Garbage Collection in Parallel Environment

Filipe Varjão

CIn, UFPE, Recife, Brazil
October, 2014

Abstract. For dynamic memory management (*Garbage Collection* - GC), we have a lot discussion for our choices, in this paper, we propose a survey about garbage collection in parallel environment and describe several techniques the state of the art. The main goal of this work will be around parallelization of sequential methods (Mark-and-Sweep and Reference Counting), different issues in each environment (Shared variable access in shared memory systems and Disjoint address spaces in distributed memory systems) and scheduling in both environments involves stopping application threads during tracing (Long pauses avoided by incremental collection and improves performance in SPMD programs since application has frequent global synchronizations).

1 Introduction

Today garbage collection is an important part of the memory management system of many modern programming languages imperative as well as declarative.

By Jones and Lins [15], "*Garbage Collection* is precisely this – the automatic management of dynamically allocated store."

We can find distinguish techniques for some authors between direct techniques, such as reference counting, and direct, tracing techniques. There are three ways in which storage can be allocated: static allocation, stack allocation and heap allocation. The mechanisms common to all technics is identify the live objects in the memory storage allocated, reclaim resources held by dead objects and periodically relocate live objects.

These several techniques are found to mount an optimized GC, but why we can find so many techniques and why not use one single technique to apply for collectors?

So, GC has a reputation for placing a large overhead on the execution of programs, but GC was an obvious scapegoat, because some implementations of some languages often ran slowly for reasons other than GC, such as less efficient parameter passing mechanisms, or support for higher order functions or delayed evaluation of expressions. Modern techniques have reduced GC overheads substantially to the point where even languages used for systems programming are supported by garbage collection.

However another limiting factors becomes important to understand the performance of several approach of GC.

In this work we address the main constraints and difficulties in distributed garbage collection, propose a taxonomy to classify some works, and presents an overview of strategies to parallel generational garbage collection.

We will looking for describe some approaches to make dynamic memory management in parallel environment, thus we believe that the first part to show how you can do it is define the difference between parallel and distributed system.

Probably always you hear something about parallel systems you will hear about distributed systems too, and contrariwise.

A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs) [30].

For a parallel system everything is multiples tasks, but maybe we can change perfectly in the Tenenbaym's words distributed to parallel, and the sentence keep making sense. Every parallel system is distributed, because when you execute some job in parallel you need separate this job in tasks, not necessarily equals tasks but you have this distinction, in the same time. On other hand, when you have a distributed system and many jobs or tasks to do, you not necessarily need execute at the same time, but rather on distinct node. This similarities keep together the parallel and distributed areas.

Other common case of similarities with parallel is concurrency. Concurrency is closer of parallel than distributed systems, although we can say that any parallel system is not a concurrency system and any concurrency system is not a parallel system. The difference is for a concurrency system all processors/job/task competes to work with scheduler, i.e. are executing simultaneously but not at the same time.

About the GC we can consider concurrent collection when the GC competes with another process and jobs of the system, and parallel collection while the all system paused to GC work, using all cores to clean up the memory, may still be able to reduce overall run-time and stop-time. To Tene [32] a parallel collector uses multiple CPUs to perform GC, maybe we can assimilate the parallel collector with stop-the-world and concurrent collector with incremental.

The literature categorizes GC as incremental, concurrent or parallel, but a parallel GC may also be concurrent. In the next sections we will list and explore the techniques and approaches used in some works.

2 Dynamic memory management

Throughout history of dynamic memory management appear three main techniques that whose variants are used today, either directly or indirectly in hybrid systems, collectors by:

- Mark-Sweep
- Copy
- Reference counting

The reference counting is a direct method, thus the main goal of this technique is identify the moment of the last reference lost the link witch the object. It is a sample idea, basically each object has a flag to indicate a number of objects linked with this, i.e. the reference counting. When the last object lost the reference the automatic memory management can do the cleanup. It is a natural increment technique.

On parallel environment each processor increments a reference count of an object when it sends a reference to another processor and decrements a reference count of an object when it no longer uses a remote reference to the objects. Parallel collectors take a new approach to the problem or reducing collection pause time, we know that parallel collector imposes some overhead cost on the system, but it avoid a long pause times stop-the-world collection to manage the memory.

The overhead of maintaining reference counts is high on conventional hardware. This has made reference counting a less attractive option for storage management than tracing methods, RC (reference counting) still require space in each cell to store the RC, in the theoretically worst case [15]. Zhao et al. [35] and Paramod [27] say that reference counting is not popular in modern programming languages.

For this reasons is hard work find the new approaches with use RC in parallel environment, but Kakuta et al. [17] propose a parallel RC algorithm and discuss some problems on parallel operations, the collector can reclaim garbage cells linked in a circular list using a method of a modified marking algorithm, they use two reference counts and two queues.

Boehm et al. [7] for other hand, consider only tracing collectors to describe the mostly parallel GC.

2.1 Mark-Sweep

The first algorithm to recycle dynamic memory was tracing technique, also called by mark-scan. Developed by J. McCarthy [22] in 1960 to Lisp language [24].

This technique is based in tracing where objects are access from the set root by tracing, following the references on pointers. Thus mark-sweep is based in separate from the tracing process the objects on the heap on actives objects and inactive objects, after this classification the inactive objects are cleanup of the memory.

The classifications occur concurrently the execution of system, when a new allocation occur and the memory management notify that cannot do it, because do not have a free memory space, the cycle of collection started, and the user process is temporarily suspended [15].

Taura and Yonezawa [31] and Jones and Lins [15] describe a natural extension to mark-sweep collector called global collector. Upon a global collection, each processor starts marking from its local root and traces remote references by sending messages to traverse the entire objects graph, objects that are not marked when marking finishes are reclaimed.

The global collector requires global coordination of all processors to detect the termination of this global marking, which has been deprecated by many studies that naive global mark-sweep is worse and less scalable than reference counting [34]. Yamamoto et al. [34] also show us a comparing reference counting and global mark-sweep on parallel computers, they use three parallel applications (Bin-tree, Puzzle and N-body) in ABCL language [1].

We can see that [35] measure for each parallel application the distribution of GC and time spent for both reference counting and global mark-sweep but they use a samples approaches of GC to both cases, i.e., using a simple GC to parallel environment and not a specific GC to expect a improve of GC to parallel tasks.

The results confirmed that a reference counting could perform much worse than a global mark-sweep.

Boehm, Demers and Shenker [8] say that sweep phase does not have a significant impact on GC pause times, and there is no reason to sweep the entire heap while the world is stopped waiting for the collection to complete. Thus they present a method for adapting the mark-sweep to trance-mark-and-sweep collector.

The collector propose by [8] splits the heap into blocks, each block contains only objects of a particular size, i.e. for small objects, the size of the block is a physical page.

The mark phase sets a bit for each accessible object, then queue pages for sweep, keeping a separate queue for each small object size. The Large object blocks are swept in large increments during allocations immediately following a collection. [8] reports that this requires very little CPU time, and does not force the data pages be become resident in physical memory.

2.2 Copy

The algorithm proposed by Marvin Minsky in 1963 [24] are recursive, thus this technique need much free space on the heap, although the collector start when the system do not have more free space. Cheney [9] propose a new version of copy collector that do not use recursion. All new algorithms based on copy collector are derivatives by Cheney [9].

Copy collector also is a tracing technique, like mark-sweep. Basically this algorithm the memory heap has to parts, and just one part is used, when the first part reaches its limits the collector identify actives objects and make a copy to another second part of memory heap. The first part and second part called by from space and to space change their jobs. Each cell reachable from nodes is a copy by from-space for to-space, is necessary take care with the rest references in the process of copy, to avoid graph memory. The formal way to avoid this problem is use a forwarding address. For all object that is copied allowed to address for forward on from-space, indicating the new address on to-space, always that one cell on from-space is visited during the copy, the algorithm checks if has already been copied, otherwise the cell is copied for to-space and the address forward is stored.

Interesting viewpoint the address is storage before the real copy, this ensures the work of algorithm, the address forward is storage on the cell itself, and make unnecessary additional free space on the memory to work.

Finally all inactive objects are cleanup of memory, and the process keep working. The copying collector is invoked concurrently with the mutator, the collector copies all reachable objects residing in from-scape to a previously unused region of memory referred to as to-space. Links in to-space are updated to reflect the new locations of the objects [8]. Furthermore, if an object resides on a clean page then its copy has the correct contents. With the world stopped, we can then run following finishing operation to ensure that all reachable objects have been copied, and all copies contain the correct contents.

2.3 Reference Counting

The automatic management of dynamic memory, become a great need in all programming languages. Between the several techniques the reference counting become advantageous for many reasons. The fact of to be incremental, avoiding a full stop of users process (stop-the-world) to manage the memory.

The incremental nature of reference counting indicates the algorithm can be change to multiprocessor and parallel version, keep the same features. However, the problems with use of synchronizations between threads could void the efficiency gains obtained with the extension, invalidating the use of reference counting in multiprocessor and parallel environment.
Further of multiprocessor capacity, the new computers become increase your memory capacity. The memory space use for programs is increasing, but some techniques of GC has been performance dependent of your available space of memory, on the other hand, techniques based in reference counting have no such dependence, the performance of GC based in reference counting is proportional to length of memory occupied [15].

Such as mark-sweep algorithm, the free cells can be allocated are keeping in a structural list called by free list. Free cells are grouped in a free list, when a new cell is allocated, it is removed from free list and liked to re root, after that each new reference created or allocated to linked this cells, and increase your count. Similarly, each reference destroyed or allocation removed reduce 1 (one) value from the counting, if some counter arrived to zero this cell cannot be reached, i.e., can be return to free list again.

Bacon et al. [5] present a new algorithm for RC by improving on techniques for avoiding RC of stack variables, and reclaiming cyclic garbage. But this new improvements Bacon et al. tour explicit that the new algorithm is concurrent and not parallel but it is possible to parallelize the algorithm, partitioned by address, with different processors handling RC updates for different address ranges. To straightforwardly parallelize the reference count updates and use fetch-and-add operations to ensure atomicity on the RC word. The problem described by Bacon et al. [5] is that new all operations on the RC field will incur a synchronization overhead.

2.4 Comparing Techniques

After present how to work some principal techniques to GC, we will discuss each features, advantages and disadvantages mainly in parallel aspects.

Both technique that use tracing like mark-sweep and copy have two big advantages than reference counting. First is all cycles are naturally recovered, without any especial work to do it, second big advantage is any overhead is imposed operations with pointers. Bacon et al. [5] and Attanasio et al. [4] describe a parallel mark-sweep collector, each processor has an associated collector thread. Collection is initiated by scheduling each collector thread to be the next dispatched thread on its processor, and commences when all processors are executing their respective collector threads. The parallel collector threads start by zeroing the mark arrays for their assigned pages, and then marking all objects reachable from roots in mutator stacks.

Collector threads complete the collection process by yielding the processor, thus allowing the waiting mutator threads to be dispatched.

On the other hand, both approaches need complete stop of user system while the GC work, on the mark-sweep this process occur when all cells is marked on the heap. With increasing the size of memory heap, it has been found the big consume the mark-sweep of execution time on collector. If we look for a real-time system the big stop is unacceptable [15] and parallel system too.

In the parallel environment we can think in broke all process to tracing and mark cells, reducing the time to make all marking. Parallel collection improves overall GC throughput by better using modern multicore CPUs. The parallelism is achieved by giving each thread a set of the roots to mark and segment of the table of objects. Unfortunately the process of detect objects are not live leaves the tenured space with gaps. Some used memory where objects live, and gaps in between where objects used to live, is not helpful the application performance because it makes it impossible to allocate objects that are bigger than the size of the holes. All this tings turn high the cost to GC. Each active cell is visited during the phase of mark, in all cells of the heap memory. The performance of these approaches is proportional the memory size and the allocation of heap, the more occupied is the memory must collection will be occur.

More than copy collector, mark-sweep have another big advantage, the capacity of use all heap, since copy divides the heap memory to make copies only active objects.

Talking about copy collector the costs allocation are low comparing with another techniques, the fragmentation is eliminate by compression of actives objects. The problem is you need the double of free space for allocate some data, but is a fact that in parallel systems have a good location of data reference can contribute with the performance. Otherwise the copy collector algorithm have a big problem called motion sickness, all cells are copies and change the space memory, thus may cause difficulties mostly to parallel systems [28].

But in parallel environment maybe we can say that reference counting has a perfect approach, because the big dial to RC is distribute all management of memory, without stop all jobs to do it [15]. Levanoni [19] show us that most of cells not are so shared and spend a little time in use. The RC algorithm allow the immediately reuse of cells, avoid the pagination on disc. Soon we can see why RC can be perfectly used in parallel environments.

The most cost to keep using this approach is always run the counting concurrently all work of the system.

The next table we can see advantages and disadvantages of three techniques:

	Advantage	**Disadvantage**
Mark-Sweep	• Uses all memory • Just need a bit to object or cell • Do not have a cost to pointers operations	• Big stop cost to tracing and collect • Sweeps all address space of process, lost the location of reference • Fragments memory space
Copy	• Avoid fragmentation by copy objects • Shorter time to collection than mark-sweep	• Expensive for make copy objects • May cause problems for the system during the process of copy between semi-spaces • Motion sickness
Reference Counting	• Small steps interleaved with application computation • Dos not dependency of total size of heap memory	• Require stop all computation to work well, to be incremental algorithm • Cyclic references • Maintenance of reference counting • Changes in pointers • Manipulation of free list

3 Incremental Garbage Collection

Incremental GC or also called by real-time, it demands guarantees for the worst-case performance whereas generational collection attempts to improve the expected pause time at the expense of the worst case [15].

Real time systems demands that results be computed on time: a late result is as useless as an incorrectly calculated one. Such systems require worst-case guarantees rather than average-case ones, and these may have to be in the order of a millisecond. Indeed, many hard real-time systems demand guaranteed space bounds and even eschew the use of virtual memory. Memory management is not the only problem faced by real-time systems.

Other advances in hardware design such as super-scalar, pipelined architectures have also made performance less deterministic, Soft real-time systems prefer results to be delivered on time, but accept that a late result is better than no result at all. It is clear that many so-called real-time GC cannot meet realistic worst-case deadlines for many hard real-time applications. At best, they offer average-case pause times that are bounded by small constants. This is often adequate for interactive applications but is not hard real-time. We shall avoid calling such incremental algorithms “real-time” [15].

Traditional GC have two main aims: Firstly to identify an object that cannot be accessed or reachability by the program and secondly to reclaim the memory used. A separate part of the memory manager is responsible for allocating memory for new objects. Incremental parallel and concurrent are sometimes used inconsistently in the GC literature, Thomas [33] follow this clear terminology:

- Incremental: An incremental collector does small chunks of work and in between allows the mutator to resume. This differs from traditional tracing collectors which must finish a complete GC cycle or stop and forget the work they have done if the mutator needs to resume.
- Parallel: A parallel collector utilizes multiple threads simultaneously to do GC work. This can be at the same time as the mutator or when the mutator is stopped.
- Concurrent: A concurrent collector runs concurrently on a separate thread to the mutator. This means that on a multi-core machine the GC would run at the same time as the mutator.

Tricolour marking is used to reason about correctness of reachability GC algorithms. The object graph is represented with each node being one of three colours: black, gray and white, let’s present the definition of Jones and Lins [15]:

Black indicates that a node and its immediate descendants have been visited: the GC has finished with black nodes and need not visit them again.

Gray indicates that node mist be visited by the collector. Either grey nodes have been visited by the collector but their constituent pointers have not been scanned, or their connectivity to the rest of the graph has been altered by the mutator behind the collector’s back. In either case, the collector must visit them again.

White nodes are unvisited and, at the end of the tracing phase, are garbage. A GC cycle terminates when all reachable nodes are black. Thomas [33] propose two approaches to incremental GC. The first one use a mark-sweep to either lower the pause times for each incremental cycle by dividing the work calculation by the number of collector threads or that have found that the longest pause times this means we need to change our estimate of the number of rounds of work we will need to do. The second approach is incremental copy collector, in the single threaded version we did not have to worry about multiple collector threads scavenging the same or interconnected objects simultaneously.

Another good point that we can see at Thomas [33] work, is the use of MMTk [6] a framework for building GC and allows programmers to create high performance memory managers, there are already several collectors built with the MMTK these include the classic three collectors: mark-sweep, reference counting and copy more a generational/hybrid versions of them.

4 Generational Garbage Collection

By Jones and Lins [15], the generational strategy is to segregate objects by age into two or more regions of the heap called generations. Different generations can then be collected at different frequencies, with the youngest generation being collected frequently and older generations much less often, or even, in the case of the oldest generation, possibly not at all. In a sense, this is the dynamic automation at run-time of segregation into read-only, non-scanned and dynamic that we discussed above. For us maybe it is look like a merge of copy, mark-sweep and RC techniques.

But the number of generations used can vary, we can find in many researches the use of generational GC with incremental collection schemes but is not necessary, and is not dependency of this two collections. Accepting the weak hypothesis the most objects die young, generational technique concentrate their effort to reclaim storage on the youngest generation since it is there that most recyclable space is to be found. Furthermore, because older objects are promoted out of younger generations, CPU cycles can be saved by not having to copy these items from one semi-space to another, although it is still necessary to scan some older objects for pointers into younger generations [15].

The use to parallel environment is clearly seen on the Glasgow Haskell Marlow et al. [21]. Although in this specific paper Marlow et al. [21] extend the copy collector to a generational scheme, an extension that the block-structured heap make quite straightforward. It is not complex idea to understand if you think in all process to segregate objects by age. On GC of GHC the number of generations are defined for two standards gens, gen 0 and gen 1. In gen 0 we have the new objects, all object that survive of collection is promoted and old go to gen 1.

4.1 Generational copying collection

The generational collector exhibits several space advantages. Its pauses for garbage collection are shorter since it has less data to trace and copy at each collection, and the total volume of data moved throughout the entire program run is smaller.

In Marlow et al. implementation, the remembered set lists all objects that contain pointers into a younger generation, rather than listing all object is repeatedly mutated. The GGC (Generational garbage collection) has two forms. First is to reduce the overall cost of dealing with long-lived objects and thereby allow the collector to concentrate its efforts on young objects, where the rewards are likely to be greater. The second objective is to reduce garbage collection pause times to a level where they no longer disturb interactive users. Both goals are achieved by segregating objects by age, and by collecting older generations much less frequently than younger ones.

5 Parallel Programming Environments

The World War II started the history about of computation and on the rise of programming language is parallel the computation. Starting with COBOL [10] and FORTRAN Monitor [18].

Since the first computer looking for improve the computer performance, FORTRAN is used today to have an excellent performance to compute numeric operations, today FORTRAN run a most of cluster to provide weather forecast. After all computational revolution and currently now well established, parallel programming is the way to keep extracting all resources available on the machine architecture and improve the machine performance.

Some researches show different way and facilities to provide a parallel environment, MacDonald and Norman [20] list some features:

1. Detecting parallelism
2. Control of partitioning
3. Control of mapping
4. Abstraction from communication

MacDonald and Norma [20] tell us that DOACROSS is a first extensions to FO RTRAN that provide mechanisms for the programmer to specify such parallelism. Parallel programming environments which embody a more general abstraction, such as Linda that we will see next, suffer from the problem of regaining information about locality in order to map in such a way as to attain acceptable efficiency. In the case of parallelization of sequential does, the identification of parallelism and partitioning require to be solved as well as the mapping.

5.1 PVM

Parallel Virtual Machine, developed by Oak Ridge National Laboratory team at University of Tennessee and Emory University [26]. PVM is a software system that permits a network of heterogeneous UNIX computers to be used as a single large parallel computer. PVM define collection of serial, parallel and vector computers appears as one large distributed-memory computer [13].

PVM based application can be regarded as a collection of remote processes that interact with each other by exchanging messages during their execution.

5.2 HeNCE

Heterogeneous Network Computing Environment is a graphical tool for the development of parallel programs. HeNCE allows a parallel application to be written at a higher level than in the other environments, this is achieved by allowing the programmer to graphically design the interrelationships between pasts of the application, and explicitly assign the execution of such pats to different machines without any reference being made to how the data should be transferred between those pasts.

Actorspace is a computationally passive container of actors which acts as a context for matching patterns. GC an actorspace correspond to deleting it any contained actors and actorspaces will be removed from it. Conversely, when an actor is no longer reachable, and furthermore cannot potentially reach a reachable actor, a GC algorithm may be able to delete it.

5.3 P4

Portable Programs for Parallel Processors, developed at ANL [8], us a software package which allows the programming of a variety of machines including both shared and distributed memory architectures [3], both types of architecture can be used in a single application. One approach of p4 is exchange message with MPI [25], either synchronously or asynchronously.

A more comprehensive solution to the MPI object life cycle problem involves a limited from of GC for heavily contended object classes.

5.4 Posix Threads

Called by **Pthreads** is a standard to build light process (threads), basic assumption is that context switches among related processes are fast enough to handle each user-level thread by one kernel. Kernel processes can have various degrees of relationship [3].

Posix thread specification requires sharing of almost all resources. One of the goal for the library is to have low startup costs for threads. The Biggest problem time-wise outside the kernel is the memory needed for the thread data strictures, thread-local storage, and the stack.

5.5 OpenMP

OpenMP is a programming interface of shared memory, i.e. API specification for parallel Programming. Is a portable, scalable model with a simple and flexible interface

for developing parallel applications, clearly a specification that compilers can implement, relatively easy way to get moderate parallelism on shared memory [29].
Thus, OpenMP is a collection of compiler directives and library functions that are used to create parallel programs.

5.6 MPI

The Message Passing model of parallel computation has emerged as an expressive, efficient, and well-understood paradigm for parallel programming. We see before that many programming languages and tools use MPI [25].

MPI includes point-to-point message passing and collective operations, all scoped to a user-specified group of processes, make heterogeneous data conversion a transparent part of its services by requiring datatype specification for all communication operations.

5.7 Linda

Linda is a traditional parallel programming language Ahuja et al. [2], the Linda programmer is responsible for detecting parallelism and partitioning his application into tasks.

In distributed systems, storage management should not be left to be dealt with by users, including Linda, users are unable to know whether a memory cell is being used by others in the systems, as a matter of fact, if the task or reclaiming cell was left to users, the creation of dangling references would be common Gelernter [14]. We see before that GC is the process of searching and reclaiming unused memory objects automatically.
The idea behind the GC of Linda is maintain a data stricture (a graph) with all information necessary to a GC. Every Linda process is linked to a tuple space. The structure maintains the invariant that once an object is unreachable it cannot become reachable again, in others words, once an object becomes garbage it cannot became useful again, this invariant guarantees that no erroneous collection occurs Menezes and Wood [23].

An objects is garbage it is unreachable from the roots and cannot reach all of them, from time to time a traversal in the graph is made and all reachable nodes are marked, the garbage are those not marked at the end of the traversal. This algorithm based on mark-sweep is due to the possibility of cycles in the graph, and mark-sweep works well with cycles, should use a variation of the mark algorithm as proposed by Dijkstra et al [12].

5.8 CUDA

The Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model invented by NVIDA, It enables dramatic increases in computing performance by harnessing the power of graphics processing unit (GPU) [11].

Parallel computing extensions to many popular languages, powerful drop-in accelerated libraries to turn key applications and cloud based compute appliances. CUDA extends beyond the popular CUDA Toolkit and the CUDA C/C++ programming language. At its core are three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization.

6 Resolutions

After all listing of ways, techniques and different approaches to build or use a parallel GC we can see that not have a one right way to parallelize all systems, but such as have many forms to build a parallel system we have a many ways to improve your GC to response and manage the memory.

Now we list the common characteristics that we find on several parallel programming environments, thus make part of their GC:

- Computation models an application is a collection of asynchronous, concurrent sequential processes, interacting through messages exchanged during their execution time.
- Portability of source code across different architectures.
- The possibility of using geographically dispersed machines.
- Seamless support for mixed language applications.
- Explicit and manual partitioning and scheduling of an application.
- Support function-based distribution of processes.

References

1. ABCL – An Object-Oriented Concurrent Language, Access in Setember: http://web.yl.is.s.u-tokyo.ac.jp/pl/abcl.html (2014)
2. Ahuja S., AT&T Bell Laboratories, Carriero N., Gelernter D. “Linda and Friends”, IEEE computer vol. 19, no. 8, pp 26-34 (1986)
3. Andres, G. R. “Foundations of Multithread, Parallel, and Distributed Programming” Addison-Wsley p. 664 (2001)
4. Attanasio C. R., Bacon D. F., Cocchi A., Smith S. “A comparative evaluation of parallel garbage collectors.”In Proceedings of the Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science. Springer-Verlang, Cumberland Falls, Kentucky (2001)
5. Bacon D. F., Attanasio C. R., Rajan V. T., Smith S. E. “A Pure Reference Counting Garbage Collector” IBM T.J. Watson Research Center and Han B. LEE University of Colorado (2004)
6. Blockburn S. M., Cheng P. and McKinley K. S. “Oil and water? high performance garbabe collection in java with mmtk” In ICSE IV: Proceeding of the 26th Internantional Conference on Software Enrineering, pages 137-146, Washington, DC, USA, IEEE Computer Society (2004)
7. Boehm H. J., Demers, A. J., Shenker S.: Mostly Parallel Garbage Collection. SIGPLAN PLDI, 26(6):157-164 (1991)
8. Butler R. and Lusk E. “User’s guide to the p4 programming system” ANS-927 Argonne National Laboratory (1992)

9. Cheney C. J. “A nonrecursive list compacting algorithm” Commun, ACM, vol. 13, pp. 667-678 (1970)
10. Common Business Oriented Languge - COBOL, http://en.wikipedia.org/wiki/COBOL
11. CUDA, NVIDEA , http://www.nvidia.com/object/cuda_home_new.html
12. Dijkstra E. W., Lamport L. Martin A. J., Schoulten C. S., and Steffens E. F. M. “On-the-fly Garbage Collection: An Exercise in Cooperation” Communications of ACM, 21(11):966-975 (1978)
13. Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R. and Sunderam V.S. PVM users’s guide and rederence manual. Research Report ORNLM-12187 (1993)
14. Gelernter D. “Generative Communication in Linda” ACM Transactions on Programming Languages and Sustems, 7(1):80-112 (1985)
15. Jones R. B. and Lins R. D. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, Chichester, July (1996).
16. Kafura, D. Mukherji M. and Washabaugh D. M. Concurrent and Distributed Garbage Collection of Active Objects, IEEE PDS, (1995)
17. KaKuta K., H. Nakamura and S. Ilda. A parallel reference counting algorithm. Information Processing Letters, 23(1):33-37 (1986)
18. Larned R. A. “FMS: The IBM Fortran Monitor System” – IBM Corporation (1987)
19. Levanoni Y. and Petrank E. “An on-the-fly reference-counting garbage collector for java” ACM Trans. Program. Lang. Syst,. Vol. 28, no. 1, pp. 1-69 (2006)
20. MacDonald N. B. and Norman M. G. “Issues in Parallel Programming Enrivonments” BCS Parallel Processing Specialist Group Wrokshop on Abstract Machine Models for Highly Parallel Computers, 25-27 (1991)
21. Marlow S., Harris T., James R. P., Jones S. P., “Parallel Generational-Copying Garbage Collection with a Block-Structured Heap” ACM ISMM – Arizona (2008)
22. McCarthy, J. “Recursive functions of symbolic expressions and their computation by machine”, Connubucatuions of the ACM, vlo.3, pp. 184-195 (1960)
23. Menezes R. and Wood A. “Garbage Collection in Open Distributed Tuple Space Systems” In Proc. 15th Brazilian Computer Networks Symposium – SBRC (1997)
24. Minsky M. “A lisp garbage collector algorithm using serial secondary storage” Cambridge, MA, USA, Tech. Rep (1963)
25. MPI – The Message Passing Interface standard, http://www.mcs.anl.gov/research/projects/mpi/
26. National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov
27. Pramod G. J. A Principled Approach to Nondeferred Reference-Counting Garbage Collection, Techinal Report MSR-TR-2007-104 (2007)
28. Printezis T., Detlefs D. A generational mostly-concurrent garbage collector. In Hosking ACM Press (2000)
29. OpenMP, http://openmp.org/wp/
30. Tanenbaum A. S. and Renesse R. van, Distributed operating systems. ACM Computing Surveys, 17(4):419-470 (1985)
31. Taura K. and Yonezawa A. “An Effective Garbage Collection Strategy for Parallel Programming Languages on Large Scale Distributed-Memory Machines” In Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming – PpoPP (1997)
32. Tene G. “The Application Memory wall”, http://qconsf.com/sf2011/dl/qcon-sanfran-2011/slides/GilTene_TheApplicationMemoryWallThoughtsOnTheStateOfTheArtInGarbageCollection.pdf
33. Thomas P. “Incremental Parallel Garbage Collection” Department of Computing – Imperial College London (2010)

34. Yamamoto H., Taura K. and Yonezawa A. “Comparing Reference Counting and Global Mark-and-Sweep on Parallel Computers” In Lecture Notes for Computer Science – LNCS. Languages, Compilers, and RunTime Systems – LCR98, volume 1511, pp. 205-218 (1998)
35. Zhao Yio, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, OOPSLA ’09, pages 361–376 (2009).