To appear in the USENIX JVM Conference, April 2001

# Parallel Garbage Collection for Shared Memory Multiprocessors

Christine H. Flood
*Sun Microsystems Laboratories*
Christine.Flood@sun.com

David Detlefs
*Sun Microsystems Laboratories*
David.Detlefs@sun.com

Nir Shavit
*Tel-Aviv University*
shanir@tau.ac.il

Xiaolan Zhang
*Harvard University*
cxzhang@eecs.harvard.edu

## Abstract

We present a multiprocessor "stop-the-world" garbage collection framework that provides multiple forms of load balancing. Our parallel collectors use this framework to balance the work of root scanning, using *static overpartitioning,* and also to balance the work of tracing the object graph, using a form of dynamic load balancing called *work stealing.* We describe two collectors written using this framework: pSemispaces, a parallel semispace collector, and pMarkcompact, a parallel markcompact collector.

## 1 Introduction

The Java™ programming language is increasingly used for large, memory-intensive, multithreaded applications run on shared-memory multiprocessors. Most Java virtual machines (JVM™'s) employ "stop-the-world" garbage collection (GC) algorithms that first halt the running threads and then perform the GC. If we have more than one processor available, it makes sense to employ them all in the GC process. This paper describes the parallelization of two sequential GC algorithms to allow them to take advantage of all available processors.

The Java Technology Research Group at Sun Microsystems Laboratories[1] has developed a JVM that includes a *GC interface* [12] to support multiple GC algorithms, thus enabling comparison of various GC strategies in a high performance virtual machine. This paper describes our augmentation of this interface with a parallel infrastructure to support multiple parallel GC strategies. We use this infrastructure to parallelize two well-known collection schemes: a two-space copying algorithm (*semispaces*) and a mark-sweep algorithm with sliding compaction (*markcompact*). The resulting algorithms have outperformed their highly-tuned product-quality sequential counterparts on multiprocessors.

In parallelizing sequential GC algorithms, one has to tackle two key issues: load balancing of all parts of the algorithm and re-engineering of any inherently sequential elements.

In both algorithms, the key to load balancing is to correctly and efficiently partition the task of tracing the object graph. This task unfortunately does not lend itself to static partitioning. Our approach, described in the following sections, is to combine static partitioning with dynamic load balancing based on work stealing. We show that this combination of static and dynamic methods leads to effective parallelization of both the semispaces and markcompact collectors. It is our belief that the effectiveness of our dynamic partitioning is the result of a finely-tuned lock-free work-stealing algorithm based on Arora *et al.* [1] whose low overhead allows us to balance our work at the individual object level.

In both algorithms there are parts that are not easily parallelizable. In the semispaces algorithm these included: installing forwarding pointers, allocating in parallel, and scanning the card table for references from the old generation. Though the installation of forwarding pointers is not parallelized,

[1] www.sun.com/research/jtech

it is performed in a lock-free manner. We make allocation less of a sequential bottleneck by globally allocating local buffers from which objects may be allocated without synchronization. Scanning the card table requires a novel partitioning scheme to achieve good load balancing. In markcompact the main inherently sequential part is the compaction phase, which involves copying all objects to one end of the heap. Here we statically partition the old generation heap into $n$ partitions, compacting even partitions in one direction and odd partitions in the other direction, thus avoiding synchronization and optimizing the size of the free areas.

### 1.1 A short description of previous work

Endo *et al.* [11] describe a parallel stop-the-world GC algorithm using work stealing. Their algorithm depends on threads with work copying some work to auxiliary queues, where the work is available for stealing. Threads without work look for an auxiliary queue with work, lock the queue, and steal half of the queue's elements. Our work extends theirs by using a lower-overhead work-stealing mechanism, and by addressing the harder problem of parallelizing relocating collectors, not just a non-relocating mark-sweep algorithm.

Halstead [6] describes a multiprocessor GC for Multilisp. Each processor has its own local heap, and they use lock bits for moving and updating forwarding pointers. Load balancing is done statically rather than dynamically.

Many collectors operate concurrently with mutator activity [9, 4, 5, 8]. This kind of concurrency is orthogonal to the style of parallel collection we describe in this paper. A collector might combine both: some concurrent collectors have stop-world phases that might be performed in parallel, and collectors with concurrent GC threads might use several such threads working in parallel to cope with high aggregate garbage-creation rates in multi-threaded programs.

Steensgaard [10] explores a clever method for partially parallelizing collection. Compile-time analysis identifies allocation sites that allocate objects that never *escape* the allocating thread (are never accessible to other threads.) Such objects are allocated in a *thread-local heap,* which can be collected independently of other threads. This technique avoids the synchronization issues that general parallel collection must address, but requires extensive and expensive static analysis, and only a subset of objects may be collected thread-locally.

### 1.2 Overview

Section 2 presents basic parallel programming techniques. Section 3 presents our parallel GC infrastructure, which applies these techniques to the garbage collection problem. Sections 4 and 5 describe two parallel algorithms we implemented using this infrastructure: pSemispaces and pMarkcompact. Section 6 presents results for three benchmarks. Section 7 presents conclusions.

## 2 Parallel Programming Basics

If we had a predetermined amount of work to do and were able to partition it perfectly across all available processors, we would achieve perfect parallelism and finish the collection in the least possible amount of time. Some tasks can be partitioned in this way; we call them *statically partitionable.* Other tasks are difficult to divide into subtasks of predictable size. For example, tracing the graph of a program's live data is difficult to subdivide *a priori,* because it depends on the shape of the object graph. Many tasks fall somewhere in between: we are able to partition them statically into roughly, but not exactly, equivalent subtasks. We *overpartition* such tasks. That is, we break the tasks into more subtasks than we have threads, and then each thread dynamically claims one subtask at a time.

There are two motivations for overpartitioning. First, the number of processors available to the GC process is unpredictable due to load on the machine from other processes. If a task were divided into exactly $n$ subtasks on an $n$-processor machine, and one of the processors were unavailable, then one processor would have to complete two subtasks, thereby doubling the time for the computation. With overpartitioning, this extra subtask would be divided into several smaller subtasks that may be distributed across the active processors. Second, when we only have a rough estimate of how much work each subtask represents, assigning just one task to each processor risks one of those tasks being significantly larger than the others. Overpartitioning both decreases this risk by making smaller subtasks, and enables processors that have finished smaller subtasks to take on additional work.

Some tasks are not even approximately statically partionable. These tasks require some form of dynamic load balancing. *Work stealing* [2] is a highly effective load balancing technique in such situations. In this approach, each thread works on its own tasks until it runs out of work, and then takes the initiative to steal work from one of the other processors.

## 2.1 A short explanation of lock-free work-stealing queues

Arora *et al.* present a non-blocking implementation of a double-ended queue data structure tailored to support work stealing with minimal synchronization. Each thread has its own work queue of tasks. There are three fundamental operations: *PushBottom* pushes an element onto the bottom of the queue, *PopBottom* pops an element from the bottom of the queue, and *PopTop* pops an element from the top of the queue. *PushBottom* and *PopBottom* are local operations that usually require no synchronization. *PopTop* is used for stealing from other threads' queues.

A parallel algorithm using work stealing starts with available tasks distributed among the work queues. Each thread uses *PopBottom* to claim tasks from its local queue. Execution of this task may reveal new subtasks, which are then added to the local queue using *PushBottom.* When a thread runs out of work it uses *PopTop* to steal a task from some other thread's work queue. Synchronization is required only when stealing an element from another queue or when claiming the last element from the local queue.

We modified the algorithm of Arora *et al.* in several ways. We added a termination detection protocol to ensure that all work is complete before any thread terminates. We also added support for fixed size queues in the form of an overflow detection and handling mechanism.

# 3 Parallel GC Infrastructure

## 3.1 Balancing root scanning

Garbage collection computes the transitive closure of objects reachable from a set of *root* pointers. In our JVM, the root set consists of class statics, thread stacks, etc. We overpartition these roots into groups, and the GC threads compete dynamically to claim root groups. Even if the static partitioning succeeds in balancing root scanning, starting off with balanced groups is not sufficient. Some roots may lead to large data structures, while others may lead to single objects.

## 3.2 Balancing traversal of live data

We solve this problem by using work stealing to dynamically balance the load. The tasks are references to objects to be *scanned,* i.e., examined for pointers to other objects.[2] A scanning GC thread acquires an object reference either from its local queue or by stealing from another thread's queue, and pushes any outgoing references found in the object onto its local queue. The termination detection protocol is used to determine the completion of the transitive closure.

Consider the behaviour of this algorithm on a large linked data structure, say a binary tree. One thread will scan a root pointer referencing the top-level node of the tree, push both child nodes onto its work queue, and then pop one of the child nodes for processing. The other child node is now available for stealing. In this way, for a sufficiently large tree, the load will be dynamically balanced.

## 3.3 Termination detection

The termination protocol is based on a status word containing one bit for each thread participating in the GC. All threads start off marked active. As long as a thread has local work, gets work from the overflow lists (see section 3.4), or succeeds in stealing work, its bit in the status word remains on. Once it is unable to find work it sets its status bit to off and loops, checking to see if all the status bits are off. If so, then all threads have offerred to terminate, so the algorithm is complete. If not, the thread peeks at other threads' queues, attempting to find one with work to steal. If it finds a thread with work to steal, the thief sets its status bit to active and tries to steal the work. If it succeeds, it goes back to processing. If it fails, it sets its status bit back to inactive and resumes the loop.

Our colleague Peter Kessler has suggested replacing the status word with an integer indicating the number of active threads. To offer termination, a thread would decrement this count with an atomic instruction; if the count goes to zero, all threads have terminated. When an inactive thread becomes active, it would increment the count, again with an atomic instruction. This avoids the parallelism limitation imposed by the bit-width of a word, but we have not yet implemented this proposal.

## 3.4 Handling overflow in GC work-stealing queues

In order to avoid allocation during GC we allocate fixed-size work-stealing queues at startup time

[2] For large objects, especially large arrays of references, it might be advantageous to consider the object as comprised of several *chunks*, and subdivide the object-scanning task into the separate tasks of scanning each chunk. We have not implemented this extension.

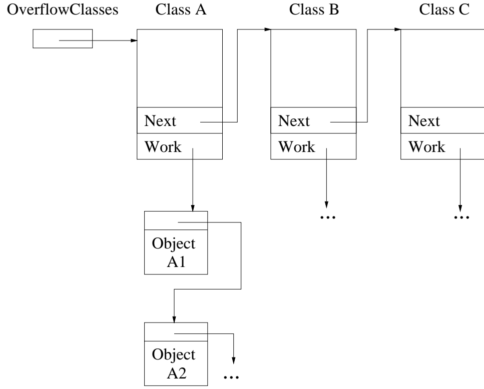



Figure 1: Overflow sets

and use them for all GC's. This required modifications to the work-stealing code to check for overflow, and a mechanism for handling overflow gracefully by offloading some items to a global *overflow set.* Threads without work look to the overflow set for work before resorting to stealing. We wished to be able to handle overflow without any additional storage space and also to avoid "thrashing" of objects between the overflow set and the work-stealing queues.

We modified *PushBottom* to check for possible overflow before adding an element. If adding an element would cause the queue to overflow, we pop all elements in the bottom half of the queue and add them to the overflow set.

The overflow set mechanism, due to our colleague Ole Agesen, exploits a class pointer header word present in all objects in our implementation. As shown in Figure 1, for each class $X$, we link all instances of $X$ in the overflow set together in a linked list whose head is contained in the class structure for $X$. Each object's class pointer is overwritten with the "next" pointer of the list. This does not destroy information, since all objects in a given class's list are instances of that class. All classes with instances in the overflow set are linked into another list. This mechanism represents the overflow set with only a small per class storage overhead.

By draining only the bottom half of a queue on overflow and filling no more than the top half of the queue when retrieving work from the overflow set, we ensure that no object will be placed in the overflow set more than once, thus avoiding "thrashing."

## 4 Parallel Semispaces

Semispaces (a.k.a. "copying") collection divides the heap equally into two regions: from-space and to-space. Objects are allocated in from-space until it fills up, then a GC is triggered. Reachable objects are copied into a contiguous area of to-space, leaving the remaining space free for allocation. As the GC traces the transitive closure, it copies each object when it is first encountered, leaving a *forwarding pointer* in the from-space copy of the object to indicate its new address in to-space. Subsequent references to this object are updated with the forwarding pointer.

In the elegant style of Cheney [7], a *copy* pointer tracks the next free address, and a *scan* pointer tracks the next object to be scanned. The GC scans the object indicated by the scan pointer; it examines references in the object, copying any referenced object still in from-space to to-space, updating the copy pointer. The scan pointer is then updated to point to the next object. Collection is complete when the scan pointer reaches the copy pointer, at which point we swap to-space and from-space, and resume the program.

Our *pSemispaces* algorithm parallelizes this sequential algorithm. We depend on the infrastructure to properly distribute the process of scanning the roots. Rather than using Cheney's copy and scan pointers to represent the set of objects to be scanned, we use explicit work-stealing queues.

With a parallel copying collector, many threads allocate objects in to-space at the same time. One approach to managing this concurrency would be for each thread to increment the copy pointer atomically for each object it copies, using some hardware operation such as fetch-and-add or compare-and-swap (CAS) [3]. However, our experiments indicate that this results in too much contention. The alternative we adopted was to have each thread use such atomic allocation only to allocate relatively large regions called *local allocation buffers* (LABs). A thread can then do local allocations within this buffer with no synchronization. A thread can also deallocate its most recent allocation, which is useful in parallelizing the insertion of the forwarding pointer, as we explain below. LABs should be large enough to reduce contention on the copy pointer, yet small enough to avoid excessive fragmentation. Note that the potential fragmentation introduced by LABs makes it possible that to-space may not hold all of the objects copied from from-space. However, this is a concern only when the heap is very nearly full.

Collection must preserve the shape of the object graph. If several threads are simultaneously processing references to the same uncopied object in from-space, only one may succeed in copying the object. The others must observe that the object has been

copied and update their references according to the forwarding pointer installed by the copying thread. We accomplish this by having each thread speculatively allocate space for the object in its LAB, and then use a CAS to update the from-space object's forwarding pointer to point to the speculative new address. If the CAS succeeds, the thread proceeds to copy the object. If the CAS fails, the CAS returns the updated forwarding pointer.[3] The thread uses this value to update its reference, and then locally retracts its speculative allocation.

The semispaces algorithm is often used in youngest generations of generational collectors [7, 13]. A generational collector has two or more generations; objects are usually allocated in younger, smaller generations, and *promoted* to older generations if they survive long enough. The hope is that youngest-generation collections are significantly faster than collections of the entire heap, and likely to reclaim sufficient space to continue computation. However, multi-threaded programs running on multiprocessors will have larger aggregate allocation rates than single-threaded programs, and will therefore fill a young generation of a given size more quickly, increasing collection frequency. It is therefore attractive to increase the size of the youngest generation to reduce collection frequency with multi-threaded programs, and to use parallelism to keep pause times low and throughput high.

Two further issues must be addressed when using the pSemispaces algorithm in the youngest generation of a generational collector. First, collector threads will allocate both in to-space and in the older generation (for promotion). Both forms of allocation must be parallelized; old-generation promotion therefore uses the same LAB-based allocation technique as to-space allocation. Second, when performing a youngest-generation collection, we treat all older generation objects as roots. We cannot traverse the entire heap to find youngest-generation references, or else youngest-generation collection will be as costly as collection of the entire heap. Therefore, generational systems, including ours, often keep track of such old-to-young references using a *card table,* an array whose entries correspond to subdivisions of the heap called cards. When mutator code updates a reference field, it also "dirties" the corresponding card table entry. The youngest-generation collector scans the card table to find these dirty entries, which are the ones whose corresponding cards might contain old-to-young references.[4]

[3] In CAS implementations of which we are aware.

[4] When a card contains old-to-young references after a collection, the collector leaves the corresponding card table entry dirty.

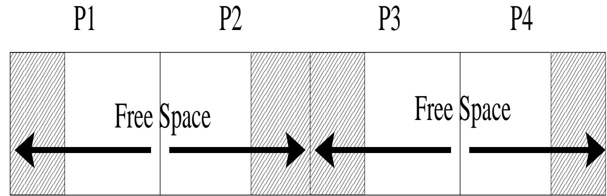



Figure 2: Parallel Compaction

For large heaps, scanning the card table may take a long time and therefore should be partitioned across threads. At first we partitioned this work in the most straightforward way: dividing the card table into consecutive contiguous blocks, which were claimed by the GC threads. Unfortunately this didn't work well on some applications, because some blocks were very dense, while others were sparse; for example, large arrays of references caused dense blocks. Scanning the dense blocks was dominating the cost of the GC. To address this problem we instead overpartitioned the card table into $N$ *strides,* each of which is a set of cards separated by intervals of $N$ cards. Thus cards $\{0, N, 2N, ...\}$ comprise one stride, cards $\{1, N+1, 2N+1, ...\}$ comprise the next, and so on. This causes dense areas to be partitioned across tasks. As usual, threads compete to claim strides.

## 5 Parallel markcompact

Our old generation uses a markcompact collector. The original sequential markcompact collector consists of four major phases:

- The marking phase, which identifies and marks live objects.
- The forwarding-pointer installation phase, which computes the new addresses live objects will have after compaction and stores these addresses as forwarding pointers in the objects' headers.
- The reference redirection phase, which updates references in live objects to the new addresses of the objects they reference.
- The compaction phase, which copies live objects to their new compacted addresses.

Our *pMarkcompact* algorithm parallelizes this single-threaded algorithm, by parallelizing each of the phases. The parallelization of the first three

phases is relatively straightforward, but the final compaction phase presented difficulties. The original sequential compaction phase compacted all live data to the low end of the heap. In the parallel case, it was difficult to ensure that one thread did not overwrite object data that another thread had yet to copy. Our solution to this problem is to break the heap into $n$ *regions,* where $n$ is the number of GC threads. Each thread claims a region and slides live objects in its own region only. (Section 5.2 discusses the criteria that influence the selection of region boundaries.) The direction to which objects are moved alternates for odd and even numbered regions. Figure 2 shows an example of a heap with 4 regions and 2 free areas after compaction. In general, a heap with $n$ regions has $\lfloor \frac{n+1}{2} \rfloor$ contiguous free areas. For practical purposes, a small number of sufficiently large contiguous free areas allows allocation as efficiently as a single free area.

The following subsections describes each parallel phase in detail.

## 5.1 Parallel marking

Similar to pSemispaces, the parallel marking phase employs the parallel GC infrastructure to statically partition the root set and to dynamically balance further marking work through work stealing. Each thread keeps a work queue of objects to be scanned for pointers to other objects. When a thread runs out of objects, it attempts to steal an object from the work queue of another thread. Unlike pSemispaces, which requires synchronization on the installation of forwarding pointers, marking is idempotent and therefore requires no synchronization.[5]

## 5.2 Parallel forwarding-pointer installation

At this point, all live objects have been marked. The next phase corresponds to the "sweep" phase of a mark-sweep collector, and also has the side-effect of computing the distribution of live data, which will guide the partitioning of the heap into the regions discussed above. First, we overpartition the heap into $m$ *units* of (roughly) equal size. (We ensure that unit boundaries are object-aligned, which leads to the approximation above.) The value of $m$ is typically $4n$, where $n$ is the number of GC threads. The GC threads compete to claim units; for each unit, the thread traverses the objects, counting the number of bytes of live data in the unit, and coalescing contiguous regions of dead objects into single blocks traversable in constant time.

When all units are processed, we know the exact amount of live data in each unit, and can partition the heap into *regions* with approximately equal amounts of live data. The partition is such that each region contains one or more of the units created in the previous pass, i.e. regions are unit-aligned. Regions are the partitions used to solve the compaction problem; they are the heap divisions in Figure 2. The region that contains an object dictates the direction in which it will be copied. Since we know how much live data is in each unit in a region, it is straightforward to calculate the new address of the first live object in a particular unit, by summing the live data of the previous units in the region (in the appropriate compaction order for the region). Thus, forwarding pointer installation can use the unit partitioning already established. GC threads dynamically claim units and install forwarding pointers in all live objects within the unit.

## 5.3 Parallel reference redirection

Redirecting object references requires scanning roots, objects in the current generation, and objects in other generations for references to objects in the current generation. The forwarding pointers inserted by the previous phase are used to update these references. We rely on the parallel GC infrastructure to balance the work of scanning the roots. Currently, the scanning of the young generation is treated as a single task; in the future, this might be further partitioned. Within the old generation we reuse the previous unit partitioning.

## 5.4 Parallel compaction

The last phase is parallel compaction. As discussed previously, we use the larger-grained region partitioning in this phase. There is a trade-off here between parallelism, which favors more, smaller partitions, and allocation efficiency, which favors fewer, larger partitions (and thus, fewer, larger free areas at the end of compaction.) We currently favor allocation efficiency, by making the region partition an exact partition (as opposed to an overpartition.) This design choice will be investigated further in the future.

[5]Note that this lack of synchronization also depends on having the mark bits present in the object; if an external marking array were used then one word might contain several marks, which would necessitate synchronization.

# 6 Results

## 6.1 Benchmarks

We present results for three benchmarks. **GCOld** is a synthetic program which can be used to present a variety of loads to a garbage collector, including large heaps requiring significant old-generation collections. **SpecJBB** is a scalability benchmark inspired by TPC-C which emulates a 3-tier system with emphasis on the middle tier. **Javac** is a compiler that translates Java programming language source code to Java class files.

The **GCOld** application allocates an array, each element of which points to the root of a binary tree about a megabyte in size. An initial phase allocates these data structures; then the program does some number of *steps*, maintaining a steady-state heap size. Each step allocates some number of bytes of short-lived data that will die in a young-generation collection, and some number of bytes of nodes in a long-lived tree structure that replaces some previously existing tree, making it garbage. Each step further simulates some amount of mutator computation by several iterations of an busy-work loop. Finally, since pointer-mutation rate can be an important factor in the performance of generational collection, each step modifies some number of pointers (in a manner that preserves the amount of reachable data). Command-line parameters control the amount of live data in the steady state, the number of steps in the run, the number of bytes of short-lived and long-lived data allocated in each step, the amount of simulated work per step, and the number of pointers modified in a step. We ran **GCOld** with 300MB of live data, allocating three bytes of short-lived data for every byte of long-lived data.

**SpecJBB** is a "throughput-based" benchmark: it measures the amount of work accomplished in a fixed amount of time, rather than the amount of time required to accomplish a fixed amount of work. To create runs that can be compared to determine parallel speedup for GC, we run with a fixed number (8) of "warehouses" (i.e., mutator threads), and considered only the first 500 collections of each run. We believe the mutator behavior between these collections is sufficiently similar to make these runs comparable.

Each graph is annotated with heap configuration parameters used for the runs. A heap configuration specifies the sizes of the young and old generations (which are fixed in all our experiments.) For example, `16m:600m` indicates a young generation of 16 MB and an old generation size of 600 MB. The number of young- and old-generation collections is similar across all runs, including the sequential run, since allocation behavior is largely unaffected by collection algorithm. (We discuss an exception below.)

The runs were performed on a Sun Enterprise™ 3500 server, with 8 336 MHz UltraSPARC™ processors sharing 2 Gbyte of memory. The collector we ran was a generational collector with a parallel semispaces young generation and a parallel mark-compact old generation.

## 6.2 Scalability

Figure 3 presents our results in terms of scalability graphs. The x-axis is the number of processors. The y-axis shows speedup relative to the performance of the parallel collector run on one processor. We also show the curve for linear speedup and the performance of the sequential form of each GC algorithm. Speedups for the young generation and old generation are shown on separate graphs; speedups are calculated on the basis of total time for collections of the given type.

Table 1 gives the average and total GC times for the sequential runs and the parallel runs with one and eight processors.

| | seq | par(1) | par(8) |
|---|---|---|---|
| **GCOld** | | | |
| young avg (ms) | 216 | 298 | 53 |
| young total (s) | 211.35 | 290.83 | 53.27 |
| old avg (ms) | 13920 | 19498 | 3632 |
| old total (s) | 876.98 | 1228.42 | 221.58 |
| **SpecJBB** | | | |
| young avg (ms) | 204 | 255 | 54 |
| young total (s) | 99.59 | 124.29 | 26.69 |
| old avg (ms) | 1880 | 3315 | 761 |
| old total (s) | 26.32 | 46.41 | 10.66 |
| **Javac** | | | |
| young avg (ms) | 33 | 40 | 10 |
| young total (s) | 2.26 | 2.69 | .66 |
| old avg (ms) | 421 | 524 | 160 |
| old total (s) | 1.68 | 2.09 | .96 |

Table 1: Average and total collection times

## 6.3 Discussion

We outperform the sequential algorithm using only two processors in most cases. The only case where we required 3 processors was in pMarkcompact for **SpecJBB**. Our hypothesis is that this is due to an optimization present in the sequential markcompact collector which we have not yet

adapted to the parallel version. This *dense prefix* optimization avoids copying large blocks of data when there is only a small amount of free area to be reclaimed. In many applications this optimization eliminates a significant fraction of markcompact copying costs. We hope to adapt this technique to realize similar savings in the parallel version.

We achieve speedup factors on 8 processors of between 4 and 5.5, with the exception of the old-generation collections of **Javac.** One reason for this is that that there were 6 old-generation collections in the 8-processor run, but only 4 in the 1-processor and sequential runs. We believe that this increase is caused by fragmentation introduced by parallel LAB allocation during young-generation collection, and thus is an inherent cost of parallel collection. Note, however, that **Javac** has by far the smallest heaps of the benchmark runs. In larger problem sizes this effect is much less significant.

In the parallel mark-compact collector, we can measure the scalability of the individual phases separately. It turns out that all phases scale about as well as overall collection. For example, in **SpecJBB**, the overall 8-processor old-generation speedup is 4.351, and the speedups of the individual phases range from 3.7, for installing forwarding pointers and redirecting references, to 5.0 for sweeping. So no particular phase stands out as a clear scalability bottleneck. Still, clearly further work is needed to attempt to increase scalability (or explain the factors that inhibit it).

## 7 Conclusions

After exploring parallel techniques, and implementing two parallel collectors, we believe that there is great potential for improving both pause times and throughput using parallelism.

Large multi-threaded applications are being written in garbage-collected languages. These applications require heaps in the gigabyte range and beyond. Sequential GC algorithms will become an ever-greater scaling bottlneck. If systems intended to support such applications stop all threads for garbage collection, they must use parallel techniques to avoid this bottleneck.

## 8 Trademarks

Sun, Sun Microsystems, Sun Enterprise, JVM, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

## References

[1] Nimar S. Arora; Robert D. Blumofe; and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[3] The SPARC Architecture Manual Version 9, Sun Microsystems, Inc.

[4] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M.Steffens. On-the-fly garbage collection: An exercise in cooperation. *CACM*, 21(11):966–975, November 1988.

[5] Damien Doligez and Georges Gonthier. Portable unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 1994 ACM Conference on Principles of Programming Languages*, pages 70–80, 1994.

[6] Robert H. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *1984 ACM Symposium on LISP and Functional Programming*, pages 9–17, New York, NY, 1984. ACM.

[7] Richard Jones and Rafael Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[8] John R. Ellis; Kai Li; and Andrew W. Appel. Real-time Concurrent Collection on Stock Multiprocessors. Technical Report 25, Digital Equipment Corporation Systems Research Center, February 1988.

[9] G. L. Jr. Steele. Multiprocessing compactifying garbage collection. *CACM*, 18(9):495–508, September 1975.

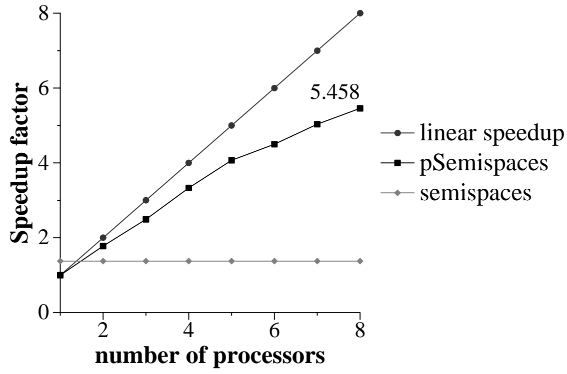
[10] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. *ACM SIGPLAN Notices*, 36(1):18–24, January 2001.

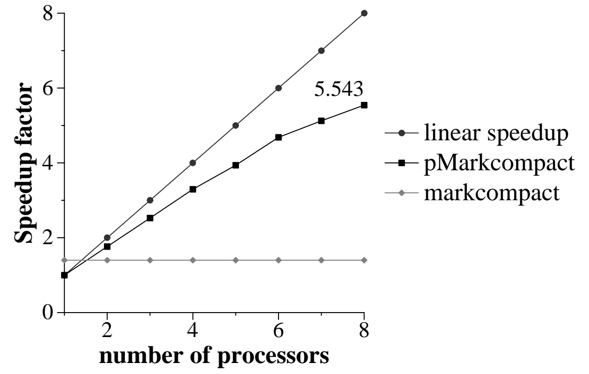
[11] Toshio Endo; Kenjiro Taura; and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. *Proceedings of High Performance Networking and Computing (SC97)*, 1997.

[12] Derek White and Alex Garthwaite. The GC interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1998.

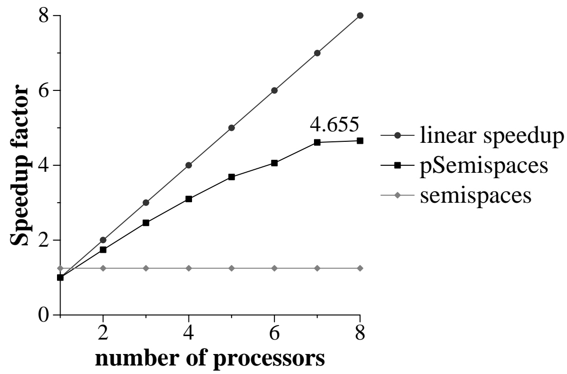
[13] Paul R. Wilson. Uniprocessor garbage collection techniques. *International Workshop on Memory Management, Springer-Verlag Lecture Notes in Computer Science*, 1992.

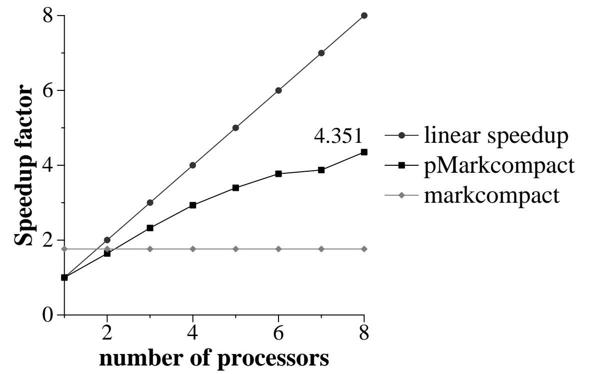
**GCOld(16m:600m)**



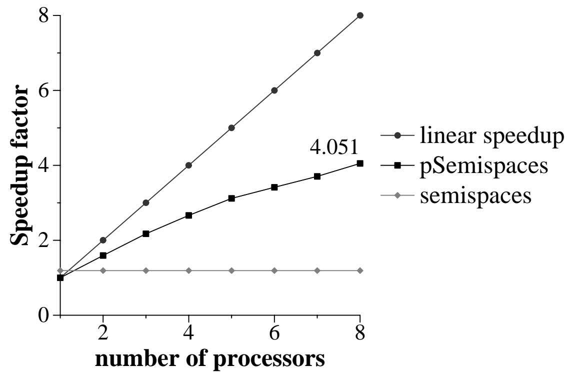




**SpecJBB(16m:300m)**




**Javac(4m:12m)**



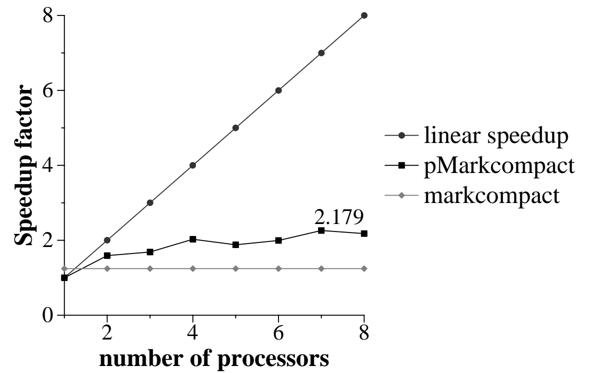




Figure 3: Speedup graphs