A Multi-Processor 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 1975 Steele [3] 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, called the mutator, is responsible for graph manipulation while the other, called the collector, performs garbage collection. In this algorithm mark-scan and computation occur simultaneously.

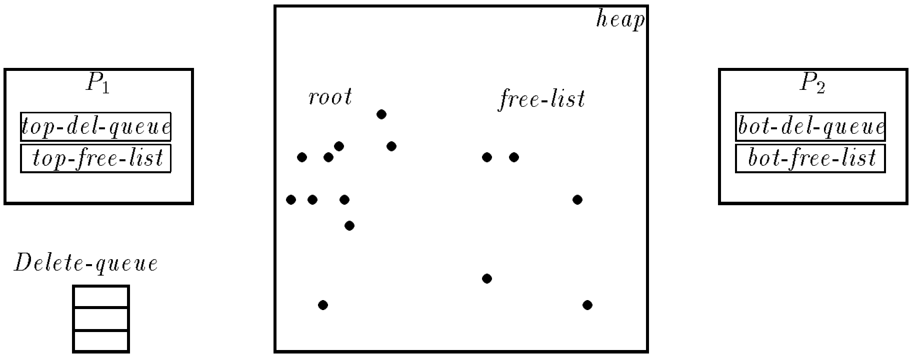
Another parallel mark-scan algorithm is presented in [2]. Kung and Song developed an improved mark-scan algorithm [4] based on the algorithm by Dijkstra et al [2]. Based on the same algorithm Ben-Ari gave [6] several parallel mark-scan algorithms with a much simpler proof of correctness then the ones presented in [4, 2]. 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. Lamport [5] generalised the architecture described in [2] for using multiple processes, with two aims: to speed up the performance of the architecture and to analyse the process of parallelising a sequential algorithm.

As an alternative to mark-scan algorithms Lins presents a shared memory architecture for parallel cyclic reference counting [8], based on the algorithm presented in [10]. In this paper we generalise this architecture in such a way that multiple mutators and collectors share the same workspace. This generalisation is simple and keeps the properties of the one-mutator-one-collector architecture [8].

1 A Shared Memory Architecture

In this section we describe the architecture presented in [8] for Parallel Cyclic Reference Counting with Lazy Mark-Scan [7]. There are two processors 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. We assume that the mutator will never point at a garbage node. However, by changing edges the mutator can turn reachable nodes into garbage. 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 internal 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 also two internal 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 synchronisation 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 synchronisation primitive can be used. In addition to the information of number of references to a cell, there is an extra field which keeps the colour of the cell. Four colours are used: green, red, blue, and black. 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, blue, and black are transient colours which indicate that we are not sure of whether these cells are needed or not.

1.1 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
          else New(R)
```

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
```

1.2 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
              if control_stack not empty then
                 scan_stack
              else
                 Delete
```

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

```
Rec_del (S) = if RC (S) = 1  then
                         set colour (S) := green
                         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
```

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
```

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*.

Processor P_2 only analyses the control stack when the Delete-queue is empty, by calling `scan_stack`.

```
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
```

`scan_stack` pops 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. 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. `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 or black 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*.

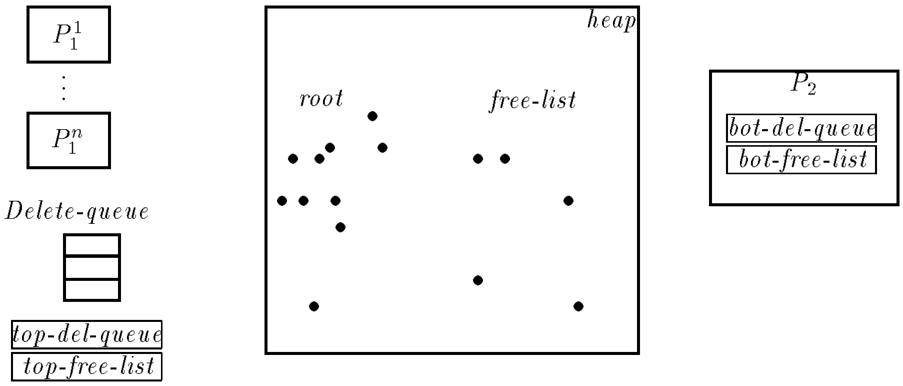
```
collect_blue (S) = if colour (S) is blue then
                          for T in Sons (S) do
                               collect_blue (T)
                               remove <S,T>
                          set RC (S) := 1
                          set colour (S) := green
                   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

2 A Multi-Mutator Architecture

In this section we generalise the architecture we presented in the last section to work with any number of mutators. The mutators must be synchronised in some way so they do not interfere with one another. This synchronisation mechanism must enforce some partial ordering on mutator's operations, which are viewed as atomic actions. This means that if a processor P_1^i has started an operation before a processor P_1^j then operations will actually take place following this order. This partial ordering must be enough to guarantee that the mutators correctly execute some sequential mutator algorithm. This avoids problems such as sending to the free-list cells still in use by performing the deletion of a pointer to a cell before a copy operation to the same cell. We will not concern ourselves with the implementation of this synchronisation, since it will depend upon the details of the individual application.

Synchronization is also needed amongst mutators when removing nodes from a common free-list. The use of several separate free-lists associated with each mutator can reduce synchronisation delays. This can be implemented without any difficulty, but we will not consider it further.

The picture below sketches our architecture.



As we can observe in the picture above, instead of pointing directly to the top of the Delete-queue now each processor will keep a reference to an external register which points at the top of the Delete-queue. Similarly for the top of the free-list.

The instruction set for the mutators is the same as we had before with only one mutator. We will change the way we work with `Copy`. Now `Copy (R, <S,T>)` tests the colour of T. If *black* we reset it as *green*. Algorithmically we have,

```
Copy (R, <S,T>) = make pointer <R,T>
                  increment RC(T)
                  if colour (T) is black then set colour (T) := green
```

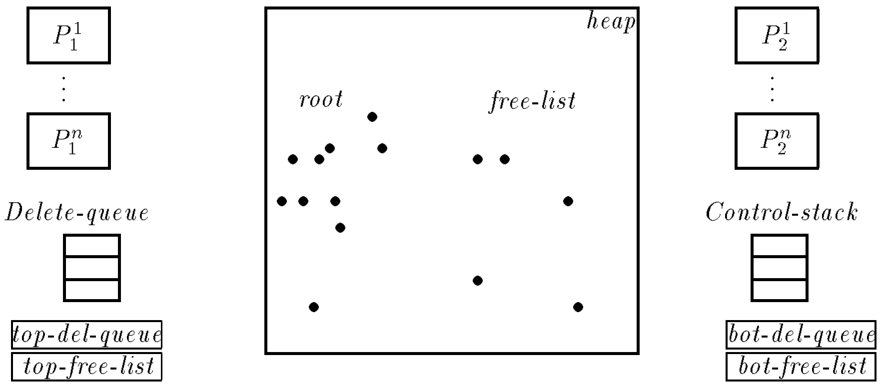
The optimisation above also works in the case of the one-mutator-one-collector architecture. In both cases all it does is to assure that the cell, which had uncertain status (*black*), was actually needed (*green*). If a mutator accesses a cell this means that cell is transitively connected to root, therefore it is in use. Painting the target cell of a `Copy` operation green avoids the possibility of unnecessary calls to the mark-scan.

3 Using Multiple Collectors

There is a number of possible ways we can extend the architecture presented in the last section to work with more than one collector. Our idea is to keep the philosophy of the one-mutator-one-collector architecture presented above as much as possible, in which:

- mutators and collectors do not talk directly to each other.
- interfaces are simple and well defined.
- synchronisation between mutators and collectors when addressing interfaces is kept to a minimum.

If we have the points above in mind the multi-mutator architecture we presented in the last section can be pictured as:



The instruction set of each collector should be modified in order to avoid confusion during mark-scan. In the distributed architecture presented in [9] there is a broadcast of a *suspension* message in the processor network when one of the processors starts to mark-scan. This condition is largely relaxed further on to allow processors to proceed with computation during mark-scan. In the multi-processor shared memory architecture presented above we synchronised all collectors in such a way as to all of them to run the mark-scan simultaneously. The kernel routine which runs on the collectors P_2^i is `Delete`.

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

In which the mark-scan process is activated by calls to `scan_stack`

```
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
```

Our control strategy for synchronisation is such as when one of the collectors start to run `scan_stack`, because the Delete-queue is empty, all the other collectors can do is to either finish or suspend their operation and run `scan_stack` also. Thus `Delete` performs the following operations:

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

Once collectors find a cell whose colour is black it can start to run `mark_red` immediately. When all processors have finished this phase synchronisation is needed before collectors are allowed to start with `scan`. Again, before `collect_blue` all processors must have stopped with `scan` or its ancilliary function `scan_green`. After all collectors have finished with `collect_blue` they are allowed to resume their tasks.

In order to stress this synchronisation mechanism we will rewrite `scan_stack` as:

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

We should stress that `scan_stack` is activated by the lack of cells in the Delete-stack, not by the lack of cells in the free-list. Therefore mutators are still independent of collectors. At no moment there is any loss of parallelism in our architecture. On the contrary, having all collectors doing mark-scan simultaneously brings the advantage of accelerating this process, in the case the mark-scan area is split between collectors.

Proof of Correctness

Formal proofs of the correctness of parallel algorithms are, in general, not simple [1, 2]. We give here an informal proof of the correctness of the architectures presented.

The approach Lamport uses [5] for assuring the correctness of his multi-processor architecture is the *parallelisation* of the sequential algorithm presented in [2] with the addition of some synchronisation elements. This was exactly the strategy adopted by the author in the development of his one-mutator-one-collector shared memory architecture [8], which was based on the sequential algorithms for uniprocessors presented in [10, 7]. The architectures presented herein also follow the same philosophy: we parallelise the original mutator algorithm and then we apply the same technique to the collector algorithm.

The Multi-Mutator Architecture

The existence of a partial ordering on the synchronisation of mutator operations is the key for the correctness of this architecture. This ordering must be such as to guarantee that mutators correctly execute some sequential algorithm, i.e. the sequence of operations is the same as the performed in the one-mutator architecture.

The Multi-Collector Architecture

In this architecture, if the Delete-queue is not empty then each collector will fetch a cell from the back of the delete-queue, if not empty, by calling `Delete`. `Rec_del` is called on it. If the value of the count of this cell is one then it is painted green, has its Sons analysed recursively and then it is sent to the free-list. Synchronization is used to avoid simultaneous access to a given cell.

If the Delete-queue is empty and the control-stack is not then the first processor which tries to fetch a cell from the control-stack will call `scan_stack`. Now, it the top cell is black mark-scan will take place. A new synchronisation mechanism is used to avoid confusion between phases of mark-scan amongst collectors. This synchronisation makes the work cooperative and increases the parallelism of the architecture. Assume each collector is allowed to fetch only one cell from the control-stack before mark-scan. Synchronization after each phase of mark-scan assures that if we have n collectors in our architecture and if the top n cells on the control-stack point at s cells with shared subgraph then after mark-scan the graph obtained is equal to $n - s + 1$ sequential calls to `scan_stack` of the one-mutator-one-collector architecture.

Conclusions

We presented a multi-processor shared memory architecture for parallel garbage collection which is a extension of a one-mutator-one-collector architecture based on cyclic reference counting. In our opinion, this architecture is simpler and more time efficient than Lamport's Garbage Collection with Multiple Processors, which is based on mark-scan. This is still to be born out by experimental results.

Acknowledgements

Research reported herein was sponsored jointly by the British Council and C.N.Pq. (Brazil) grants No 40.9110/88.4. and 46.0782/89.4.

References

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

[2] 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.

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

[4] 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.

[5] L.Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *Proc. of IEEE Conference on Parallel Processing*, pages 50–54. IEEE, 1976.

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

[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. to appear in Information Processing Letters.

[8] R.D.Lins. A shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 32:53–58, August 1991.

[9] R.D.Lins and R.E.Jones. Cyclic weighted reference counting. Technical Report 95, UKC Computing Lab. Report, The University of Kent at Canterbury, December 1991. submitted for publication.

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