Global Garbage Collection for Distributed Heap Storage Systems

by

Khayri A.M. Ali and Seif Haridi

SICS R87005
Research Report
ISSN 0283-3638

Global Garbage Collection for Distributed Heap Storage Systems

Khayri A. M. Ali and Seif Haridi

Logic Programming Systems
Swedish Institute of Computer Science
P.O. Box 1263
S-163 13 Spånga, Sweden

Abstract

We present a garbage-collection algorithm, suitable for loosely-coupled multiprocessor systems, in which the processing elements (PE's) share only the communication medium. The algorithm is global, i.e. it involves all the PE's in the system. It allows space compaction, and it uses a system-wide marking phase to mark all accessible objects where a combination of parallel breadth-first/depth-first strategies is used for tracing the object-graphs according to a decentralized credit mechanism that regulates the number of garbage collection messages in the system. The credit mechanism is crucial for determining the space requirement of the garbage-collection messages. Also a variation of the above algorithm is presented for systems with high locality of reference. It allows each PE to perform first its local garbage collection and only invokes the global garbage collection when the freed space by the local collector is insufficient.

Key Words: Garbage collection, storage heap, parallel algorithms, parallel architecture.

1. Introduction

Several distributed architectures for efficient execution of programs that have large potential parallelism have been proposed [1-6]. One realization of such architectures is based on loosely coupled multiprocessor systems having a number of PE's that share only a communication network. The local stores of the PE's are used to build a global address-space which is maintained by an ***object storage-system*** (OSS). Execution of programs on such architectures usually utilize the global address-space as a distributed heap by invoking operations defined by the OSS to create and manipulate objects. One of the main functions of an OSS is automatic storage reclamation. In this paper, we are interested in a distributed global garbage-collection scheme to reclaim the space of inaccessible objects. The associated OSS is discussed only for the sake of better understanding of the task of the garbage collector. The OSS of interest is suitable for applications that do not have real-time constraints, e.g., computational and simulation applications. The OSS is suitable for distributed systems that have a medium number of PE's and have a relatively small communication delay.

A reason for investigating a non-real-time garbage-collection scheme is due to the cost of real-time garbage collection. Wadler [7] analyses two algorithms for real-time garbage collection. One applies to Dijkstra [8] and Steele [9] algorithms, which use two parallel processors: the mutator and the collector. He shows that parallel garbage collection requires at least twice as much processing power as a sequential garbage collection. Wadler also analyses the algorithm in which the tasks of the mutator and collector are time-shared by a single processor. He finds, also in this case, the same result as in the first analysis provided that the collector is not wasting time attempting to do unnecessary garbage collection.

The first demonstrated global garbage-collection scheme for distributed environment, that serves the same purpose as our algorithm, has been proposed by Hudak and Keller [10]. It is called the marking-tree collector. It is a real-time, distributed, global garbage collector of the mark-sweep variety without memory compaction. It adapts the parallel real-time scheme of Dijkstra [8] to distributed environments. The storage reclamation is performed in parallel with the main computation. It assumes that each PE consists of two processors, the mutator and the collector. Each processor (mutator or collector) has its own task-queue from which it sequentially executes tasks. The tasks of the collector perform various functions of the garbage collection. Each task locks all objects that it will access during the execution of an operation to guarantee exclusive access to the objects. If a task finds that some object was already locked by another task, all locked objects by this task are unlocked and the task is rescheduled in its queue to prevent deadlock. Since the objects manipulated by a task operation may reside in different processors, the locking mechanism causes costly processing time especially when the mutator and the collector are contending for shared objects. There is a unique root of the graph of objects which is distributed over the entire system. The collector of the PE containing this root initiates one garbage-collection cycle after the other. Each garbage-collection cycle consists of a mark phase followed by a sweep phase. During the mark phase, mark-tasks are spawned by the collectors to mark the graph of accessible objects. The graph is traversed by a parallel breadth-first strategy. Unfortunately, the space requirement of the collector queues is very difficult to determine due to the dynamic nature of the graph and the breadth-first traversal strategy. This scheme may work only if the collector does not run out of space, otherwise deadlock will arise.

Ali gives a number of distributed garbage collection schemes: *global, local-global, local, and real-time* [11]. The global scheme engages all PE's in every garbage collection invocation. The local-global scheme allows each PE to perform first its local garbage collection and only invokes the global garbage collection when the freed space by the local collector is insufficient. The local scheme allows each PE to perform a local garbage collection independently and asynchronously without any use of global garbage collection. In the real-time scheme each memory operation performs some garbage collection work as in Baker's scheme [12]. This paper presents the first two schemes.

Hughes gives a scheme which is closest to Ali's local scheme [13]. Hughes's scheme is able to reclaim distributed circular structure, and Ali's local scheme cannot. On the other

hand, Hughes's scheme takes longer time to recover remotely-referenced garbage than Ali's local scheme.

The global garbage-collector presented in this paper is a scheme of the mark-sweep-compact variety where every PE has its own root of accessible objects. A system-wide mark phase is used to trace all accessible objects by sending mark messages. We use a combination of depth-first/breadth-first strategies to trace the distributed graphs. The space requirement of the garbage collection messages is determined and can be guaranteed in advance. There is a credit mechanism that regulates the number of garbage collection messages in the system. When the space of the mark messages is exhausted, the object graphs are traversed in a parallel depth-first mode. When there is enough message space, the graphs are traversed in a parallel breath-first mode. The algorithm is asynchronous except at the end of the marking phase where a single synchronization point is needed. The synchronization is performed by a master that starts the sweep phase. Any PE in the system can become the master of a garbage-collection cycle. A garbage-collection cycle starts when any PE exhausts its local available space. The scheme is distributed homogenously over the whole system and each PE executes the same algorithm. Since any PE can be a master, it may happen that many PE's simultaneously want to be the system master. In this case a master-selection protocol is used which is integrated in the marking phase.

We continue by describing our distributed-processing model, the object representation and the parts of our OSS that are relevant to the garbage-collection algorithm. We proceed by the global garbage-collection algorithm and thereafter present a variant of the OSS that allows each PE to perform a local garbage collection as a first trial. Our global scheme will be invoked only when the local one cannot reclaim enough space. This last algorithm is suitable for systems with high locality of references. Finally the space, time and communication performance of our schemes are estimated.

2. Distributed-Processing Model

The specification of our distributed processing model conforms to the following description:

1. There are many identical PE's.
2. Each PE has a local store that is used to hold a portion of the objects that are involved in the program execution.
3. Each PE has a processor that interprets instructions and performs object-storage operations.
4. The PE's are interconnected by some arbitrary communication network that allows any PE to communicate with any other PE in the system.
5. There is a global addressing-scheme whereby an object may reference any other object in the system.
6. Communication between the PE's occurs by message-passing.
7. Communication is reliable, that is, any message sent by one PE to another PE is received correctly after an arbitrary but finite delay.
8. The communication channel between each pair of PE's is *order-preserving*, which means that message transmissions obey the following first-in-first-out rule: messages sent by any PE i to any other PE j are received by j in the sequence in which they are sent by i.

3. Object Structure

We assume a general structure for objects, object references and object descriptor table entries. Other representations with potentially lower space overhead do not significantly alter the results reported in this paper.

3.1. Objects

All objects in the system are stored in physically separated areas (local heaps) that together make up the storage space of the whole system. Each area is located in the local store of a PE. A new object is created by obtaining space to store its fields in a contiguous series of equal cells (or words) in one area. Each object in the system is associated with a unique

identifier called its *object reference*. A cell of an object may contain either an object reference or an arbitrary value. A one-bit field of each cell, called *Tag*, is used to distinguish between a value cell and a reference cell. (When a cell contains an object reference its Tag field is marked by R, otherwise by V.)

3.2. Object Reference

Before giving the structure of an object reference we discuss the advantages of using one level of indirection for referring to the physical locations of objects.

We allow memory compaction. All objects that reside in one area and are still in use will be compacted towards one end of that area leaving a large contiguous free space that is equal to the storage spaces of the scattered garbage objects reclaimed. Without memory compaction, we cannot allocate space for a new object that is larger than any free space yet smaller than the total available free spaces. Memory compaction allows efficient space utilization.

During memory compaction, objects that are still in use are moved from their current locations to new locations. All references to each moved object must be updated. If many objects refer directly to the old location of a moved object, it would be very expensive, in a distributed environment, to find and update those references. This is due to the high communication overheads associated with remote reference updates.

To make reference updates cheap, objects refer indirectly to the locations of other objects through an Object-Descriptor Table (ODT) that is provided in each PE. Any object in the system can be uniquely identified by its reference. The object reference consists of three parts: *Tag*, *PeId*, and *Index*. The PeId contains the name of the PE that has the object in its local area, and the Index contains an offset in the ODT of the PE, where a pointer to the object's location is found.

3.3. Object-Descriptor Table

There is one ODT in each PE's memory. The ODT consists of a fixed number of entries. Each entry contains information pertinent to the locally residing object and also to the garbage collection. The entry in an object descriptor table consists of four parts: *In-Use, Gc-Fields, Size,* and *Loc*.

The In-Use bit is *Free* only when the entry is free. If the In-Use bit is *Busy*, then the size field *Size* indicates the number of cells that the object occupies. The location field *Loc* points to the beginning of occupied space of the object. The garbage collection fields (Gc-Fields) are described in Section 12. The free entries of an ODT are not associated with any space.

The drawbacks of using an ODT are (1) there is extra space overhead, and (2) more processing power is required to maintain the ODT. These drawbacks are outweighed by saving the communication costs for updating remote references. Using fast memory for ODT and/or caching object references improves the performance of the object-storage system considerably.

3.4. Free-Entry Pool

Each PE maintains a linked list of all the free entries of its ODT. The *Loc* field of each free entry can be used to construct such a list *(free-entry pool)*. A free entry is needed when creation of a local object is requested. An entry is inserted into the free-entry pool when its object's storage is reclaimed during the garbage collection. Requesting a free entry when all entries are busy, invokes the garbage collector.

3.5 Local Heap

Each local heap consists of a finite number of cells. When an object of size n is created, n contiguous cells are allocated in the heap, and a pointer to the first cell is stored in the Loc field of the associated ODT's entry. Creation of an object with larger size than the available free space invokes the garbage collector.

4. Processor Model

In each PE, the unit of execution is a *process*. It has an associated code object that contains a sequence of operations. Processes in the system execute concurrently, while the operations in the same process are executed sequentially. If all objects involved in the execution of a current operation are local, then the execution will be completed locally without the cooperation of any other PE in the system. When some objects involved are nonlocal, the process is suspended until the remote operation is completed. For operations that involve nonlocal objects, remote requests are sent to the other PE's that have such objects in their areas to complete the operation. When such requests have been received and processed by the destination PE's, a reply message is sent to the PE that has the suspended process to reschedule it for execution.

Each PE maintains two queues: (1) the ready queue (RQ) contains all processes that are ready for execution, and (2) the message queue (MQ) contains all messages that have been received from other PE's for performing operations on local objects. Each PE selects for execution either an operation from a process of its RQ or a received request from its MQ. A typical processor cycle may be specified as follows.

```
var   priority-flag: Boolean := true
LOOP
      IF NOT Empty(MQ) AND priority-flag THEN
            Execute(Next-Message-From(MQ))
      ELSE
            Execute(Current-Operation-From-First-Process-Of(RQ))
      END IF
      priority-flag := NOT priority-flag
END LOOP
```

According to this cycle the priority of MQ and RQ changes every cycle to allow fairness of the selection between them.

The PE's communicate asynchronously by message passing. We use a non-waiting send mechanism; that is, the statement "*SEND* Msg(y) *TO* p1", executed by PE p0, causes a message of type Msg with y as transmitted data, to be sent to PE p1. No waiting is done by p0 for delivery of the message; rather it continues executing the next statement. In the specification of a message (Appendices A and C) the keyword *Message* is used instead of *Procedure.*

5. Object-Storage System

The OSS (Object-Storage System) provides the user with a set of object operations. Usually there are typical operations for creating an object, accessing certain fields of an object, and storing a value in certain fields of an object. We are going to present only two operations: *Create* and *Access*, because they are directly relevant to the garbage collector. These operations are of general form. For example, *CONS* can be considered as a specialization of *Create*, and *CAR, CDR*, and *RPLACA* as specializations of *Access*. Processes are also objects, therefore all object operations can be performed on processes. For specialized process-operations the reader may consult [11].

5.1. Create Operation

The Create operation: *Create(n: Size, c0,c1,...,cn-1: CellValue, p: PE, u: ObjectRef, i: ObjectOffset),* creates an object of size n cells in the local heap of the PE p; its cells are initialized to c0, ..., cn-1 where a ci is a value or an object reference, and a reference to the created object is stored in the i^{th} cell of the object referred to by u.

Let us assume that the current process resides in PE p1, and the object u in PE p2. In Create operation, one of the following five situations may occur:

1. The entire operation is performed locally, p=p1=p2.
2. The process and the created object are in the same PE, the object u is in another PE,

p=p1≠p2.

3. The process and the object u are in the same PE, and the created object is in another PE, p1=p2≠p.
4. The object u and the created object share the same PE, but the process is in another PE, p=p2≠p1.
5. The process, the object u and the created object are in three different PE's, p≠p1≠p2≠p.

During Create operation a number of messages may be sent between the involved PE's . Table I below summarizes the types of messages that are used. (Appendix A contains the specification of the Create operation in full details.)

In the description we denote the executing process by a, the created object by v and the updated object by u.

TABLE I

a. Update-and-Ack-Back(u,v: ObjectRef, i: ObjectOffset, a: ProcessRef)
b. Create-and-Ack-Back(n: Size, c0,c1,..,cn-1: CellValue, u: ObjectRef, i: ObjectOffset, a: ProcessRef)
c. Update-and-Reschedule(u,v: ObjectRef, i:ObjectOffset, a: ProcessRef)
d. Create-Update-and-Reschedule(n:Size, c0,c1,..,cn-1:CellValue, u:ObjectRef, i:ObjectOffset, a:ProcessRef)
e. Reschedule-Msg(a: ProcessRef)

When the first situation occurs (p1=p2=p), the new object is created in p (if the local heap of p has sufficient space and the ODT of p has a free entry, otherwise a global garbage collection will be invoked). A reference to the created object is stored in the i^{th} cell of the object u. That is to say, the operation is completed locally.

In the second situation (p=p1≠p2), a new object v is created in p; the message Update-and-Ack-Back(u,v,i,a) is sent to p2, and the process a is suspended. The sent message requests p2 to store v in the i^{th} cell of the object u and to reply back, by the message Reschedule-Msg(a). When Reschedule-Msg is processed by p, the suspended process a is rescheduled.

In the third situation (p1=p2≠p), the process a and the object u reside on p1. The message Create-and-Ack-Back(n,c0,c1,..,cn-1,u,i,a) is sent to p and the process a is suspended. The sent message requests p to create a new object locally and to reply back with a reference to the created object. When Create-and-Ack-Back is processed by p, the reply message Update-and-Reschedule(u,v,i,a) is sent back to p1. When Update-and-Reschedule(u,v,i,a) is processed by p1 a reference to v is stored locally in the i^{th} cell of u and the suspended process a is rescheduled.

In the fourth situation (p=p2≠p1), the object u and the object to be created v are in p. The process a resides on p1. The remote request Create-Update-and-Reschedule (n,c0,c1,..,cn-1,u,i,a) is sent to p and the process a is suspended. When Create-and-Update-and-Reschedule message is processed by p, a new object v is created, and a reference to it is stored in the i^{th} cell of the object u. Thereafter the reply message Reschedule-Msg(a) is sent back to p1. Processing the Reschedule-Msg by p1, as explained before, completes the Create operation.

In the last situation where our objects reside on different PE's, (p≠p1≠p2≠p) the remote request Create-and-Update-and-Reschedule (n,c0,c1,..,cn-1,u,i,a) is sent to p and the current process is suspended. This step is exactly as the first step in the previous situation. However, the processing of this message is slightly different. When the Create-and-Update-and-Reschedule is processed by p, a new object v is created, and the message Update-and-Ack-Back(u,v,i,a) is sent to p2. When this message is processed by p2, as described above, the reply message Reschedule-Msg(a) is sent to p1. Processing this Reschedule-Msg by p1 completes the Create operation.

An important remark should be made about the Create operation. A remote create

operation may cause some references to accessible objects to exist transiently only in the communication network. For example, in the second situation when Update-and-Ack-Back(u,v,i,a) is sent, references to the objects u and v and the process a may exist transiently only in the communication network. Nonetheless, a global garbage-collection algorithm should be able to reclaim back those references from the network and use them to mark the accessible objects.

5.2. Access Operation

The Access operation: *Access(v,u: ObjectRef, j,i: ObjectOffset)*, reads the content of the j^{th} cell of object v and writes it in the i^{th} cell of the object u. Appendix B describes this operation briefly, and the full details are found in [11].

6. Global Garbage-Collection

Since each local heap and each ODT is finite, a time will come when a PE creates a new object (by invoking Allocate-Locally procedure in Appendix A), and there is no free storage space. It is usually the case, in object-oriented systems, that some objects have already been created, but are no longer needed in future computation. As a result a PE that cannot create a new object in its local heap has to recycle the space of such inaccessible objects to allow the computation to proceed. Since a PE does not know what local objects are still needed by other PE's, all PE's have to cooperate to identify such objects. We refer to distributed schemes that involve the whole system to identify and recycle the space of garbage objects in the whole system as *global garbage-collection* schemes. They consist mainly of two phases: (1) mark phase and (2) sweep-compaction phase. In the mark phase, all accessible objects in the system will be marked. In the sweep-compaction phase, each PE collects the space of unmarked objects and their associated ODT entries, and compacts its local heap. Marking all needed objects in the entire system requires cooperation between the PE's, while sweeping and compaction of local heaps is performed independently by each PE.

Before presenting our garbage collection schemes, in Sections 9-14, we describe briefly the method used in the specification of the schemes and revisit the processor cycle to include garbage collection messages.

7. Specification Method

In the specification of our distributed garbage-collection scheme (protocol), we use a variant of the finite state-transition diagrams method described in [14]. In this method a protocol is specified by a number of finite state-transition diagrams. Each diagram consists of a set of nodes, called *states*, connected by directed labeled arcs, called *transitions*. Each transition is characterized by an enabling predicate (EP) and an action. The enabling predicates and actions are specified in terms of Pascal-like programming constructs (Concurrent Euclid). A transition is atomic. It has exactly one source state and one destination state. It may fire only when its source state contains a token, (designating the location of the instruction counter of the associated PE), and the associated enabling predicate is true. The firing of a transition consists of transferring the token to the destination state, and then executing the associated action. At any time, at most one transition is being fired in a given state transition diagram. One of the states in each diagram is identified as its initial state. Finally, all the states in a diagram are reachable from its initial state.

8. Processor Model Revisited

In our garbage-collection scheme we assume that each PE has one more queue, called GQ, which contains only garbage-collection messages. Each PE is either in the computation phase or in the garbage-collection phase. When a PE is in garbage-collection phase, no message on RQ or MQ are processed. The new processor cycle may be specified as follows.

```
var priority-flag: Boolean := true
LOOP
    IF NOT Empty(GQ) THEN
        Execute(Next-Message-From(GQ))
    ELSEIF phase = Computation THEN
        IF NOT Empty(MQ) AND priority-flag THEN
            Execute(Next-Message-From(MQ))
        ELSE
            Execute(Current-Operation-From-First-Process-Of(RQ))
        END IF
        priority-flag := NOT priority-flag
    END IF
END LOOP
```

We present our garbage-collection algorithm in two steps. In the following section, we outline a simple global garbage-collection scheme in which a distinguished PE called *master* starts and synchronizes the garbage-collection phases. In Section 10, the problems pertaining any global scheme are described and solutions are discussed. In Sections 11-13, we present our main global garbage collection scheme. It is an asynchronous scheme in which any PE can be master. It allows more parallelism and faster marking of the objects graphs than the simpler scheme.

9. Global Garbage Collection with a Distinguished Master

In this scheme, there is one PE called the *master* that starts and synchronizes the garbage collection phases. Any other PE, called *normal*, wishing to invoke the garbage collection, sends a *Start-Garbage-Collection* request to the master and waits. When the master either runs out of space or receives and processes the first Start-Garbage-Collection request, it broadcasts a *Start-Mark-Phase* request to all other PE's, and starts its local mark phase. The master discards Start-Garbage-Collection requests arriving after starting mark phase. When a Start-Mark-Phase request is processed by any normal PE, the PE switches to its local mark phase. In the local mark phase, each object that is reachable from a specific set of local roots will be marked. In our execution model, the local roots of PE q are: the RQ, objects referenced from messages in the MQ, and objects referenced from messages in transit arriving at the local MQ of q. Marking is done by sending forward a *Mark-Object* message through every arc of the accessible graphs starting from the roots. The leaves of an accessible graph respond by sending backward *Mark-Object-Reply* messages. These messages are propagated backward until the root receives such a message from every immediate child. Thereby marking all reachable objects of q is completed.

We have to notice that a normal PE p might receive a Mark-Object message from another normal PE before receiving a Start-Mark-Phase request from the master. In this case, p should interpret the Mark-Object message as both a Start-Mark-Phase request and a Mark-Object message. When p receives the Start-Garbage-Collection request, p discards it.

When any normal PE completes marking all reachable objects, it sends to the master an *End-Mark-Phase* message and waits. The global mark phase is completed exactly when the master completes its local mark phase and processes an End-Mark-Phase message received from each other PE. When the system reaches this state, all reachable objects from all roots in the system are marked, the master broadcasts to all other PE's a *Sweep-and-Compact* request, and starts its local sweep-compaction phase. When a Sweep-and-Compact request is processed by any normal PE, the local sweep-compaction phase is started. Any PE that completes its sweep-compaction phase switches immediately to its computation phase. Figures 1-2 and Tables II-III show the state-transition diagrams and tables that outline the above scheme for the master and a normal PE. In the diagrams the computation phase is the initial state of each PE.

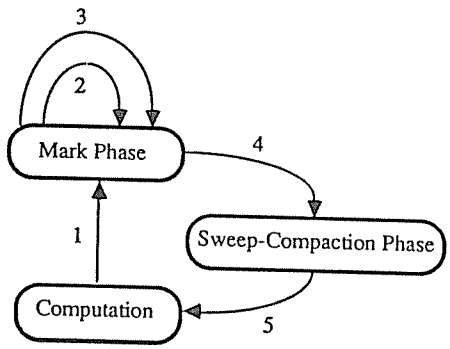

Figure 1: The master state-transition diagram.

TABLE II

Transition	Enabling Predicate	Action
1	No more free space or Start-Garbage-Collection message.	Broadcast a Start-Mark-Phase message to all other PE's; Start the local Mark phase.
2	Start-Garbage-Collection message.	Ignore.
3	End-Mark-Phase message.	Record ID of PE sending message.
4	All End-Mark-Phase messages received and the local Mark phase completed.	Broadcast a Sweep-and-Compact message to all other PE's; Start the local Sweep-Compaction phase.
5	The local Sweep-Compaction phase completed.	Switch to Computation phase.

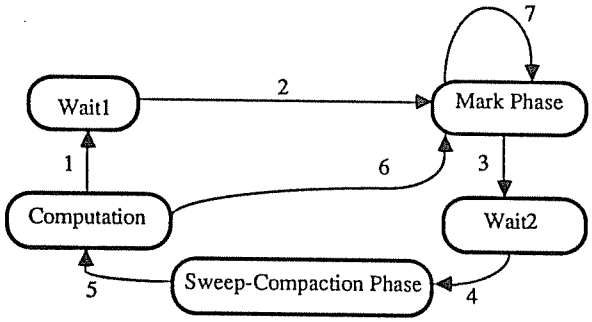

Figure 2: A normal PE state-transition diagram.

TABLE III

Transition	Enabling Predicate	Action
1	No more free space.	Send a Start-Garbage-Collection message to the master.
2,6	Start-Mark-Phase msg or Mark-Object message.	Start the local Mark phase.
3	The local Mark phase completed.	Send an End-Mark-Phase message to the master.
4	Sweep-and-Compact message.	Start the local Sweep-Compaction phase.
5	The local Sweep-Compaction phase completed.	Switch to Computation phase.
7	Start-Mark-Phase message.	Ignore.

This specification gives only the general structure of the scheme. In most global schemes there is a number of problems pertaining to the mark phase. We now turn our attention to these problems.

10. Global Mark-Phase

The global mark-phase starts when the master broadcasts a Start-Mark-Phase request to all other PE's and starts its local mark-phase. The task of the global mark-phase is to mark all accessible objects in the whole system. As mentioned before, each PE has its own roots: the RQ, objects referenced from messages in the MQ, and objects referenced from messages in transit arriving to the local MQ of the PE.

One task of the global mark-phase is to guarantee that all objects referenced from messages in transit are investigated before completing this phase. Guaranteeing arrival and investigation of such messages is crucial, because it may happen that objects in the system are reachable only from these messages. Therefore if these messages are not investigated, some reachable objects may not be marked during the global mark-phase. Such objects will erroneously be garbage-collected during the sweep-compaction phase. In the following subsection we discuss some solutions to this problem. Thereafter we explain the storage problem of marking messages and our solution to it.

10.1. Computation Messages in Transit During the Mark-Phase

To guarantee that there are no computation messages still stored in the communication subsystem, when the global mark-phase is completed, one of the following two characteristics of the underlying communication subsystem is assumed:

I. The maximum possible transmission delay of a message in the communication subsystem T_{max} is known.

II. The communication channel between each pair of PE's is order-preserving.

It is possible to find a solution based on Assumption I. However, we think that this assumption is too severe because a reasonable choice of T_{max} is not always easy. Longer T_{max} may cause the PE's to be idle longer than necessary. Shorter T_{max} may not guarantee the arrival and investigation of all messages in transit.

Under Assumption II , we give the following two solutions:

First Solution

1. When a PE switches to the mark phase, it broadcasts a message, called *Shove*, to all other PE's, and thereafter starts marking objects from its available roots.
2. Each normal PE completes its local mark phase and sends an End-Mark-Phase to the master exactly when all objects reachable from its roots (old and new) are marked and (n-1) Shove messages have been received from the other PE's, where n is the number of PE's in the system.
3. The master finishes the global mark phase and starts the sweep-compaction phase exactly when it completes marking all objects that are reachable from its roots (old and new), and receives (n-1) Shove messages and (n-1) End-Mark-Phase messages from the normal PE's.

Since all channels are order-preserving and Shove messages are sent by any PE to all other PE's as the last computation messages, then when any PE receives (n-1) Shove messages, this implies that all computation messages that were sent before these Shove messages have been received. When the master finishes the global mark-phase, all accessible objects in the whole system are marked and no previous computation messages are still stored in the communication subsystem.

As an optimization to this scheme, the master does not need to broadcast any Shove message because it broadcasts a Start-Mark-Phase to all other PE's at the beginning of the global mark-phase. Such a Start-Mark-Phase can be interpreted by the destination PE as both a Shove message and a Start-Mark-Phase message. Similarly, each normal PE does not need to send a Shove message to the master, because each normal PE will send to the

master an End-Mark-Phase message when it completes its local mark-phase. That is, the master sends (n-1) Start-Mark-Phase messages to all other PE's when it starts the global mark phase, and receives from them (n-1) End-Mark-Phase messages when they complete their local mark phases. While, a normal PE sends (n-2) Shove messages to the other normal PE's when it receives a Start-Mark-Phase message from the master, and receives (n-2) Shove messages from the other normal PE's before sending an End-Mark-Phase message to the master.

The advantage of this scheme compared to one based on Assumption I is that the problem of choosing the interval T_{max} is avoided. It is easy to know exactly the moment at which all accessible objects in the whole system are marked. Furthermore, the PE's will not be waiting for a timeout.

The clear disadvantage of this solution is that $(n-2)^2$ Shove messages are needed. By the following second solution we show how the number of Shove messages can be reduced and in some cases becomes zero.

Second Solution

Each PE is provided with two additional one-bit vectors of size n each. Each entry in the vectors corresponds to one PE. Let us call these two vectors *In-Vector* and *Out-Vector*. Initially, all vector entries are *Off*. No Shove messages are sent at the beginning of the local mark-phases. Instead, sending the necessary Shove messages is delayed until each PE completes marking all objects that are reachable from its available roots. Each PE, after leaving its computation phase, maintains its In-Vector and Out-Vector as follows.

1. When a PE sends any garbage-collection message to another PE p, it sets (to *On*) its Out-Vector entry that corresponds to p.
2. When a PE receives a garbage-collection message from another PE p, it sets (to On) its In-Vector entry that corresponds to p.
3. When a PE completes marking all reachable objects from all its available roots, it investigates its Out-Vector and sends a Shove message to each PE with an Off Out-Vector entry.
4. Each PE completes its local mark-phase exactly when all objects reachable from its roots have been marked and all entries of its In-Vector are On.
5. Each normal PE sends an End-Mark-Phase message to the master exactly when it completes its local mark-phase.
6. The master completes the global mark-phase exactly when its local-mark phase has been completed and it has received an End-Mark-Phase message from all other PE's.

As mentioned in Section 8, each PE that leaves its computation phase will not send any computation message until the current cycle of garbage collection is completed. Under Assumption II, the reception of a garbage-collection message (including a Shove message), by a PE q from another PE p, indicates that all computation messages that have been sent from p to q before starting the current cycle of garbage collection have been received by q. Since q sets the In-Vector entry corresponding to p when receiving a garbage-collection or Shove message from p, q is guaranteed the reception of all computation messages destined to it that were in transit at the beginning of the current cycle of garbage collection exactly when all the entries of its In-Vector are On. We also guarantee that each PE will receive garbage collection messages from all other PE's by allowing each PE to know locally the other PE's that are waiting for it by consulting its Out-Vector and sending Shove message to them if needed.

If each PE communicates with all other PE's during the garbage collection, the number of Shove messages is reduced to zero. In cases where a PE communicates only with a part of the system, the number of Shove messages sent is reduced to the number of remaining PE's. A disadvantage of this solution is that sending the necessary Shove messages at the end of each local mark-phase may delay the completion of the global mark phase.

For the sake of simplicity, we use the first solution in this paper.

10.2. Storage Problem of Marking Messages

In any uniprocessor garbage-collector a work space is needed and has to be available before invoking the collector. Shortage of the required work space may stop the task of the collector. In distributed garbage-collectors, there are two sources of storage-space requirement; the first one corresponds to what is needed in a uniprocessor collector, while the other is needed for garbage-collection messages. Garbage-collection messages are all messages that are used by the collector. We can divide such messages into two categories: (1) synchronization messages and (2) marking messages. Messages mentioned in Section 10.1 belong to the first category, whereas Mark-Object and Mark-Object-Reply messages mentioned briefly in Section 9 belong to the second category. The number of synchronization messages in the worst case can easily be determined. Also, their storage-space requirement can be determined and guaranteed to be available before performing the garbage collection. This number is always linearly proportional to the number of PE's in the system, except for Shove messages in 10.1, and the size of each one consists of a few storage cells. However, marking messages that are used to mark reachable objects of the distributed graphs in the system represent the largest part of the garbage-collection messages. Their number in the worst case is in the order of the number of arcs in the graphs. Of course, most important is the maximum number of such messages that may exist in the system simultaneously. This number can be less than the number of arcs in all graphs, but unfortunately is not known before performing the garbage collection. It is uneconomical to reserve the worst-case amount of storage space required to store these messages. On the other hand, shortage of such space may stop the task of the collector and possibly lead to deadlock. Therefore, we want to design a marking scheme with a limited and predetermined storage space requirement to store these marking messages. Such a scheme should also allow much parallelism in the system as possible.

The idea is as follows. We keep the total number of marking messages in the whole system at any moment less than or equal to say m messages. In the worst case, all these m messages may be received by one PE. Therefore, each PE reserves an additional storage-space that is required to store m marking messages. This idea can be implemented by allowing each PE in the system to have an initial credit k, such that $m = k\,n$. (where n is the number of PE's). Each PE decrements its credit by one when it generates a new marking message and increments its credit by one when it consumes a marking message. The minimum current value of the credit of any PE is zero. In this case marking will proceed but one marking message is produced for every consumed one. The consequence of this method is that at any instant the sum of all current credit values plus the number of existing marking messages in the system is equal to m. Each PE can generate up to its current credit value a number of marking messages that exist simultaneously.

This solution allows high parallelism for the following reasons:

1. Each PE starts its local-mark phase with credit k. This allows each PE to start marking up to k of its nodes.

2. When a PE did not consume all k of its own credit for marking objects reachable from its roots, the remaining credit can be used by this PE for marking objects belonging to nonlocal roots. This allows a PE p, which did not yet complete marking all its reachable objects, to borrow unused credits of other PE's with surplus credit. This will happen when the graph of p is distributed over those PE's. Borrowing unused credits increases the parallelism of marking objects which in turn speeds up the completion of the global mark phase.

In all published distributed garbage-collection schemes known to us, there is no solution to the storage problem of marking messages. Our solution guarantees that the collector always has enough storage space to complete its task. The speed of marking objects can be tuned according to the chosen value of m. The larger m is chosen, the more storage space required and the more concurrency in marking. For example, if m = n (i.e. k=1), n distributed graphs will simultaneously be traversed in depth-first manner. If m > n and we have n distributed graphs, the graphs will be traversed as a combination of depth-first and breadth-first.

11. Global Garbage-Collection with any PE as a Master

The main differences between this scheme and the one with distinguished master are (1) any PE can be a master of a garbage-collection cycle, and (2) there is no wait state for any PE before starting its local mark-phase.

Since the master of the previous scheme has no wait states and it starts both the local mark and sweep-compaction phases before any other PE, it is desirable to give this right to each PE. Allowing any PE to be the master makes each PE obey the same algorithm.

In the previous scheme (in Section 9) each normal PE starts its local mark-phase after receiving either a Start-Mark-Phase message from the master, or a Mark-Object message from any other normal PE. A normal PE enters its Wait1 state due to requesting the invocation of the garbage collection. In the scheme presented now, no PE has a wait state before starting its local mark-phase, thereby allowing more parallelism and faster global mark-phase.

This Section is organized as follows. 11.1 discusses a simple solution to the problem of selecting a master for any garbage collection cycle. 11.2 presents a general description of the global garbage collection scheme.

11.1 Simultaneous Invocation of the Global Garbage Collection

Since any PE can be a master, it is natural to make the invoker of the garbage collection the master of the current cycle. But it might happen that many PE's simultaneously invoke the garbage collection. In this case, only one of them should be the master. The simplest solution is to assign a static priority to all PE's in the system. When many PE's simultaneously invoke the garbage collection, the one with the highest priority of them becomes the master. Each PE can locally decide which one of the invokers should be the master if it knows all invokers and their priorities. Therefore, when any PE invokes the global garbage-collection, it sends to each other PE a Start-Mark-Phase message with its name in the message. A simple protocol for master selection during the current cycle of the global mark-phase is described in the following subsection.

11.2 General Description of the Scheme

Here, we use the first solution in Section 10.1, using Shove messages, to empty the communication subsystem from computation messages during the global mark-phase. The scheme works as follows.

1. Each invoker sends a Start-Mark-Phase message to each other PE; assigns itself as the current master of this cycle of garbage collection, and starts the local mark-phase.

2. If a PE receives a Start-Mark-Phase message before it has started its local mark-phase, it sends a Shove message to each other PE; starts its local mark-phase, and assigns the sender as the current master.

3. If a PE receives a Start-Mark-Phase message when its local mark-phase has already started, it compares the priority of the sender with the priority of the current master. If the sender has higher priority, the sender will be assigned as the current master, otherwise the current master remains as before.

4. If a PE receives a Shove message before it has started its local mark-phase, it sends a Shove message to each other PE and starts its local mark-phase.

5. If any PE receives a Shove message when its local mark-phase has already started, it does nothing (no Shove messages are sent).

6. Each PE completes its local mark-phase exactly when it receives either a Start-Mark-Phase message or Shove message from each other PE, and it marks all objects reachable from its roots.

7. Each PE that has completed its local mark-phase and is not the current master sends an End-Mark-Phase message to the current master.

8. The current master completes this global mark-phase exactly when it receives an End-Mark-Phase message from each other PE, and completes its local mark-phase.

9. When the global mark-phase has completed, the current master sends a Sweep-and-Compact message to each other PE and starts its local sweep-compaction phase.

10. When a PE receives a Sweep-and-Compact message, it starts its local sweep-compaction phase.

11. Each PE that completes its local sweep-compaction phase resumes its local computation.

In this scheme, we note that each invoker sends a Start-Mark-Phase message and each non-invoker sends a Shove message to all other PE's. Each PE has to receive a Start-Mark-Phase message or Shove message from all other PE's before ending its local mark-phase. By this method we guarantee that (1) the communication subsystem is emptied from communication messages in transit, and (2) the master of the current cycle of garbage collection is selected uniquely by each PE at the end of its local mark-phase.

We should mention here that it is possible to design a scheme without using any master by allowing each PE to send an End-Mark-Phase message to all other PE's when it completes its own mark phase. Also each PE has to receive an End-Mark-Phase from all other PE's before starting its own Sweep-Compaction phase. This solution costs n(n-1) End-Mark-Phase messages while the former solution requires only (n-1) End-Mark-Phase messages. In what follows we assume the former solution to avoid the increase of traffic in the communication subsystem.

12. Marking Phase

This section presents the idea of the marking scheme, and Appendix C contains the marking algorithm in detail. The original idea of sending mark messages is due to Hudak and Keller in their work on the Marking-Tree Collector [10, 15]. We extend their work by incorporating the credit mechanism to solve the storage problem of marking messages.

Assume for simplicity the following:
1. Each PE has its own local root of the graph of reachable objects.
2. Each root has an extra dummy object which resides in the same PE; this object is called *Parent.*
3. Each PE has an initial positive credit k.
4. Each object has the following four garbage collection fields (Gc-Fields in Section 3.3): *Mark-Bit, Parent-Ref, Child-Count,* and *Last-Child.* The role of these fields is described later in this section.

Marking objects is performed by two types of messages:

(1) Mark-Object(o,p: Object-Ref)
(2) Mark-Object-Reply(p: Object-Ref)

The first argument of a Mark-Object message is a reference to the object to be marked and the second argument is that of the object's parent. The argument of a Mark-Object-Reply is a reference to the object to reply to. A Mark-Object message is used to mark an object's child and a Mark-Object-Reply message is used as a reply to the object when the object's child has been marked.

The idea of the marking scheme is as follows.

1. When marking starts, each PE sends a Mark-Object message to its local root and decrements its credit by one.

2. A Mark-Object(o,p) is processed by any PE, as follows. If the object o is already marked, a Mark-Object-Reply message is sent to its parent object p. Otherwise, the

object is marked and investigated. If it is a leaf, a Mark-Object-Reply message is sent to its parent. If it is not a leaf and has only one child, say c, a Mark-Object(c,o) is sent to the child c. (It is noted that the credit is not modified because, one Mark-Object message is consumed and only one message generated.) If there are more than one child, first the local credit is incremented; then as long as the local credit is greater than zero a Mark-Object message is sent to each child of the object o and the local credit is decremented accordingly. The Child-Count field of the object o keeps track of the number of Mark-Object messages that have been sent to its children and have not yet received replies.

The Last-Child field of an object refers to the last object-cell that has been investigated. This field is needed because in most cases each object sends Mark-Object messages to some of its children and it has to remember which children it has sent Mark-Object messages to and which it has not. The Mark-Bit is used for marking purposes. The Parent-Ref field allows each object to remember its parent that has sent to it the first Mark-Object message.

3. A Mark-Object-Reply(o) is processed as follows. The Child-Count of the object o is decremented. If it is zero and the Last-Child of o indicates that there are no more children to be marked, a Mark-Object-Reply is sent to the object's parent indicated by the Parent-Ref field. If the Last-Child of o indicates that there are more children to be marked, the local credit is incremented and Mark-Object messages are sent to the remaining children as long as the credit is positive (the credit is decremented and the Child-Count field is incremented for each message sent). If the Child-Count of o is not zero and Last-Child of o indicates that there are no more children to be marked, more replies are awaited.

4. Each PE completes marking its distributed graph exactly when it receives a Mark-Object-Reply message from its local root to its dummy Parent.

Notice that the above scheme is deadlock free because all storage requirement can be determined and reserved before performing the marking. The credit mechanism guarantees that the marking messages in the system at any instant of time have sufficient storage space.

All reachable objects from each root will be marked because the marking of each graph starts from its root and each object reachable from any root will receive at least one Mark-Object message. The first message to each object marks it.

The marking terminates because each graph consists of a finite number of nodes (i.e. objects) and arcs, and for each arc there is only one Mark-Object/Mark-Object-Reply pair of messages.

One straight forward optimization of the marking phase can be made. Since the task of marking objects in any distributed system represents the largest part of the whole task of the garbage collection, it is worthwhile to improve (i.e. speed up) the marking scheme. We know from this marking scheme that for each arc in any graph there is a pair of Mark-Object/Mark-Object-Reply messages. We can minimize the number of marking messages by avoiding sending Mark-Object messages to already locally marked objects. This in turn avoids sending their respective Mark-Object-Reply messages. Decreasing the number of local Mark-Object/Mark-Object-Reply messages allows using their credits to increase the number of nonlocal Mark-Object/Mark-Object-Reply messages that could simultaneously exist in the system. Increasing the number of simultaneous nonlocal messages decreases a potential idle time of each PE which in turn speeds up the global mark-phase.

So far, we have presented separately, the structure of the global garbage collection scheme (in Section 11), the idea of marking distributed graphs of objects (in Section 12) and the idea of emptying the communication subsystem from computation messages during the global mark-phase (in Section 10.1). The detailed specification of the global mark-phase is found in Appendix C.

13. Memory Sweep-Compaction Phase

This section describes the sweep-compaction phase that completes our global garbage-collection scheme. As described in Section 11.2, the master (highest invoker) starts its local Sweep-Compaction phase and broadcasts a Sweep-and-Compact message to each other PE when the global mark-phase is completed. When a PE processes a Sweep-and-Compact message it starts its local sweep-compaction phase, which compacts all marked objects into one end of the local heap leaving spaces corresponding to unmarked objects in the other end of the local heap, and returns to the free pool all ODT entries associated with unmarked objects.

The usual way of performing heap compaction is to sweep the whole heap from one end towards the other end and investigate all objects. All marked objects will be compacted into one side of the heap leaving a contiguous free space in the other side. But in our case, a heap can be seen as an array of cells. Objects can only be seen at the ODT level, because all information about the local objects are kept in the ODT. That is, sweeping a heap to perform heap compaction requires investigating the local ODT for each object heap-space to know its associated entry for accessing information about the object. To avoid searching the local ODT for each object heap-space, each object heap-space has to keep a pointer (called reverse pointer) which points to its associated ODT-entry. There are two methods to achieve this.

In the first method, one cell per each object heap-space is reserved for this purpose. This cell is initialized to a reverse pointer when the object is created. This requires one additional cell per object heap-space.

The second method does not require the extra space-overhead but requires one complete scanning of the local ODT. During the scanning for each occupied and marked entry (i.e. In-Use field is Busy and Mark-Bit field is On) the first heap-cell of the associated object is saved in the Parent-Ref field of the same entry and a reverse pointer is stored in the cell. For each occupied and unmarked entry, it is enough to store a reverse pointer in the first heap-cell of the associated object, because the whole object will be garbage collected.

Suppose that the second method is used to avoid the extra space requirement; then after initializing reverse pointers the heap is swept from the lowest address towards the highest address by two pointers, say S and F. F points to the beginning of the freed area and S points to the beginning of the area to be scanned. Initially, F and S point to the lowest cell of the heap (see Fig. 3.a). The idea of heap sweeping and compaction is as follows. The heap cell pointed to by S is investigated. If the associated entry is unmarked, it is returned to the free-entry pool and S is advanced by the object size (found in Size field). If the associated entry is marked and there is a free space (hole) below the object heap-space (i.e. S>F as in Fig. 3.b), the representation of the object is moved into the beginning of the free space (i.e. at the position pointed by F), F and S are advanced by the size of the object, the saved cell in the Parent-Ref field is stored back in the first cell of the moved object, and its Loc field is updated to point to the beginning of the new location. If the associated entry is marked and there is no hole below the object's space (i.e. S=F), F and S are advanced by the size of the object. The sweeping terminates when there are no more object heap spaces to be investigated (i.e. S=B as shown in Fig. 3.c). The area from the lowest address up to one cell below F contains the representations of all needed objects while the remaining part of the heap (i.e. from F up to the highest address) represents a free area that is available to the program for creating new objects. Appendix D contains the details of the second method.

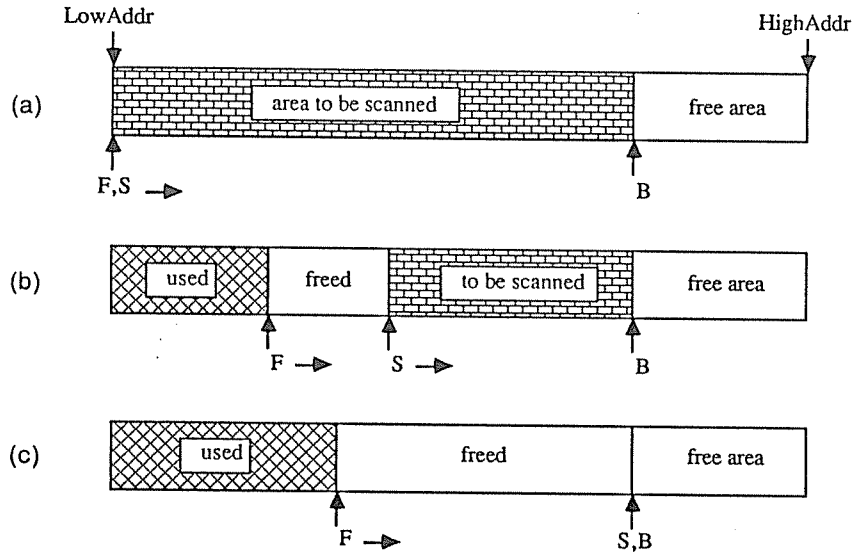

Figure 3: Sweep and compaction of a local heap.

14. Object Storage-System Based on Local-Global Garbage Collection

In this section we extend the previous scheme to avoid invocation of the global garbage-collection each time a PE needs to perform garbage collection. In systems with high locality of reference and with high generation of local garbage, performing one or more local garbage-collections, before a global collection, one may free up enough storage space to continue the normal computation without needing to engage the whole system in a global garbage-collection. Using the local-global scheme, the number of global garbage-collection invocations can be decreased and thereby the long-term system performance will be increased. This is because the space, time and communication overhead of distributed global garbage-collection is much higher than a local sequential uniprocessor garbage-collection. In this scheme, each PE that wishes to perform garbage collection performs first a local garbage-collection. If the freed storage-space is sufficient it continues its normal computation, otherwise it invokes the global garbage-collection. To make a local garbage-collection possible, a PE has to know which of its local objects may be reachable from remote objects. We use the following strategy: ***when a reference to a local object leaves the boundary of its PE store, it is assumed accessible in each local garbage-collection invocation until the next global garbage-collection invocation.***

In the following subsections, we revisit the ODT-entry format, and discuss the modification of the implementation of object operations. Appendix E contains the modifications to our global garbage-collection scheme (i.e. the local-global garbage collector).

14.1. ODT-Entry Format

A one-bit additional field, called Ext, for each ODT entry is needed to indicate whether its associated local object is remotely referenced or not. Ext field is Off when its associated object is only locally referenced, otherwise the bit is On. Initially, all Ext fields of all ODT entries are Off.

14.2. Object Operations

As a rule when a reference to a local object leaves its local heap-boundary, the Ext field associated with that object has to be On. Local garbage-collection does not modify this field, and treats all objects that have their Ext fields On as local roots. Global garbage-collection sets the Ext field of all garbage objects to Off. This means that a local object, which was remotely accessible and becomes garbage, can only be garbage collected in the next global collection. The local collector collects only strictly local garbage.

In some cases, it is known in advance that a reference to a local object leaves the local heap temporarily, e.g. a reference to a suspended process after issuing a remote Create or

Access operation. In such cases according to our mentioned rule, these local objects are not garbage collected until the global collector is invoked. Since the aim of performing local garbage-collection is to minimize the global garbage-collection invocations, it is better to find another way to identify local objects whose references leave the local heap boundaries temporarily and to allow the local collector to garbage collect them.

We solve this problem by assuming that each PE has another queue, called SQ, which contains suspended processes that are temporarily referenced remotely. A reference to a suspended process leaves its local heap-boundary when the current operation must be completed remotely. That reference returns when the operation is completed. Therefore, a Suspend operation (in Appendix A) should insert the current process in the local SQ (without modifying its Ext field), while a Reschedule operation (in Appendix A) should remove that process from the local SQ. The local garbage-collector treats its local SQ as a local root of reachable objects.

The other type of objects whose references may leave a local heap temporarily is the updated object in Create and Access operations (i.e. the object u in Section 5). These objects are referenced from the suspended processes. Thus they are reachable from the SQ. The modifications to the object operations and the extension to the global garbage-collection scheme are found in Appendix E.

15. Performance and Discussion

In this section we discuss the important performance aspects of our global scheme and our local-global scheme. We discuss and determine the space and communication overhead for each scheme. We also discuss the relative performance of our scheme compared to the marking-tree collector [10].

The two primary sources of space overhead in our schemes are the extra space embedded within the graphs of objects and the space required for the garbage-collection queues.

15.1. Control-Flow Mechanisms for Computation Messages

There is an important aspect concerning the space requirement for computation-message queues and the flow-control mechanisms for the computation messages in the system. This is not treated in our work, because we believe that it should be integrated with the communication subsystem or with a system layer above ours. The most important problem is the space requirement for the garbage-collection queues and its flow control mechanism. This problem is solved by our credit mechanism, which regulates the number of marking messages in the system to guarantee that deadlock cannot occur. The space for garbage-collection message queues is discussed and determined in this section.

In all our OSS's presented in this paper we have the following assumption concerning the space of the message queues and the flow-control mechanism. Each PE has a separate space (outside the heap) for the message queues. There is a mechanism that regulates the computation messages in the system such that at any instant of time the available space is as much as the required space for computation messages in transit.

15.2. Object Overhead

Each object requires extra space that is needed for garbage-collection purposes. This extra space is estimated as follows. A single bit is required for each In-Use and Mark-Bit fields. Few bits are required for Child-Count field. The size of this field depends on the maximum number of children of an object. Few bits are required for Last-Child field. The size of this field depends on the maximum size of an object. The size of the Loc field is equal to the number of bits needed to address the highest memory cell in the local heap (i.e. $\log_2 x$ where x is the local-heap size). One object-reference cell is required for each object for the Parent-Ref field. Only a single bit more is required for the Ext field in the last scheme. This means that the extra space required for each object is about two memory cells.

For example, in the Or-Parallel execution of logic programs [16], the average object size is about ten memory cells. Suppose that the local-heap size is 1 Mcells, the maximum object-size is 32 memory cells, maximum number of children is 16 and the memory cell

size is 32 bits. In this case, the space overhead per object is equal to two memory cells (5 bits for Last-Child, 4 bits for next-child, 1 bit for In-Use, 1 bit for Mark-bit, 1 bit for Ext, 20 bits for Loc, and one memory cell for Parent-Ref). This means that for similar systems the additional space requirement is equal to 20%.

15.3. Garbage-Collection Queue Overhead

The second form of space overhead is that required for the garbage-collection queue of each PE. We have mentioned before that the garbage-collection messages are synchronization and marking messages. We first determine the total number of synchronization messages and their space requirement. Secondly we discuss and estimate the total number of marking messages and their space requirement. Finally we discuss and determine the space required for the garbage-collection queue in each PE.

15.4. Synchronization Messages

The total number of synchronization messages is estimated as follows.

$$t = s(n - 1)\,a + (n - s - 1)^2\,b + (n - 1)\,c + (n - 1)\,d$$

where,

- t total number of synchronization messages,
- a Start-Mark-Phase,
- b Shove,
- c End-Mark-Phase,
- d Sweep-and-Compact,
- n the total number of PE's in the system, and
- s the number of normal PE's that simultaneously invoke the global garbage collection. The range of s is from 1 to n - 1.

The number of Shove messages can be reduced if we use the second solution given in Section 11.1. In that case poor locality of references decreases the number of Shove messages considerably. That is when each PE refers to objects in all other PE's, the total number of Shove messages reduces to zero. This number increases with increasing locality of reference. If each PE refers only to local objects the total number of Shove messages is $(n-2)^2$. In this case the local-global scheme is more suitable.

The space required for the synchronization messages in each PE is determined as follows. In the worst case, (n-1) Start-Mark-Phase or Shove messages, and (n-1) End-Mark-Phase messages may simultaneously be received by any PE. The size of a Start-Mark-Phase is larger than the size of a Shove message by the size needed for coding the sender. Therefore each PE requires storage space equal to the size of (n-1) Start-Mark-Phase messages plus the size of (n-1) End-Mark-Phase messages. The size of a Start-Mark-Phase message is equal to the size needed for coding the message type plus the size needed for coding PE's name (i.e. few bits) plus the size needed for the link field. The size of an End-Mark-Phase message is equal to the size needed for coding the message type plus the size needed for the link field.

15.5. Mark Messages

The number of mark messages is determined as follows. The number of Mark-Object messages is equal to the number of Mark-Object-Reply messages which, in the worst case, is equal to the number of reachable arcs in the graphs. The total number is usually less than that because there is no communication overhead for already marked local objects. The amount of this communication overhead that can be reduced depends on the locality of reference. With increased locality of reference communication overhead of marking messages is decreased.

The space required by each PE for the marking messages is determined by the maximum number of these messages that may simultaneously be received by any PE. From Section 11.2 this number is m. The size of a Mark-Object message is larger than the size of a Mark-Object-Reply message. Then each PE requires a space for m Mark-Object messages. The size of each Mark-Object message is the same as the size needed for coding the message type plus the size of two object references (i.e. two memory cells) plus the size needed for the link field.

15.6. Total Space-Requirement for Garbage-Collection Messages

Since both synchronization and marking messages may simultaneously be received by any PE during the mark phase, then the total space requirement for each PE is the space required for the synchronization messages plus the space required for the marking messages. That is, the required space for each garbage-collection queue is determined as follows.

$$f = 2(n - 1)\, g + m\, h$$

where,

- f space of a garbage-collection queue,
- g size of Start-Mark-Phase message, and
- h size of Mark-Object message.

Let us give some hypothetical figures for the space needed for each garbage collection queue. Suppose that n = 64, k = 3 (i.e. m = 192), local-heap size > 1 Mcells, and the size of the link field for any message = one memory cell. Suppose also that the number of types of computation messages = 58. We have only six types of garbage collection messages. Then six bits field can be used for coding both computation and garbage collection messages. Six bits field can be used for coding names of PE's. The room for the garbage collection queue for this example can be estimated by:

$$\begin{aligned} & 2(n - 1)(\text{message-type-field} + \text{PE-name-field} + \text{link-field}) + \\ & m(\text{message-type-field} + 2\ \text{memory-cells} + \text{link-field}) \\ = \ & 2(64 - 1)(12/32 + 1) + 192(6/32 + 3) = 785.25\ \text{memory cells} \end{aligned}$$

This space requirement is negligible compared to the local-heap size.

15.7. Speed

In general, each arc is visited twice; one visit for a Mark-Object message, the other visit is a result of a Mark-Object-Reply message. Messages to already local marked objects are avoided. That is, in our global scheme an arc is visited at most twice.

There is no idle time for any PE before starting the global mark-phase. Any PE starts the garbage collection immediately when it either runs out of space or receives a garbage collection message from any other PE. The idle time for PE's during the global mark-phase decreases by increasing m and by having approximately equal use of the local heaps. Increasing m allows more simultaneous marking messages which in turn decreases the idle time of PE's. High locality of reference and no marking messages to already local objects allow the major part of the credit m to be used for marking nonlocal objects. This increases the number of simultaneously marking messages to nonlocal objects which in turn decreases the idle time of PE's.

Equal use of the local heaps increases the possibility that all PE's invoke the global garbage collection at the same time. It also increases the possibility of all PE's to complete their local mark-phases at the same time. This decreases the idle time of PE's at the end of the global mark-phase.

The idle time of PE's can further be minimized by allowing each PE that has completed its local mark phase to select and execute computation messages or process's operations that do not cause garbage collection invocation.

In a local-global scheme, each PE first performs a fast local garbage collection. Global garbage-collection is invoked only when the local one cannot reclaim enough space. The local garbage-collection is much faster that the global one. When the locality of references in the system increases, the invocation of the global garbage-collection decreases and the overall performance of the system increases.

We have assumed an object reference representation with one level of indirection in order to save the communication costs for updating remote references during memory compaction. This level of indirection introduces a considerable processing overhead

during execution of the user program. This overhead can be reduced if local references are efficiently represented, with no indirections, as in [13].

15.8. Relative Performance

For applications that do not require real-time constraints, running in tens of PE (say 64), and connected by a fast communication network, our global and local-global schemes are more efficient than the marking-tree collector for the following reasons.

First, our schemes allow memory compaction without any communication overhead for updating remote pointers during the memory compaction. The marking-tree collector [10] is not able to perform storage compaction. The copying version of marking-tree collector performs memory compaction [15], but it is much more complex and the communication overhead for updating remote pointers during the memory compaction is very high.

Secondly, the space requirement of our schemes are limited, small and known in advance. The marking-tree collector requires space proportional to the number of arcs is the graph, which is not known in advance.

Third, arcs are visited in the marking tree collector **at least** twice (during the marking phase) because the collector and mutator are working in parallel. The mutator must investigate an arc before using it. That is, if an arc has already been visited twice by the collector, the mutator will visit the same arc twice too. More visits occur when the collector tries to access an object used by the mutator (see below). In our schemes an arc is visited **at most** twice.

Finally, the idle time of PE's in our schemes is negligible compared to the high communication and processing overhead of the locking mechanism in the marking-tree collector. Either the collector or mutator locks an object that it will access to guarantee exclusive access to the shared objects. If an object is already locked by the other, the locked object will be unlocked. This costs more visits to the arcs, which in turn increases the communication and processing overhead. The speed of our schemes, however, can be tuned by adjusting the value of m.

Actually, this comparison is not fair because, the marking-tree collector is a real-time scheme, while our scheme is not. But this discussion may show the complexity and the cost of a real-time scheme in comparison with the respective non real-time one.

16. Conclusion

We have presented two distributed garbage collection schemes; global and local-global. The global scheme engages all PEs in the system in every garbage collection cycle. The local-global scheme is an extended version of the global scheme. It avoids suspending the computation in the entire system in every garbage collection invocation. It allows each PE to perform first its local garbage collection without involving the other PEs, and only invokes the global collector when the freed space by the local collector is insufficient. The cost of this extension is very low, and the local collector is fast as any sequential collector.

Several implementation problems are mentioned/tackled with solutions proposed. The most important problem is the determination of the space required for marking messages of the global collector. We have solved this problem by using a simple decentralized credit mechanism, which keeps the total number of marking messages in the system at any instance of time less than or equal to a certain limit m. The space required for marking messages is determined by m.

Our global scheme can be seen roughly as a combination of a nonreal-time version of the marking-tree collector [10] and our credit mechanism to solve the space problem of the marking messages. In all previously demonstrated distributed garbage collection schemes, there is no solution to this problem.

Our schemes allow memory compaction with no communication cost for updating remote references. One level of indirection is used in our representation of object references. This

level of indirection introduces a considerable processing overhead during execution of the user program. This overhead can be reduced if local references are efficiently represented, with no indirections, as in [13].

The performance of our schemes is estimated and compared with the respective ones. No implementation or simulation of our schemes is done so far.

For computations that have high rate of local/global garbage generation with balanced use of local stores, we expect that our local-global scheme to perform well on systems with a medium number of processors. For computations that have high rate of distributed garbage generation with unbalanced use of local stores, our schemes perform bad as most other schemes.

Acknowledgements
Most of this work was done while the authors were employed at the Department of Computer Systems of the Royal Institute of Technology, Stockholm. We thank Professor Lars-Erik Thorelli, for making this work possible.

Appendix A: Create Operation

In this appendix we give the specification of Create operation described in Section 5.1. A number of auxiliary procedures used in the specification is defined first. The function procedure *CurrentPE* identifies the PE executing the process a. *CurrentProcess* returns a reference to the process a. The procedure *Allocate-Locally*(n,c0,c1,..,cn-1) creates a local object of size n initialized by c0,c1,..,cn-1, and returns a reference to the created object. This procedure will be described, later on, in detail since it invokes the garbage collection. *Update*(u,i,v) stores a reference to the object v in the i^{th} field of the object u. *Suspend* operation switches the processor to select for execution either a received message from MQ or an operation from a ready process in RQ. *Reschedule*(a) inserts the process object a into the local RQ.

The Create operation is specified as follows.

```
Procedure Create(n: Size, c0, c1, ..., cn-1: CellValue, p: PE, u: ObjectRef, i: ObjectOffset)
BEGIN
    var v: ObjectRef
    IF p = CurrentPE THEN
        v := Allocate-Locally(n, c0, c1, ..., cn-1)
        IF u.PeId = CurrentPE THEN          {First situation}
            Update(u,i,v)
        ELSE                                {Second situation}
            SEND Update-and-Ack-Back(u,v,i,CurrentProcess) TO u.PeId
            Suspend
        END IF
    ELSEIF u.PeId = CurrentPE THEN          {Third situation}
        SEND Create-and-Ack-Back(n,c0,c1,..,cn-1,u,i,CurrentProcess) TO p
        Suspend
    ELSE                                    {Fourth and Fifth situation}
        SEND Create-Update-and-Reschedule(n,c0,c1,..,cn-1,u,i,CurrentProcess) To p
        Suspend
    END IF
END Create
```

Here follows the specification of the five types of messages used in the implementation of the Create operation.

```
Message Update-and-Ack-Back(u,v: ObjectRef, i: ObjectOffset, a: ProcessRef)
BEGIN
    Update(u,i,v)
    SEND Reschedule-Msg(a) TO a.PeId
END Update-and-Ack-Back


Message Create-and-Ack-Back(n: Size, c0,c1,..,cn-1: CellValue, u: ObjectRef, i: ObjectRef, a:
ProcessRef)
BEGIN
    var v: ObjectRef := Allocate-Locally(n,c0,c1,..,cn-1)
    SEND Update-and-Reschedule(u,v,i,a) TO u.PeId
END Create-and-Ack-Back


Message Update-and-Reschedule(u,v: ObjectRef, i:ObjectOffset, a: ProcessRef)
BEGIN
    Update(u,i,v)
    Reschedule(a)
END Update-and-Reschedule
```

```
Message Create-Update-and-Reschedule(n: Size, c0,c1,..,cn-1: CellValue, u:ObjectRef, i: ObjectOffset, a:
ProcessRef)
BEGIN
    var v: ObjectRef := Allocate-Locally(n,c0,c1,..,cn-1)
    IF u.PeId = CurrentPE THEN
        Update(u,i,v)
        SEND Reschedule-Msg(a) TO a.PeId
    ELSE
        SEND Update-and-Ack-Back(u,v,i,a) TO u.PeId
    END IF
END Create-Update-and-Reschedule


Message Reschedule-Msg(a: ProcessRef)
BEGIN
    Reschedule(a)
END Reschedule-Msg
```

In our specification, an ODT is represented as an array of entries and a local heap as an array of cells (Heap). The Update procedure is specified as follows.

```
Procedure Update(u: ObjectRef, i: ObjectOffset, v: ObjectRef)
BEGIN
    var addr: Pointer
    var entry: ODT-Entry
    entry := ODT[u.Index]
    addr := entry.Loc
    Heap[addr + i] := v                    {store v in the i^th cell of object u}
END Update
```

We now turn our attention to the procedure Allocate-Locally. Let us first assume that the free pool of the local ODT's entries has its last entry pointing to a dummy entry, called *Nil-Index* (see Figure 4). The local heap has the lowest address, called *LowAddr*, and the highest address, called *HighAddr*. A pointer, called *B*, points to the beginning of the free part of the local heap; initially B is equal to LowAddr, and the whole heap represents a free space (see Figure 5).

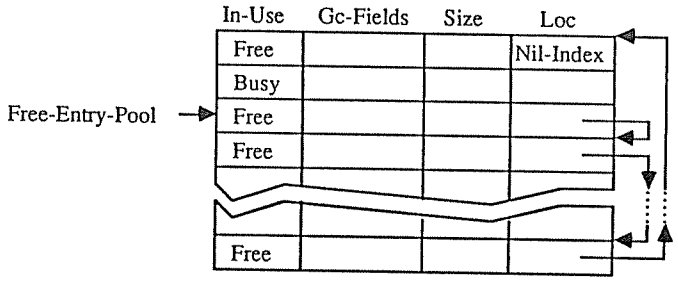

Figure 4: Object descriptor table

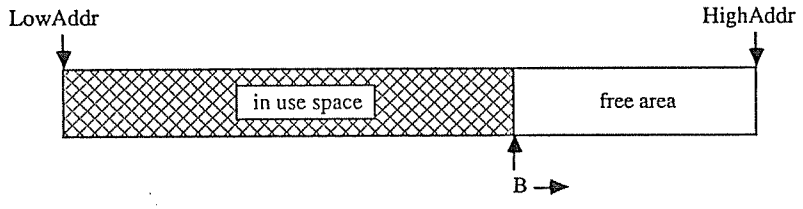

Figure 5: Local heap

The Allocate-Locally(n,c0,c1,..,cn-1) works as follows. If there are more than n contiguous free cells in the local heap and there is a free entry in the local ODT, then the first n cells of the free space are allocated and initialized, and an object reference to the created object is returned. Otherwise the current local operation (whether Create or a message operation) is undone by the procedure Undo-Current-Operation and the garbage collection is started. (The mechanism needed for Undo-Current-Operation is similar to that used in virtual memory management when an instruction tries to access information which is not yet in the main memory.) The current operation will be retried after the garbage collection.

```
Procedure Allocate-Locally(n: Size, c0,c1,..,cn-1: CellValue) RETURNS v: ObjectRef
BEGIN
    var i: Entry-Index := Free-Entry
    IF HighAddr - B > n  AND  i ≠ Nil-Index THEN
                                        {there is a free entry and enough free space}
        FOR j := 0 TO n-1 DO Heap[B+j] := cj END FOR
        ODT[i].In-Use := Busy           {initialize ODT entry}
        ODT[i].Size := n
        ODT[i].Loc := B
        B := B + n                      {advance B}
        RETURN(Mk-ObjectRef(R,CurrentPE,i))
    ELSE                                {save current operation and start garbage collection}
        IF i ≠ Nil-Index THEN
            Add-Free-Entry(i)           {return back entry of index i}
        END IF
        Undo-Current-Operation
        Start-Global-Collector
    END IF
END Allocate-Locally
```

Free-Entry returns either an index to a free entry of the local ODT or Nil-Index. Add-Free-Entry(i: Entry-Index) adds a free ODT-entry to the local ODT. Mk-ObjectRef(r: Tag, p: PE, i: Entry-Index) returns an object reference-cell that contains r as a Tag, p in the PeId field, and i in the Index field.

Appendix B: Access Operation

This appendix describes the operation Access(v,u: ObjectRef, j,i: ObjectOffset) briefly. Let us assume that the object v resides in the PE p, the process executing this operation (called a) in p1, and the object u in p2. One of the following five situations may occur:

1. The process a and the objects v and u are in the same PE, (p=p1=p2).
2. The process and the object v are in the same PE, the object u is in another PE, (p=p1≠p2).
3. The process and the object u are in the same PE, and the object v is in another PE, (p1=p2≠p).
4. The objects u and v share the same PE, but the process is in another PE, (p=p2≠p1).
5. The process, the object u and v all are in different PE's, (p≠p1≠p2≠p).

Besides the types of messages listed in Section 5.1 the following ones are used in the implementation of Access operation.

a. Access-and-Ack-Back(u,v: ObjectRef, i,j: ObjectOffset, a: ProcessRef)
b. Access-Update-and-Reschedule(u,v: ObjectRef, i,j: ObjectOffset, a: ProcessRef)

Let us denote the content of the j^{th} cell of the object v by v_j and the i^{th} cell of the object u by u_i.

In the first situation, v_j is copied into u_i, and the operation is completed locally.

In the second situation (p=p1≠p2), the message Update-and-Ack-Back(u,vj,i,a) is sent to p2, and the current process a is suspended. When this Update-and-Ack-Back message is processed by p2, Reschedule-Msg(a) is sent back to p1. Processing Reschedule-Msg(a), as explained above, completes the operation.

In the third situation (p1=p2≠p), the message Access-and-Ack-Back(u,v,i,j,a) is sent to p. The sent message requests p to read the j^{th} cell of the object v and send it back. When this request is processed by p, the reply message Update-and-Reschedule(u,vj,i,a) is sent back to p1. When this Update-and-Reschedule is processed, the access operation will be completed.

In the fourth situation (p=p2≠p1), the remote request Access-Update-and-Reschedule (u,v,i,j,a) is sent to p and the current process is suspended. When this request is processed by p, vj is stored in the i^{th} cell of the object u, and Reschedule-Msg(a) is sent back to p1. Processing of the Reschedule-Msg completes the operation.

In the fifth situation (p≠p1≠p2≠p), the remote request Access-Update-and-Reschedule (u,v,i,j,a) is sent to p and the current process is suspended. When this request is processed by p the message Update-and-Ack-Back(u,vj,i,a) is sent to p2. When this request is processed by p2 and Reschedule-Msg(a) is sent and processed by p1, the access operation will be completed.

Appendix C: Specification of the Marking Phase

This appendix presents the detailed specification of the global mark-phase described in Section 12. We use the first solution in Section 10.1 to empty the communication subsystem from computation messages.

Before invoking the garbage collector, the Mark-Bit field for every busy ODT entry is Off, the Child-Count field is zero, and the Last-Child field is set to the size of the object.

In the following specification we specify the task of the highest invoker, any other invoker, and a non-invoker, separately. Any PE in the system can behave as one of those depending on the its situation in the current cycle of the garbage collection. It should be noted that each PE has an identical algorithm. Each PE has the following constants and variables.

Constants

Parent: Object-Ref	{index entry of the local dummy object}
K: Integer	{initial value of local credit}
N: Integer	{number of PE's in the system}
LowPe: PE	{a dummy PE name with lowest priority}

Variables

credit: Integer := K	{maximum number of Mark-Object/Mark-Object-Reply messages that can simultaneously be sent by the PE}
RQ: ObjectRef	{refers to the first object in the local queue}
MQ: ObjectRef	{refers to the computation message in the queue of computation messages}
current-root: Pointer := NIL	{pointer keeps track of the computation message to be investigated from the MQ}
data-offset: Integer := 0	{keeps track of the next cell of the data part of the computation message referenced by current-root}
root-count: Integer := 0	{counts the number of Mark-Object messages that have been sent to the local roots and not yet replied}
root-flag:= false	{more roots have been received}
start-count: Integer := 0	{counts the number of Start-Mark-Phase or Shove messages received from other PE's}
end-count: Integer := 0	{counts the number of End-Mark-Phase messages received from other PE's}
master: PE := LowPe	{the highest master so far}

C.1. Transitions of Highest Invoker

Figure 6 shows the state transition diagram of the master. The transitions are specified as follows.

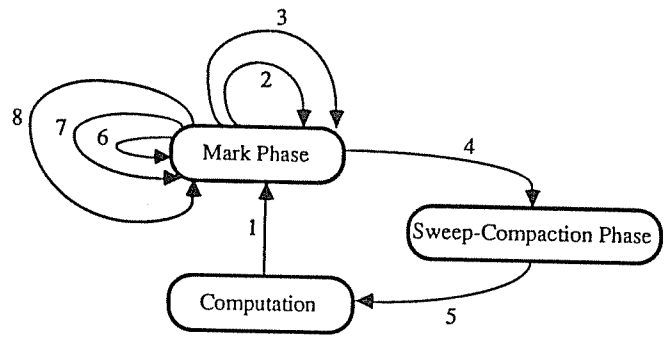

Figure 6: The state transition diagram of the highest master.

```
TN: 1
EP: no more free storage space
Action:
BEGIN
    SEND Start-Mark-Phase(CurrentPE) TO each other PE
    master := CurrentPE
    Start-Marking-Local-Roots
END TN: 1
```

where Start-Marking-Local-Roots is defined as follows.

```
Procedure Start-Marking-Local-Roots
BEGIN
    SEND Mark-Object(RQ,Parent) TO RQ.PeId
    Decrement(credit)
    Increment(root-count)
    current-root := first message is MQ
    data-offset := first reference cell to be investigated
    Mark-MQ-Roots
END
```

Mark-MQ-Roots marks all the roots that is reachable from the message queue. It is defined as follows.

```
Procedure Mark-MQ-Roots
BEGIN
    var x: ObjectRef
    WHILE credit > 0 AND There-are-More-MQ-Roots DO
        x := Next-MQ-Root
        SEND Mark-Object(x,Parent) TO x.PeId
        Decrement(credit)
        Increment(root-count)
    END WHILE
END Mark-MQ-Roots
```

The procedure There-are-More-MQ-Roots uses current-root and data-offset to check if there are more MQ roots to be marked. The procedure Next-MQ-Root returns an object reference and advances data-offset and current-root to the next root.

```
TN: 2
EP: reception of a Mark-Object(o,p) message
Action:
BEGIN
    IF Marked(o) THEN
        SEND Mark-Object-Reply(p) TO p.PeId
    ELSE
        Mark(o)
        IF Leaf(o) THEN
            SEND Mark-Object-Reply(p) TO p.PeId
        ELSE
            Increment(credit)
            ODT[o.Index].Parent-Ref := p
            Mark-Children(o)
        END IF
    END IF
END
```

Mark-children avoids sending Mark-Object messages to already locally marked objects.

```
Procedure Mark-Children(o: Object-Ref)
BEGIN
    var c: ObjectRef
    WHILE credit > 0 AND there are more children of o to be marked DO
        c := Next-Child(o)
        IF Local(c) THEN
            IF NOT Marked(c) THEN
                Mark-Child(c,o)
            END IF
        ELSE
            Mark-Child(c,o)
        END IF
    END WHILE
END Mark-Children
```

Next-Child(o) uses ODT[o.Index].Last-Child to return an Object-Ref of the next child; it also decrements the Last-Child field of the object o, skipping value cells of o. The test that there are more children of o is ODT[o.Index].Last-Child > 0. Local(x) returns true when x refers to a local object, false otherwise.

```
Procedure Mark-Child(c,o: ObjectRef)
BEGIN
    SEND Mark-Object(c,o) TO c.PeId
    decrement(credit)
    Increment(ODT[o.Index].Child-Count)
END Mark-Child
```

```
TN: 3
EP: reception of a Mark-Object-Reply(o) message
Action:
BEGIN
    var p: Object-Ref
    Increment(credit)
    IF o = Parent THEN
        Decrement(root-count)
        Mark-MQ-Roots
    ELSE
        Decrement(ODT[o.Index].Child-Count)
        IF o has more children to be marked THEN
            Mark-Children(o)
        ELSEIF ODT[o.Index].Child-Count = 0 THEN
            Decrement(credit)
            p := ODT[o.Index].Parent-Ref
            SEND Mark-Object-Reply(p) TO p.PeId
        ELSEIF root-flag THEN
            root-flag := false
            Mark-MQ-Roots
        END IF
    END IF
END TN: 3
```

The last case needs an explanation; root-flag is set true when there are more local roots to be marked but there is not enough credit. Now because the credit is guaranteed positive we can continue marking the local root.

```
TN: 4
EP: end-count = N-1 AND root-count = 0 AND there are no more local roots to be marked
Action:
BEGIN
    start-count := 0
    end-count := 0
    master := LowPe
    SEND Sweep-and-Compact TO each other PE
    Perform-Local-Memory-Sweep-and-Compaction
END TN: 4

TN: 5
EP: Sweep-Compaction phase completed
Action: Resume local computation
TN: 6
EP: reception of a Start-Mark-Phase(sender) message
Action:
BEGIN
    Increment(start-count)
    master := Higher(master, sender)
    Check-Received-Roots
END TN: 6
```

Higher(p1,p2) returns the PE name of the higher priority of p1 and p2.

```
Procedure Check-Received-Roots
BEGIN
    IF start-count = N-1 AND
        root-count = 0 AND there are more local roots to be marked THEN
        IF credit > 0 THEN
            Mark-MQ-Roots
        ELSE
            root-flag := true
        END IF
    END IF
END Check-Received-Roots


TN: 7
EP: reception of a Shove message
Action:
BEGIN
    Increment(start-count)
    Check-Received-Roots
END TN: 7


TN: 8
EP: reception of an End-Mark-Phase message
Action:
BEGIN
    Increment(end-count)
END TN: 8
```

C.2. Transitions of any Other Invoker

A PE that invokes garbage collection and is not the highest invoker behaves according to the following specification (see Figure 7).

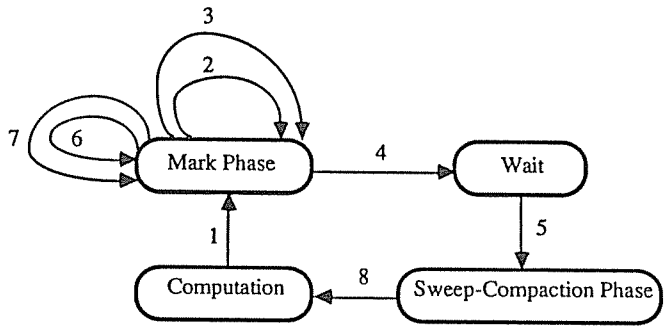

Figure 7: The state transition diagram of any other invoker.

TN: 1, 2, 3, 6, 7 the same as in the highest invoker in Section C.1.

TN: 4
EP: start-count = N -1 AND root-count = 0 AND there are no more local roots to be marked
Action:
BEGIN
    start-count := 0
    SEND End-Mark-Phase TO master
    master := LowPe
END TN: 4

TN: 5
EP: reception of Sweep-and-Compact message.
Action: Perform-Local-Memory-Sweep-and-Compaction

TN: 8
EP: Sweep-Compaction phase completed
Action: Resume local computation

C.3. Transitions of any non-invoker

A PE that has received one or more garbage collection messages when it is in the computation state behaves according to the following specification (see Figure 8). The difference between the task of a non-invoker and any other invoker is only in TN 1 and TN 9.

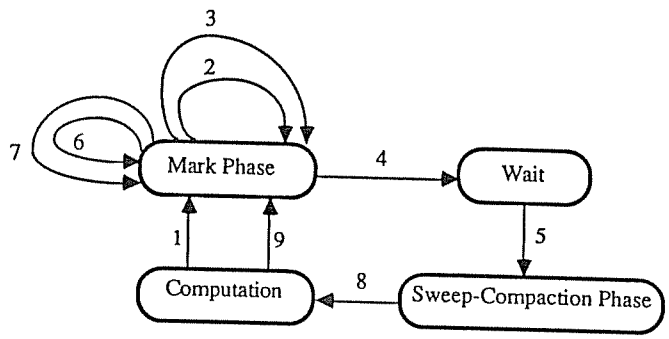

Figure 8: The state transition diagram of any non-invoker.

```
TN: 1
EP: reception of Start-Mark-Phase(sender) message
Action:
BEGIN
        master := sender
        Increment(start-count)
        SEND Shove TO each other PE
        Start-Marking-Local-Roots
END TN: 1

TN: 9
EP: reception of Shove message
Action:
BEGIN
        Increment(start-count)
        SEND Shove TO each other PE
        Start-Marking-Local-Roots
END TN: 9
```

Appendix D: Specification of the Sweep-Compaction Phase

This appendix completes the specification of the global garbage collection scheme specified in Appendix C. It specifies the second method described in Section 13. The local heap is considered below as an array of cells. In addition to constants and variables in Appendix C, each PE has the following constants and variables.

Constants

```
LowAddr: Address          {the address of the lowest cell of the local heap}
HighAddr: Address         {the address of the highest cell of the local heap}
```

Variables

```
Heap: ARRAY LowAddr..HighAddr of Cell
B: Address                {points to the first cell of the free space in the local heap}
```

The procedure Perform-Local-Memory-Sweep-and-Compaction mentioned in Appendix C (TN: 5 in Section C.1, and TN: 4 in Section C.2) is specified as follows.

```
Procedure Perform-Local-Memory-Sweep-and-Compaction
BEGIN
    Reverse-Pointers
    Memory-Compaction
END

Procedure Reverse-Pointers
BEGIN
    FOR i := 0 TO (Size-of-ODT - 1) DO
        IF ODT[i].In-Use = Busy THEN
            IF ODT[i].Mark-Bit = On THEN
                ODT[i].Parent-Ref := Heap[ODT[i].Loc]
            END IF
            Heap[ODT[i].Loc] := i
        END IF
    FOR END
END Reverse-Pointers
```

```
Procedure Memory-Compaction
BEGIN
    var F,S: Pointer
    var e: ODT-Index
    F := S := LowAddr                          {F, S point to the lowest cell}
    WHILE S<B DO                               {S scans the used storage space}
        BEGIN
            e := Heap[S]                       {get the associated entry}
            IF ODT[e].Mark-Bit = On THEN
                Heap[S] := ODT[e].Parent-Ref   {restore the first cell}
                ODT[e].Last-Child := ODT[e].Size   {reset the Last-Child}
                ODT[e].Mark-Bit := Off         {turn off the Mark-Bit}
                IF F=S THEN                    {hole between F and S?}
                    F := S := S + ODT[e].Size  {No, advance F and S}
                ELSE                           {Yes}
                    FOR i := 0 TO ODT[e].Size -1 DO
                        Heap[F + i] := Heap[S + i]   {move object}
                    END FOR
                    ODT[e].Loc := F            {point to new location}
                    F := F + ODT[e].Size       {advance F}
                    S := S + ODT[e].Size       {advance S}
                END IF
            ELSE                               {unmarked object}
                S := S + ODT[e].Size           {advance S only}
                ODT[e].In-Use := Free          {reset In-Use field}
                Add-Free-Entry(e)              {return freed entry to pool}
            END IF
    END WHILE
    B := F                                     {make B points to the beginning of freed
                                                space}
END Memory-Compaction
```

Appendix E: OSS Based on Local-Global Garbage Collection

In this appendix we mention first the modifications to the object operations presented in Appendices A and B, and then the modifications to the global garbage collector presented in Appendix C.

E.1. Object Operations

Create(n,c0,c1,...,cn-1,p,u,i): In case of a remote creation (second, third, and fifth situations in Section 6.1), each ci is investigated, if it is a local reference then its Ext is set (to On). When the message Create-and-Ack-Back is processed, the Ext field of the created object is set. Finally when the message Create-Update-and-Reschedule is processed and a reference to the created object is sent away, then its Ext field is set.

Access(u,v,j,i): Similarly, in the second, third and fifth situation (in Appendix B), the updated object u and the accessed object v do not reside on the same local heap. In these three situations only if the accessed cell v_j refers to a local object, then the Ext field associated with that referenced local object must be set.

We mention bellow the modifications to Suspend, Reschedule, and Alocate-Locally procedures described in Appendix A.

Suspend: Inserts the current process into the local SQ and performs the same task as before.

Reschedule(a): Removes the target process from the local SQ, and performs the same task as before.

Allocate-Locally(n,c0,...,cn-1): If there is not enough local storage-space to create the requested object or there is no free ODT-entry, the local garbage-collector is invoked first and if the freed storage-space and ODT entries cannot allow creation the requested object then the global garbage collector is invoked.

E.2. Local-Global Garbage Collector

In this scheme, a PE that wishes to perform garbage collection (invoker) first performs the local garbage-collection. If the freed storage-space is large enough the invoker continues its normal computation, otherwise it invokes the global garbage-collection. Figure 9 shows the modifications to the state transition diagrams shown in Figures 6 and 7.

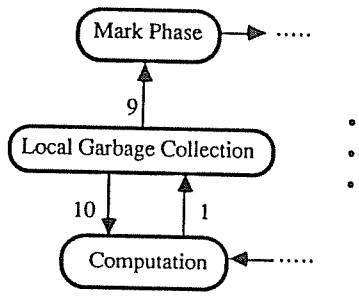

Figure 9: The modified invoker state transition diagram.

TN: 1
EP: no more free storage space
Action:
BEGIN
    perform local garbage-collection
END TN: 1

Any local garbage-collection scheme can be used, but in [11] we have used a sequential mark-sweep-compact scheme. In this case, the local roots are RQ, SQ, and all ODT

entries that have their Ext fields set. We use a sequential marking algorithm [17] for marking local objects and use the same local sweep and compaction algorithm described in Section 13 and in Appendix D.

```
TN: 9
EP: freed space is not enough
Action
BEGIN
    SEND Start-Mark-Phase(CurrentPE) TO each other PE
    master := CurrentPE
    Start-Marking-Local-Roots
END TN: 9
```

This action is exactly the same as in TN 1 of any invoker in Appendix C. The only modification is that before returning a freed ODT entry to the free pool list, its Ext field is changed to Off.

```
TN: 10
EP: freed space is enough
Action: resume local computation
```

17. References

[1] A. Bechtolsheim, K-H Lai, J. Ousterhout, and R.J. Swan "The Implementation of the CM* Multiprocessor", Proceedings of the National Computer Conference 1977.

[2] H. Fuller, D.P. Siewiorek, and R.J. Swan, "CM* - A Modular Multiprocessor", Proceedings of the National Computer Conference, 1977.

[3] M. Keller, G. Lindstrom, and S. Patil "A loosely coupled applicative Multiprocessor System" in Proc. AFIPS National Computer conference Vol. 48, pp. 613-622, 1979.

[4] P. Harbison, R. Levin, and W.A. Wulf "Hydra/C.mmp: An Experimental Computer System", McGraw-Hill, 1981.

[5] Arvind, and R.A. Iannucci, "A Critique of Mutiprocessing Von Neumann Style", Proceedings of the 10th symposium on Computer Architecture, pp. 426-436, 1983.

[6] A. Ciepielewski and S. Haridi "Control of Activities in the Or-Parallel Token Machine", Proceeding of 1894 International Conference on Logic Programming Atlantic City, Feb. 1984.

[7] P.L. Wadler, "Analysis of an Algorithm for Real-Time Garbage Collection, CACM 19, 9 pp. 491-500, Sep 1976.

[8] E.W. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens, "On the fly Garbage Collection: An Exercise in Cooperation", in Lecture Notes in Computer Science, No 46, Springer-Verlag, 1976; also appeared in CACM 21, 11, pp. 966-975, 1978.

[9] G. L. Steele "Multiprocessing Compactifying Garbage Collection", CACM 18, 9, pp. 495-508 Sep 1975.

[10] P. Hudak and R. Keller "Garbage Collection and Task Deletion in Distributed Applicative Processing Systems", Proceeding of the ACM Symposium on LISP and Functional Programming" Pittsburgh Aug. 1982.

[11] K. A. M. Ali, "Object Oriented Storage Management and Garbage Collection in Distributed Processing Systems", Ph D Th Rep. TRITA-CS, Royal institute of Technology Stockholm, 1984.

[12] H. G. Baker "List-processing in Real-time on a Serial Computer", CACM 21, 4, pp. 280-294, April 1978.

[13] J. Hughes "A Distributed Garbage Collection Algorithm", Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, No 201, Springer-Verlag, pp. 256-272, Sep. 1985.

[14] G.V. Bochmann and J. Gecsei, "A Unified Method for the Specification and Verification of Protocols", Proceedings IFIP Congress, pp. 229-234, Toronto 1977.

[15] P. Hudak "Object and Task Reclamation in Distributed Applicative Processing Systems", Ph.D. th., Department of Computer Science, University of Utah, July 1982.

[16] A. Ciepielewski, "Towards a Computer Architecture for Or-Parallel Execution of Logic Programs", Ph D Th Rep. TRITA-CS, Royal institute of Technology Stockholm, 1984.

[17] L-E Thorelli, "Marking Algorithms," Bit 12, 4, pp. 555-568, 1972.