Incorporating Locality Management into Garbage Collection in Massively Parallel Object-Oriented Languages *

Kenjiro Taura
Satoshi Matsuoka
Akinori Yonezawa
Department of Information Science,
Faculty of Science, The University of Tokyo †

Abstract

This paper discusses how *locality* between objects affects the performance, and proposes a software architecture for enhancing locality while keeping load-balance reasonable at the minimum sacrifice of runtime overhead. Objects are created locally by default and long-lived objects are selectively *migrated* during garbage collection. By enhancing locality, message passings are likely to be local and objects are likely to be referred to from only local objects, thus they are quickly reclaimed when becoming garbage. By integrating migration process into garbage collection, load-balance is achieved and information useful for migration (e.g., reference counting) are collected at a low cost during garbage collection.

1 Introduction

1.1 Why Locality is Important

When we spawn a new concurrent object (task), where should the new object be located? Should the object be created on the *local* node, i.e., on the same node where the creater object resides, or on a *remote* node, i.e., which is some other node where the creater object does not resides? In order to answer to this question, we should consider two issues and address the conflict between them.

- *Locality*, which refers to how likely a reference to an object points to an object on the local node.
- *Load-balance*, which means how evenly objects are distributed among nodes.

If we forget locality and concern only load-balance, the problem is easy; some randomized distribution scheme will suffice in practice. Unfortunately, this scheme creates too many *remote references*, resulting in severe loss in locality which causes a severe degradation of the overall performance by the following two reasons:

- *Message passing overhead/latency.* A remote message passing involves larger overhead than local message send in many multicomputers. This is especially true on current commercial multicomputers (e.g., AP1000) which consist of high performance, conventional RISC processors where the overhead of a local message passing can be reduced to a small factor of that of a procedure invocation.[9]
- *Difficulty of garbage collection.* In garbage collection environment for high-level languages such as concurrent object-oriented languages or functional languages, the problem is harder. We must reclaim the space used by *garbage objects*, which are no longer referred to from any objects. In order to reclaim an object, we must detect if the object is no longer referred to from remote nodes, as well as from local nodes. Existing garbage collection schemes for this purpose are roughly classified into the following two categories:
  - *Reference count*, which keeps how many references exist in the system relying upon corporation by the user process (mutator). Since reference counts must be handled when creating/copying/deleting a reference to an object (e.g., message passing, assignments), it imposes large overhead to the user process.

*超並列オブジェクト指向言語における局所性管理とガーベジコレクションの融合

†田浦健次朗 松岡聡 米澤明憲
東京大学理学部情報科学科
E-mail: {tau,matsu,yonezawa}@is.s.u-tokyo.ac.jp

- *Distributed mark-and-sweep*, which is similar to the mark-and-sweep garbage collection on single CPU, except for marking phase transmits a message (mark message) when it finds a remote reference. Since this scheme generates a large amount of network traffic for a garbage collection, it works well *only if it is rare*, otherwise the overall performance of the system will be significantly degraded by the messages for garbage collection.

1.2 Overview of Our Approach

Based on the above observation, we devised a scheme which maximizes locality while keeping load distribution reasonably balanced. Our main concern is how to do this with minimum sacrifice of runtime node-local performance. The key observation is that recently created objects tend to have shorter lifetime, as in most high-level languages; thus, it is reasonable to place a newly created object on the same node as the creater object to enhance locality, expecting that the object will become garbage soon. Once allocation area is exhausted, *local* garbage collection takes place. It reclaims objects which are no longer referred to locally and whose references have never been *exported*. Local garbage collection is very efficient because it needs no message transmission for traversing pointers to objects. Furthermore, it reclaims much garbage because our creation scheme enhances locality; most objects are referred to from only local objects. During garbage collection, the garbage collector migrates some long-lived objects to other nodes, naturally incorporating migration and *copying/compacting* collection in the processor. At the same time, the garbage collector gathers information necessary for deciding whether or not an object should be migrated or not (such as reference count), which is used in the next garbage collection. Such information are collected with few additional overhead to standard garbage collection.

Our primary contributions are:

- Maximum node-local performance by quick allocation/reclamation of local objects.
- The creation scheme which enhances locality.
- Efficient local garbage collection which needs no message transmission and still effectively collects garbage thanks to locality.
- Migration scheme which utilizes information gathered by the garbage collector.

Throughout this paper, our proposal concentrates on *runtime mechanisms* for an object creation scheme which enhances locality but is *orthogonal* to approaches by compile-time analysis or programmer's annotation. Neither compile-time analysis nor programmer's annotation is sufficient. Compile-time analysis is difficult for programs which involves many dynamic task creations. Programmer's annotation, where programmers explicitly specify whether or not the task is created on remote node might be a good compromise between runtime overhead and good task distribution, but it has also limited effects. In this kind of scheme, it is difficult to access *global information*, such as load of other processors because keeping these information available causes runtime overhead which is not always necessary and choosing available runtime information raises difficult trade-off between good task distribution and runtime overhead.

2 Related Work

Several researchers are working on efficient implementations of high-level concurrent languages which involves dynamic task creation, some are in concurrent object-oriented languages ABCL[11, 9], Concurrent Smalltalk[3], others are in functional languages Id[2, 10] or logic programming languages KL/1[8, 4].

ABCL and CST concentrate on and achieved efficiencies of basic operations such as message passing, or task creation. Although the above implementations have achieved good performance of simple operations such as message passing or object creation, it has not been shown that they perform excellently in real, large scale applications. One of the biggest problems which has not been addressed in any of these proposals is where a newly created task should go and how to reclaim spaces occupied by garbage data at a low cost. To our knowledge, existing proposals are not currently taking *locality* into account, that is, newly created tasks are located on a remote node by default. More importantly, they have no mechanism for local garbage collection, where global garbage collection takes place whenever one of the node memory overflows, that may degrade the overall performance significantly.

Culler et al[2] has shown how overhead of fine-grain dataflow synchronization can be reduced without elaborate hardware support in their implementation of Id. It again concentrates on basic operations such as synchronization or function invocation and addresses little about task creation schemes.

The KL/1 implementation[8, 4] is similar to ours in that their implementation allows efficient local copying garbage collection in spite of presence of external references. Their scheme works because data are referred to from other nodes indirectly through *export* table, which keep track of all externally referenced data. This involves large runtime overhead in usual computation because table managements are necessary for exporting references[4].

3 Representation of Object Address

Representation of remote references has a large impact on the performance of ordinary computation and garbage collection. Many studies on distributed garbage collection employ "export/import tables" and objects are referenced indirectly[6] via this table. While this scheme simplifies implementation because it allows system to move objects freely for garbage collection/migration, it incurs significant overhead for accessing objects and table managements. Since the purpose of the export table is to serve as the "root pointer" in local garbage collection, the export table must hold some superset of all the references from remote nodes while it must not be too conservative to make effective local garbage collection. This is usually done by reference counting or its variation, where every reference transfer/disappearance operation consumes several tens of instructions solely for table management[4].

Instead, our representation of remote reference is a simple pair $\langle NodeID, Pointer \rangle$, where the second component is the real pointer that is valid on the node *NodeID*. Furthermore, there is no reference counting mechanism, minimizing the interference by garbage collection.

Seemingly, this scheme could cause several problems:

- This scheme might seem to prohibit local copying garbage collector because garbage collector cannot move objects which may be referred to from a remote node.
- This scheme might seem to require round trip packet-transfer latency for remote object creation or migration (i.e., memory allocation) because the real address of the allocated/migrated object could only be determined by the memory manager of the remote node.

Nevertheless, we can overcome these deficiencies and achieve better performance:

- For the first problem, temporal objects that have never been exported to remote node can be safely copied or reclaimed by local garbage collector. Since temporal objects tend to have locality as a consequence of our allocation scheme (Section 4.1), they are likely to be collected by local garbage collector.
- For the second problem, we devised a "*prefetch scheme*[9]" to effectively hide this latency where remote chunks are previously allocated and their address are kept in a "stock" in each node. On the remote object creation, requester node gets an address from the stock, and use it as an address of the newly created object. The requested node replies another chunk to replenish the stock of the requester, thereby keeping the amount of stock chunks nearly constant as long as requested node has enough chunks to serve.

4 Object Creation and Memory Management

4.1 Structure of the Heap

The heap is separated into two areas; one is a large contiguous area and the other contains a number of chunks which are initially linked as free list (Figure 1). A newly created object is allocated on the contiguous area and it can move freely on local garbage collection *as long as* its reference has never been exported. On a local garbage collection, an object whose reference has been exported moves to the chunk area and it can no longer be relocated. We call the contiguous *relocatable region*, while the chunk area is called *unrelocatable region*. For brevity, the later discussion assumes there is only one generation in the relocatable area and it is separated into two spaces, namely *from-space* and *to-space*

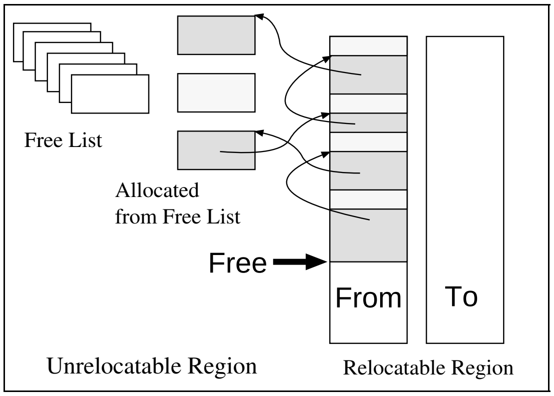

Figure 1: Structure of the Heap.

as in traditional copying collection. (In practice, we employ a more elaborate generational copying scheme in the relocatable region.)

Since the relocatable region is contiguous, an allocation is quickly done by incrementing the free pointer which points the head of the free space. Heap limit check requires only 1 instruction by keeping the free pointer in a special register and offsetting it so that heap exhaustion cause integer addition overflow trap[1]. This quick allocation helps for maximum node-local performance.

When the allocation area is exhausted, local garbage collection takes place. Local garbage collector takes contents of registers, stack, scheduling queue, and chunks in unrelocatable region to which store operation has occurred[1] as root pointers.

4.2 Exporting References

Since local collector cannot move/delete remotely referenced objects freely, exported references cannot directly point to the relocatable region. Thus, some maintenance is necessary on exporting references. More specifically, when the reference of an object is exported to another node, we create an *internal proxy* of the object in the following way:

- Check whether or not it is the first time the object is exported,
- If it is, allocate a chunk which is large enough for the object in the old region as an *internal proxy*, doubly-link the object body and the chunk, and initialize the virtual table for the chunk to a forwarding procedure that invokes the object body via indirection. The address of the chunk, instead of the body, is supplied as the exported address of the object. The object is copied into this chunk on the *next garbage collection.*
- Else, the address of the proxy, which must have been created before, is the exported address.

[1]This is because they may contain pointers to relocatable region.

```
/* <node_id, p> is the address
    of the object */
struct object *export(node_id, p)
    int node_id; object_t p;
{
  if(node_id == MYCELLID){
    object *proxy = p->link;
    /* p has already been exported? */
    if(proxy == NULL){
      proxy = allocate_from_unrelocatable_region();
      /* double link body and proxy */
      proxy->link = p;
      /* virtual table for forwarding */
      proxy->vftbl = fwd_tbl;
      /* export the proxy */
      return(proxy);
    } else {
      return(proxy);
    }
  } else return(p);
}
```

Figure 2: Exporting Reference.

Each object/proxy has an additional word which links itself to its proxy/object. Having a null pointer in this word indicates that the object has not been exported yet. Figure 2 shows the required overhead on exporting a reference. Instruction counts are 5∼8 instructions except for creating a new proxy, which occurs only once for each exported object. After exporting, the object is referenced directly from objects in the same node, while remote reference to the object is performed indirectly via the proxy. When a remote message arrives at an object which has its internal proxy, it looks up the virtual table of the proxy and invokes a forwarding procedure which forwards the message to the object body. When a garbage collection occurs, the body of the object is copied into its internal proxy so that this indirection disappears.

Although the role of internal proxies are similar to that of export table, there are essential differences:

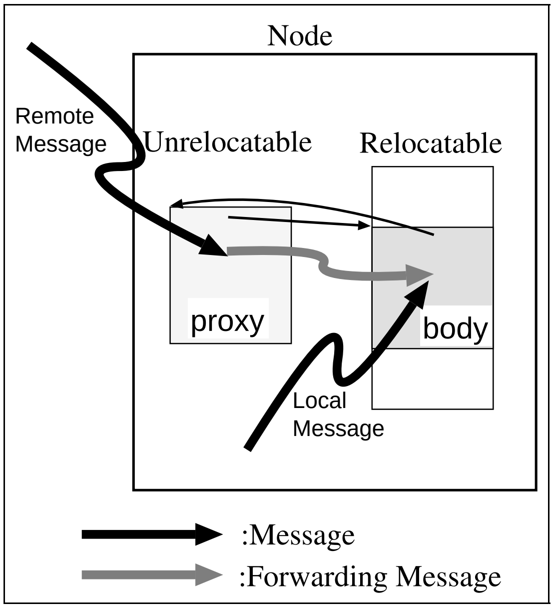

Figure 3: Exported object and its proxy.

- Local collector copies the object body to the address of the proxy so that indirection is naturally resolved after a local garbage collection.
- Local references still have direct access to the object body *without* any extra checking by utilizing assumption that ensure object is accessed only via method invocation.

The structure of an object whose reference was exported and its proxy is shown in Figure 3.

4.3 Activity of Local Collector

Once a local collector is invoked, it performs a variant of copying/compacting collection algorithm. Differences between normal copying garbage collection are: (1) it may *migrate* some objects in the garbage collection process and (2) it gathers information for subsequent local garbage collection to decide whether or not an object should be migrated.

During the pointer traversal, it encounters the following (live) data.

Data other than object Local collector simply copies it to the to-space.

An object which has already been exported Since these object has its proxy, local collector copies the object body into the proxy space, thereby freeing the space occupied by the object body and eliminating the indirection through the proxy. Any local references to the object is scavenged to point to the proxy in the garbage collection process.

An object which has never been exported Since no remote reference to this object exists, the local collector can move the object freely to any place, including other nodes (i.e., migration). Pointers to the object is naturally *forwarded* in the process of copying collection.[2] The local collector decides whether or not it should migrate an object using various information gathered by previous garbage collections. Although such information may not be up-to-date because they were gathered on previous collections, they give a reasonable approximation at very low additional overhead. Details of migration is described in the next section.

5 Algorithm of Migration

Our migration scheme is integrated in the process of local garbage collection, which is a non-trivial extension of a well-known copying garbage collection. When the garbage collector finds a pointer to an object in the process of pointer traversing, it decides whether or not the object should be migrated. If it decides the object should be migrated (how to this is described later), the garbage collector moves the object and data which are reachable from the object by only *local* pointers to to-space and then send them to the target node. This is essentially a copying collection with only one root object. (Notice that simple depth-first copying does not handle cyclic references.) After sending the object, the space for sending the object is freed.

An important problem is how to decide whether or not the garbage collector should migrate an object. In fact, many factors affect the preference of migration:

1. Load of other nodes.
2. Reference counts of the object.
3. Size of the object and other data pointed to from the object.

Since it is too expensive to maintain them always up-to-date, we must use some approximation by periodically gathering these information. We again gather these information in the process of a local garbage collection. Gathering them in the process of a local garbage collection

[2] When we have two or more generations, we must collect generations which contains moved objects.

is essential improvement because it traverses local pointers in any case. For instnce, reference counts of an object is obtined by incrementing the reference count of the object whenever a local garbage collector finds a pointer to the object.

6 Global Garbage Collection

Our local garbage collection employs a conservative view about remote references; i.e., once a reference of an object is exported, the object cannot be reclaimed by the local garbage collection even if it is already garbage. Therefore, we must use some *global garbage collection* scheme as well, which reclaims garbage objects whose reference have been exported. We employ a distributed mark-and-sweep (DMS) algorithm as global garbage collection because it imposes smaller overhead to user processes compared to reference counts. The details of the algorithm is described in [5]. This section concentrates on *when* the global collection takes place.

A DMS, which runs concurrently with user processes, takes place when an unrelocatable region is exhausted. If a user process requires more unrelocatable memory in the process of a DMS, it saves its context into the relocatable region and waits for the DMS to give enough memory. In this scheme, there is still a possiblity that there is not enough room for saving the information that the context is waiting. When this occurs, we try a local garbage collection to get memory for saving the information. If this again fails, we use a message buffer of one of other nodes as an emergency buffer. This is done by sending a message which returns to the sender after some period and checks to see if there is enough memory in the sender node. If there is, it resumes the context, else it repeats the same process.

7 Implementation Status

Actual implementation of the proposed scheme is being developed on Fujitsu laboratory's MPP, AP1000[7], which consists of 32–1024 SPARC chips. The node-local garbage collection has been implemented and is operating on a single CPU SPARC. Currently we are developing the proposed migration routines and examining better algorithm for deciding which objects should be migrated.

8 Conclusion

We have proposed a software architecture for concurrent object-oriented computing in massively parallel computers which includes mechanism for locality management, load-balancing, and garbage collection. It achieves maximum node-local performance by quick local allocation and indirection-free object access. Enhancing locality allows efficient local garbage collection and the proposed migration scheme is low-overheaded because it is free from bookkeeping of migrated objects which no longer reside on the node. Furthermore, it can be intelligent by taking advantages of information gathered by the local garbage collecter, such as reference counts.

References

[1] Andrew W. Appel. *Compiling with Continuation*. Cambridge University Press, 1992.

[2] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 26, pages 166–175, Snata Clara, California, April 1991.

[3] Waldemar Horwat. Concurrent Smalltalk on the message-driven processor. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[4] Nobuyuki Ichiyoshi, Kazuaki Rokusawa, Katsuto Nakajima, and Yu Inamura. A new external reference management and distributed unification for KL1. In *New Generation Computing*, volume 7, pages 159–177. Springer-Verlag, 1990.

[5] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. An algorithm of distributed garbage collection on a multicomputer and its performance evaluation. Senior's thesis, Department of Information Science Tokyo University, 1992.

[6] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world.

In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–58, Albuquerque, New Mexico, January 1992.

[7] Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Low-latency message communication support for the AP1000. In *The 19th Annual International Symposium on Computer Architecture*, volume 20, pages 288–297, Gold Coast, Australia, May 1992.

[8] Yasutaka Takeda, Hiroshi Nakashima, and Kanae Masuda. A load balancing mechanism for large scale multiprocessor systems and its implementation. In *New Generation Computing*, volume 7, pages 179–195. Springer-Verlag, 1990.

[9] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, 1993. to appear.

[10] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, volume 20, pages 256–266, Gold Coast, Australia, May 1992.

[11] Masahiro Yasugi, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/onEM4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *Conference Proceedings of 1992 International Conference on Supercomputing*, pages 93–103, Washington,D.C., July 1992.