A SCALABLE MARK-SWEEP GARBAGE COLLECTOR ON LARGE-SCALE SHARED-MEMORY MACHINES

大規模共有メモリ並列マシンにおけるスケーラブルなマークスイープ法ガーベージコレクタ

by

Toshio Endo

遠藤 敏夫

A Master Thesis

修士論文

Submitted to

the Graduate School of

the University of Tokyo

on February, 1998

in Partial Fulfillment of the Requirements

for the Degree of Master of Science
in Information Science

Thesis Supervisor: Akinori Yonezawa　米澤 明憲
Professor of Information Science

ABSTRACT

This thesis describes an implementation of a mark-sweep garbage collector (GC) for shared-memory machines and reports its performance results. 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 collection phase. Implementation is based on the Boehm-Demers-Weiser conservative GC library (Boehm GC). Experiments have been conducted on two systems. One is Ultra Enterprise 10000, a symmetric shared-memory machine with 64 Ultra SPARC processors, and the other is Origin 2000, a distributed shared memory machine with 16 R10000 processors. The application programs used for our experiments are BH (an N-body problem solver with Barnes-Hut algorithm), CKY (a context free grammar parser), Life (a life game simulator) and RNA (a program to predict RNA secondary structure).

On both systems, load balancing is a key to achieving scalability; a naive collector without load redistribution hardly exhibits speed-up. Performance can be improved by dynamic load balancing, which moves objects to be scanned across processors, but we still observe several performance limiting factors, some of which reveal only when the number of processors is large.

On 64 processor Enterprise 10000, the straightforward implementation achieves at most 12-fold speed-up. There are several reasons for this. The first one is that large objects became a source of load imbalance, because the unit of load redistribution was a single object. Performance is improved by splitting a large object into small pieces before pushing it onto the mark stack. Secondly, the marking speed drops as the number of processors increases, because of serializing method for termination detection using a shared counter. By employing non-serializing method using local flags, the idle time is eliminated. Thirdly, processors were sometimes blocked long to acquire locks on mark bits in BH application. The useless lock acquisitions are eliminated by using optimistic synchronization. With all these careful implementation, we achieved 14 to 28-fold speed-up on 64 processors.

Physical memory allocation policy has a significant effect on Origin 2000, a CC-NUMA architecture. With the default policy that allocates a physical page to the node that first touches that page, the performance does not improve on more than eight processors. By distributing memory regions in the round robin policy, we achieved 3.7 to 6.3-fold speed-up on 16 processors.

論文要旨

共有メモリ型並列マシン上でのマークスイープ法ガーベージコレクタ (GC) の実装と性能評価を報告する。実装した GC は並列 GC であり、共有ヒープ中のオブジェクトに対して全プロセッサが協調的に GC 処理を行なう。 GC 処理が開始する時、アプリケーションプロセスを停止させる。本 GC の実装を、 Boehm-Demers-Weiser conservative GC library(Boehm GC) を基にして行なった。実験を対称型共有メモリマシン Ultra Enterprise 10000 (Ultra SPARC プロセッサ × 64) 上と、分散共有メモリマシン Origin 2000 (R10000 プロセッサ × 16) 上で行なった。アプリケーションとして、 BH(Barnes-Hut N 体問題プログラム)、 CKY (文脈自由文法パーザ)、 Life(ライフゲームシミュレータ)、 RNA(RNA2 次構造予測プログラム) を用いた。

いずれのマシンであっても、スケーラビリティを実現するためには GC 処理の負荷分散を行なう必要がある。負荷分散を行なわない場合、並列化による速度向上はほとんど見られなかった。本 GC はマークフェイズ中に、これからスキャンすべきオブジェクトをプロセッサ間でやりとりすることにより動的負荷分散を行なう。この動的負荷分散により性能の向上が見られるが、いくつかの要素が、特にプロセッサ数が多い場合にスケーラビリティを低下させる。

Enterprise 10000 上における安直な実装の場合、台数効果は 12 倍程度で頭打ちになった。安直な実装の持つ問題の一つは、巨大なオブジェクトが負荷の不均衡をもたらすことである。この問題は、負荷の移動がオブジェクト単位で行なわれるために起こった。巨大なオブジェクトが探索中に見つかった時にそれらを小さく分割することにより、より細粒度な負荷分散が可能になり性能が向上した。また、 32 プロセッサを超えた時にマーク速度の低下が見られた。これはマーク処理の終了判定に用いていた共有カウンタが原因であった。代わりに各プロセッサ専用のフラグを用いることにより、ロック待ち時間がなくなり性能が向上した。また BH アプリケーションで、マークビットに対するロック待ち時間が長いため、性能が上がらない問題が見られた。マークビットに対してロックなしで read を行ない、それから compare&swap で変更を行なうことにより、無駄なロック獲得を減少させた。これらの点に留意して実装することにより、 Ultra Enterprise 10000 上で 64 プロセッサ用いたとき平均 14 〜 28 倍の台数効果を得た。

Origin 2000 上では、メモリのノード間の配置も性能に大きく影響する。メモリ領域を、それを初めにアクセスしたプロセッサと同じノードに割り当てるというデフォルトの方式の場合、 8 プロセッサ以上での速度向上が見られなかった。一方、アクセスするプロセッサに関係なくメモリ領域をノード間で均等に割り当てることにより、性能はプロセッサ数に伴い向上し、 16 プロセッサで 3.7 〜 6.3 倍の台数効果を得た。

Acknowledgements

First of all, I am deeply grateful to my supervisor, Professor Akinori Yonezawa for leading me to the research area of programming language design and implementation. This work would not be possible without his encouragement and advice. And I learned a lot from his optimistic and strong attitude toward the research.

I especially thank Dr. Kenjiro Taura, who has been the best advisor of my research since I became a member of Yonezawa Laboratory. He suggested the direction of my work and supported me with his idea and a broad range of background. He always brushed up my poor English patiently.

I thank Dr. Naoki Kobayashi for many useful advices from theoretical view. He always gave me critical comments my incomplete presentation.

I wish to express my gratitude to all members of Yonezawa laboratory and Kobayashi laboratory. Especially, Yoshihiro Oyama helped me not only in implementing the system and writing paper, but also in daily life. I would have missed some deadlines without his advices. Toshihiro Shimizu gave many advices during lunch and supper when I have trouble in my work. Discussion with Hirotaka Yamamoto was the great source of my inspiration. Norifumi Gotoh spent a lot of time for teaching me how to use software such as Latex2ϵ. I really thank all members here who make the good research environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Shared-memory architecture is attractive platform for implementation of general-purpose parallel programming languages that support irregular, pointer-based data structures [4, 20]. 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, 17], 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.

We implemented the collector by extending the Boehm-Demers-Weiser conservative garbage collection library (Boehm GC [2, 3]) on two systems: a 64-processor Ultra Enterprise 10000 and a 16-processor Origin 2000. 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). At present, we achieved 14–28-fold speed-up on Ultra Enterprise 10000, and 3.7 to 6.3-fold speed-up on Origin 2000.

The rest of the paper is organized as follows. Chapter 2 compares our approach with previous work. Chapter 3 briefly summarizes Boehm GC, on which our collector is based. Chapter 4 describes our parallel marking algorithm and solutions for performance limiting factors. Chapter 5 describes the experimental conditions. Chapter 6 shows experimental results, and we conclude in Chapter 7.

Chapter 2

Previous Work

Most previous published work on GCs for shared-memory machines has dealt with *concurrent GC* [6, 10, 17], 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 2.1). 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, 16] and we believe there are many unpublished work too, but there are relatively few published performance results. To our 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. Most of previous publications have reported only preliminary measurements.

Uzuhara constructed a parallel mark sweep collector on symmetric multiprocessors [22]. When the amount of free space in the shared heap becomes smaller than a threshold, some processors start a collection, while other processors continue application execution. The collector processors cooperatively mark all reachable objects with dynamic load balancing by using global task pool. Then they sweep the heap and

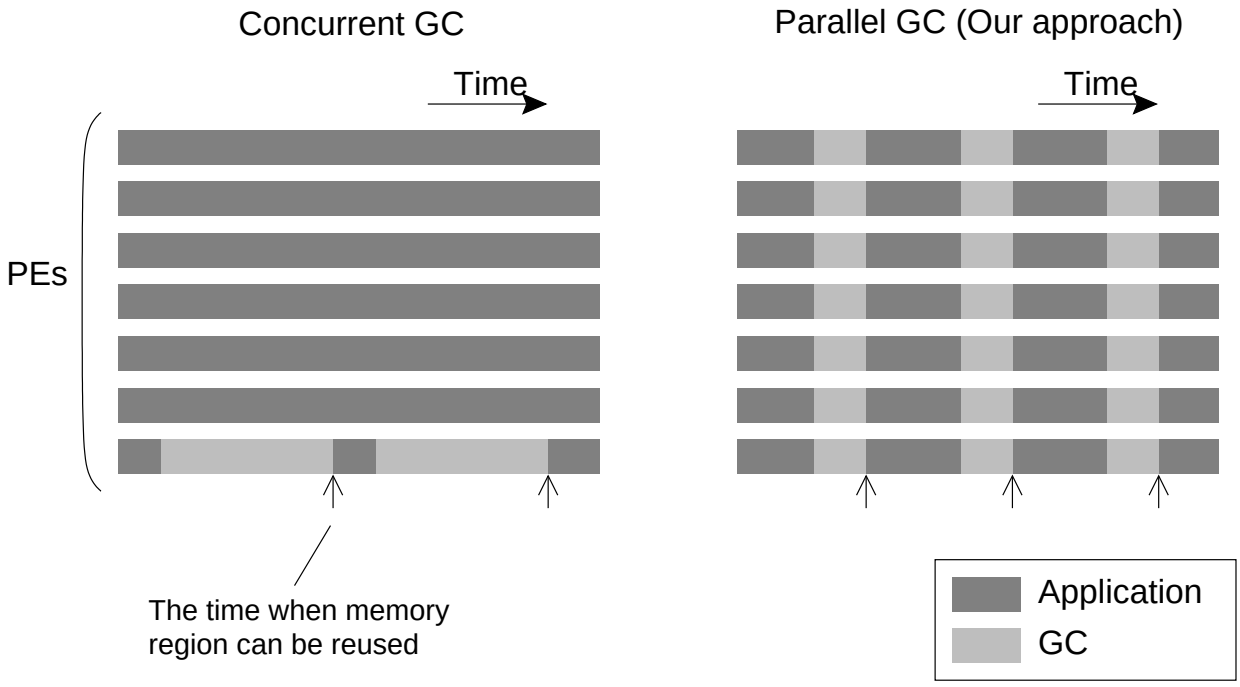

Figure 2.1: Difference between concurrent GC and our approach. If only one dedicated processor performs GC, a collection cycle becomes longer in proportion to the number of processors.

join the application workers. This approach has the same advantage as concurrent GC, and it can prevent a single collection cycle from becoming longer on large-scale machines.

Ichiyoshi and Morita proposed a parallel copying GC for a shared heap [11]. It assumes that the heap is divided into several local heaps and a single shared 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 works in the new shared-heap (i.e., to-space).

Our collector is much simpler than both of Uzuhara's collector and 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 Ichiyoshi and Morita 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 Chapter 6 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] Both of Uzuhara's method and Ichiyoshi and Morita's method may interact badly with such applications because it exhibits a very long marking cycle, during which the applications cannot utilize all the processors. Taura also reached a similar conclusion on distributed-memory machines [21].

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

[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.

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 Chapter 6, such factors become quite significant, especially in large-scale and multiprogrammed environments.

Chapter 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.

3.1 Sequential Mark-Sweep Algorithm

The mark-sweep collector's work is to find all garbage objects, which are unreachable from the *root set* (machine registers, stacks and global variables) via any pointer paths, and to free those objects. To tell whether an object is live (reachable) or garbage, each object has its *mark bit*, which shows 0(= 'unmarked') before a collection cycle. We mention how Boehm GC maintains the mark bits in section 3.2. A collection cycle consist of two phases; in the mark phase, the collector traverse objects that are reachable from root set recursively, and sets (marks) their mark bits at 1(= 'marked'). To mark objects recursively, Boehm GC uses a data structure called *mark stack* as shown in section 3.3. In the sweep phase, the collector scans all mark bits and frees

objects whose mark bits are still 'unmarked'. The sweeping method heavily depends on how the free objects are managed. We describe aspects relevant to the sweep phase in section 3.4.

3.2 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 and be word-aligned. 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); thus, a mark bitmap is 128-byte length. The jth bit of the ith byte in the mark bitmap describes the state of an object that begins at ($BlockAddr + i \times 32 + j \times 4$) where $BlockAddr$ is the start address of the corresponding heap block. 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.

3.3 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.

Figure 3.1 shows the marking process in pseudo code; each iteration pops an entry from the mark stack and scans the specified object,[1] possibly pushing new entries onto

[1] 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.

```
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 {
            mark bit of o[i] = 'marked'
            push(o[i], mark stack)
        }
    }
}
```

Figure 3.1: The marking process with a mark stack.

the mark stack. A mark phase finishes when the mark stack becomes empty.

3.4 Sweep

In the sweep phase, Boehm GC does not free each garbage object actually. Instead, it distinguish empty heap blocks from other heap blocks.

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 [2]. 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.

[2] The system does not free garbage objects on nonempty heap blocks, until the program requests objects of the proper size (lazy sweeping). In order to find a garbage objects from those heap blocks, mark bitmaps are preserved until next collection

Chapter 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

4.1.1 Parallel Marking

Each processor has its own local mark stack. When GC is invoked, all application processes are suspended by sending signals to them. When 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

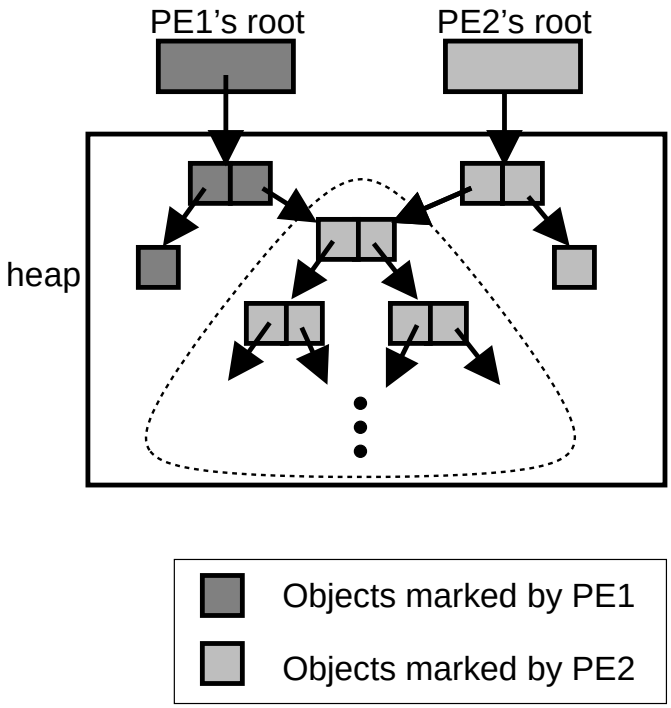

Figure 4.1: In the simple algorithm, all nodes of a shared tree are marked by one processor.

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 4.1). To improve marking performance, our collector performs dynamic load balancing by exchanging entries stored in mark stacks.

4.1.2 Dynamic Load Balancing of Marking

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 4.2). 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

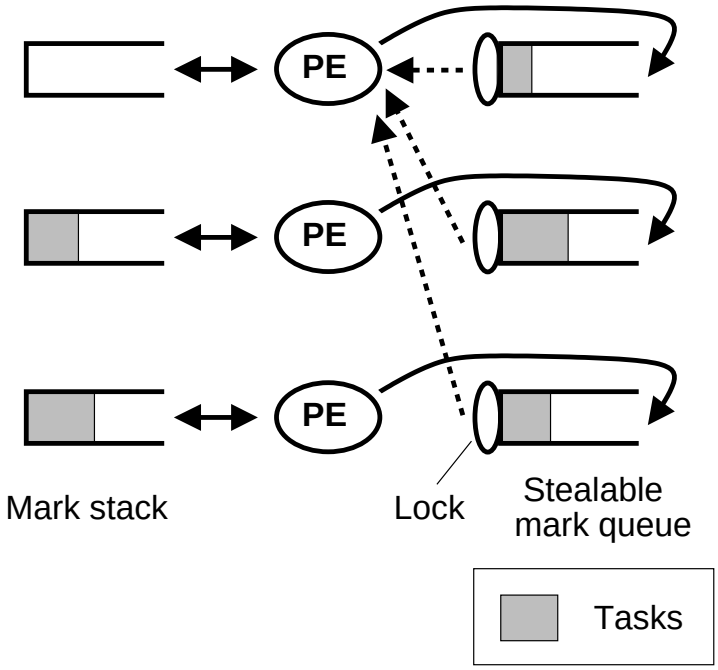

Figure 4.2: Dynamic load balancing method: tasks are exchanged through stealable mark queues.

(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[1] in the 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.

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

4.1.3 Parallel Sweeping

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 the heap and links empty heap blocks 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 in previous section exhibits acceptable speed-up on small-scale systems (e.g., approximately fourfold speed-up on eight processors). As we will see in Chapter 6, 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 800-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 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" operation with optimistic synchronization. We tests a mark bit first and quit if the bit is already set. Otherwise, we calculate the new bitmap for the word and write the new bitmap in the original location, if the location is the same as the originally read bitmap. This operation is done atomically by compare&swap instruction in SPARC architecture or load-link and store-conditional instructions in MIPS architecture. We retry if the location has been overwritten by another processor. These operations eliminate useless lock acquisitions on mark bits that are already set. We refer to this optimization as MOS (Marking with Optimistic Synchronization) in the experiments below.

Another advantage of this algorithm is that it is a non-blocking algorithm [8, 18, 19], 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.

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).

Chapter 5

Experimental Conditions

We have implemented the collector on two systems: the Ultra Enterprise 10000 and the Origin 2000. The former has a uniform memory access (UMA) architecture and the latter has a nonuniform memory access (NUMA) architecture. The implementation is based on the source code of Boehm GC version 4.10. We used four applications written in C++: BH (an N-body problem solver), CKY (a context free grammar parser), Life (a life game simulator) and RNA (a program to predict RNA secondary structure).

5.1 Ultra Enterprise 10000

Ultra Enterprise 10000 is a symmetric multiprocessor with sixty-four 250 MHz Ultra SPARC processors. All processors and memories are connected through a crossbar interconnect whose bandwidth is 10.7 GB/s. The L2 cache block size is 64 bytes.

5.2 Origin 2000

Origin 2000 is a distributed shared memory machine. The machine we used in the experiment has sixteen 195 MHz R10000 processors. That system consists of eight modules, each of which has two processors and the memory module. The modules are connected through a hypercube interconnect whose bandwidth is 2.5 GB/s. The memory bandwidth of each module is 0.78 GB/s and the L2 cache block size is 128 bytes.

In the default configuration, each memory page (whose size is 16 KB) is placed on

the same node as the processor that accessed the page first. Therefore processors can have desired pages on local by touching the pages at the initializing phase of the program. We used two physical memory allocation policies in the experiment:

Local to allocator (LA) Each heap block and corresponding mark bitmap are local to the processor that allocate the heap block first.

Round-robin (RR) The home node of a heap block is determined by its address rather than the allocator of the block. The heap block is local to processor P such that

$P = (\ Address\ /\ PAGESIZE\ ) mod\ \#Processors.$

The home of a mark bitmap is determined in the same rule.

5.3 Applications

We used following four applications written in C++. BH and CKY are parallel applications. We wrote Enterprise version of those applications in a parallel extension to C++ [14]. This extension allows programmer to create user level threads dynamically. The runtime implicitly uses fork system call at the beginning of the program. Since this extension does not work on Origin now, we wrote those applications on Origin by using fork system call explicitly. Life and RNA are sequential applications; even those sequential applications can utilize our parallel collection facility.

BH simulates the motion of N 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. In the experiment, we simulate 10000 particles for 50 time steps.

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. CKY calculates all nonterminal symbol for each substring of the input sentence in bottom-up. In the experiment, each of the given 256 sentences consists of 10 to 40 words.

Life solves "Conway's Game of Life". It simulates the cells on square board. Each cell have either of two states, ON and OFF. The state of a cell is determined by states of adjacent cells at the previous time step. The program takes a list that contains ON cells in an initial state. The number of initial ON cells is 5685 in our experiment. We simulate them for 150 time steps.

RNA predicts the secondary structure of an RNA sequence. The input data is a set of *stack regions* and each stack region has its position and energy. A set of stack regions is called *feasible* if any pair of its elements fulfills a certain condition. The problem is to find all feasible subsets of given stack regions whose total energy is no smaller than a threshold. The size of input stack regions is 119 in our experiment.

5.4 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 [1]. 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. It took 20 us to scan a 1000-word object on Enterprise 10000; that means 0.020 us per word. From this result, we let $a_3 = 0.020$. The other constants were determined similarly. The intention of this preliminary experiment is to measure the time for the workload without any cache misses.

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 $a_5 = 1.3$ on Enterprise 10000, and $a_1 = 0.42$, $a_2 = 0.13$, $a_3 = 0.028$, $a_4 = 2.0$, and $a_5 = 1.3$ on Origin 2000.

[1] The marking workload is derived from x_1, x_2, x_3 and the sweeping workload is from x_4, x_5.

Chapter 6

Experimental Results

6.1 Speed-up of GC

Figures 6.1-6.16 show performance of GC using the four applications on two systems. We measured several versions of collectors. "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 three 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. "Full" is the fully optimized version. We measured an additional version on Origin 2000, "Full-LA". This is the same as "Full" but takes different physical memory allocation policy. "Full-LA" takes "Local to Allocator" policy, while all other versions do "Round-robin" policy.

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 all 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 6.1–6.4 and 6.8–6.11 compare three versions, namely, Simple, Basic, and Full. The graphs show that Simple does not exhibit any recognizable speed-up in any application. As Figure 6.1–6.4 show, Basic on Enterprise 10000 performs reasonably until a certain point, but it does not scale any more beyond it. The exception is RNA, where we do not see the difference between Basic and Full. The saturation point of Basic depends on the application; Basic of CKY reaches the peak on 32 processors, while that of BH reaches the saturation point on 8 processors. The peak of Life is 48 processors. Full achieved about 28-fold speed-up in BH and in CKY, and about 14-fold speed-up in Life and RNA on 64 processors.

On 16-processor Origin 2000, the difference between Basic and Full is little, except in BH. The some performance problems in Basic, however, appear only when the number of processors becomes large as we have observed on Enterprise 10000; thus, Full would be more significant on larger system. Full achieved 3.7–6.3-fold speed-up on 16 processors.

6.2 Effect of Each Optimization

Figure 6.5–6.7 show how each optimization affects scalability on Enterprise 10000. Especially in BH and in CKY, removing any particular optimization yields a sizable degradation in performance when we have a large number of processors. Without the improved termination detection by the non-serializing barrier (NSB), neither BH nor CKY achieves more than a 17-fold speed-up. Without NSB, Life does not scale on more than 48 processors, too. Sensitivity to optimizations differs among the applications; Splitting large objects (SLO) and marking with optimistic synchronization (MOS) have significant impacts in BH, while they do not in other applications.

SLO is important when we have a large object in the application. In BH, we use a single array named `particles` to hold all particles data, whose size is 800 KB in our experiments. This large array became a bottleneck when we omitted SLO optimization. This phenomenon was noted on Origin 2000, as Figure 6.12 indicates.

Generally, MOS have significant effects when we have objects with big reference counts, because these objects cause many contentions between collectors that try to

visit them. The experiment revealed that the array `particles` was the source of problem again; in one collection cycle, we observed that we had about 70,000 pointers to this array. That caused significant contentions. This big reference count was produced by the stack of user threads. Because our BH implementation computes forces to the particles in parallel, each thread has references to its responsible particles. Although those references are directed to distinct addresses (for example, `i`th thread has references to `particles[i]`), all of them are regarded as pointers to a single object `particles`. MOS optimization effectively alleviate the contentions in such case and improve the performance.

We observe significant impact of NSB optimization on GC speed in three applications, but we do not see that in RNA even on 64 processors. Although the reason for this difference is not understood well, in general, NSB is important when collectors tend to become idle frequently. This is because collectors often update the idle counters, when we do not implement NSB. In RNA, the frequency of the task shortage may be low. We will investigate whether this hypothesis is the case in the future.

6.3 Effect of Physical Memory Allocation Policy

Figure 6.13–6.16 compare two memory placement policies: 'Local to allocator (LA)' and 'Round-robin (RR)' on Origin 2000, described in Section 5.2. Full adopts RR policy and Full-LA LA. As we can see easily, collection speed with RR is significantly faster than that with LA in three applications, BH, CKY and RNA. When we adopt LA policy, GC speed does not improve on more than eight processors.

While we have not fully analyzed this, we conjecture that this is mainly due to the imbalance in the amount of allocated physical pages among nodes. With LA policy, the access to objects and mark bitmaps in the mark phase contend at nodes that have allocated many pages. Actually, BH has significant memory imbalance because only one processor construct a tree of particles. And all objects in RNA are naturally allocated by one processor because our RNA is sequential program [1].

[1] We will investigate why this is not the case in Life, which is also sequential program.

6.4 Discussion on Optimized Performance

As we have seen in Section 6.1, the GC speed of fully optimized version always get faster as the number of processors increases in any applications. But they considerably differ in GC speed; for instance, it is 28-fold speed-up in BH and CKY, while 14-fold in Life and RNA on Enterprise 10000. In order to try to find the cause of this difference, we examined how processors spend time during the mark phase [2]. Figure 6.17–6.20 show the breakdowns. From these figures, we can say that the biggest problem in Life is load imbalance, because processors spend a significant amount of time in idle. The performance improvement may be possible by refining the load balancing method. On the other hand, we currently can not specify the reasons of the relatively bad performance in RNA, where processors are busy during 90% of the mark phase.

[2] In most collection cycles, the sweep phase is five to ten times shorter than mark phase. We therefore focus on the mark phase.

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 object) are done.
No-MOS	All optimizations but MOS (marking with optimistic synchronization) are done.
No-NSB	All optimizations but NSB (non-serializing barrier) are done.
No-SLO	All optimizations but SLO (splitting large object) are done.
Full	All optimizations are done.
Full-LA	Origin 2000 only. Same as Full, but the physical memory allocation policy is "Local to Allocator".

Table 6.1: Description of labels in following graphs. Except Full-LA, the physical memory allocation policy on Origin 2000 is "Round-robin".

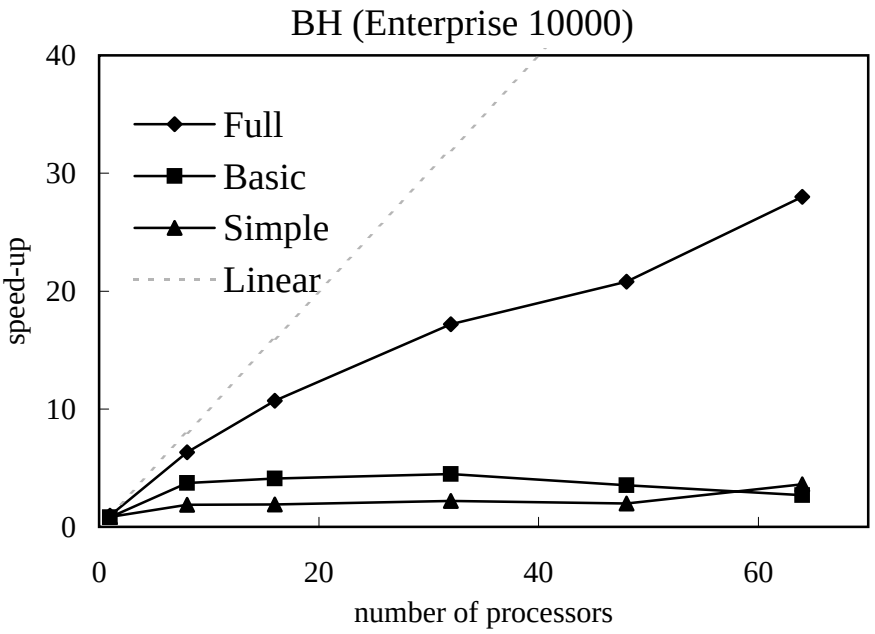

Figure 6.1: Average GC speed-up in BH on Enterprise 10000.

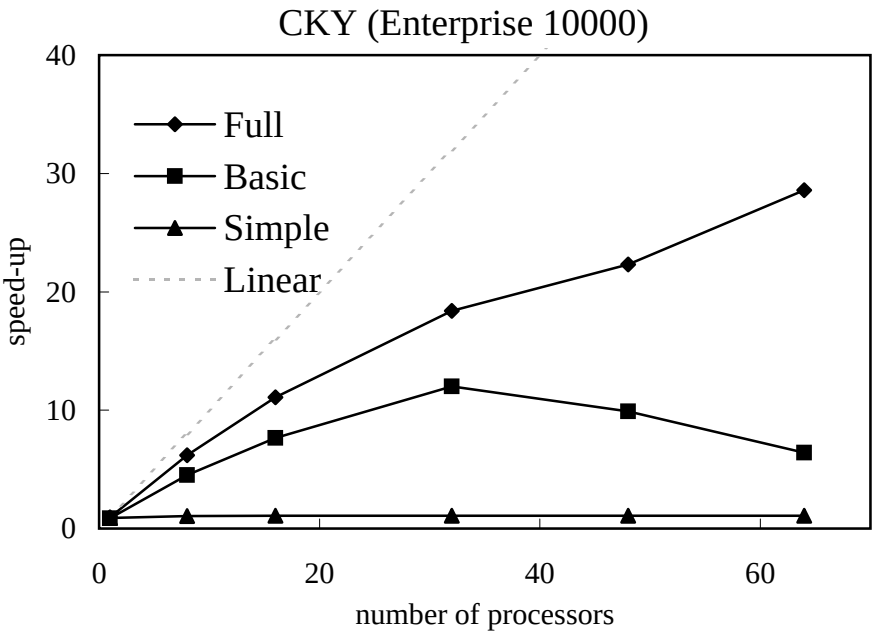

Figure 6.2: Average GC speed-up in CKY on Enterprise 10000.

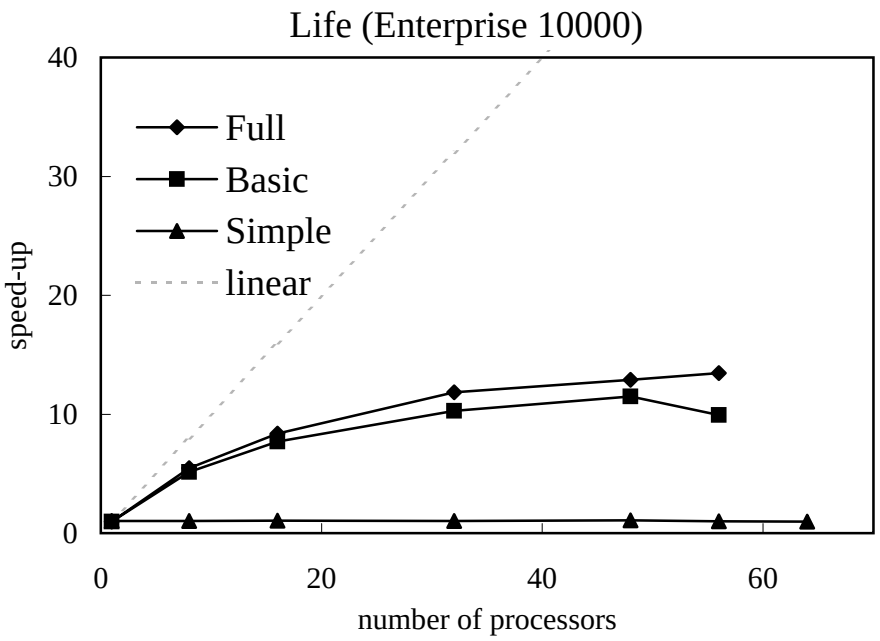

Figure 6.3: Average GC speed-up in Life on Enterprise 10000.

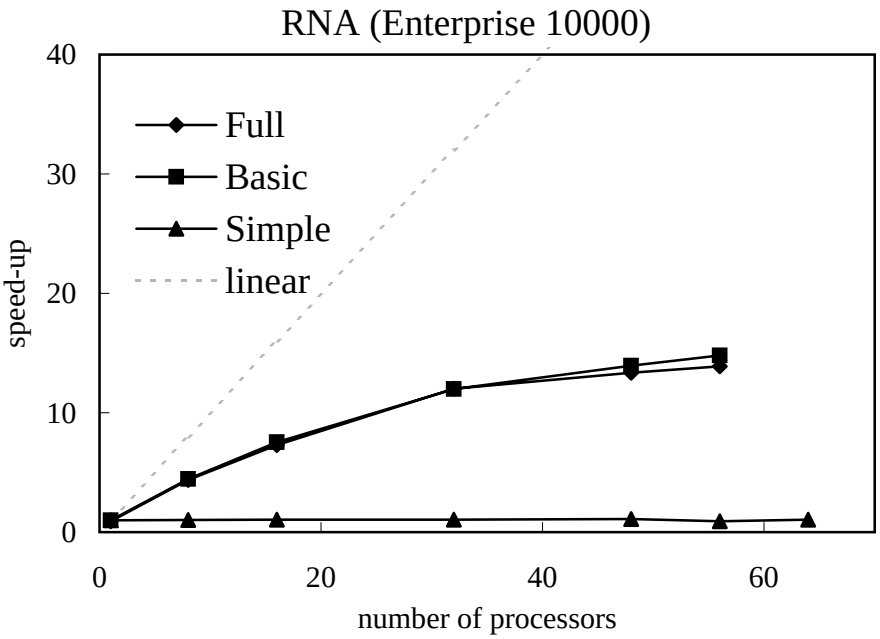

Figure 6.4: Average GC speed-up in RNA on Enterprise 10000.

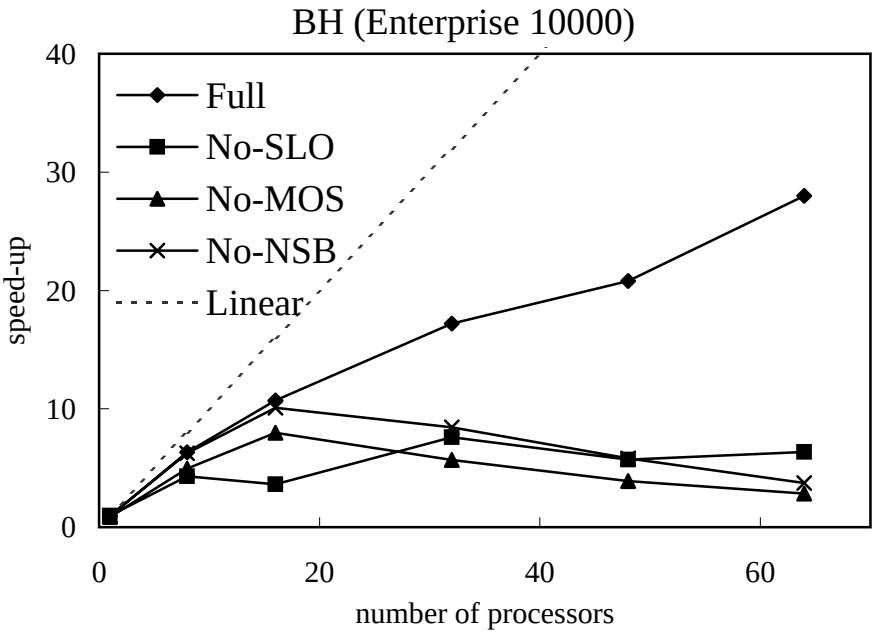

Figure 6.5: Effect of each optimization in BH on Enterprise 10000.

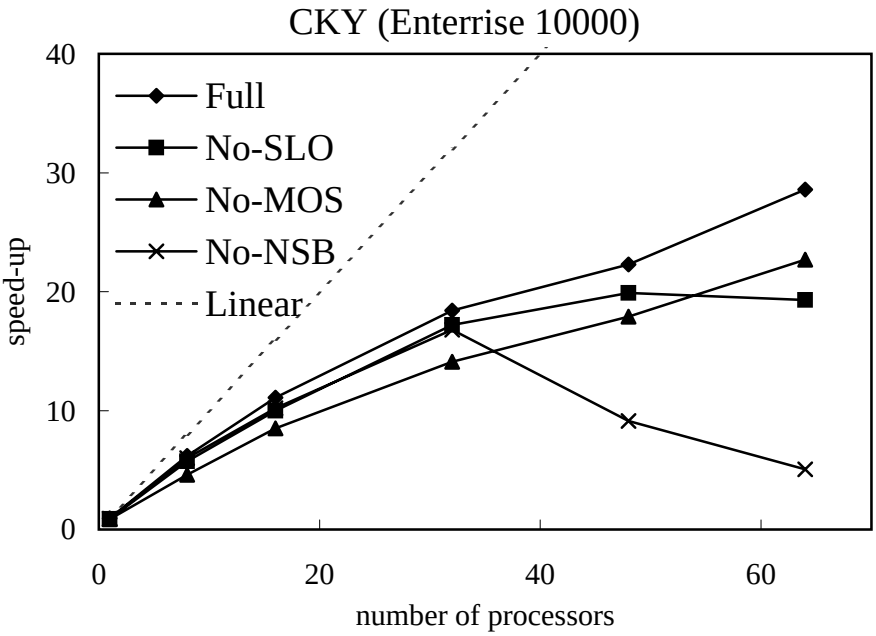

Figure 6.6: Effect of each optimization in CKY on Enterprise 10000.

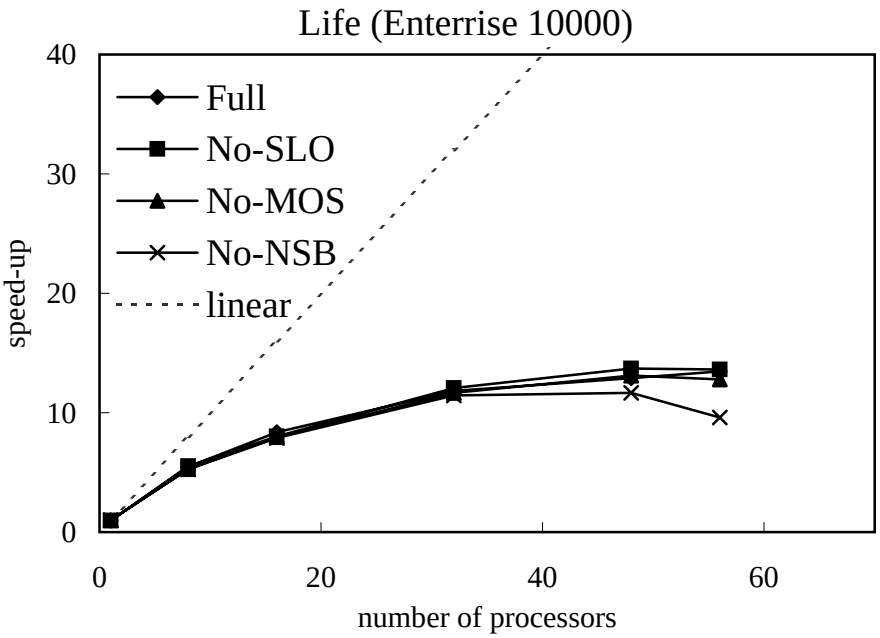

Figure 6.7: Effect of each optimization in Life on Enterprise 10000.

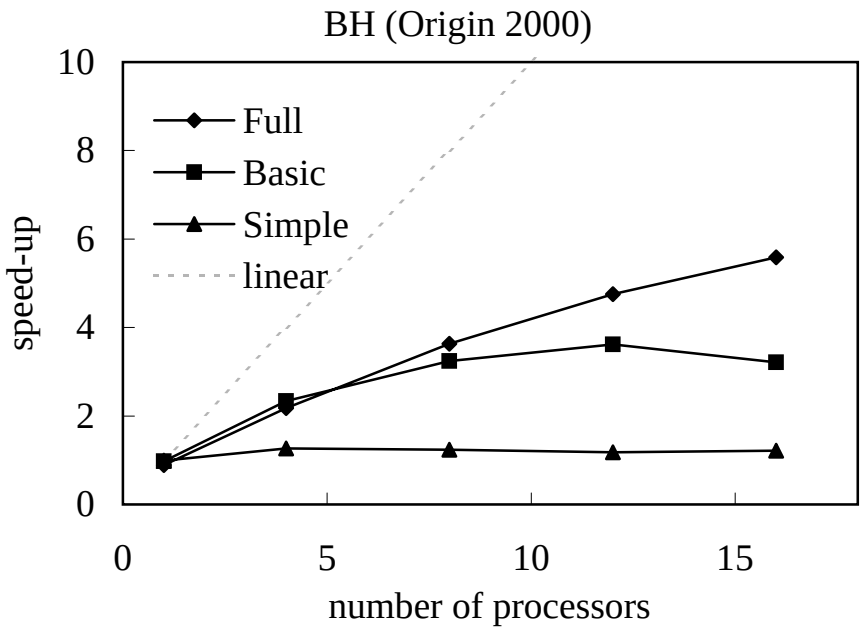

Figure 6.8: Average GC speed-up in BH on Origin 2000.

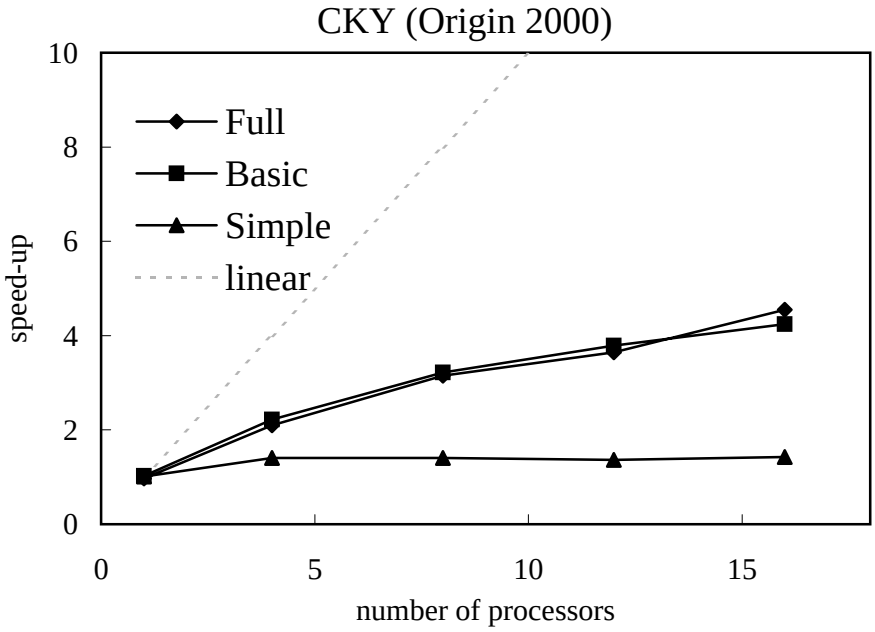

Figure 6.9: Average GC speed-up in CKY on Origin 2000.

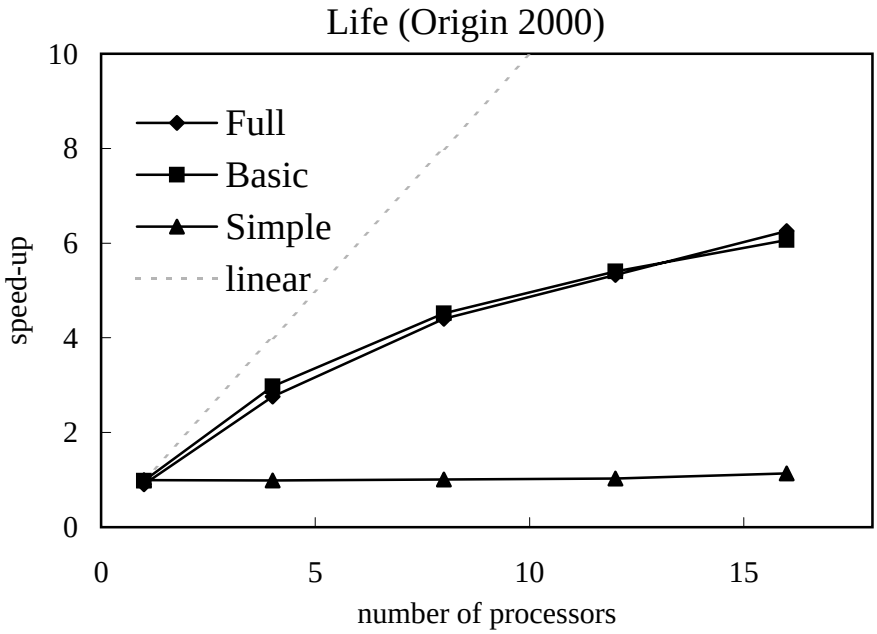

Figure 6.10: Average GC speed-up in Life on Origin 2000.

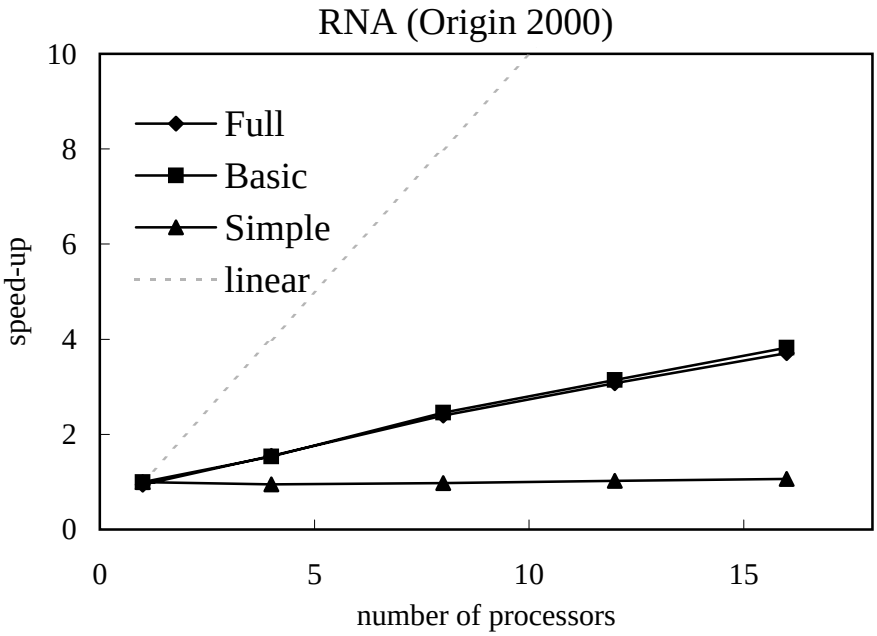

Figure 6.11: Average GC speed-up in RNA on Origin 2000.

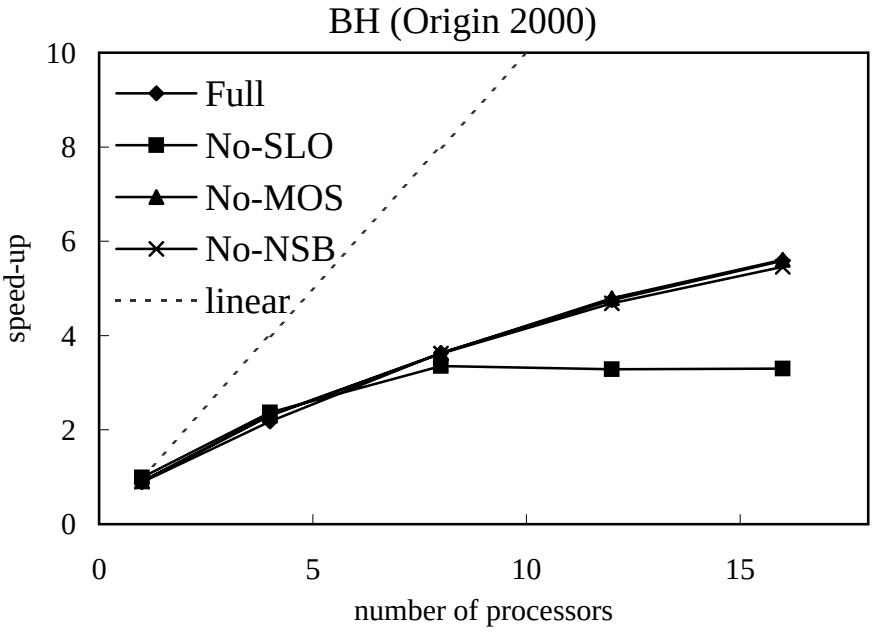

Figure 6.12: Effect of each optimization in BH on Origin 2000.

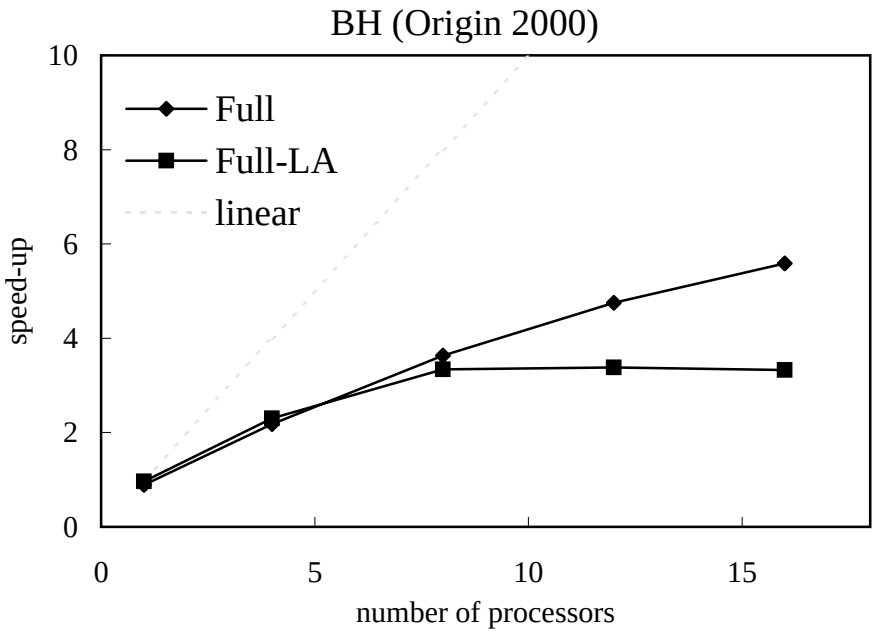

Figure 6.13: Effect of physical memory allocation policy in BH on Origin 2000.

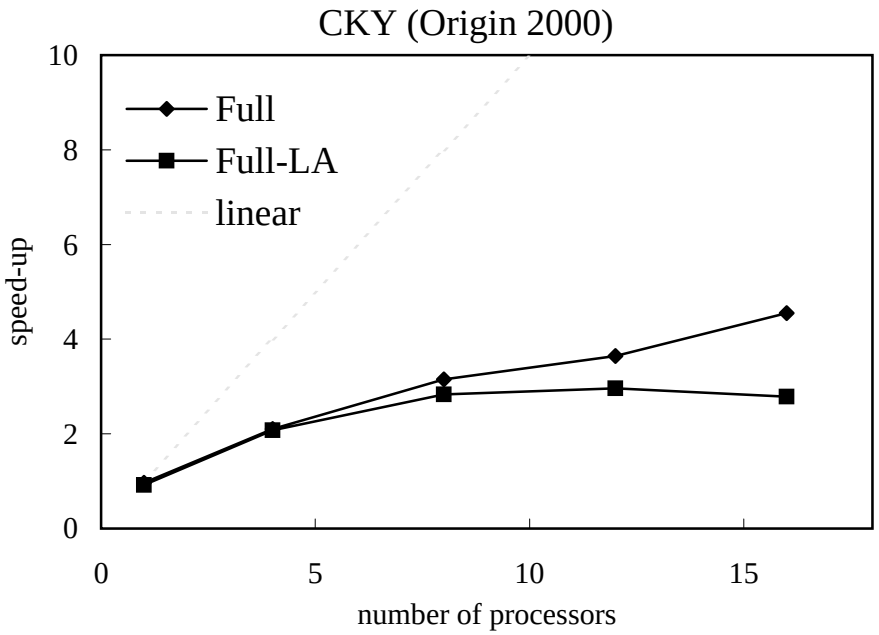

Figure 6.14: Effect of physical memory allocation policy in CKY on Origin 2000.

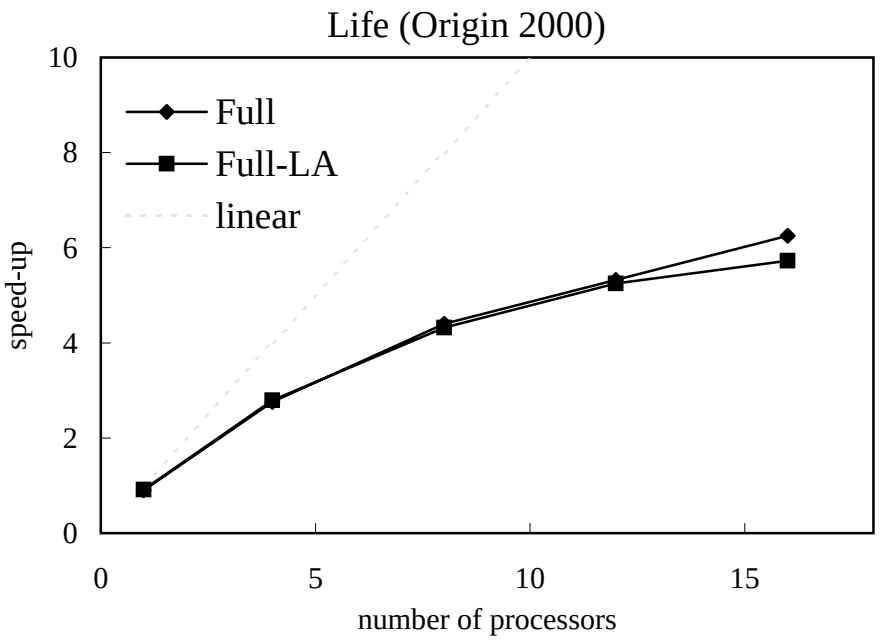

Figure 6.15: Effect of physical memory allocation policy in Life on Origin 2000.

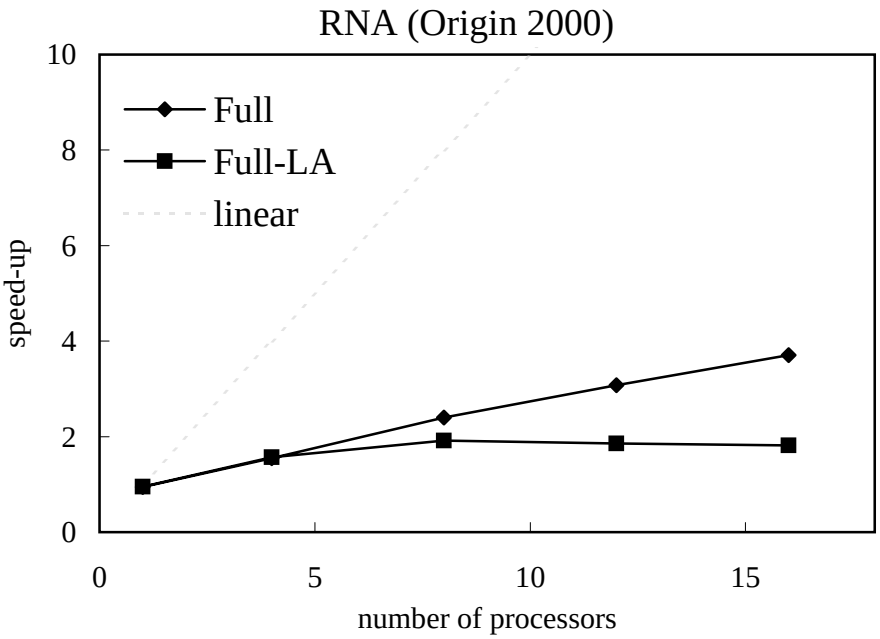

Figure 6.16: Effect of physical memory allocation policy in RNA on Origin 2000.

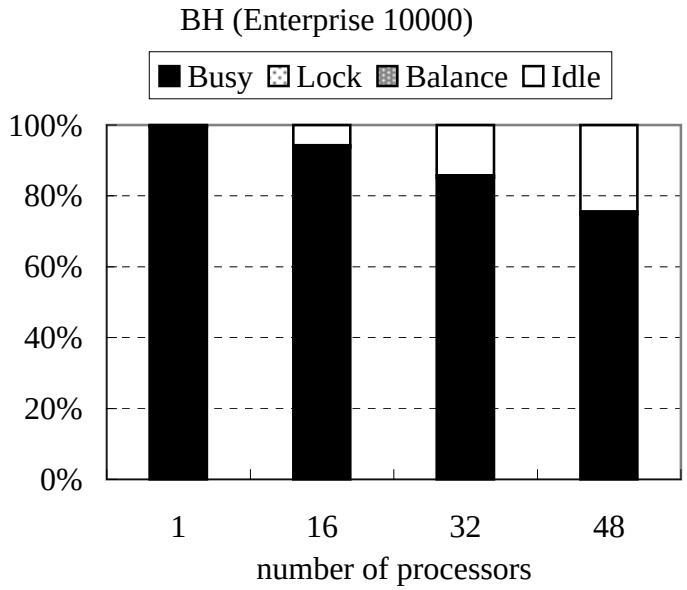

Figure 6.17: Breakdown of the mark phase in BH on Enterprise 10000. This shows busy, waiting for lock, moving tasks, and idle.

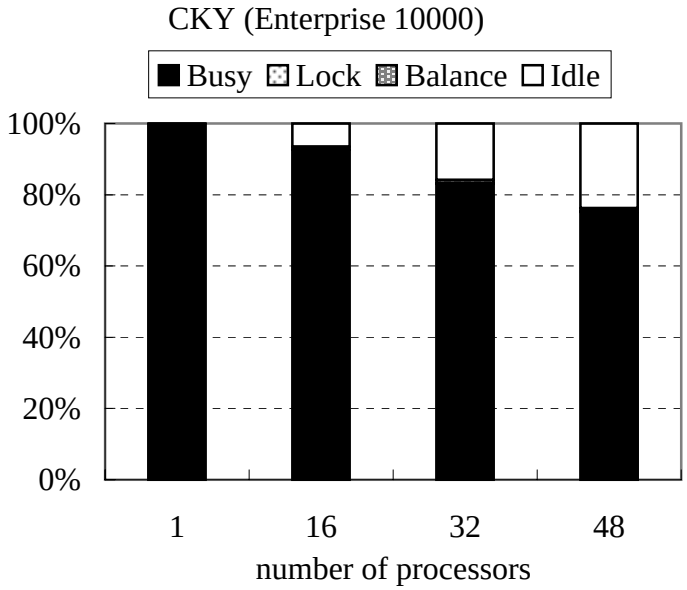

Figure 6.18: Breakdown of the mark phase in CKY on Enterprise 10000.

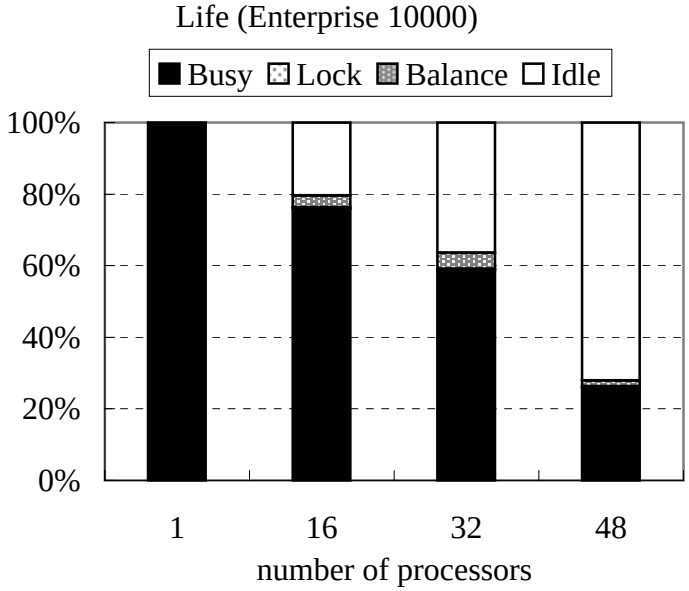

Figure 6.19: Breakdown of the mark phase in Life on Enterprise 10000.

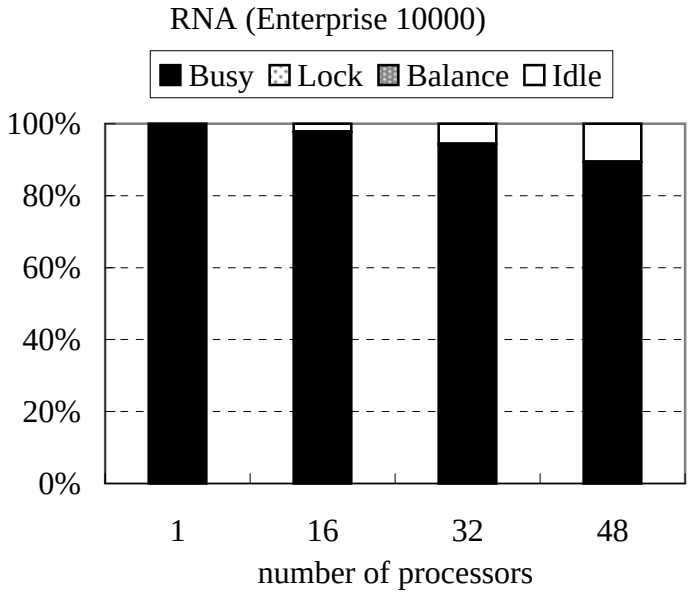

Figure 6.20: Breakdown of the mark phase in RNA on Enterprise 10000.

Chapter 7

Conclusion

We constructed a highly scalable parallel mark-sweep garbage collector for shared-memory machines. Implementation and evaluation are done on two systems: Ultra Enterprise 10000, a symmetric shared-memory machine with 64 processors and Origin 2000, a distributed shared-memory machine with 16 processors. This collector performs dynamic load balancing by exchanging objects in mark stacks.

Through the experiments on the large-scale machine, we found a number of factors that severely limit the scalability, and 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) 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 optimistic synchronization instead of a "lock-and-test" operation. (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 non-serializing method using local flags without locking, and the long critical path was eliminated.

On Origin 2000, we must pay attention to physical page placement. With the default policy that places a physical page to the node that first touches it, the GC speed was not scalable. We improved performance by distributing physical pages in a round robin fashion. We conjecture that this is because the default policy causes imbalance

of access traffic between nodes; since some nodes have much more physical pages allocated than other nodes, accesses to these highly-loaded nodes tend to contend, hence the latency of such remote accesses accordingly increases. For now, we do not have enough tools to conclude.

When using all these solutions, we achieved 14 to 28-fold speed-up on 64-processor Enterprise 10000, and 3.7 to 6.3-fold speed-up on 16-processor Origin 2000.

Chapter 8

Future Work

We would like to improve the GC performance further. In Section 6.4, we have seen that the collectors in some applications still spend a significant amout of time in idle. We will investigate how we can improve the load balancing method. Instead of using buffers for communication (the stealable mark queues), stealing tasks from victim's mark stack directly may enable faster load distributing.

We have also noticed that we cannot explain the relatively bad performance of RNA by the load imbalance alone. This may be due to the number of cache misses, which are included by 'Busy' in Figure 6.20. We can capture the number of cache misses by using performance counters, which recent processors are equipped with. We can use the R10000's counters through /proc file system on Origin 2000. And we have constructed a simple tool to use the Ultra SPARC's counters on Enterprise 10000. With these tools, we are planning to examine how often processors meets cache misses.

In Section 6.3, we mentioned that we can obtain better performance with the RR (Round-robin) physical memory allocation policy than with the LA (Local to allocator) policy. So far, the focus of our discussion is the speed of GC alone. But the matter will be complicated when we take account into the locality of application programs. LA policy may be advantageous when a memory region tends to be accessed by the allocator. The ideal situation would be that most accesses by application are local, while collection task is balanced well.

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 Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, 1997.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] Shigeru Uzuhara. Parallel garbage collection on shared-memory multiprocessors. In *IPSJ SIG Notes 90-ARC-83 (Proceedings of Summper Workshop on Parallel Processing)*, pages 205–210, July 1990. (in Japanese).

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