A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines

Toshio Endo Kenjiro Taura Akinori Yonezawa
{endo,tau,yonezawa}@is.s.u-tokyo.ac.jp
Department of Information Science, Faculty of Science
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Abstract

This work describes implementation of a mark-sweep garbage collector (GC) for shared-memory machines and reports its performance. It is a simple "parallel" collector in which all processors cooperatively traverse objects in the global shared heap. The collector stops the application program during a collection and assumes a uniform access cost to all locations in the shared heap. Implementation is based on the Boehm-Demers-Weiser conservative GC (Boehm GC). Experiments have been done on Ultra Enterprise 10000 (Ultra Sparc processor 250 MHz, 64 processors). We wrote two applications, BH (an N-body problem solver) and CKY (a context free grammar parser) in a parallel extension to C++.

Through the experiments, We observe that load balancing is the key to achieving scalability. A naive collector without load redistribution hardly exhibit speed-up (at most fourfold speed-up on 64 processors). Performance can be improved by dynamic load balancing, which exchanges objects to be scanned between processors, but we still observe that straightforward implementation severely limits performance. First, large objects become a source of significant load imbalance, because the unit of load redistribution is a single object. Performance is improved by splitting a large object into small pieces before pushing it onto the mark stack. Next, processors spend a significant amount of time uselessly because of serializing method for termination detection using a shared counter. This problem suddenly appear on more than 32 processors. By implementing non-serializing method for termination detection, the idle time is eliminated and performance is improved. With all these careful implementation, we achieved average speed-up of 28.0 in BH and 28.6 in CKY on 64 processors.

Keywords: garbage collection, parallel algorithm, shared-memory machine, scalability, dynamic load balancing

1 Introduction

Shared-memory architecture is attractive platform for implementation of general-purpose parallel programming languages that support irregular, pointer-based data structures [4, 19]. The recent progress in scalable shared-memory technologies is also making these architectures attractive for high-performance, massively parallel computing.

One of the important issues not yet addressed in the implementation of general-purpose parallel programming languages is scalable garbage collection (GC) technique for shared-heaps. Most previous work on GC for shared-memory machines is concurrent GC [6, 10, 16], by which we mean that the collector on a dedicated processor runs concurrently with application programs, but does not perform collection itself in parallel. The focus has been on shortening pause time of applications by overlapping the collection and the applications on different processors. Having a large number of processors, however, such collectors may not be able to catch up allocation speed of applications. To achieve scalability, we should parallelize collection itself.

This paper describes the implementation of a parallel mark-sweep GC on a large-scale (up to 64 processors), multiprogrammed shared-memory multiprocessor and presents the results of empirical studies of its performance. The algorithm is, at least conceptually, very simple; when an allocation requests a collection, the application program is stopped and all the processors are dedicated to collection. Despite its simplicity, achieving scalability turned out to be a very challenging task. In the empirical study, we found a number of factors that severely limit the scalability, some of which appear only when the number of processors becomes large. We show how to eliminate these factors and demonstrate the speed-up of the collection. At present, we achieved approximately 28-fold speed-up on 64 processors.

We implemented the collector by extending the Boehm-Demers-Weiser conservative garbage collection library (Boehm GC [2, 3]) on a 64-processor Ultra Enterprise 10000 system. The heart of the extension is dynamic task redistribution through exchanging contents of the mark stack (i.e., data that are live but yet to be examined by the collector).

The rest of the paper is organized as follows. Section 2 compares our approach with previous work. Section 3 briefly summarizes Boehm GC, on which our collector is based. Section 4 describes our parallel marking algorithm and solutions for performance limiting factors. Section 5 shows experimental results, and we conclude in Section 6.

2 Previous Work

Most previous published work on GCs for shared-memory machines has dealt with **concurrent GC** [6, 10, 16], in which only one processor performs a collection at a time. The focus of such work is not on the scalability on large-scale or medium-scale shared-memory machines but on shortening pause time by overlapping GC and the application by utilizing multiprocessors. When GC itself is not parallelized, the collector may fail to finish a single collection cycle before the application exhausts the heap (Figure 1). This will occur on large-scale machines, where the amount of live data will be large and the (cumulative) speed of allocation will be correspondingly high.

We are therefore much more interested in "parallel" garbage collectors, in which a single collection is performed cooperatively by all the processors. Several systems use this type of collectors [7, 15] and we believe there are many unpublished work too, but there are relatively few published performance results. To the authors' knowledge, the present paper is the first published work that examines the scalability of parallel collectors on real, large-scale, and multiprogrammed shared-memory machines. Previous publications have reported only preliminary measurements or have examined scalability only by simulation.

Ichiyoshi and Morita proposed a parallel copying GC for a shared heap [11]. It

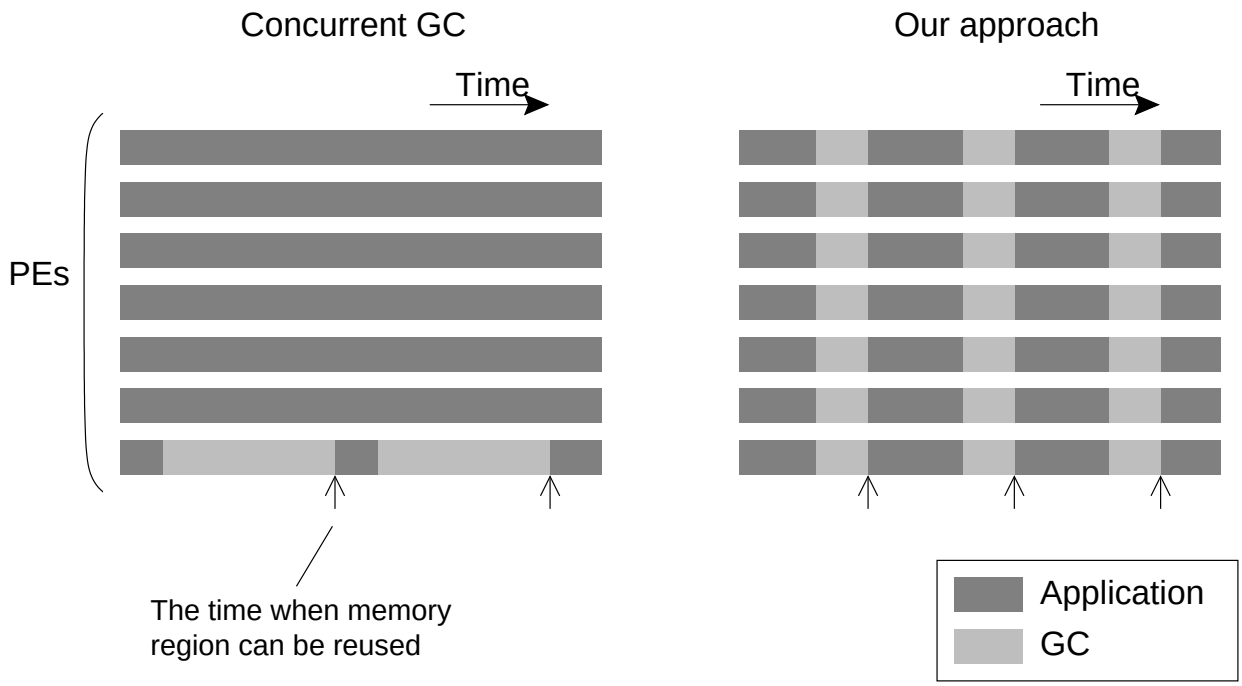

Figure 1: Difference between concurrent GC and our approach. If only one dedicated processor performs GC, a collection cycle becomes longer in proportion to the number of processors.

assumes that the heap is divided into several local heaps and a single shared heap. Data move from a local heap to the shared heap, maintaining the invariant that there are no pointers from the shared heap to a local heap. Each processor collects its local heap individually. Collection on the shared-heap is done cooperatively but asynchronously. During a collection, live data in the shared-heap (called 'from-space' of the collection) are copied to another space called 'to-space'. Each processor, on its own initiative, copies data that is reachable from its local heap to to-space. Once a processor has copied data reachable from its local heap, it can resume application on that processor, which moves data in its local heap to the new shared-heap (i.e., to-space). Consult [11] for how to deal with potential overflow in the to-space.

Our collector is much simpler than Ichiyoshi and Morita's collector; it simply synchronizes all the processors at a collection and all the processors are dedicated to the collection until all reachable objects are marked. Although they have not mentioned explicitly, we believe that a potential advantage of their method over ours is its lower susceptibility to load imbalance of a collection. That is, the idle time that would appear in our collector is effectively filled by the application. The performance measurement in Section 5 shows a good speed-up up to our maximum configuration, 64 processors, and indicates that there is no urgent need to consider using the application to fill the idle time. We prefer our method because it does not interfere with SPMD-style applications, in which global synchronizations are frequent.[1] Their method may interact badly

[1] A global synchronization occurs even if the programming language does not provide explicit barrier synchronization primitives. It implicitly occurs in many places, such as reduction and termination detection.

with such applications because it exhibits a very long marking cycle, during which the applications cannot utilize all the processors. We also reached a similar conclusion on distributed-memory machines [20].

Our collector algorithm is most similar to Imai and Tick's parallel copying collector [12]. In their study, all processors perform copying tasks cooperatively and any memory object in one shared heap can be copied by any processor. Dynamic load balancing is achieved by exchanging memory pages to be scanned in the to-space among processors. Speed-up is calculated by a simulation that assumes processors become idle only because of load imbalance—the simulation overlooks other sources of performance degrading factors such as spin-time for lock acquisition. As we will show in Section 5, such factors become quite significant, especially in large-scale and multiprogrammed environments.

3 Boehm-Demers-Weiser Conservative GC Library

The Boehm-Demers-Weiser conservative GC library (Boehm GC) is a mark-sweep GC library for C and C++. The interface to applications is very simple; it simply replaces calls to `malloc` with calls to `GC_MALLOC`. The collector automatically reclaims memory no longer used by the application. Because of the lack of precise knowledge about types of words in memory, a conservative GC is necessarily a mark-sweep collector, which does not move data. Boehm GC supports parallel programs using Solaris threads. The current focus seems to support parallel programs with minimum implementation efforts; it serializes all allocation requests and GC is not parallelized. Below we describe aspects of Boehm GC that are relevant to subsequent sections.

3.1 Heap Blocks and Mark Bitmaps

Boehm GC manages a heap in units of 4-KB blocks, called **heap blocks**. Objects in a single heap block must have the same size. For each block separate header record (**heap block header**) is allocated that contains information about the block, such as the size of the objects in it. Also kept in the header is a **mark bitmap** for the objects in the block. A single bit is allocated for each word (32 bits in our experimental environments). Put differently, each word in a mark bitmap describes the states of 32 consecutive words in the corresponding heap block, which may contain multiple small objects. Therefore, in parallel GC algorithms, visiting and marking an object must explicitly be done atomically. Otherwise, if two processors simultaneously mark objects that share a common word in a mark bitmap, either of them may not be marked properly.

3.2 Mark Stack

Boehm GC maintains marking tasks to be performed with a vector called **mark stack**. It keeps track of objects that have been marked but may directly point to an unmarked object. Each entry is represented by two words:

- the beginning address of an object, and
- the size of the object.

```
push all roots (registers, stack, global variables) onto mark stack.
while (mark stack is not empty) {
    o = pop(mark stack)
    for (i = 0; i < size of o; i++) {
        if (o[i] is not a pointer) do nothing
        else if (mark bit of o[i] == 'marked') do nothing
        else {
            push(o[i], mark stack)
            mark bit of o[i] = 'marked'
        }
    }
}
```

Figure 2: The marking process with a mark stack.

Figure 2 shows the marking process in pseudo code; each iteration pops an entry from the mark stack and scans the specified object,[2] possibly pushing new entries onto the mark stack. A mark phase finishes when the mark stack becomes empty.

3.3 Sweep

In the sweep phase, Boehm GC examines the mark bitmaps of all heap blocks in the heap. A heap block that contains any marked object is linked to a list called **reclaim list**, to prepare for future allocation requests. Heap blocks that are found empty are linked to a list called **heap block free list**, in which heap blocks are sorted by their addresses, and adjacent ones are coalesced to form a large contiguous block. Heap block free list is examined when an allocation cannot be served from a reclaim list.

4 Parallel GC Algorithm

Our collector supports parallel programs that consist of several UNIX processes. We assume that all processes are forked at the initialization of a program and are not added to the application dynamically. Interface to the application program is the same as that of the original Boehm GC; it provides `GC_MALLOC`, which now returns a pointer to shared memory (acquired by a `mmap` system call).

We could alternatively support Solaris threads. The choice is arbitrary and somewhat historical; we simply thought having private global variables makes implementation simpler. We do not claim one is better than the other.

4.1 Basic Algorithm

In the parallel marking algorithm, each processor has its own local mark stack. When GC is invoked, all application processes are suspended by sending signals to them. When

[2] More precisely, when the specified object is very large (> 4 KB), the collector scans only the first 4 KB and keeps the rest in the stack.

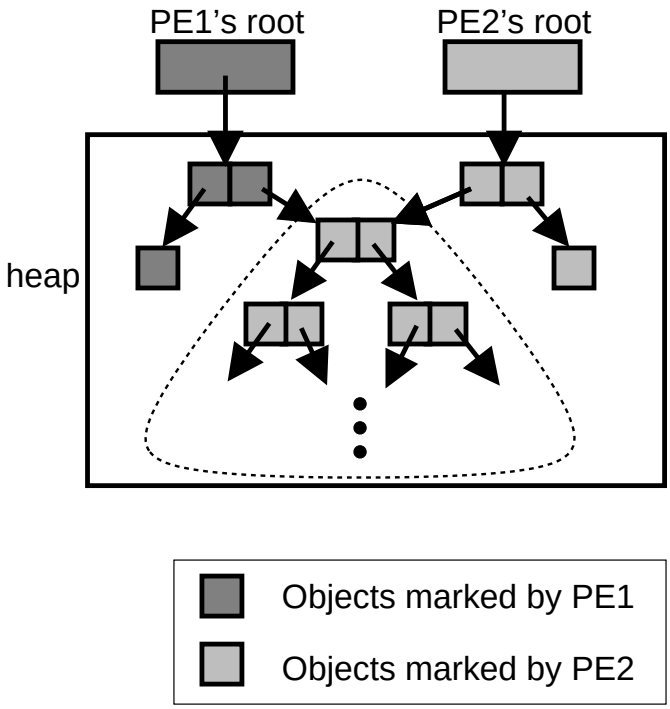

Figure 3: In the simple algorithm, all nodes of a shared tree are marked by one processor.

all the signals have been delivered, every processor starts marking from its local root, pushing objects onto its local mark stack. When an object is marked, the corresponding word in a mark bitmap is locked before the mark bit is read. The purpose of the lock is twofold. One is to ensure that a live object is marked exactly once, and the other is to atomically set the appropriate mark bit of the word. When all reachable objects are marked, the mark phase is finished.

This naive parallel marking hardly results in any recognizable speed-up because of the imbalance of marking tasks among processors. Load imbalance is significant when a large data structure is shared among processors through a small number of externally visible objects. For example, a significant imbalance is observed when a large tree is shared among processors only through a root object. In this case, once the root node of the tree is marked by one processor, so are all the internal nodes (Figure 3). To improve marking performance, our collector performs dynamic load balancing by exchanging entries stored in mark stacks.

Besides a local mark stack, each processor maintains an additional data structure named **stealable mark queue**, through which "tasks" (entries in mark stacks) are exchanged (Figure 4). During marking, each processor periodically checks its stealable mark queue. If it is empty, the processor moves all the entries in the local mark stack (except entries that point to the local root, which can be processed only by the local processor) to the stealable mark queue. When a processor becomes idle (i.e., when its mark stack becomes empty), it tries to obtain tasks from stealable mark queues. The processor examines its own stealable mark queue first, and then those of other processors, until it finds a non-empty queue. Once it finds one, it steals half of the entries[3] in the

[3]If the queue has n entries and n is an odd number, $(n+1)/2$ entries are stolen.

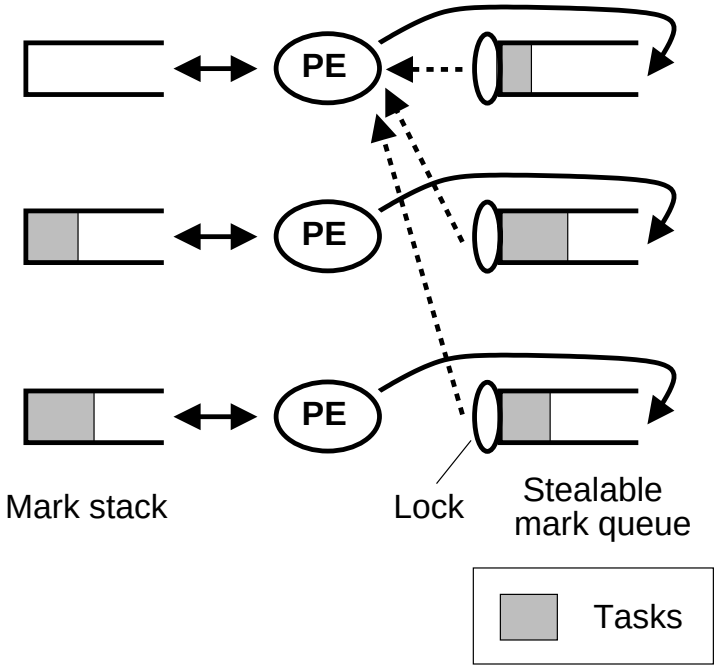

Figure 4: Dynamic load balancing method: tasks are exchanged through stealable mark queues.

queue and stores them into its mark stack. Because several processors may become idle simultaneously, this test-and-steal operation must acquire a lock on a queue. The mark phase is terminated when all the mark stacks and stealable mark queues become empty. The termination is detected by using a global counter to maintain the number of empty stacks and empty queues. The counter is updated whenever a processor becomes idle or obtains tasks.

In the parallel algorithm, all processors share a single heap block free list, while each processor maintains a local reclaim list. In the sweep phase, each processor examines a part of heap blocks and links empty ones to the heap block free list and non-empty ones to its local reclaim list. Since each processor has a local reclaim list, inserting blocks to a reclaim list is straightforward. Inserting blocks to the heap block free list is, however, far more difficult, because the heap block free list is shared, blocks must sorted by their addresses, and adjacent blocks must be coalesced. To reduce the contention and the overhead on the shared list, we make the unit of work distribution in the sweep phase larger than a single heap block and perform tasks as locally as possible; each processor acquires a large number of (64 in the current implementation) contiguous heap blocks at a time and processes them locally. Empty blocks are locally sorted and coalesced within the blocks acquired at a time and accumulated in a local list called **partial heap block free list**. Each processor repeats this process until all the blocks have been examined. Finally, the lists of empty blocks accumulated in partial heap block free lists are chained together to form the global heap block free list, possibly coalescing blocks at joints. When this sweep phase is finished, we restart the application.

4.2 Performance Limiting Factors and Solutions

The basic marking algorithm described above exhibits acceptable speed-up on small-scale systems (e.g., approximately fourfold speed-up on eight processors). As we will see in Section 5, however, several factors severely limit speed-up and this basic form never yields more than a 12-fold speed-up. Below we list these factors and describe how did we address them in turn.

Load imbalance by large objects: We often found that a large object became a source of significant load imbalance. Recall that the smallest unit of task distribution is a single entry in a stealable mark queue, which represents a single object in memory. This is still too large! We often found that only some processors were busy scanning large objects, while other processors were idle. This behavior was most prominent when applications used many stacks or large arrays. In one of our parallel applications, the input data, which is a single 400-KB array caused significant load imbalance. In the basic algorithm, it was not unusual for some processors to be idle during the entire second half of a mark phase.

We address this problem by splitting large objects (objects larger than 512 bytes) into small (512-byte) pieces before it is pushed onto the mark stack. In the experiments described later, we refer to this optimization as SLO (Split Large Object).

Delay in searching tasks: In the basic algorithm, processors sometimes spent a significant amount of time acquiring locks on stealable mark queues. This was particularly the case when many processors compete for trying to obtain tasks. When a processor fails to acquire a lock, it should examine the next queue immediately. We therefore replaced a simple lock-then-steal sequence with a try-lock-then-steal sequence. When a lock acquisition fails, the processor simply gives up acquiring the lock and examines the next queue immediately. We refer to this optimization as SLQ (Skip Locked Queues).

Serialization in termination detection: When the number of processors becomes large, we found that the GC speed suddenly dropped. It revealed that processors spent a significant amount of time to acquire a lock on the global counter that maintains the number of empty mark stacks and empty stealable mark queues. We updated this counter each time a stack (queue) became empty or tasks were thrown into an empty stack (queue). This serialized update operation on the counter introduced a long critical path in the collector.

We implemented another termination detection method in which two flags are maintained by each processor; one tells whether the mark stack of the processor is currently empty and the other tells whether the stealable mark queue of the processor is currently empty. Since each processor maintains its own flags on locations different from those of the flags of other processors, setting flags and clearing flags are done without locking.

Termination is detected by scanning through all the flags in turn. To guarantee the atomicity of the detecting process, we maintain an additional global flag **detection-interrupted**, which is set when a collector recovers from its idle state. A detecting processor clears the detection-interrupted flag, scans through all the flags until it

finds any non-empty queue, and finally checks the detection-interrupted flag again if all queues are empty. It retries if the process has been interrupted by any processor. We must take care of the order of updating flags lest termination be detected by mistake. For example, when processor A steals all tasks of processor B, we need to change flags in the following order: (1) stack-empty flag of A is cleared, (2) detection-interrupted flag is set, and (3) queue-empty flag of B is set. We refer to this optimization as NSB (Non-Serializing Barrier).

Delay in testing mark bitmap: We observed cases where processors consumed a significant amount of time acquiring locks on mark bits. A simple way to guarantee that a single object is marked only once is to lock the corresponding mark bit (more precisely, the word that contains the mark bit) before reading it. However, this may unnecessarily delay processors that read the mark bit of an object to just know the object is already marked. To improve the sequence, we replaced this "lock-and-test" sequence with a "test-and-compare&swap" sequence; we first read the mark bit without lock and quit if the bit is already set. Otherwise, we calculate the new bitmap for the word and swap the word in the original location and the new bitmap, if the original location is the same as the originally read bitmap. This compare&swap is done atomically by one instruction. We retry if the location has been overwritten by another process. These operations eliminate useless lock acquisitions on mark bits that are already set. We refer to this optimization as TCS (Test-and-Compare&Swap) in the experiments below.

Another advantage of this algorithm is that it is a non-blocking algorithm [8, 17, 18], and hence does not suffer from untimely preemption. A major problem with the basic algorithm is, however, that locking a word in a bitmap every time we check if an object is marked causes contention (even in the absence of preemption). We confirmed that a "test-and-lock-and-test" sequence that checks the mark bit before locking works equally well, though it is a blocking algorithm.

5 Experimental Results

We have implemented the collector on the Ultra Enterprise 10000 (Ultra Sparc processor 250 MHz, 64 processors). The implementation is based on the source code of Boehm GC version 4.10, and the added code is about 3000 lines in C language. We used two applications, BH (an N-body problem solver) and CKY (a context free grammar parser) written in a parallel extension to C++ [14]. This extension allows application programmers to create user-level threads dynamically. In the experiment, the stack size of each thread was 8 KB. The stacks were allocated by `GC_MALLOC` and became garbage after the thread was terminated. BH simulates the motion of N (= 5000) particles by using the Barnes-Hut algorithm [1]. At each time step, BH makes a tree whose leaves correspond to particles and calculates the acceleration, speed, and location of the particles by using the tree. CKY takes sentences written in natural language and the syntax rules of that language as input, and outputs all possible parse trees for each sentence. In the experiment, each of the given 67 sentences consists of 10 to 40 words.

5.1 Evaluation Framework

Ideally, the speed-up of the collector should be measured by using various numbers of processors and applying the algorithm to the same snapshot of heap. It is difficult, however, to reproduce the same snapshot multiple times because of the indeterminacy of application programs. The amount of data is so large that we cannot simply dump the entire image of the heap. Even if such dumping were feasible, it would still be difficult to continue from a dumped image with a different number of processors. Thus the only feasible approach is to formulate the amount of work needed to finish a collection for a given heap snapshot and then calculate how fast the work is processed at each occurrence of a collection.

A generally accepted estimation of the workload of marking for a given heap configuration is the amount of live objects, or equivalently, the number of words that are scanned by the collector. This, however, ignores the fact that the load on each word differs depending on whether it is a pointer, and the density of pointers in a live data may differ from one collection to another. Given a word in heap, Boehm GC first performs a simple test that rules out most non-pointers and then examines the word more elaborately.

To measure the speed-up more accurately, we define the workload W of a collection as

$$W = a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_4 + a_5 x_5$$

where x_1 is the number of marked objects, x_2 the number of times to scan already marked objects, x_3 the number of times to scan non-pointers, x_4 the number of empty heap blocks, and x_5 the number of non-empty blocks. Each x_n is totaled over all processors. The GC speed S is defined as $S = W/t$, where t is the elapsed time of the collection. And the GC speed-up on N processors is the ratio of S on N processors to S on a single processor. When we measure S on a single processor, we eliminate overhead for parallelization. The constants a_n were determined through a preliminary experiment. To determine a_3, for example, we created a 1000-word object that contained only non-pointers and we measured the time to scan the object. We ran this measurement several times and used the shortest time. The other constants were determined similarly. In the experiment, the constants were set at $a_1 = 0.50$, $a_2 = 0.16$, $a_3 = 0.020$, $a_4 = 2.0$, and $a_5 = 1.3$.

5.2 Speed-up of GC

Table 1 and Figures 5-8 show performance of GC using the two applications. We measured eight versions of collectors, each of which corresponds to a row of the table. "Sequential" refers to the original Boehm GC and "Simple" refers to the algorithm in which each processor simply marks objects that are reachable from the root of that processor without any further task distribution. "Basic" refers to the basic algorithm described in Section 4.1, and the following four versions refer to ones that implement all but one of the optimizations described in Section 4.2. "No-XXX" stands for a version that implements all the optimizations but XXX. Finally, "Full" is the fully optimized version.

The applications were executed four times in each configuration and invoked collections more than 40 times. The table shows the average performance of the invocations. When we used all or almost all the processors on the machine, we occasionally observed invocations that performed distinguishably worse than the usual ones. They were typically 10 times worse than the usual ones. The frequency of such unusually bad invocations

was about once in every five invocations when we used 64 processors. We have not yet determined the reason for these invocations. It might be the effect of other processes. For the purpose of this study, we exclude these cases.

Figure 5 and 6 compare three versions, namely, Simple, Basic, and Full. The graph shows that Simple does not exhibit any recognizable speed-up in either application. Basic performs reasonably in CKY until 32 processors. However, it does not scale any more beyond it. Full achieved a 28.6-fold speed-up in CKY and a 28.0-fold speed-up in BH on 64 processors.

Figure 7 and 8 show how each optimization affects scalability. Removing any particular optimization yields a sizable degradation in performance, especially when we have a large number of processors. Without the improved termination detection by the non-serializing barrier (NSB), neither application achieves more than a 17-fold speed-up. Sensitivity to optimizations differs between the two applications. Most notably, the lack of TCS has the most significant impact on BH, whereas it is benign in CKY. The experiment revealed that BH exhibits a much larger value of x_2/x_1. That is, an object in BH tends to be visited by the collector much more than an object in CKY is. The value is approximately five on BH, while close to zero in CKY. The value for CKY is quite understandable; the reference count for an object is one most of the time, so an object is rarely visited several times. In BH we found that GC often misidentifies floating-point numbers, such as the location of particles, as pointers. Furthermore, GC finds the same floating-point numbers several time and takes them as pointers. So there is a greater chance that processors will visit the same (misidentified) pointers simultaneously in BH than there is in CKY, which does not use floating-point numbers. Thus the performance of the No-TCS version is poor in BH. This result in BH is peculiar to conservative GC. The performance of No-TCS in BH would be closer to that of Full if GC had information about types of objects.

6 Conclusion

We constructed a highly scalable parallel mark-sweep garbage collector for shared-memory machines. This collector performs dynamic load balancing by exchanging objects in mark stacks. Through the experiments on the shared-memory machine Ultra Enterprise 10000, we found that a number of factors severely limit the scalability, and we presented the following solutions: (1) Because the unit of load balancing was a single object, a large object that cannot be divided degraded the utilization of processors. Splitting large objects into small parts when they are pushed onto the mark stack enabled a better load balancing. (2) Idle processors were sometimes delayed for lock acquisitions on the stealable mark queues. They can obtain tasks faster by skipping queues already locked by another processor. (3) Especially on 32 or more processors, processors wasted a significant amount of time because of the serializing operation used in the termination detection with a global counter. We implemented another method using local flags without locking, and the long critical path was eliminated. (4) We observed that processors spent a significant time for lock acquisitions on mark bits in BH. The useless lock acquisitions were eliminated by using a "test-compare&swap" sequence instead of a "lock-and-test" sequence. When using all these solutions, we achieved an average speed-up of 28.0 to 28.6 on 64 processors.

CKY

		1PE	8PE	16PE	32PE	48PE	64PE
Sequential	speed	367.4	-	-	-	-	-
	speed-up	1.00	-	-	-	-	-
Simple	speed	323.2	388.2	399.3	392.6	397.8	395.3
	speed-up	0.880	1.06	1.09	1.07	1.08	1.08
Basic	speed	320.5	1659	2815	4403	3642	2359
	speed-up	0.872	4.52	7.66	12.0	9.91	6.42
No-SLO	speed	329.0	2106	3667	6311	7328	7098
	speed-up	0.895	5.73	9.98	17.2	19.9	19.3
No-SLQ	speed	344.8	2254	3841	6422	7910	10229
	speed-up	0.938	6.14	10.5	17.5	21.5	27.8
No-NSB	speed	336.5	2200	3747	6178	3355	1857
	speed-up	0.915	5.99	10.2	16.8	9.13	5.05
No-TCS	speed	314.1	1691	3124	5188	6562	8341
	speed-up	0.855	4.60	8.50	14.1	17.9	22.7
Full	speed	345.1	2271	4085	6778	8201	10523
	speed-up	0.939	6.18	11.1	18.4	22.3	28.6

BH

		1PE	8PE	16PE	32PE	48PE	64PE
Sequential	speed	406.2	-	-	-	-	-
	speed-up	1.00	-	-	-	-	-
Simple	speed	343.5	757.8	768.3	890.1	806.5	1464
	speed-up	0.846	1.87	1.89	2.19	1.99	3.60
Basic	speed	326.2	1511	1669	1819	1438	1090
	speed-up	0.803	3.72	4.11	4.48	3.54	2.68
No-SLO	speed	398.9	1744	1470	3087	2315	2583
	speed-up	0.982	4.29	3.62	7.60	5.70	6.36
No-SLQ	speed	388.4	2477	4265	6694	8065	11044
	speed-up	0.956	6.10	10.5	16.5	19.9	27.2
No-NSB	speed	382.7	2534	4085	3433	2358	1520
	speed-up	0.942	6.24	10.1	8.45	5.81	3.74
No-TCS	speed	345.9	2006	3237	2308	1581	1151
	speed-up	0.852	4.94	7.97	5.68	3.89	2.83
Full	speed	381.3	2570	4364	7001	8464	11368
	speed-up	0.939	6.33	10.7	17.2	20.8	28.0

Sequential Sequential code without overhead for parallelization.

Simple Parallelized but no load balancing is done.

Basic Only load balancing is done.

No-SLO All optimizations but SLO (splitting large objects) are done.

No-SLQ All optimizations but SLQ (skipping locked queue) are done.

No-NSB All optimizations but NSB (non serializing barrier) are done.

No-TCS All optimizations but TCS (test and compare&swap) are done.

Full All optimizations are done.

Table 1: Average speed and speed-up of marking in CKY and BH. The speed is the amount of workload executed per 1 millisecond. The speed-up is the ratio of the speed to that of sequential code on single processor.

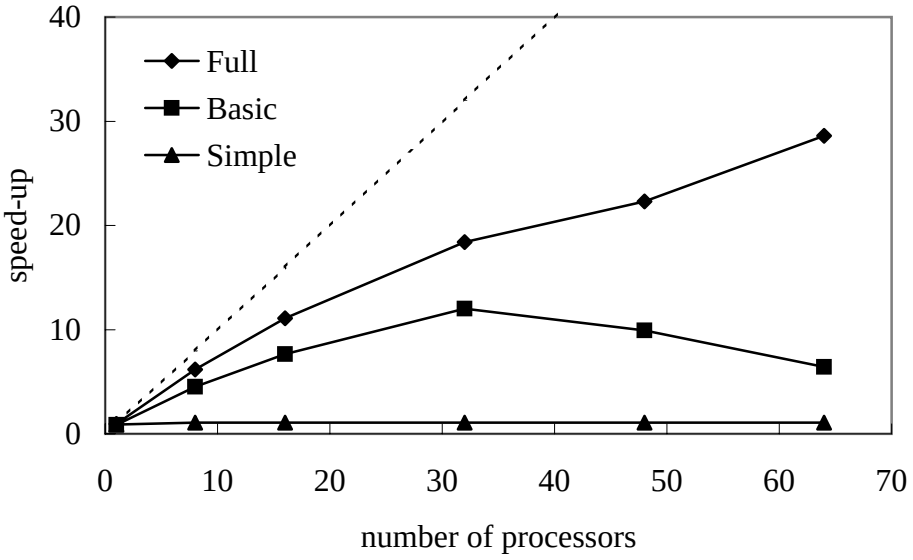

Figure 5: Average marking speed-up in CKY.

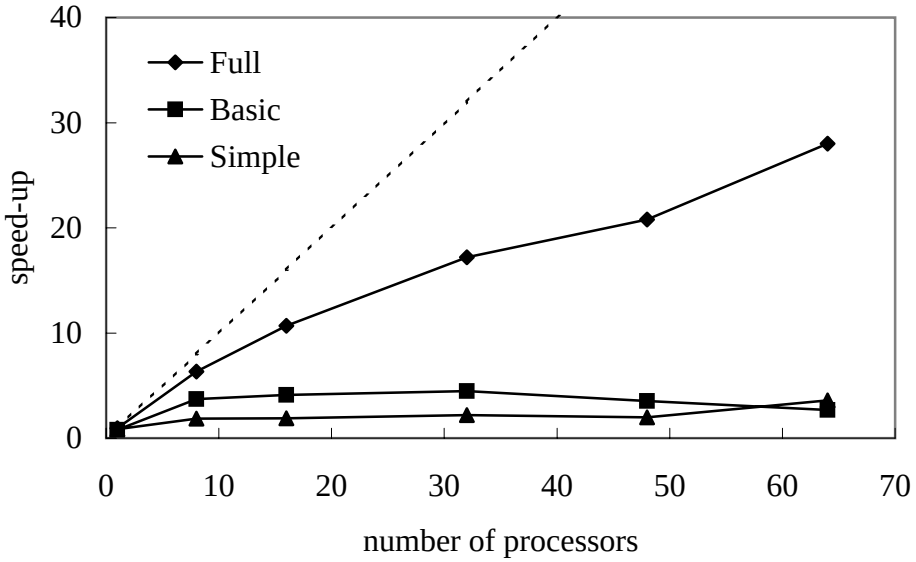

Figure 6: Average marking speed-up in BH.

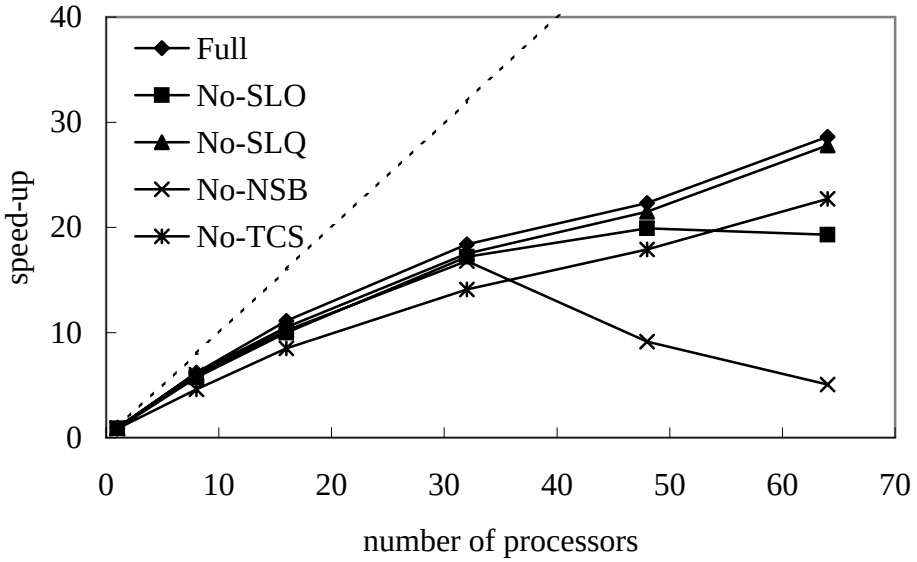

Figure 7: Effect of each optimization in CKY.

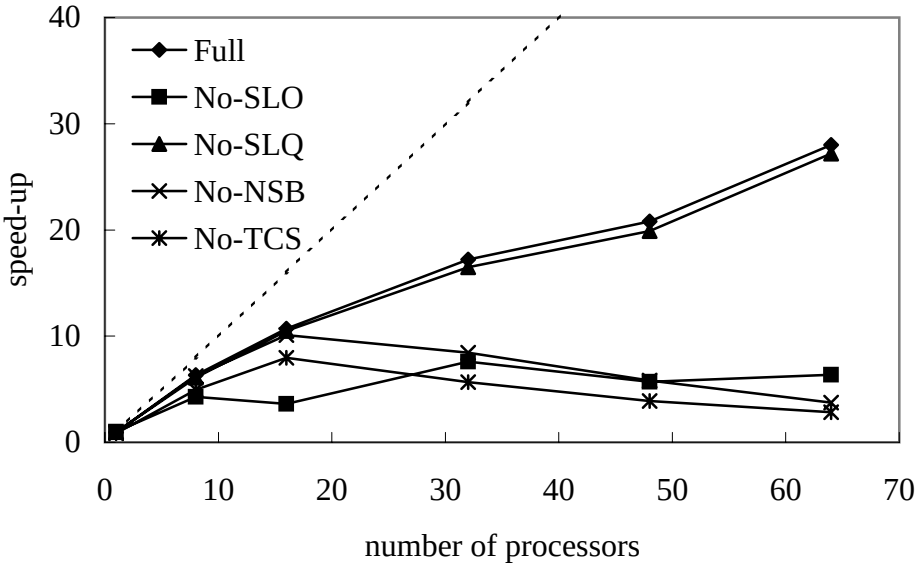

Figure 8: Effect of each optimization in BH.

References

[1] Josh Barnes and Piet Hut. A hirarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[2] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[3] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[4] A. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ - a C++ dialect for high performance parallel computing. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*, volume 1049 of *Lecture Notes in Computer Science*, pages 76–95, 1996.

[5] David L. Detlefs. Concurrent garbage collection for C++. In *Topics in Advanced Language Implementation*, chapter 5, pages 101–134. The MIT Press, 1991.

[6] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multithreaded implementation of ML. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123, January 1993.

[7] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transaction on Programming Languages and Systems*, 7(3):501–538, July 1985.

[8] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[9] Maurice P. Herlithy and J. Eliot B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

[10] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (Im)mutable data. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 1993.

[11] Nobuyuki Ichiyoshi and Masao Morita. A shared-memory parallel extension of KLIC and its garbage collection. In *Proceedings of FGCS '94 Workshop on Parallel Logic Programming*, pages 113–126, 1994.

[12] Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1030–1040, September 1993.

[13] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Wiley & Sones, 1996.

[14] Yoshikazu Kamoshida. HOCS: A C++ extension with parallel objects and multithreading, February 1997. (senior thesis), The University of Tokyo.

[15] James S. Miller and Barbara S. Epstein. Garbage collection in MultiScheme (preliminary version). In T. Ito and R. H. Halstead, Jr., editors, *Proceedings of US/Japan Workshop on Parallel Lisp*, volume 441 of *Lecture Notes in Computer Science*, pages 138–160, Sendai, Japan, June 1989. Springer-Verlag.

[16] James O'Toole and Scott Nettles. Concurrent replicating garbage collection. In *Proceedings of 1994 ACM Conference on LISP and Functional Programming*, pages 34–42, 1994.

[17] Joseph P. Skudlarek. Remarks on a methodology for implementing highly concurrent data objects. *ACM SIGPLAN Notices*, 29(12):87–93, 1994.

[18] Joseph P. Skudlarek. Notes on "a methodology for implementing highly concurrent data objects". *ACM Transactions on Programming Languages and Systems*, 17(1):45–46, 1995.

[19] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation -. In *number 18 in Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 275–292. the DIMACS work shop on Specification of Algorithms, 1994.

[20] Kenjiro Taura and Akinori Yonezawa. An effective garbage collection strategy for parallel programming languages on large scale distributed-memory machines. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 264–275, June 1997.

[21] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the 1992 SIGPLAN International Workshop on Memory Management*, pages 1–42, 1992.