Scalable Dynamic Memory Management Module on Shared Memory Multiprocessors

共有メモリ並列計算機上のスケーラブルな動的メモリ管理モジュール

by

Toshio Endo

遠藤 敏夫

A Dissertation

博士論文

Submitted to
the Graduate School of
the University of Tokyo
on June, 2001
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science

Thesis Supervisor: Akinori Yonezawa　米澤 明憲
Professor of Information Science

ABSTRACT

This thesis describes implementation of a memory management module on shared memory multiprocessors. The focus is to achieve scalability; for this purpose, both garbage collector (GC) and memory allocator are parallelized. Scalability of memory management module is important, especially for programs that aggressively allocate memory objects. This thesis presents optimization techniques that make memory management scalable, and evaluates their effects through experimentation. Performance evaluation is done on Sun Ultra Enterprise 10000 (symmetric multiprocessor) and Origin 2000 (distributed shared memory machine). To obtain scalability, eliminating mutual exclusion alone is insufficient; module should be aware of memory architecture of each machine. The goal is to obtain optimal performance on each machine, by using techniques that are suitable for each architecture. This thesis evaluates the effects of optimizations through experiments on real machines. Moreover, it evaluates them on a performance prediction model of parallel GC that this thesis presents. As for memory allocator, this thesis discuss proposed optimizations of allocator from several viewpoints: the speed of allocator itself, memory utilization, and effects on performance of user programs.

Memory allocator should consider both time efficiency and space efficiency, however, there is a trade-off between them. If only allocation speed is the issue, a simple allocation strategy that divides the heap and gives a sub-heap for each processor would work. This strategy requires no mutual exclusion in allocation, while it makes space efficiency worse. This thesis discusses the strategies that fulfill both parallelism of allocation and memory utilization. Not only the performance of allocator itself, allocation strategy has impact on the performance of user programs. For example, an allocation strategy that achieves good space efficiency may degrade locality and increase false sharing of user programs and degrade their performance. It is important to take performance of user programs into account when the trade-off is discussed. Memory allocator that achieves good space and time efficiency is implemented and its performance is evaluated through experimentation.

Implemented GC is a parallel mark sweep GC, on which all threads cooperatively performs GC task. One of optimizations is the fair placement of mark bit among memory nodes. On DSM, memory access congestion from several processors to a memory node heavily raises access cost. This optimization is proposed in order to alleviate access contention cost, and it is confirmed to be effective through experiments on DSM machines. However, even if an optimization is effective on some machines, it may be ineffective on other machines, which have different memory latency and occupancy time. To discuss performance portability, it is required to understand quantitatively the relation between the performance of parallel GC and memory architecture. This thesis presents a performance prediction model of parallel GC. This model takes the heap snapshot and architecture parameters as input, and shows predicted GC time. the model obtains the amount of live objects, critical path length of object graph, and the predicted number of cache misses. On the other hand, architecture parameters that the model utilizes include not only memory latency, memory node occupancy time of access requests to take access contention into account. This thesis evaluates the validity of the model by comparing the predicted performance and the real performance obtained through experiments on parallel machines. It turned out that without taking contention costs into account, the model can never give a reasonable prediction. By using the presented model, this thesis estimates effects of optimizations on various architectures. Moreover, it will enable automatic adaption of parallelism and task steal strategy that use results of the model.

論文要旨

本論文は共有メモリ並列計算機のためのメモリ管理モジュールの実装について述べる。このモジュールはメモリ確保ルーチンとガーベージコレクタ (GC) のそれぞれを並列化したものであり、スケーラビリティに焦点をおく。メモリ管理モジュールのスケーラビリティは、特にメモリ確保を頻繁に行なうプログラムにおいて重要である。本論文はメモリ管理モジュールをスケーラブルにする技法を提案し、実験により効果を評価する。実験には対象型共有メモリ並列計算機 (SMP) Sun Ultra Enterprise 10000 と、分散共有メモリ並列計算機 (DSM) SGI Origin 2000 を用いる。スケーラビリティを達成するためには、排他制御によるボトルネックを削減するだけでなく、計算機のメモリアーキテクチャの差異を考慮することが必要不可欠である。アーキテクチャに適応した技法を採用することにより、それぞれのマシン上で最適な性能を得ることを目的とする。 GC の最適化技法の効果を実験により評価するだけでなく、並列 GC の性能予測モデルを提案し、そのモデルを通した議論も行なう。メモリ確保ルーチンの最適化技法の効果について、確保ルーチン自身の速度、メモリ利用効率、ユーザプログラム性能への影響の観点から議論する。

メモリ確保ルーチンの効率には時間的効率と空間的効率があり、これらはトレードオフになりやすい。メモリ確保の高速化のみを追求するのであれば、プロセッサ毎にヒープを分離させるのが良い。この場合確保時の排他制御は必要なく、完全に並列にメモリ確保することができる一方、メモリ使用量が増大してしまう。メモリ確保時の並列性と、メモリ利用効率の両方を満たす方式を提案する。また、メモリ確保ルーチンの設計はそれ自身だけでなくユーザプログラムの性能にも大きく影響する。例えば空間的効率を追求した結果としてユーザプログラムの局所性が低下したり、 false sharing が増大してしまう場合が考えられる。このため、ユーザプログラムの性能も考慮に入れ、トレードオフについて議論する。両方の効率が良好なメモリ確保ルーチンを実装し、実験により性能評価を行なう。

実装した GC は並列マークスイープ GC であり、全スレッドが協調して処理を行なう。この並列 GC の最適化の一つとして、 DSM 上のマークビットのメモリノード間平均化を提案した。 DSM においては多数のプロセッサから特定メモリノードへアクセスが集中するとアクセスコストが非常に高くなる。アクセス集中を低減するためにこの最適化を提案し、実計算機上の実験により効果があることを確認した。しかし、提案した最適化が特定の計算機で効果があったとしても、レイテンシやメモリ占有時間の異なる別の計算機で効果があることは保証されない。この問題を解決するには、並列 GC の性能とメモリアーキテクチャの関係を定量的に理解することが必要である。このために、GC 性能の予測モデルを構築した。このモデルはヒープ状態とアーキテクチャパラメータを入力とし、予測 GC 時間を出力する。ヒープ状態から、生きたオブジェクトの総量、オブジェクトグラフの並列度、キャッシュミス数を得る。一方、アーキテクチャの性質を捉えるために、メモリアクセスのレイテンシだけでなくアクセス要求の衝突も考慮する。本モデルの正当性を、予測結果と並列計算機上の実験による実測結果を比較することにより示す。衝突コストを入れないモデルによる結果との比較により、正確な予測を行なうには衝突コストを考慮することが必須であることが分かった。この手法の応用には、 GC の並列度の調整や仕事移動アルゴリズムの変更などの適応的アルゴリズムが考えられる。

Contents

List of Tables **4**

List of Figures **6**

Acknowledgments **8**

1 Introduction **9**

1.1 Evaluation Settings 12

1.2 Organization of Thesis 14

2 Scalable Parallel Allocator **18**

2.1 Introduction 18

2.2 Algorithm 20

2.2.1 BIBOP Allocator 22

2.2.2 Assumption 23

2.2.3 Naive Parallel Algorithms and the Problem 23

2.2.4 LPS Algorithm 26

2.3 Analysis of Memory Consumption 27

2.4 Performance Evaluation 29

2.4.1 Scalability of Allocation 29

2.4.2 Memory Consumption and Locality 31

2.4.3 Performance of Application Programs 31

2.5 Related Work 35

2.6 Summary 39

3 Scalable Parallel Garbage Collector **40**

3.1 Introduction 41

3.2 Previous Work 42

3.3 Overview of Parallel GC 45
3.4 Parallel Marking Algorithm 46
3.4.1 Data Structure 46
3.4.2 Dynamic Load Balancing 47
3.4.3 Performance Limiting Factors and Solutions 51
3.5 Experimental Results 54
3.5.1 Experimental Conditions 54
3.5.2 Effect on Application Running Time 55
3.5.3 Speed-up of GC 56
3.5.4 Effect of Each Optimization 59
3.6 Summary 64

4 Predicting Scalability of GC **66**
4.1 Introduction 66
4.2 Parallel Mark-Sweep Garbage Collector 67
4.3 Prediction Method 68
4.3.1 Overview 68
4.3.2 Assumption 68
4.3.3 Architecture Parameters 69
4.3.4 Heap Inspection 69
4.3.5 Performance Prediction without Miss Cost 71
4.3.6 Number of Cache Misses 71
4.3.7 Cost of Cache Misses 72
4.4 Experimental Results 72
4.4.1 Evaluation of Predicted Performance 74
4.4.2 Overhead of Predictor 75
4.4.3 Performance Prediction on Future Machines 75
4.4.4 Adaptive GC algorithm 75
4.5 Related Work 77
4.6 Summary 78

5 Concurrent Parallel Garbage Collector **81**
5.1 Introduction 82
5.2 Algorithm 83
5.2.1 Optimization 85
5.2.2 Software Write Barrier 86

5.3 Performance Evaluation 86
5.3.1 Total Execution Time 87
5.3.2 Pause Time 87
5.4 Discussion 91
5.5 Related Work 91
5.6 Summary 93

6 Conclusion **95**

Bibliography **98**

A Application Interface of Scalable Memory Management Module **105**
A.1 Supported Environments 106
A.2 Description of API 106

List of Tables

2.1 Memory allocation throughput. The number of malloc ($\times 1,000,000$) per second by all threads is shown. 34

2.2 The amount of memory objects allocated, execution time without GC time, and allocation speed of applications. This table shows statistics on Origin 2000 with 64 threads. LPS ($k = 1$) allocator is used. Except for execution time, the total numbers among all threads are shown. 34

3.1 The number of GC invocation, execution time, the maximum amount of living objects, and the heap size of applications. This table shows statistics on Enterprise 10000 with 60 threads. Fully optimized collector is used. 65

3.2 Description of labels in following graphs. 65

4.1 Memory access cost on O2K and E10K, obtained from benchmark tests. "RW" stands for atomic read-modify-write access. 70

4.2 The difference between predicted performance and real performance with 48 processors. 76

4.3 Overhead and accuracy of two predictors, in BH-st/O2K. Overhead is the ratio of prediction time to running time of mark phase. 76

4.4 The result of automatic regulation of GC parallelism, in BH-st/O2K. 80

5.1 Average pause time and worst pause time on Enterprise 10000, in milliseconds. 'Conc-VM' shows the pause time of finalize phase. 89

5.2 Average pause time and worst pause time on Origin 2000, in milliseconds. 'Conc-VM' shows the pause time of finalize phase. 90

List of Figures

1.1 Data structure of Matmul benchmark. 15
1.2 Data structure of BH benchmark. 15
1.3 Data structure of CKY benchmark. 15
1.4 Data structure of Cube benchmark. 17

2.1 Graphs show transition of memory usage by parallel application with two threads. For program (A), both PS algorithm and AL algorithm consume same amount of memory region. For program (B), AL method may consume much more region than PS algorithm. 21
2.2 Heap structure of BIBOP allocator. The heap consists of fixed-sized pages. Each page includes objects of the same size. To keep track of free regions, allocator uses free objects lists and free page list. 24
2.3 Some design policies of parallel BIBOP allocators. Figures show All-shared(AS) algorithm, Page-shared(PS) algorithm, All-local(AL) algorithm, Locality-aware-page-shared(LPS, ours) algorithm. The arrows show the flow of memory allocation requests. 30
2.4 Memory allocation throughput. The horizontal axis shows the number of threads, and the vertical axis shows the number of malloc ($\times 1,000,000$) per second by all threads. 30
2.5 Memory consumption and remote page allocation ratio on unfair benchmark. 32
2.6 Performance of matmul application on Origin 2000. 36
2.7 Performance of BH-pt application on Origin 2000. 37
2.8 Performance of CKY application on Origin 2000. 38

3.1 Difference between concurrent GC and our approach. If only one dedicated thread performs GC, a collection cycle becomes longer in proportion to the number of threads. 43
3.2 The marking process with mark bitmap and mark stack. Marking is done atomically. The code relevant to dynamic load balancing is omitted. 48
3.3 In the simple algorithm, all nodes of a shared tree are marked by one thread. 50
3.4 Dynamic load balancing method. An idle thread becomes a thief and try to the bottom of other mark stacks. 52
3.5 Application running time on Enterprise 10000. 57
3.6 Application running time on Origin 2000. 58
3.7 Average GC speed-up on Enterprise 10000. 60
3.8 Average GC speed-up on Origin 2000. 61
3.9 Effect of each optimization on Enterprise 10000. 62
3.10 Effect of each optimization on Origin 2000. 63

4.1 Overview of our performance prediction method. T_P^M is the final result; the marking time with access costs and contention costs. 70
4.2 The behavior of cache lines, in serial execution and parallel execution. 73
4.3 Behavior of a processor and a main memory node. Cache miss cost is at least $M_1 = 2S_L + S_O$, and gets longer if contention occurs. 73
4.4 GC Speed-up. Graphs compare real speed-up("Real") and predicted speed-up("Pred"). 76
4.5 Predicted marking performance where the number of processors is larger than real machine. The graphs assume that the number of main memory nodes and memory performance are the same as real machine. 77

5.1 A GC cycle consists of start phase, concurrent phase, and finalize phase. 85
5.2 Total execution time of each application. The results are normalized to that of 'Stop'. 88

Acknowledgments

First of all, I am deeply grateful to my supervisor, Professor Akinori Yonezawa for leading me to the interesting research area: programming language implementation and parallel programming. He has supported my research life in many ways. And I learned a lot from his optimistic and strong attitude toward the research.

I especially thank Lecturer 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. I learned all things about research from him, such as writing papers, writing presentation slides, the attitude at conferences, the implementation of programming languages, the concept of scalability, tuning parallel programs, and so on.

I also thank my colleague, Dr. Yoshihiro Oyama. It was a joyful and fruitful experience for me to work and discuss with him.

I thank Professor Naoki Kobayashi, who gave me many useful advice from theoretical view.

For my experimentation, I have used large scale multiprocessor machines from Human Genome Center, Institute of Medical Science, University of Tokyo.

I thank all members of Yonezawa laboratory and OB for help me to my research and daily life. Especially, I enjoyed working and talking with Tatsurou Sekiguchi, Toshihiro Shimizu, Haruo Hosoya, Atsushi Igarashi, Norifumi Gotoh, Kunio Tabata, Hirotaka Yamamoto, and Eijiro Sumii. And I wish to express my gratitude to machine administrators, because this work would not be possible without their tremendous effort.

Chapter 1

Introduction

Shared memory architecture is attractive platform for implementation of general-purpose parallel programming languages that support irregular, pointer-based data structures [15, 48, 6]. The recent progress in scalable network technologies has realized large scale shared memory multiprocessors. As machines get larger, it becomes more difficult for programmers to obtain sufficient performance of parallel application programs, especially, irregularly structured programs. Those application programs are divided into two groups:

scientific applications First, most of symbolic computation programs such as language processing application and searching algorithm heavily use irregular data structure to solve irregular problems. Besides, some numerical application such as simulation programs have irregular structures. For example, those programs may use tree data structure or sparse matrix structure as results of optimization to reduce redundant computation. Since recent parallel computers have complex memory hierarchy, processor architecture, and network structure, predicting application behavior has been more difficult. Thus even for programs that have regular problem structures, their performance may be degraded without dynamic task stealing or dynamic data structure creation, because processors are not always well-behaved.

network server applications Recent the world-wide expansion of Internet has made network server applications more important. Especially, HTTP servers, which may be accessed by a great number of peo-

ple are responsible to create dynamic contents fast. Therefore many contents providers have adopted parallel machines, such as shared memory multiprocessors and workstation/SMP clusters. Those applications heavily use dynamic data objects on memory, in addition to hard disks and network. For example, HTTP servers perform massive calculation of string objects to create HTML contents. To achieve high throughput, creating contents must be done in highly scalable method.

In many application programs with irregular structures, we cannot tell the memory usage beforehand, and programs need to extend their working set dynamically. Thus one of the key factors that determines the performance of irregular application programs is *dynamic memory management*. When the application program requires a memory object, it requests desirable size to dynamic memory management module. The memory management module allocates a memory region of requested size from the pool of free memory regions, originally obtained from OS.

The dynamic memory management module affects application performance by two reasons. (1) The speed of memory management itself: The frequency of allocation requests is sometimes very high; each application thread may request an object every $10\mu s$ user computation [37]. In such cases, allocation speed should be enough fast. Especially, if allocation task is serialized, like standard libc `malloc`, application performance would be heavily degraded. Not only allocation, an automatic memory reclamation module, called garbage collection (GC) [16, 40, 52] also consumes a substantial time during application execution. For some programs, GC time occupies more than 10% of overall execution time. (2) Memory locality of application: As memory hierarchy of computer architecture becomes deep, we cannot achieve good performance without accounting for memory locality. Since memory management module determines the address of memory objects that application programs access, the allocation policy affects application performance. Especially, this is important on distributed shared memory (DSM) machines, where memory access costs differ with regard to memory location among physical memory nodes.

This thesis describes the design and implementation of a scalable dynamic memory management module for large scale shared memory multi-

processors. The goal of this thesis is to improve application performance by construction of high performance memory management module. The module consists of two submodules:

- *The scalable parallel allocator* allows user threads to allocate objects in parallel. Our allocator achieves scalability of allocation, good locality of application, and high memory utilization. There is a tradeoff between locality and memory utilization. Our allocator struggles against the tradeoff by using thread local heap and dynamic region stealing.

- *The scalable parallel garbage collector* automatically reclaims memory objects unused by application. We have constructed two versions of parallel collector: One is *stop parallel garbage collector*. When the allocator notices memory shortage, our garbage collector module stops all threads and starts garbage collection (GC). To shorten application pause time, several threads cooperatively performs GC task. The key to achieve scalability is dynamic task stealing of GC. The other is *concurrent parallel garbage collector*, where collector threads and application threads run in parallel, and collection itself is done by multiple threads.

This thesis describes how we achieve scalability on large scale machines and machines with different architectures, such as symmetric multiprocessors (SMP) and distributed shared memory (DSM) machines. The specific contributions of this thesis is as follows:

- It proposes scalable allocation and garbage collection algorithm. It specifies bottlenecks that did not appear on smaller scale machines, and proposes optimization to eliminate them. It also empirically evaluates effects of the optimization through experimentation on large scale share memory machines and irregularly structured application programs.

- It proposes and evaluates optimization techniques specialized to memory architecture of each parallel machine. For DSM, on which remote memory access cost is large, our allocator gives local memory region to the requester if possible.

- It proposes and evaluates new allocation algorithm that enables users to control tradeoff between locality and memory utilization. Our module gives a chance to users to customize memory management behavior.
- It describes and justifies a performance prediction model of the parallel garbage collector. The model takes a heap snapshot as input, and outputs the predicted GC running time with any memory architectures and any number of processors. Understanding GC's behavior enables us to construct adaptive GC algorithm that customizes itself with regard to heap snapshot and architecture. This thesis shows the results of preliminary experimentation.

Our memory management module is independent from specific programming language and does not require any support of compilers. Garbage collection technique without compiler support has been proposed by Boehm et al. [11] before. Our implementation is based on their implementation, and adopts BIBOP allocator and mark-sweep collector as theirs does. We have not integrated yet our technique into other settings, such as copying collector, because garbage collectors that move memory objects after allocation require tight support from compilers. We believe, however, the technique that this thesis proposes is applicable to those environments.

1.1 Evaluation Settings

This thesis evaluates our memory management module on shared memory multiprocessors Sun Ultra Enterprise 10000 [12] and SGI Origin 2000 [38].

Ultra Enterprise 10000 Ultra Enterprise 10000 is a symmetric multiprocessor (SMP) with sixty-four 250 MHz Ultra SPARC processors. All processors and memories are connected through a crossbar interconnect whose total bandwidth is 10.7 GB/s. The system has sixteen memory modules, and latency between any processor and any memory module is uniform; the latency of read access is about 560 ns. Memory regions are automatically located fairly among all memory modules in round-robin fashion. Thus impact of memory access contention is much smaller than Origin 2000.

Origin 2000 Origin 2000 is a distributed shared memory machine (DSM). The machine we used in the experiment has eighty 195 MHz R10000 processors. That system consists of forty 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. Remote memory access latency is about 2–4 times larger than local access latency. The local latency of read access is about 270 ns, if no access contention occurs. In the default configuration, each page (whose size is 16 KB) is placed on one of physical memory node in 'first touch' rule; when a processor P accesses a page out of physical memory and raises swap-in, OS places the page on the memory node that is local to P. The benefit of first touch rule is that each processor can usually use fast local memory. On the other hand, if many objects that are shared among several processors live in a single memory node, memory access contention may cause a performance problem.

In order to evaluate how memory management module affects application performance, we have written some parallel benchmark applications in C++. CKY and Cube are parallelized by using StackThreads/MP [49], a fine grained thread library. Matmul and BH is written with low level thread libraries (Solaris threads on Enterprise 10000, and sproc systemcall on Origin 2000). We describe their characteristic that are relevant to this thesis.

Matmul Matmul is a simple program that computes product of dense matrices. In the experimentation, it multiplies two $N \times N$ matrices several times. Each row of matrices are allocated by threads in round-robin fashion (See Figure 1.1). To evaluate effects of allocator clearly, this program reallocate matrix rows before each multiplication. The allocation rate of Matmul is smallest among all benchmarks. Its performance is hardly affected by allocation speed itself; placement of allocated objects is more important on DSM.

BH BH solves N-body problem using the Barnes-Hut algorithm [4]. Each simulation step consists of two phase: in tree construction phase,

BH makes a tree whose leaves correspond to particles (See Figure 1.2). In calculation phase, it calculates the acceleration, speed, and location of the particles by using the tree. Memory objects are allocated only in the first phase.

We have implemented two versions of BH.

BH-st (sequential tree construction) : Only calculation phase is parallelized, and tree construction phase is sequential. Because calculation phase dominates the total runtime of BH application, this implementation is proper at earlier phase in developing application. However, only a single thread performs tree construction, the placement of memory objects is imbalanced on DSM.

BH-pt (parallel tree construction) : Both phase is parallelized. All threads allocate nodes and add them to a tree, thus the placement of memory objects is almost balanced on DSM.

CKY CKY is a parser of context free grammars. It takes sentences written in natural language and the syntax rules of that language as input, and outputs all possible parse trees for each sentence. For each input sentence, CKY constructs a single large matrix M (See Figure 1.3). M's cell $M[i, j]$ contains a list that consists of all nonterminal symbols for substring from i th word to j th word. Threads calculate lists for each cell in bottom-up, and finally obtains top cell, that corresponds to a list of parse trees for whole sentence.

Cube Cube searches an approximate solution of the Rubik's cube puzzle in breadth first fashion. Cube calculates all states within three steps from the first state, and selects the best twenty states (See Figure 1.4). Cube repeats these processes five times. Because threads allocate the state records in parallel, live objects are distributed in all memory nodes on DSM.

1.2 Organization of Thesis

The rest of the thesis is organized as follows. Chapter 2 describes algorithm of the scalable parallel allocator and its performance. Chapter 3 presents our

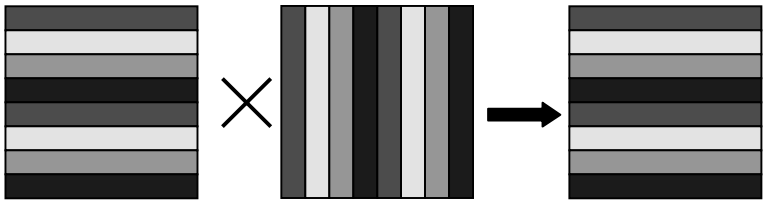

Figure 1.1: Data structure of Matmul benchmark.

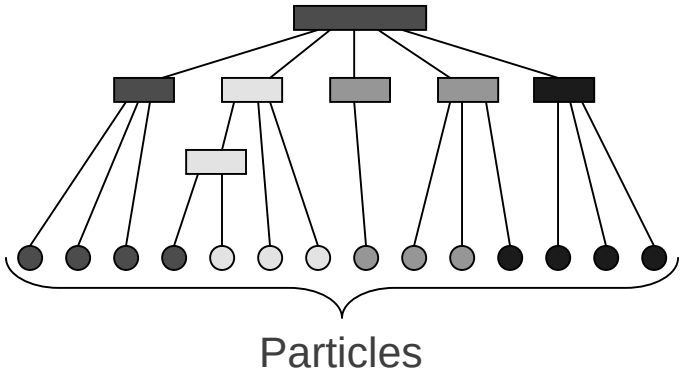

Figure 1.2: Data structure of BH benchmark.

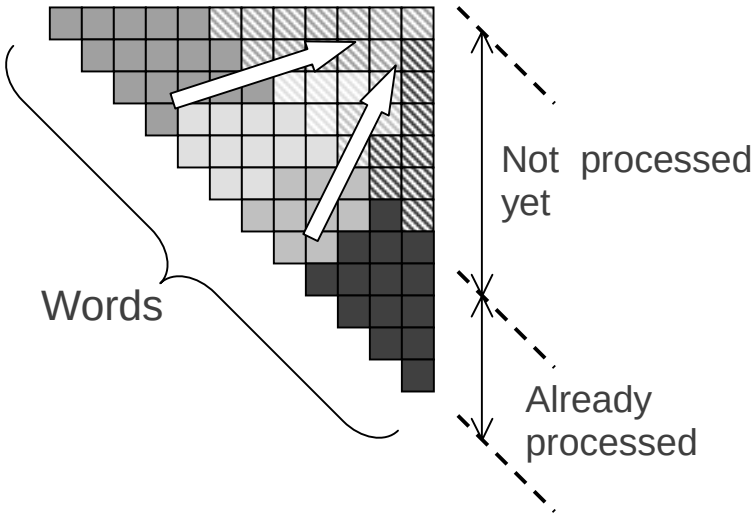

Figure 1.3: Data structure of CKY benchmark.

stop parallel garbage collector, and shows its scalability. The performance of parallel GC heavily depends on memory architecture. To understand GC performance in detail, Chapter 4 gives a performance prediction model of GC. Chapter 5 describes another version of GC; a concurrent parallel garbage collector. Finally, we summarize the work and mention future work in Chapter 6.

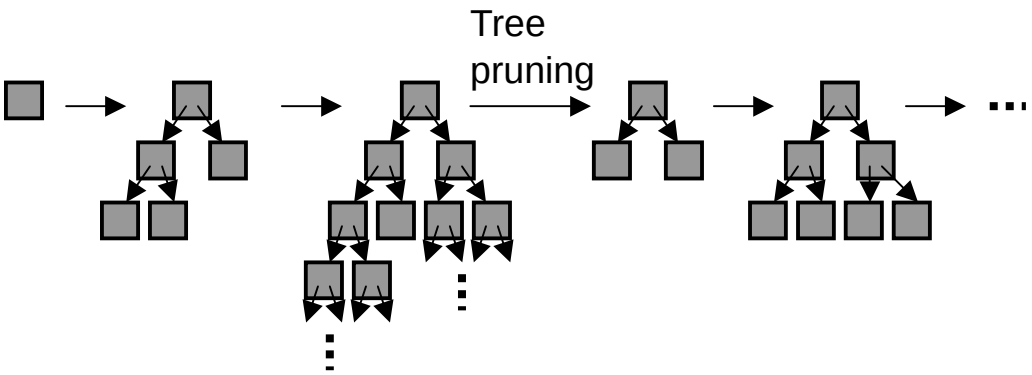

Figure 1.4: Data structure of Cube benchmark.

Chapter 2

Scalable Parallel Allocator

This chapter presents a parallel algorithm of a BIBOP memory allocator on shared memory multiprocessor, named Locality-aware page shared(LPS). The goal is scalability, locality, and high memory utilization. LPS algorithm allows users to control the tradeoff between locality and memory utilization on distributed shared memory (DSM) machines. The memory consumption of LPS allocator is k times larger than that of the most economical method (where k is a given constant). By using more memory, each thread can allocate local memory in a higher probability. To achieve this property, each thread maintains free pages locally. By comparing the current memory consumption and a threshold derived from k, each thread determines whether it should obtain remote pages or consume new pages. The experimental results on Origin 2000 DSM shows that users of LPS algorithm can control the balance between locality and memory utilization by adjusting k. LPS allocator is highly scalable because of thread local free list. It achieves 36 fold speedup with 64 threads.

2.1 Introduction

Parallel application programs that have irregular structure tend to allocate memory objects very frequently. The performance of such programs is heavily affected by the performance of memory allocator. However, many widely used allocators, such as `malloc` function in libc library, are not parallelized. While many OS vendors state that their libraries are 'multi-threaded', many of them do not assure that the libraries are parallelized. Actually, many allocators that OS vendors provide are sequential; only one thread can allocate

an object at a time. If several application threads call such allocators for many times, the allocator would become bottleneck and limit the scalability of application significantly. Therefor, for memory-intensive programs, we require parallel allocators, with which several threads allocate objects simultaneously.

Not only allocation speed itself, allocators should take care of locality. Locality is important especially on distributed shared memory (DSM) machines such as Origin 2000 [38], where memory access cost depends on memory location. Allocators should make an effort to give local memory regions to requester thread, rather than remote memory regions. Because allocated memory regions tend to be accessed by their requester thread, better locality of allocation may be able to improve application speed [1]. As we will see below, however, when allocators focus on only locality, it may consume much more memory regions than the application requires essentially. If the memory consumption of allocators is not bound, it would cause bad effects on other processes running on the machines; so we claim that allocators should consider memory utilization.

This chapter proposes a parallel allocator algorithm. The goal of our allocator is to achieve good scalability of allocation, good locality, and good memory utilization. The key point to achieve these three condition is the management policy of free regions. Generally, any memory allocators must manage free memory regions to fulfill allocation requests from application threads. Many allocators, including ours, use data structures called *free lists* to stock free regions.

Achieving scalability and locality is not a hard problem; using thread local free lists would suffice these two conditions. If allocator manages distinct free lists for each application thread, the thread can obtain new memory regions from its own free lists without any synchronization. Besides, we can achieve good locality, by keeping free regions that are local to thread i in thread i's free lists. However, this simple method can not achieve good memory utilization; it may consume much memory regions for some programs. Consider a program where the timing of memory utilization among

[1] Of course, there is an exception; not all objects are accessed by their own requesters, and some may be shared among several threads. If almost objects in the application are shared, the efforts of locality-aware allocators may be useless.

threads differ may cause problem, as shown in Figure 2.1 (B). In this program, thread 1 uses memory regions of m bytes in some times, and thread 2 uses m bytes in other times. The total size of used memory regions at one time is at most m bytes. For this program, simple allocators with thread local free lists would consume m bytes for thread 1, and m bytes for thread 2. Thus the allocators totally consume $2m$ bytes for this program, though m bytes would suffice for program execution. Generally, in worst case, the simple allocators may consume P (the number of threads) times larger memory regions than that program requires.

To achieve all of three conditions, this chapter proposes a new algorithm, named *Locality-aware page shared (LPS)* algorithm. It is based on Big bag of pages (BIBOP) allocator. It adopts thread level free lists for scalability and locality. To limit memory consumption, it allows threads to steal free regions from other threads' free lists. Moreover, LPS allocator allows users to control a tradeoff between locality and memory utilization. Users have only to set a constant variable named *allowable consumption ratio* k. LPS allocator consumes up to k times larger memory regions than requisite, for the purpose of better locality. If users focus on memory consumption rather than locality, k should be set to a small number, such as 1.0. If users want to improve locality, k should be set to a larger number.

Although we use garbage collection (GC) in the experimentation, the discussion in this chapter can be applied for explicit memory release, such as `free` function.

Section 2.2 describes our allocator algorithm, and Section 2.3 analyzes the limitation of memory usage. Section 2.4 shows experimental results on SGI Origin 2000 DSM. Section 2.5 mentions related work, and Section 2.6 summarizes this chapter.

2.2 Algorithm

The target programs of our parallel allocator are multi-threaded application programs with kernel level threads, such as pthreads. We assume each application thread is bound to distinct processor. Thus we will use the words 'thread' and 'processor' similarly. All objects are included by the single shared heap, and any thread can access any objects in the heap.

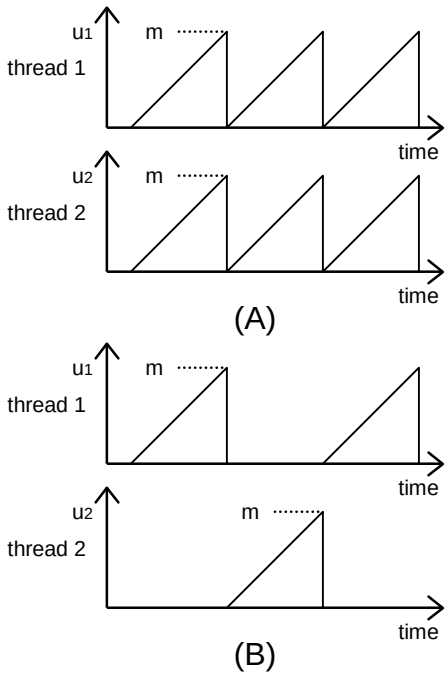

Figure 2.1: Graphs show transition of memory usage by parallel application with two threads. For program (A), both PS algorithm and AL algorithm consume same amount of memory region. For program (B), AL method may consume much more region than PS algorithm.

The rest part of this section describes our parallel allocator algorithm. Section 2.2.1 describes BIBOP allocator, which our allocator is based on. Section 2.2.2 shows some assumption about underlying architecture. Section 2.2.3 explains naive allocator algorithms and their problems. Section 2.2.4 gives our new algorithm, named LPS.

2.2.1 BIBOP Allocator

The BIBOP allocator divides the heap into fixed sized pages, as shown in Figure 2.2. In this thesis, we assume the BIBOP page size is equal to OS page size, although this condition is not essential. Each page contains objects of a particular size [2]. The allocator manages two kinds of free lists: *free object lists* and *free page list*. The former lists are for free regions that are smaller than page size. The allocator maintains a set of free object lists; each list contains objects of corresponding size. The free page list contains pages that are completely free.

When an application thread requests a memory region, the allocator provides a new region in the following algorithm. Here, we assume requested size is smaller than page size.

1. Allocator examines the free object list for requested size. If the list contains any free regions, the allocator returns one of them to the requester thread. Otherwise, allocator goes to the next step.

2. Allocator examines the free page list. If allocator finds any free pages, it takes one of them. Then the allocator divides the page into small regions of the requested size, and pushes them onto a free object list. Then it returns one of the regions to the requester thread. If free page list is empty and allocator fails, allocator goes to the next step.

3. Now the heap is exhausted. The allocator makes up free pages by invoking garbage collection(GC), or requesting new pages from OS by using systemcall such as `mmap` [3].

[2] In real implementation, objects of similar size may be placed into a single page by rounding up the size.

[3] To reduce the number of invoking systemcall, the allocator requests several pages at once, and pushes them into another list for fresh pages.

Note that the allocation processes usually finish in Step 1, because Step 2 (that is rarely invoked) fills the free object list with many small regions. The exception occurs because of large objects; when a thread requests a larger region than page size, allocator skips Step 1 and examines free page list first. In this chapter, we assume that application threads rarely request such large objects.

When an allocated object is released by user (`free()`) or GC, the region is pushed into free object list that corresponds to the size of the region. If allocator finds the page that contains freed region empty, the page is returned to the free page list.

2.2.2 Assumption

Our parallel allocator works on any shared memory multiprocessors: SMP (symmetric multiprocessor) and DSM (distributed shared memory). Our locality-aware algorithm especially focuses on DSM machines, where locality is more important. The DSM machine Origin 2000, on which we evaluate our allocator, consists of several nodes, each of which has two processors and physical memory node. Remote memory access latency is $2 \sim 4$ times slower than local access latency.

We make some assumptions about underlying architecture. First, we assume that each page is placed on one of physical memory node in 'first touch' rule; when a processor P obtains a page from OS and raises swap-in, OS places the page on the memory node that is local to P. Origin 2000 adopts the first touch rule.

Next, in the following discussion, we assume no swap-out and no page migration occurs during application execution. Under this assumption, the placement of each memory node is permanent.

2.2.3 Naive Parallel Algorithms and the Problem

By using two kinds of free lists, we can invent some simple parallel allocators (see Figure 2.3) [4].

[4]We could adopt other approaches; for example, we can use both local free object lists and shared free object lists, as Hoard allocator [5] does. The author plans to investigate such approaches in future.

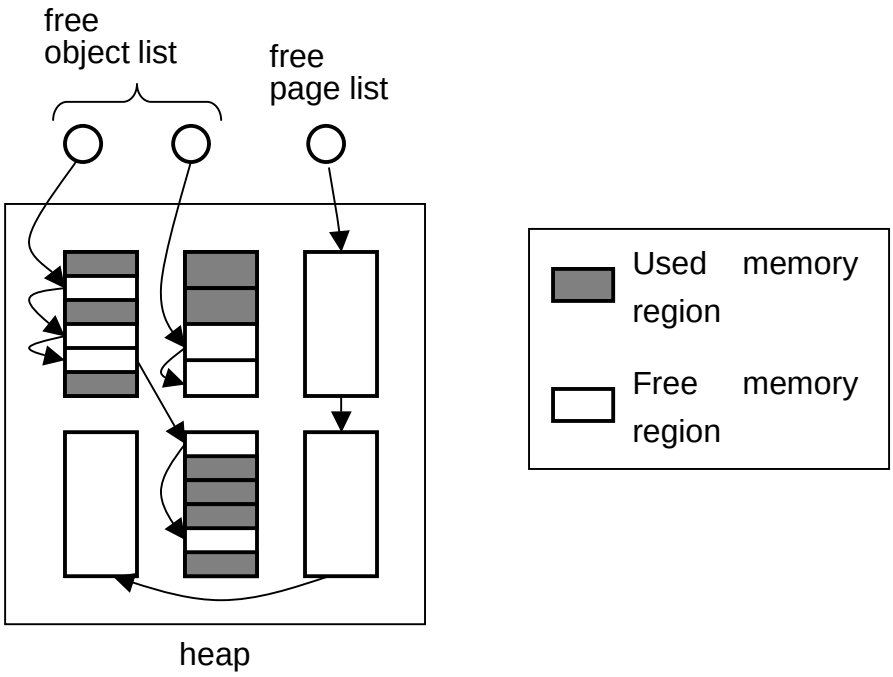

Figure 2.2: Heap structure of BIBOP allocator. The heap consists of fixed-sized pages. Each page includes objects of the same size. To keep track of free regions, allocator uses free objects lists and free page list.

- *All-shared (AS) algorithm*: All free lists are shared by all threads. Actually, this is not a parallel allocator; the rest part of the thesis ignores this approach.
- *Page-shared (PS) algorithm*: Each thread maintains free object lists locally, while the free page list is shared.
- *All-local (AL) algorithm*: Each thread maintains all free lists locally.

AL algorithm is a good choice for scalability and locality. It is scalable because allocation requests from several threads are processed independently owing to thread local free lists. AL can achieve good locality, because free regions are completely separated among threads and each memory region is always reused by a single thread. However, it may not achieve good memory utilization, as we have seen in Section 2.1. To suppress memory consumption, threads should share free regions.

PS algorithm takes a middle position between AS and AL. It can achieve scalability by using thread local free object list. The allocation process usually finishes in Step 1 described in Section 2.2.1, where synchronization is not required. Besides, PS bounds memory consumption by sharing free page list. However, it does not achieve good locality on DSM, because free pages are reused by any threads. PS never accounts for the affinity between threads and memory regions.

Discussion about Locality

In spite of above description, PS algorithm achieves better locality than AS (All-shared) algorithm. PS, AL (and LPS described below) algorithms manage thread local free object lists. The main purpose is scalability; each thread can usually allocate an object without synchronization. Moreover, the local free object lists produce a secondary effect on locality. With thread local lists, we can segregate objects allocated by different threads into different pages. This can prevent unintended false sharing and reduce cache invalidation. Not only DSM, SMP machines can gain this benefit. This is impossible with AS algorithm, where any threads obtain object from a single set of free object lists.

2.2.4 LPS Algorithm

This section describes Locality-aware Page-Shared (LPS) allocator algorithm, which achieves scalability, locality, and good memory utilization. LPS allows users to control the tradeoff between locality and memory utilization. The upper bound of consumption by LPS algorithm is as k times as that by PS algorithm, where k is *allowable consumption ratio*, which are specified by users. The range of k is $1 \leq k \leq P$, where P is the number of threads.

In LPS algorithm, each thread maintains its own free object lists and free page list, as in AL algorithm. Unlike AL algorithm, LPS allows threads to steal free pages from other threads' free page lists.

Allocation process by thread i is as follows.

- 1. Allocator examines the free object list of thread i and tries to obtain a free region. If it fails, allocator goes to the next step.
- 2-1. If allocator finds a page in free page list of thread i, that page is used for requested object. Otherwise, allocator goes to next step.
- 2-2. If function *heap-expansion-allowed()* described below returns 'true', allocator goes to Step 3 immediately. Otherwise, allocator goes to Step 2-3.
- 2-3. Allocator examines free page lists of all other threads in turn. If allocator finds an empty page in a list of any thread, allocator steals the page from its owner thread and uses the page for the request. Otherwise, allocator goes to Step 3.
- 3. Allocator invokes GC, or expands the heap by obtaining new pages from OS.

In Step 2-2, we use a function named *heap-expansion-allowed()* to determine whether the thread should steal free pages from others, or obtain new pages. Intuitively, this function returns 'true', if the current memory consumption by allocator is less than a certain threshold. In this case, LPS allocator consumes new pages and makes them local to the requester thread i. If the function returns 'false', It is more important to suppress consumption than to improve locality.

Before the definition of the function *heap-expansion-allowed()*, we define some variables: *the amount of used pages* u, *the amount of living pages* l, and *the amount of consumed pages* a. Their values change as program execution proceeds. We let u be the number of pages that contain objects allocated by application program. Within u, we let u_i be the number of pages whose owner is thread u_i (Thus $u = \sum_i u_i$). We let l be the number of pages that contains objects that have survived the most recent GC. If application program releases objects explicitly instead of GC, we simply let l be $l = u$. We let a be the number of pages that allocator has obtained from OS since program started. From their definition, a condition $a \geq l \geq u$ holds. a increases monotonously as program proceeds, because our allocator does not return any pages to OS. While u and l is determined by application program behavior [5], the consumption a is affected by both of u, l and allocator algorithm. The goal of LPS allocator is to restrict growth of a.

Now we define the function *heap-expansion-allowed()*. It returns 'true', when $a < k(\max l)$ and $a < \sum_i(\max u_i)$. Here $(\max l)$ and $(\max u_i)$ are the maximum number of l and u_i since program has started.

Intuitively, $(\max l)$ is the approximation of the number of pages the application program requires essentially. LPS algorithm aggressively obtain new pages from OS, until the consumption reaches $k(\max l)$. When users specify a larger number for k, LPS tries to achieve better locality, because each thread has more chances to obtain new pages and localize them, rather than to steal remote pages. When $k = P$, LPS behaves like AL algorithm. As k is smaller, LPS consumes less memory regions. When $k = 1$, LPS consumes as much regions as PS algorithm does. Notice that even if $k = 1$, locality of LPS algorithm is better than PS algorithm, because each thread examines its own free page list at first.

2.3 Analysis of Memory Consumption

This section analyzes the amount of consumed pages of each allocator algorithm. In this discussion, we let a_{ps}, a_{al}, a_{lps} be the consumption of PS, AL, LPS algorithms, respectively. They are determined by algorithm itself and the page usage by application program u, u_i.

[5] To be exact, the timing of GC invocations affects u and l. This thesis ignores its impact to make a discussion simpler.

This section will show that conditions $a_{lps} \leq k a_{ps}$ and $a_{lps} \leq a_{al}$ hold, for any transition of u and u_i.

To analyze consumption of PS algorithm, We let d be the number of free pages in shared free page list. Those free pages are used for allocation requests from any threads, therefore allocator obtains new pages from OS (and a_{ps} increases) only if $d = 0$ and $u = (\max u)$. Therefore the condition $a_{ps} = (\max u)$ holds at any time.

In AL algorithm, we let d_i be the number of free pages in thread i's free page list. Since free pages does not migrate among threads in AL algorithm, the memory consumption of each thread is determined independently from other threads. Here we define local consumption of thread i as $a_{al,i} = u_i + d_i$. Thread i obtains new pages from OS whenever $d_i = 0$, thus $a_{al,i} = (\max u_i)$. The total consumption is $a_{al} = \sum_i a_{al,i} = \sum_i (\max u_i)$.

Now we analyze the consumption of LPS algorithm. The definition of d_i is the same as in AL algorithm, and we let d be $d = \sum_i d_i$.

First, we show that $a_{lps} \leq k a_{ps}$ holds. When thread i tries to obtain a free page and finds its own free page list empty ($d_i = 0$), one of following tasks is done.

1. If $a_{lps} < k(\max l)$, allocator obtains new page from OS. In this case, a_{lps} increases.

2. If $a_{lps} \geq k(\max l)$ and $d > 0$, some other threads have free pages. In this case, since allocator steals a free page from other threads, a_{lps} does not change.

3. If $a_{lps} \geq k(\max l)$ and $d = 0$, no thread have any free pages. Allocator obtains new page from OS effectively and a_{lps} increases. In this case $a_{lps} = (\max u)$ holds because $d = 0$.

The value a_{lps} increases in case (1) and (3), thus we can say $a_{lps} < k(\max l)$ or $a_{lps} = (\max u)$ at any time. From definition of l, $l \leq u$ and $k(\max l) < k(\max u)$ holds. And since $k \geq 1$, $(\max u) \leq k(\max u)$ holds. Therefore $a_{lps} \leq k(\max u) = k a_{ps}$ holds.

Secondly, $a_{lps} \leq a_{al}$ holds because of the function *heap-expansion-allowed()* asserts $a_{lps} < \sum_i (\max u_i)$. From above discussion, the consumption of LPS algorithm is bounded by $a_{lps} \leq k a_{ps}$ and $a_{lps} \leq a_{al}$. The consumption of

LPS is same as that of PS when $k = 1$, and when k is large enough, it is similar to that of AL.

2.4 Performance Evaluation

We have implemented PS, AL, LPS parallel allocation algorithms. This chapter evaluates the performance of our parallel allocator through experimentation on SGI Origin 2000 DSM.

First, we will show the peak throughput of allocation itself by using a micro benchmark. Then we will show the performance of some applications and compare the impact of allocator algorithms.

2.4.1 Scalability of Allocation

We have measured throughput of parallel allocation by using a simple parallel benchmark that repeats allocation and release of objects for many times. In this benchmark, each thread allocates 100,000 objects, each of which has the size of 16 bytes, and releases all of them. We repeat the task 30 times, and measure the execution time.

Figure 2.4 and Table 2.1 shows total throughput of allocation among threads. The horizontal axis corresponds to the number of threads, and the vertical axis is the number of total allocation operation per second. The figure and table show the performance of PS, AL, LPS algorithms. It also includes the performance of libc `malloc` for comparison. We can see while libc is faster than our allocators on serial execution, it cannot achieve scalability at all. Throughput on parallel execution is lower than that on serial execution. On the other hand, all of PS, AL, LPS can achieve high scalability, owing to thread local free object lists. Within these allocators, the performance of AL and LPS is better than PS. We consider this is due to access contention to shared free page list. AL and LPS achieves similar performance; 36-fold speed-up with 64 threads. PS achieves 21-fold speed-up. We can see the allowable consumption ratio k does not visible effects on the performance.

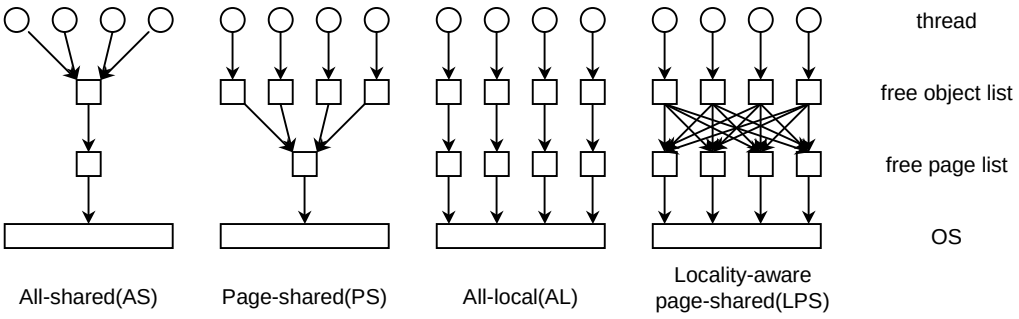

Figure 2.3: Some design policies of parallel BIBOP allocators. Figures show All-shared(AS) algorithm, Page-shared(PS) algorithm, All-local(AL) algorithm, Locality-aware-page-shared(LPS, ours) algorithm. The arrows show the flow of memory allocation requests.

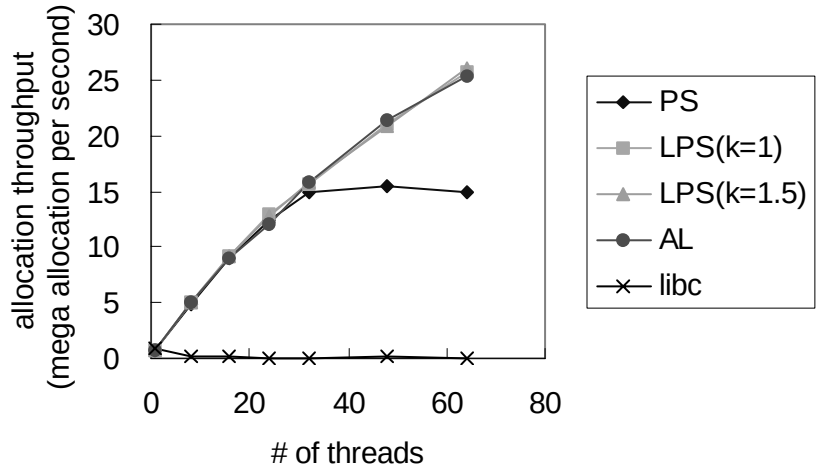

Figure 2.4: Memory allocation throughput. The horizontal axis shows the number of threads, and the vertical axis shows the number of malloc ($\times 1,000,000$) per second by all threads.

2.4.2 Memory Consumption and Locality

This section shows that LPS algorithm enables users to control tradeoff between locality and memory utilization. For this purpose, we use a benchmarks where memory usage by each thread is unfair, like a program (B) in Figure 2.1. In this benchmark, only one thread can allocate objects at one time and after it releases those memory regions, another thread allocates objects. The amount of memory region each thread allocates is about 5MB.

Figure 2.5 shows the memory consumption and the locality on this benchmark. The upper graph shows memory consumption of our allocators. The vertical axis corresponds to amount of memory consumption; the product of page-size and the number of consumed pages by allocator during whole benchmark execution. With PS allocation algorithm, memory consumption remains almost constant regardless of the number of threads. On the other hand, AL algorithm consumes much more memory as threads increase; it consumes about $5P$ MB, where P refers to the number of threads. The graph shows LPS algorithm can control consumption by changing the allowable consumption gap k. When $k = 1$, its consumption is similar to that of PS algorithm, and as k gets larger, LPS consumes more memory regions.

The lower graph shows locality on the same benchmark. The vertical axis shows *remote page allocation ratio*; which means the ratio that each thread has obtained local free pages from free page list. In AL algorithm, each thread always obtain local page; the ratio is zero. On the other hand, in PS algorithm, the remote page allocation ratio is about $1 - 1/P$. The locality performance of LPS algorithm is between AL and PS. Even if $k = 1$, LPS algorithm exhibits better locality than PS, because of local free page list per thread.

2.4.3 Performance of Application Programs

This section evaluates effects of allocator on application performance. We use three application programs matmul, BH-pt, and CKY, which we have described in Section 1.1.

Table 2.2 shows the behavior of applications, such as the amount of memory objects allocated, and allocation speed (the number of allocation requests per second). The table describes execution with 64 application

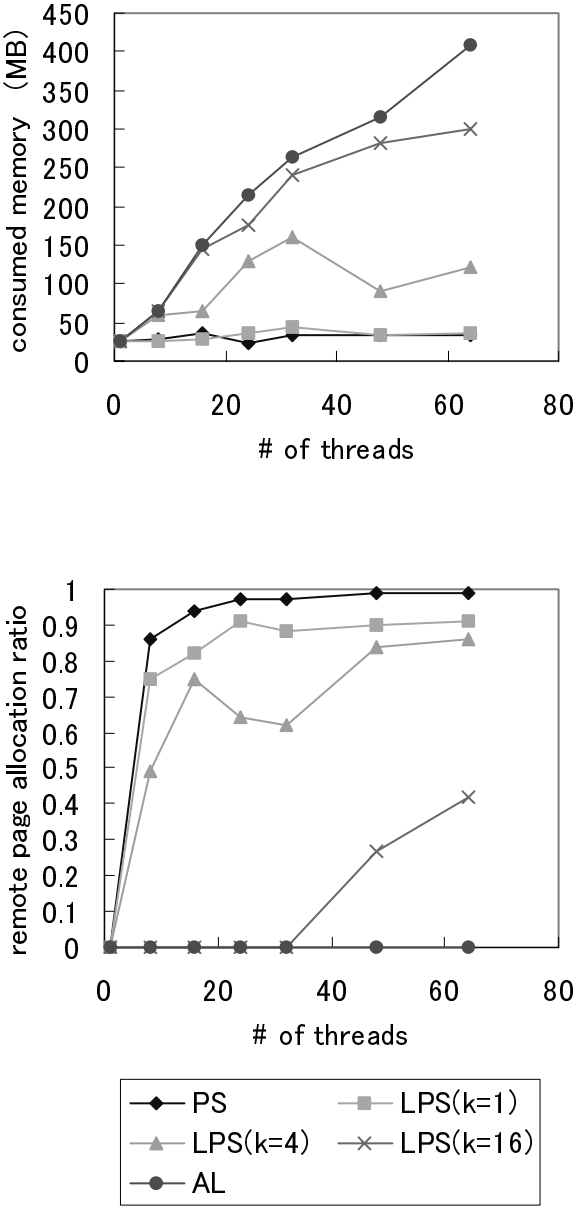

Figure 2.5: Memory consumption and remote page allocation ratio on unfair benchmark.

threads and LPS ($k = 1$) allocator. Except for execution time, it shows total numbers among all threads.

Matmul (matrix multiply): In the experiment in this section, Matmul multiplies two 1000×1000 matrices 30 times. As Table 2.2 shows, The allocation speed is much less than other benchmarks.

BH-pt (N-Body solver): In the experiment, we simulate 100,000 particles for 30 time steps. We use BH-pt version, where both tree construction phase and force calculation phase are done in parallel.

CKY (CFG parser): In the experiment, we parse 200 sentences, each of which consists of 36 to 100 words. CKY's allocation speed (1.16 M) is the highest among three benchmarks, though it is much less than the peak throughput (26M) shown in previous section.

Figure 2.6–2.8 show the performance of three applications on Origin 2000. Each figure contains graphs of memory consumption, remote page allocation ratio, and application execution time. We have omitted garbage collection time from execution time, while it includes memory allocation time. The lower right graph shows the change of execution time compared with that with LPS algorithm.

Figure 2.6 shows performance of matmul. Since all threads allocate same amount of memory, the memory consumption is similar among allocator algorithms. On the other hand, the choice among allocator algorithms affects locality heavily. The remote page allocation ratio of LPS and AL is much lower than LPS. With LPS ($k = 1.5$) and AL, the ratio is zero. The execution times with LPS and AL algorithms are 3–19 % faster than that with LPS algorithm. The frequency of memory allocation of this application is much lower than other applications, and execution time are dominated by computation time. Therefore we consider this speed-up is due to improvement of locality. With LPS or AL allocator, each thread has more chance to access local memory than with PS allocator.

Figure 2.7 shows performance of BH-pt. AL algorithm tends to consume more memory regions than PS algorithm. Currently the reason why LPS sometimes more memory than AL, or less memory than PS is not clear. While AL and LPS achieves good locality, the effect on overall performance

# of threads	1	8	16	24	32	48	64
PS	0.70	4.88	8.96	12.47	14.95	15.42	14.93
LPS(k=1)	0.69	5.01	9.16	12.94	15.66	21.08	25.68
LPS(k=1.5)	0.69	4.99	9.13	12.73	15.78	20.82	26.06
AL	0.70	5.05	8.90	12.09	15.88	21.33	25.40
libc	0.98	0.16	0.10	0.07	0.07	0.10	0.05

Table 2.1: Memory allocation throughput. The number of malloc ($\times 1,000,000$) per second by all threads is shown.

Benchmark	Total memory allocated (MB)	# of allocation	Exec. time (sec)	Allocation speed (per sec)
Matmul	496.2	65.1K	25.7	2.54 K
BH-pt	2223	28.6M	174.6	164 K
CKY	1203	79.2M	68.5	1.16 M

Table 2.2: The amount of memory objects allocated, execution time without GC time, and allocation speed of applications. This table shows statistics on Origin 2000 with 64 threads. LPS ($k = 1$) allocator is used. Except for execution time, the total numbers among all threads are shown.

is less than matmul application. With LPS ($k = 1.5$) algorithm, the execution time is 2 % faster than that with PS with 64 threads. With AL, the improvement is about 4%.

Figure 2.8 shows performance of CKY. This application involves less parallelism than other applications, and we cannot achieve speed-up with more than 16 threads. In CKY, memory consumption is significantly affected by the choice of allocator algorithms. AL algorithm consumes three times larger memory region than PS algorithm. The remote page allocation ratio with AL and LPS is much lower than that with PS. The improvement of overall performance with AL or LPS is 2–5 %. LPS algorithm can achieve similar execution speed as AL, while LPS consumes much less memory than AL.

2.5 Related Work

Because many standard memory allocator such as libc malloc is not parallelized, it is not easy to achieve good scalability for memory intensive application. Many parallel programmer have constructed their own memory management routine for better performance. For example, Apache multi-threaded HTTP server [27] adopts custom-made region allocator. This approach is a heavy burden to programmers.

We believe desirable approach is adopting the general purpose parallel allocator. Some researchers have designed and implemented parallel allocator.

The parallel allocator by Larson et al. [37] maintains several partial heaps to reduce contention to shared resources. Empty regions larger than page size are reused by any threads.

In the parallel BIBOP allocator by Boehm [8], each thread maintains a part of free objects and other resources are shared. The focus of the parallel BIBOP allocator by Berger et al.

[5] is achieving scalability and memory utilization, and their allocator maintains both thread local heaps and shared heap. Unlike our allocator, threads share non-empty pages in addition to empty pages, therefore its memory consumption is lower than that of our allocator. On the other hand, application may suffer from false sharing. However, they claim that

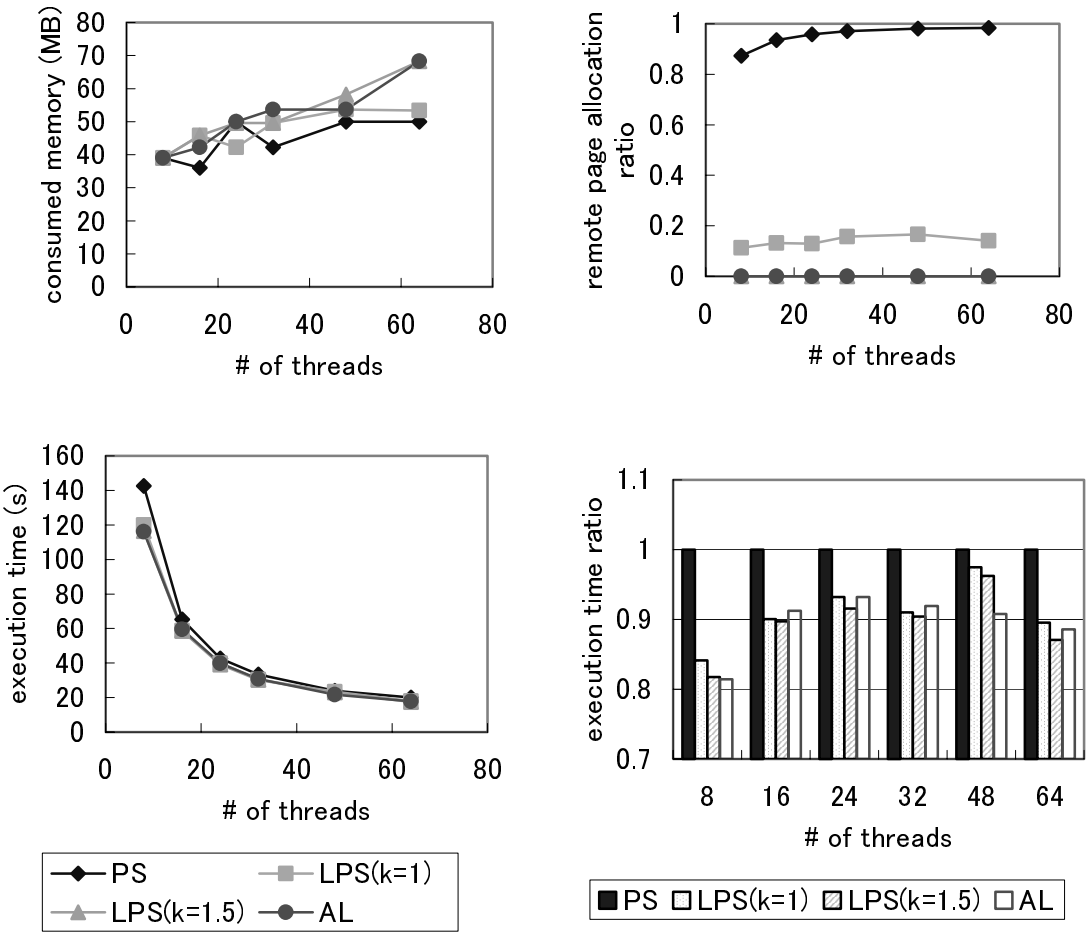

Figure 2.6: Performance of matmul application on Origin 2000.

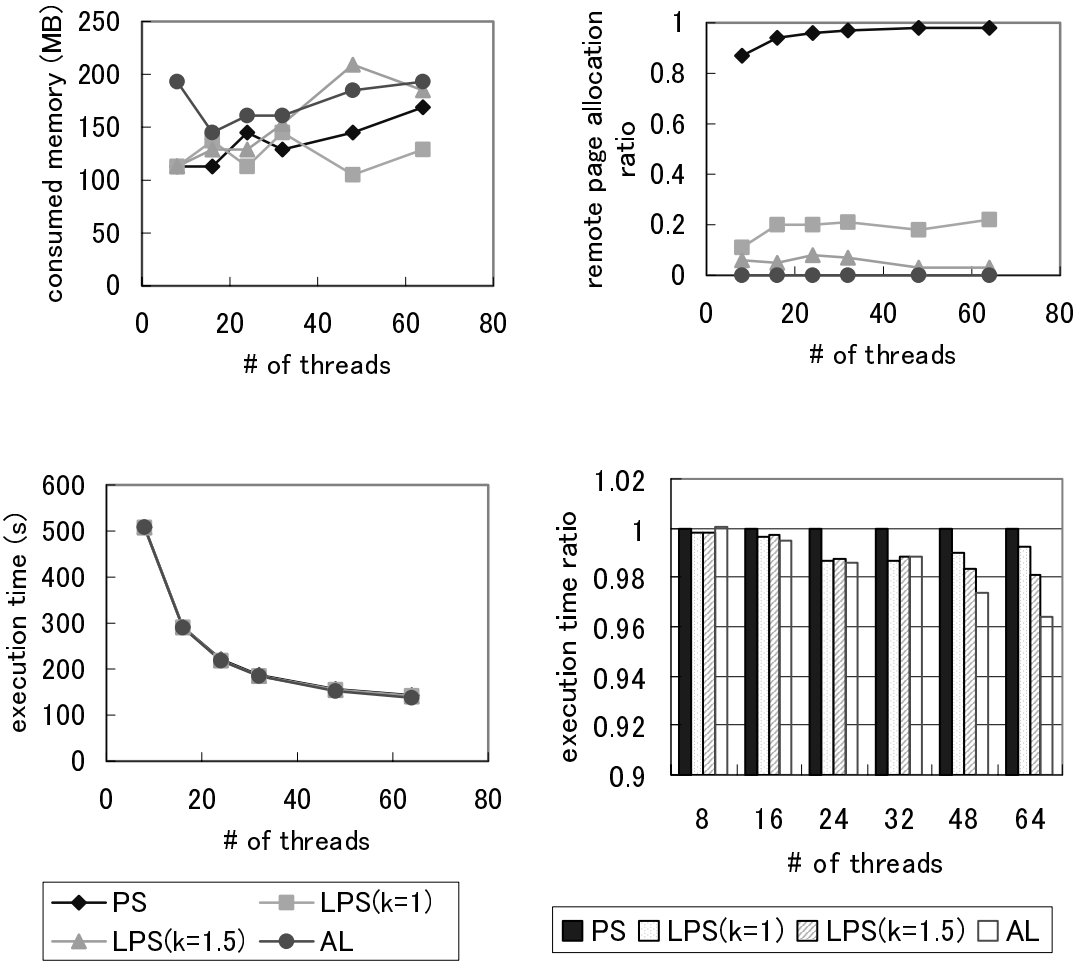

Figure 2.7: Performance of BH-pt application on Origin 2000.

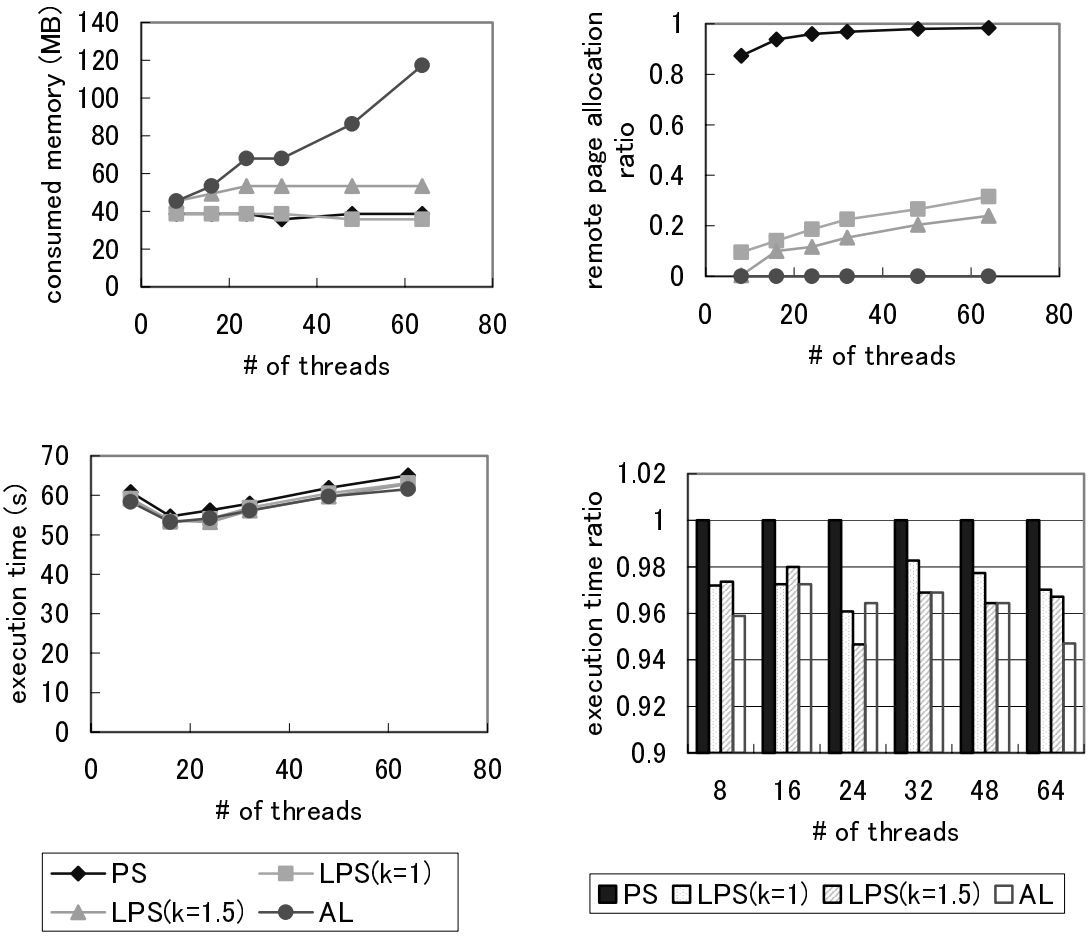

Figure 2.8: Performance of CKY application on Origin 2000.

the effect of false sharing is very small. They have also shown the upper bound of memory consumption analytically.

All parallel allocators focus on scalability of allocation task, however, to my knowledge, they does not mention locality on DSM machines. The parallel allocator this chapter proposed accounts for scalability, memory utilization, and locality.

2.6 Summary

This chapter has proposed a parallel extension to BIBOP allocator for shared memory multiprocessor, named Locality-aware-page-shared(LPS) algorithm. The goal is to achieve scalability, locality on DSM machines, and high memory utilization. There is tradeoff between locality and memory utilization. LPS algorithm enables users to control the tradeoff by simply changing a constant variable named allowable consumption ratio k. LPS algorithm guarantees the upper bound of its memory consumption is as k times as that of the most economical allocator. As the allocator consumes more memory, application locality is improved.

Through experiments on large scale multiprocessor, SGI Origin 2000, we evaluated the performance of our allocator. We found our allocator is highly scalable; the speed-up of allocation task is 36-fold with 64 threads. With our LPS allocator, application execution time is improved by good locality and fast allocation; application gets 2–19% faster than the economical (PS) allocator.

Future work concerned with this chapter includes investigation of the relation between GC invocation timing and memory consumption. We require more research on bound of locality, and automatic regulation of allowable consumption ratio.

Chapter 3

Scalable Parallel Garbage Collector

This chapter 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 80 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) and Cube (a Rubik's cube puzzle solver).

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 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.

3.1 Introduction

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 [20, 32, 43], 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 chapter describes the implementation of a stop 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 [10, 11]) on Ultra Enterprise 10000 SMP and SGI Origin 2000 DSM. 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 chapter is organized as follows. Section 3.2 compares our approach with previous work. Section 3.3 briefly summarizes the memory management method. Section 3.4 describes our parallel marking algorithm and solutions for performance limiting factors. Section 3.5 shows experimental results, and we conclude in Section 3.6.

3.2 Previous Work

Most previous published work on GCs for shared-memory machines has dealt with *concurrent GC* [20, 32, 43], in which only one thread 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 3.1). 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 several threads. Several systems use this type of collectors [30, 41] and we believe there are many unpublished work too, but there are relatively few published performance results. 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 [34]. It 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 thread collects its local heap individually. Collection on the shared-heap is done cooperatively but asynchronously. During a collection,

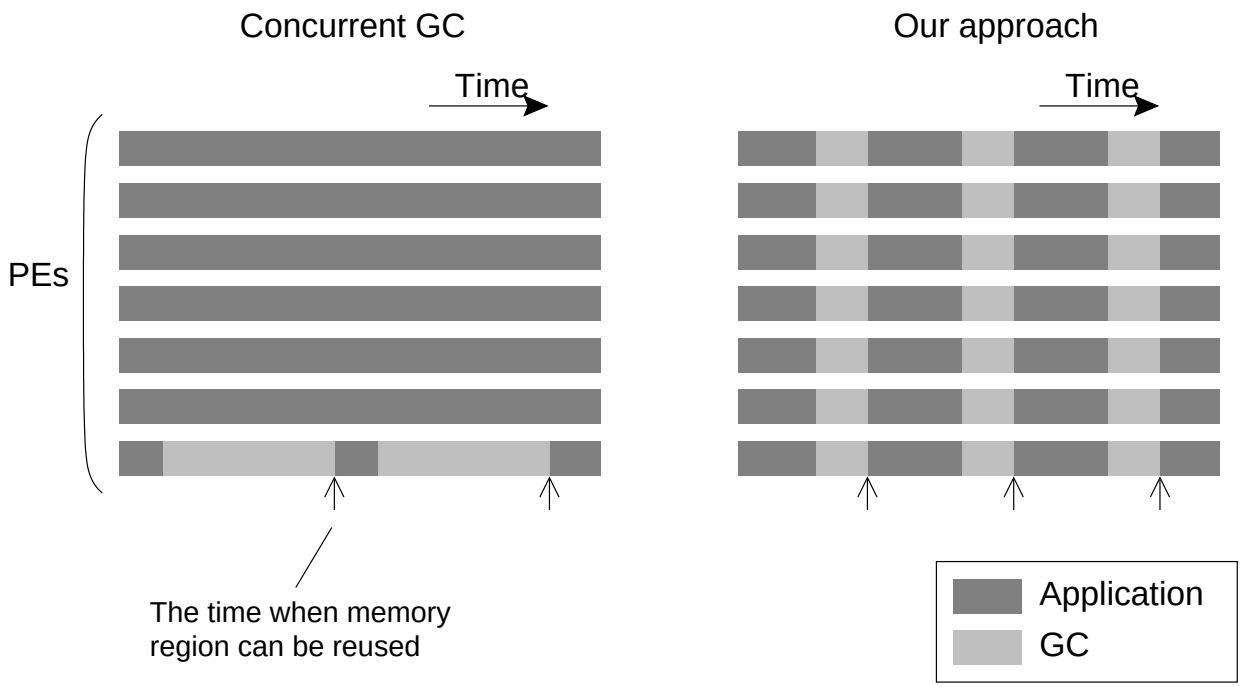

Figure 3.1: Difference between concurrent GC and our approach. If only one dedicated thread performs GC, a collection cycle becomes longer in proportion to the number of threads.

live data in the shared-heap (called 'from-space' of the collection) are copied to another space called 'to-space'. Each thread, on its own initiative, copies data that is reachable from its local heap to to-space. Once a thread 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).

In this chapter, we adopt "Stop-Parallel" approach, which is much simpler than Ichiyoshi and Morita's collector; it simply synchronizes all threads at a collection and all threads 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.

Our collector algorithm is most similar to Imai and Tick's parallel copying collector [35]. In their study, all threads perform copying tasks cooperatively and any memory object in one shared heap can be copied by any processor. Their algorithm is a parallel extension to Cheney's breadth first copying algorithm [13]. 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 lock acquisition, and memory access costs. As we will show in Section 3.5, such factors become quite significant, especially in large-scale and multiprogrammed environments.

The collector algorithm described in this chapter is an improved version of the algorithm that the author and colleagues have constructed before [22]. After that, some researchers have adopted Stop-Parallel GC approach. Recent version of the conservative GC library by Boehm (version 6.0 and later [8]) supports parallel allocation and GC. However, their focus seems to be compatibility with previous version; its scalability is not sufficient on large scale machines.

Flood et al. proposed two parallel GC algorithm for multi-threaded Java virtual machine: a parallel copying algorithm and a parallel mark-compact algorithm. In both algorithm, each thread maintains its own task pool and performs dynamic load balancing via the pools. Their load balancing

method is based on a lock-free work stealing algorithm by Arora et al [3] [1]. They have shown parallel copying algorithm achieves 4–5.5 fold speed-up on 8-processor Enterprise 3500, and parallel mark-compact yields 2.2–5.5 fold speed-up. They use a status bitmap for termination detection that contains all threads' status. When a certain thread becomes idle, it changes its corresponding bit. However, as we will describe in Section 3.5, termination detection method that uses a shared word sometimes causes bottleneck on larger machines.

3.3 Overview of Parallel GC

Our stop-parallel GC implementation is based on the Boehm-Demers-Weiser conservative GC library (Boehm GC), which is a mark-sweep GC library for C and C++. Although some aspects described in this section come from Boehm GC, their implementation that we utilized is not parallelized [2]. 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.

Our collector supports parallel programs that consist of several kernel level threads, such as pthreads or Solaris threads. Although our collector does not limit the number of threads, we can obtain the best performance when each application thread is bound to distinct processor.

The memory allocation module adopts Big-Bag-of-Pages (BIBOP) method as described in Chapter 2. It manages a heap in units of pages with fixed size, and objects in a single page must have the same size. Free regions in the heap are maintained by two level free lists: free object lists and free page list. To support parallel allocation, the free lists are thread local. When an application thread requests a memory region, the allocator examines the free lists. If it fails, it tries to expand the heap, or start garbage collection.

When GC is invoked, all application threads are suspended by sending signals to them. When all the signals have been delivered, every thread starts GC task. To be exact, our collector adopts mark-lazy-sweep algorithm

[1] We currently plan to integrate this algorithm into our implementation

[2] Recent version of Boehm GC supports parallel allocation and GC.

rather than naive mark-sweep algorithm; threads perform mark phase while the world is stopped, and then restarts the world immediately after the mark phase. Sweep phase, in which unmarked memory objects are pushed onto free lists, is done lazily and incrementally as application program proceeds and requests more memory objects. To support lazy sweep phase, all contents of mark bits must be preserved until sweeping is done [3].

3.4 Parallel Marking Algorithm

When GC is invoked, several threads starts mark phase. All objects that are reachable from root sets (registers, execution stacks, and global variables) are marked recursively. This section describes parallel algorithm of mark phase. First, it describes data structure, basic algorithm and load balancing. And then we show some optimization to improve scalability.

3.4.1 Data Structure

mark bitmaps For each page, separate header record is allocated that contains information about the page, such as the size of the objects in it. Also kept in the header is a *mark bitmap* for the objects in the page. 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 page, which may contain multiple small objects. Therefore, in parallel GC algorithms, visiting and marking an object must explicitly be done atomically. Otherwise, if two threads simultaneously mark objects that share a common word in a mark bitmap, either of them may not be marked properly.

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). However, this increases memory consumption for lock objects. Thus instead of using locks, we adopt "test-and-compare&swap" sequence; we first read the mark bit without lock

[3] Boehm GC supports lazy sweeping partially. While it reconstructs free object lists lazily, it constructs the free page list eagerly while the world is stopped. We have found the eager page level sweeping sometimes prolongs the pause time significantly, and have made sweep phase completely lazy.

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. We retry if the location has been overwritten by another thread. Almost all modern processors, such as Ultra-SPARC and Pentium series support this atomic compare&swap instruction. Although MIPS architecture support compare&swap, it can be replaced with a load-linked/store-conditional pair.

Fortunately, this algorithm is a non-blocking algorithm [31, 45, 46], and hence does not suffer from untimely preemption.

mark stack The collector maintains marking tasks to be performed with an array 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 3.2 shows the marking process in pseudo code; each iteration pops an entry from the mark stack and scans the specified object, possibly pushing new entries onto the mark stack. A mark phase finishes when the mark stack becomes empty.

As the pseudo code shows, all threads involved in GC heavily access the mark stack. Thus each thread should maintain its own mark stack, rather than shared mark stack to avoid bottleneck. This decision arises another problem: load imbalance. Below we describe the dynamic load balancing algorithm to alleviate the problem.

3.4.2 Dynamic Load Balancing

In the parallel marking algorithm, each thread has its own local mark stack. When GC starts and all application threads are suspends, each threads starts marking from its local root, pushing objects onto its local mark stack. When all reachable objects are marked, the mark phase is finished.

```
push all roots onto mark stack.
while (mark stack is not empty) {
   {o, s} := pop(mark stack) ; object and its size
   for (i = 0; i < s; i++) {
      c := o[i]
      if (c is not a pointer) do nothing
      else {
         addr := address that contains mark bit of c
         mask := bitmask of mark bit of c
      retry:
         word := *addr
         if ((word & mask) != 0) do nothing ; already marked
         else {
            new-word := word | mask
            compare&swap(addr, word, new-word) ; atomic mark
            if (failed) goto retry
            push({c, size of c}, mark stack)
         }
      }
   }
}
```

Figure 3.2: The marking process with mark bitmap and mark stack. Marking is done atomically. The code relevant to dynamic load balancing is omitted.

Note that any application threads can not be resumed until all reachable objects are marked. If some application threads are resumed while other threads are still marking, the coherence problem occurs; some objects that are reachable from roots may be kept unmarked in unfortunate cases, due to unexpected object updates by application. To avoid the leak, we require write barrier, which informs any object updates to GC, as incremental / concurrent collectors do. In this chapter, we prohibit concurrent running of application threads and marking threads. Instead, we focus on shortening the stop-mark phase by dynamic load balancing.

As we will show in Section 3.5, without dynamic load balancing, the parallel marking hardly results in any recognizable speed-up because of the imbalance of marking tasks among threads. Load imbalance is significant when a large data structure is shared among threads through a small number of externally visible objects. For example, a significant imbalance is observed when a large tree is shared among threads only through a root object. In this case, once the root node of the tree is marked by one thread, so are all the internal nodes (Figure 3.3). To improve marking performance, our collector performs dynamic load balancing by exchanging entries stored in mark stacks.

Our implementation of dynamic load balancing is based on Lazy Task Creation method [42], which is a fine grained thread scheduling method. Usually each thread performs marking with its own mark stack; it pops a task (reference to an object to be scanned) from the top of the stack, and pushes its children onto the top of the stack if necessary. When a thread finds its stack empty, the thread becomes 'thief' and tries to steal a task from other mark stacks. When it founds a thread that holds tasks, the thief moves a task at the bottom of victim's stack to its own stack, and resumes marking.

Since several threads, including the owner and thieves access a mark stack, we require mutual exclusion. Generally, a mark stack is accessed most frequently by its owner, thus it is desirable that the owner can access the stack without lock acquisition. For this purpose, each stack maintains a pointer named *watermark* between top and bottom, as described in Figure 3.4. Tasks stored between top and watermark are 'private' to the owner of mark stack. The owner can pop private tasks freely. Tasks stored between

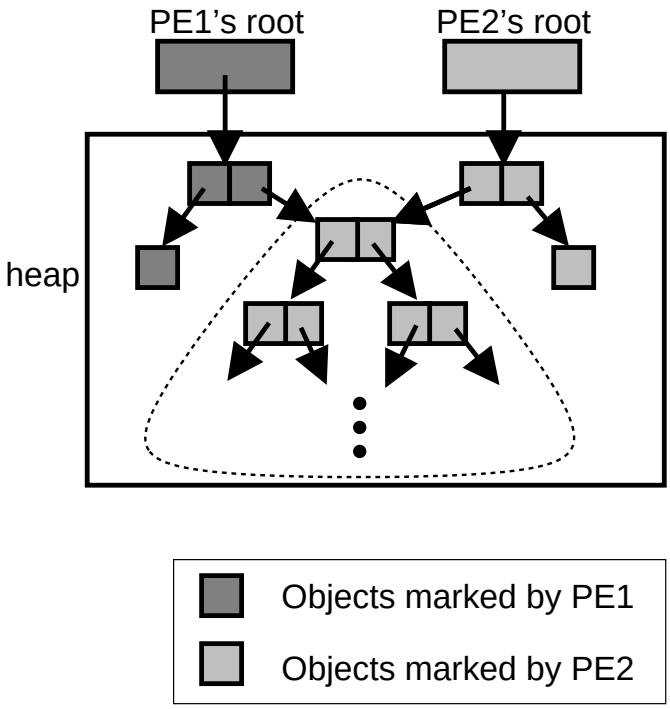

Figure 3.3: In the simple algorithm, all nodes of a shared tree are marked by one thread.

watermark and bottom are 'public' tasks, which thieves can steal. To arbitrate among thieves, lock acquisition is required. When owner thread finds either task group empty, it locks the stack and moves watermark. In current implementation, watermark is moved to the center between top and bottom.

3.4.3 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 3.5, however, several factors severely limit speed-up and this basic algorithm never yields good speed-up with more than 32 processors. 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 mark stack, which represents a single object in memory. This is still too large! We often found that only some threads were busy scanning large objects, while other threads were idle. This behavior is most prominent when applications use large matrices or large arrays. In one of our parallel applications (BH), a single 120-KB array to hold the particle data causes significant load imbalance. In the basic algorithm, it was not unusual for some threads to be idle during the entire second half of a mark phase.

We address this problem by splitting large objects (objects larger than 256 bytes) into smaller (than 256-byte) pieces. When a thread pops a large object from mark stack, it divides the object into halves and pushes the two pieces again, instead of scanning popped object. The thread recursively divides the object until it becomes sufficiently small. In the experiments described later, we refer to this optimization as SLO (Split Large Object).

Serialization in termination detection: When the number of threads becomes large, we found that the GC speed suddenly dropped. It revealed that threads spent a significant amount of time to acquire a lock on the global counter that maintains the number of idle threads.

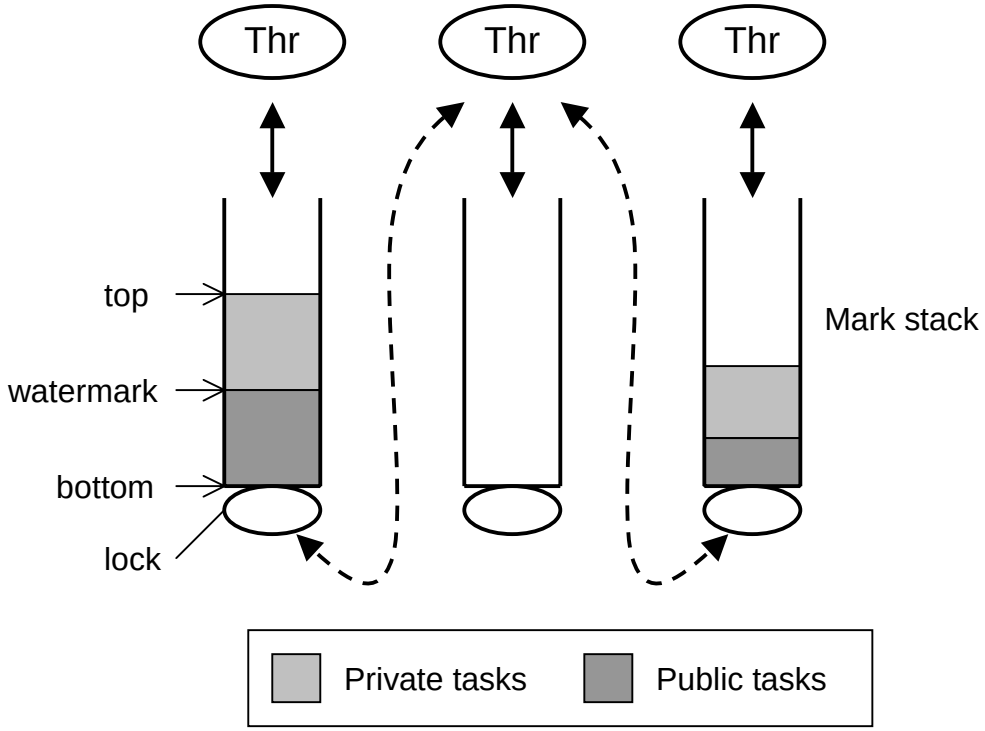

Figure 3.4: Dynamic load balancing method. An idle thread becomes a thief and try to the bottom of other mark stacks.

We updated this counter each time a stack became empty or tasks were thrown into an empty stack. This serialized update operation on the counter introduced a long critical path in the collector.

Note that this termination detection is a more difficult problem than the popular barrier synchronization. This is because threads can be resurrected after they become idle. The number of idle threads does not increase monotonously.

We implemented another termination detection method in which two flags are maintained by each thread; one tells whether the private tasks in mark stack of the thread is currently empty and the other tells whether the public tasks is currently empty. Since each thread maintains its own flags on locations different from those of the flags of other threads, 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 thread 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 thread. We must take care of the order of updating flags lest termination be detected by mistake. For example, when thread A steals all tasks of thread B, we need to change flags in the following order: (1) public-empty flag of A is cleared, (2) detection-interrupted flag is set, and (3) private-empty flag of B is set. We refer to this optimization as NSB (Non-Serializing Barrier).

Access contention at memory node on DSM: In some parallel application, only a single thread (or a few threads) allocates almost all memory objects in the heap. Typically, the allocator thread initializes the objects and gives them to other threads. On distributed shared memory architecture (DSM) architecture that adopts 'first-touch' memory placement policy, such application causes imbalanced memory placement; almost all memory objects are placed in a single physical memory node.

Generally, since GC heavily accesses memory, memory placement policy affects GC performance. Especially, it accesses allocated objects and mark bitmaps aggressively. As will show in Section 3.5, we have seen memory access contention at physical memory node sometimes limit GC scalability. Although changing placement policy can improve GC performance, we should not change policy of allocated objects, because it may go against users' intention and degrade application performance. On the other hand, moving mark bitmaps does not cause any effects on application.

In basic implementation, the allocator thread of a certain page sets up the corresponding mark bitmap. Thus when placement of objects is imbalanced, so are placement of mark bitmaps. We have implemented another placement policy of mark bitmaps, where they are distributed among all physical memory nodes. We refer to this optimization as BMB (Balanced Mark Bitmaps).

3.5 Experimental Results

3.5.1 Experimental Conditions

We have measured performance of the collector on two systems: the Ultra Enterprise 10000 SMP and the Origin 2000 DSM.

In the measurement, we bind each application thread to distinct processor. This section uses the words "thread" and "processor" in the same sense.

To measure GC performance, we used following parallel application programs described in Section 1.1.

BH (N-Body solver): We use both versions of BH: BH-st (sequential tree construction), and BH-pt (parallel tree construction). In BH-st, only a single thread performs tree construction, thus many objects are placed on a single memory node on DSM machines. In BH-pt, the placement of memory objects is almost balanced on DSM machines because all threads allocate objects. In the experiment, we simulate 30000 particles for 20 time steps.

CKY (CFG parser): In the experiment, we parse the given 200 sentences that consists of 36 to 100 words.

Cube (Rubik's cube solver): At each iteration, we traverse three steps from the given state in breadth first, and select the best twenty states for next iteration. We repeat the iteration for ten times. Because all threads allocate the state records, living objects are fairly distributed on DSM.

Table 3.1 shows the behavior of applications, such as the amount of living objects and execution time. The table shows the maximum living objects among several GC invocations.

3.5.2 Effect on Application Running Time

Figures 3.5-3.6 show how highly-optimized parallel collector affects on total application running time. The graphs show application performance with two versions of collectors. "Opt" refers to the fully optimized version, and "Simple" refers to the naive collector without dynamic load balancing. The graphs show breakdown of running time. "GC" indicates total time application thread is stopped for GC. "User" refers to running time except "GC". Memory allocation time and lazy sweeping time are included in "User". "GC" includes not only parallel mark phase, but synchronization costs after signal has been sent, and clearing mark bitmaps, and so on.

When we used all or almost all the processors on the machine, we occasionally observed GC invocations that performed distinguishably worse than the usual ones. They were typically 10 times worse than the usual ones. 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 used a little less processors; at most sixty processors on Enterprise 10000, and sixty-four processors on Origin 2000.

With "Simple" collector, scalability of application is limited, especially in CKY. "GC" time amounts to 30 percent of total running time on 60 processors on Enterprise 10000, while it amounts to only 9 percent with "Opt" collector. On Origin 2000, the ratio is 35 percent with "Simple" collector and 6 percent with "Opt" collector. Unfortunately, however, CKY

application itself is less scalable than other applications; its speed degrades on more than 16 processors even with "Opt".

In Cube application, we can also better performance with "Opt" collector. The difference gets larger as the number of processors becomes larger. On 64 processors on Enterprise, GC time accounts for 4 percent of running time with "Opt", while it accounts for 17 percent with "Simple".

The effect of GC is less significant in BH application, because the calculation phase, which never allocates memory objects dominates application running time. As for BH-st (sequential tree construction) version, GC time accounts for 1.9 percent of running time with "Opt" on 60 processors on Origin. The ratio of GC time is 4.8 percent with "Simple". In BH-pt (parallel tree construction) version, the ratio of GC time is relatively larger because application itself is faster. GC time accounts for 3.6 percent with "Opt" and 8.5 percent with "Simple". On Enterprise, we can see that "User" time with "Simple" collector is 10–30 percent longer than that with "Opt" collector. The reason for this phenomenon is currently not understood.

3.5.3 Speed-up of GC

This section shows performance of parallel mark phase on two systems. We define the speed of mark phase as the ratio of the total amount of reachable objects to the time of parallel mark phase. We discuss the average marking speed among all invocations during application execution.

Figures 3.7-3.8 show speed-up of mark phase. We measured several versions of collectors. "Simple" refers to the algorithm without dynamic load balancing. "No-Opt" supports dynamic load balancing, but it implements no optimization. "Opt" is the fully optimized version, described in Section 3.4.3. The BMB (Balanced Mark Bitmaps) optimization, however, is implemented only on Origin 2000 DSM. We let the speed of "Opt" collector on single processor be 1.

The graphs show that Simple does not exhibit any recognizable speed-up in any application. As Figure 3.7 show, No-Opt on Enterprise 10000 performs reasonably until a certain point, but it does not scale any more beyond it. The exception is Cube, where we do not see the difference between No-Opt and Opt. The saturation point of No-Opt depends on the application; Basic of CKY reaches the peak on 32 processors, while that of

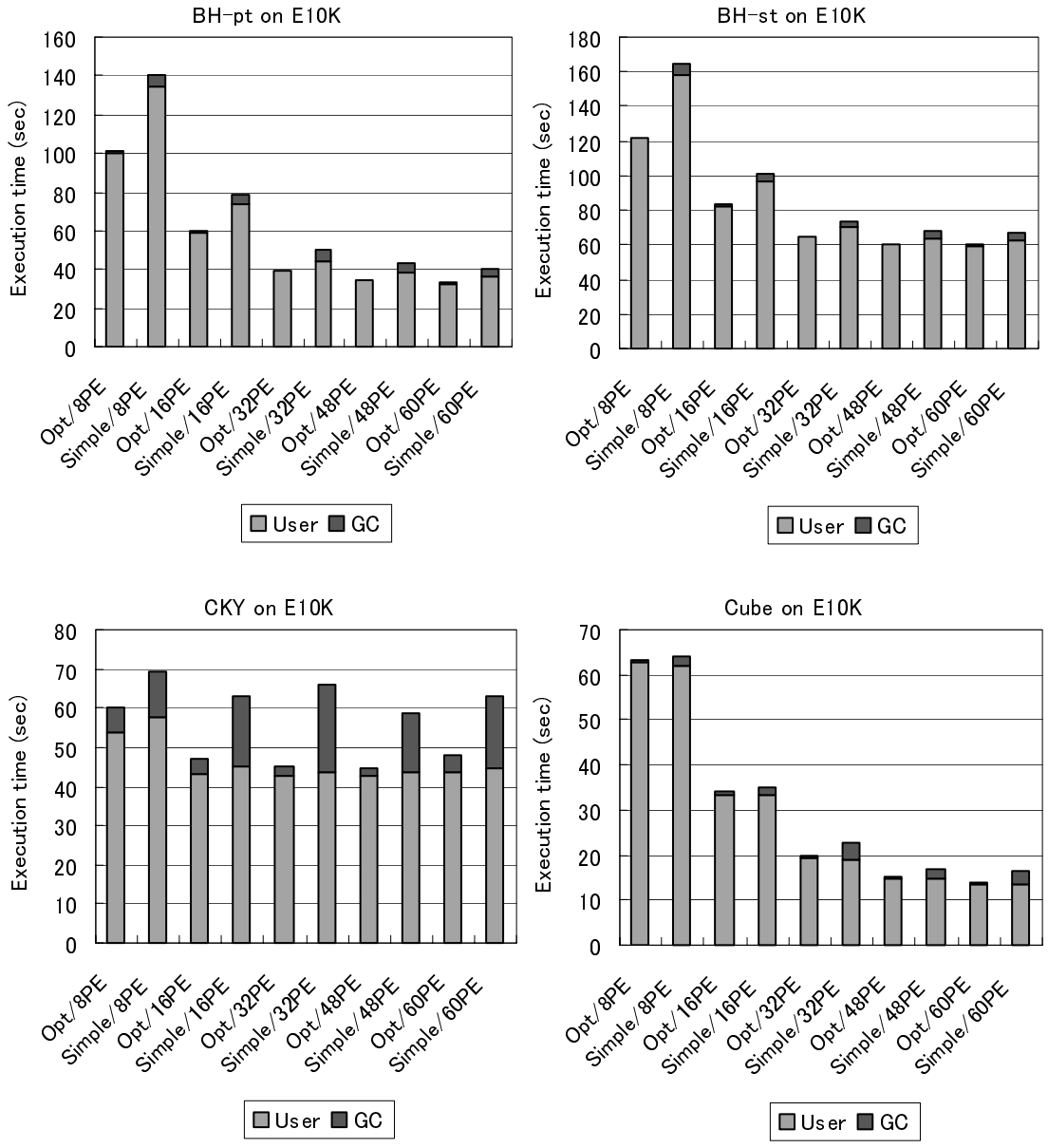

Figure 3.5: Application running time on Enterprise 10000.

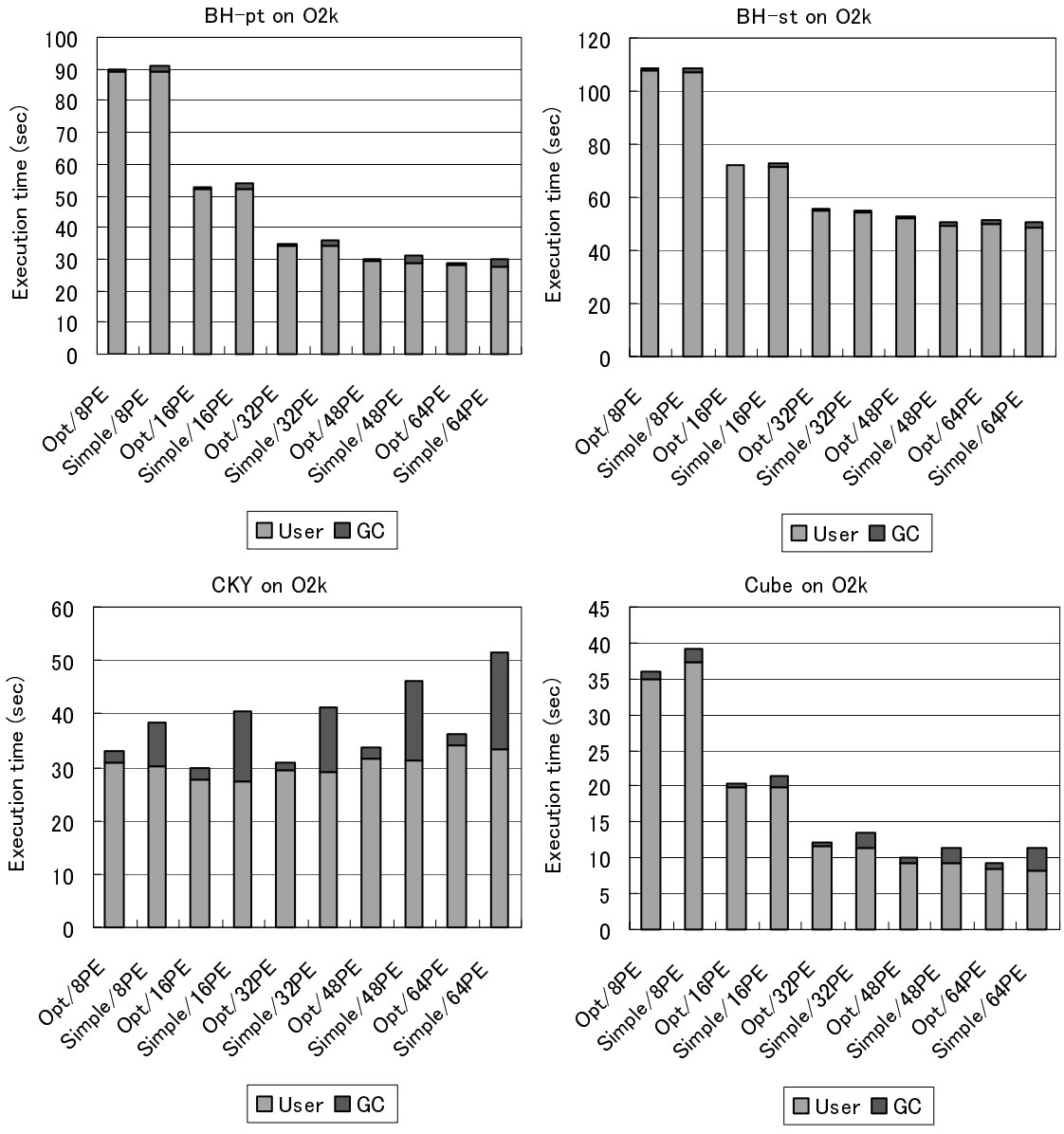

Figure 3.6: Application running time on Origin 2000.

BH-st reaches the saturation point on 8 processors. Opt achieved about 26 to 32 fold speed-up in BH and in CKY, and about 19-fold speed-up in Cube on 60 processors.

On Origin 2000, the GC speed-up is not so good as on Enterprise. Even with Opt, the GC scalability is limited except in CKY, where Opt achieves 20 fold speed-up on 64 processors. However, the difference between Opt and No-Opt is still important. In all application programs, Opt is 2–3 times faster than No-Opt. Unlike on Enterprise, we can see the significant performance gap between in BH-st and in BH-pt. Opt achieves 7-fold speed-up in BH-st, and 13-fold speed-up in BH-pt. We consider this gap comes from the difference in placement of memory objects. In BH-st, GC comes across more access contention than in BH-pt. The effects of placement of mark bitmaps is discussed in the following section.

3.5.4 Effect of Each Optimization

Figures 3.9–3.10 show how each optimization affects GC scalability. "No-XXX" stands for a collector that implements all the optimizations but XXX.

Especially in BH, removing the non-serializing barrier (NSB) optimization yields a sizable degradation in performance when we have a large number of processors. Without NSB, BH-pt cannot achieve more than a 14-fold speed-up. In CKY and Cube, interestingly, NSB has considerable effects only on Origin 2000.

The splitting large objects (SLO) is important when we have a large object in the application. In BH, we use a single array to to hold all particles data. In CKY, we use a large matrix to hold the result of parsing all subsentences. These large objects became a bottleneck when we omitted SLO optimization.

As we have expected, the balanced mark bitmaps (BMB) is important in BH-st, where distribution of memory objects is imbalanced. Without BMB, the collector shows only 4-fold speed-up on 64 processors. Fair distribution of mark bitmaps improve GC performance up to 7-fold speed-up. In other application, BMB has less effects on performance. We consider this is because memory objects are allocated by several threads and distributed among physical memory nodes.

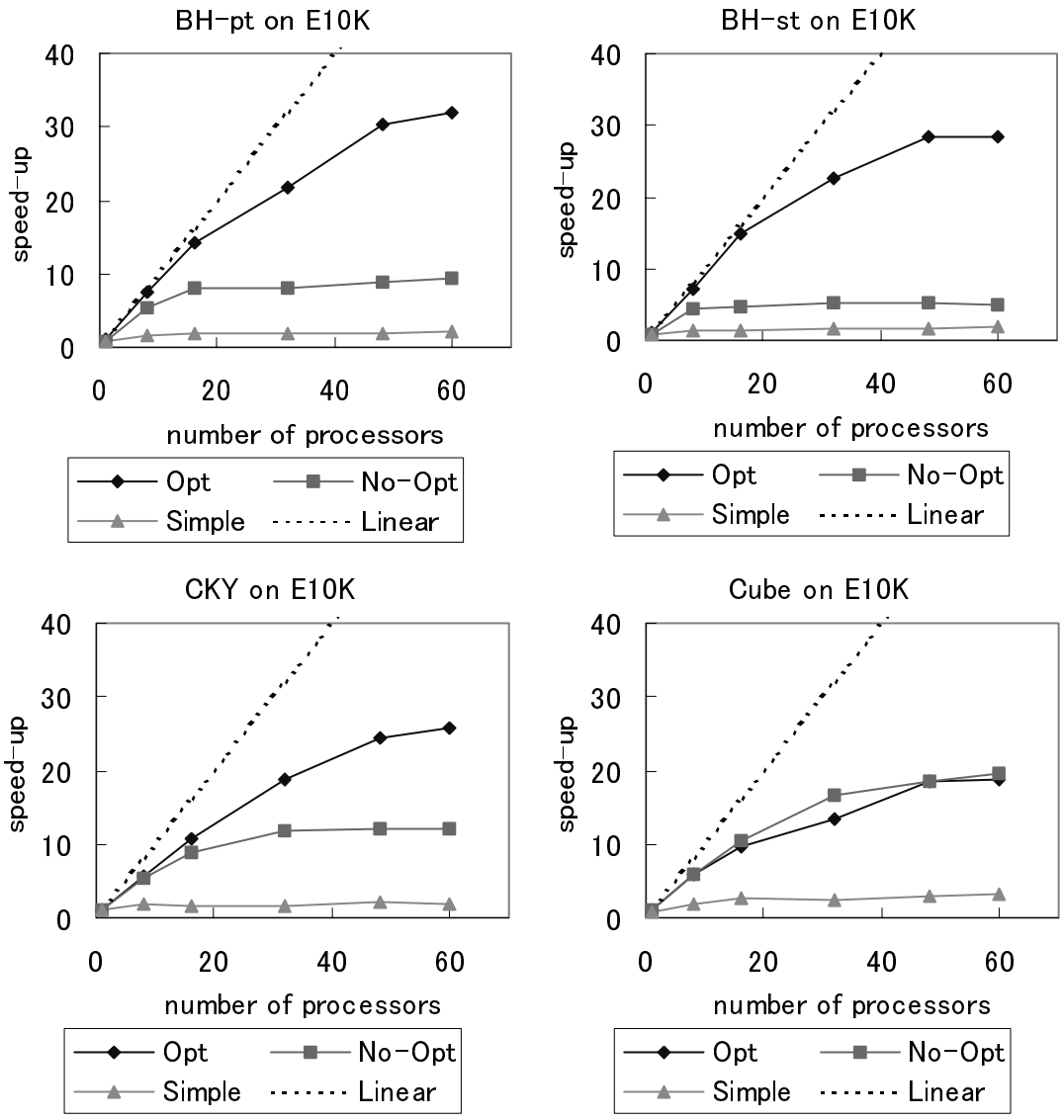

Figure 3.7: Average GC speed-up on Enterprise 10000.

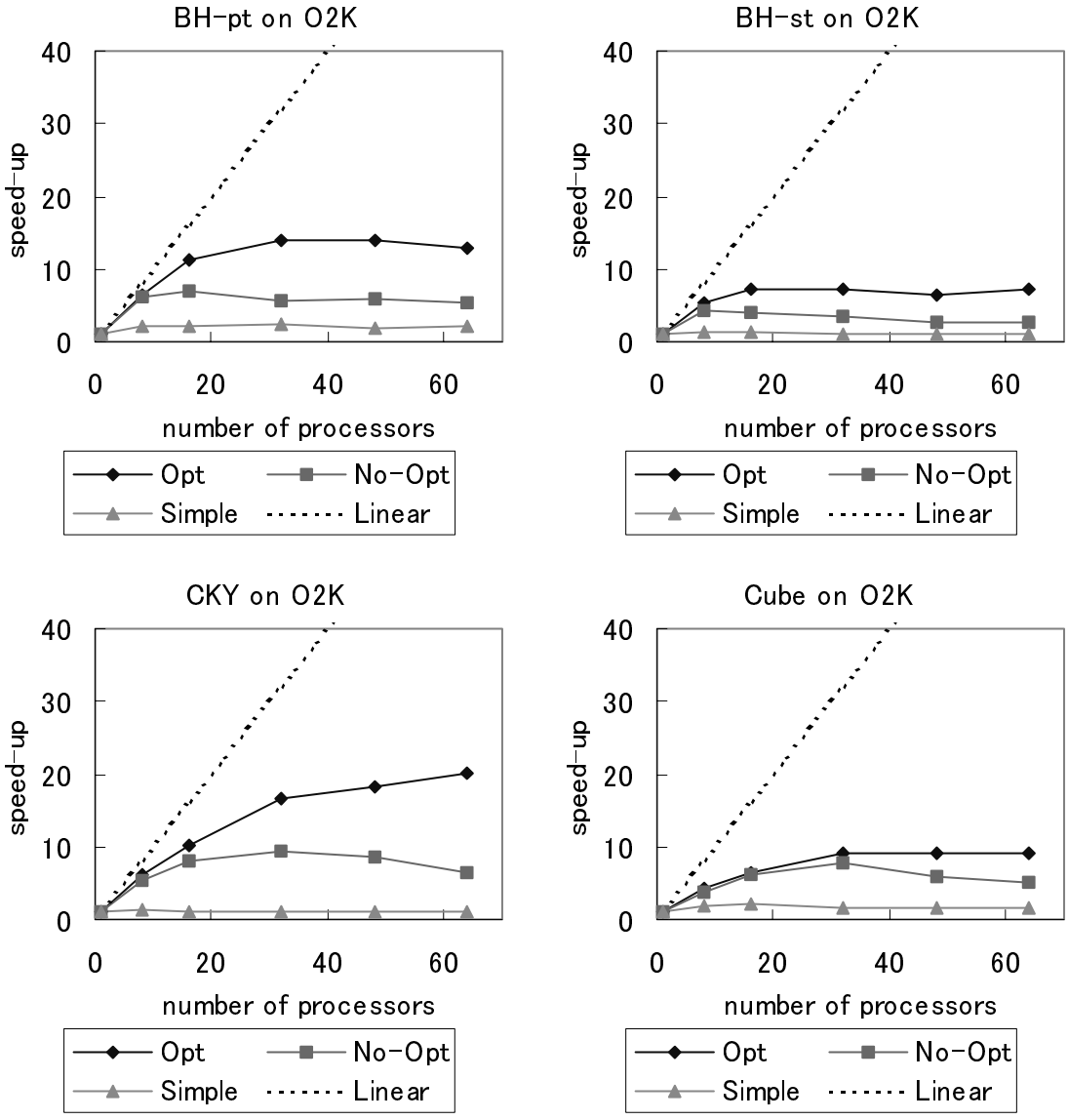

Figure 3.8: Average GC speed-up on Origin 2000.

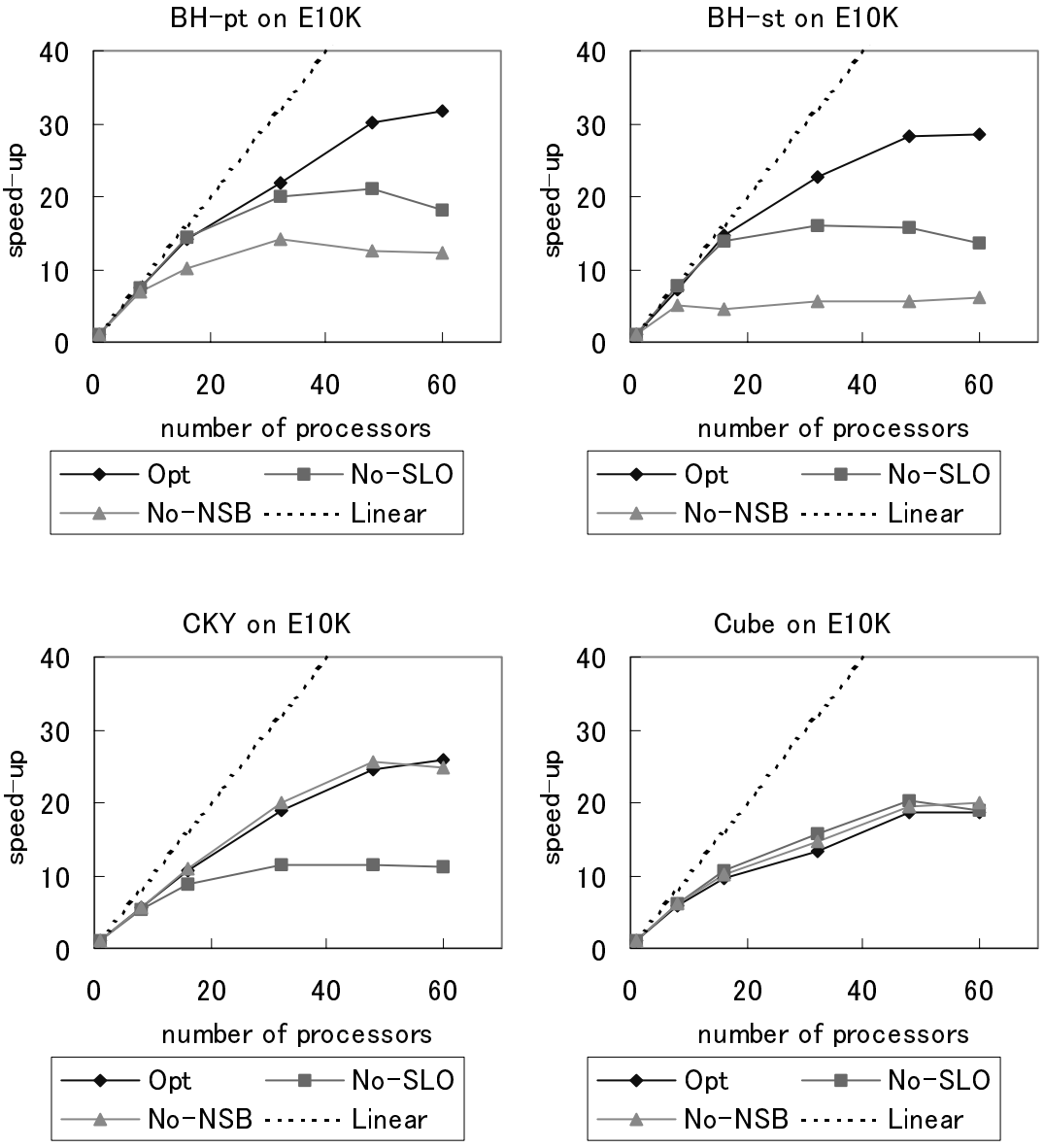

Figure 3.9: Effect of each optimization on Enterprise 10000.

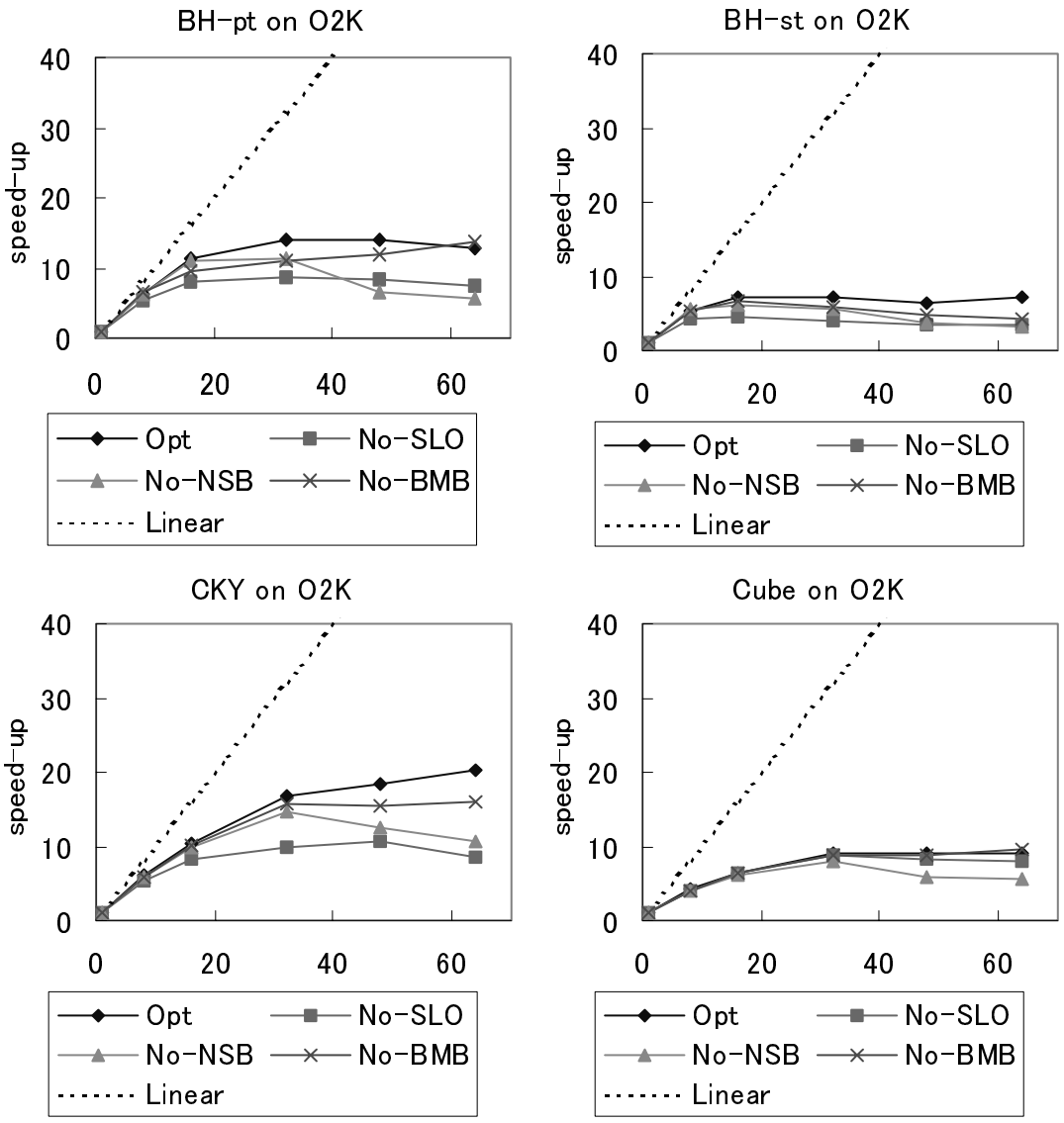

Figure 3.10: Effect of each optimization on Origin 2000.

3.6 Summary

We constructed a highly scalable parallel mark-sweep garbage collector for shared-memory machines. Implementation and evaluation are done on two systems: Sun Ultra Enterprise 10000, a symmetric multiprocessor machine with 64 processors and Origin 2000, a distributed shared memory machine with 80 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) 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. (3) 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 fairly among physical memory nodes. We conjecture that this is because the basic policy causes imbalance of access traffic among 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. We will discuss this problem in Chapter 4.

When using all these solutions, we achieved 19 to 32-fold speed-up with 60 processors on Enterprise 10000, and 7 to 20-fold speed-up with 64 processors on Origin 2000.

Application	# of GC	Exec. time (sec)	Max. living objects (MB)	Heap size(MB)
BH-pt	20	33.4	34.9	50
BH-st	18	63.3	29.4	50
CKY	38	47.8	33.7	50
Cube	15	15.29	17.2	30

Table 3.1: The number of GC invocation, execution time, the maximum amount of living objects, and the heap size of applications. This table shows statistics on Enterprise 10000 with 60 threads. Fully optimized collector is used.

Simple	Parallelized but no load balancing is done.
No-Opt	Load balancing is done, but no optimization is done.
No-SLO	All optimizations but SLO (splitting large object) are done.
No-NSB	All optimizations but NSB (non-serializing barrier) are done.
No-BMB	Origin 2000 only. All optimizations but BMB (balanced mark bitmaps) are done.
Opt	All optimizations are done.

Table 3.2: Description of labels in following graphs.

Chapter 4

Predicting Scalability of GC

This chapter describes a performance prediction model of parallel mark-sweep garbage collectors (GC) on shared memory multiprocessors. The prediction model takes the heap snapshot and memory access cost parameters (latency and occupancy) as inputs, and outputs performance of the parallel marking on any given number of processors. It takes several factors that affects performance into account: cache misses costs, memory access contention, and increase of misses by parallelization. We evaluate this model by comparing the predicted GC performance and measured performance on two architecturally different shared memory machines: Ultra Enterprise 10000 (crossbar connected SMP) and Origin 2000 (hypercube connected DSM). Our model accurately predicts qualitatively different speedups on the two machines that occurred in one application, which turn out to be due to contentions on a memory node. In addition to performance analysis, applications of the proposed model include adaptive GC algorithm to achieve optimal performance based on the prediction. This chapter shows the effect of automatic regulation of GC parallelism.

4.1 Introduction

The performance of tracing garbage collectors (GC) such as mark-sweep GC and copying GC is heavily affected by the characteristic of memory architecture, because GC incurs a large number of memory accesses. Especially, the impact of memory performance is significant when several processors cooperatively perform GC work on parallel machines. In previous chapter, we have reported that the performance of such parallel GC is sometimes

severely limited on distributed shared memory (DSM) machine, while it achieves good scalability on symmetric multiprocessors (SMP) [22, 21].

There are many factors that affects parallel GC performance: memory access contention, task stealing, and so on. The goal of this chapter is to analyze the effect of each factor quantitatively. For this purpose, this chapter proposes a performance model of parallel GC. The predictor takes a heap snapshot at GC starting time as input and architecture parameter, and outputs the running time of the mark phase on any given number of processors. We evaluate the validity of this model by comparing the predicted performance and the real performance obtained through experiments on parallel machines. The experiments are done on two shared memory machines: the Sun Enterprise 10000 (crossbar connected SMP) and the SGI Origin 2000 (hypercube connected DSM).

Applications of our work include construction of an adaptive GC algorithm, which regulates itself to achieve the best performance. For example, GC will be able to regulate the number of processors that are devoted to collection, by using the predicted result.

Section 4.2 shows our parallel GC, which is the target of prediction. Section 4.3 describes our prediction method. Section 4.4 compares the predicted performance and the real performance, and Section 4.5 mentions related work.

4.2 Parallel Mark-Sweep Garbage Collector

We focus on the stop parallel mark-sweep GC for shared memory machines, which we have described in Chapter 3 and [22]. Our GC is a parallel extension to Boehm-Demers-Weiser conservative GC library [11]. When any thread detects memory shortage, it suspends all application threads, and then all threads cooperatively perform marking and sweeping. They perform dynamic load balancing to achieve scalability.

Threads traverse the graph of all live objects in the heap with the lazy task creation (LTC) strategy [42]; each thread traverses objects in depth-first, by using its own task pool, called mark stack. When a thread finds its mark stack empty, it tries to steal a task from other mark stacks. If the attempt is successful, it steals one task from the bottom of the target stack,

and restarts marking. The mark phase terminates when all stacks become empty.

Like previous chapters, this chapter assumes that each thread is bound to distinct processor. We use the words "thread" and "processor" in the same sense.

4.3 Prediction Method

4.3.1 Overview

Our predictor takes a heap snapshot at GC starting point as input, and shows the predicted running time of mark phase with P processors. Figure 4.1 shows the overview of our method.

1. We collect some information about GC workload and memory access pattern by inspecting the heap snapshot (Section 4.3.4).

2. We estimate parallel running time T_P that excludes cache miss costs from T_1 and T_∞ (Section 4.3.5).

3. We estimate the number of cache misses on parallel execution Q_P. This may be larger than that on serial execution Q_1, because of task stealing. Q_P is estimated through analysis of live cache lines (Section 4.3.6).

4. We calculate the cache miss cost on parallel execution M_P by using Mean Value Analysis. Generally M_P is larger than the miss cost on serial execution M_1, because of access contention (Section 4.3.7).

5. Finally, we obtain the overall running time T_P^M as $T_P^M = T_P + Q_P M_P / P$.

4.3.2 Assumption

We make the following assumptions to simplify the model. We believe the effects of them on typical heap snapshots is small.

- On DSM, we assume that a certain memory region is accessed by any processor at same probability. In other words, we assume the task stealing scheduler is oblivious to locality.

- We ignore the costs of cache invalidation. We assume that an access request from processors is sent to the home memory node, and returns directly to the source processor.
- We ignore the overlap between memory access latency and other computational instructions.

4.3.3 Architecture Parameters

This section describes the basic memory access costs of parallel machines. The access costs on these machines are shown in Table 4.1. In the table, M_1 corresponds to the round-trip time of memory access request without contention costs, and S_O is the occupancy time of the receiver memory node by each request. We have determined them through benchmark tests that aggressively access memory.

4.3.4 Heap Inspection

We obtain parameters that represent workload and memory access pattern by inspecting heap snapshot.

First, we obtain the total computation work T_1 and the depth of live object graph T_∞ from the living object graph. Intuitively, T_1 is the serial running time and T_∞ is the minimum running time with an infinite number of processors. Both T_1 and T_∞ exclude cache misses costs.

Next, we keep track of memory access pattern by simulating the serial mark phase, and feed the pattern to the cache simulator. Thus we obtain the number of serial cache misses Q_1. We also calculate the number of average living cache lines L during mark phase, which is used to estimate the number of misses on parallel execution, as described in Section 4.3.6. Besides, on DSM, we obtain the distribution of target of memory accesses V_j. We let V_j be the ratio of access requests to the j th memory node, to all requests in the machine. This is used for estimation of access contention costs.

Finally, we estimate the total number of task stealing N_S in parallel execution with P processors. It is difficult to know a precise value of N_S beforehand, because of nondeterminism. Our predictor adopts a rough esti-

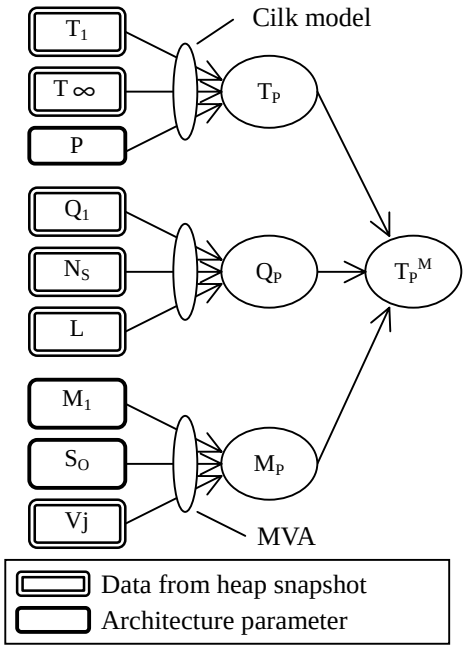

Figure 4.1: Overview of our performance prediction method. T_P^M is the final result; the marking time with access costs and contention costs.

O2K

access type	local M_1(ns)	remote M_1(ns)	S_O(ns)
read	270	> 590	230
RW	850	> 1400	490

E10K

access type	M_1(ns)	$S_O(ns)$
read	560	250
RW	610	420

Table 4.1: Memory access cost on O2K and E10K, obtained from benchmark tests. "RW" stands for atomic read-modify-write access.

mate of N_S; $N_S = Plog(T_1)$ [1].

4.3.5 Performance Prediction without Miss Cost

In this section, we obtain T_P, which is predicted parallel marking time without costs of cache misses from T_1.

When the object graph has very large width, our GC can utilize all collector threads during mark phase. In such cases, parallel marking time would be close to T_1/P, when we ignore cache miss costs. On the other hand, the object graph is narrow, speedup of parallel marking is limited because of critical path. The parallel marking time cannot be less than T_∞. Therefore, we let T_P be $T_P = \min(T_1/P, T_\infty)$.

Until now, the Cilk performance model [6, 7, 29] has shown that the parallel running time is $T_P = O(T_1/P + T_\infty)$. Besides, Cilk group has shown that effective thread schedulers achieve $T_P = T_1/P + O(T_\infty)$.

As for our parallel application programs, described in Section 4.4, Cilk's estimation and ours give similar result. This is because the object graphs are wide enough in most cases, and T_∞ is much less than T_1/P.

4.3.6 Number of Cache Misses

We derive the number of cache misses on parallel execution Q_P from the number of serial cache misses Q_1. In LTC style execution, the computation order is preserved in most cases between serial execution and parallel execution. The exception is caused by task stealing; tasks which were contiguous in serial execution may be performed by different processors in parallel execution.

Figure 4.2 shows the behavior of cache lines during serial execution and parallel execution of the same workload. Black and gray circles correspond to cache misses, and white ones are cache hits. The circles arrayed horizontally stand for accesses to a single cache line. In serial execution, we have five cache misses in this case. In parallel execution, suppose two task steals (vertical lines in the figure) occur during this execution. Then three task groups separated by two vertical lines are executed by distinct processors,

[1] This estimation is justified only through experimentation. Better estimation is under investigation.

which have respective cache memories. Thus some memory accesses are no more contiguous and the total number of cache misses is larger than Q_1.

The predictor estimates Q_P as $Q_P = Q_1 + N_S L$, where N_S is the total number of task steals and L is the average number of live cache lines. Intuitively, we assume each task steal incurs about L additional cache misses.

4.3.7 Cost of Cache Misses

This section estimates the cache miss costs on parallel execution M_P, from sequential access costs M_1. We utilize Mean Value Analysis to account for access contention. Current model considers only contention at memory nodes, and ignores contention at other parts of the network. When a processor raises a cache miss, it sends a access request to home memory node and waits for the response (Figure 4.3). The miss cost M_P may be larger than M_1, because of contention. To estimates contention costs, we use the occupancy time S_o and the access distribution V_j, which may be unfair among memory nodes on DSM.

The frequency of incoming access requests to j th memory node is $V_j Q_P / T_P^M$, where T_P^M is overall running time. Thus the average waiting time of each access request at j th memory node is $S_O \rho / (1 - \rho)$, where $\rho = S_O V_j Q_P / T_P^M$. Here we obtain M_P as $M_P = M_1 + S_O \rho / (1 - \rho)$. Because M_P depends on the final result T_P^M, the definition of M_P is recursive. The predictor uses the Newton method to calculate it.

4.4 Experimental Results

This section compares the predicted performance of parallel mark phase by our model with the measured performance on parallel machines. We show the average speed of the mark phase of several GC invocations through the execution of parallel application programs. We use three parallel application programs: BH-st, Cube, CKY.

BH (N-body solver): In the experiment of this section, we use only BH-st version, where only a single thread performs tree construction. In BH-st, GC performance significantly differ between two parallel machines. We simulate 50000 particles for 10 time steps.

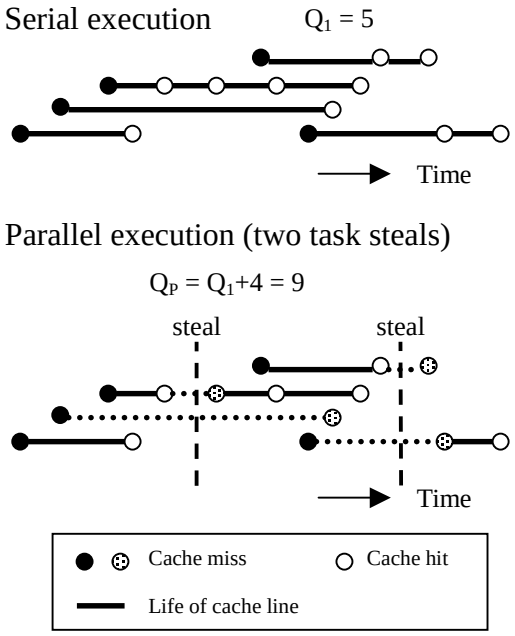

Figure 4.2: The behavior of cache lines, in serial execution and parallel execution.

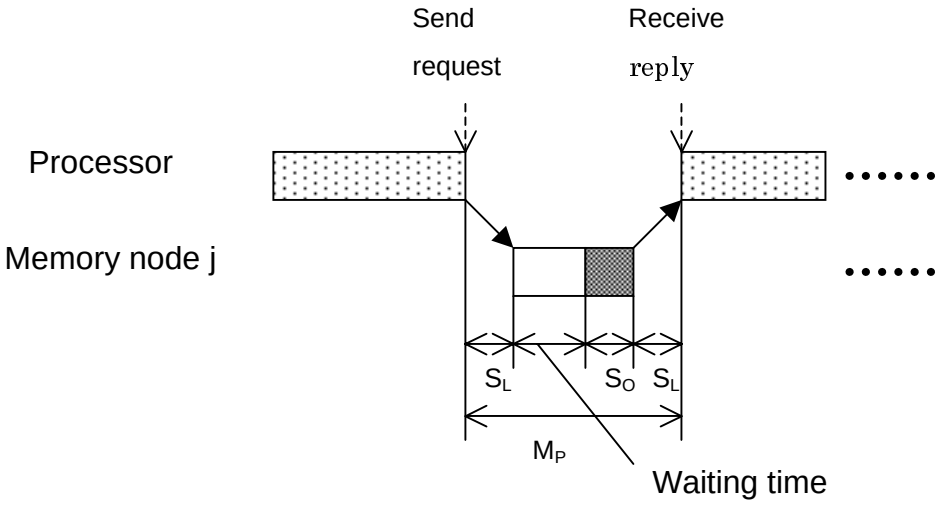

Figure 4.3: Behavior of a processor and a main memory node. Cache miss cost is at least $M_1 = 2S_L + S_O$, and gets longer if contention occurs.

Cube (Rubik's cube solver): At each iteration, we select the best ten states for next iteration. We repeat the iteration for ten times. Because processors allocate the state records in parallel, live objects are distributed in all memory nodes in Origin 2000.

CKY (CFG parser): In the experiment, we parse ten sentences that consists of 95 to 105 words. Like Cube, live objects are distributed in all memory nodes in Origin 2000.

4.4.1 Evaluation of Predicted Performance

Figure 4.4 and Table 4.2 compare the predicted result and the real performance on Origin 2000 and Enterprise 10000. The graphs show GC speed-up by parallelization; the upper graphs corresponds to the results on Origin 2000 (O2K), and the lower ones show the results on Enterprise 10000 (E10K). The horizontal axis of each graph corresponds to the number of processors and the vertical axis shows the speed-up, which is normalized to the real sequential speed. "Real" refers to the measured speed-up and "Pred" refers to the speed-up predicted by our predictor. "Pred($Q_P = Q_1$)" corresponds to another prediction that ignores miss increase by parallelization. "Pred($M_P = M_1$)" ignores access contention cost.

The mark phase in BH-st achieves good scalability on Enterprise 10000. The performance rises well with an increase of processors. On Origin 2000, however, it reaches the peak on 32 processors and does not scale any more. The predicted graph succeeds in capturing the difference between the two machines. The graph shows that there is a significant gap between "Pred($M_P = M_1$)" and "Pred" on Origin 2000; we can see that the access contention heavily degrades the performance. Without contention costs, the model can never predict behavior of the measured performance; this result justifies our model that takes contention costs into account.

In Cube on Origin 2000, the "Pred($Q_P = Q_1$)" is far from "Pred", thus we can see that Cube suffers from effects of miss increase by parallelization.

In CKY, the gap among "Pred", "Pred($Q_P = Q_1$)" and "Pred($M_P = M_1$)" is much less than in other application programs. This suggests that effects of contention costs and increase of misses on the performance is not very large.

Table 4.2 shows the error of predicted result with 48 processors. In all cases, the predictor tends to output faster speed than "Real"; the errors are 7 to 38%. This result suggests that there may be still some performance limiting factors that our model does not account for yet.

4.4.2 Overhead of Predictor

To utilize the predicted result for online optimization, the predictor itself must be fast enough. However, the predictor we have described is slow ("slow ver." in Table 4.3), because it tracks all memory accesses and feeds them to a cache simulator. We have made another predictor that is faster, but less accurate ("fast ver." in the table). The fast predictor takes the amount of live objects and V_j as input, rather than all memory access pattern. Therefore, it tends to underestimate the number of cache misses. We use this predictor for an adaptive GC algorithm in next section. Slow version is still be useful to analyze GC performance in detail.

4.4.3 Performance Prediction on Future Machines

By using our predictor, we can predict the performance on future machines. Consider two machines that have same memory architecture as O2000 and E10000, but have much more processors than real machines. Figure 4.5 shows the predicted performance on the imaginary machines. For the prediction, we used the heap snapshot of three application programs executed with 32 processors. The graphs show that the performance will reach a peak at 128 processors on O2000 in all application programs. We can see that main memory access becomes bottleneck in BH-st and Cube, even on E10000. Without accounting for contention costs, we cannot obtain these results.

4.4.4 Adaptive GC algorithm

One of the application of our predictor is construction of adaptive GC algorithms. As an example, this section describes automatic regulation of GC parallelism. For some applications such as BH-st on Origin 2000, it is meaningless to devote too many processors to GC. In such cases, we use less processors than the number specified by user. By reducing processors, other processes may gain profit in multiprogramming environment.

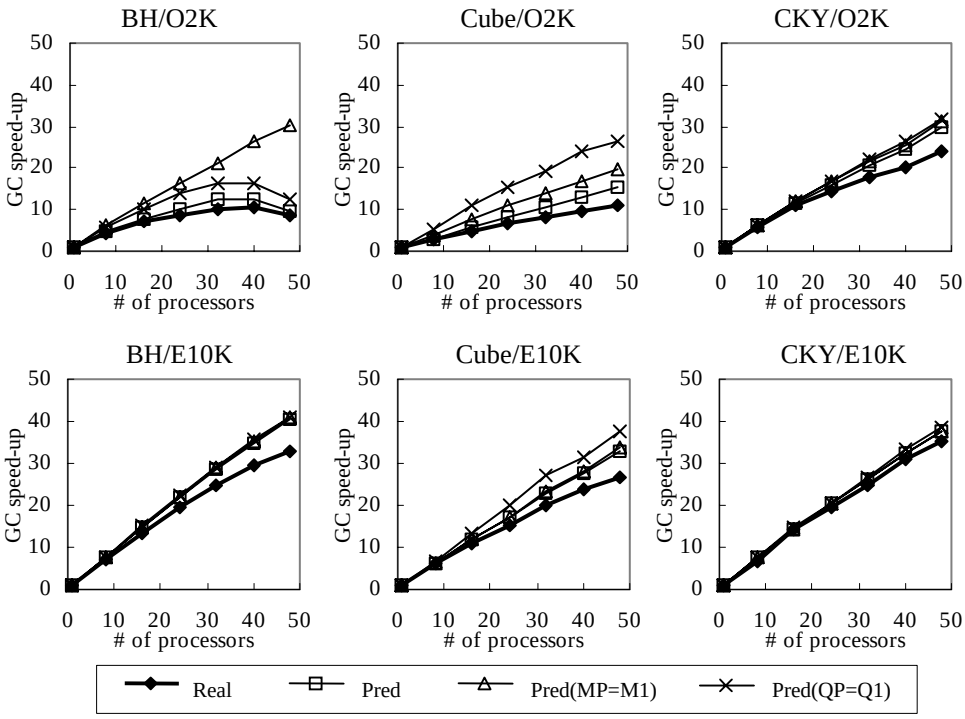

Figure 4.4: GC Speed-up. Graphs compare real speed-up("Real") and predicted speed-up("Pred").

application /machine	pred	pred ($M_P = M_1$)	pred ($Q_P = Q_1$)
BH-st/O2K	+15 %	+260 %	+49 %
Cube/O2K	+38 %	+77 %	+140 %
CKY/O2K	+24 %	+28 %	+31 %
BH-st/E10K	+22 %	+24 %	+24 %
Cube/E10K	+23 %	+26 %	+41 %
CKY/E10K	+6.8 %	+7.1 %	+9.4 %

Table 4.2: The difference between predicted performance and real performance with 48 processors.

predictor	overhead (1PE)	error (1PE)	error (48PE)
slow ver.	760 %	-2.4 %	+15 %
fast ver.	7.4 %	+23 %	+260 %

Table 4.3: Overhead and accuracy of two predictors, in BH-st/O2K. Overhead is the ratio of prediction time to running time of mark phase.

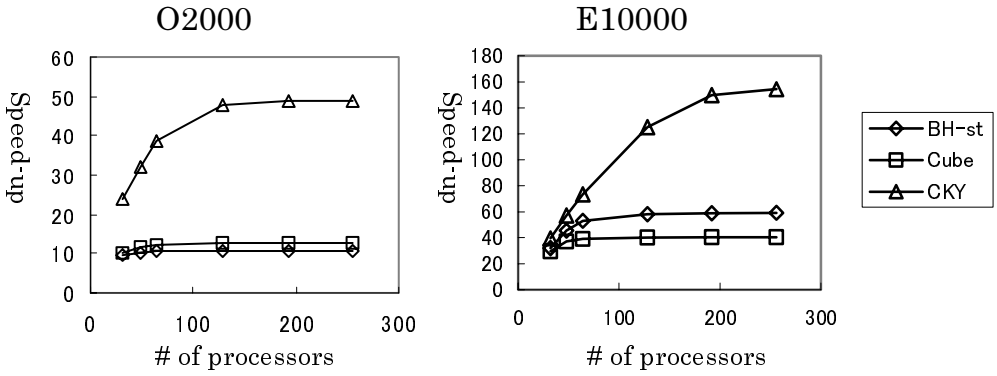

Figure 4.5: Predicted marking performance where the number of processors is larger than real machine. The graphs assume that the number of main memory nodes and memory performance are the same as real machine.

Table 4.4 shows the result of regulation of parallelism. In "full", GC always use all of specified processors. In "Adapt", GC uses the predicted result of the fast predictor to find sufficient number of processors. The bottom row of the table shows the average number of used processors in "Adapt". We can see that only 22 processors are sufficient to achieve almost same performance as that with 48 processors in this case.

4.5 Related Work

There are many pieces of work on performance analysis on parallel machines that mention the importance of communication costs. The major part of the researches focus on programs that have regular structures, on which it is easier to estimate communication costs than on irregular programs.

The Cilk performance model [6, 7, 29] estimates the parallel running time of both regular and irregular programs that are executed in LTC strategy. We utilize this model to estimate T_P, the parallel running time without cache misses cost. Though the earlier model does not account for the memory hierarchy, the recent model [29] by Frigo analyzes the costs of cache misses. However, current Frigo's model may underestimate cache miss costs, because it assumes that miss costs are always uniform. Through the experiments, we have found that we cannot apply that assumption for DSM such as Origin 2000, where software can control the location of memory pages. The contention of access requests to a certain node may heavily degrade

performance on such machines. Another factor that Frigo's model accounts for is the increase of cache misses by parallelization. However, it tends to overestimate the increase, because it estimates the number of cache misses as $Q_P = Q_1 + O(HN_S)$, where H is the number of all cache lines each processor has. In our model, $Q_P = Q_1 + LN_S$, where L is the number of live cache lines, which depends on application behavior.

The LogP model [17] analyzes the behavior of distributed parallel machines with four architectural parameters: the latency, the overhead of message processing, the bandwidth, and the number of processors. Several researches have used LogP model to study parallel programs. More recently, Frank et al. proposed the LoPC model [28], which is based on LogP, but accounts for contention costs by using Mean Value Analysis. Our estimation method of cache miss costs is strongly affected by the LoPC model.

4.6 Summary

This chapter proposed a performance prediction model for a parallel garbage collector on shared memory parallel machines. This model takes a heap snapshot at GC starting time as input, and estimates parallel running time of mark phase. This model takes contention costs of memory accesses into account, which are especially important on DSM. It also accounts for the increase of cache misses by task stealing in parallel execution.

We have compared the predicted GC performance with the measured performance through experiments on two parallel machines: Sun Enterprise 10000 SMP and SGI Origin 2000 DSM. The prediction error of parallel marking time with 48 processors is 7 to 38%. In BH-st application, which incurs unfair memory location, the GC scalability on Origin 2000 is much worse than that on Enterprise 10000. Without taking contention costs into account, the model can never explain behavior of the performance. As an example of applications of our model, we have shown the experimental result of automatic regulation of GC parallelism.

The future work includes improving the accuracy of prediction. In addition to accuracy, the predictor should be fast when we use it for online optimization. We also plan to apply our model for general parallel application programs. It would be interesting to investigate how we can manage

programs with more complex synchronization pattern and memory access pattern than GC.

# specified processors	8	16	32	48
GC speed-up / full	3.8	4.9	6.1	6.3
GC speed-up / adapt	3.8	4.5	5.8	6.0
# avg. used processors	8.0	12.4	19.9	21.8

Table 4.4: The result of automatic regulation of GC parallelism, in BH-st/O2K.

Chapter 5

Concurrent Parallel Garbage Collector

Chapter 3 has shown that a scalable stop parallel garbage collector can improve execution times of applications by reducing GC time. However, some applications such as server applications and GUI applications require short pause times in addition to good execution times. For this purpose, this chapter describes a concurrent parallel GC, where collector threads and application threads run in parallel, and collection itself is done by multiple threads. The experimental results of this collector on Enterprise 10000 and Origin 2000 are shown. This collector is based on the incremental update algorithm, on which the finalization phase determines the worst pause time. Compared with stop parallel GC, the advantage of this concurrent parallel GC on pause times gets smaller relatively as the number of threads increases, because stop parallel GC scales better. The most important element that determines application execution times with concurrent collectors is the implementation of a write barrier. This chapter compares two implementation: one uses virtual memory facility, and the other is implemented by code insertion. The former yields overhead of memory protection, which becomes larger as the number of threads increases. The latter also causes considerable overhead on execution times, which is 70 % in the worst case. However, its scalability is better, and gets faster than the former implementation on 48 threads and more. Potentially, the application execution times may be improved by using concurrent parallel collectors rather than stop parallel collectors, though such situations did not occur in the experimentation.

5.1 Introduction

The performance of application programs heavily depends on memory management modules. In previous chapters, we have focused on the scalability of GC to improve the execution times (or throughput) of applications. Moreover, some applications require real time collectors. For example, the response time of server applications and GUI applications depends on the pause time of collectors.

Many researchers have tackled on the GC pause time by using concurrent collection algorithm[20, 32, 43], on which a GC thread and application threads run in parallel. However, if garbage collection itself is not parallelized, the GC thread may fail to catch up with memory allocation requests from application threads, on large scale multiprocessors. To solve this scalability problem, this chapter describes a *concurrent parallel* collector, where collector threads and application threads run in parallel, and collection itself is done by multiple threads. Cheng et al. [14] have described algorithm and performance of a concurrent parallel collector for SML. Unlike theirs, this chapter describes the implementation that requires no compiler support.

Concurrent parallel collectors have a potential ability to improve not only pause time but application execution time, because they can make idle processors during application execution participate collection work. Moreover, they may be able to alleviate memory bus traffic especially on DSM. In spite of such advantages, they have the following disadvantages.

- It imposes a *write barrier* overhead on applications. The write barrier is required to guarantee that all reachable objects are retained even if applications update object graph during GC cycle.

- Generally, concurrent collectors are more 'conservative' than stop collectors; they reclaim less garbages because of floating garbages. Thus more collection tasks are required to proceed applications.

The effect of write barrier on execution times is important, particularly. For the experiments, we have implemented two versions of write barrier: one uses virtual memory facility, and the other is implemented by code insertion.

Our collector is based on the incremental mark sweep GC for C/C++ by Boehm et al [9], and parallelized in a similar manner described in Chap-

ter 3. The collector adopts *incremental update* method [19, 52] to achieve concurrency. This chapter evaluates the performance of our collector on Enterprise 10000 SMP and Origin 2000 DSM. We compare pause times and application execution times of the concurrent parallel collector and the stop parallel collector.

Section 5.2 describes the algorithm of concurrent parallel collector, and we evaluate the performance in Section 5.3. Section 5.4 discuss the relation between pause times and algorithm. Section 5.5 mentions previous work on concurrent collectors, and we summarize this chapter in Section 5.6.

5.2 Algorithm

Our concurrent parallel GC implementation has the following features:

- It uses Big bag of pages (BIBOP) allocator, though it is not an essential condition.
- It is based on incremental update method.
- It uses page level memory protection facility of the underlying OS [1].
- It treats new objects allocated during collection as 'unmarked'. This condition allows very short lived objects to be reclaimed.

Data structure our collector uses is similar to that we have described in Chapter 3. Each thread maintains its local mark stack, and marking is done by using mark bitmaps.

Since application programs run during collection, collection must start before the heap is exhausted. When page level allocator finds the number of empty pages is lower than a threshold (15% of the heap in the experimentation below), it triggers a collection. As Figure 5.1 shows, GC cycle consists of three phases: start phase, concurrent phase and finalize phase.

Start phase: The collector changes the access protections of all pages in the heap to 'Read-only' by using `mprotect` systemcall. It empties a set named *dirty page set*.

[1] We describe an implementation by software afterward.

Then it determines the GC speed k to prevent starvation. It determines $k = (M - F)/F$, where M is the number of all pages in the heap, and f is the number of empty pages [2].

It stops all application threads by sending a signal. The suspended threads push their own roots (stack, registers) onto their mark stack. Global variables are also pushed on one of mark stacks. Then all threads restarts application immediately.

concurrent phase: Each thread proceeds marking with its mark stack incrementally, when it allocates memory objects. Whenever the allocation count exceeds a page size (= P bytes), it examines memory regions more than fP bytes. In addition to allocator threads, idle threads can participate in the concurrent phase. Several threads can proceed marking simultaneously, and marking algorithm is similar to that of the stop parallel algorithm in Chapter 3; threads performs dynamic load balancing. When all mark stacks become empty, concurrent phase finishes.

finalize phase: The incremental update method requires finalize phase to prevent any reachable objects from being left unmarked. Unfortunately, the work amount of this process is unbound, though it is smaller than that of full garbage collection in many cases. When one of threads detects the termination of concurrent phase, it sends signals to all threads to suspend application again. The threads examines pages included in dirty page set and finds objects already marked, then push them onto mark stacks. Besides, they push roots onto mark stacks again. Then they start parallel marking cooperatively. We expect that almost reachable objects have been already marked, and this phase finishes early. After finishing the second traversing, the collector prepares for lazy sweeping and resumes application.

As any concurrent/incremental GC does, our collector needs to know accesses information to objects by application threads. Our collector uses write barrier, which informs the collector about object updates during GC

[2]Unfortunately, we cannot guarantee that collection always catch up with allocation requests, because of external fragmentation. When starvation occurs, the allocator thread is delayed until collection finishes.

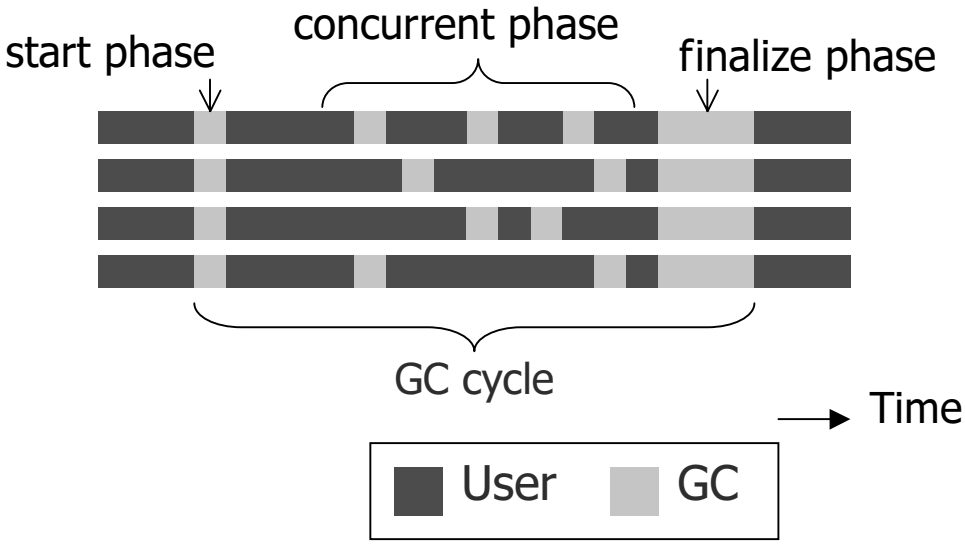

Figure 5.1: A GC cycle consists of start phase, concurrent phase, and finalize phase.

cycle. When application threads try to update objects that are protected at the beginning of GC cycle, an exception occurs and signal handler is invoked. The thread adds the target page into the dirty page set, changes its protection to 'Read-write', and resumes application. In the finalize phase, the collector can know all pages in the heap that are updated during GC cycle. Because some unmarked objects may be installed in marked objects, the finalize phase must rescue such unmarked objects.

5.2.1 Optimization

The collector changes protection of pages over and over. It protects all pages at the start phase, and unprotects pages that are updated by application. When the page level allocator allocates a new page, it also unprotects the page because it will be updated immediately. Changing memory protection imposes overhead on application execution time. As Appel et al. [1] has described, the overhead gets larger as the number of processors increases, in order to maintain TLB consistency. This section describes a technique to reduce the protection overhead.

We utilize the fact that changing protection of consecutive pages at one time incurs smaller overhead than operating them individually. When the

collector tries to unprotect a page because of updating or allocation, it examines the status of adjacent pages. If they are currently empty pages, they will be used by allocator in the near future. Therefore the collector unprotects those pages at one time beforehand. This technique reduces the frequency of changing memory protection.

We could adopt an alternative method, in which the collector protects only live pages selectively at start phase. We have found, however, this method increases the number of protection and incurs more overhead.

5.2.2 Software Write Barrier

We have described a write barrier that uses virtual memory facility, without compiler support. Many compilers of modern languages such as ML and Java can support incremental/concurrent GC by inserting code fragments at object updating codes. While this software write barrier approach produces overhead even out of GC cycles, its overhead does not grow on multiprocessors. Section 5.3 estimates the performance of such write barrier by software. For this purpose, we have instrumented the source code of applications by hand. We expect overhead of this instrumented version performs as well as that of write barrier with compiler support.

In the experiments, the collector remembers dirty regions per page. With software write barrier, we could maintain the dirty regions in a finer grain manner. This will enable us to reduce the amount of GC tasks in the finalize phase, though we have not implemented it.

5.3 Performance Evaluation

This section compares the concurrent parallel GC and the stop parallel GC on two multiprocessors: Enterprise 10000 SMP and Origin 2000 DSM. We use three applications: BH-pt, CKY and Cube, which we have described in Section 1.1. We also measure the cost of software write barrier, by inserting write barrier codes in source codes of BH-pt and Cube by hand.

CKY and Cube are parallelized by using StackThreads/MP, a fine grain thread library. We have made a modification to its user-level thread scheduler, and allow idle threads to participate in concurrent marking.

5.3.1 Total Execution Time

Figure 5.2 shows the execution times of applications with several GC versions. 'Stop' represents the stop parallel collector described in Chapter 3. 'Conc-VM' refers to the concurrent parallel collector introduced above. Its write barrier is implemented by using virtual memory facility. 'Conc-Soft' refers to the concurrent parallel collector with software write barrier. We have measured another version for comparison; 'Stop-Soft' refers to a stop parallel collector that suffers from overhead of software write barrier. The results in the figure are normalized to that of 'Stop'.

There are no considerable difference between 'Stop-Soft' and 'Conc-Soft' in all cases. This result suggests that the most important element that affects the execution times is the overhead of write barrier, rather than the increase of GC tasks on current collectors.

In all applications, the overhead of 'Conc-VM' gets larger as the number of threads increases. The reason seems to be that changing memory protection takes a longer time on multiprocessors. With 60 threads on Enterprise 10000, the execution times of 'Conc-VM' are 24%–83% longer than that of 'Stop'. The differences are 21%–94% with 64-four threads on Origin 2000.

On the other hand, overhead of software write barrier does not grow. The overhead in Cube tends to decrease as threads increase when we have more than 8 threads. In BH-pt, 'Conc-Soft' is always faster than 'Conc-VM'. In cube, 'Conc-Soft' outperforms 'Conc-VM' with more than 32 threads on Enterprise 10000, and more than 48 threads on Origin 2000. The overhead of software write barrier is larger on Enterprise 10000 in BH-pt, while Origin 2000 incurs more overhead in Cube. The reason of this phenomenon is unclear yet.

5.3.2 Pause Time

This section shows GC pause times, which have heavy effects on application response times. We focus on pause time of finalize phase, rather than that of concurrent phase. The reason is that reducing pause time of concurrent phase is not hard in principle [3]. On the other hand, finalize phase needs to atomic from a viewpoint of applications; dividing this phase is hard.

[3] We do NOT assert that it is trivial to guarantee the sufficient progress of applications.

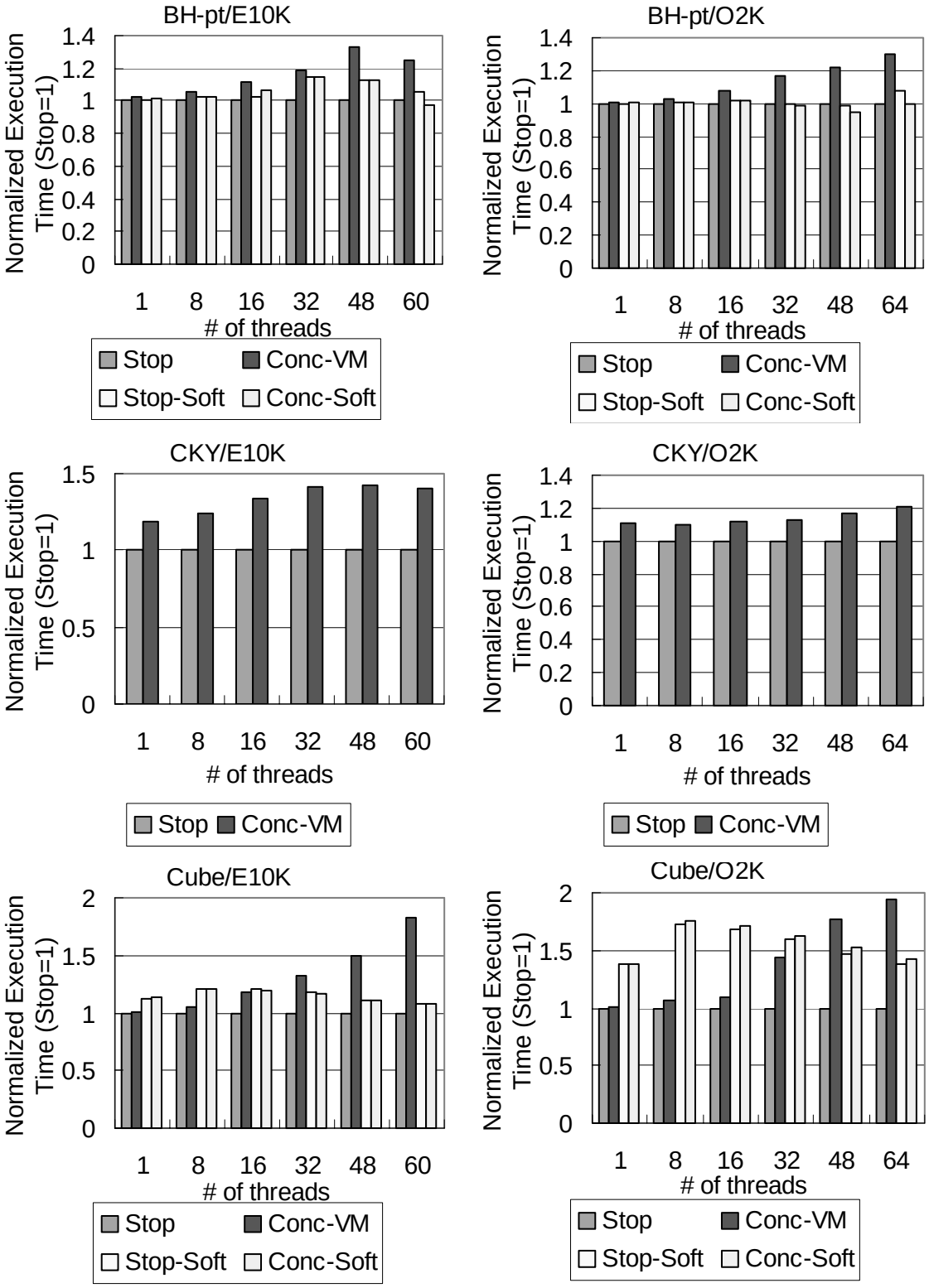

Figure 5.2: Total execution time of each application. The results are normalized to that of 'Stop'.

BH-pt/E10K

# of threads	1	8	16	32	48	60
Stop avg.	259	77	43	34	34	37
Stop max.	360	104	53	50	42	46
Conc-VM avg.	193	24	18	19	23	27
Conc-VM max.	347	41	27	29	37	40

CKY/E10K

# of threads	1	8	16	32	48	60
Stop avg.	518	125	65	42	41	47
Stop max.	1307	214	99	61	55	57
Conc-VM avg.	161	51	33	27	30	35
Conc-VM max.	269	79	52	43	38	46

Cube/E10K

# of threads	1	8	16	32	48	60
Stop avg.	288	59	47	36	46	47
Stop max.	452	88	83	55	58	67
Conc-VM avg.	206	34	36	27	35	42
Conc-VM max.	283	56	62	49	59	57

Table 5.1: Average pause time and worst pause time on Enterprise 10000, in milliseconds. 'Conc-VM' shows the pause time of finalize phase.

Tables 5.1–5.2 compare the full collection time of 'Stop' and the time of finalize phase of 'Conc-VM'. Tables show average pause times and worst pause times. We can see the pause times of 'Conc-VM' are shorter than those of 'Stop' in many cases. The advantage is especially large with less than 16 threads; the pause times of 'Conc-VM' are 1.3–3 times shorter. However, the advantage becomes smaller with more threads. Finalize phase of 'Conc-VM' does not scale as well as 'Stop'. We consider this is because the costs that are independent of the amount of GC task, such as synchronization costs, dominate the time of finalization phase on many threads. Unfortunately, 'Conc-VM' sometimes yields worse pause times on 60–64 threads.

BH-pt/O2K

# of threads	1	8	16	32	48	64
Stop avg.	131	53	51	52	58	73
Stop max.	285	72	62	62	73	91
Conc-VM avg.	221	54	35	38	44	66
Conc-VM max.	338	87	50	59	68	126

CKY/O2K

# of threads	1	8	16	32	48	64
Stop avg.	611	161	99	75	76	90
Stop max.	1580	247	143	101	97	111
Conc-VM avg.	321	89	55	48	53	68
Conc-VM max.	639	148	81	73	68	85

Cube/O2K

# of threads	1	8	16	32	48	64
Stop avg.	288	73	73	58	72	87
Stop max.	443	124	94	78	88	113
Conc-VM avg.	251	69	54	62	59	95
Conc-VM max.	380	96	75	100	100	164

Table 5.2: Average pause time and worst pause time on Origin 2000, in milliseconds. 'Conc-VM' shows the pause time of finalize phase.

5.4 Discussion

Section 5.3.2 has shown that current implementation cannot shorten pause times extremely. There some approaches to reduce pause times.

Using fine grain write barrier: In the finalize phase of current algorithm, the collector performs (1) marking from dirty pages, and (2) marking from roots. While the latter must be atomic in principle, we can move up the former to concurrent phase. Although this modification can reduce the finalize phase, the execution time would suffer from very large overhead when we use write barrier using virtual memory. If the collector removes a page from the dirty page set and scans it in concurrent phase, the page must be protected again to catch updates in the future. Thus the collector may change the protection of each page for multiple times during a single GC cycle. This modification causes much larger overhead than current algorithm, where each page is protected once and unprotected at most once during a GC cycle. When we use software write barrier, it would be possible to implement this modification without so heavy overhead.

Using another algorithm: Another approach is to use an algorithm that does not require finalize phase. The snapshot-at-beginning algorithm [53, 52] retains all objects that are live at the beginning of GC cycle, and objects allocated during GC cycle. It does not require finalize phase to rescue reachable and unmarked objects. This algorithm is more 'conservative' than incremental update algorithm; it reclaims less garbages. Therefore the larger number of GC cycles are required to proceed application, and may extend the execution time. More detailed comparison is under investigation.

5.5 Related Work

Incremental/concurrent garbage collection has a long history [19, 2, 53, 52, 20, 32, 43]; many researchers have described tracing GC algorithms that allow applications to proceed during collection. The advantages of this approach are twofold: reducing the pause time and efficient utilization of multiprocessors. However, most of previous work do not work well on large scale

multiprocessors because the collector itself is not parallelized. If only a single dedicated thread performs GC, starvation may occur when we have a lot of application threads. GC speed may fail to catch up with the allocation request speed by application.

Some implementations of concurrent GC use virtual memory facility rather than compiler support, such as an incremental mark-sweep collector by Boehm et al.[9]. In their implementation, only a single thread can process collection work. Our implementation is a parallel extension to their collector. Their collector is based on incremental update method, and requires the finalize phase. It supports generational collection, that investigates only newly allocated objects, though we did not use that facility. Their implementation [8] provides two methods to implement a write barrier on general purpose OS: (1) protecting memory pages and catching the write access to objects, and (2) reading dirty bits of all pages on SVR4 OS. As they have already described, it takes a long time to read dirty bits, which must be read atomically. Especially, the overhead is larger on multiprocessors. Thus we used memory protection facility for our experimentation. Protecting the heap need not be atomic in principle; we can protect pages in the heap incrementally.

Cheng et al.[14] described and evaluated a concurrent parallel copying garbage collector for multiprocessors. Their algorithm is based on a replicating garbage collector by Nettles et al.[43], which requires only a write barrier, not a read barrier. Cheng's collector achieves scalability by using thread local task pool, and performs work sharing via a shared pool. The space and Time that their collector consumes is theoretically bound. They implemented the collector within the runtime system of a SML compiler. The worst pause times of their collector range from 3 to 5 ms on uniprocessor, which is much better than that of our current implementation. We believe this difference comes from the following reasons.

- They adopt a technique to enable fine grain root scanning.
- Their collector does not require a long atomic phase, such as the finalize phase. However, their collector is more 'conservative' than ours; all objects allocated during GC cycle is retained. Thus it may require more collection cycles to proceed applications.

- Since their collector uses software write barrier, it can maintain updated objects in a fine grain manner [4].

Ichiyoshi and Morita's collector [34] is also concurrent (asynchronous) and parallel. Collection on the shared-heap is done as follows. Each thread asynchronously traverses objects that are reachable from its own root and local heap. Their algorithm allows for several threads to proceed collection simultaneously. However, their algorithm seems to suffer from starvation if the amount of reachable objects are not fairly distributed, because no dynamic load balancing is performed.

5.6 Summary

This chapter has described implementation and performance of a concurrent parallel GC, in which collector threads and application threads run in parallel, and collection itself is done in parallel. Our current implementation is based on the incremental update method. Through the experimentation on Enterprise 10000 SMP and Origin 2000 DSM, we have compared performance of concurrent parallel GC and stop parallel GC. With concurrent parallel GC, execution times of applications are longer because of some factors such as increase of GC tasks and overhead of a write barrier. We have shown overhead of a write barrier is important. The implementation that uses virtual memory facility incurs heavier overhead as the number of application threads increases. With 60–64 threads, execution times suffer from 21–94% overhead. GC pause times of incremental update method depends on the performance of finalize phase. The running time of finalize phase is 1.3–3 times shorter than that of stop parallel GC, with 16 threads or less. Unfortunately, this advantage becomes smaller when we have more threads. Concurrent parallel collectors may potentially be able to improve execution times, by utilizing idle processors during application execution for collection work. However, we did not see such cases.

We are planning to brush up our implementation. For example, making the software write barrier fine grain would reduce pause time of finalize phase. We also require comparison between incremental update and

[4] Although our collector supports software write barrier, it currently maintains updated objects coarsely. More implementation efforts are required.

snapshot-at-beginning both in theory and through experiments.

Chapter 6

Conclusion

This thesis has described design and implementation of a dynamic memory management module, which achieves scalability on large scale shared memory machines. High performance general purpose memory allocator relieves programmers' job to implement application specific allocators. High performance garbage collectors makes the approach of automatic memory reclamation more attractive.

Many parallel allocators described before maintain thread-local heaps to serve allocation requests in parallel. While they achieve scalability of allocation, most of them does not account for memory utilization and locality. Our allocator achieves scalability, locality on DSM machines, and high memory utilization. It also enables users to control the tradeoff between locality and memory utilization. Through experiments, the allocation speed with our allocator with 64 threads is 36 times faster than that on serial execution. By allocation speed-up and better locality, overall application performance improved by 2–19 %, compared to simpler parallel allocator.

Although there are many studies about garbage collectors on multiprocessors, researchers tend to focus on concurrent garbage collectors, where one thread performs GC while other thread proceeds application. This approach has an advantage that pause time of application is negligible. However, with much more processors, reclamation speed may fail to catch up allocation speed. We have constructed parallel garbage collectors; multiple threads cooperatively perform GC. In stop parallel GC, all program threads are stopped and all threads cooperatively perform GC. The some studies have adopted this approach, however, most of them do not achieve enough

performance. We have proposed some optimizations to eliminate bottleneck. With the optimizations, our collector is 14 to 28 times faster than serial GC with 60 threads on Sun Enterprise 10000. In concurrent parallel GC, application threads can run with multiple collector threads simultaneously. We have evaluated the overall execution times and pause times, and shown overhead of write barrier heavily affects execution times.

While our stop parallel collector scales well on SMP, the performance is sometimes severely limited on distributed shared memory (DSM) machines. To understand the performance, we constructed the GC performance model that accounts for difference among architectures. The model is conscious of access contention at receiver memory node, as LoPC model is. We found the contention cost heavily affects GC performance on DSM machines.

We list a part of future work below.

Using compiler support This thesis has described a memory management module without any compiler support. By using compiler support, we could have more efficient module. First, if compiler provides type information of objects, we can adopt moving garbage collectors such as copying GC and mark-compact GC. Such collectors can eliminate fragmentation and enable fast allocation that uses an allocation pointers rather than free lists. Secondly, if compiler emits a write barrier code, more scalable write barrier than that uses virtual memory would be possible. Thirdly, we can alleviate the frequency of accesses to the single shared heap, by using the results of static analyses such as the escape analysis [18, 51] and the region analysis [50]. These analyses the scope of allocated objects in given programs. We can allocate thread-local objects, which are accessed by a single threads, on stacks or thread private heap [47]. Thus we can reduce the frequency of allocation and garbage collection of the shared heap, which involve multiple threads. Note that each thread cannot collect thread local heaps described in Chapter 2 in local, because any objects can be shared by several threads in our current system.

Support for distributed architecture Recently, distributed architecture such as PC/workstation cluster is preferred as a method to realize high performance computing system in cost-effective approach.

We are planning to construct high performance memory management module for distributed architecture. An approach is to construct our module that this thesis described on top of software distributed shared memory [39, 36]. However, we would need to new optimization techniques for this environment, as software DSM has different characteristic from SMP or hardware DSM. Chapter 2 focused on how we decides the home of each object on hardware DSM. The decision is more important software DSM, where access costs to remote memory is much larger than on hardware DSM. Many tracing garbage collector algorithms for distributed environment have been proposed [33, 44]. Their algorithm assumes message passing, and each processor only traces local objects. On the other hand, our GC algorithm allows collector threads to trace any objects in the heap, even on hardware DSM. While the distributed approach that accounts for locality can reduce the number of communication, it may suffer from load imbalance and fail to shorten GC time. By mixing the distributed GC algorithm and our algorithm for shared memory, it may be possible to construct a better GC system.

Bibliography

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 96–107, 1991.

[2] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–20, 1988.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

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

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–358, November 1994.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An

efficient multithreaded runtime system. *The Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[8] Hans-J. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[9] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–164, 1991.

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

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

[12] Alan Charlesworth, Nicholas Aneshansley, Mark Haakmeester, Dan Drogichen, Gary Gilbert, Ricki Williams, and Andrew Phelps. The Starfire SMP interconnect. In *Proceedings of ACM/IEEE Conference on High Performance Networking and Computing (SC97)*, November 1997.

[13] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM,13(11)*, pages 677–678, 1970.

[14] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 125–136, 2001.

[15] 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.

[16] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM,2(12)*, pages 655–657, 1960.

[17] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[18] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the 17st ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[19] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[20] 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.

[21] Toshio Endo. A scalable mark-sweep garbage collector on large-scale shared-memory machines. master thesis, Department of Information Science, The University of Tokyo, February 1998.

[22] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of ACM/IEEE Conference on High Performance Networking and Computing (SC97)*, November 1997.

[23] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *15th Conference Proceedings of Japan Society for Software Science and Technology*, September 1998. (in Japanese).

[24] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Locality-aware parallel bibop memory allocator on shared memory multiprocessors. In *Proceedings of IPSJ Joint Symposium on Parallel Processing 2001*, pages 141–148, June 2001. (in Japanese).

[25] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Predicting performance of parallel garbage collectors on shared memory multiprocessors. *JSSST Computer Software*, 18(2):54–58, March 2001. (in Japanese).

[26] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Predicting scalability of parallel garbage collectors on shared memory multiprocessors. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, April 2001.

[27] The Apache Software Foundation. Apache http server project. http://httpd.apache.org.

[28] Matthew I. Frank, Anant Agarwal, and Mary K. Vernon. LoPC: Modeling contention in parallel algorithms. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 276–287, June 1997.

[29] Matteo Frigo. *Portable High-Performance Programs.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

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

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

[32] 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.

[33] John Hughes. A distributed garbage collection algorithm. *LNCS 201*, pages 256–272, 1985.

[34] 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.

[35] 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.

[36] Peter Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Computer Science, Rice University, January 1995.

[37] Per-Ake Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 176–185, October 1998.

[38] 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.

[39] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[40] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM,3(4)*, pages 184–195, 1960.

[41] 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.

[42] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[43] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 217–226, June 1993.

[44] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the 1995 SIGPLAN International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, 1995.

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

[46] 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.

[47] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 18–24, October 2000.

[48] 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.

[49] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 60–71, May 1999.

[50] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pages 188–201. ACM, 1994.

[51] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 187–206, November 1999.

[52] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the 1992 SIGPLAN International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, 1992.

[53] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11:181–198, 1990.

Appendix A

Application Interface of Scalable Memory Management Module

This chapter describes appliaction program interface (API) of our memory management module, which this thesis has shown. Our module is named SGC (GC for Shared memory multiprocessors), and it is a parallel extention to Boehm's conservative garbage collector library [11, 8]. It is based on version 4.10 of Boehm's GC, and some ideas and code fragments are from newer version. We hereafter call the library by Boehm "the original version".

SGC is a garbage collecting storage allocator for parallel C/C++ programs. Programs may use a thread library (such as solaris threads or pthreads).

Each thread allocates objects from the shared heap. In the multiple process version, the heap is shared using a memory mapped file (mmap in Unix) implicitly. The library has the following features to achieve good performance on multiprocessors:

parallel memory allocation Memory allocation is thread-safe and multiple allocations can be done in parallel.

parallel garbage collection When a garbage collection is requested, all user threads are stopped. Then multiple threads cooperatively perform the collection. We hereby shrink the time of a single garbage collection (For now, we do not support so-called 'concurrent garbage collection' by which we mean garbage collection that runs concurrently

with the application. We instead 'parallelize' a single collection, assuming the application is stopped during a collection).

There are some functions that exist in the original version but are missing in the parallel version. There are also some changes in their interfaces. For example, we require the user program to explicitly call the initialization function at appropriate points.

A.1 Supported Environments

The SGC library currently works on the following environments.

- Solaris on SPARC (v8plus or higher) processor
- Solaris on Intel (Pentium or higher) processor
- Linux on Intel (Pentium or higher) processor
- IRIX on MIPS (MIPS II or higher) processor

A.2 Description of API

The user program must include sgc.h (sgc_cpp.h for C++ programs).

All functions are thread-safe. Public variables such as GC_free_space_divisor are shared by all threads. Thus, changes by one thread are visible to all threads. It is the responsibility of the programmer to safely share these variables among threads.

`SGC_init(int ngcp, SGC_attr_t *attrp)` initializes the garbage collector. This must be called exactly once, before any GC function is called. The library uses NGCP threads for garbage collection. Currently ATTRP must be NULL.

`int GC_pthread_create(...)` is valid only when the library is compiled for the pthread version. It is a replacement of pthread_create. It creates new thread. Do not directly use pthread_create. Parameters and return values are the same as pthread_create.

`int GC_pthread_join(...)` is valid only when the library is compiled for the pthread version. It is a replacement of pthread_join. It waits for

thread termination. Do not directly use pthread_join. Parameters and return values are the same as pthread_create.

`int GC_thr_create(...)` and `int GC_thr_join(...)` are valid only when the library is compiled for the Solaris thread version. They are replacements of thr_create or thr_join. Parameters and return values are the same as thr_create or thr_join.

`pid_t GC_sproc(...)` and `pid_t GC_sprocsp(...)` are valid only when the library is compiled for the sproc version. They are replacements of the 'sproc' and 'sprocsp' in IRIX. It creates a new process that shares the heap. Do not directly use sproc. Parameters and return values are the same as sproc. You must use PR_SADDR option, because the sproc version library assumes shared address space.

`pid_t GC_wait(...)` and `pid_t GC_waitpid(...)` are valid only when the library is compiled for the sproc version. They are replacement of the 'wait' or 'waitpid' in Unix. It waits for process termination. Do not directly use wait or waitpid. Parameters and return values are the same as wait or waitpid.

`void *GC_malloc(size_t size)` allocates SIZE bytes of memory. The allocated object is automatically freed when it becomes unreachable.

`void *GC_malloc_atomic(size_t size)` is similar to GC_malloc, but GC_malloc_atomic assumes that there are no pointers relevant to GC.

`void *GC_malloc_uncollectable(size_t size)` is similar to GC_malloc, but the allocated object is not automatically freed.

`void *GC_malloc_atomic_uncollectable(size_t size)` is simliar to GC_malloc, but the allocated object is treated as atomic (pointer-free), and not automatically freed.

`void GC_free(void *object)` explicitly frees the object allocated by GC_malloc and friends.

`int GC_expand_hp(size_t size)` explicitly expands heap size.

`void GC_gcollect()` explicitly invoke a garbage collection.

`void GC_add_roots(char *low, char *high)` registers the region from low and high as a root.

`void GC_clear_roots()` removes all roots added by GC_add_roots.

`void GC_register_finalizer(GC_PTR obj, GC_finalization_proc fn, GC_PTR cd, GC_finalization_proc* ofn, GC_PTR *ocd)` registers a final-

izer. The finalizer is a user-defined function that are called automatically when associated object is reclaimed.

`int GC_free_space_divisor` controls the frequency of garbage collections (the default is 4).

`int GC_dont_gc` controls invocation of garbage collections. When set to 1, garbage collector never performs garbage collection even if the heap overflows. It instead expands the heap (the default is zero).

`int GC_gc_no` counts the number of GCs.