Predicting Scalability of Parallel Garbage Collectors on Shared Memory Multiprocessors

Toshio Endo, Kenjiro Taura, and Akinori Yonezawa
Department of Information Science, Faculty of Science
University of Tokyo
7-3-1 Hongo Bunkyo-ku, Tokyo 113-0033, Japan
{endo, tau, yonezawa}@is.s.u-tokyo.ac.jp

Abstract

This paper 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 paper shows the effect of automatic regulation of GC parallelism.

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. 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) [6, 5].

There are many factors that affects parallel GC performance: memory access contention, task stealing, and so on. The goal of this paper is to analyze the effect of each factor quantitatively. For this purpose, this paper 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) [3] and the SGI Origin 2000 (hypercube connected DSM) [10].

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 2 shows our parallel GC, which is the target of prediction. Section 3 describes our prediction method. Section 4 compares the predicted performance and the real performance, and Section 5 mentions related work.

2 Parallel Mark-Sweep Garbage Collector

We focus on our parallel mark-sweep GC for shared memory machines, which we have formerly developed [6]. Our GC is a parallel extension to Boehm-Demers-Weiser conservative GC library [2]. When any thread detects memory shortage, it suspends all application threads, and then several GC threads cooperatively perform marking and sweeping. They perform dynamic load balancing to achieve scalability.

GC threads traverse the graph of all live objects in the heap with the lazy task creation (LTC) strategy [11]; each GC thread traverses objects in depth-first, by using its own task pool, called mark stack. When a GC 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.

3 Prediction Method

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 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 3.4). Then we estimate T_P, which is the running time that excludes cache miss costs.

2. 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 3.5).

3. We calculate the cache miss cost on parallel execution M_P by using Mean Value Analysis(MVA). Generally M_P is larger than the miss cost on serial execution M_1, because of access contention (Section 3.6).

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

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 ignore the overlap between memory access latency and other computational instructions.

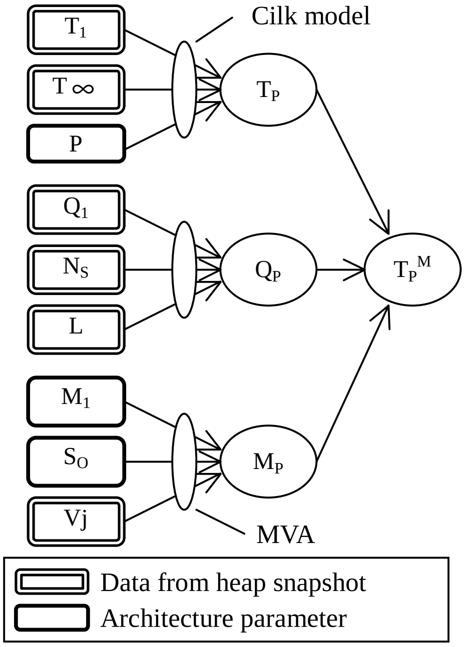

Figure 1. Overview of our performance prediction method. T_P^M is the final result; the marking time with access costs and contention costs.

3.3 Architecture Parameters

This section describes the basic memory access costs of parallel machines. We let M_1 be the round-trip time of memory access request without contention costs, and S_O be the occupancy time of the receiver memory node by each request. We have determined them through benchmark tests that aggressively access memory.

Origin 2000 SGI Origin 2000 (O2K) DSM machine consists of several nodes, each of which includes two R10000 processors and one memory node. We used a 64 processors machine for experiments. Each 16KB memory page becomes local to the processor that first touched the page. This rules may incur unbalanced memory distribution.

Enterprise 10000 Sun Enterprise 10000 (E10K) SMP machine has 64 Ultra SPARC processors and 16 memory nodes. Memory regions are automatically located fairly among all memory nodes.

The access costs on these machines are shown in Table 1.

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 1. Memory access cost on O2K and E10K, obtained from benchmark tests. ‘RW’ stands for atomic read-modify-write access.

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. Now we can estimate T_P, which is predicted parallel marking time without costs of cache misses, as $T_P = T_1/P + T_\infty$. This estimate comes from the Cilk performance model [1].

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, and the number of average living cache lines L during mark phase. On DSM, we also 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 estimate of N_S; $N_S = Plog(T_1)$.

3.5 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.

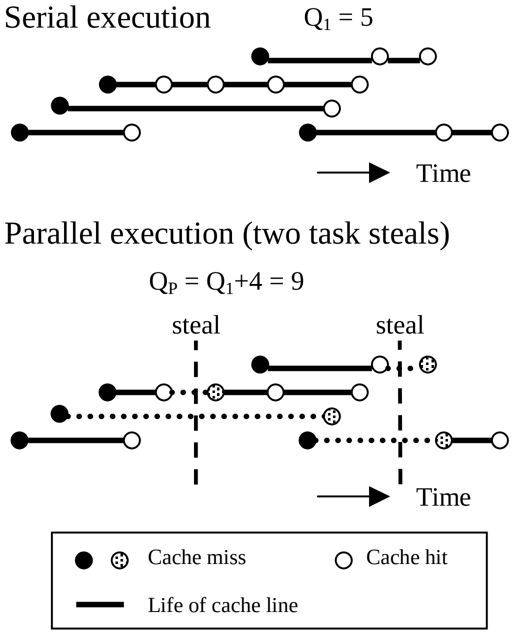

Figure 2. The behavior of cache lines, in serial execution and parallel execution.

Figure 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, 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.

3.6 Cost of Cache Misses

This section estimates the cache miss costs on parallel execution M_P, from sequential access costs M_1. We utilize MVA to account for access contention. This paper describes only contention at memory nodes. 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 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, Cube, CKY. They are written by using StackThreads/MP [12], a fine-grain thread library for C/C++.

BH simulates the motion of several particles by using the Barnes-Hut algorithm. Our BH implementation is not completely parallelized; only one thread creates the data structure to keep track of motion of the particles. Thus most live objects are located on a certain main memory node in O2K.

Cube searches an approximate solution of the Rubik's cube puzzle in breadth first fashion. Because all threads allocate the state records in parallel, live objects are distributed in all memory nodes in O2K.

CKY takes sentences written in natural language and the syntax rules of that language as input, and outputs all possible parse trees for each sentence [9].

4.1 Evaluation of Predicted Performance

Figure 3 and Table 2 compare the predicted result and the real performance on O2K and E10K. The graphs show GC speed-up by parallelization. "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.

In BH on O2K, our GC achieves only 10 fold speed-up, while it achieves much better scalability on E10K. 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 O2K; 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.

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

Table 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 3. Overhead and accuracy of two predictors, in BH/O2K. Overhead is the ratio of prediction time to running time of mark phase.

In Cube on O2K, the "Pred($Q_P = Q_1$)" is far from "Pred", thus we can see that Cube suffers from effects of miss increase by parallelization.

Table 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 are still some performance limiting factors that our model does not account for yet.

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 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.

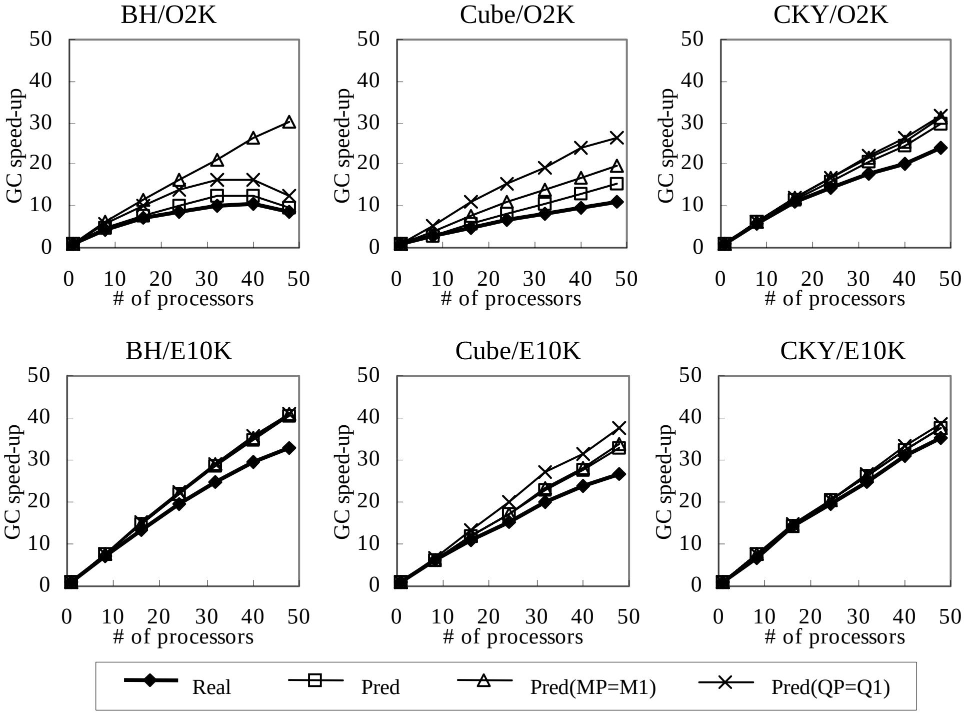

Figure 3. GC Speed-up. Graphs compare real speed-up("Real") and predicted speed-up("Pred").

4.3 Adaptive GC algorithm

One of the application of our performance model is construction of adaptive GC algorithms. As an example, this section describes automatic regulation of GC parallelism. For some applications such as BH on O2K, 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.

Table 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.

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.

# 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. The result of automatic regulation of GC parallelism, in BH/O2K.

The Cilk performance model [1, 8] 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. The recent model [8] by Frigo analyzes the costs of cache misses. However, current Frigo's model ignores contention costs, which heavily degrade the performance of parallel programs, especially on DSM. Frigo's model accounts for 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.

One of performance models that use MVA is the LoPC model [7]. It is based on the LogP model[4], and accounts for contention costs. We calculate miss costs with a similar method to the LoPC model.

6 Conclusion

This paper 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: E10K SMP and O2K DSM. The prediction error of parallel marking time with 48 processors is 7 to 38%. In BH application, which incurs unfair memory location, the GC scalability on O2K is much worse than that on E10K. 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.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, 1965.

[10] 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.

[11] 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.

[12] 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.