the Hybrid algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Node visits</strong></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>LLD</td>
</tr>
<tr>
<td>LLS</td>
</tr>
<tr>
<td>CD</td>
</tr>
<tr>
<td>CS</td>
</tr>
<tr>
<td>IND</td>
</tr>
<tr>
<td>INS</td>
</tr>
<tr>
<td>RND1</td>
</tr>
<tr>
<td>RND2</td>
</tr>
<tr>
<td>RND3</td>
</tr>
<tr>
<td>RND4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2. Performance of the modified Dijkstra–Lamport algorithm</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Node visits</strong></th>
<th><strong>Elapsed time</strong></th>
<th><strong>Number of resets</strong></th>
<th><strong>Node visits</strong></th>
<th><strong>Elapsed time</strong></th>
<th><strong>Number of resets</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>LLD</td>
<td>1484</td>
<td>1.09</td>
<td>1</td>
<td>1509</td>
<td>0.73</td>
</tr>
<tr>
<td>LLS</td>
<td>1025</td>
<td>0.48</td>
<td>1</td>
<td>1025</td>
<td>0.19</td>
</tr>
<tr>
<td>CD</td>
<td>1449</td>
<td>1.08</td>
<td>1</td>
<td>1966</td>
<td>0.62</td>
</tr>
<tr>
<td>CS</td>
<td>1052</td>
<td>0.53</td>
<td>1</td>
<td>1552</td>
<td>0.27</td>
</tr>
<tr>
<td>IND</td>
<td>1484</td>
<td>1.60</td>
<td>1</td>
<td>2022</td>
<td>0.57</td>
</tr>
<tr>
<td>INS</td>
<td>1105</td>
<td>0.69</td>
<td>1</td>
<td>1606</td>
<td>0.31</td>
</tr>
<tr>
<td>RND1</td>
<td>2712</td>
<td>1.55</td>
<td>4</td>
<td>2712</td>
<td>0.53</td>
</tr>
<tr>
<td>RND2</td>
<td>2776</td>
<td>1.62</td>
<td>4</td>
<td>2778</td>
<td>0.56</td>
</tr>
<tr>
<td>RND3</td>
<td>2816</td>
<td>1.68</td>
<td>4</td>
<td>2821</td>
<td>0.56</td>
</tr>
<tr>
<td>RND4</td>
<td>2175</td>
<td>1.26</td>
<td>3</td>
<td>2175</td>
<td>0.46</td>
</tr>
</tbody>
</table>

| Table 3. Best parallel algorithm from performance study| **Algorithms** | **Node visits** | **Elapsed time** |
|------------------------------------------|----------------|-----------------|
| **Node visits** | **Elapsed time** | **Node visits** | **Elapsed time** |
| LLD          | Chaining       | 516             | 0.71            | Chaining        | 516             | 0.28            |
| LLS          | Chaining       | 57              | 0.09            | Chaining        | 57              | 0.06            |
| CD           | Chaining       | 790             | 1.09            | Chaining        | 726             | 0.30            |
| CS           | Chaining       | 120             | 0.19            | Chaining        | 120             | 0.10            |
| IND          | Stacking      | 500             | 1.79            | Mod. D–L       | 4000             | 0.90*           |
| INS          | Stacking      | 121             | 0.40            | Chaining        | 324             | 0.19            |
| RND1         | Stacking      | 416             | 1.07            | Chaining        | 932             | 0.39            |
| RND2         | Stacking      | 416             | 1.11            | Chaining        | 1018            | 0.43            |
| RND3         | Stacking      | 416             | 1.16            | Chaining        | 1114            | 0.44            |
| RND4         | Stacking      | 345             | 0.88            | Chaining        | 744             | 0.34            |

* The version of Modified Dijkstra–Lamport used is as described in Refs. 7 and 14, and not the new Dijkstra–Lamport algorithm proposed here.
Table 4. Performance of a sequential ‘Stacking’ algorithm

<table>
<thead>
<tr>
<th>Node visits</th>
<th>Elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLD</td>
<td>500</td>
</tr>
<tr>
<td>LLS</td>
<td>41</td>
</tr>
<tr>
<td>CD</td>
<td>500</td>
</tr>
<tr>
<td>CS</td>
<td>80</td>
</tr>
<tr>
<td>IND</td>
<td>500</td>
</tr>
<tr>
<td>INS</td>
<td>121</td>
</tr>
<tr>
<td>RND1</td>
<td>416</td>
</tr>
<tr>
<td>RND2</td>
<td>416</td>
</tr>
<tr>
<td>RND3</td>
<td>416</td>
</tr>
<tr>
<td>RND4</td>
<td>345</td>
</tr>
</tbody>
</table>

Initially:
Set colour of all nodes white;
Initialise subroot list pointer;
Initialise stack pointer (for each marker);
Add all the fixed roots to the subroot list;

```
type
  node_pointer = ↑node;
  node = record
    colour : (white, grey, black);
      { The colour field used for marking }
    datavalue : datatype;
      { Arbitrary data }
    successors : array [1:?] of node_pointer;
      { Pointers to other nodes }
  end;

critical_resource = { Mutual Exclusion mechanism = system dependent }

var
  n : local node_pointer;
  stack : local array of node_pointer [1:?]; { variable size }
  current, successor : local node_pointer;
  subroot : local node_pointer;
  subroot_list : shared array of node_pointer [1:?]; { variable size }
  srl : shared critical_resource;
    { for exclusive access to the subroot list }
  fallen_back : local boolean;
  rescan : shared boolean;

{ Add a node to the shared subroot list to spread work to other markers and also to act as a stack overspill.
If the subroot list turns out to be full then this function returns ‘false’ so the caller can take alternative action. }

function add_to_subroot_list (n : node_pointer) : boolean;
begin
  enter (srl);
  if subroot_list not full then { the list may now be full }
    begin
      n↑.colour := black;
      push (n, subroot_list);
      add_to_subroot_list := true;
    end
  else add_to_subroot_list := false;
end;

begin;
{ The procedure to be called every time a link between two nodes is made }

procedure addlink (source, destination : node_pointer);
var
  added : boolean;
begin
  if source↑.colour = black and
    destination↑.colour = white then
    begin
      if subroot_list not full then added := add_to_subroot_list(destination)
        else added := false;
    if not added then
      begin
        destination↑.colour := grey;
        rescan := true;
      end
    end
  end;

procedure fallback;
var
  n, next, current, successor : local node_pointer;
begin
  rescan := true;
  while rescan do
    begin
      rescan := false;
      { scan through all nodes in the nodespace }
      for n := each node in the nodespace in turn do
        begin
          next := n;
          while next <> nil do
            begin
              current := next;
              if current↑.colour = grey then
                begin
                  next := nil;
                  for all successors of current do
                    begin
                      shade (successor);
                        { set successor node grey if it was white }
                      if successor↑.colour = grey and not rescan then
                        begin
                          next := successor;
                          rescan := true;
                        end
                    end
            end
        end
    end

{ The main program – executed by all the marker processes }

begin
  while subroot_list not empty do
    begin
      enter (srl);
      if subroot list empty then exit (srl)
        else begin
            ```
pop (subroot, subroot list);
exit (srh);
push (subroot, stack) ; {put on local stack}
fallen_back := false;

while stack not empty and not fallen_back do
begin
pop (n,stack) ;
for each white successor of (n) do
begin
successor. colour := black;
{ See if need to spread work to other markers }
if subroot_list needs refilling then
added := add_to_subroot_list (successor)
else added := false;
{ Otherwise add to local stack }
if (not added) and stack not full then
begin
push (successor.stack);
added := true
end
{ If local stack is full then must try to add to subroot list }
else if (not added) and subroot list not full then
added := add_to_subroot_list (successor);
if not added then
begin
{ both stack and subroot list are full use fallback algorithm }
for each node on the stack do stacked-
node. colour := grey;
set stack empty;
successor. colour := grey;
fallback;
fallen back := true { force to choose new subroot }
end
end
end
end;
if rescan then fallback
end.

Figure 2. Hybrid algorithm

Random' structures because the marker will be able to trace the linear list structures faster than more highly branched structures. As expected, the four-processor performance of this algorithm is poor due to the duplication of work.

Table 3 gives a summary of results obtained for parallel marking algorithms on these structures in a previous study. This table lists the best algorithm in terms of execution time for each type of structure. Comparing these figures with the results for the hybrid algorithm (Table 1) shows that the hybrid is the fastest parallel algorithm for all the structures except for 'LLD', where Chaining is slightly faster due to its very rapid list-marking time. The speed-up obtained by using four markers as against one varies from 1.8 ('LLS') to 3.1 ('RND1'). The timings for the sparse structures are dominated by the fixed initialisation costs and it is likely that improved speed-up factors would be obtained with larger structures.

For comparison, the results for a sequential 'Stacking' algorithm are given in Table 4, so the overhead of parallel operation can be seen. This overhead is between 10 and 50% for the structures tried. When the marking algorithm is run in parallel on four processors a speed-up over the sequential method of between 1.3 and 2.8 is obtained.

The hybrid algorithm has been shown to be reliable in the presence of mutators. It has been run in parallel with one, two and three mutators, where each mutator continually modified pointers within a tree accessed from a different root. This test was designed to generate as many mutator–marker conflicts as possible, and ran without producing any data-structure corruption or the garbage collection of any of the accessible nodes.

4. CONCLUSIONS

It has been shown that existing parallel-marking algorithms can be combined to produce a new, faster algorithm with reduced workspace requirement for shared-memory multiprocessor systems.

Performance figures show that parallel execution of the hybrid algorithm gives improved marking execution time when compared to a sequential algorithm. As the number of markers increases, the overheads of shared workspace access will increase. It is likely, therefore, that at some point this will outweigh the benefit gained from distributing the marking to more processors. This effect has already been observed in the case of other algorithms.4

Further work could investigate a combined reference count and marking scheme. It would also be possible to partition the workspace into sections dependent on the time of creation of the node in the same way as Baker's algorithm,3, 15 so that the observation that most nodes have a very short 'lifetime' can be exploited. This scheme marks the recently created nodes most frequently, and older sections are only marked when insufficient garbage is found. This modification can dramatically reduce the number of node visits and hence marker execution time, as well as reduce the workspace size requirement. The compaction phase of garbage collection also requires further study with a view to discovering a more efficient parallel algorithm.5, 16
ACKNOWLEDGEMENTS

The authors are grateful to the U.K. Science and Engineering Research Council for its support of Mr R. Stallard and the development of the Neptune System under grants GR/A 90302 and GR/C 38692. The authors would also like to thank the referee for helpful suggestions for improving this paper.

REFERENCES