Parallel garbage collection algorithms: a literature review

Junshuai Zhang(juz1)
Aberystwyth University

***Abstract*—Garbage collection is an indispensable part of modern programming languages. It affects the performance of each application. How to improve garbage collection, reduce the pause time is always a hot topic for so long. Meanwhile, in the last decade, multiprocessors have become a common architecture for computers. How to use multiprocessors to reduce the pause time of garbage collection naturally become a attractive field of programming language theory. In this paper, we first introduce what garbage collection is, why it is useful, and how it works. Then we will give a literature review about parallel garbage collection algorithms in the aspect of the fundamental approaches of garbage collection. We found a common pattern to parallelise garbage collection algorithms is to use techniques like work stealing, packets or blocks, and channels.**

Keywords*—parallel garbage collection, garbage collection, memory management, programming languages.

I. Introduction

Around 1959, garbage collection embedded in LISP was invented by John McCarthy [1]. As one of nine new programming language ideas imported by LISP [3], garbage collection is absorbed by almost every new programming language, becoming an indispensable part of modern programming language.

There must be a reason for this. It starts with the modern computer architecture. Modern computers use a heap-stack memory layout. Instead of storing data on the stack and manipulating data directly, programs prefer to keep more data objects on the heap and hold the reference to objects. In this way, applications can use the more efficient pass-by-reference way rather than a pass-by-value way to avoid to copy a large object repeatedly. Besides, the heap allows programs to create dynamically sized objects, to define recursive data structures and to use objects outside of the domain of definition (The definition of object here is entirely different from the concept of objects in object-oriented programming. The object we talk about refers to the variables which store values on the heap, and we only use them for their references.).

To create an object on the heap, programs have to allocate specified sized memory for a new object. Once the object is useless, programs should deallocate the memory of the object so that the memory could be recycled and reused. In explicit deallocation programming languages like C and C++, it requires programmers to have skills of managing memory to deal with the deallocation of objects. It is quite hard because the situation is always complicated. For example, a typical case is multiple reference points an object. It is hard to know which one is the last reference and objects should be freed after the last reference is dismissed. Moreover, an object is commonly referenced by another object. So one object's deallocation will result in another object's deallocation. And it becomes more complicated when objects reference to each other. If programmers make a mistake, it may cause the memory leak, or worse, wild pointers, which make programs crash. The requirement of memory management somehow increases the difficulty programmers develop applications. In many cases, it is almost impossible for developers to write a comprehensive memory management.

The garbage collection, sometimes called automatic memory management, is responsible for allocating and collecting memory space automatically for memory recycle. The existence of garbage collection could help programmers focus on the logic of software rather than complicated memory management. Thus, an increasing number of software development teams turn to use programming languages with garbage collection like Java and Go.

Due to the importance of garbage collection, from a long time, people were developing lots of algorithms for garbage collection. However, most algorithms are serial. Since increasing clock speeds of CPU has become increasingly impossible to implement, more and more hardware manufacturers turn to offer an increasing number of processors in the modern hardware architectures. This trend leads enterprises to deploy shared memory multi-core or multiprocessor computers. More and more programs run on this kind of computer architectures. Thus, there is little doubt that parallel garbage collection algorithms will play an imperative role in the future computing environment.

The purpose of this paper is to introduce some parallel garbage collection algorithms to understand how garbage collection could be implemented to take full advantage of a multiprocessor. Section II describes the basic idea and fundamental approaches to garbage collection. Section III explains how to parallelise garbage collection from the perspective of the fundamental approaches. Section IV discusses the parallelisation of garbage collection. Conclusions are summarised in Section V.

II. How Garbage Collection Works

Most garbage collection systems can be divided into two parts [2]. The first part is the *mutator* which is responsible for allocating space for new objects and modifying reference fields of objects. The second part is the *collector* which performs the garbage collection actions. The collection action usually includes discovering unreachable objects and reclaiming their

memory space. In parallel garbage collection algorithms, there are more than one mutators and collectors.

The garbage collection systems usually execute collection programs when the mutator can not allocate memory for new objects. When the garbage collection is running, all mutators have to stop working so that programs have to stop executing. This will stop programs that people run, called the stop-the-world problem.

Given the same programs, once the type of algorithms is confirmed, the frequency of garbage collection is fixed. So the pause time becomes a direct indicator of how well a garbage collection algorithm perform. The parallel algorithms we will talk about can significantly reduce pause time.

Before we introduce some parallel garbage collection algorithms, it is necessary to understand the fundamental approaches of garbage collection. There are four fundamental approaches: *mark-sweep garbage collection*, *mark-compact garbage collection*, *copying garbage collection* and *reference counting*. Almost all of garbage collection algorithms are based on one or two of these four approaches.

A. Mark-sweep collection garbage collection

The basic idea of the mark-sweep algorithm is to search *root objects* to mark active objects, then sweep unmarked objects. The search could be depth-first or breadth-first, depending on stack or queue they used as work lists. The root objects refer to the objects which are directly accessible in the current program runtime, such as objects in the program stack or global objects. There are two ways to record whether an object is marked: a mark bit in the object header, or a bitmap table in one side of a heap.

In the sweeping phase, collectors check every object in the heap, find out unmarked objects, which can be considered as garbage, then reclaim its memory space. The memory that is recycled will be stored in a list called a *free list* so that subsequent allocation could search an area from the free list to store objects.

A common implementation of sweeping is *lazy sweeping*, which means we could not sweep the unmarked objects until the memory is going to use that memory that it occupies because once an object becomes garbage, it is always garbage, no longer be referenced again [16].

When people talk about marking, they usually use a tricolor abstraction [2]. In tricolor abstraction, objects are considered as in three colors, which stand for three states. The *white* objects describe whose objects which has been allocated in the memory but not sure if they are still alive (referenced). After they are scanned by any collector, they become *black* objects. Their children objects will be marked as *grey*, that means objects are surely alive but have not been processed to mark their children.

- *black* (presumed live): has been scanned and its children identified
- *grey*: know it but not yet processing it
- *white* (possibly dead): every node is initially white

B. Mark-compact garbage collection

Mark-sweep is simple, but suffering from fragmentation. Even though mark-sweep algorithms use free lists to reuse reclaimed space between live objects, fragmentation is still a potential trouble causing a large object can not be stored in memory even there is enough space in total. To eliminate fragmentation and support quick allocation, mark-compact algorithms compact object into a contiguous memory, usually a side of the heap. By making the object more "compact", the fragmentation issue has been solved and whenever to allocate an object, just move the pointers.

There are many ways to compact objects, but we only talk about the most popular way, which is based on the LISP 2 algorithm. After the same marking phase as the mark-sweep algorithm, it makes three passes over the heap. Each pass finishes a little work. The first pass determines the forwarding address of each object. The second pass updates the references of marked objects so that they could refer to the new addresses. The last pass finishes the moving jobs to move objects to target locations. Objects can be rearranged on the heap in the three ways: arbitrary, linearising or sliding. To achieve a good locality, most collectors arrange objects in the way of sliding.

C. Copying garbage collection

There is a garbage collection algorithm seems to be a combination of mark-sweep and mark-compact. It finished the compact during the process it marks objects. It is called copying algorithm.

In copying algorithm, the heap is divided into two equally sized *semispaces*, called *fromspace* and *tospace*. The objects are allocated in space. When space is full, garbage collection starts working. The fromspace and space are *flipped*. So the from space becomes tospace, and tospace becomes from space. Then collectors copy root objects into to space. After that, they will scan root objects, update the reference of gray objects and copy them from fromspace to tospace.

Like other tracing collection, the semispace copying collection also needs a work list to store pending objects. However, this can be implemented by a simple FIFO queue onto tospace without extra space other than a single pointer [17]. It uses a *scan* pointer, which points to the next unscanned objects. Combined with the *free* pointer which points the next usable space, the objects in the work list store between the scan pointer and the free pointer. Thus, to determine termination, we can just check whether the scan pointer catches up the free pointer.

We have mentioned locality many times. The locality is the phenomenon in which the same or related storage locations will frequently be accessed. Almost every operating system will employ page cache techniques to optimise the performance according to this. So we can conversely use this point to make related objects stay together so that operating systems could optimise the access speed.

Generally speaking, the copying algorithm reduces a half of memory than the mark-sweep algorithm. It causes the collection programs will be executed more frequently. So the copying algorithm should be less efficient than the mark-sweep

algorithm. However, it is not true. The copying algorithm makes objects more compacted, and it is more efficient in fact due to the locality. Compared with mark-compact, the advantage of copying algorithm is that it just need one pass to finish the algorithm.

D. Reference counting

The previous three algorithms are all based on tracing live objects. However, the last fundamental algorithm is bidirectional. That means the object knows how many references point to it so that they can free memory whenever no reference refers to it. Moreover, we call this algorithm *reference counting* [20]. The reference counting is straightforward and fast. It does not require a period of pause time to stop programs so that the collectors can do their work. After an object is unreachable, its memory will be released immediately.

However, the reference counting suffers from a fatal flaw. When some situations like two objects hold references to each other or some objects keep pointers to themselves, it is hard for reference counting to find out these "isolated" objects. It results in memory leak because the memory which stores these objects will never be released.

Due to the flaw, reference counting can only be used in a few limited spaces. The generational garbage collection, which is most common algorithm we use now, trend to use tracing algorithms. In another way, reference counting naturally supports parallel and concurrent, so we do not have much to discuss it.

E. Generational garbage collection

Tracing algorithm is efficient when there are only a few live objects. However, when we are using mark-compact or copying algorithms, there is a problem that long-lived objects will be moved again and again leading to performance loss.

The idea of generational garbage collection is to divide objects into young objects and old objects. Then scan young objects frequently and old objects occasionally. For the young objects which have been scanned many times, we can consider them become "older" then move them into the old generation.

Most garbage collections now are generational collections. However, they are still based on the three tracing algorithms we mentioned before: mark-sweep, mark-compact, and copying. Usually, they will apply different algorithms to different generations. Most young generation uses copying algorithm.

III. Parallel Garbage Collection

The key to designing parallel algorithms is how work can be distributed in a way that keeps all processors busy. A simple way is to separate work into N pieces, where N is the number of processors, then allocate these N pieces of work onto n processors. However, it can not avoid the situation that some processors run faster than another processor so that the fast processors have to wait for the slow processors. In other aspect, separating work into average N pieces itself is sometimes very hard to accomplish. A fairly common practice is called *over partition*, which tries to divide work into very small pieces. The number of pieces divided is much larger than the number of processors so that each processor can take as many pieces of work as they can finish. However, it can not be simply done by over partition work into as small as possible, because it needs synchronisation when processors take or generate work. It will lead to unavoidable waiting. Hence, the granularity becomes important in this topic.

In this section, we are going to describe some algorithms introduced by some papers. As we can separate garbage collection into four basic operations, we are going to discuss them in the following aspects and order:

1) parallel marking
2) parallel sweeping
3) parallel compact
4) parallel copying

A. Parallelise marking

In all tracing algorithms, marking is the most important step. It usually takes up the majority of time spent on garbage collection. Therefore, parallelising marking is the most important part to improve a garbage collection algorithm.

Many parallel marking algorithms are based on a concept of *work stealing*, which means that if threads finish the work in their local work list, they will *steal* work from others'. Endo *et al.* [5] describe an algorithm which allocates each thread a local mark stack and a *stealable* work queue. Each thread checks its stealable mark queue periodically. If the mark queue is empty, it will transfer a half of work in its local mark stack to the queue. If it runs out of marking work in its local mark stack, it will check its stealable work queue before it steals work from others' stealable work queue.

In case that multiple threads steal work from the same stealable queue, the access to stealable queues has to be protected by locks. A normal way to use locks is to claim locks then steal works, other late steals have to wait for the first thread leave and unlock. There is no doubt that this claim-lock-then-steal approach will lead to a terrible performance. Endo et al. use a try-lock-then-steal-else-skip approach, namely, try to lock the stealable queue, steal work if success, otherwise just skip to steal another stealable queue. This makes algorithm lock-free.

To process large objects with a proper load balancing, Endo et al. split a large object into 512-byte sections. To detect the termination of garbage collection, Endo et al. give each processor two flags to indicate if their mark list is empty. A global detection-interrupted will be set when a thread is attempting to steal others' work in case of the moment during stealing in which the work has not yet been done, but the flags of threads could be set. A thread which just finished work will clear the detection-interrupted flag and checks all other processors' flags. If every thread's flag is set, then the garbage collection mission has been accomplished.

Based on Arora et al.'s work stealing algorithm [19], Flood et al. [7] use a single deque (double-ended-queue) rather than a stack and a stealable queue. The deque combines a local stack and a stealable queue: the bottom of the deque is considered as the thread's local work stack while the top of the deque

is stealable. The synchronization is only required to claim the last element of the deque. The idle threads steal an object from other deques.

The deque of each thread is fixed size, so it risks an overflow. To avoid overflow, Flood *et al.* provide a global overflow set, which is a list of lists distinguished by classes. When the overflow, the object will be put on an overflow list classified by its class. The idle threads will first try to fill half of their deque from the overflow set, then steal others.

Flood *et al.* detect termination through a status word, which has one bit for each collector thread. Each thread set their status bit atomically. When an thread becomes idle, it first sets its status bit, then checks others' status to detect termination. If not terminate, it sets its status bit then attempts to steal another thread's work, it should first set its status bit. If it fails to steal, then it reverts the bit. However, because every thread visit the status word atomically, when there are a large number of threads, this technology will be not efficient. The authors also suggested another way to use a counter of active threads stead.

Siebert [12] also implemented work stealing algorithm for Jamaica real-time Java virtual machine. His algorithm breaks objects into linked blocks. Each block uses a word to represent the colour. For load balancing, a thread steals all of another thread's grey lists. A new colour anthracite is introduced for blocks while they are being scanned in a marking step, to prevent two threads from working on the same grey block.

The work stealing concept implies that the idle thread is active to take others' jobs away. Wu and Li's *Task-pushing* algorithm [10] shows another way in which the thread can ask for marking work and be passive to accept whatever work generated by other threads. In this algorithm, given 'n' marking threads, each thread will have $N-1$ *channels* which are responsible to store work which they generate and assign to another thread. So an idle thread can search for work in every channel that go directly from other threads to it. Obviously, this way avoid atomic operations, on servers with a large number of processors, this improves performance much than previous work stealing. However, the question is how to decide the length of channels. On a machine with 16 Intel Xeon processors, one or two were best according to their tests.

Wu and Li use a similar termination detection solution provided by Kolodner and Petrank [6]. In Kolodner and Petrank's solution, only a collector thread is responsible for detecting the termination. A synchronised, global, detector-identity word is used to ensure there is only one thread detecting termination. In Wu and Li's algorithm, a fixed detector thread is assigned to detect termination to avoid the conflicting detector problem.

Ossia *et al.* [8] employ an entirely different way to parallelise marking. They use the concept of a *packet* which contains a certain amount of work. Each thread has two packets: an input packet to acquire work and an output packet to store the generated work. Each time threads finish the work in a packet, they compete to gain a new packet from the global pocket pool. Moreover, whenever they yield enough work to fill a packet, they put the packet into the global pocket pool. There are 1000 available packets with a fixed size 512 entries.

This algorithm is also implemented in Insignia's Jeode Java Virtual Machine and called "grey packet" by Thomas et al. [18] because all objects in the packet are grey from the perspective of tricolor abstraction.

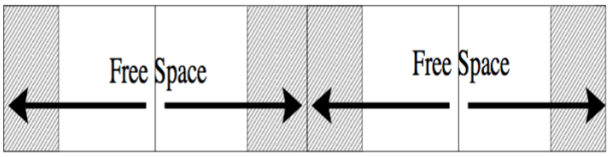

Fig. 1. The way of compaction in Flood *et al.*'s algorithm [7]

It requires synchronisation when packets are acquired from or return to the global list. The termination is easily detected by the numbers of empty packets is same as the total number of packets available.

B. Parallelise Sweeping

Parallelising sweeping is quite straightforward and natural because lazy sweeping is naturally parallel and concurrent. A simple way is to over-partition the heap into blocks and have threads compete for blocks to sweep and then add to a global free-list. However, it is hard to merge blocks into the shared global heap block free list, because the blocks must be sorted and adjacent blocks have to be coalesced. The global heap block will become the bottleneck for this algorithm.

In Endo *et al.*'s algorithm [5], each processor has a local reclaim list and all processors share a single global heap block free list. When a thread scan blocks, it will link empty blocks to the global free list and non-empty ones to its local reclaim list. Before each processor insert blocks to the shared list, they let each processor sort and coalesce blocks locally as many as possible. The local reclaim list will not merge into the global heap block free list until all work has been accomplished. To reduce the contention and the overhead, each processor can get a large number of blocks to process locally and not further load balancing technique like work stealing is used.

C. Parallelise Compaction

The difficulty to parallelise compact phrase is that we are not able to ensure one thread did not overwrite object data before another thread moved the object. To avoid this problem, Flood *et al.* [7] break the heap into *N regions* that contain almost the same number of live objects, where N is the number of collector threads. Each thread will slide objects in its region only. To reduce fragmentation, they slide objects in different directions in even and odd numbered regions. As shown in Fig. 1, this could reduce almost a half of the number of gaps from N to $ceil((N+1)/2)$.

Flood *et al.*'s algorithm are based on the Lisp 2 algorithm. So they have three phases: calculate forwarding addresses, update reference, and move objects.

In the first phase, they over-partition the heap into roughly same sized units. Then threads compete to claim units to calculate the total size of live objects in each unit. After that, they split the heap into N regions as we mentioned before.

At the same time, they compute the forwarding address of the first live object in each unit. Then collectors compete for units again to install forwarding pointers in every live object of their units. In the next phase, they reuse the unit partitions to update references to point to objects' new locations. In the last phase, they give each thread a region to move objects.

Even we can see that the abnormal slide directions can reduce the number of gaps to reduce some degree of fragmentation, but if there are a large number of processors, it is too many regions to be partitioned so that the allocation of a large object becomes difficult.

Abuaiadh *et al.* [21] attempts to improve Flood et al.'s algorithm by using the mark bitmap and an offset vector which holds the new address of the first live object in each small block of the heap to calculate the new forwarding address instead of storing them. This could reduce the number of passes. Then they over-partition the heap into a number of fairly large areas. As they suggested, the number of areas could be 16 times as many as collector threads. Each thread compete to claim an area to compact and an area into which it can move objects. They use a table to record the free pointers of each area. When a thread got an area to write objects, it will write null atomically into the corresponding table slot. Once it has finished with the area, it updates the area's free pointers in the table and remains null if the area is full. The heap areas will be compacted in order. Moreover, a thread can compact an area into itself.

Kermany and Petrank [25] adopted a mark-compact algorithm with a similar idea of copying algorithms. The Compressor, which is what they called the garbage collector, compacts the heap into a second space and preserves the order of objects. They use some virtual memory operations to implement the compaction. First, they use the mark-bit vector from the output of marking procedure as input to calculate an offset table, which records the new address to which the object is moved. Then each Compressor thread competes to get a page that has not yet been moved. A new virtual page may be allocated at this point to accept the moved objects. Moreover, a successful thread maps a new physical page for its virtual page. According to the new addresses computed from the offset table, it moves objects on this page to their new location. Later on, the thread checks the moved objects and uses the offset table to fix their pointers. Finally, the thread will return the pages from which the objects were moved to the operating system.

At first sight, this may look as if it is a copying algorithm rather than a mark-compact one. However, Compressor truly is a sliding mark-compact collector. It does not need a large space for the collection.

D. Parallelise Copying

Before we start to introduce the parallel copying algorithms, it is worthy to know that the difference between copying algorithms and others is, we can only copy objects once. Otherwise, there will be duplicate objects to break data consistency.

To implement a parallel copying algorithm, Blelloch and Cheng [24] gives each thread a local stack. For loading balance, the threads will transfer work between their local stacks and a global shared stack frequently. In consideration of unavoidable synchronisation when threads access the shared stack, they follow a discipline to reduce overhead. It is that the operations can be accumulated on the stack if they are all push or all pop elements to the stacks. Rely on this discipline, they use a *room* to guarantee that there are no pushing threads and poping threads at the same time. The room is used like the following way: before a thread entering a room, it has to confirm that the room is empty. If not, they have to wait until every thread leaves the room. If they enter a pop room, then they can get work from the shared stack, generate grey objects into their own stacks. If they are in a push room, they throw all of work from their stacks to the shared stacks. To detect the termination, the last thread to leave the room should check if the shared stack is empty. The problem of this algorithm is that any thread wishes to enter another room have to wait for all thread exit from the room.

This algorithm is processor-centric, that tend to split work by size. However, to achieve a better locality, there are some memory-centric algorithms, which split work according to their memory location.

As the first parallel copying algorithm, Halstead's algorithm [23] is quite simple and poor performance. However, it is inspirational as a memory-centric copying algorithm. Halstead assigns per collector thread its fromspace and tospace. Each thread compete to copy objects and install forwarding pointers. It is obvious that this design risks poor load balancing because some threads will run out their work first and others will work slowly. It also requires some mechanism to handle the case that one thread's tospace overflows.

Imai and Tick [4] provide the first block-based approach. Their algorithm divides tospace into small, fixed size blocks. Each thread competes for chunks which contain grey objects to scan and empty blocks for storing copying objects. If a copy block is filled, it has to be transferred to a global shared pool so that some idle threads could compete to acquire them, and an empty one will be allocated to the thread. If a scan block is completed, a fresh one can be obtained from the shared pool. As for some objects which are larger than the size of a block, they will be allocated and reallocated from the shared global pool directly using a lock.

Barabash *et al* [14] present an idea of "packets" which is similar to the block in Imai and Tick's algorithm. The difference between packets and blocks is that packets are shared between processors at a "whole-packet" granularity. And the scan-packets and copy-packets are quite distinct. Also in their algorithm, the termination is more easier to detect. And the algorithm is flexible for a variable number of processors.

Marlow *et al.* [13] found that Imai and Tick's block approach has a weakness when there is a little work. A thread exports work to other threads only if scan and free opinters are separated by more than a block. So they allow incomplete blocks to be added to the global pool when one of the following conditions is satisfied: a. the size of the pool is below some threshold; b. The thread's copy block has a sufficient work; c. There are idle processors.

Marlow *et al.* [13] also compared two approaches to copy objects in the GHC Haskell system. In the first approach, a

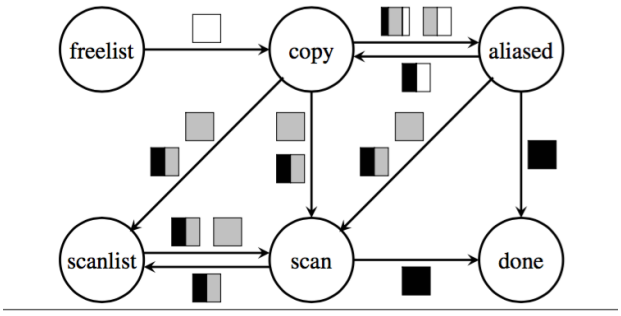

Fig. 2. Block states and transitions [9]

copy	scan aliased	scan ◧ or ▩	scan ■
◧ or ▩	(no action)	scan → scanlist copy → aliased	scan → done copy → aliased
□ or ◧	aliased → copy scanlist → scan	(no action)	scan → done scanlist → scan
◧ or ▩	aliased → scan freelist → copy	copy → scanlist freelist → copy	scan → done copy → scan freelist → copy
■	aliased → done freelist → copy scanlist → scan	(can't happen)	(can't happen)

Fig. 3. Transition logic in a garbage collection thread [9]

thread trying to copy an object first tests whether it has been forwarded. If not, the thread attempts to compare-and-swap a "busy" value into the forwarding address. If the operation succeeds, the thread copies the object, writes the address. If failed, the thread will wait for another thread finishes it. In the second approach, they try to avoid the waiting by having threads copy the object without hesitation, then compare-and-swap the forwarding address. If the operation fails, they rollback the copy operation by retracting the free pointer to its original location. They found the latter approach offer little benefit due to the rareness of thread collision. But they considered it worthwhile.

The algorithms we talked about so far are all breadth-first copying. To improve locality, we would like to use some depth-first copying algorithms. Moon [15] and Wilson *et al.* [22] emploied *hierarchical* copying algorithms that provide almost depth-first traversal without the cost of a stack. However, the algorithms are sequential. Siegwart and Hirzel [9] added hierarchical decomposition to the Imai and Tick parallel copying collector to manage the young generation of IBM's J9 Java virtual machine.

In Siegwart and Hirzel's implementation of the parallel hierarchical copying garbage collection, copy blocks and scan blocks are aliases for achieving hierarchical copy order.

As shown in Fig. 2 and Fig. 3, there are three shared work pool in this algorithm: *freelist*, *scanlist*, and *done*. When a thread is copying objects, it will get empty blocks from the freelist, producing blocks with grey objects, or blocks with grey objects and space. Those blocks with grey objects and space will be aliased to become blocks with only grey objects (including black objects but no space) or blocks with only space (including black objects but no grey objects) so that they can be used for copying and scanning respectively. The blocks with grey objects and no space will be scanned or be transferred to the scanlist if the thread has too much work. Moreover, the block with all black objects will be transferred to the done work pool.

Oancea et al. [11] provide an algorithm that is akin to Wu and Li's channels marking algorithm. They divide heap into more partitions than the number of processors. For each partition, there is a work list and at most one thread can own that work list. If a processor finds a reference to an object in its partition, the reference is sent to that partition. Each thread processes work in its incoming channels and its work list.

IV. Discussion

The parallel algorithms are commonly used in the reality. However, a question that should be asked is: is parallel collection worthwhile? The parallel collection will require some synchronisation between cooprating garbage collection threads, and this will incur overhead. So it should be confirmed that there is sufficient garbage collection work available for the gains offered by a parallel solution to more than offset these costs.

Another serious problem for designing parallel garbage collection algorithms is: there is not a fixed environment setting. Different algorithms may have different performance and different advantages or disadvantages in various environments where there are different numbers of CPUs and different sizes of memory. This also makes the benchmark very difficult to make. Usually, people have to be an armchair strategist and make decision as the rule of thumb.

V. Conclusion

This review has presented the basic conception of garbage collection and key aspects of parallel garbage collection algorithms. We found that an usual design of parallel algorithms to avoid synchronisation is to give each processor a local work list. The processor can finish their local work without requirements to wait for others. If a thread runs out of its work sooner than others, it will take work from another thread's work list or get work from a global work pool. Another common technique is to over-partition to get processor-centric packets or memory-centric blocks. Then allocate work to different threads. Some algorithms use the idea of channels. Each thread pushes work into a private line to another thread. It is similar with work stealing, but has less overhead. To avoid the wait time brought by locks, most algorithms use a try-lock-then-do-else-skip method to be lock-free or wait-free.

References

[1] J. McCarthy, "Recursive functions of symbolic expressions and their com-putation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184-195, 1960.

[2] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. Steens,"On-the- y garbage collection: An exercise in cooperation," *Communications of the ACM*, vol. 21, no. 11, pp. 966-975, 1978.

[3] P. Graham, "What Made Lisp Different", paulgraham.com, 2002. http://www.paulgraham.com/diff.html. Accessed: 20-May-2016.

[4] A. Imai and E. Tick, "Evaluation of parallel copying garbage collection on a shared-memory multiprocessor," *Parallel and Distributed Systems, IEEETransactions on*, vol. 4, no. 9, pp. 1030-1040, 1993.

[5] T. Endo, K. Taura, and A. Yonezawa, "A scalable mark-sweep garbage collector on large-scale shared-memory machines," in *Supercomputing, ACM/IEEE 1997 Conference*. IEEE, 1997, pp. 48-48.

[6] A. Azagury, E. K. Kolodner, and E. Petrank, "A note on the implementation of replication-based garbage collection for multithreaded applicationsand multiprocessor environments," *Parallel processing letters*, vol. 9, no. 03, pp. 391-399, 1999

[7] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang, "Parallel garbage collec-tion for shared memory multiprocessors." in *Java Virtual Machine Research and Technology Symposium*, 2001.

[8] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko, "A parallel, incremental and concurrent gc for servers," in *ACM SIGPLAN Notices*, vol. 37, no. 5. ACM, 2002, pp. 129-140.

[9] D. Siegwart and M. Hirzel, "Improving locality with parallel hierarchicalcopying gc," in *Proceedings of the 5th international symposium on Memory management*. ACM, 2006, pp. 52-63

[10] M. Wu and X.-F. Li, "Task-pushing: a scalable parallel gc marking algorithm without synchronization operations," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1-10.

[11] C. E. Oancea, A. Mycroft, and S. M. Watt, "A new approach to parallelisingtracing algorithms," in *Proceedings of the 2009 international symposium on Memory management*. ACM, 2009, pp. 10-19.

[12] F. Siebert, "Concurrent, parallel, real-time garbage-collection," in *ACM Sigplan Notices*, vol. 45, no. 8. ACM, 2010, pp. 11-20.

[13] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones, "Parallelgenerational-copying garbage collection with a block-structured heap," in *Proceedings of the 7th international symposium on Memory management*. ACM, 2008, pp. 11-20.

[14] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank, "A parallel, incremental, mostlyconcurrent garbage collector for servers," *ACM Transactions on Program-ming Languages and Systems (TOPLAS)*, vol. 27, no. 6, pp. 1097-1146,2005.

[15] D. A. Moon, "Garbage collection in a large lisp system," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM, 1984, pp. 235-246.

[16] R. J. M. Hughes, "A semi-incremental garbage collection algorithm," *Soft-ware: Practice and Experience*, vol. 12, no. 11, pp. 1081-1082, 1982.

[17] C. J. Cheney, "A nonrecursive list compacting algorithm," *Communications of the ACM*, vol. 13, no. 11, pp. 677-678, 1970.

[18] S. P. Thomas and R. E. Jones, "Garbage collection for shared environment closure reducers," *Computing Laboratory, The University of Kent at Canterbury. Technical Report*, pp. 31-94, 1994.

[19] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling formultiprogrammed multiprocessors," *Theory of Computing Systems*, vol. 34, no. 2, pp. 115-144, 2001.

[20] G. E. Collins, "A method for overlapping and erasure of lists," *Communications of the ACM*, vol. 3, no. 12, pp. 655-657, 1960.

[21] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein, "An ecient parallel heap compaction algorithm," in *ACM SIGPLAN Notices*, vol. 39, no. 10.ACM, 2004, pp. 224-236.

[22] P. R. Wilson, M. S. Lam, and T. G. Moher, "Eective "static-graph" re-organization to improve locality in garbage-collected systems," in emphACM SIGPLAN Notices, vol. 26, no. 6. ACM, 1991, pp. 177-191.

[23] R. H. Halstead Jr, "Implementation of multilisp: Lisp on a multiprocessor," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM, 1984, pp. 9-17.

[24] G. E. Blelloch and P. Cheng, "On bounding time and space for multiprocessor garbage collection," in *ACM SIGPLAN Notices*, vol. 34, no. 5. ACM,1999, pp. 104-117.

[25] H. Kermany and E. Petrank, *The Compressor: concurrent, incremental, and parallel compaction*. ACM, 2006, vol. 41, no. 6.