

## PARALLEL GARBAGE COLLECTION BY PARTIAL MARKING AND CONDITIONALLY INVOKED GC

YOSHIO TANAKA\*

*Department of Mathematics, Faculty of Science and Technology, Keio University,  
Yokohama-shi, 223, JAPAN  
tanaka@nak.math.keio.ac.jp*

SHOGO MATSUI†

*Department of Information Science, Faculty of Science, Kanagawa University,  
Hiratsuka-shi, 259-12, JAPAN  
sho@info.kanagawa-u.ac.jp*

ATSUSHI MAEDA\*, NAOKO TAKAHASHI\*, MASAKAZU NAKANISHI\*

*Department of Mathematics, Faculty of Science and Technology, Keio University,  
Yokohama-shi, 223, JAPAN  
mad@nak.math.keio.ac.jp, naoko@nak.math.keio.ac.jp, czl@math.keio.ac.jp*

### ABSTRACT

Automatic storage management such as garbage collection is essential to list processing systems. Unfortunately, many reclamation algorithms cause long pauses of execution. The pause time is getting shorter in modern garbage collection algorithms, however can not be eliminated perfectly. Parallel garbage collection is one of the efficient garbage collection scheme. It provides high performance list processing and is a key approach to the real time system. No practical parallel garbage collections have been implemented yet. Real time garbage collection can be considered in the same category of parallel garbage collection. There are two disadvantages common to parallel garbage collection and real time garbage collection. One is that collection efficiency becomes 1/2 compared with traditional sequential garbage collection, and the other is List processor's overhead when programs that do not consume many cells. We present solutions for these disadvantages, Partial Marking GC and Conditionally Invoked GC, and report an implementation and evaluation of our garbage collection algorithm. Our experiments shows that our algorithm reduce the disadvantages. The algorithm is applicable to various parallel garbage collection and real time garbage collection.

*Keywords:* Lisp, parallel garbage collection, real time garbage collection, generation scavenging

### 1. Introduction

List processing system needs large number of cells, which are basic units of a linked list, and garbage collection (GC) is required to collect and recycle disposed cells. GC normally causes long pause of execution, which restricts application of list processing systems.

The pause times is getting shorter in modern GC algorithms such as generation

---

\*3-14-1, Hiyoshi, Kouhoku-ku, Yokohama-shi, Kanagawa-ken, 223, JAPAN

†2946, Tsuchiya, Hiratsuka-shi, Kanagawa-ken, 259-12, JAPAN

scavenging GC<sup>5</sup>, but can not be eliminated perfectly. More efficient and/or real time (non-disruptive) GC algorithm is still a very important research theme.

In this paper, we point out the disadvantages common to parallel GC algorithms, and present solutions for such disadvantages. We also report an implementation and evaluation of our system.

## 2. Algorithms of Parallel Garbage Collection

Most parallel GC algorithm is based on Kung and Song's parallel mark and sweep algorithm<sup>3</sup>. In previous studies, we have designed and implemented a parallel garbage collector for Lisp machine SYNAPSE<sup>1</sup>. Yuasa's snapshot GC<sup>2</sup> is the same as our garbage collector except that the snapshot GC is not parallel but incremental. We call these algorithms generally as **Synapse Algorithm**.

Synapse Algorithm uses three colors as a tag for marking; black, white and off-white. off-white is a tag for cells in the free list. GC process (GP) repeats following three phases.

1. root insertion phase  
GP collects roots from the List Processor (LP).
2. marking phase  
GP marks all cells (i.e. change the tags to black) which are reachable from the roots.
3. collecting phase  
GP connects all cells whose tag is white to the free list. Black and off-white tag is changed to white except the cells in the free list.

LP notifies the GP that cells are modified during the marking phase. The destination cells are marked by GP. The cells which are created during the marking phase are never collected in the subsequent collecting phase, because the tags of newly created cells by CONS are off-white. Synapse Algorithm's correctness is guaranteed by these operations.

Synapse Algorithm are known to have following disadvantages when compared their efficiencies with sequential mark and sweep algorithm.

1. marking cost  
On Synapse Algorithm, cells active at the time of the root insertion phase and created during the marking phase are never collected in the subsequent collecting phase. Therefore, cells which died during the marking phase are collected not in the subsequent collecting phase, but in the next collecting phase. As shown in Hickey's analysis<sup>7</sup>, most garbage is collected in the next collecting phase, thus collection efficiency becomes 1/2 compared with traditional sequential GC algorithm. However, marking cost of those algorithm is the same as that of sequential mark and sweep GC algorithm. Furthermore, since most cells are short lived, the marking cost becomes more serious if the applications consume many cells.
2. List Processor's Overhead  
On Synapse Algorithm, GP always repeats three phases even if the application does not consume many cells. If we perform list processing and GC simultaneously, access conflict to the common resources occurs. That conflict is one of the causes to make parallel GC less efficient. It is more serious when the application does not consume many cells (hardly enter GC).

Our experiments showed the collection efficiency of 50% and the list processor's overhead of 10%<sup>4</sup>. And some programs run slower with parallel garbage collector than with sequential garbage collector.

### **3. Partial Marking GC**

Partial Marking GC is a variant of generational GC incorporated into Synapse Algorithm. Partial Marking GC tries to reduce the time spent for marking and improve the collection efficiency.

Instead of marking all cells in every marking phase, it marks all cells only once per several GC cycles (we call this phase full marking). In the rest of GC cycles, cells marked in previous GC cycles are assumed to be alive, and only part of cells are examined (we call this phase partial marking). Partial Marking GC decreases the average marking cost of more than 50%.

Partial Marking GC is realized by leaving the black tag which was set in the previous full marking phase. The time spent for marking is very short because partial marking marks very few cells. It is possible to collect cells which died during the previous full marking phase immediately.

Fig. 1 shows the algorithm of Partial Marking GC. A cell has two pointer fields (left, right) and a tag field (color). The total number of cells is M. The free list is pointed to by FREE. FREE.left points the head of the free list, and FREE.right points the tail of the free list. A special pointer f is stored in left part of every free cells. LP notifies the GP of rewriting cells during the marking phase. The source cell and the destination cell are both marked by GP.

### **4. Conditionally Invoked GC**

It is not necessary to keep garbage collector running when there are enough free cells. Conditionally Invoked GC controls the execution of GC and reduce the list processor's overhead. It is possible to improve the efficiency of GC by pausing or resuming GC according to the amount of the rest (usable) free cells when there is no urge demand to execute GC. To put it in the concrete, start GC only when the amount of free cells becomes less than a threshold. And when the collecting phase has finished, GC stops until the time GC becomes necessary again. By Conditionally Invoked GC, applications that do not consume many cells can be executed more efficiently. The efficiency is the same as sequential list processors that has no overhead of parallel execution.

### **5. Implementation**

Partial Marking GC and Conditionally Invoked GC have been implemented on OMRON LUNA-88K workstation. LUNA-88K comprises 4 processors and uses MACH operating system developed in CMU. MACH provides a set of low-level primitives for manipulating threads of control. A thread facility allows as to write programs with multiple simultaneous points of execution, synchronizing through shared memory. In our experiments, two threads are created; one is for list processing (we call this LP thread) and the other is for GC (we call this GC thread).

We used a Lisp system based on Klisp, a Lisp 1.5 dialect developed at Keio University. In order to execute GC concurrently, ps\_stack is added as new data area. The LP thread notifies the GC thread of rewriting cells during the marking phase through the ps\_stack. The ps\_stack is shared data between GC thread and LP thread, and must be accessed only from a thread. Some semaphores and flags are also added.

```

procedure root_insert ;
begin
  << push all roots onto the stack >>
end

procedure mark ;
begin
  while << the stack is not empty >> do
  begin
    n := pop ;
    while (n <> NIL) and
      (n.left <> f) and
      (n.color <> black) do
    begin
      n.color := black ;
      push(n.right) ;
      n := n.left ;
    end
  end
end

procedure collect_leave_mark ;
begin
  for i := 1 to M do
    if i.color = white then
      APPEND(i)
    else if (i.color = off-white) and
      (i.left <> f) then
      i.color := white
  end

procedure collect_clear_mark ;
begin
  for i := 1 to M do
    if i.color = white then
      APPEND(i)
    else if i.left <> f then
      i.color := white
  end

procedure PartialMarkingGC ;
begin
  while true do
  begin
    PARTIAL := true ;
    root_insert ;
    mark ;
    collect_leave_mark ;
    root_insert ;
    mark ;
    PARTIAL := false ;
    collect_clear_mark ;
  end
end

procedure LPa ;
begin
  if PARTIAL then
  begin
    push(m.left) ;
    push(n) ;
    m.left := n
  end
end

procedure LPd ;
begin
  if PARTIAL then
  begin
    push(m.right) ;
    push(n) ;
    m.right := n
  end
end

procedure LPc ;
begin
  << sleep while
    FREE.left = FREE.right >>
  NEW := FREE.left ;
  FREE.left := FREE.left.right ;
  NEW.left := m ;
  NEW.right := n
end

```

Figure 1: Partial Marking GC Algorithm (GP and LP)

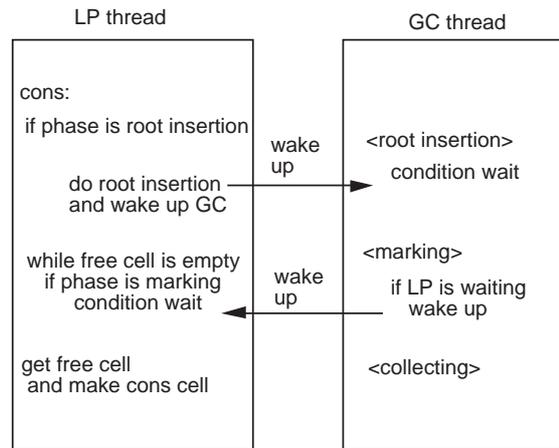


Figure 2: Relations of the LP thread and the GC thread

### 5.1. Implementation of Partial Marking GC

List processing and GC is processed concurrently by following two threads.

#### 5.1.1. LP thread

The LP thread executes list processing the same with the traditional Lisp interpreter except that it does not collect garbage cells by itself. The LP thread checks the root insertion flag every time CONS is called because interruption is not available in C-threads library. The root insertion flag is set when the GC thread enters the root insertion phase. The LP thread inserts all roots into the root\_stack when it finds that the root insertion flag is set. The LP thread notifies the GC thread of rewriting cells during the marking phase through the ps\_stack.

The LP thread is blocked while the free list is empty. The GC thread awakens the blocked LP thread at entering the collecting phase.

#### 5.1.2. GC thread

The GC thread repeats the root insertion phase, the marking phase and the collecting phase. In the root insertion phase, the GC thread sets the root insertion flag and blocks until the LP thread finishes to insert all roots into the root\_stack. Fig. 2 shows relations between the LP thread and the GC thread.

### 5.2. Implementation of Conditionally Invoked GC

The LP thread checks the root insertion flag every time CONS is called on Synapse Algorithm. The LP thread stops list processing and inserts all roots when the request flag has been set. The GC thread repeats the three phases throughout the execution of the application even if the application does not consume much cells.

Similarly, the LP thread checks the root insertion flag on Conditionally Invoked GC, however the LP thread inserts roots into the root\_stack only when it finds that the root insertion flag has been set and the amount of free cells is less than a threshold. Once the GC thread requests the LP thread to insert roots, it is blocked until the LP thread finishes to insert all roots into the root\_stack as shown in Fig. 2. The GC thread sleeps until amount of the free cells becomes less than a threshold. Once the GC thread is

invoked, it executes marking and collecting, then it requests the LP thread to insert roots and is blocked again. It is possible to incorporate Conditionally Invoked GC into Partial Marking GC. To implement Conditionally Invoked GC, we choose one of the following three strategies:

1. Do not perform Partial Marking GC  
The GC thread executes full marking in every marking phase.
2. Execute full marking alternately with partial marking.  
In the first marking phase, the GC thread executes full marking. In the second marking phase, the GC thread executes partial marking. The GC thread executes full marking in the odd marking phase, and executes partial marking in the even marking phase.
3. Execute full marking and partial marking at a stretch.  
The GC thread execute full marking and collection, and continuously execute partial marking and collection.

(1) has no advantage of Partial Marking GC. The advantage of Partial Marking GC is to collect garbage cells which died during the previous marking phase quickly. (3) is better than (2) because (2) loose the advantage of Partial Marking GC.

## 6. Evaluation

### 6.1. The Efficiency of GC

Let us define the following parameters. seq-lisp is the traditional Lisp interpreter which has stop-and-mark and sweep garbage collector. para-lisp is the Lisp interpreter which has parallel garbage collector.

$T_{seq.gc}$  The time spent on garbage collection in seq-lisp.

$T_{seq.lp}$  The time spent on list processing in seq-lisp.

$T_{seq.total}$  The total processing time in seq-lisp.

$$T_{seq.total} = T_{seq.gc} + T_{seq.lp}$$

$T_{para.gc}$  The time spent on garbage collection in para-lisp.

$T_{para.lp}$  The time spent on list processing in para-lisp.

$T_{para.total}$  The total processing time in para-lisp.

Let define **GC ratio** and Improvement ratio as follows.

$$G = \frac{T_{seq.gc}}{T_{seq.total}} \quad (1)$$

$$I = \frac{T_{seq.total} - T_{para.total}}{T_{seq.total}} \quad (2)$$

$I$  is the ratio of reduction time of the total processing time in para-lisp to the total processing time in seq-lisp.

Relations between  $G$  and  $I$  is loaded on the assumption that the application is stable and create cells at the constant rate.

In this case,  $T_{para.total}$  is

$$T_{para.total} = \max(T_{para.lp}, T_{para.gc}). \quad (3)$$

So,  $I$  is

$$I = \min\left(\frac{T_{seq.total} - T_{para.lp}}{T_{seq.total}}, \frac{T_{seq.total} - T_{para.gc}}{T_{seq.total}}\right). \quad (4)$$

$T_{para.gc} < T_{para.lp}$  if the collection speed is enough fast and the free list never be empty. In this case,

$$T_{para.total} = T_{para.lp}.$$

The LP thread waits for free cells in each GC cycle if the LP thread exhausts all free cells before the GC thread finish to collect garbage cells. In this case,

$$T_{para.total} = T_{para.gc}.$$

Now, let define  $T_{para.oh}$  be the overhead of LP in para-lisp, and  $n$  be the ratio of the efficiency of GC in para-lisp to in seq-lisp.

$$T_{para.lp} = T_{seq.lp} + T_{para.oh}T_{para.gc} = nT_{seq.gc} \quad (n > 0) \quad (5)$$

Let define overhead ratio be  $O = T_{para.oh}/T_{seq.lp}$ , then  $I$  is as follows.

$$\begin{aligned} I &= \min\left(\frac{T_{seq.total} - nT_{seq.gc}}{T_{seq.total}}, \frac{T_{seq.total} - T_{seq.lp} - T_{para.lp}}{T_{seq.total}}\right) \\ &= \min(1 - nG, G - O) \end{aligned}$$

$$I = \min(G - O, 1 - nG) \quad (6)$$

## 6.2. Evaluation of GC

We experimented the Synapse Algorithm on MACH, and graphed the  $I$  against  $G$  in Fig. 3. A dotted line shows the ideal graph ( $n=1, O=0$ ).

We run the stable application which repeats to cons waste cells fixed times. We use the length of the list for active data as a parameter.

We can estimate that  $O$  is about 10% and  $n$  is 2. The value of  $n$  is agreed with Hickey's analysis. The increasing part (the value of  $G$  is 0 to the maximum point) of the graph indicates the stable state (LP never waits for cells) which is described by Hickey. The decreasing part is divided into two parts: one is positive part and the other is negative part. The positive part indicates the alternating state (LP waits for cells in once per 2 GC cycles). The negative part indicates the critical state (LP waits for cells in every GC cycle). Relations between  $G$  and  $I$  is analyzed by Hickey. The evaluation of  $I$  on Synapse Algorithm is analyzed by Teramura<sup>8</sup>.

$n$  becomes 2 because the tags of newly created cells by CONS are off-white, and most of them becomes garbage immediately. However, they are collected not in the subsequent collecting phase, but in the next collecting phase.

## 6.3. Evaluation of Partial Marking GC and Conditionally Invoked GC

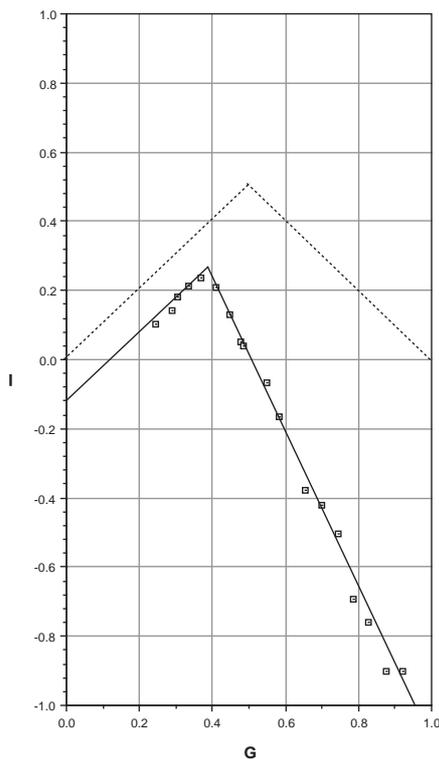


Figure 3: Improvement Ratio vs. GC Ratio (Synapse Algorithm)

### 6.3.1. Improvement ratio vs. GC ratio (Sequential GC vs. Partial Marking GC)

To compare the efficiency of traditional (sequential) GC and Partial Marking GC, we ran the same application as before. We graphed the  $I$  against  $G$ . The results are shown in Fig. 4.

The efficiency of Partial Marking is obviously improved. The efficiency of GC is almost the same efficiency as sequential mark and sweep GC, and graph became closer to the ideal graph. The peak of the graph which indicates real time performance slides to right. This means that real time processing is possible until GC ratio exceeds 0.5.

### 6.3.2. Execution Time

To compare the efficiency of traditional GC, Partial Marking GC and Conditionally Invoked GC, we ran many applications (Bit, Boyer, Destruct, ackerman and hanoi) and measured their execution time. The results are shown in Fig. 5. As shown in Fig. 5, the execution time in Conditionally Invoked GC is shorter than sequential GC's one by incorporating Conditionally Invoked GC into Partial Marking GC. However, the difference between them are little. The time is the longer part of LP's time or GC's time. On Conditionally Invoked GC, GC thread is blocked while there are many cells. Conditionally Invoked GC has an another advantage than execution time. Conditionally Invoked GC prevents to give a load with the system on which application is running. In fact, when we run applications on Conditionally Invoked GC, the load average which indicates the system's load is about 50% compare with one when we run applications on Partial Marking GC. We graphed the total execution real time of applications in Fig. 6. The result shows that to incorporate Conditionally Invoked GC into Partial Marking GC

Parallel garbage collection by Partial Marking and Conditionally Invoked GC

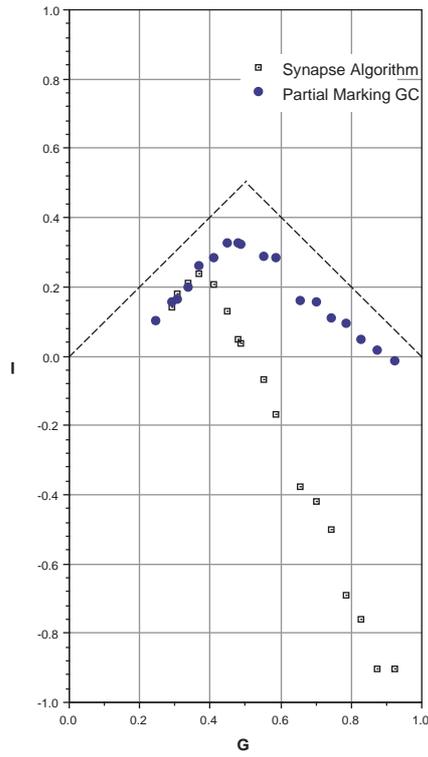


Figure 4: Improvement Ratio vs. GC ratio

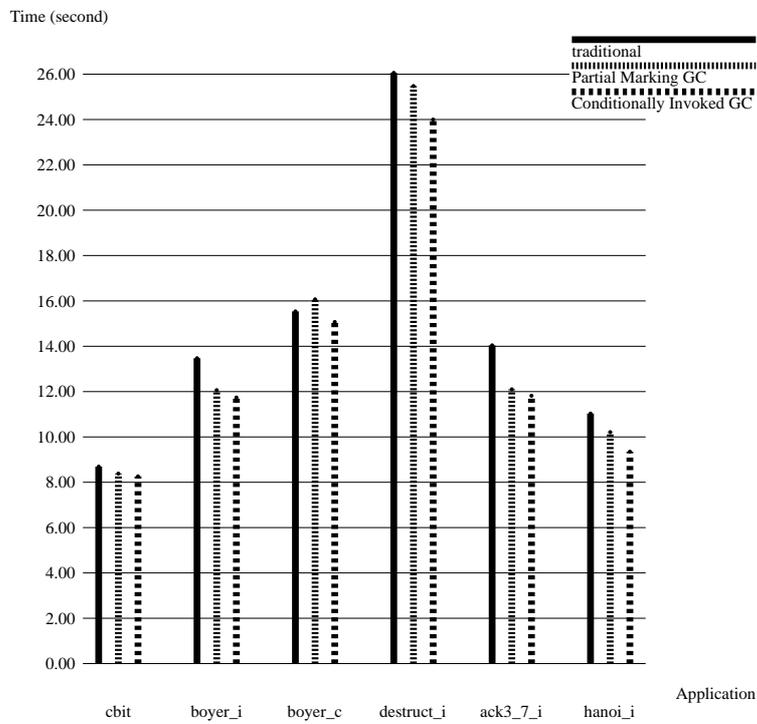


Figure 5: Execution Time

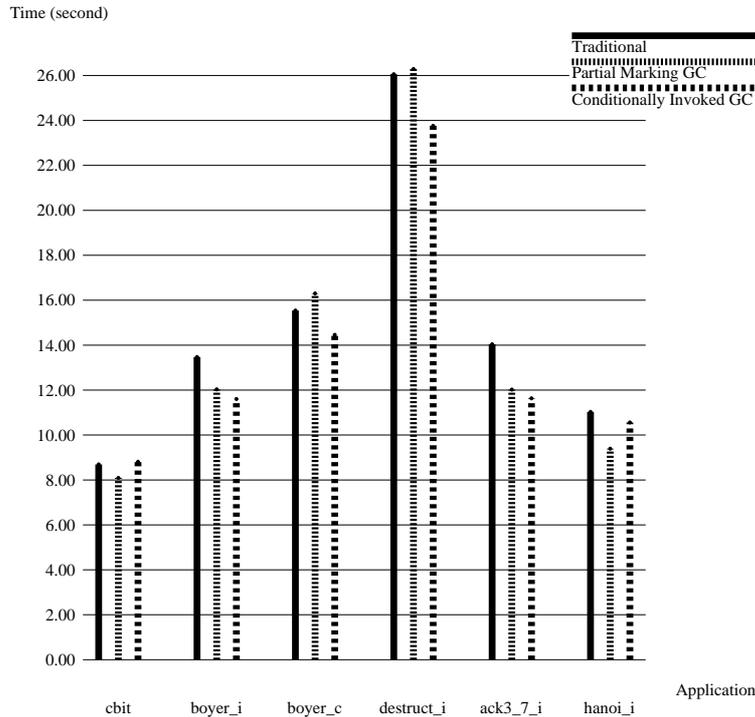


Figure 6: Execution Real Time

improves the efficiency of GC.

### 6.3.3. Invoking Threshold

The threshold (**Invoking Threshold**) is 10% in the above experiments. Invoking Threshold can be considered to effect the efficiency of GC. We examined relations between the Invoking Threshold and the efficiency of GC and graphed the efficiency of GC against Invoking Threshold. Clearly, the efficiency of GC is worst in case of Invoking Threshold is set to 20%. 15% is also worse than 5% and 10%. However, there are not major difference between 5% and 10%, but they are different between applications. This means that application has its own suitable Invoking Threshold.

## 7. Conclusion

We pointed out the following disadvantages common to parallel GC algorithms.

1. bad collection efficiency
2. list processor's overhead when programs do not consume many cells

We presented, implemented and evaluated the following two solutions for the above disadvantages.

1. Partial Marking GC
2. Conditionally Invoked GC

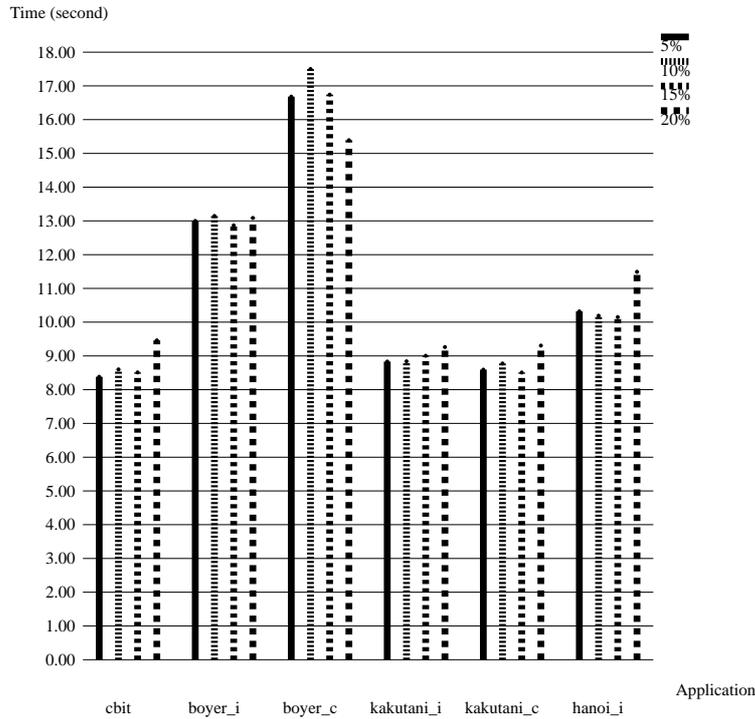


Figure 7: GC ratio vs. Invoking Threshold

On most applications, Partial Marking GC and Conditionally Invoked GC is effective, and the efficiency of GC and list processing is much improved.

Following two points are considered as our future work.

1. As described before, the best Invoking Threshold is different between running applications (between the rate of consuming cells). By measuring the rate of consuming cells and changing Invoking Threshold dynamically, adaptive Conditionally Invoked GC (suitable for any applications) can be realized.
2. In our experiments, 1 GC thread and 1 LP thread are created. Synapse Algorithm is applicable for multiple GC threads and multiple LP threads. We will implement the parallel Lisp interpreter which has multiple threads. At first, all threads execute list processing. When there is a necessary to collect garbage, some of the threads stop list processing and execute GC. The threads start list processing when GC finishes. We are planning to attach list processing or GC to multiple threads dynamically.

## **References**

1. Matsui, S. et al. SYNAPSE: A Multi-microprocessor Lisp Machine with Parallel Garbage Collector, Proceedings of the International Workshop on Parallel Algorithms and Architectures (1987) pp131-137
2. Yuasa, T. Real-Time Garbage Collection on General-Purpose Machines. The Journal of systems and software Vol 11, No.3, (1990) pp181
3. Kung, H. T. and Song, S. W. An Efficient Parallel Garbage Collection System and its Correctness Proof. Tech. Note, Dept. of Computer Science, Carnegie-Mellon University (Pittsburgh, Pennsylvania, 1977).
4. Matsui, S. and Tanaka, Y. et al. An implementation and evaluation of parallel garbage collector on Unix workstations, IPSJ SIGSYM 67-5,(1993) pp33-40 (in Japanese)
5. David Ungar. Generation Scavenging:A Non-disruptive High Performance Storage Reclamation Algorithm. ACM SIGPLAN Notices Vol.19, No.5, (1984) pp 157-167
6. Richard Rashid. et al. Mach: A System Software kernel. Proceedings of the 34th Computer Society International Conference COMPCON 89, February 1989.
7. Hickey, T. et al. Performance Analysis of On-the-fly garbage collection, Comm. ACM, Vol.27, No.11 (1984) pp1143-1154
8. Teramura, S. Analysis of Parallel Garbage Collection With Multiple List Processors and Garbage Collectors, Journal of Information Processing, Vol.12, No.3, (1989) pp229-238
9. Gabriel, Richard P. Performance and Evaluation of Lisp systems. The MIT Press (Cambridge, Massachusetts, 1985)