(1) 89/12/2 SSE研究会 於:ASTEM

Generation Scavenging GC on Distributed-Memory Parallel Computers

Hanpei Koike and Hidehiko Tanaka
{koike,tanaka}@mtl.t.u-tokyo.ac.jp
Dept. of Electrical Engineering, The University of Tokyo,
Hongo 7-3-1, Bunkyo, Tokyo, 113 JAPAN

1 Introduction

High expressive power of symbol processing languages mainly comes from their ability to allocate data dynamically and freely. Automatic storage management mechanism such as garbage collection (GC) is a fundamental technique which is indispensable for this symbol processing. On the other hand, parallel computer systems, especially large scale ones, need to adopt distributed memory configuration. Therefore, in order to attain high performance in symbol processing on such machines, it is necessary to develop an efficient GC algorithm for distributed memory environment.

In this paper, a garbage collection method for distributed-memory parallel computers is discussed. At first, it is stressed that the lifetime of objects is an important notion especially for garbage collection on distributed memories. Then, parallel generation scavenging GC method, in which efficient GC is realized using the notion of the lifetime of objects, and pseudo-realtimeness is realized as well, is proposed, and its implementation method is discussed.

2 Garbage Collection for Parallel Computers

In this paper, a garbage collection method for distributed-memory parallel computers is discussed with the following assumptions:

- Symbol processing is performed on a distributed-memory parallel computer which consists of hundreds of high performance processing elements (PE) connected by a high performance interconnection network(Fig.1). Data objects are allocated to distributed heap memories and reference each other.

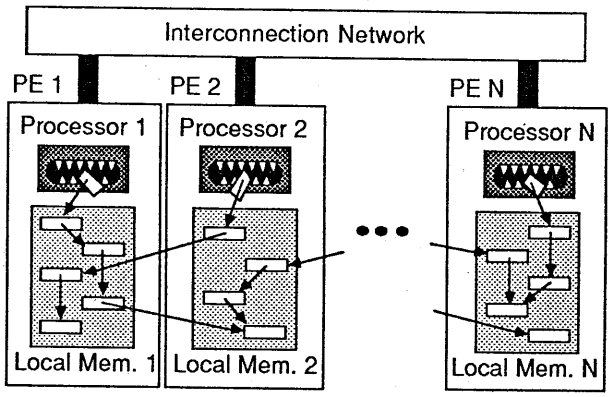


Figure 1: Parallel Symbol Crunching

- Certain amount of remote references are produced because of frequent dynamic load distribution.
- Performance of communication mechanism between processing elements is high.
- Variable-size object such as vector can be used.

Symbol processing on a large scale parallel computer requires high memory band width, because each processor accesses symbol data on its memory frequently among a large working set. Therefore, distributed memory configuration is necessary.

Symbol processing application programs like knowledge information processing behave rather

irregularly and dynamically. Although it is important to make an effort to get locality and to reduce communication overhead, dynamic load distribution and dynamic data allocation among processing elements are necessary as well to achieve high parallelism and high performance. This results in a large amount of remote references of data between processing elements. Thus, high communication capability is necessary.

Variable-size object such as vector is necessary to develop practical application programs. This condition requires a method for reclaiming memory area of variable-size garbages. Compaction by moving objects and pointer relocation is necessary.

The goal in our discussion is to minimize both the overhead of memory management and the overhead of normal symbol processing caused by memory management on the conditions described above. Of course, it is also important to reclaim memory area and to reuse it as much as possible to reduce the overhead, if the reclamation time can be known at compile time by a static analysis of programs [2].

Several garbage collection methods have been proposed for single-processor computers so far [3]. On the other hand, a garbage collection method for distributed-memory parallel computers requires the following additional conditions:

- Efficient manipulation of pointer references among distributed memories is necessary.
- Performance improvement in memory management is required as the performance of the normal processing improves by parallel processing.

The first problem of garbage collection on distributed memories is the efficient manipulation of pointer references among distributed memories. The following problems must be solved:

- How can we know that a data object is referenced by a pointer in a *remote* memory and that it is *not* a garbage?
- How can we reflect the change of the object address on all the *remote* pointers which reference it, when the object is relocated for compaction?

The second problem is the performance of memory management. As the number of the processors increases and the performance becomes higher, memory consumption also becomes faster. In order to avoid relative increase in the memory management overhead, the performance of memory management must be improved corresponding to the performance improvement of the normal processing by parallel processing. The following problems must be solved:

- How can we attain high parallelism in memory management itself corresponding to the number of processors?
- How can we reduce the additional overhead caused by parallel processing such as communication and synchronization?

3 Global GC and Local GC

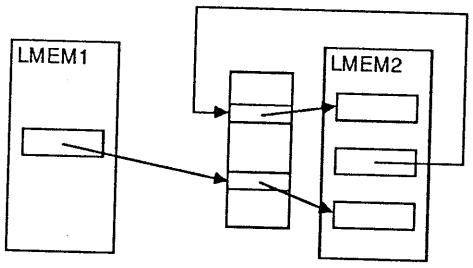
The garbage collection method of distributed memories can be categorized into the following two methods:

- Global Garbage Collection
- Local Garbage Collection

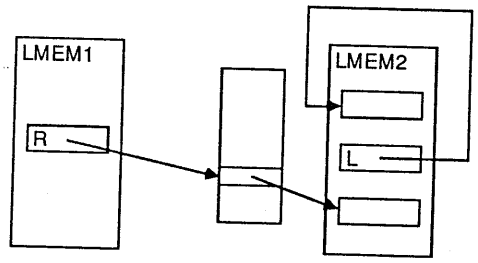
In the global garbage collection method, when a processing element runs short of memory, all the processing elements stop their processing, and start garbage collection at the same time. In the local garbage collection method, only the processing element which runs short of memory executes garbage collection on its own local memory asynchronously, while other processing elements continue their normal processing.

Defects of the global garbage collection method are:

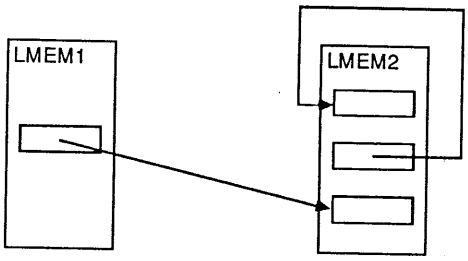
- The interval of the global garbage collection is determined by the first processing element which runs short of memory. The rest of the processing elements must join global garbage collection even if they have enough amount of free memories.
- High parallelism must be achieved in the global garbage collection itself to correspond to the performance improvement in the normal processing.



(a) All the references are indirected.



(b) Only remote references are indirected.



(c) No reference is indirected.

Figure 2: Reference Management Methods

- There exists overhead of synchronization of all the processing elements, when global garbage collection starts and finishes. This overhead is especially substantial, when realtime garbage collection is adopted and the stop time is short.

On the other hand, if the local garbage collection method is adopted, it requires *reference management table* as in Fig.2 in order to determine whether the object is still necessary or not only from the *local* information, and to hide the change of the object address from outside the processing element when the object is relocated for compaction.

Reference management table can be used in the following two ways:

- All the references are indirected through reference management table entries (Fig.2(a)) [4].

- Pointer references are separated into local references and remote references, and only the remote references are indirected through reference management table entries (Fig.2(b)) [5].

However, each method has the following defects respectively.

When all the references are indirected as in Fig.2(b),

- even local references are slowed down because of indirection.

This will result in a substantial performance decline of each processing element, because memory access is very frequent in symbol processing.

In order to solve the problem described above, you can separate local references and remote references, and the table is used only for remote references as in Fig.2(b). However, this method also has the following defect:

- Conversion of remote reference to/from local reference is necessary every time transfer of object containing pointers occurs between processing elements. Especially, in order to maintain uniqueness of remote pointer, the conversion from a local pointer to a remote pointer requires an associative operation such as hash, and will be a substantial overhead.

This will result in a decline of communication performance.

In addition to the defects above, the local garbage collection methods in general have other defects such as:

- Another scheme is necessary to reclaim reference management table entry anyway. This will lead to an additional overhead.

- When a processing element starts the local garbage collection, it is difficult for other processing elements to access the data in this processing element. Therefore, the local garbage collection can lower the operating ratio of other processing elements.

- Reclamation of remotely-referenced data is postponed until the next local garbage collection after the reclamation of reference management table entry. Garbage is treated to be necessary during this period. This will lead to greater overhead of garbage collection and shorter interval of garbage collections.

As we can see from the discussions above, the local garbage collection method, which requires a reference management table, has defects such as data transfer overhead and complexity of processing. They would lose the flexibility of parallel processing, which is particularly essential for symbol processing.

Several implementation techniques for memory reusing have been proposed [2, 6]. They reduce memory consumption rate and improve efficiency of memory usage. They would make the importance of memory management as compared to normal processing relatively low. This view also contradicts the local garbage collection method, which puts a penalty on the performance of the normal symbol processing itself for the memory management.

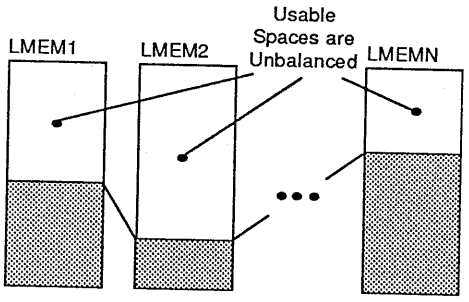
On the other hand, if a global garbage collection method, without *any* reference management table as shown in Fig.2(c), can be invented, and if its defects described earlier can be solved, simplicity of processing and substantial reduction of data transfer overhead will be achieved, and it is much necessary for flexible symbol processing. In the next section, the solution of the problems of the global garbage collection described earlier will be examined.

4 Improvement of Global Garbage Collection

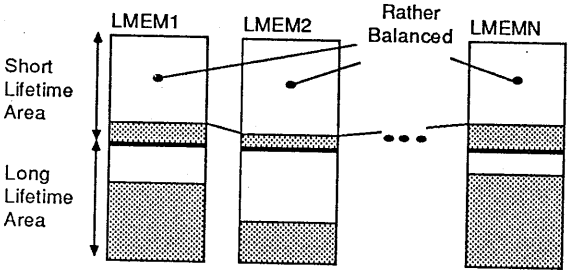
The goals in solving the problems of the global garbage collection are as follows:

- Each processing element must run short of its memory as simultaneously as possible.
- High parallelism must be attained.
- Synchronization overhead must be small.

In addition to these goals, the knowledge information processing may be rather interactive, therefore



(a) In Ordinary GC



(b) In GC based on Lifetime

Figure 3: Deviation of Usable Spaces

- realizing realtime garbage collection

is also an important issue. We will discuss how these goals can be achieved below.

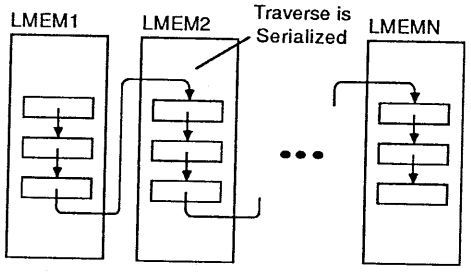
The first goal is to balance the time to run short of the memories among the processing elements. The deviation comes from two reasons:

- deviation of the amount of usable memory
- deviation of memory consumption rate

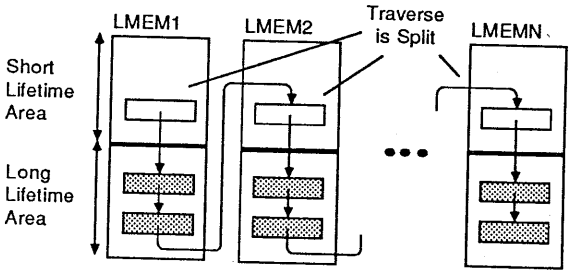
Since the latter can be expected to be rather balanced for a long term between garbage collections, the former would affect mainly.

If the garbage collection method based on the lifetime of objects is adopted and short-lifetime objects are specially treated, the ratio of the used memory area to the usable memory area becomes relatively small. Especially, if a fixed-size allocation area is provided, usable memory areas are completely balanced. As a result, the deviation of the time to run short of memory among processing elements is expected to be balanced (Fig.3).

The second goal is to attain high parallelism in global garbage collection. The basic operation of



(a) In Ordinary GC



(b) In GC based on Lifetime

Figure 4: Improvement in Parallelism of GC

garbage collection is traverse of live objects. One of the reasons why parallelism of global garbage collection would be suppressed is that there exists a long chain of data such as a semantic network along the processing elements, which serializes the traverse. However, the most part of such data is expected to have relatively long lifetime. Therefore, if a garbage collection method based on lifetime is adopted, the whole traversal of large data becomes rare, and thus the parallelism in the global garbage collection can be improved (Fig.4).

Synchronization overhead between all the processing elements can be reduced by using dedicated hardware and signal lines for synchronization. This would improve the performance especially when a realtime global garbage collection method is adopted.

In summary, the key point to solve the problems caused by global garbage collection is the introduction of lifetime of the objects into the garbage collection method. If realtime garbage collection could be realized, it would be much better. In the next section, such a garbage collection method will be examined.

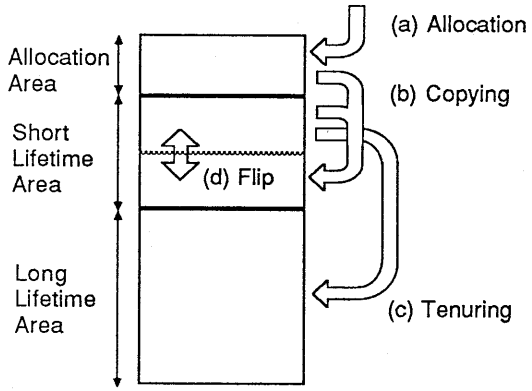


Figure 5: Generation Scavenging GC

5 Generation Scavenging GC

Generation scavenging GC [7] is a garbage collection method in which GC overhead is reduced by introducing the notion of the lifetime of objects, and pseudo-realtimeness is realized by limiting GC area and making the stop time short enough.

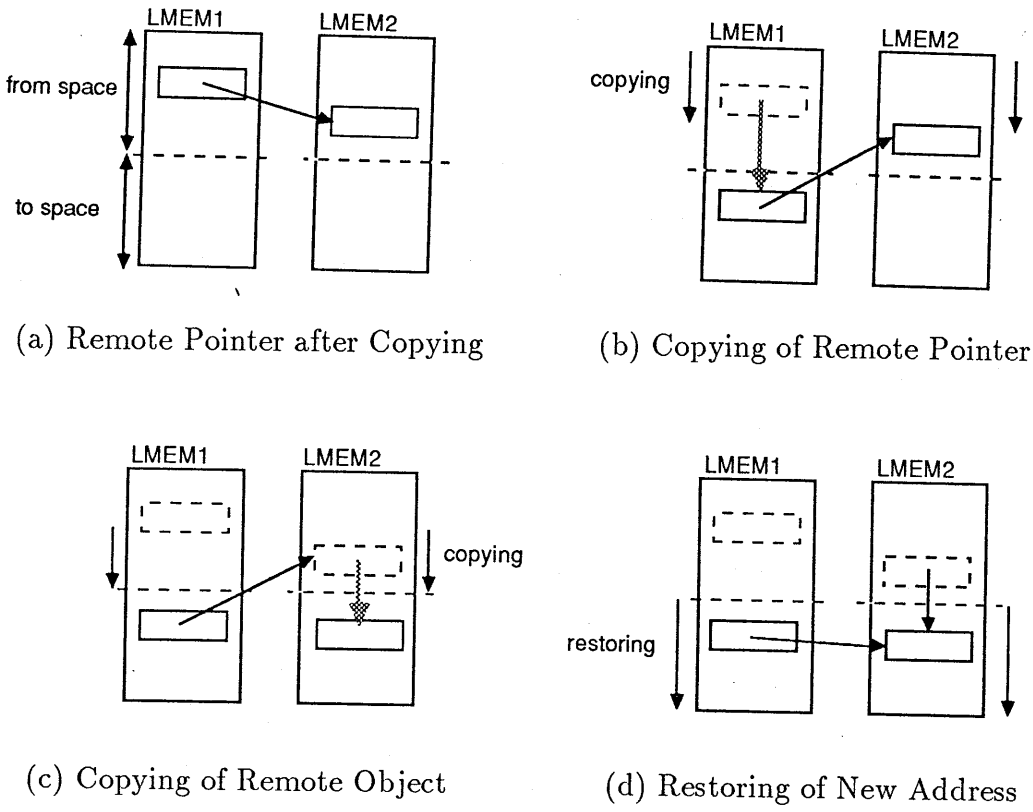
In this method, the heap area is split into four areas: an allocation area, two short-lifetime areas, and a long-lifetime area (Fig.5).

A new object is allocated onto the allocation area (a). When symbol processing runs short of the allocation area, only live objects in the allocation area and the current short-lifetime area are copied onto the other short-lifetime area (b). In addition to this, objects which remain alive for several times of garbage collection are assumed to have long lifetime, are moved to the long-lifetime area, and are no longer the subject of garbage collection (c). When copying finishes, the two short-lifetime areas exchange their role (d), and normal symbol processing resumes.

Since only live objects in the short-lifetime area are copied, the stop time caused by GC can be small enough. Thus, realtimeness is realized practically.

Neither objects in the long-lifetime area nor objects referenced by them are traversed any more. If a object in the short-lifetime area is referenced by a pointer in a object in the long-lifetime area, such a pointer must be remembered and treated as a root to avoid reclaiming the short-lifetime object area incorrectly.

Since the allocation areas of each processing



(a) Remote Pointer after Copying

(b) Copying of Remote Pointer

(c) Copying of Remote Object

(d) Restoring of New Address

Figure 6: Copying Operation on Distributed Memories

element have the same fix size, deviation of the time to run short of memory will be minimized. Since long-lifetime objects are no longer subject of the garbage collection, serialization of traverse will be improved. Finally, pseudo-realtime garbage collection can be realized.

In the following sections, implementation of generation scavenging GC on distributed memories will be discussed. First, realization of copying operation on distributed memories is presented. Then, management method of root pointers among distributed memories is presented.

6 Copying GC on Distributed Memories

Copying GC is the basic operation in generation scavenging GC. The copying GC method on distributed memories is presented in this section.

In general, ordinary copying GC is inefficient in memory usability, because the whole memory space is split into *from* space and *to* space, and only half of the memory space can be used. However, memory usability is improved in generation scavenging GC, because only the short-lifetime area needs to be split.

Copying GC has the property that the time is proportional to the amount of live objects only, and is independent of the amount of the garbage. This is an advantage for the GC method of the short-lifetime area, where most of objects become garbage immediately.

The copying operation among distributed memories consists of two phases: (1) copying phase, and (2) restoring phase. All the processing elements transit each phase synchronously. Since the operation is split into two phases, fine grained synchronization is unnecessary. This leads to both reduction of overhead and high parallelism. Each phase only takes time proportional to the amount of live objects.

In the copying phase, all the processing elements copy live objects in their own *from* space to their *to* space in parallel. Forwarding address is put at the top of the copied object with mark bit set to indicate that it has already been copied. When a remote pointer appears, a copy message including the remote pointer is sent through the network to request copying to

the destination processing element (b). When a processing element receives the message, the pointer is treated as a root, and the remote object is copied (c).

In the restoring phase, all the processing elements scan the roots and the copied objects in their own local memories in parallel to find remote pointers. When a remote pointer appears, the forwarding address is fetched through the network, and is written back to the remote pointer (d).

Using the operation described above, copying GC among distributed memories can be realized without *any* reference management table. As a result, both the memory access and data transfer in normal processing are very simple and fast.

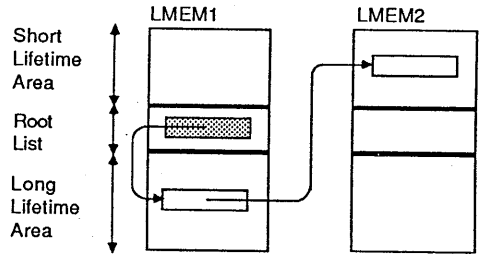
In addition to the basic copying operation above, the following two extensions are required to realize the generation scavenging GC. At first, objects in the long-lifetime area need not to be copied. Secondly, objects which remain alive for several times of garbage collection are assumed to live longer, and they are moved to the long-lifetime area. Since both of the extensions are local operations, no new concern is necessary to realize them in distributed memory environment.

7 Management of Root Pointers

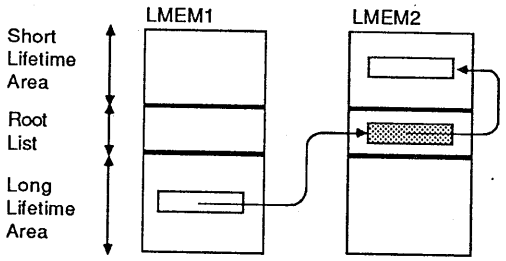
Another point in realizing generation scavenging GC on distributed memories is a management method of the root pointers among distributed memories.

In generation scavenging GC, neither objects in the long-lifetime area nor the objects referenced by them are traversed. When an object in the short-lifetime area is referenced by an object in the long-lifetime area, such a pointer must be recorded and treated as a root to avoid reclaiming this object area incorrectly. Some processors [8, 9] adopt a hardware mechanism to detect such pointer storing. In addition to this, this pointer must always be refreshed to the new address whenever a short-lifetime object is moved.

In distributed memory environment, the memory where the detected root pointer is recorded is a problem to be concerned. If a short-lifetime object and a long-lifetime object, into which a pointer to the short-lifetime object is stored, reside in different distributed memories, possible solutions that meet the requirement described



(a) With Long-Lifetime Object



(b) With Short-Lifetime Object

Figure 7: Management of Root Pointers

above are the following two ways:

- The address of the pointer in the long-lifetime object is recorded in the local memory in which the long-lifetime object resides.
- The address of the short-lifetime object is recorded in the local memory in which the short-lifetime object resides, and the reference is indirected.

These two methods are compared below.

In the former method, the registration of a root pointer is done locally when such pointer storing is detected. In the latter, the registration requires network communication, because the detection and the registration occur in different processing elements in general.

In the former method, address change of the remote short-lifetime object can be reflected on the pointer directly in the restoring phase. Since it can be known that the pointer no longer references a short-lifetime object as well at the same time, such a root can be removed from the root set to avoid monotonous increase in the root set size. In the latter method, reference must be indirected to relocate the short-lifetime object

locally, and the access cost becomes higher. In addition to this, the size of the root set increases monotonously, because it is difficult to remove an unnecessary root from the root set.

Network traffic caused by the root pointers is compared. In the former method, network traffic occurs on every GC to request copying of the remote short-lifetime objects pointed by long-lifetime objects. The traffic repeats to occur as long as the referenced objects reside in the short-lifetime area. In the latter method, network traffic occurs only once to register a root pointer to a remote processing element when the pointer storing is detected.

In the former method, no new communication primitive to register a root pointer is necessary. In the latter method, a new primitive to register a root pointer is necessary. Therefore, it is easier to realize the former method.

As a result of the discussions above, although the former method causes network traffic on every GC as long as the pointer references a short-lifetime object, it seems to be advantageous, because registration can be performed locally, unnecessary root pointer can be removed from the root set, and the implementation is easier.

8 Support from the Communication Mechanism

A high performance communication mechanism with highly functional communication primitives, which meet the subject of the processing, is a very important issue in designing a parallel computer. The following communication primitives are suitable to support the garbage collection method presented so far effectively.

In order to request a remote processing element to copy an object on its local memory,

```
copy(remotepointer)
```

is used, where `remotepointer` is an address of an object on remote memory. A copy request message is sent to the destination processing element. When a remote pointer appears in the copying phase, this primitive is executed.

In order to restore a new address of a remote object to a pointer in the local memory,

```
restore(remotepointer,location)
```

is used, where `remotepointer` is an address of an object on a remote memory, and `location` is an address of a remote pointer in the local memory. Forwarding address of an object which is set at the top of the object is fetched through the network, and is written back to the pointer in the local memory. When remote pointer appears in the restoring phase, this primitive is executed.

The Network Interface Processor (NIP) [11] of the Parallel Inference Engine PIE64 [10] which we are currently developing has these communication primitives. Copy command can be executed in 16 clocks, and restore command can be executed in 15 clocks in parallel with the main processor, if network contention does not occur.

9 GC of the Long-Lifetime Area

Long-lifetime object area is not reclaimed even if they become garbage. The long-lifetime area is consumed monotonously and will run short of memory sooner or later. Therefore, another scheme to collect garbage in the long-lifetime area is necessary. Mark & sweep type compaction GC for distributed memories [13], where no other working space is required because of the reversal pointer technique, and cyclic structures can be reclaimed as well, is an example of such a scheme.

10 Moving Strategy to the Long-Lifetime Area

Objects which remain alive after several times of garbage collection are moved to the long-lifetime area, and are no longer the subject of garbage collection. This strategy is based on the assumption that such objects would live further long. However, if such a object becomes a garbage immediately after being moved to the long-lifetime area, neither this object nor the objects referenced by it can be reclaimed, and the performance of garbage collection becomes substantially low. In order to avoid such a situation, the moving strategy to the long-lifetime area must be determined carefully, based on the nature of the object, the nature of the program, and the nature of the language [15].

11 Concluding Remarks

In this paper, a generation scavenging garbage collection method on distributed-memory parallel computers is proposed and discussed.

First, global GC and local GC are compared as a GC method on distributed-memory parallel computers, and the problems of reference management table, which is indispensable for the local GC method, are pointed out.

Secondly, the problems of the global GC method are examined, and it is shown that an introduction of the object lifetime will be a solution to them.

Thirdly, generation scavenging GC is introduced, where overhead of GC is reduced by making use of the object lifetime, and pseudo-realtime GC is possible as well. Its implementation on distributed-memory parallel computers is presented in the order of (1) the implementation of copying GC on distributed memories, and (2) the management method of the root pointers.

Finally, support of communication mechanism, GC method of the long-lifetime area, and the moving strategy to the long-lifetime area are discussed. We are planning to implement and to experiment this generation scavenging GC method on our Parallel Inference Engine PIE64.

The problems discussed in this paper are a kind of problem which will be apparent in the execution of the large scale programs with complicated structure. A simulation study using small benchmark programs is not enough for precise quantitative discussions on such a problem. Therefore,

- evaluation on a real machine using real application programs

will be an important future work for further quantitative discussions. Based on such researches,

- effective moving strategy to the long-lifetime area

will be determined.

The copying method proposed in this paper is *intra* processing element copying. Extending the copying method to *inter* processing elements copying, combining the GC method with load distribution and data distribution, which are also important themes in parallel processing research, and realizing

- the optimal object relocation among processing elements by object migration on GC

are other important future works.

This work is supported by Grant-in-Aid for Specially Promoted Research of the Ministry of Education, Science and Culture (No.62065002).

References

[1] Koike, H., Xu, L. and Tanaka, H.: *Garbage Collection on Distributed Memories*, 6th Conference Proceedings Japan Society for Software Science and Technology, C5-1, Oct., 1989. (in Japanese)

[2] Koike, H. and Tanaka, H.: *Optimizing Compilation of Logic Programs using Single Reference Annotation*, 38th Annual Convention of IPSJ, 6Q-1, March, 1989. (in Japanese)

[3] Cohen, J.: *Garbage Collection of Linked Data Structures*, ACM Computing Surveys, Vol.13, No.3, 1981.

[4] Mohamed-Ali, K.A. and Haridi, S.: *Global Garbage Collection for Distributed Heap Storage Systems*, TRITA-CS-8502, Dept. of Computer Systems, Royal Institute of Technology, Stockholm, April,1985.

[5] Ichiyoshi, N., Miyazaki, T. and Taki, K.: *A distributed implementation of flat GHC on the Multi-PSI*, Proc. of the 4th International Conference on Logic Programming, 1987.

[6] Chikayama, T. and Kimura, Y: *Multiple Reference Management in Flat GHC*, Proc. of the 4th International Conference on Logic Programming, pp.276-293, 1987.

[7] Ungar,D.: *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm*, SIGPLAN Notice, Vol.19, No.5, May, 1984.

[8] Ungar,D. et al.: *Architecture of SOAR: Smalltalk on a RISC*, Proc. 11th Int'l Symp. on Computer Architecture, p.1887, June, 1984.

[9] Hill M. et al.: *Design Decisions in SPUR*, IEEE COMPUTER, pp.8-22, Nov., 1986.

[10] Koike, H. and Tanaka, H.: *Parallel Inference Engine PIE64*, in *Parallel Computer Architecture*, special issue of bit, Vol.21, No.4, 1989. (in Japanese)

[11] Shimizu,T., Koike,H., Shimada,K. and Tanaka,H.: *The Network Interface Processor of Parallel Inference Machine PIE64*, Proc. of JSPP'89, pp.99-106, IPSJ, Feb., 1989. (in Japanese)

[12] Morris, F.L.: *A Time- and Space-Efficient Garbage Compaction Algorithm*, CACM Vol.21, No.8, Aug., 1978.

[13] Xu,L., Koike,H. and Tanaka,H.: *Distributed Garbage Collection of the Parallel Inference Machine PIE64*, Proc. of the 11th Computer Congress, IFIP, Aug., 1989.

[14] Xu,L., Koike,H. and Tanaka,H.: *The Garbage Collection System for Parallel Inference Machine PIE64*, Proc. of NACLP 89, Oct., 1989.

[15] Ungar,D. et al.: *Tenuring Policies for Generation Based Storage Reclamation*, Proc. of OOPSLA'88, Sept., 1988.