Efficient Parallel Global Garbage Collection on Massively Parallel Computers

Tomio KAMADA Satoshi MATSUOKA Akinori YONEZAWA
Dept. of Information Science, The University of Tokyo

Abstract

On distributed-memory high-performance MPPs where processors are interconnected by an asynchronous network, efficient Garbage Collection (GC) becomes difficult due to inter-node references and references within pending, unprocessed messages. Our parallel global GC algorithm (1) takes advantage of reference locality, (2) efficiently traverses references over nodes, and (3) admits minimum pause time of ongoing computations. The algorithm employs global weight counting scheme to substantially reduce message traffic. The two methods for confirming the arrival of pending messages are available: one counts numbers of messages and the other uses network 'bulldozing'. Performance evaluation in actual implementations on a multicomputer with 32-1024 nodes, AP1000, exhibits good performance.

1 Introduction

Several distributed GC (garbage collection) algorithms have been proposed in the past, but realistic applications to MPPs (Massively Parallel Processors) have not been entirely successful. This is because the algorithms (1) incurred excessive message traffic, (2) involved prohibitive runtime overhead, and/or (3) have not been implemented to test their validity/efficiency. The algorithms can be largely classified into two types:

- Using reference counts: This type of GC algorithm mainly performs local GC on each node and uses reference counting to decide which objects are referenced from other nodes [2, 9, 4, 6]. However, no algorithms we know to date can collect cyclic garbage ranging over nodes with low runtime overhead required for parallel computation.
- Distributed marking over nodes: This type of GC algorithm is an extension of the standard mark-and-sweep algorithm, and traverses/marks both local and inter-node references[1, 3, 5, 8]. They can collect cyclic garbages, but many involve long pause time and/or excessive message traffic. Messages by the garbage collector not only marks objects in other nodes, but also used to detect termination of marking traversal. Some algorithms allow independent local GC on each node to exploit reference locality[3, 5, 8].

We present an efficient real-life distributed GC algorithm based on an asynchronous message passing model and distributed marking. It assumes distributed-memory high-performance MPPs where processors are interconnected by a high-speed asynchronous network. The first prototype has been implemented and running on a 32-1024node MPP, Fujitsu AP1000.

Our GC algorithm has the following favorable properties:

- Our global GC traverses inter-node references, and can collect distributed cyclic garbages quickly and efficiently by exploiting parallelism of all the processors.
- GC algorithms that traverse inter-node reference tend to involve extensive message traffic, incurring high runtime overhead and also disturbs ongoing computation. Our global GC reduces message traffic by over 50% compared to straightforward algorithms, featuring efficient handling of outstanding references within messages and low-communication termination detection of tracing/marking.
- Our global GC admits minimum pause time for user applications (mutators), and allows concurrent mutator/collector operations for both local and global computations.
- Our GC algorithm allows local generational GC on each node to operate independently of other nodes (without invoking global GC), facilitating efficient collection of local garbage, taking advantage of reference locality.

Where object references may change by message passing, it is necessary for global GC to consider pending messages on asynchronous networks and confirm all arrivals of messages that were sent before global GC starts. We use two different methods for

confirming the arrival of pending messages: one is the *message counting method* that counts the numbers of incoming and outgoing messages, and the other is an improved variant of *bulldozing method* [8] that clear the network of outstanding messages by utilizing hardware network FIFOness. We evaluate their respective performances and compare their advantages/disadvantages.

Termination of marking must also be detected with low message traffic, but it is difficult to do so efficiently in distributed environments; our algorithm uses *distributed weight counting* to reduce message traffic.

We have implemented prototype version of our algorithm on Fujitsu AP1000, a multicomputer with 32-1024 nodes, and validated its correct operation and measured reduction in message traffic of global GC. Section 2 presents the system overview of our GC algorithms. Section 3 describes the objects to be marked. Section 4 covers the algorithm of termination detection. Section 5 evaluates the performance of our global GC, and we conclude in Section 6.

2 The GC System

2.1 Our Global GC System

Our global GC system consists of *local collectors* on each node and a *GC host*(Fig.1). Each local collector traverses references in its node. If an inter-node reference is found, the local collector sends a mark message to the owner node of the referenced object to traverse the reference. Local collectors that receive mark messages mark the referenced objects and traverse the references from there.

The GC host is a special node that decides the initiation of global GC, and also detects termination of marking. Messages for termination detection are sent only between the local collectors and the GC host. A local collector requests the GC host to start global GC when its local collection has failed to reclaim sufficient memory. The GC host judges whether global GC is needed, and if so announces to all the nodes to initiate global GC. Each local collector starts the global marking task when an announcement of global GC initiation or a marking message arrives. The execution of the mutator proceeds concurrently with local and global marking, and thus no pause time is incurred by, say, global barrier synchronization. The GC host determines termination of marking based on information from each local collector, and it sends messages to all the local collectors to release unmarked memory.

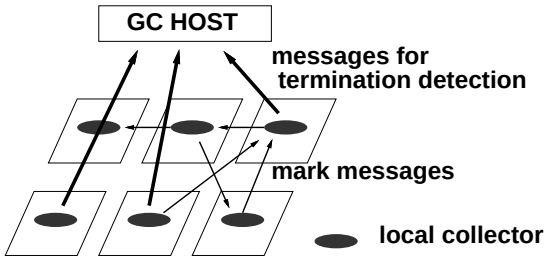

Figure 1: global GC system

2.2 Local GC

To allow local GC to operate independently of other nodes without invoking global GC, each node has an *export set*, that consists of pointers to objects possibly referenced from other nodes. When a pointer to an object is exported to other nodes, the node adds the pointer to its export set. Deletions from export set are done only via global GC to reduce the overhead during ordinary computation. When global GC starts, nodes clear their export sets, and local collectors reenter objects that are marked by mark messages from other nodes to the export set.

During local GC, each local collector traverses node-local references from active objects, unprocessed messages and objects in its export set as the root set. Untraversed objects are collected locally as garbage.

3 Which Objects to Mark in Global GC?

Global GC in a distributed environment poses difficulties in determining (1) which objects to mark and (2) how to detect termination of marking. This section discusses the issue (1), and the issue (2) is discussed in Section 4.

First, we define live objects and garbage in our global GC:

1. Objects that are executing methods, or (potentially) have unprocessed messages, are live (called *active objects*).

2. Objects referenced from (lexically) top-level variables are live *.

3. The transitive closure of references from live objects are live.

4. All other objects are garbage.

*These variables are typically called 'global variables'. However, we use this different terminology to distinguish lexical and runtime globalness.

Our global GC effectively takes a logical snapshot of the collective state of the global heap and the network at the start of the mark phase. We reclaim objects that were already garbage at the snapshot, and do not reclaim objects that become garbage during global GC. Newly created objects during GC are allocated as marked objects.

The global GC algorithm marks objects by globally traversing references from top-level variables, the execution context and the instance variables of logically active objects, and logically unprocessed messages. Pending messages in the network, however, complicate the treatment of reference relationships among objects. In Fig.2, a mutator message with a sole global reference to object C is sent from object A and has not arrived at its destination object B, whose sole global reference held at A is lost immediately after the message has been sent. If global GC starts at this point, B and C are live because B will become active when the message arrives, and C will be referenced by B. However, if the global GC would fail to recognize the message, both B and C would be mistakenly collected as garbage.

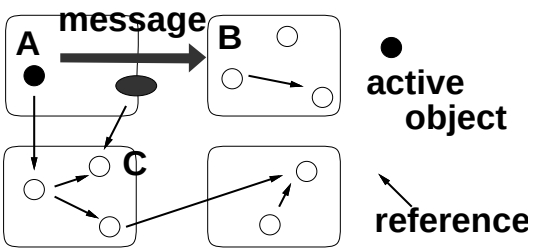

Figure 2: Old Messages sent before the initiation of global GC

Unprocessed messages that are sent just prior to the start of global GC and have not yet arrived at their destination (either traversing through the network or residing in some message buffer) are called *old messages*, and their destination objects are logically active. When a node receives an old message, the local collector on the node marks from the destination object and the objects that are referenced by the message. Algorithms for judging whether a given mutator message is an old message or not is described in Section 4.1.

On a given node, if old messages arrive after marking termination, objects referenced by old messages may be wrongly reclaimed in the collection phase that immediately follows. Confirming the arrival of all the old messages must be done before determination of marking termination. On the other hand, when we have marked all objects that had been accessible from each node at the initiation of global GC, and also marked objects that had been accessible from messages that had been sent before initiation of global GC on each node, we can guarantees that no unmarked objects will be referenced from any of the nodes, even if the initiations of global GC are not logically consistent. Algorithms for arrival confirmation are discussed in Section 4.1, and marking termination algorithm is presented in Section 4.2.

Finally, in our global GC, as marking and mutators are concurrently processed, mutators may write into unmarked objects and destroy the reference relationship at the initiation of global GC. In order to maintain the global virtual snapshot, we use the standard technique of *write barrier* to detect when pointer informations are overwritten by mutators, and local collectors mark from the references that will be lost by the writing.

Altogether, local collectors perform the following marking actions during global GC:

1. Mark from global variables, local active objects, and their unprocessed messages.

2. Mark from remote references via mark messages from other nodes.

3. Mark from old messages.

4. Mark from references lost in local writes into unmarked objects.

4 Termination Detection

It is also difficult to detect termination of marking with low message traffic in a distributed environment.

Reviewing the discussions in Section 3, marking can be judged as being terminated when the following three conditions are satisfied:

1. The arrivals of all the old messages have been confirmed.

2. All the local collectors have no further objects to mark.

3. No mark messages are in transit within the network.

We refer to the set of global system states that satisfy conditions 2 and 3 as *quiescent states*.

If above conditions are satisfied, all the live objects are marked, and there will be no further writes into unmarked objects, and local collectors will not mark any more objects.

Section 4.1 presents two methods for confirming the arrival of all the old messages that work concurrently with mutators and marking, and Section 4.2 describes an algorithm to detect the quiescent state of marking with low message traffic.

4.1 Arrival Confirmation of Old Messages

Our global GC confirms the arrival of all the old messages to assure marking from all the references contained therein. We have two alternative methods to confirm the arrival: one is *message counting method* that eliminates the mutator ACK (acknowledge response) messages, and the other is *bulldozing method* that improves the network clearing operation algorithm described in [8]. Both algorithms work concurrently with marking, and do not block sending of new messages by the mutator.

4.1.1 Message Counting Method

ACKs are messages that are sent to the sender node to confirm the arrival of mutator messages. However, naive employment of ACKs would double the number of messages sent by mutators during the entire computation, and furthermore, all the mutator messages would have to contain the sender node ID, possibly increasing the cost of message sending †.

The message counting method eliminates the need of ACKs. All the nodes count the number of outgoing and incoming mutator messages, but no ACKs are sent. During global GC, the GC host watches the sum of the number of sent and received old messages in the system, in order to confirm the arrival of all the old messages. To distinguish the old messages from mutator messages sent after the start of global GC (called *new messages*), we assign a color to each mutator message. Nodes have a common color that changes at the initiation of each round of global GC, and all the mutator message are given this color. Each node distinguishes the number of sent or received messages according to their respective colors. The aggregate number of sent and received messages in the system should be equivalent for a given color.

Each node sends the number of sent old messages to the GC host at the initiation of global GC. The GC host will know the number of all the sent messages of a particular color in the system when it receives the information from all the nodes. Then, each node sends the number of received old message to the GC host when it has no objects to mark for a certain period of time. If an old message arrives afterwards, the local collector resumes marking from it, and sends the number of newly received old messages to the GC host again when it has no more objects to mark. The GC host will have confirmed the arrival of all the old messages when the sum of the number of all the sent messages and the sum of the number of all the received messages received from the nodes are equal. When marking terminates, i.e. the above three conditions are satisfied, all the nodes will have received their old messages, and the GC host can confirm this fact in finite time if old messages are guaranteed to arrive at their destination in finite time, otherwise there could still be unprocessed old message in the network,

In practice, one bit is sufficient as a color bit since executions of global GCs do not overlap at any given time. The GC host and the mutators flip the color bit per each round of global GC. Local collectors recognize mutator messages that do not match current color as old messages.

In comparison, naive usage of ACKs would require all the mutator messages to contain the sender node IDs. Moreover, even if ACK information were to be stored/combined, and exchanged only during global GC, ACK information would have to be exchanged between all the node pairs leading to a $O(n^2)$ algorithm where n is the number of nodes. In contrast, the message counting method sends messages for arrival confirmation only during global GC, and more importantly, the number of messages required for global arrival confirmation is reduced to $O(n)$.

4.1.2 Bulldozing Method

The bulldozing method uses hardware-assisted network clearing operation that can be used in networks that guarantee message FIFOness [8]. It does not block sending of new messages. *Bulldozing messages* are sent through all the network paths of the system, and arrivals of bulldozing messages through a given path guarantee the arrival of old messages through that path. Note that some new messages sent after the initiation of global GC but sent before the passage of the bulldozing message may arrive before bulldozing completes, but this fact does not affect the correctness of global GC (Fig.3).

No actual global GC implementations on MPPs have used bulldozing to our knowledge, however. In [8], an algorithm is given such that the nodes store and forward bulldozing messages $O(n^{1/2})$ times for each particular path, and $O(n)$ times in the system. Thus the latency of processing bulldozing messages is high for a large n. On the other hand, if bulldozing messages are sent between all node pairs, store-forwarding would not be necessary resulting in O(1) latency, but message traffic becomes $O(n^2)$.

In our implementation on AP1000, we send bulldozing messages to sweep out old messages, but reduce message traffic and latency by taking advantage of the physical network topology. Nodes store and forward bulldozing messages 6 times for each particular path for an arbitrary n, and $8 \times n$ bulldozing messages are in the system in total. Bulldozing is

†Increase in message size is not of importance here; rather, we are concerned with runtime overhead caused by fetching the node ID, sending it onto the network, receiving and processing ACK messages, etc.

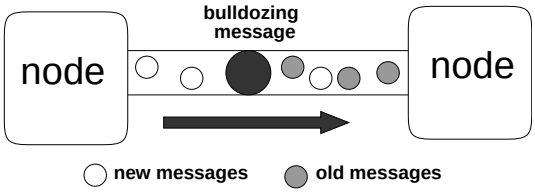

Figure 3: Bulldozing

done concurrently with marking.

Only messages that arrive before announcement of bulldozing termination can be old. Thus, we do not need to color mutator messages, as the arrival of the bulldozing message guarantee that the old message have been processed. As a result, we totally eliminate the runtime overhead in mutator message passing (no counting/coloring is required), although we might recognize a small fraction of new messages as old and mark from them (Fig.3).

4.2 Detection of Quiescent State

Another problem of marking in a distributed environment is the termination detection of marking with low message traffic.

Many global GC algorithms return an acknowledgement message [‡] per every mark message to indicate the arrival of the mark message, or the termination of marking from it. Thus, the number of messages required for termination detection would be more than twice the number of mark messages.

In order to reduce this cost, we use *distributed weight counting* to detect quiescent state of marking. Mark messages and marking local collectors are given positive weights by the GC host. The GC host finds the quiescent state of marking when all weights are returned to the GC host.

The GC host gives positive weights to all the local collectors at the initiation of global GC. Local collectors send mark messages with positive weight dividends. A local collector that receive a mark message adds the weight of the message to its own current weight(Fig.4). The invariant is that the sum of the weights of mark messages and local collectors are equal to the weight that the GC host gives out.

Each local collector returns its weight to the GC host if it has no objects to mark for a certain period(Fig.5). If all the weights are returned to the GC host, the GC host can judge that the system is in the quiescent state of marking.

The GC host judges the termination of marking when it confirms the arrival of all the old messages and the marking state is quiescent.

[‡]Not to be confused with ACKs, which are acknowledgements to *mutator* messages.

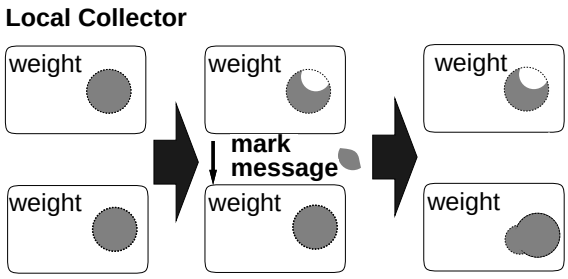

Figure 4: Marking Messages with Weight

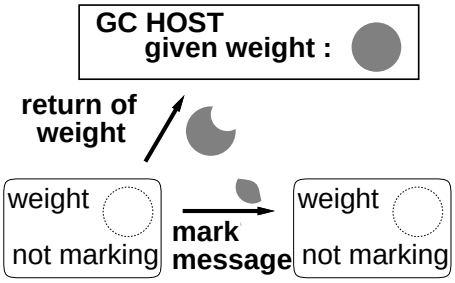

Figure 5: Returning of Weight

One exceptional case to consider is as follows: because our concurrent global collector marks from lost references to unmarked objects caused by mutator writes, it is conceivable that after the local collector has returned all its weight to the GC host, a mutator write could resume marking in the node, while the GC host would judge the termination of marking and wrongly start the reclamation phase. However, such cases are guaranteed not to occur. The usual correctness criterion of the local concurrent collector guarantees that all locally referenced live objects are marked and the weight of a node is not returned until such local markings are finished. Thus, such unmarked old object must either be referenced from old messages or other nodes, and as a result, when the arrival of all the old messages have been confirmed, there is a node or a mark message in the system whose weight is positive. Since the GC host does not start reclamation until the arrival of all the old messages are confirmed and all the weights are returned to the GC host, no such exceptional case could occur. On the other hand, if marking terminates, all the weights will be returned to the GC host in finite time, and the GC host will certainly judge termination of marking.

Comparing with methods that return acknowledgement for each mark message, our method only needs to communicate infrequently with the GC host to return/request weight that is managed by each local collector as a unit, and it requires substantially less message traffic. On the other hand, our method could result in concentration of weight messages to the GC host, but experimental results so far show that this is

not a problem because weight message traffic is very infrequent compared to other types of messages.

4.3 Optimizations

We can further optimize our algorithm to reduce messages traffic to the GC host. Our implementation on AP1000 currently incorporates the following optimizations:

- For the message counting method, we can combine the messages that indicate the number of received old message and that return weight into one message to the GC host.
- Each local collector could delay the returning of weight for a certain time period after local marking terminates, letting the mutator execute in the node. In our global GC, it is sometime the case that, immediately after returning the weight, an old messages or a mark message would arrive and marking must be resumed. By incorporating such a delay, we can reduce such cases and reduce messages to the GC host. In a similar way, a local collector can delay returning of weight for a certain period of time from the initiation of global GC. This technique also reduces messages to the GC host, with a tradeoff that it could set an unnecessary lower bound on global GC time.

5 The Algorithm

Altogether, we can formalize our global GC algorithm as follows:

5.1 In Local Collectors

Ordinary Computation Phase If a marking message or an announcement of initiation of global GC (or a bulldozing message in case of bulldozing method) arrives, the node starts the marking phase.

Marking Phase A local collector marks concurrently using a marking stack. At the initiation of marking phase, the marking stack initialized to contain global variables, local active objects, and their unprocessed messages. For the message counting method, it sends the number of sent old messages to the GC host.

Local collectors examine received messages during the marking phase, and behave as follows depending on the message kind:

- Marking messages
  The reference information in the marking message is added to the marking stack.
- (Possibly) old mutator messages
  The node adds reference information in old messages to the marking stack.
  - For message counting method, old-colored messages are judged as old.
  - For bulldozing method, messages that arrived before the announcement of termination of bulldozing are judged as old.
- Announcement message of marking termination
  Each node starts to reclaim their unmarked areas, and return to the ordinary computation phase.

Local collectors send messages to the GC host for termination detection of marking, as follows:

- For the message counting method, when a node has no objects to mark in its marking stack for a certain period, and a given time has passed since the the initiation of global GC, the node returns its weight and the number of received old messages to the GC host.
- In case of bulldozing method, when a node has received the announcement message of termination of bulldozing [§], and it has no objects to mark in its marking stack for a certain period, and a given time has passed since the the initiation of global GC, the node returns its weight.

5.2 At the GC Host

Ordinary Computation Phase The GC host decides the initiation of global GC based on the requests from nodes.

Marking Phase The GC host determines termination of marking based on the information from each local collector, upon which it sends messages to all the local collectors to release unmarked memory.

- For the message counting method, the GC host determines that marking has terminated if the following three conditions are satisfied.
  - It has received the number of sent old messages from all the nodes.
  - The sum of the number of all the sent old messages and the sum of the number of all the received old messages are equal.
  - All weight has been returned to the GC host.

[§] For the bulldozing operations, all nodes are announced the termination of bulldozing.

- For the bulldozing method, the GC host determines termination of marking if following two conditions are satisfied.
  - Bulldozing operations have been terminated.
  - All weight has been returned to the GC host.

6 Performance Evaluation

We have implemented a prototype of our global GC algorithm on a 32-1024 nodes Fujitsu AP1000 MPP (each node consists of 25MHz SPARC and 16MB memory). In order to evaluate the performance, we employed a fine-grain, naive combinatorial N-Queen search problem (Table1,Table2,Fig.6). The heap size was fixed to 8MB per each nodes.

method for arrival confirmation	message counting	bulldozing
# of gc messages excluding mark messages: A	410	640
# of mark messages: B	13700	15700
A / B	3.0 %	4.1 %
marking time(msec)	190	190
# of gc messages to host	410	210
# of marked objects ($\times 10^5$)	6.7	6.6
# of garbage objects ($\times 10^5$)	27.8	25.3
percentage of marked objects	19 %	21 %

Table 1: Performance Evaluation of Global GC (12-Queens,64-nodes)

method for arrival confirmation	message counting	bulldozing
# of gc messages excluding mark messages: A	3700	9200
# of mark messages: B	24000	23000
A / B	15 %	40%
marking time(msec)	170	190
# of gc messages to host	3700	2100
# of marked objects ($\times 10^5$)	1.5	1.3
# of garbage objects ($\times 10^5$)	34.3	32.8
percentage of marked objects	4.4%	3.8 %

Table 2: Performance Evaluation of Global GC (13-Queens,1024-nodes)

The number of messages for global GC excluding mark messages, i.e. weight return/request or bulldozing messages (row A), is 3 $\sim$ 10% of the number of mark messages (row B), when the number of remote references are substantially larger than the number of the nodes in the system by orders of magnitude

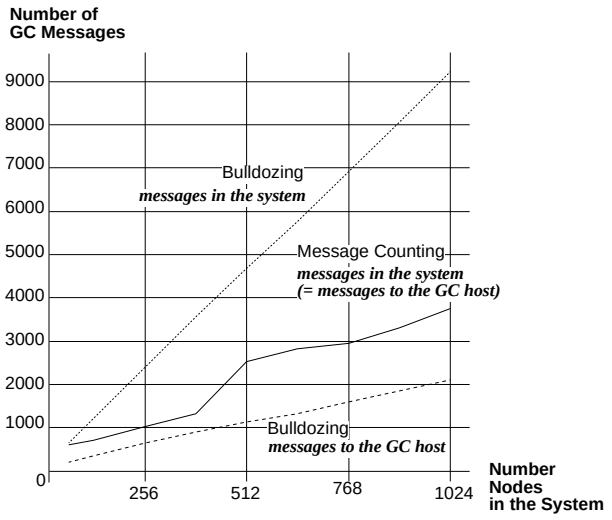

Figure 6: The Number of GC Messages Excluding Mark Messages (13-queens)

(Table1). In other words, the total number of messages for global GC is a small fraction greater than the number of remote references. In the case of acknowledging every mark message, the number of message for global GC would be more than twice the number of remote references. Our method reduces message traffic of global GC by about half.

Both the message counting method and the bulldozing method terminates marking in about 100$\sim$300 msec. Furthermore, since mutators are concurrently executed during that time, the latency involved in remote marking is effectively hidden. The only case the mutator becomes blocked is when the node phisically runs out of memory during this period.

Comparing the message counting method with the bulldozing method, the bulldozing method involves more message traffic than message counting method (Table1,Table2). This is because the latter uses bulldozing messages exclusively for arrival confirmation, while message counting method can combine messages for arrival confirmation and for detection of quiescent state. In addition, whether or not effective bulldozing algorithm would be possible depends on the network architecture of the underlying hardware; for example, most adaptive routing networks does not guarantee message FIFOness and thus precludes bulldozing. On the other hand, the bulldozing method has the advantage that it would involve little mutator overhead as it does not have to deal with runtime color infromation or message counting.

In this benchmark, the number of messages to the GC host are small, and thus message concentration on the GC host does not become a problem. More specifically, the number of messages to the GC host is 3$\sim$20 times as many as the number of nodes, and

it can be reduced to 3∼5 times by postponing returning of weight(Table1,Table2,Fig.6). In our performace evaluation with 1024 nodes AP1000, the number of the GC messages to the GC host was less than 4000, and that did become a bottleneck in our benchmark. However, concentration of messages to the GC host would become higher and could potentially cause problems if the number of nodes increases by orders of magnitude. In such a case, improvements will be needed such as creating a hierarchy of GC hosts.

In our implementation on AP1000, messages for global GC cannot be given priority over mutator messages due to hardware restriction, and as a result, latency of receiving global GC messages tend to be longer, resulting in longer marking time and thus potentially increasing the chance of mutator block. If global GC messages can be processed with higher priority over mutator messages, global GC would take shorter time, and we can exploit more effective variations of the algorithms.

7 Conclusion and Future Work

Our parallel global GC algorithm reduces the runtime overhead and message traffic for global GC significantly. The number of messages for termination detection of marking is about 3∼10% of the number of mark messages. This implies our global GC needs less than half the message traffic compared to previous algorithms.

Comparing the two alternatives of old message arrival detection, the bulldozing method and the message counting method, the bulldozing method has the advantage that it does not require coloring or counting of messages, but its viability depends on the underlying hardware, and latency of bulldozing could be longer as a result.

In our implementation on AP1000, processing of global GC message is not given priority over mutator messages. If global GC messages could be given higher priority over mutator mesages, global GC would take lesser time, and thus significantly reducing the chance of mutator block on each node.

As a future work, since our implementation is still a prototype version, we plan to evaluate realistic performances in various parallel program benchmarks such as SPLASH[7]. We also would like to extend system so that it would have beter scalability for systems with over thousands of nodes, by making the GC host and the arrival/termination detection algorithms to be hierarchical. Finally, our global GC needs to be extended to consider object migration.

Acknowledgements

We would like to express our thanks to Fujitsu Laboratory for allowing us to use AP1000(1024 nodes version). Finally, we thank the member of the ABCL project group at our laboratory for their advice and discussions.

References

[1] Lex Augusteijn. Garbage collection in a distributed environment. volume 259 of *LNCS*. PARLE, 1987.

[2] D I Bevan. Distributed garbage collection using reference counting. volume 259 of *LNCS*. PARLE, 1987.

[3] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. ACM Transactions on Computer Systems, 2 1988.

[4] Bernard Lang, Christian Queinnec, and Jose Piquer. Garbage collecting the world. POPL, 1992.

[5] Niels Christian Luul and Eric Lul. Comprehensive and robust garbage collection in a distributed system. In *Memory Management*, volume 637 of *LNCS*, 1992.

[6] Marcel Schelvis. Incremental distribution of timestamp packets: A new approach to distributed garbage collection. OOPSLA, October 1989.

[7] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Standard Parallel Applications for Shared-Memory.

[8] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable distributed garbage collection for system of active objects. In *Memory Management*, volume 637 of *LNCS*, 1992.

[9] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. volume 259 of *LNCS*. PARLE, 1987.