# A Shared Memory Architecture for Parallel Cyclic Reference Counting

Rafael D. Lins

Dept. de Informática - Universidade Federal de Pernambuco - Recife - Brazil
Computing Laboratory - The University of Kent - Canterbury - England.

## Introduction

In late 1950's a whole number of applications in computer science started to make use of complex data structures which were unable to be allocated and de-allocated by using a stack discipline. These complex applications, such as early list based theorem provers and programming languages like LISP, made intensive use of space and needed either implicit or explicit run-time memory management. The two most simple algorithms used for memory management are mark-scan and reference counting.

The mark-scan garbage collection algorithm works in two phases. If a machine runs out of space, the computation process is suspended and the garbage collection algorithm is called. First, the algorithm traverses all the data structures (or cells) in use putting a mark in each cell visited. Then the scan process takes place collecting all unmarked cells in a *free-list*. When the mark-scan process has finished, computation is resumed. The amount of time taken for garbage collection by the mark-scan algorithm is proportional to the size of the heap (the work space where cells are allocated). The copying algorithm is a modified version of the mark-scan algorithm which appeared with the advent of virtual memory operating systems. In this algorithm the heap is divided in two halves. The algorithm copies cells from one half to the other during collection. Its time complexity is proportional to the size of the graph in use. Mark-scan and copying algorithms generally traverse all the reachable data structures during garbage collection, which makes them unsuitable for real-time or large-virtual-memory applications.

In reference counting, each data structure or cell has an additional field which counts the number of references to it, i.e. the number of pointers to it. During computation, alterations to the data structure imply changes to the connectivity of the graph and, consequently, re-adjustment of the value of the count of the cells involved. Reference counting has the major advantage of being performed in small steps interleaved with computation, while other algorithms imply suspending computation for much longer. The disadvantage of the trivial algorithm for reference counting is the inability to reclaim cyclic structures. To solve this problem, a mixture of mark-scan and reference counting has already been used in the past. We refer to [2] for a detailed analysis of these algorithms.

In 1975 Steele [10] proposed what was possibly the first algorithm for parallel garbage collection. In his architecture two processors share the same memory space. One of the processors is responsible for graph manipulation while the other performs garbage collection. In this algorithm mark-scan and computation occur simultaneously.

Another parallel mark-scan algorithm is presented in [3], and was considered by Ben-Ari "one of the most difficult concurrent programs ever studied" [1]. This algorithm was

implemented in hardware/software in the Intel iAPX-432 computer and iMAX operating system [9]. Kung and Song developed an improved mark-scan algorithm [6] based on the algorithm by Dijkstra et al. Based on the same algorithm Ben-Ari gave [1] several parallel mark-scan algorithms with a much simpler proof of correctness then the ones presented in [6, 3].

All the algorithms mentioned above for parallel mark-scan seem to spend a lot of time colouring non-garbage cells and scanning the whole heap. As an alternative to mark-scan algorithms Wise proposed a on-board reference count architecture [11] and Hughes developed an algorithm for distributed reference counting [5]. In this paper we present a shared memory architecture for parallel cyclic reference counting.

# 1 Standard Reference Counting

In reference counting, each data structure or cell has an additional field which counts the number of references to it, i.e. the number of pointers to it. During computation, alterations to the data structure imply changes to the connectivity of the graph and, consequently, re-adjustment of the value of the count of the cells involved. We assume that cells are of constant fixed size.

Free cells are linked in a structure called a *free-list*. For a matter of convenience all cells in the free-list will have their reference count set to one. A cell $B$ is *connected* to a cell $A$ ($A \rightarrow B$) if and only if there is a pointer $\langle A, B \rangle$. A cell $B$ is *transitively connected* to a cell $A$ ($A \stackrel{*}{\rightarrow} B$) if and only if there is a chain of pointers from $A$ to $B$. The initial points of graphs to which all cells in use are transitively connected are called *roots*. For the sake of simplicity and without any loss of generality we will assume that there is only one root in the graph.

There are three operations on the graph:

1. `New` which selects a cell from the free-list and links it to the graph.

   ```
   New (R) = select U from free_list
             make pointer <R,U>
   ```

   `make pointer <R,U>` fills one of the fields of cell $R$ with a pointer to $T$. For the sake of simplicity of notation we will not especify in which field of a cell this filling takes place.

2. `Copy` copies information between cells and increments the reference count of the cell we made a new reference to. Algorithmically we have,

   ```
   Copy (R, <S,T>) = make pointer <R,T>
                     increment RC(T)
   ```

3. `Delete` which deletes a pointer to the graph and proceeds the necessary re-adjustments as follows:

```
Delete <R,S> = remove <R,S>
               if RC (S) = 1  then
                        for T in Sons (S) do
                             Delete <S,T>
                        link_to_free-list (S)
               else
                    decrement RC (S)
```

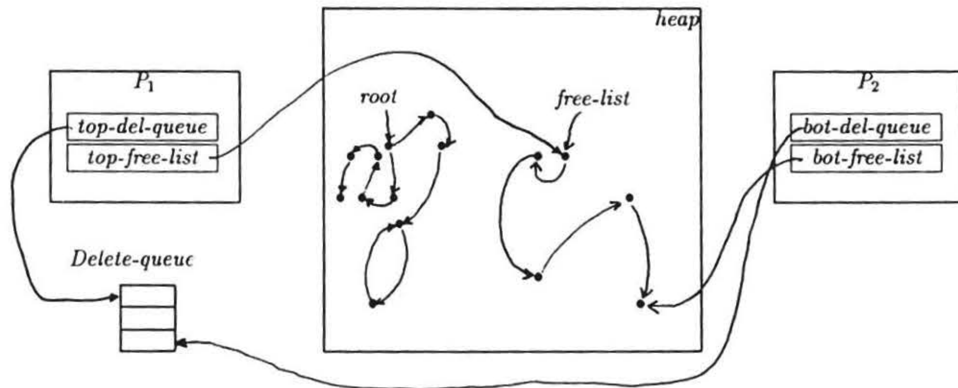
`remove <R,S>` performs the actual deletion of the pointer to $S$ in one of the information fields of cell $R$. To keep our notation simple fields will not be especified. `Sons (S)` is the bag of all the cells $T$ such that there is a pointer `<S,T>`.

As we mentioned before, standard reference counting is not able to recycle cyclic structures. Later on in this paper we will present another algorithm which overcomes this drawback.

## 2 A Shared Memory Architecture

In this section we propose a shared memory architecture to implement the algorithm above. Our idea is to have two processors say $P_1$ and $P_2$, which will perform graph rewriting and garbage collection simultaneously.

Both processors share the same memory area, the working space which is organised as a heap of cells. In case of simultaneous access from both processors to a given cell semaphores are used such as to guarantee that processor $P_1$ will have priority over processor $P_2$. There is also another shared data structure: the *Delete-queue*, which is organised as a FIFO. Processor $P_1$ is only allowed to push data onto the Delete-queue. Conversely, processor $P_2$ is only allowed to dequeue data from the Delete-queue. Processor $P_1$ has two registers called *top-free-list*, which stores a pointer to the top cell in the free-list, and *top-del-queue*, which stores a pointer to the top of the Delete-queue. Processor $P_2$ has two registers called *bot-free-list*, which stores a pointer to the last cell in the free-list, and *bot-del-queue*, which stores a pointer to the bottom of the Delete-queue.



For the sake of simplicity we ignore the syncronization that must be done when $P_1$ attempts to remove a node from an empty free-list or $P_2$ tries to get a reference from an empty *Delete-queue.* These situations should happen infrequently and any convenient syncronisation primitive can be used.

We now split the algorithm above between these two processors.

### 2.1 Processor $P_1$ Instruction Set

Processor $P_1$ will be in charge of rewritings of the graph. Its instruction set comprises three basic operations: *New*, *Copy*, and *Del.*

*New* tests if there are free cells on the free-list. If not empty it reads the information in register top-free-list and links it to the graph. *New* also gets the address of the new top of the free-list and saves it in register top-free-list. These operations are described as,

```
New (R) = if top-free-list not nil then
             make pointer <R,top-free-list>
             top-free-list := ^top-free-list
```

where `^A` means the information stored in `A`.

*Copy* copies information between cells. No special care is needed in order to keep the correct management of the data structures. If processor $P_1$ wants to copy some information, i.e. to make a pointer to a cell then this cell must be transitively connected to root. Copy increments the reference count of $T$. Algorithmically we have,

```
Copy (R, <S,T>) = make pointer <R,T>
                  increment RC(T)
```

*Del* deletes pointers in the graph, it pushes a reference to a cell onto the top of the Delete-queue. (Processor $P_2$ will perform the remaining operations for the effective re-adjustment of the graph.) Thus,

```
Del (<R,S>) = remove <R,S>
              ^top_del-queue := S
              increment top_del-queue
```

### 2.2 Processor $P_2$ Instruction Set

Processor $P_2$ is the processor in charge of the deletion of pointers and feeding free cells onto the free-list. The main routine in $P_2$ is called *Delete* a routine which will run forever as the kernel of the operating system of processor $P_2$.

```
Delete = if Delete-queue not empty then
              S := bot_del-queue
              increment bot_del-queue
              Rec_del (S)
         else
              call Delete
```

If the Delete-queue is not empty `Delete` calls `Rec_del`, as follows

```
Rec_del (S) = if RC (S) = 1  then
                          for T in Sons (S) do
                               Rec_del (T)
                          link_to_free-list (S)
              else
                     decrement RC (S)
```

The linking of a cell to the free-list is performed by the operations:

```
link_to_free-list (S) = ^S := bot-free-list
                        bot-free-list := S
```

# 3 The Local Mark-Scan Algorithm

Reference [8] presents a simple algorithm for cyclic reference counting. The general idea of that algorithm is to perform a local mark-scan whenever a pointer to a shared structure is deleted. The algorithm works in three phases. In the first phase we scan the graph below the deleted pointer, rearranging counts due to internal references and marking the nodes as possible garbage. In phase two, the sub-graph is re-scanned and any cells to which there are external references are remarked as ordinary cells, and their counts reset. All other nodes are marked as garbage. Finally, in phase three all garbage cells are collected and returned to the free-list.

In addition to the information of number of references to a cell, there is an extra field which keeps the colour of the cell. Three colours are used: green, red and blue. Colours are used to control the status of cells. As initial condition one has all cells painted green and every cell except *root* is on the *free-list*. Green is the stable colour of cells. Red and blue are transient colours which indicate that we are not sure of whether these cells are needed or not.

To make the architecture proposed above work with this new algorithm we change the instructions set of processor $P_2$, keeping processor $P_1$ unchanged. `Rec_del` the routine recursively invoked by `Delete` whenever a pointer is removed now performs the following operations:

```
Rec_del (S) = if RC (S) = 1  then
                          for T in Sons (S) do
                               Rec_del (T)
                          link_to_free-list (S)
              else
                     decrement RC (S)
                     mark_red (S)
                     scan (S)
                     collect_blue (S)
```

`mark_red` paints the transitive closure of $S$ red and decrements the counts of these cells, as follows:

```
mark_red (S) =  if colour (S) is green then
                    set colour (S) := red
                    for T in Sons (S) do
                         decrement RC (T)
                         mark_red (T)
```

`scan` searches for external pointers to the subgraph under inspection. If found the transitive closure of these cells will be painted green.

```
scan (S) = if colour (S) is red then
               if RC (S) > 0  then
                    scan_green (S)
               else
                    set colour (S) := blue
                    for T in Sons (S) do
                         scan (T)
```

`scan_green` paints green all the subgraph below its calling point and increases the reference count of the cells visited, to take into account the internal pointers within the subgraph (which had been set to zero by `mark_red`).

```
scan_green (S) = set colour (S) := green
                 for T in Sons (S) do
                    increment RC (T)
                    if colour (T) is not green then
                         scan_green (T)
```

`collect_blue` recovers all the blue cells in the subgraph below its calling point (garbage) and links them to the *free-list*.

```
collect_blue (S) = if colour (S) is blue then
                         for T in Sons (S) do
                              collect_blue (T)
                              remove <S,T>
                         set RC (S) := 1
                         set colour (S) := green
                         link_to_free_list (S)
```

## 4 The Lazy Mark-Scan Algorithm

An important optimisation of the algorithm above is introduced in reference [7]. In this new algorithm the mark-scan phase is performed lazily, i.e. whenever the free-list is empty. The lazy algorithm uses a stack as an extra control structure to avoid performing the local mark-scan every time we delete a pointer to a cell with multiple references. A reference to

these cells is placed on the control stack. We paint these cells *black*.
In order to introduce this optimisation of our architecture we will introduce a control stack to processor $P_2$ and modify its instruction set. `Delete` operations are performed as follows:

```
Delete = if Delete-queue not empty then
             S := bot-del-queue
             decrement bot-del-queue
             Rec_del (S)
         else
              if control_stack not empty then
                 scan_stack
              else
                 call Delete
```

If the Delete-queue is not empty `Delete` calls `Rec_del`, as follows

```
Rec_del (S) = if RC (S) = 1  then
                         for T in Sons (S) do
                              Rec_del (T)
                         link_to_free-list (S)
              else
                   decrement RC (S)
                   if colour (S) not black then
                       set colour (S) := black
                       top_of_control_stack := S
```

Processor $P_2$ only analyses the control stack when the Delete-queue is empty. Now let us explain how the control stack is used in [7]. We pop the cell from the top of the control stack and test its colour. If it remains black this means that we are still not sure if we have deleted the last pointer to a cycle. (Note that a cell painted black and pushed onto the control stack may be sent to the free-list by another call to delete. From the free-list it may be recycled while it still has a reference from the control stack.) If the cell from the top of the stack is black then we perform a local mark-scan.

```
scan_stack = S := top_of_control_stack
             pop_control_stack
             if colour (S) is black then
                  mark_red(S)
                  scan(S)
                  collect_blue(S)
             else
                  if control_stack not empty then
                      scan_stack
```

`mark_red` is modified to allow black cells as well.

```
mark_red (S) =  if colour (S) is green or black then
                    set colour (S) := red
                    for T in Sons (S) do
                         decrement RC (T)
                         mark_red (T)
```

the other routines invoked in this section which are not redefined are left as presented in the last section.

## 5 Proof of Correctness

A formal proof of the correcteness of parallel algorithms is, in general, not simple [3, 4]. We will give an informal proof of the correctness of the architectures we proposed for parallel reference counting.

The three architectures presented above are based on simple algorithms. It is general knowledge the correctness of the uniprocessor version of standard reference counting, as well as its inability to reclaim cyclic structures. An informal proof of the correctness of the cyclic reference count with local mark-scan algorithm appears in the original paper [8]. The lazy algorithm [7] is an optimisation of the local mark-scan one which can be easily proved correct. Therefore, our major concern is to analyse the interaction between the two processors for each of the architectures presented.

### 5.1 Standard Reference Counting

In the standard reference counting architecture above there are two interfaces between processors: the *free-list* and the *Delete-queue*. In general, there is no direct interaction between processors $P_1$ and $P_2$. If cells are claimed by processor $P_1$ and the free-list is empty $P_1$ must wait for $P_2$ to recycle cells. Conversely, if the Delete-queue is empty, processor $P_2$ waits for processor $P_1$ to push information onto the Delete-queue. This relationship between processors is producer/consumer coupling; this form of syncronisation can be achieved without any need for mutual exclusion.

If the two processors try to access the same cell, let's say $P_1$ is copying a pointer to a cell and tries to increment its reference count while $P_2$ is in the process of decrementing this same count, then semaphores are used such as processor $P_1$ to perform its operation first.

### 5.2 Cyclic Reference Counting

The only difference between the architecture for parallel cyclic reference counting above and the standard reference count one is that we perform a local mark-scan whenever one deletes a pointer which has the possibility of isolating a cycle.

Altough this new architecture keeps the same interfaces of the standard reference count one new aspect must be analysed. The mark-scan phase of processor $P_2$ may occur simultaneously with rewritings of processor $P_1$. In case both processors try to access a given cell processor $P_1$ will have priority over processor $P_2$ as in the case of standard reference counting. The only piece of information in a cell both processors may try to modify simultaneously is

the reference count of a given cell. Observe that the colour information of cells is irrelevant to processor $P_1$. Two points need to be stressed:

- One always makes a connection before breaking a previous one, i.e. one does copy before delete. One does not discard a pointer to a cell and try to copy the same pointer afterwards.
- All cells $P_1$ may copy a pointer to are transitively connected to root, i.e. are accessible from root.

The points above imply that if processor $P_2$ is mark-scanning a subgraph and processor $P_1$ makes a copy to one of the cells in the same subgraph this new pointer to the subgraph is **not** the only connection between this subgraph and root. The only effect this copy may have is to abbreviate the mark-scan phase of processor $P_2$, by finding this new external reference before the old ones. We can conclude that we have the correct interaction between processors $P_1$ and $P_2$.

The lazy mark-scan architecture for processor $P_2$ presents the same external behaviour as the local one, if observed from processor $P_1$ and vice-versa.

# 6 Efficiency

Comparing the efficiency between our algorithm and the ones based on mark-scan presented in references [10, 3, 6, 1] is not a simple task. We will give some empirical reasons why we think our approach is better than the previous ones.

According to Ben-Ari [1] the inefficiency of parallel mark-scan algorithms lies in spending a lot of time colouring non-garbage cells. He proposes an algorithm which can minimise this drawback if the application contains large data structures that are modified only occasionally. In our lazy algorithm the mark-scan is local and takes place only as a last resort.

In our opinion, the greater the dependency between the processors the less efficient is the algorithm and the more complex its proof of correctness. The distribution of the mark-scan phase brings problems to the mark-scan based algorithms as the one known as Woodger's scenario. In the algorithm presented in reference [3] the mutator (the equivalent to our processor $P_1$) also performs part of the marking phase by colouring cells. If the mutator makes a long pause between the linking and colouring phases the collector may claim the whole subgraph below the cell just linked as garbage. Ben-Ari's solution [1] for this problem consists of making the mutator paint the cell first and then link it to the graph. The collector now visits each node at least twice. In our architecture the instruction set of processor $P_1$ is extremely simple and the local mark-scan phase is performed by processor $P_2$ only. So colour change is done by $P_2$ only.

The larger the working area of each processor the higher the probability of simultaneous access to a cell. In practice only one processor will access a cell at a time. (In our case processor $P_1$ has priority over $P_2$.) In mark-scan based algorithms the work space of processor $P_2$ is the whole heap, instead of a subgraph as in our case, this may cause additional inefficiency, as one processor may need to wait for the other to finish its operations before having access to the same cell.

The advantages and simplicity of our algorithm in relation to mark-scan based ones rest on the fact that:

- In reference counting each cell keeps in itself the measure of how much the whole graph needs the information it stores. This also means space overhead for storing counts.
- The interfaces between the two processors are simple and well-defined. This also implies extra space cost for the Delete-queue.
- We need two bits to store four colour information. Ben-Ari's algorithm needs only one bit for this purpose.

In our opinion, our algorithm trades a small space overhead for greater independence between processors which implies in time efficiency and simplicity.

## Conclusions

We presented a simple shared memory architecture for parallel garbage collection. In our opinion, this algorithm is simpler and more time efficient than shared parallel algorithms based on mark-scan. This is still to be born out by experimental results.

## Acknowledgements

I am most grateful for the comments of Prof.Peter Welch, Simon Thompson, S.Vedat Demiralp, José Dias dos Santos, Rudnei da Cunha, and David Beckett on a previous version of this paper.

This paper was written during the author's research visit to the The University of Kent at Canterbury, sponsored jointly by the British Council and C.N.Pq. (Brazil) grants No 40.9110/88.4. and 46.0782/89.4. The author would like to express his gratitude to the Computing Lab. of The University of Kent for having him during his visit and also to the Departamento de Informática - U.F.PE. for granting his research leave.

## References

[1] M.Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[2] J.Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[3] E.W.Dijkstra, L.Lamport, A.J.Martin, C.S.Scholten & E.M.F.Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of ACM*, 21(11):966–975, November 1978.

[4] D.Gries. An exercise in proving parallel programs correct. *Communications of ACM*, 20(12):921–930, December 1977.

[5] R.J.M.Hughes. A distributed garbage collection algorithm. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume LNCS 201, pages 256–272. Springer-Verlag, 1985.

[6] H.T.Kung and S.W.Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science*, pages 120–131. IEEE, 1977.

[7] R.D.Lins. Cyclic reference counting with lazy mark-scan. Technical Report 75, UKC Computing Lab. Report, The University of Kent at Canterbury, July 1990.

[8] A.D.Martinez, R.Wachenchauzer and R.D.Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[9] F.J.Pollack, G.W.Cox, D.W.Hammerstein, K.C.Kahn, K.K.Lai, and J.R.Rattner. Supporting Ada memory management in the iAPX-432. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 117–131. SIGPLAN Not. (ACM) 17,4, 1982.

[10] G.L.Steele. Multiprocessing compactifying garbage collection. *Communications of ACM*, 18(09):495–508, September 1975.

[11] D.S.Wise. Design for a multiprocessing heap with on-board reference counting. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume LNCS 201, pages 289–304. Springer-Verlag, 1985.