Memory Management in Shared Memory Architectures: *A Tutorial*

Rafael D.Lins (rdl@di.ufpe.br)
Dept.de Informática - U.F.PE. - 50.739 - Recife - PE - Brazil

Abstract

This paper presents a survey of the most important algorithms and architectures for dynamic memory management in shared memory machines.
Keywords: Multiprocessors, Memory management, Shared memory, Garbage collection

Introduction

With today's technology the cost of adding extra processors to a machine is small. Most new large mainframes are multiprocessors already and shared memory multiprocessors are becoming widespread. On the other hand, automatic garbage-collection has become a fundamental issue for software modularity and portability, besides freeing programmers of an unnecessary burden. Thus memory management of shared memory architectures, a research area that started almost two decades ago as an academic exercise, has now gained renewed interest. This paper presents a brief survey of the most important ideas on dynamic memory management in shared memory architectures.

Donald Knuth credits Marvin Minsky for first suggesting parallelism as a way to avoid suspension of operations (Exercise 2.3.5–12, pp.422 in [31]). Parallelism need not imply concurrency. Garbage collection could occur, for example, during keyboard input, as long as it could be suspended on short notice to continue processing on the input and later be resumed without losing all the previously expended effort.

Guy Steele's Multiprocessing Compactifying Garbage Collection algorithm was the first published parallel and concurrent architecture for garbage collection [58]. In addition to freeing unused storage, Steele's algorithm compacted remaining list structures to give better performance in a virtual memory environment. The mass of detail presented by Steele contributed to make understanding his ideas difficult.

Independently, Dijkstra proposed a similar scheme in some unpublished notes [16], later published in [17]. Dijkstra and his colleagues tackled this problem 'as one of the more challenging — and hopefully instructive — problems' in parallel programming. Their architecture attracted considerable interest in the computer science community. Woodger's scenario showed that if the granularity of the coarse grain algorithm was made finer, a bug would appear; in describing his proof of the algorithm Gries reported that he had 'seen five purported solutions to this problem, either in print or ready to be submitted for publication' each of which contained errors [22]. A correct version of the algorithm appeared in [18]. S.Ramesh and S.L.Mehndiratta formalised the proof of termination and absence of live-lock in Dijkstra's algorithm [50] by using Owicki and Lamport's proof procedure [47]. Wadler, and Hickey and Cohen analysed its performance. Wadler showed that, for time-sharing rather than multiprocessor systems, such algorithms require a greater percentage of processor time than classical sequential collection does [62]. Hickey and Cohen showed that a mutator-collector system could offer no more than a 50% performance improvement on the sequential one [27].

Kung and Song developed a version which used four colours but did not need to trace the free-list [32], and Ben-Ari, who considered it 'one of the most difficult concurrent programs ever studied', presented several parallel mark-scan algorithms based on it but with much simpler proofs of correctness than those presented by Kung and Song, Gries and Dijkstra *et al.* [5, 6]. Ben-Ari's algorithms used only two colours. Gries accredits Stenning for an unpublished version of the on-the-fly algorithm which also used only two colours. Lamport generalised the architecture for using multiple processes [33]. The on-the-fly algorithm was also implemented in hardware/software in the Intel iAPX-432 microprocessor and iMAX operating system [49].

An efficient algorithm for concurrent collection was proposed by Appel-Ellis-Li [1] for machines that support virtual memory. This algorithm uses paging information provided by the operating system to synchronise the operations between processors.

The previously mentioned algorithms are based on marking and scanning the workspace. A different strategy is offered by reference counting. The first reference counting algorithm for shared memory architectures is presented by Kakuta-Nakamura-Iida in reference [30]. A much simpler architecture was proposed by Lins [36] and later generalised into a multiprocessor architecture described in [37].

Parallel Mark-Scan

In uniprocessors, the Mark-Scan garbage collection algorithm works in two phases: if the user process requests cells when the free-list is empty, the garbage collection routine is invoked. All the cells in the transitive closure of root are marked before scanning the entire heap and returning unmarked cells to the free-list. Mark-Scan is a stop/start algorithm: its minor disadvantage is that the user process is suspended while the garbage collector runs. Its major disadvantage is the unpredictability of the garbage collection interludes, which makes it hard to design systems to meet real-time requirements.

In this section we survey an architecture and algorithms designed to overcome the hesitancy of the Mark-Scan algorithm.

The Architecture

The architecture consists of two processors, called the *mutator* and the *collector*, working in parallel and sharing the same the workspace (which is organised as a heap of cells). The mutator does all the 'useful' work, modifying the connectivity of the data structure and the values of data fields within that structure. The collector is solely responsible for identifying and re-cycling garbage cells. Its algorithm is based on Mark-Scan but runs continuously in two phases. In the first phase active cells are identified and marked; in the second phase the heap is scanned and cells that are known to be certain garbage are returned to the free-list. The collector makes no change to shape of the heap other than to link garbage cells into the free-list. It does, however, use and make changes to a mark field within each cell. These mark fields are also used by the mutator; indeed, they provide all the necessary communication between the two processes.

Interference between collector and mutator should be minimal[1]. Any overhead on the activity of the mutator required for co-operation should be kept as small as possible. The activity of the mutator should impair the collector's ability to identify garbage as little as possible. Every garbage cell must be appended to the free-list eventually.

Dijkstra's algorithm used colours to represent the status of cells. The mutator informs the collector that the connectivity of the graph has been altered by changing the colour of a cell.

Dijkstra's Algorithm

Dijkstra's on-the-fly garbage collection algorithm painted cells one of three colours to indicate their status. [*Black*] cells are recognised by the collector as in use. [*White*] cells are seen as garbage by the collector during mark-scan. [*Gray*] cells represent uncertain status. This is the way the mutator will tell the collector it has changed the connectivity of the graph.

Mutator's Instruction Set

The mutator's three basic operations, `New`, `Copy`, and `Delete`, are assumed to be atomic. Dijkstra omits all other details of the mutator's program as well as the (straightforward) details of the pointer manipulation required to implement the free-list. A further abstraction is to ignore the synchronisation that must be done, for instance, if the mutator attempts to remove a node from an empty free-list

In the definition of `New` and `Copy`, whenever the mutator increases the connectivity of the graph it must first 'shade' the target cell to inform the collector that this cell's linkage has been altered.

Collector's Instruction Set

The collector is the processor in charge of the mark-scan and consequently responsible for returning free cells to the free-list. Active cells are marked until there are no gray cells left. The heap is then scanned, returning garbage cells to the free-list.

Unlike the Mark-Scan algorithm [41], the free-list is not necessarily empty at the start of a garbage collection cycle. The collector must mark the free-list as well as all active cells to ensure that the scanning phase does not attempt to add cells onto the free-list if they are already there.

[1] Some mutator overhead is inevitable, because some communication is needed from mutator to collector to indicate that the connectivity of the data structure has altered.

The marking process, `mark`, blackens all cells that are reachable from its starting points (root and the free-list). In effect it treats the heap as a circular buffer: whenever it encounters a gray cell, marking is restarted from this cell. A gray cell may be due to the activity of `mark` or it may mean that the mutator has increased the connectivity of the graph and so the scanning phase, `scan`, should not take place yet. No cells in the free-list will be gray. `mark` continues until it completed a full tour of the heap without meeting any gray cells. Most of the marking techniques described [12] would suffice, provided that they were restarted whenever a gray cell was found. A notable exception is the Deutsch-Schorr-Waite pointer-reversal method which would render objects inaccessible to the mutator during the trace [54].

`mark` first shades the sons of the cell being marked. The shading of the sons and the blackening of the parent should be considered to be an atomic action.

Once the heap has been checked for gray cells, the whole heap is scanned sending white cells to the free-list and resetting black cells to white. The mutator may be active while the `scan` runs, continuing to gray cells after `mark` has successfully completed its check. Such cells are ignored by the collector for the time being. Any that are actually garbage will be collected in the next cycle. The collector now re-starts the mark-scan process.

Woodger's scenario

In general, coarse-grained parallel algorithms are less efficient than fine-grained ones as more synchronisation is needed. The algorithm as presented above required all mutator, and some marking, operations to be atomic. Dijkstra and his colleagues made the granularity of operations finer and were led to the erroneous situation known as Woodger's scenario, in which if for any reason the mutator suspended activity, colour information could be changed by the collector and the synchronisation between mutator and collector would be lost.

The natural solution to this problem is to require the mutator to link cells to the graph first and then assign colour information. The collector is unchanged.

Dijkstra *et al.* presented an informal proof of the correctness of the on-the-fly algorithm in this section. Gries advocated the use of formal methods as a safer way to prove the correctness of parallel algorithms [22]. He used a proof method developed by Owicki and himself [46] to prove the on-the-fly algorithm correct.

The Kung-Song Algorithm

Several other algorithms have been developed, either based on or similar to Dijkstra's. Kung and Song developed a version based on Dijkstra which used a fourth colour, *off-white*, for cells on the free-list [32]. This extra colour meant that it was not necessary to mark free-list. Like Steele, Kung and Song also used a stack[2] for marking. The cost of their marking algorithm is proportional to the number of cells in use (rather than the size of the heap). The disadvantage is that space must be found for this dequeue.

[2] Actually an output restricted dequeue.

Ben-Ari's Algorithms

Ben-Ari looked for an on-the-fly garbage collection algorithm with a simpler proof than that of Dijkstra's [5, 6]. His new algorithm was then used to develop other algorithms that might be significantly better in practice.

His algorithm uses only two colours for marking, black and white, but requires an extra pass over the heap before the appending phase can start to return white cells to the free-list. On the other hand it is robust with respect to the seemingly innocent variation that introduced the bug discovered by Woodger. The mutator's instruction set is similar to that of Dijkstra: the only difference is that when new links between cells are made, the target cell is painted *black*. Whether the cell is painted before the link is made or after is immaterial.

Ben-Ari's `mark` is similar to Dijkstra's but omits the checking part: it simply propagates blackening from the root cells (including the free-list). As soon as the number of black cells has been observed not to have increased, the heap is scanned and white cells are returned to the free-list.

Lamport's Multi-Processor Architecture

The previous architectures used just two processors: one for the mutator and one for the collector. Leslie Lamport generalised the Dijkstra algorithm to use multiple processes for both mutation and garbage collection [33]. He also parallelised the sequential collection algorithm. The cost to be paid is an increase in synchronisation overheads between processes.

The same restriction that was placed on a single mutator must be placed on multiple mutators: an edge from an active cell cannot be changed to point to a garbage one. This implies some synchronisation mechanism between the mutators that enforces a partial ordering on their operations. The partial ordering must be sufficient to guarantee that the mutators correctly execute some sequential mutator algorithm. Mutators must also synchronise removal of nodes from the free-list. Lamport suggests using multiple free-lists to reduce delays: this can be implemented without difficulty. Parallel marking is done by partitioning the heap into (not necessarily disjoint) sets, P_i. Cells in one set P_i may point to cells in another set, P_j. For simplicity each marker process can be thought of as marking exactly one of these sets. Each marking process is essentially the same as Dijkstra's `mark`. The only difference is that when a gray cell is discovered by one of the markers, Lamport must restart *all* of the markers, not just the one that discovered the cell. Each `scan` process is the same as in the sequential algorithm, but operates upon one partition. To run marking and scanning concurrently, Lamport pipelined these phases so that the $(i+1)^{th}$ execution of the marking phase runs concurrently with the i^{th} execution of the scanning phase.

In Dijkstra's and Lamport's algorithms each time a grey node is found the marker resets itself to the start of its heap. In [44] Newman, Stallard and Woodward suggest that if a link from a black to a white node is made then the mutator shades the white node and resets 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 nodes.

A hybrid Steele-Lamport multiprocessor architecture is presented in reference

[45] with the aim of reducing Steele's communication bottleneck of having a single stack and also avoiding Lamport's repeated marking. The hybrid algorithm makes the stack local and uses colours to control the status of nodes.

Steele's Algorithm

The first published architecture for on-the-fly garbage collection was Guy Steele's Multiprocessing Compactifying Garbage Collection algorithm published in [58] (see also [59]). Although widely referenced Steele's algorithm never became as popular as Dijkstra's algorithm. The reason for that is, in our opinion, the thorough presentation and considerable level of detail taken by Steele. His paper included descriptions of compaction, parameter passing mechanisms and synchronisation, as well as mutator-collector garbage collection. If we ignore these complications, Steele's on-the-fly architecture is more simple. The algorithm uses a mark-bit in each cell and a stack. The stack (rather than the third, grey, colour used by Dijkstra) is used for communication between the mutator and the collector.

Steele uses a stack to ensure that all reachable cells are marked. Entries in the stack are added by the mutator when it alters the connectivity of the graph, when it initiates a collection cycle, and as it reaches white (unmarked) cells. During the marking phase items are popped from the stack, and their children are pushed back onto the stack where necessary, until it is empty. The collector checks the stack after each mark phase to see if it is empty. If so the mark phase is finished and the collector starts with the scan phase. Otherwise mark is called again on each of the entries on the stack.

Steele's algorithm also compacts the heap by relocating active cells into one end of the heap. There are several methods that might be used for compaction (see [12]). However, a multiprocess architecture raises extra complications. New objects may be being created while old ones are being relocated. This makes things awkward for a copying collector. Sliding collectors may destroy occupied cells before all references to them have been updated. For these reasons, Steele chose the 'two pointer' scheme for compaction (see [25, 53, 31]), that updates any pointers to relocated objects (for instance, any used by the parameter passing mechanism). To deal with variable-sized objects, Steele assumes that heap memory is organised into *spaces* of homogenous objects, and that each space is relocated separately. scan is the same as in Dijkstra's algorithm. Gries' optimisation [23] to Dijkstra's algorithm yields an algorithm similar to Steele's uncompactifying algorithm.

The use of more than one marker process was also suggested by Steele [58]. All markers add and remove nodes from a shared stack with suitable locking to avoid conflicting access. According to [45] this locking causes the stack to become a bottleneck because it is frequently accessed by the markers, and the processes will therefore spend a significant amount of time waiting for access to the stack. To solve this problem Newman and Woodward [43] suggest the stack to be replaced by a list of *subroots*. Markers remove a node from the subroot list and commence marking the subtree emanating from it. Whenever a node with two or more unmarked successors is found the marker will add one or more of the unmarked successors of the current node to the subroot list. Unfortunately this method may not terminate in the case of some cyclic structures.

Crammond's Algorithm

Morris' algorithm [42, 12] is a popular choice for garbage collection in sequential Prolog [4, 14, 2], systems since it is both space efficient and retains the order of cells on the heap; i.e., it is a *sliding* compaction algorithm. This feature allows the heap to be reclaimed through backtracking as well (refer to [2] for a detailed description of Morris' algorithm applied to a WAM based Prolog). The main problem with compaction is not in calculating the final location of a cell but rather in finding and updating all cells pointing to this cell. The Morris algorithm obtains this by forming relocation chains as the heap is scanned which can be updated once the final destination of a cell is known. It makes use of two extra bits in each cell on the heap (one for marking and the other for relocation chains) and consists of three phases. First all active cells in the heap are marked. Then a *downward* scan is performed (top-to-bottom) and all downward pointers are relocated. Finally, the heap is re-scanned from bottom to top relocating all *upward* pointers.

In Crammond's architecture [14] the shared-memory is divided into small heaps of the same size. Each processor manages a heap and a stack. Marking distinguishes between internal and external pointers to the heap. External pointers are sent to the stack, which at the end of the marking phase stores all external pointers to the corresponding heap. The stack is used in the compaction phase, in which a parallel version of Morris' algorithm is adopted. When garbage collection is needed processors suspend computation and synchronise. They must also synchronise after the marking phase and at the end of the compaction phase.

Crammond's algorithm is not concurrent. Its aim is to obtain efficient garbage collection by allowing all processors to work in a cooperative way, thus reducing the suspension time. A similar strategy for a non-compacting algorithm was devised by Boehm-Demers-Shenker [8].

Copying Architectures

Copying garbage collection is widely used in uniprocessors and is particularly suited for machines with virtual memory [20, 11]. In this scheme the heap is divided into two contigous semispaces. During normal program execution only one of the semispaces is used. Space allocation happens linearly. When the current semispace is exhausted the user process is stopped and the collector copies live data (in the *from-space*) into the other semispace (the *to-space*). At the end of this process the names of the two semispaces are flipped and the execution of the user program is resumed.

One way to make the copying algorithm efficient in shared memory architectures consists in synchronising processors' activity and letting each of them copy parts of the heap simultaneously. Halstead in the Multilisp garbage collector [24] assumes that the number of active cells is small. He allows a processor to lock a cell that needs to be copied, copy it into the processor's to-space, set the forwarding address in the old heap cell and then unlock it.

Crammond's approach [14] divides the heap into as many smaller heaps as the number of processors, each of these consists of a from and a to-space. On running out of space all processors synchronise and start copying as in the sequential algorithm, but if a pointer to a cell in a non-local from-space is found a reference to the source cell is stored in a stack local to each processor. In a second phase, the references in the stack are used in finding the corresponding to-space address of a non-local pointer. After all processors have finished copying they are allowed to resume computation.

The Appel-Ellis-Li Algorithm

Baker modified the copying algorithm to avoid long pauses, making it suitable for real-time applications in uniprocessors [3]. When the to-space fills up the user processor is suspended, but then only the root objects are copied. The user process is re-started immediately. More reachable objects are copied incrementally from from-space to to-space, every time the user process allocates more objects. Thus every fetch and allocation is slowed down by a small, bounded amount of time. According to reference [1], in the absence of hardware support Baker's algorithm is not efficient, since a few extra instructions must be performed on every fetch. Brooks' variant [9] is intended to be efficient on stock hardware, but neither Baker's nor Brooks' algorithm is concurrent.

Appel and his colleagues [1] devised a concurrent version of Baker's copying algorithm for shared machines that use virtual memory support. Their main idea is to use virtual-memory page protections to detect from-space memory references by the mutator.

If the mutator runs out of space it suspends all the mutator threads. The collector scans any remaining unscanned objects, flips the role of the two spaces, copies the reachable objects from from-space, and resumes the mutator threads. The collector also sets the virtual-memory protection of the unscanned area's pages to be "no access". Whenever the mutator tries to access an unscanned object, it will raise a page-access exception. The collector fields the exception and scans the object on that page, copying from-space objects and forwarding pointers as necessary. Then it unprotects the page and resumes the mutator at the faulting instruction. To the mutator, that page appears to have contained only to-space pointers all along, and thus the mutator will fetch only to-space pointers to its registers.

The collector also executes concurrently with the mutator, scanning pages in the unscanned area and unprotecting them as each is scanned. The more pages scanned concurrently, the fewer page-access traps taken by the mutator.

Appel-Ellis-Li's algorithm relies on the virtual-memory hardware to provide an efficient, medium-grained synchronisation between the collector and the mutator. A flip suspends mutator operation and thus the existence of a large number of root objects may cause a rather high latency.

Le Sergent and Barthomieu [55] describe a similar copying garbage collecting scheme for virtually shared memory architectures.

Generational Copying

In many applications in computer science, independently of languages or programs, one can observe the fact that most objects live a very short time, while a small percentage of them live much longer [34, 61, 57, 63, 15, 26]. This means that most objects that survive one garbage collection tend to live all computation long. Thus, copying these objects is wasted computational effort. Generational copying in uniprocessors [34] avoids much of this repeated copying by segregating objects into two or more areas by age, and scavenging areas containing old objects less often than the younger ones. Objects in younger areas that survive a few scavenges are moved a few scavenges to older areas to keep the copying costs down.

Sharma and Soffa [56] describe a way of introducing generations to the Appel-Ellis-Li algorithm. Their simulation results show that when compared against a parallel copy collection algorithm [1], the parallel generational collector performs better in the case of programs with larger amounts of of longer-lived cells. For these programs, the parallel generational collector performed up to 67% less copying than Appel-Ellis-Li's algorithm and reduced elapsed times up to an additional 12%; corresponding reductions in mutator overhead were also observed.

Similar work to Sharma-Soffa's was presented by Röjemo in [51], in the context of implementing lazy functional languages, in which a reduction of garbage collection time of almost 20% was observed.

Reference [8] presents a generational mark-scan algorithm that makes use of page locking mechanisms.

Parallel Reference Counting

Reference Counting is a simple memory management technique [13, 12]. It consists of storing in each data structure the number of external references to it in a counter. The count of newly allocated cells is one, copying references increments the count of the target cell, and deleting decrements the count, testing for zero, in which case the cell is recycled by being sent to the free-list.

Standard reference counting has the drawback of not being able to recycle cyclic structures, as their count never fall to zero [40]. Several uniprocessor algorithms present a solution for this problem in the context of implementing Lisp and functional languages [21, 7, 28]. General cyclic algorithms have also been proposed, but they were either incorrect [10], non-terminating in pathological cases [52] or extremely expensive to implement [48, 60]. A simple and efficient algorithm for cyclic reference counting was proposed and optimised by Lins and his colleagues [39, 35, 38].

In this section we present algorithms based on reference counting.

The Kakuta-Nakamura-Iida Architecture

In this architecture each cell has fixed size, two reference counts and a one-bit tag. The tag bit is a mark field for tree traversals. The first reference counter stores the number of external references to a cell, while the second reference counter is used in the detection of cyclic structures. The mutator has direct access to the free-list. Only the collector accesses the reference counts and the mark-bit.

For communication between the mutator and the collector two queues are used. Every mutator operator causes communication between processors and the mutator queues instructions and respective pointers. The collector dequeues instructions and pointers and executes the operation requested by the mutator. This causes a communication overhead and this algorithm is far too complicated to provide an efficient implementation.

Parallel Cyclic Reference Counting

A much simpler approach to reference counting in shared memory architectures is reported in [36]. Here the workspace is accessible directly by the *mutator* and the *collector*.

In case of simultaneous access from both processors to a given cell semaphores are used such as to guarantee that the mutator will have priority over the collector. There is also another shared data structure: the *Delete-queue*, which is organised as a FIFO. The mutator is only allowed to push data onto the Delete-queue. Conversely, the collector is only allowed to dequeue data from the Delete-queue. The mutator has two registers called *front-free-list*, which stores a pointer to the head of the free-list, and *front-del-queue*, which stores a pointer to the front of the Delete-queue. The collector has two registers called *back-free-list*, which stores a pointer to the last cell in the free-list, and *back-del-queue*, which stores a pointer to the back of the Delete-queue.

For the sake of simplicity Lins follows Dijkstra and his colleagues [18] and his description ignores the synchronisation that must be done when the mutator attempts to remove a node from an empty free-list or the collector tries to get a reference from an empty *Delete-queue*. These situations should happen infrequently and any convenient synchronisation primitive can be used.

`New` tests if there are free cells on the free-list. If so, it links it to the graph and adjusts the free-list accordingly. References are copied in the same way as in the standard reference counting algorithm. When pointers are deleted a reference to the target cell is pushed onto the Delete-queue.

The collector is the processor in charge of the deletion of pointers and feeding free cells into the free-list. The main routine in the collector runs forever as the kernel of the operating system, and operates by removing cells from the Delete-queue and testing their reference count. If there are no references to the cell, its descendants are examined recursively and the cell is appended to the free-list. Otherwise the reference count is decremented and Lins' tricolor local mark-scan algorithm is invoked [39]. First the transitive closure of the cell is painted red and have their count decremented. Then a local scan searches for external pointers to the subgraph under inspection. If any are found the subgraph below this point is painted green and have the reference count of the cells visited increased, to take into account the internal pointers within the subgraph (which had been set to zero in the first phase). Finally the garbage cells (non-green) are are linked to the free-list. The local mark-scan algorithm is explained in detail in [39, 35].

The Lazy Mark-Scan Algorithm

An important optimisation of the algorithm above is introduced in reference [35]. In this new algorithm the mark-scan phase is performed lazily, i.e. whenever the free-list is empty. The lazy algorithm uses a queue as an extra control structure to avoid performing the local mark-scan every time a pointer to a cell with multiple references is deleted. Instead a reference to these cells is placed on the control queue. These cells are painted *white*.

The collector only analyses the control queue when the Delete-queue is empty. Cells are popped from the front of the control queue and their colour is tested. If it remains white then it is still not clear wheather the last pointer to a cycle has been deleted and so a local mark-scan is performed. (Note that a cell painted white and pushed onto the control queue 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 queue.)

A Multi-Processor Architecture

Reference [37] presents a generalisation of the architecture in the last section to work with any number of mutators. Following Lamport [33], 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 mutator i has started an operation before a mutator j then operations will take place according to this precedence. 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.

Synchronisation 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.

Instead of pointing directly to the front of the Delete-queue, now each processor will keep a reference to an external register which points at the front of the Delete-queue, and similarly for the front of the free-list.

The instruction set for the mutators is the same as we had before with only one mutator. In this architecture all collectors are synchronised in such a way as to allow all of them to run each phase of the mark-scan simultaneously. The control strategy for synchronisation is such that when one of the collectors starts to mark-scan all the other collectors can do is either to finish or suspend their operation and mark-scan also.

Summary

The table below summarises the algorithms presented according to its philosophy, the number of processors, whether mutator and collector activity is concurrent or not, the way mutator and collector communicate, the space overhead, the most important references to it and to uniprocessor algorithms on which it is based, and finally some general comments.

Algorithm	Config.	Conc.	mut-col communication	space overhead	Refs.	Uniproc.	Comments
			Mark-Scan				
Dijkstra et al.	1mut. 1 col.	yes	non-direct 3 colours	2 bits	[18, 22]	[41, 31]	best known alg.
Kung-Song	1mut. 1 col.	yes	stack 4 colours	2 bits stack	[32]	[41, 31]	optimises [18]
Ben-Ari	1 mut. 1 col.	yes	2 colours	1 bit	[5, 6]	[41, 31]	optimises [18] extra scan
	1 mut. 1 col.	yes	3 colours	2 bits	[5, 6]	[41, 31]	optimises [18] similar to [19]
	1 mut. 1 col.	no	3 colours	2 bits	[5, 6]	[41, 31]	optimises [18] incremental
Ehn	1mut. 1 col.	yes	3 colours	1 bit	[19]	[41, 31]	similar to [5]
Gries	1mut. 1 col.	yes	2 colours stack	1 bit stack	[23]	[41, 31]	similar to [58]
Lamport	n-mut. p-col.	yes	3 colours	2 bits	[33]	[41, 31]	generalises [18]
Steele	1-mut. 1-col.	yes	stack	1 bit stack	[58, 59]	[41, 31]	compactifying
	n-mut. p-col.	yes	global stack	1 bit stack	[58, 59]	[41, 31]	communication bottleneck
Newman et al.	n-mut. p-col.	yes	3 colours	2 bits	[44]	[41, 31]	optimises [33] phys.ord. scan
	n-mut. p-col.	yes	4 colours local stacks	2 bits p stacks	[44]	[41, 31]	hybrid of [58] and [33]
Crammond	many proc.	no	synchronises processors	1 bit stack	[14]	[42]	sliding Prolog impl.
Boehm et al.	many proc.	no	synchronises processors	1 bit	[8]	[34]	generational
			Copying				
Halstead	many proc.	no	synchronises processors	1 bit stack	[24]	[20, 11]	Multilisp impl.
Crammond	many proc.	no	synchronises processors	1 bit stack	[14]	[20, 11]	
Appel et al.	1 mut. 1 col.	no	direct page-lock	1 bit	[1]	[3, 9, 1]	real-time virtual mem. O.S.
Sharma-Soffa	1 mut. 1 col.	no	direct page-lock	1 bit	[56]	[1, 34]	generational, based on [1]
Röjemo	1 mut. 1 col.	no	direct page-lock	1 bit	[51]	[1, 34]	similar to [56] based on [1]
			Reference Counting				
Kakuta et al.	1 mut. 1 col.	yes	2 stacks	2cnt.1bit 2 stacks	[30]	[13, 40]	communication bottleneck
Lins	1 mut. 1 col.	yes	stack 3 colours	1cnt.2bits 1 stack	[36]	[39, 35]	
	n-mut. p-col.	yes	stack 3 colours	1cnt.2bits 1 stack	[37]	[39, 35]	generalises [36] based on [33]

Conclusions

We have presented most of the algorithms for dynamic memory management in shared memory architectures. Reference [38] presents a comprehensive description of these algorithms.

Acknowledgements

This work was sponsored by CNPq (Brazil) grants 40.9110/88-4 and 80.4520/88-7.

References

[1] A.W. Appel, J.R. Ellis, and K.Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[2] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, 1988.

[3] H.G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[4] Y.Bekkers, O. Ridoux, and L. Ungaro. A survey on memory management for logic programming. In *IWMM'92*, LNCS 637,Springer Verlag, 1992.

[5] M.Ben-Ari. On-the-fly garbage collection: new algorithms inspired by program proofs. *Automata, languages and programming.*, 14–22, Springer-Verlag, 1982.

[6] M.Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Trans.Prog.Lang. & Syst.*, 6(3):333–344, July 1984.

[7] D.G. Bobrow. Managing reentrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[8] H-J.Boehm, A.J. Demers, and S.Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[9] R.A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. SLFP'82, 256–242, 1984. ACM.

[10] D.R. Brownbridge. Cyclic reference counting for combinator machines. FP&CA'85, Springer-Verlag.

[11] C.J.Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

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

[13] G.E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.

[14] J.Crammond. A garbage collection algorithm for shared memory parallel processors. *International Journal of Parallel Programming*, 17(6):497–522, 1988.

[15] J.DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DECSRC, Palo Alto, California, August 1990.

[16] E.W. Dijkstra. Notes on a real time garbage collection system. From a conversation with D.E.Knuth (private collection of D.E.Knuth), 1975.

[17] E.W. Dijkstra, L.Lamport, et al. On-the-fly garbage collection: An exercise in cooperation. *LNCS 46*, Springer-Verlag, 1976.

[18] E.W. Dijkstra, L.Lamport, et al. On-the-fly garbage collection: An exercise in cooperation. *CACM* 21(11):965–975, November 1978.

[19] L. Ehn. A contribution to the increase of efficiency of on-the-fly garbage collection. *Computers And Artificial Intelligence*, 8(1):83–91, 1989.

[20] R. Fenichel and J. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, Nov. 1969.

[21] D.P. Friedman and D.S. Wise. Reference counting can manage the circular environments of mutual recursion. *Inf Process. Lett.*, 8(1):41–45, Jan. 1979.

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

[23] D.Gries. On believing programs to be correct. *Communications of the ACM*, 20(1):49–50, Jan. 1977.

[24] R.H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. L&FP'84, ACM, 1984.

[25] T.P. Hart and T.G.Evans. Notes on implementing LISP for the M–460 computer. *The Programming Language LISP: Its Operation and Applications.*, 191–203, 1964. Information International, Inc.

[26] B.Hayes. Using key object opportunism to collect old objects. In *OOPSLA91*, pages 33–46. ACM, October 1991. Phoenix, Arizona.

[27] T.Hickey and J.Cohen. Performance analysis of on-the-fly garbage collection. *Communications of the ACM*, 27(11):1143–1154, Nov. 1984.

[28] R.J.M.Hughes. Managing reduction graphs with reference counts. Departmental Research Report CSC/87/R2, University of Glasgow, March 1987.

[29] R.E.Jones and R.D.Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, 1996. ISBN 0 471 94148 4.

[30] K.Kakuta, H.Nakamura, and S.Iida. Parallel reference counting algorithm. *Information Processing Letters*, 23(1):33–37, 1986.

[31] D.E. Knuth. *The art of computer programming*, volume I: Fundamental algorithms, chapter 2. Addison-Wesley, Reading, Ma., 2nd edition, 1973.

[32] H.T. Kung and S.W. Song. An efficient parallel garbage collection system and its correctness proof. *IEEE Foundations of CS*, 120–131. IEEE, 1977.

[33] L.Lamport. Garbage collection with multiple processes: An exercise in parallelism. *ICPP'76*, 50–54, 1976.

[34] H.Lieberman and C.Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, 26(6):419–29, 1983.

[35] R.D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, vol 44(4): 215-220, North-Holland, December 1992.

[36] R.D. Lins. A shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 34:31–35, September 1991.

[37] R.D. Lins. A multi-processor shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 35:563–568, 1992.

[38] R.D.Lins and R.E.Jones. Dynamic Memory Management: Algorithms for Garbage Collection. to be published by *John Wiley & Sons*.

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

[40] J.H.McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[41] J.McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[42] F.L. Morris. A time-and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–5, 1978.

[43] I.A. Newman, R.P. Stallard, and M.C. Woodward. Performance of parallel garbage collection algorithms. *Computer Studies*, 166, September 1982.

[44] I.A. Newman, R.P. Stallard, and M.C. Woodward. Improved multiprocessor garbage collection algorithms. In *IC Parallel Processing*, 367–368, 1983.

[45] I.A. Newman, R.P. Stallard, and M.C. Woodward. A hybrid multiple processor garbage collection algorithm. *Computer Journal*, 30(2):119–127, 1987.

[46] S.Owicki and D.Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[47] S.Owicki and L.Lamport. Proving liveness properties of concurrent programs. *ACM T.Programming Languages and Systems*, 4(3):455–495, 1982.

[48] E.J.H. Pepels, M.C.J.D. van Eekelen, and M.J. Plasmeijer. A cyclic reference counting algorithm and its proof. TR-CS 88–10, University of Nijmegen, 1988.

[49] F.J. Pollack, G.W. Cox, et al. Supporting Ada memory management in the iAPX–432. pages 117–131. SIGPLAN Notices (ACM) 17,4, 1982.

[50] S. Ramesh and S.L. Mehndiratta. The liveness property of on-the-fly garbage collector — a proof. *Information Processing Letters*, 17(4):189–195, 1983.

[51] N. Rojemo. A concurrent generational garbage collector for a parallel graph reducer. *PIWMM'92*, LNCS 637, Springer Verlag, 1992.

[52] J.D. Salkild. Implementation and analysis of two reference counting algorithms. Master's thesis, University College, London, 1987.

[53] R.A. Saunders. The LISP system for the Q–32 computer. *The Programming Language LISP: Its Operation and Applications.*, 220–231, Inf.International Inc, 1964.

[54] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *CACM*, 10(8):501–506, Aug. 1967.

[55] T. Le Sergent and B. Barthomieu. Incremental multi-threaded garbage collection on virtually shared memory architectures. *IWMM'92*, LNCS 637, Springer Verlag, 1992.

[56] R.Sharma and M.L.Soffa. Parallel generational garbage collection. In *Proceedings of OOPSLA '91*, pages 16–32, 1991.

[57] R.A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, 1988. Tech. Rep. CSL–TR–88–351.

[58] G.L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[59] G.L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.

[60] S.J. Thompson and R.D. Lins. Cyclic reference counting: a correction to Brownbridge's algorithm. unpublished notes, 1988.

[61] D.M. Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[62] P.L. Wadler. Analysis of an algorithm for real-time garbage collection. *Communications of the ACM*, 19(9):491–500, September 1976.

[63] B.Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. *ACM Lisp and FP*, June 1990.