An Algorithm for Garbage Collection in Multicomputer Systems

A Thesis

Submitted to the Faculty of Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of

Master of Science

in the

Department of Electrical Engineering

University of Saskatchewan

by

Eshwari Prasad Komarla

Saskatoon, Saskatchewan

September 1988

Dedicated to my

beloved parents

Distribution Notice

The author has agreed that the Library, University of Saskatchewan, may make this thesis freely available for inspection. Moreover, the author has agreed that permission for extensive copying of this thesis for scholarly purposes may be granted by the professor or professors who supervised the thesis work herein or, in their absence, by the Head of the Department or the Dean of the College in which the thesis work was done. It is understood that due recognition will be given to the author of this thesis and to the University of Saskatchewan in any use of the material in this thesis. Copying or publication or any other use of this thesis for financial gain without approval by the University of Saskatchewan and the author's written permission is prohibited.

Requests for permission to copy or make any other use of material in this thesis in whole or in part should be addressed to:

Head
Department of Electrical Engineering,
University of Saskatchewan,
Saskatoon, Saskatchewan,
S7N 0W0, Canada.

Acknowledgements

I would like to take this opportunity to thank all those who were instrumental in successfully completing this work. First and foremost I would like to thank my supervisors Professors Carl McCrosky and Ron Bolton for their able guidance and useful suggestions. My sincere regards to Professor Carl McCrosky who was more than a supervisor. He was always ready for suggestions and advice, his informal discussions, and all his help in the course of this work are thankfully acknowledged.

I would like to thank my family and friends for their encouragement and assistance in pursuing my higher studies. I am grateful to my parents who always believed in my strengths and encouraged me all through these years in building up my career. My thanks to both my sisters for their help and good wishes. My thanks to all friends both here and back home for their co-operation and useful suggestions.

University of Saskatchewan

Electrical Engineering Abstract 88A294

An Algorithm for Garbage Collection in Multicomputer Systems

Candidate: Eshwari Prasad Komarla

Supervisors: C.D. McCrosky and R.J. Bolton

M.Sc. Thesis presented to the College of Graduate Studies

University of Saskatchewan

September 1988

Abstract

There is widespread interest in multicomputer parallelism. Functional languages with their inherent parallelism can form a basis for programming these machines. These languages dynamically allocate memory, objects are created when required and eventually objects may lose links with active objects and become unreachable garbage. The process of recovering these inactive objects is called garbage collection. Garbage collection in a multicomputer system has to manage objects in physically separated memories. This introduces consistency and synchronization problems with the shared data.

The development of an algorithm for garbage collection in a multicomputer system is reported. A description of the algorithm is presented. The design of the simulator and the simulation experiments are presented. The application of Petri nets to the modelling of this algorithm is discussed. Verification of some properties of this algorithm using the invariants of the Petri net model are presented.

Table of Contents

List of Figures

CHAPTER 1
Introduction

1.1 Models of Multicomputer Systems

Recently there has been a shift in research interest from centralized serial computing machines to parallel machines in order to exploit potential parallelism in programs [33, 51]. There are two well known types of parallel computer systems namely SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data). A SIMD computer consists of a control unit, N processors and N memory modules. Processors and memories are connected by an inter-connection network. The control unit broadcasts instructions to the processors, and all processors execute the same instruction at the same time on the data stored in their associated memory modules. There are two models of MIMD computer systems. In the shared-memory model (also called *tightly-coupled*), shown in Fig. 1.1, data is stored in a shared memory, which can be accessed by all processors through an inter-connection network (for example the Ultra Computer [24]). In a message-passing model (also called *loosely-coupled*), shown in Fig. 1.2, each processor has an associated local memory, and data is passed from the producing processors to the consuming processors through an inter-connection network.

The parallelism that can be achieved in a shared memory multi-computer system is constrained because of the need to access shared data. Loosely-coupled multicomputer systems require slicing of problems into tasks that minimize communications between processors. Information is exchanged via message passing. Each task must be scheduled for execution on one or more processors. Synchronization of control and data flow is performed during execution. In this thesis we consider only a loosely-coupled message-passing system.

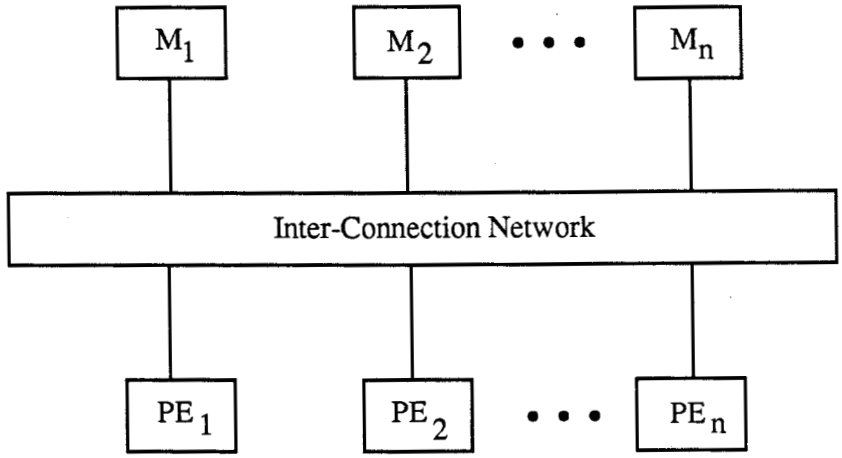


Figure 1.1 Model of a Tightly Coupled Multicomputer System.

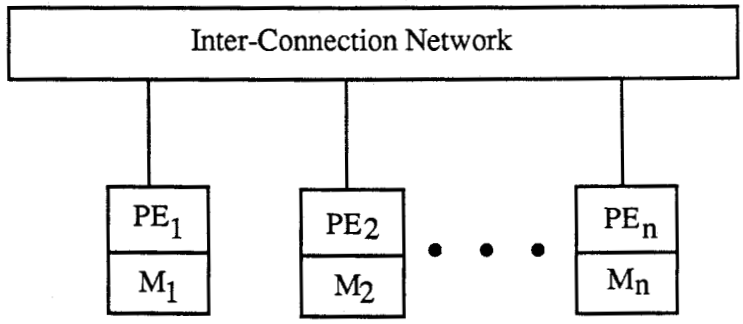


Figure 1.2 Model of a Loosely Coupled Multicomputer System.

1.2 Introduction to basic Data Structures

In this section we present definitions of some frequently mentioned data structures in the thesis. Graphs and trees are the most important data structures required for our purpose.

1.2.1 Graphs

Definition: A *graph* $G = (V, E)$ consists of a set V of vertices (points) and a set

of E of edges (arcs) connecting pairs of vertices. Fig. 1.3 illustrates some examples of graphs. In Fig. 1.3(a), edges *p and q* form loops with vertex *a*. Another important concept in graphs is that of a *path.* In a graph G, a path from V_0 to V_n is the sequence $V_0, E_0, V_1, E_1, \ldots V_{n-1}, E_{n-1}, V_n$ where $E_i = (V_i, V_{i+1})$, an edge between vertices V_i and V_{i+1}. The length of a path is the number of edges traversed from the source vertex to the destination vertex. A path is a *closed path* (circuit) if $V_0 = V_n$, i.e. the source and terminating vertices are the same. A path which is closed and repeats no edges or vertices is a *cycle.* A graph is *directed,* if every edge has direction.

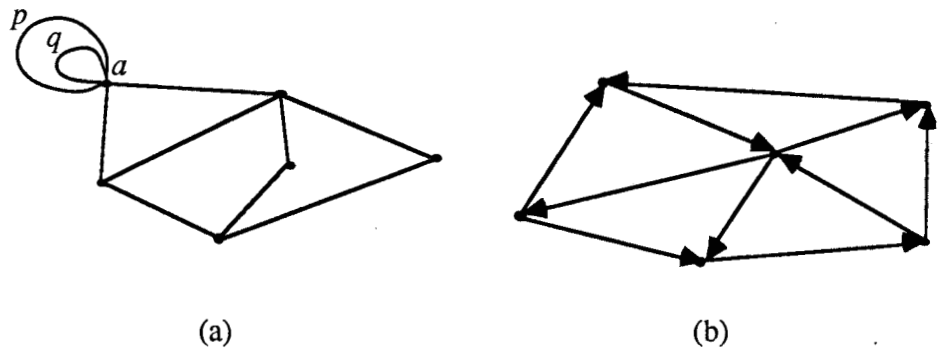


Figure 1.3 Examples of graphs: (a) Undirected graph (b) Directed Graph.

1.2.2 Trees

A *connected* and *acyclic* graph is called a *tree.* In a tree each pair of distinct vertices is connected by exactly one simple path (not circuit). A rooted tree is one in which exactly one vertex has been designated as its *root.* A binary tree is a rooted tree in which every interior vertex has at most two children. Fig. 1.4 shows some examples of trees, Fig. 1.4(d) shows a binary tree.

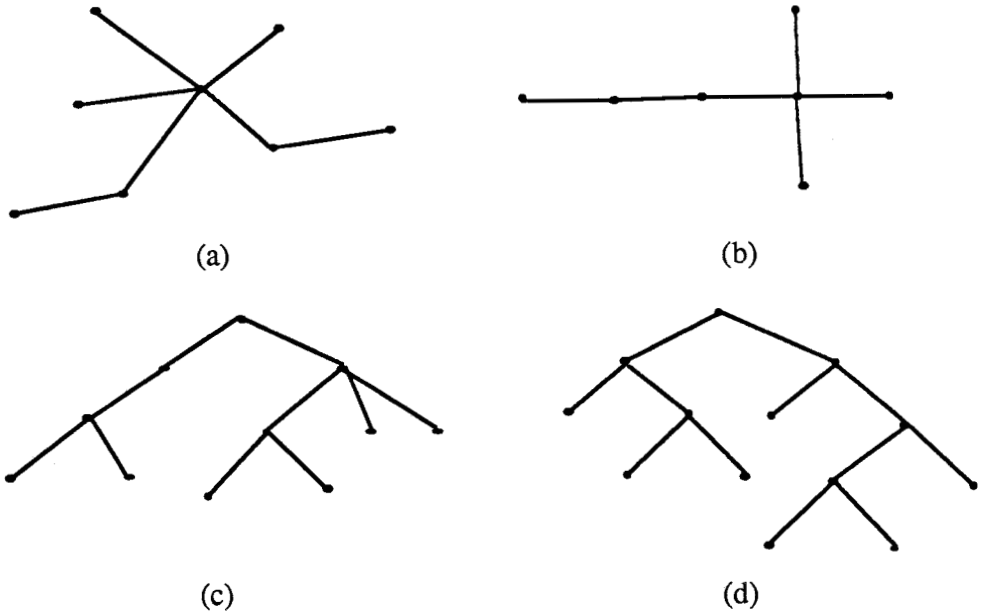


Figure 1.4 Some examples of trees, (d) is a binary tree.

1.3 Graph Reduction

Functional programming languages appear to be a natural way to program multi-computer systems. The main advantages of functional languages is their semantic simplicity and their avoidance of unnecessary sequentiality. This facilitates the implementation of multiple instruction, multiple-data (*MIMD*) parallelism. A functional programming language achieves this goal by separating the task that the program is to perform from the way that the computer is to do it. That is, unlike imperative languages, functional languages do not specify the flow of control but only the flow of data in the program.

Most functional programming languages are quite similar to each other, and differ more in their syntax than their semantics. A functional language can be implemented in three steps. The first step is the translation of high level functional program into an intermediate language. The intermediate language is the notation of lambda calculus [44].

Lambda expressions can be implemented in three ways: *string reduction*, *environment interpretation* and *graph reduction.* In string reduction, the program is represented as one long string of lambda expression, and numerous parts of the string can be evaluated at once. In an environment interpreter, the code sequence for the lambda abstraction has access to an environment which contains values for each of the variables. Binding of actual parameters to formal parameters is done in the environment.

There are two strategies in the implementation of any functional programming language, *strict* (Normal Order) and *non-strict* (lazy) evaluation. In strict evaluation, the evaluation of arguments of a function is done first before the invocation of the function, whereas in lazy evaluation arguments are evaluated only when it is necessary. Environment interpretation approach is more suited for the implementation of languages with strict semantics (such as ML [23] and Hope [14]).

In graph reduction, lambda calculus expressions with variables are converted into *CAFs (Constant Applicative Forms).* These CAF expressions are also called *combinators* or *supercombinators.* The transformation to supercombinators is called *lambda lifting.* The transformed program is optimized and compiled into a linear instruction sequence. This compiled code is represented as a cyclic graph. Cycles appear in the graph because of recursive functions and data structures in the program. The evaluation of the program corresponds to the reduction of this graph. Hence this evaluation is called *graph reduction.* Graph reduction is the most natural way to implement languages with non-strict semantics (*lazy evaluation*). Examples of languages that use this strategy are SASL [52], Ponder [21, 22], LML [5], Miranda [53] and Orwell [54].

Suppose a function f is defined (e.g. in Miranda, a modern functional language [53]) like this:

$f\ x\ y = (x + 3) * (y - 2)$

This definition specifies that f, a function of two arguments x and y, computes $(x + 3) * (y - 2)$. Let this function be evaluated for the values $x = 3$ and $y = 5$, i.e. function f applied to 3 and 5. The application of function f may be graphically represented as shown in Fig. 1.5.

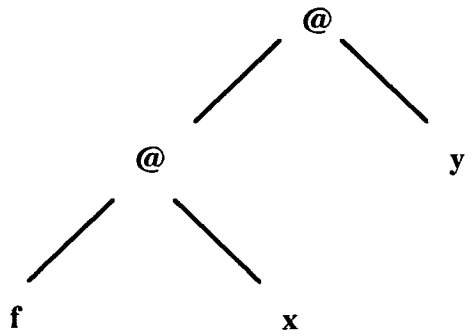


Figure 1.5 Graphical representation of application of *f to x y*

This tree denotes the expression $f\ x\ y$ (Fig. 1.5). The '@' sign is called the tag of the node, and indicates that the node is an application. The application of this function for the given values of x and y corresponds to the reduction of the graph shown in Fig. 1.6.

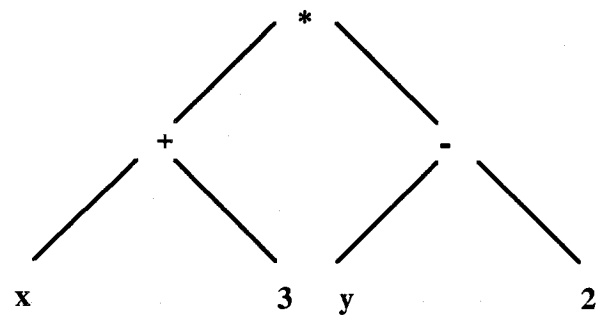
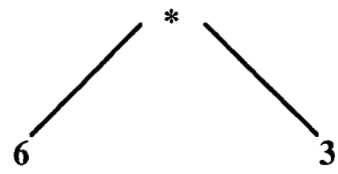


Figure 1.6 Graph of *f x y = (x + 3) * (y - 2)*

We may execute the addition and subtraction in either order resulting in the graph

shown in Fig. 1.7. Finally we can execute the multiplication, to give the result



Result = 18

Figure 1.7 Graph after reductions *(x + 3)* and *(y - 2)*

The evaluation of a function has been transformed into graph reduction. Graph reduction gives an appealingly simple and elegant model for the execution of a functional program, and one that is radically different from the execution model of a conventional imperative language. Some of the salient features of graph reduction are:

i) Executing a functional program consists of *transforming* an expression.

ii) A functional program has a natural representation as a *tree* (or more generally a *graph*).

iii) Evaluation proceeds by means of a sequence of simple steps, called *reductions*. Each reduction performs a local transformation of the graph (hence the term *graph reduction*).

iv) Reductions may safely take place in a variety of orders, or indeed in parallel, if they do not interfere with each other.

v) Evaluation is complete when there are no further reducible expressions. Normally this results in the production of an answer.

1.4 Storage Management

As reduction proceeds we will need to build new pieces of graph, due to new function invocations. In order to do so we have to allocate new cells (each vertex in the graph is referred to as a cell or node). Cells are allocated from a (large) area of storage called a heap, which is simply an unordered collection of cells. As well as allocating new

cells, the reduction process discards cells, or rather it discards pointers to cells. A cell in the graph may have many pointers to it. A cell is declared unusable when all the pointers to it are deleted. Since storage is finite, it is necessary to reclaim these garbage cells. In the example shown in Fig. 1.6, on evaluation of the addition *(x + 3)*, two cells *x* and *3* loose their links with other nodes in the graph, and become garbage. Similarly, the reduction *(y - 2)* results in two garbage cells. Overall in the reduction of function *f*, six garbage cells are generated. The implementation of any functional language includes a garbage collector whose purpose is to identify and recycle garbage cells. The whole activity of cell allocation and garbage collection is called storage management.

Storage management schemes including garbage collectors are needed in all declarative languages (*functional* and *logic*) that have dynamic heap semantics. The performance of implementations of these languages depends to a great extent on the efficiency of the storage management scheme. Also the implementation of a storage management scheme depends very much on the underlying system architecture.

1.5 Garbage Collection in Uniprocessor Systems

This section outlines some of the garbage collection methods available in the existing uniprocessor systems. There are two well known techniques namely *mark and sweep* method and *reference counting* [16]. In the mark and sweep technique, the system accumulates garbage cells until the heap storage is empty. Garbage collection is then initiated. In the first (marking) phase all reachable cells in the program graph are marked. Garbage cells are not marked. During the sweep phase, a linear search of the entire memory is done, detecting cells that are unmarked. A free list is built of these free cells. Generally the sweep phase is followed by a *compaction phase* wherein all the free cells are moved to one end of the address space. This scheme has the advantage that it can

reclaim detached cyclic structures in the program graph. But the main disadvantage of this scheme is that this algorithm is not real-time in nature as evaluation of the program halts during garbage collection.

In the reference counting technique each cell has a count (reference count) of the number of pointers to it. All accessible cells in the program graph have a reference count greater than zero. A cell is declared garbage when all pointers to it are deleted, i.e. when its reference count decreases to zero. During each reduction operation reference counts of cells involved in the operation are updated and any cell whose reference count decreases to zero is reclaimed. This scheme has the advantage that it is real-time in nature as program evaluation is not halted and garbage collection proceeds concurrently with the program evaluation. But this scheme has the disadvantage that it can not reclaim cyclic structures and much time is spent updating reference counts during each graph operation.

To overcome the disadvantages of both these techniques, hybrid techniques have been proposed [11, 16, 18]. These methods attempt to achieve real-time response as well as reclaim cyclic structures. Algorithms for systems with virtual memory are known as *copying algorithms* [8, 28, 30], here the virtual address space of the processor memory is logically partitioned into semispaces. Only one of the semispaces is active at a time, cells are created in the active semispace. When its space is exhausted, all active cells in the current semi space are copied to other semispace. The garbage cells in the current active semispace are automatically reclaimed.

1.6 Parallel Graph Reduction

One of the most attractive features of functional programming languages is that they are not inherently sequential, as are conventional imperative languages. At any moment there are a number of *reducible expressions* (or *redexes*) in the program graph and in principle they could all be reduced simultaneously. Writing parallel programs is easier in a

functional language. The following points can be stated to substantiate the above statement.

i) In a functional language the parallelism can be dynamic; there is no static division of the problem into tasks. Maximal parallelism can be dynamically exploited.

ii) In a functional program the synchronization between different reductions is mediated entirely by the shared graph. A reduction is made known to the graph by the indivisible operation of overwriting the root of the redex with the result of the reduction, and no other synchronization is necessary. Procedural parallelism, however requires a large run-time investment in synchronization.

iii) There are no extra language constructs required to write parallel functional programs.

The above mentioned advantages apply to any parallel implementation of a functional language, but graph reduction is particularly attractive. A number of evaluator tasks simultaneously work on the graph. Each evaluator task reduces some particular sub-graph. Each sub-graph is located in physically separated memories. During its execution, a task may anticipate that it will require the value of a certain sub-graph at some future time. In this case it may generate a new task to evaluate the sub-graph in parallel by sparking the root node of the sub-graph. The new (child) task will evaluate the graph rooted at the sparked node, concurrently with the continued execution of the (parent) task that sparked it.

1.7 Garbage Collection in Multicomputer Systems

The model of a parallel graph reduction machine is shown in Fig.1.8. The program graph is logically partitioned into sub-graphs and each sub-graph is located in the private memory of a processor. In the Fig. 1.8, we have partitioned a graph into four sub-graphs. Each sub-graph resides in the local memory of a processor. Each sub-graph has a root of

its own. *Local links* link nodes in the same sub-graph, *remote links* span across memory boundaries. With good locality in the graph partitioning and memory allocation, the number of remote links in the system are assumed to constitute a small fraction of the local links. Each processor has access to local links and nodes of its sub-graph. Remote nodes are accessed by sending messages through the inter-connection network. Cycles appear in the sub-graphs (because of recursive functions and data structures) that may be local or global. Local cycles reside in a single memory. Due to the partitioning of program graph among many physically separated memories, cycles in the graph may happen to span across different memories. Detecting these detached global cyclic structures is difficult and expensive in terms of communication and synchronization overhead.

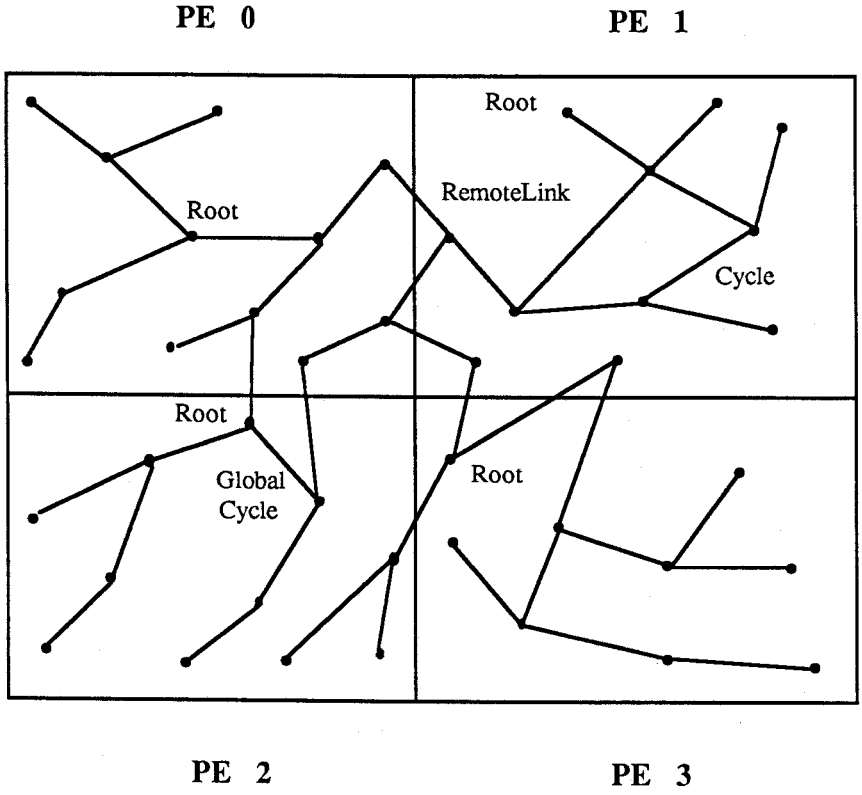


Figure 1.8 Model of a Parallel Graph Reduction Machine

Implementation of storage management schemes in a multicomputer system is

complex because the garbage collection scheme has to manage objects in physically separated stores. This introduces consistency and synchronization problems with shared data. Message passing introduces communication overhead in the system. A simple extension of algorithms proposed for uniprocessor systems to the multicomputer systems is not practical due to the presence of these overheads. Many algorithms have been proposed [3, 4, 26, 27, 29]. These algorithms range from completely global schemes to parallel real-time algorithms. Not only are these algorithms complex, their analysis and proof of correctness is very hard. A detailed description and a comparison of these algorithms is given in Chapter Four.

1.8 Goals of the Thesis

The main thrust of this thesis is to develop a new storage management scheme for parallel multicomputer systems. The specific goals are:

i) To study storage management schemes in multicomputer and uniprocessor systems.

ii) To propose a new algorithm for garbage collection in a multicomputer system.

iii) To perform a simulation study of our proposed algorithm, and thereby gain insight into the dynamic behavior of our algorithm.

iv) To establish arguments for the correctness of the proposed algorithm using *Petri Nets*.

This thesis presents a model for the development of any distributed algorithm. Analytic techniques are used to obtain approximate performance results. Simulator models with empirical loads provide insights into the behavior of the algorithm. A semi-formal proof of correctness based on modelling techniques gives confidence in the overall algorithm design.

1.9 Outline of the Thesis

Chapter Two deals with the garbage collection algorithms used in uniprocessor systems. Mark and sweep algorithm and reference counting algorithms are studied. Hybrid techniques that attempt to combine the advantages of both these schemes and copying algorithms for systems with virtual memory are also presented.

The Chapter Three deals with the garbage collection algorithms that have been proposed for multicomputer systems. Specific algorithms are presented and a comparison of these techniques is given.

We present our proposed algorithm in the fourth chapter. Some of the advantages of our algorithm over existing techniques are highlighted by analytical modelling. In Chapter Five we report a simulation of our algorithm. Motivation for the simulator model is discussed. Experiments conducted to verify the algorithm are presented.

In Chapter Six we discuss formal modelling of our algorithm using Petri Nets. Analysis of Petri Nets using *Reachability trees* and *Linear Algebraic methods* are presented. The motivation for using a linear algebraic method and verification of system properties using P and T invariants of the network is given. The thesis concludes with a chapter highlighting significance of the work and summary of results and conclusions drawn from the research conducted.

CHAPTER 2

Garbage Collection in Uniprocessor Systems

2.1 Introduction

This chapter surveys techniques for garbage collection in uniprocessor systems. A survey article by Cohen [16] gives an excellent overview of various garbage collection (GC) algorithms that have been proposed for uniprocessor systems. There are two traditional approaches for GC in uniprocessor systems, *mark and sweep* and *reference counting*.

Mark and sweep collection is done whenever the free list is empty. First, all accessible cells are marked by traversing the entire accessible structure. Then a linear scan through memory recovers all unmarked cells. This algorithm is discussed in section 2.2.

In reference-counting each cell has an extra field, called reference-count that holds the number of references (i.e. the number of pointers) to it [15]. The reference-count is incremented whenever a link is made to the cell and is decremented whenever a pointer is discarded. A cell whose reference-count drops to zero is reclaimed. These collectors can not reclaim cyclic structures. Modified reference-counting algorithms have been proposed that can reclaim cyclic structures [11, 12, 15, 28]. *Hybrid* systems, which have a limited-width reference-count field have been suggested. These collectors use mark and sweep collection to reclaim cyclic structures. A discussion of these algorithms is presented in section 2.3.

The main requirement of garbage collectors in virtual memory systems is the locality of references. Algorithms that improve locality and perform compaction are necessary for such systems. These are known as *copying* garbage collectors. These

algorithms are presented in section 2.4. The approaches presented so far are not suited for systems with real-time constraints. Parallel algorithms have been proposed for these systems. In these schemes, two processors work in parallel, one processor (mutator) is responsible for graph mutation and the other (collector) is for garbage collection [19, 50]. Section 2.5 describes some of these algorithms.

2.2 Mark and Sweep Algorithm

In this approach the system consumes cells until the free list is empty. At this stage execution of the user's program is stopped and a garbage collection cycle is initiated. The mark and sweep garbage collection takes place in three phases, the first is the *marking phase*, its task is to identify the objects accessible from the root by traversing the graph, and marking each object encountered. As all active cells in the graph are accessible from the root, all these cells are marked. Inactive (garbage) cells are unmarked. The marking phase is followed by a sweep phase. Its task is to sweep the entire memory space and to examine each existing object: if the object is not marked, it is reclaimed by returning its space to free storage. A free list is built of these reclaimed cells. The sweep phase is followed by a *compaction phase* wherein all the active cells are pushed into contiguous locations of memory. The other end contains a contiguous area of unused space that can be used for creating new objects.

From the above discussion, two points are obvious: the garbage collector is able to detect all inactive cells including cyclic structures, and collection halts program evaluation. The duration of garbage collection varies and it can not be predicted. Unpredictable program interruptions makes this algorithm unsuitable for real-time applications. The duration of the sweep phase is proportional to the size of the memory. Hence this scheme is not appropriate for systems with very large virtual address space. This has prompted the development of copying algorithms for virtual memory systems.

The marking phase requires an explicit stack for storing pointers to the cells being marked. A pointer is pushed onto the stack just before marking the cells's right branch. Marking terminates when the stack is empty. Consequently each node in the graph is visited twice: once before marking the left field and once before marking the right field. A requirement of this technique is the need to have a stack space to hold at least n pointers, where n is the maximum possible depth of the graph (in the worst case this will be equal to the number of cells in the graph). To reserve this much additional space initially is uneconomical. Several algorithms have been proposed to circumvent this difficulty [18, 49]. All these algorithms involve reducing the required storage by trading it for the time to perform marking.

The algorithm proposed by Deutsch [18] and by Schorr and Waite [49] dispenses with the use of a stack but requires one additional bit per cell. The main idea of this algorithm is that the nodes of a tree or a directed graph can be inspected without using a stack by reversing successive links until leaves or already visited nodes are found. The link reversal can then be undone to restore the original structure of the graph. The additional bit per cell (called a *tag* bit) indicates the direction in which the restoration of reversed links should proceed (i.e. whether to follow the left or the right pointer). The disadvantage of this marking scheme is that the cells are visited three times. This additional visit and the overhead for restoring pointers and for checking and setting bits makes this algorithm less time efficient than the classical algorithm. The authors Schorr and Waite have suggested a modification in which there is a fixed-size stack with link reversal technique. The stack algorithm is used whenever possible. If the stack overflows, the tracing and marking proceed by the link reversal method. Many variations of this marking scheme using fixed-size stack and/or modifications of link reversal techniques have been proposed [16].

2.3 Reference Counting

In reference counting, each cell has a field having a count that indicates the number of pointers to it. All active have a reference count greater than or equal to one. Any unreferenced cell has a reference count equal to zero. The garbage collector becomes operational whenever a link is made or broken. Garbage collection in this case is incremental, the GC time is distributed over the entire program period. But the main drawback of this scheme is that it can not reclaim cyclic structures.

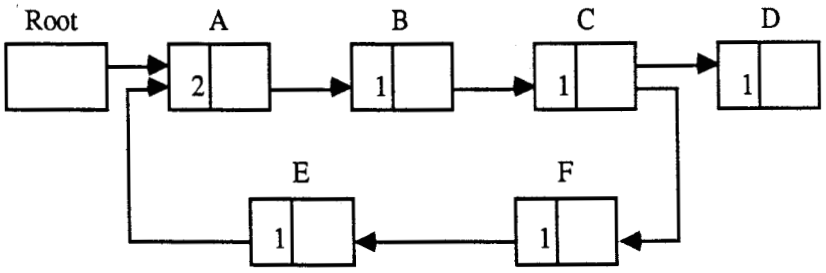


Figure 2.1 Reference Counting can not detect Cyclic Structures

The Fig. 2.1 illustrates the weakness of reference counting algorithm in detecting cyclic structures. In the figure is shown a section of a cyclic graph, cells *A, B, C, E* and *F* form a cyclic structure. All cells are accessible from the root only by traversing through the link *(root, A)*. The reference count of *A* is equal to two and that of other cells is equal to one. Let us assume that the link $(root, A)$ is deleted. The reference count of *A* decreases by one. The reference counts of all other nodes remain unaffected. This deletion of link $(root, A)$ delinks the entire section of graph from the root, the entire structure is garbage now. But as the reference count of none of the objects in the structure is equal to 0, the algorithm fails to detect them as garbage. As recursion is a natural way to program in a functional language, cyclic structures are generated frequently in a functional program. This algorithm can not reclaim space used by these cyclic structures, resulting in continual loss of space. Modified reference counting algorithms have been proposed by

Hughes [28], Brownbridge [12] and Moon [36] that can reclaim cyclic structures. Section 2.3.2 discusses these algorithms in greater detail.

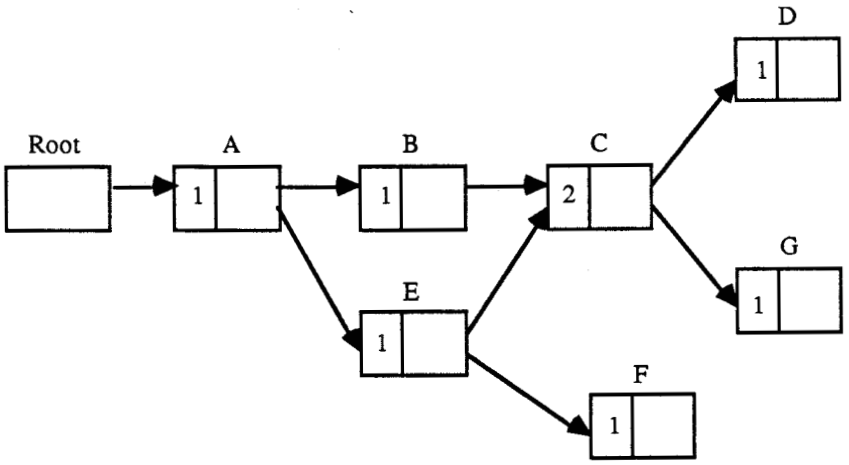


Figure 2.2 Reference Counting slows down graph mutations

The other drawback of reference-counting is that it introduces extra overhead on every object operation slowing down graph mutations. This is illustrated in Fig. 2.2. The figure shows a section of graph with objects *A, B, C, D, E, F* and *G*. The reference counts of *A, B, D, E, F* and *G* are equal to one and that of *C* is equal to two. There is a pointer from cell *A* to *B*, cell *B* can be accessed from the root only by traversing through this pointer. Let us say this link is broken, the reference count of *B* decreases to zero. It is no more accessible from the root, it has become a garbage cell. Any outgoing pointers from a garbage cell must be deleted. This in turn may generate some more garbage cells. The process of pointer deletion is to be done recursively from garbage cells generated at every step. In the example shown above, the deletion of link *AB* results in the deletion of pointer *BC*. The reference count of *C* decreases to one, i.e. a single pointer operation resulted in the breaking of one additional link and updating of one reference count. If instead of one pointer from *B* there were many pointers, many reference count updates needs to be done. If these cells scatter across different parts of the memory, accessing these storage cells becomes very expensive. In a virtual memory

system this results in increased page faults slowing down graph operations. Hence an algorithm for garbage collection in virtual memory system must ensure good locality in the program graph.

Reference-counting introduces space overhead per each cell. Theoretically the width of the reference-count field must be equal to the maximum number of cells in the address space (i.e. if every other cell in the system points to one single cell). But observations in practical Lisp systems have shown that 97% of cells have a reference count equal to one [16]. Modifications have been suggested to the reference counting algorithm to circumvent this space overhead. The idea is to have a limited width reference-count field. Reference counts are incremented until the maximum value is reached. Those reference-counts that have reached their maximum value are not updated any more; these cells are assumed to be permanently active. When the heap becomes empty, mark and sweep collection is initiated to reclaim these permanent cells. The logical extreme of the limited width reference-counting idea is a *one-bit reference-count* field. This idea was proposed by Wise and Friedman [16]. Any cell that has this bit set indicates that this cell is referenced once or more. As in the limited width reference-count algorithm, mark and sweep collection is resorted to reclaim circular structures and cells that are referenced more than once.

2.3.1 Hardware Support for Reference Counting

As seen in Fig. 2.2, reference-counting slows down graph mutation. Much of the time overhead of reference-counting would be alleviated if hardware support were available. Wise [56] proposes a 'smart' memory module that is capable of handling the increment/decrement operations on reference counts. The memory module translates read/write instructions into increment/decrement operations on reference counts. Increment/decrement reference-count messages are transmitted across memory modules.

In software reference-counting, the processor is responsible for reclaiming and updating the available heap space when a reference-count becomes zero. In this scheme, the memory module is responsible for maintaining this list. A cell is automatically included into the heap without interrupting processor operation. This hardware reference-counting scheme is also suitable for tightly-coupled multiprocessor systems wherein many processors share a common memory distributed across many modules.

2.3.2 Cyclic Reference Counting Algorithms

Bobrow [11], Hughes [28] and Brownbridge [12] have suggested extensions to the conventional reference counting algorithm that would allow it to reclaim circular structures. A brief description of the Brownbridge's algorithm is given below:

In this algorithm, graph pointers are divided into two types, *weak* and *strong*, with their own separate reference counts. Each cell has two reference count fields. Strong reference counters are used whenever normal reference count pointers would be used. Weak reference counts are used when a pointer forms a cycle. Consequently the computation graph satisfies the following two rules:

i) Cells in use are accessible from the root through a continuous chain of strong pointers.

ii) No cycles are formed by a continuous chain of strong pointers.

The process of creating and deleting a pointer is the same as in standard reference counting algorithm. When a pointer is created with the appropriate type, the relevant reference count of the cell is incremented. If the pointer created is of *strong* type, SRC(ptr_to) is incremented and if it is a *weak* pointer, WRC(ptr_to) is incremented. Here, SRC(ptr_to) represents the strong reference count and WRC(ptr_to) the weak reference count of the pointer object *ptr_to*. Deletion of a pointer is more complex than

creation, here four cases may arise:

i) If the pointer is weak then,
WRC(ptr_to) = WRC(ptr_to) - 1

ii) If the pointer is strong and SRC(ptr_to) > 1, then
SRC(ptr_to) = SRC(ptr_to) - 1

iii) If the pointer is strong, SRC(ptr_to) = 1 and WRC(ptr_to) = 0, then delete *ptr_to*. This cell can be garbage collected.

iv) If the pointer is strong, SRC(ptr_to) = 1 and WRC(ptr_to) > 0 then this is the case standard reference counting algorithm will fail and special action needs to be taken.

In this case, it can not determined whether the cell is free or not. Brownbridge [12] in his paper suggests a method to handle this situation. The SRC(ptr_to) is set to 0. All weak pointers to *ptr_to* are inverted to strong pointers. To determine whether *ptr_to* is free, a recursive search is initiated by visiting sub-objects of *ptr_to* and undo any strong cycles created by pointer inversion. He proposes a routine *suicide* to do this recursive searching. If SRC(ptr_to) = 0, then recursively delete all objects reachable from *ptr_to*. He suggests an implementation method to handle this pointer inversion. Associated with each pointer is a status bit indicating what type of the pointer it is. The cell body also has an associated bit, and it is the relationship between the pointer status bit and its target cell body bit that determines the strength of the pointer, eg. if both bits are the same then the pointer is strong otherwise it is weak.

In his thesis Salkild [48] has analyzed this algorithm. He observes that this *suicide* routine performs well in a graph with a high proportion of strong pointers, but with larger programs the algorithm fails. He analyzes the weakness of the *suicide* routine, saying that this routine changes all weak pointers to strong ones at any one site and it fails when more than one weak pointer needs to be turned into a strong one. The *suicide*

cell, having no strong pointers to it is now incorrectly collected. He also observes that the efficiency of the algorithm is critically dependent on the distribution of weak pointers. This strong dependency on graph structure makes this algorithm highly unpredictable. Moreover the algorithm expects the language implementation to know when a cycle is being created which is not always possible.

2.4 Garbage Collection in Virtual Memory Systems

A heap can be implemented in real-memory or virtual memory. In the case of real memory, the whole heap resides in the primary memory and all objects in the heap are directly accessible to a user program and the garbage collector. In the case of virtual memory, the heap resides in more than one level of memory hierarchy. For an efficient GC that operates in virtual memory, it is necessary to minimize page faults. Grouping related objects in one page (or a few pages) and compaction (to improve locality of references) are important properties of a GC that operates in such an environment.

Many algorithms have been proposed for GC in these virtual memory systems. These are called *copying* algorithms [8, 30]. A copying algorithm performs all the three tasks of a mark and sweep GC in one phase. In Baker's algorithm, the memory space is divided into two areas called *semispaces*. At a given time, only one semispace is used for creating new objects. When garbage collection begins, all accessible objects are copied from the current semispace into the other semispace and the role of semispace is reversed. The new semispace contains only accessible objects.

GC efficiency (by efficiency we mean the rate of collecting garbage cells) is enhanced by taking hardware assistance in the form of tagged memory architecture. In *tagged memory*, every word is divided into two parts: the data and the *tag*. The tag distinguishes words whose data part is an address from words whose data part is a number

or a bit pattern. Special conventions and machine instructions are provided for the efficient processing of these tagged words. As a copying garbage collector needs to distinguish addresses from numbers, the tagged architecture enables the garbage collector to scan memory without regard for object boundaries, hence it can scan non-sequentially through memory.

In a typical functional programming environment, object life time is not uniform. A great percentage of the objects have been observed to have a very short life time [16, 30]. A GC that treats all objects equally does not perform very well. By concentrating garbage collection effort in the most productive places, the maximum amount of space can be reclaimed for the minimum cost of computation time, virtual memory paging and impaired interactive response. Objects may be classified into groups based on their life time. Most of the space can be reclaimed by concentrating GC effort on ephemeral objects. The older objects may be assumed to be permanent, they need to be garbage collected less frequently.

The algorithm proposed by Lieberman and Hewitt [30] takes into account the life time of objects. Their algorithm is an extension to Baker's algorithm. In Baker's algorithm, the address space is logically divided into two semispaces, whereas in this algorithm, the address space is allocated in small regions. Objects recently created contain a larger percentage of garbage and will be garbage collected more frequently than older regions. The process of garbage collecting a region is initiated by *condemning* it. This is followed by *scavenging*, wherein all accessible objects in the region are evacuated. Many scavenging processes may be working in parallel on different regions. The rate of condemning regions is related to the age of regions. Older regions are condemned less frequently than recent ones. Inter-region pointers are maintained using *entry* and *exit* tables. To reduce space overhead of these tables and to cover-up the cost of scanning large numbers of regions, the older regions are merged and inter region pointers are deleted.

2.5 Real-Time Algorithms

The mark/sweep GC algorithm introduces substantial program interruptions, ranging from any where between a few seconds to tens of minutes. These unpredictable program interruptions makes this algorithm unsuited for real-time applications. In reference counting, collector is active in parallel with the mutator, but each graph operation may be slowed down considerably.

Several real-time garbage collection algorithms that work in parallel with a user program have been proposed [19, 50, 17]. These are based on two processors working in parallel: one is responsible for GC, called *collector*, and the other for program execution, called *mutator*. Two algorithms based on this approach have been proposed by Steele [50] and Dijkstra et al., [19]. In Steele's method, the collector has three phases: *mark, sweep and relocate*. During the mark phase, all accessible cells are marked. It uses a stack to hold objects that have been marked but whose children have not been examined. During the sweep phase, the storage space of inaccessible objects is picked up. The relocation phase relocates accessible objects to minimize the storage space required. Since the mutator is running in parallel with the collector, the free list must have enough free storage space to keep the mutator from starvation. Semaphores control mutual exclusion of shared object between the collector and mutator processors.

In any parallel garbage collection method, the mutator must co-operate with the collector for performing the proper marking of accessible objects. Since the collector and mutator are running in parallel, the operation of these algorithms is much more difficult to understand or prove correct than any sequential GC. The algorithms proposed by Baker [8] and Lieberman and Hewitt [30], can also be considered as real-time GC algorithms. Here semispaces are simultaneously active. These approaches are less complex than the algorithms suggested by Dijkstra et.al., [19] and Steele [50], since normal activity and GC

represent only one sequential process.

2.6 Summary

An overview of uniprocessor garbage collection algorithms has been presented. A mark and sweep collector is able to detect all garbage cells, it has the disadvantage that it introduces arbitrarily long program interruptions. A reference-counting collector works in parallel with user's program. It is not able to detect cyclic structures in the program graph and it slows down graph operations.

Copying collectors are used for garbage collection in virtual memory systems. Their main requirement is to improve locality in the program. Real-time algorithms use two processors, a *mutator* for graph reductions and a *collector* for garbage collection. Both mutator and collector work in parallel.

CHAPTER 3

Garbage Collection in Multicomputer Systems

3.1 Introduction

This chapter presents an overview of the important garbage collection algorithms proposed for loosely-coupled multicomputer systems. These algorithms are adaptations to the distributed environment of ideas that have been developed for uniprocessor systems. In a loosely-coupled multicomputer system, the heap is distributed over many physically separated PE (Processor Element) memories. This organization is called a *distributed heap*. Each PE has direct access only to objects that reside in its local memory. Due to parallel activities that take place in a distributed heap, its management is much more difficult than a single processor implementation.

Parallel graph reduction places unique demands on the performance of a garbage collector. A distributed garbage collection scheme must address these issues – inter-processor communications and utilization of communication network, synchronization and mutual exclusion of shared data, wastage of CPU cycles, and real-time response.

With present day technology, the cost of communications between parallel processors is much higher than the cost of communications within a processor. Inter-connection network bandwidth is a critical performance factor of a distributed system. Processors need to communicate with each other during garbage collection. Bandwidth of the communication network is limited, a garbage collector in a distributed environment must make optimal use of the network bandwidth.

Protecting shared data in a parallel multicomputer system is quite complex. It is essential to maintain *mutual exclusion* (only one process has write access to the shared

data [25]) and *synchronization* (allowing processes to exchange information and communicate with each other [25]) of shared data. Efficient schemes are needed to prevent *race conditions* (in which the outcome of a computation depends on the speeds of processes, i.e. parts of computation are *time critical* [25]), and *deadlocks* (in which process(s) wait for event(s) that can never occur [25]) in the system. The mechanisms to protect shared data should have minimal effect on the parallelism in application programs.

A distributed garbage collection algorithm must not waste much computational power. Simple distributed GC algorithms (*global* algorithms) restrict parallelism in the system and thereby waste much of the computational power. These algorithms have a central synchronization and control point. They require less space, less communication bandwidth and synchronization overhead. As computation in the system comes to a halt during GC, maintaining data consistency and synchronization is straightforward. On the other hand, non-global GC algorithms allow greater parallelism and thereby do not waste much computational power. These schemes typically have high space requirement, and high communication and synchronization overheads.

The design of a distributed garbage collection scheme must take all these factors into consideration. The designer has to make tradeoffs between these conflicting requirements. In addition to all these factors, the performance of a distributed garbage collector depends on the underlying system architecture and the application.

We review a variety of previous solutions for parallel distributed garbage collection:

- *Distributed reference-counting* has been suggested [40] for garbage collection in message-passing systems. This algorithm is presented in Section 3.2.1.

- Hudak [26, 27] has presented a *marking tree* garbage collection algorithm that is an adaptation of the real-time mark and sweep algorithm proposed by Steele [50] and Dijkstra et. al., [19]. Section 3.3

presents this algorithm.

- Ali [3, 4] has proposed four distributed garbage collection algorithms. These algorithms can be categorized as *global, local-global, distributed-local* and *distributed real-time* algorithms. These algorithms are presented in Section 3.4.

- Hughes [29] presents a distributed garbage collection algorithm that attempts to overcome deficiencies of Hudak and Ali's algorithms. This algorithm is presented in Section 3.5.

3.2 Reference Counting

Nori [40] adapted the reference-counting approach to the distributed environment. In this scheme, all of the reference-counting operations are performed by spawning remote tasks on the appropriate processor. An *increment-reference-count* task is generated when a new object reference is generated. Similarly a *decrement-reference-count* task is generated and sent to the appropriate PE when an object reference is destroyed. A non-trivial problem with Nori's approach is to guarantee that reference counting operations (increment/decrement of reference counts) are executed in the same order that they were generated, otherwise a reference count may reach zero prematurely, as illustrated in Fig. 3.1(a, b).

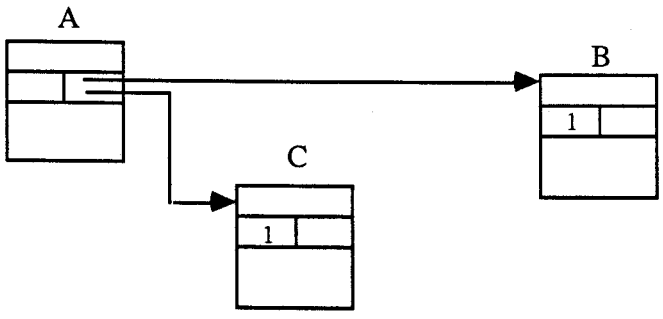


Figure 3.1 Reference counting is order dependent.

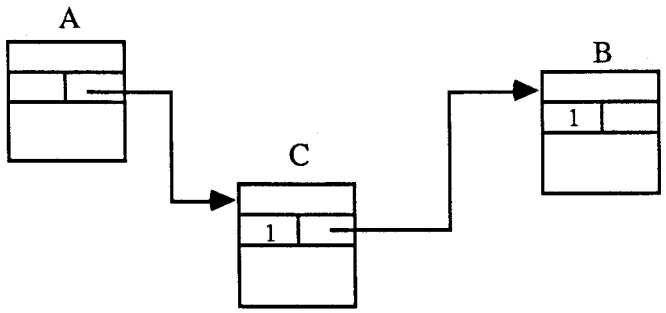


Figure 3.2 After reference count operations.

In Fig. 3.1 is shown a graph with objects *A, B* and *C*. There is a reference from object *A* to *B* and from *A* to *C*. In a distributed system these objects could be scattered across different memories. Let us say two tasks *increment-reference-count CB* and *decrement-reference-count AB* are spawned in that order. The resulting graph is dependent on the order in which these tasks are executed. If these tasks are executed in the order they are spawned, we get the graph shown in Fig. 3.2. If these tasks are executed in the reverse order (task *decrement-reference-count AB* executed before *increment-reference-count CB), B* is reclaimed before the arrival of *increment-reference-count CB* task. The increment can not be executed, since *B* is non-existent. Hence the system must ensure that reference tasks are executed in the order they are spawned. A direct solution of this difficulty appears to require detection of distributed termination for every decrement reference-count; this is clearly infeasible. Nori [40] has not attempted to solve this problem. As discussed in Chapter 1, the reference counting approach has the additional disadvantage of being unable to detect and subsequently reclaim cyclic structures. The frequency of reference count updates requires many messages.

3.2.1 Reference Weighting

The main problem with Nori's scheme is that it is difficult to maintain reference-

counting operations in the same order that they are generated. To overcome this problem, a *reference weight* scheme has been suggested [55]. The idea is as follows: when an object *A* creates a new object *B*, a weight is associated with *B* and this value becomes a part of the created object reference to *B* as shown in Fig. 3.3.

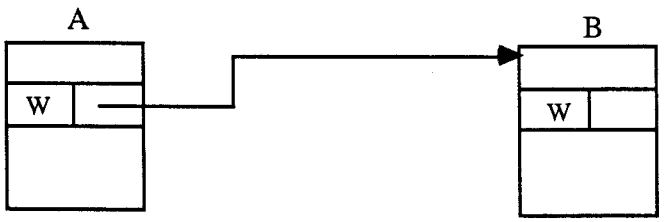


Figure 3.3 An object reference using reference weights.

This object reference may be represented by a tuple $(W, address_B)$. When such an object reference is copied into object *C*, the weight *W* is split into two parts and the two object references will be represented by $(W_1, address_B)$ and $(W_2, address_B)$ such that $W = W_1 + W_2$ as shown in Fig. 3.4.

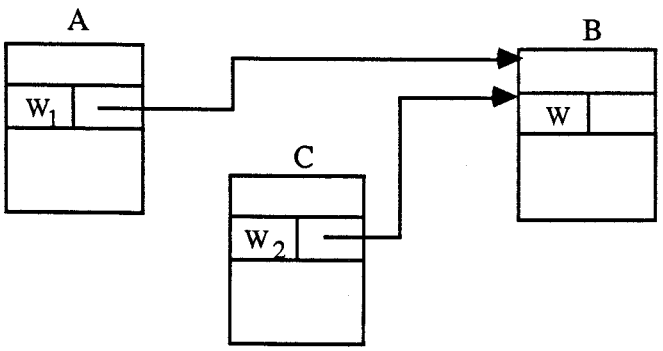


Figure 3.4 Copying an object reference in reference weighting.

Making a copy of an object reference does not require communicating with the PE that has the object in its local memory. On the other hand, destroying an object reference requires generating a *reduce-reference-weight* task and sending it to the PE that has the

object to decrease the current object's weight by the weight associated with that object reference. An object will be identified as garbage only when its weight reaches zero. The advantage of this scheme is that it is not necessary to process reference weight tasks in the same order that they were generated. This scheme has less communication overhead than reference-counting as making a copy of an object reference does not require communicating with the PE that has the object. Intuitively this improves the communication performance by 50%. The disadvantage is that it requires more space in each object reference for saving the weight part. In addition when the weight of an object reference reaches one and that reference needs to be copied, the object's weight has to be increased before making a new copy of that object reference. An *add-reference-weight* task has to be generated and sent to that object to increase its weight by some value. Associating high weight with each newly created object minimizes such *add-reference-weight* tasks but requires more space for the weight field in each object and each object reference.

3.3 Marking Tree Collector

Hudak and Keller have proposed algorithms suited to highly parallel distributed multicomputer systems [26, 27]. Their algorithms are adaptations of the real-time GC algorithm proposed by Steele [50] and Dijkstra et. al., [19]. First, all processors co-operate to mark all accessible cells, then all processors collect unmarked cells. The distributed graph marking represents the major effort in the GC process. A brief description of this algorithm is presented below.

There are two issues involved in parallel graph reductions, namely, the connectivity of the computation graph and the propagation of work. A *task* is the smallest unit of work, and a *process* is a collection of tasks that accomplish some particular goal. Tasks propagate between adjacent vertices in the graph and may cross partition boundaries. A task t may be represented by a tuple $<s,d>$ where s is the source vertex and d is the

destination vertex. In this respect a task may be viewed as a message from one vertex of the graph to another. A key aspect in *lazy evaluation* graph reduction is the inclusion of mechanisms to *eagerly* invoke computations whose result may not be needed. This strategy is intended for increased parallelism. *Vital* tasks are those computations whose outcome is known to be needed. *Eager* tasks compete with *vital* ones for system resources. *Vital* tasks may be given higher priority than *eager* tasks. During the course of time, it may be observed that the outcome of an *eager* task is needed, in this case the priority of *eager task* is raised to that of a *vital* task. Similarly, the system may discover that the outcome of an *eager* task is not needed – this task and the tasks it spawned are now called *irrelevant* tasks. These tasks may be distributed through the system generating an arbitrarily large load in the system. It is necessary for the graph reduction system to identify and eliminate such tasks.

For example, consider the evaluation of the expression *if (B, T, F)*. The boolean expression *B* needs to be evaluated first, if the outcome is true, *T* is to be evaluated else *F* is to be evaluated. In lazy evaluation, the expressions *T* and *F* are evaluated in parallel with *B*. *B* is vital, *T* and *F* are eager. If B evaluates to true then *T* becomes a vital task and *F* becomes an irrelevant task.

The other important issue in parallel graph reduction is the possibility of deadlock, which in graph reduction corresponds to an expression whose ultimate value is undefined (generally denoted $\perp$) and is manifested by a subgraph whose task activity has ceased, yet the subgraph's value is being awaited by some other vertex. The language implementation needs to discover deadlocked regions of the graph. A deadlocked vertex *v* can be modelled as a vertex reachable from the root but not from any task, because this implies that the root depends on *v*'s value yet no task can ever propagate there to compute it.

Hudak and Keller's algorithm works by dynamically building a spanning tree on

the computation graph called the *marking tree*. Marking is initiated by spawning a *mark task* at the root, which propagates additional *mark tasks* to the children of the root. These mark tasks propagate in the *forward* direction until an already marked vertex or a leaf node is encountered. A *return task* is then spawned that works its way *backward* through the graph. When all the mark tasks spawned from a vertex v have returned in this manner, a return task is recursively spawned on v's parent in the marking tree. To implement this a count is maintained in each vertex v of the number of mark tasks that have been spawned from it but have not returned, in addition to a pointer to the vertex's parent in the marking tree. Marking is complete when a *return task* reaches the root.

The main strength of this algorithm is that it is highly parallel and decentralized. The *marking tree* is embedded in the computation graph instead of a centralized stack. The other advantage of this algorithm is that it can detect *deadlocked* vertices and *irrelevant* tasks. Its main weakness is its space overhead. Each vertex needs to keep a count of the number of marking tasks spawned from it to the children vertices and a pointer to its parent vertex. The maximum number of marking tasks that could exist at any one instant of time should be determined and their required space should be reserved for the collector to avoid running out of space. In the worst case the number of marking tasks is equal to the number of arcs in the graph due to parallel *breadth-first* strategy of tracing the graph. In addition, memory contention due to the two processors (mutator and collector) working in parallel needs to be addressed. This scheme needs two messages per remote edge per garbage collection, hence the communication overhead of this scheme is quite high.

3.4 Ali's Algorithms for GC in Multicomputer Systems

In his dissertation, Ali [3] presents an *object oriented storage system* or OSS. He discusses the implementation of garbage collection schemes in a parallel distributed

multicomputer system. The primary functions of OSS is to provide the following facilities to user programs: *create* an object, *access* an object, *update* an object field and *garbage collecting* of the space of unused objects. The implementation of the OSS in a distributed processing environment should allow high parallelism in order to promote the efficient execution of parallel programs. The set of operations that provides the functions of the OSS defines the user interface to the system. The user does not need to know how these operations are implemented – the details of all network operations are invisible to the user.

Ali proposes a number of distributed garbage collection algorithms. These algorithms are adaptations of the mark/sweep algorithm to distributed environments. These algorithms can be categorized into four classes: *global, local-global, distributed-local* and *distributed real-time*. A brief overview of each of these algorithms is given below.

3.4.1 Global algorithm

The main feature of Ali's global algorithm [3, 4] is that the execution of the user program halts over the whole system during the GC process. The marking phase spans all accessible objects in the system. Two global schemes have been proposed. In the first scheme, a *master* PE is responsible for starting GC and controlling the synchronization of GC phases. Any other PE that wishes to invoke GC sends a request to the *master* to start a GC. The master starts a global GC by sending a request to each PE to suspend its computation and to start marking its own roots and all objects reachable from them. The master waits until each PE in the system completes marking of all objects reachable from its roots. Then, the master requests each PE to perform a local sweeping and memory compaction before resuming normal computation.

In the second global scheme, any PE can become the master. The PE that first runs out of space initiates global GC and becomes master. On initiating global GC normal

computation is halted and all PE's co-operate in marking accessible objects. This algorithm is faster and more optimal than the first one.

The main characteristics of Ali's global schemes are: The whole system comes to a halt during GC, and these schemes can reclaim all garbage including global cyclic structures. They are suitable for loosely coupled multicomputer systems with few processors without real-time constraint.

3.4.2 Local-Global Algorithm

In this scheme, a combination of *local* and *global* GC techniques are used [3, 4]. Each PE that runs out of space does a local GC, if the space reclaimed is sufficient it continues with the normal computation, otherwise it will invoke global GC. To allow a PE to perform local garbage collection, it has to know locally which of its local objects may be reachable from remote objects that reside in other PE's. When a reference to a local object leaves the boundary of its PE's store, it is assumed accessible in each local garbage collection invocation until the next global garbage collection invocation. Assuming a high locality of data, the global GC's are rarely invoked. As local GC's are simple and do not require much computation time, this scheme is much more efficient than the global techniques. But in cases where locality is poor, the number of global GC's may be significant and in such a situation, the performance of this scheme will be as poor as the completely global scheme.

3.4.3 Distributed-Local Algorithm

In a distributed system with many PE's the rate of utilization of local space is not uniform. Forcing all PE's to co-operate in the global GC is inefficient, as even PE's that may not have any garbage will have to halt. In Ali's distributed-local algorithm, there is

no global garbage collection as in the previous schemes. Each PE independently and asynchronously performs a local garbage collection. A PE co-operates with the other PE's in the system only at the end of its local GC by informing them about remote objects still referred by it. Each PE keeps tables of the references to local objects referred from remote objects to allow any PE to independently perform its local GC. This scheme is suitable for loosely coupled multicomputer system that has many PE's, provides a high locality of reference and generates cyclic structures locally.

3.4.4 Distributed Real-Time Algorithm

This algorithm is an adaptation of Baker's real-time garbage collection algorithm [8] to the distributed environment. It is also an extension to the above *distributed-local* scheme. This algorithm is the last of Ali's algorithms. Since it suggests a real-time solution to garbage collection in any distributed system, it is described in greater detail. In this case, each memory operation performs a small set of the GC work. Each PE keeps tables of references to local objects referred from other objects. Also each PE sends GC messages to the other PE's in the system at the end of its local cycle of GC.

A brief overview of this algorithm is given below. In this case each PE has the following:

- a local heap which is divided into two semispaces (*fromspace* and *tospace*) whose roles interchange in each local GC cycle.
- an ODT (*Object Descriptor Table*) with a fixed size, which keeps track of objects in a PE memory.
- a set of local roots – an *OutTable* in which references to non-local objects are saved temporarily.
- a MQ (*Message Queue*) which contains computation messages that have been received from the other PE's.

- a GQ (*Garbage collection Queue*) which contains GC messages that have been received from the other PE's.

When local GC is invoked, the following actions are performed in one atomic operation:

1) All local cells accessible from the local roots are moved into the other semispace (i.e. from *fromspace* to *tospace*).

2) All moved cells are investigated as follows:
   i) If a cell refers to a local object, the object is moved only if it has not already been moved.
   ii) If a cell refers to a non-local object, a copy of the cell is saved on the respective entry of the OutTable only if it has not already been saved.

3) When all moved cells are investigated, different sets of references to accessible remote objects are stored in their respective entries of the OutTable.

4) All garbage entries in the ODT are collected by scanning the whole ODT.

5) A garbage collection message containing the respective set of references to remote objects is sent to every other PE.

6) Now the local collection is done and the computation can proceed and new cells can be allocated in the current semispace.

The above large atomic operation of GC is partitioned into small operations that are interleaved with the primitive memory operations. The task of local GC can be divided into two sub-tasks that have to be performed in order. In the first task, all accessible cells are moved from fromspace to tospace. References to accessible remote objects are saved in the OutTable. In the other sub-task, ODT is scanned to collect garbage entries and send GC messages to the other PE's. At the local GC cycle all local accessible objects are in

tospace and all non-local accessible objects have copies of their references in the OutTable.

This scheme has essentially the same space overhead as the *global-local* scheme. The most important problem that this algorithm addresses which is not taken care of in Baker's algorithm is the problem of how to guarantee moving all accessible objects from old semispace into the new semispace before running out of space. This problem has been solved by dynamically changing the amount of garbage collection work that has to be done by each memory operation to guarantee completion of each local garbage collection cycle. The consequence of this solution is that each local garbage collection cycle will be started earlier than the true flipping time (time of copying from fromspace to tospace). This requires additional space overhead, which is equal to the size of the free area at the end of each cycle. In this scheme a reference to an object may be investigated more than once since both computation and GC are simultaneously performed. The communication overhead here is equal to that in the *distributed-local* scheme, as messages are sent at the end of a garbage collection to all processors that have references to objects in the local memory. The other disadvantage of this algorithm is that it can not detect distributed cyclic structures.

3.5 Hughes Distributed GC Algorithm

Hughes [29] has presented a distributed garbage collection algorithm that is an adaptation of the mark/sweep algorithm. In this algorithm, many global garbage collections are executed in parallel. A global GC marks nodes by stamping them with the time it started, and treats a node marked if its time-stamp is the same as or later than this start time. A local GC propagates the time-stamps of root nodes on that processor to the leaves on that processor. At the end of a local GC marking messages are sent to remote objects whose time-stamps have increased. Each processor keeps track of the earliest global GC for

which it has more work to do. When no processor has more work to do for global collection T, all nodes with time-stamps less than T can be deleted.

The system supports a global address space: i.e. it is possible for objects in one processor to contain pointers to objects in another. There are two kinds of pointer objects, local and remote. Local pointers refer to objects residing in the local memory. Remote pointers span across memory boundaries. A remote pointer consists of three indirections, a *local pointer* which points to a *remote pointer* object on the same processor, the remote pointer refers to a *root* on another processor, and the *root* contains a local pointer to the final object. Roots of a processor are organized off the heap in the form of a table.

All roots and remote pointers contain a time-stamp. Time-stamps are propagated from remote pointers to roots, the time-stamp of a root is always greater than or equal to the time-stamp of any pointer to it. All the children of a root bear a time-stamp the same as or later than the time-stamp of the root. This is the condition that exists just after a local garbage collection, but time-stamps do become out of date until the next garbage collection. To keep track of how out of date time-stamps are, each processor maintains a variable *redo* which is the earliest time-stamp which may not have been properly propagated from the roots stored on that processor to the remote pointers. Time-stamps less than *redo* have properly propagated from roots to their children, and that greater time-stamps have propagated at least to the extent that the children have time-stamps greater or equal to *redo*. After a garbage collection, *redo* is set to the current time *now* which is greater than or equal to all time-stamps. Any root whose time-stamp is less than the global minimum value of *redo (minredo)* is garbage. All objects in the system whose time-stamp is less than *minredo* are inaccessible, these objects are garbage collected. All PE's co-operate in determining *minredo*.

The proposed algorithm is not truly real-time in nature – local computation comes to

a halt during local garbage collection. As compared to Hudak's algorithm [26, 27], it can not detect *irrelevant* tasks and *deadlocked* vertices. It may take a long time to detect garbage, but the Hudak and Keller's algorithm guarantees to recover garbage during the first GC after the last reference to an object is deleted. Compared to Ali's algorithm, this algorithm is efficient in storage utilization. Ali's algorithm has high space overhead as it does not distinguish between local and remote pointers, both are represented in the same manner increasing storage overhead. An ODT (Object Descriptor Table) is needed to keep track of objects in a PE memory in addition to an *OutTable* to keep references to non-local objects. This algorithm can detect and reclaim distributed cyclic structures.

Since messages need to be sent only after a local garbage collection, these messages can be batched together for efficient utilization of the communication medium. The disadvantage of this algorithm is that all processors are required to communicate to determine the value of *minredo*. If the number of processors is large this process may take a long time delaying reclamation of garbage cells. In addition this introduces high communication overhead on the inter-connection network. Hence this algorithm is suited for multicomputer systems with small number of processors.

3.6 Summary

An overview of previous distributed garbage collection algorithms is presented in this chapter. These are adaptations of algorithms developed for uniprocessor systems namely mark and sweep, reference-counting and copying collectors.

Distributed reference-counting has the advantage that it can recover garbage objects immediately, but it requires FIFO ordering of reference-count tasks which is difficult to satisfy and requires a lot of messages. The Reference-weighting algorithm overcomes this problem and also improves utilization of the communication medium.

Hudak's marking tree collector is truly real-time in nature. It can detect global garbage (including distributed cyclic structures) as well as irrelevant tasks and deadlocked nodes in the program graph. This algorithm has high space and communication overhead.

Ali's algorithms are suited for loosely-coupled multicomputer systems ranging from few processors to a large number of processors. *Global* algorithms are simple to implement. Much computational power is wasted as all processors are halted during a global garbage collection. These algorithms are suited for systems with few processors that have no real-time requirements. Real-time distributed algorithm does not waste much computational power as garbage collection proceeds in parallel with graph mutations. This algorithm has high space overhead and is suited for systems with large number of processors with real-time constraints. Ali's global algorithms can detect global cyclic structures, but distributed-local and distributed real-time algorithms are unable to detect and reclaim them.

Hughes algorithm permits global cyclic structures. This algorithm guarantees detection of all global garbage but it may take a long time to do so. This algorithm has high communication overhead and is suited for systems with a small number of processors.

CHAPTER 4

An Algorithm for Garbage Collection in a Distributed System

4.1 Introduction

This chapter presents our algorithm. The chapter begins by giving a statement of the problem. Then we describe the algorithm and present an informal proof. An analysis of distributed garbage collection algorithms is given. An analysis of this algorithm in comparison with a global distributed garbage collection algorithm is presented. Evaluation of the analytical models are presented. Section 4.2 describes the problem of garbage collection in multicomputer systems. Section 4.3 presents an overview of the algorithm. Section 4.4 presents a detailed description of the algorithm, the *mutator* and *master* algorithms and the various data structures used in the algorithm. Section 4.5 presents arguments for the correctness of the algorithm. An analysis of distributed garbage collection algorithms is presented in Section 4.6. A comparison of global garbage collection models with our proposed algorithm is presented.

4.2 Statement of the Problem

This algorithm supports a generalized model of a parallel graph reduction system (shown in Fig. 1.8). The program graph is partitioned into sub-graphs, each sub-graph residing in the local memory of a processor. Each processor evaluates the sub-graph in its local memory. During mutation, mutators *read nodes*, *allocate nodes*, and *make* or *break local* or *remote* edges. The problem is to reclaim local as well as global garbage cells with the minimum computational power and communication overhead.

4.3 Overview of the Algorithm

The algorithm is based on a loose, message-driven coordination of local garbage collections (*lgc's*) in mutating processors. A mark and sweep algorithm is used for a local garbage collection. In the marking phase all reachable nodes in the graph are marked. Unmarked local nodes are reclaimed. Nodes reachable from remote memories are painted with one of four phase colours. Active remote entries that are still reachable from a remote memory are painted with one of three colours (the active colours). These nodes are used as roots for further marking. Remote entry nodes which are painted the fourth colour (the *erase* or *stale* colour) are no longer reachable (i.e. remote entries are reclaimed two phases later after they are last referenced). We do not mark from these entries, so they and nodes reachable from them are collected as garbage. Thus nodes not reachable from the local root, nor from any active remote entry are garbage collected.

As the system slowly cycles through colour phases, to keep the colour of remote nodes up to date with the phase colours, *retrace* messages are sent once in each phase. When the last reference to a remote entry is broken, retracing messages fail to flow, and eventually the remote entry's colour becomes stale. Hence the algorithm guarantees to reclaim all remote garbage including global cyclic structures. The rate of phase transitions determines the rate of collecting global garbage, a slower phase change results in slower collection. A master algorithm co-ordinates the system phase changes. The algorithm does not require FIFO message delivery, but does require fair delivery to prevent deadlock.

4.4 Algorithm

The algorithm is described in terms of four-phase global cycles, the contents of each node, a set of inter-processor message types, an algorithm executed by a master processor(s), and algorithms executed by each mutator.

4.4.1 System Phases

To avoid strict synchronization, four cyclic colour phases are used. Our loose synchronization implies that adjacent phases overlap in real-time, so the two colours adjacent to the current phase colour are used as protective buffers against erroneous collection. The phases are identified by a cyclical sequence of colours: *red, blue, green* and *yellow*. *White* implies the absence of any phase colour. The functions *previousColour*, *nextColour*, and *eraseColour* map from colour to colour. previousColour maps from currentColour to preceding phase colour whereas the function nextColour maps from currentColour to the succeeding phase colour. eraseColour(X) is equal to nextColour(nextColour(X)) or previousColour(previousColour(X)); eraseColour(currentColour) is the colour separated from currentColour by one phase in both directions in the colour cycle. Fig. 4.1 shows a pictorial representation of the cyclic colour phases.

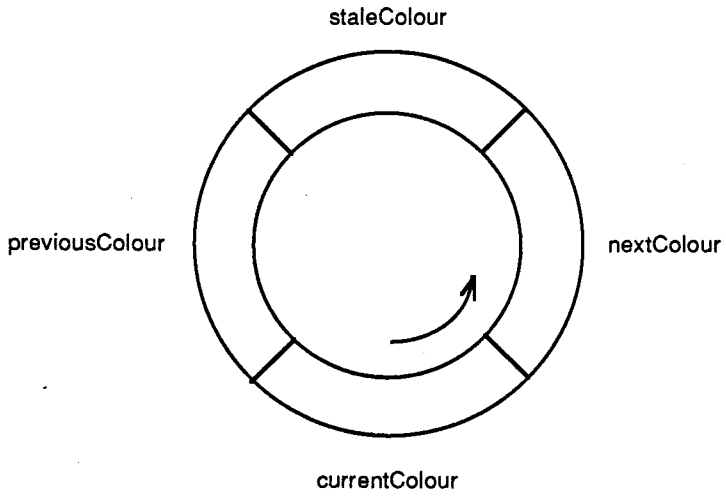


Figure 4.1 Pictorial representation of cyclic colour phases.

4.4.2 Data Structures and Node Contents

The format of each node of a graph is shown in Fig. 4.2 in Pascal-like notation. Each node has a boolean one bit field (markBit) for marking during a local garbage collection. There is a field for the *nodeColour*. The nodeColour distinguishes locally reachable nodes from remotely reachable nodes. Local nodes are painted white, whereas remote nodes are painted with phase colours. We assume (with no loss of generality) that each node represents two outgoing edges, a left edge and a right edge. Local edges are painted white. Remote edges are painted with the phase colour in which it was created or last repainted. The destination mutator and the node address to which a link is being made are stored in each outgoing remote edge.

```
type   colour = (red, blue, green, yellow, white)

type   node = record {node definition}
    markBit : boolean;
    nodeColour : colour;
    edges : array [1..2] of record
        edgeColour:colour;
        mutator : 0..mutatorCount - 1;
        node : 0..localNodeCount - 1;
    end record;
end record;
```

Figure 4.2 Representation of a node in the graph.

4.4.3 Message Types

There are six types of messages employed in our scheme, they are:

changePhase	–	master signalling a phase change
forceLgc	–	master requests that *lgc's* be undertaken, if none has been started on the phase indicated by the message.
lgcStart	–	mutator reports that it is starting an *lgc*.

lgcDone	–	a mutator reports completion of an *lgc*.
paintNewLink	–	a mutator creates a remote edge and requests the destination mutator that the destination node be painted.
refreshLink	–	an *lgc* discovers a live remote edge and requests that the destination node be re-painted.

The format of a message in Pascal-like notation is shown in Fig. 4.3. All messages carry a colour tag, the source and destination processor addresses, and message-specific information. For every message type, there is a corresponding acknowledgement message. Every message sender increments a count for each type of message sent. When acknowledgements are received, the count is decremented. Acknowledgements return the original contents of the message with a modifier bit turned on. Thus the sender receives an echo of the information sent.

```
type  messageClass = (changePhase, forceLgc, lgcStart,lgcDone, paintNewLink,
                      refreshLink);

type message = record
    class    : messageClass;                    { message type }
    phase    : colour;                          { phase colour of mutator }
    source   : 0..mutatorCount - 1;             { source mutator address }
    destination      : 0..mutatorCount - 1;     { destination mutator address }
    nodeAddress      : 0..nodeCount - 1;        { node address to paint }
end record;
```

Figure 4.3 Representation of a message.

4.4.4 Master Algorithm

One or more processors are specialized as a tree of masters. The number of master processors is dependent on the size of the system. Each master maintains the local garbage collection status of a subset of processors. Co-ordination exists among master processors to determine the overall *lgc* status of all processors. Assume for now without loss of

generality only one *master* is required. The master determines when phase changes occur. It keeps counts of outstanding changePhase and forceLgc messages, and maintains a table *mutatorStatus* which records the *lgc* status of each mutator in the current phase.

A mutator notifies the master when it starts an *lgc* by sending an *lgcStart* message and an *lgcDone* message on completing *lgc*. No action is taken if the colour tag on the *lgc* message does not match with that of master (i.e. late lgcStart and lgcDone reports from the previous phase are ignored).

There are three states for a mutator in mutatorStatus table. A processor is in the *noLgcStart* state if it has not yet started an *lgc* in the current phase. On receiving the first lgcStart message, its status is updated to *lgcStart* state. On completing the first *lgc* the master sets the processor status to *lgcDone*. The mutator status remains in the lgcDone state no matter how many *lgc's* it completes in the current phase.

The master algorithm is shown in Fig. 4.4. A phase change can be initiated by the master if all mutators have reported lgcDone at least once during the current phase, i.e. they are not in the noLgcStart state and the counts of outstanding changePhase and forceLgc messages (of previous phase) are zero. To start a new phase, the master sets its masterColour variable to *nextColour(masterColour)*, sets all mutatorStatus's to noLgcStart and broadcasts a changePhase message to every processor declaring the colour of the new phase. Although the master's phase change may be unblocked, it may wait some minimal time – *expectedPhaseTime* – before it starts a new phase. If elapsed time approaches expectedPhaseTime with initial *lgc's* not reported by some mutators, the *master* sends a forceLgc message to the delinquent mutators. The expectedPhaseTime and the forceLgc's are optional to the working of the algorithm.

```
algorithm master

on arrival of  lgcStart message                          { receive an lgcStart  message }
begin
    case mutatorStatus[i] of
        noLgcStart      :       mutatorStatus[i] = lgcStart;
        lgcDone         :       mutatorStatus[i] = lgcDone
    end
end;

on arrival of lgcDone message                            { receive an lgcDone message }
begin
    case mutatorStatus[i] of
        lgcStart        :       mutatorStatus[i] = lgcDone;
        lgcDone         :       mutatorStatus[i] = lgcDone
    end
    checkMutatorStatus();                                { check mutators status }
end;

procedure checkMutatorStatus();
    noOfMutatorsDoneLgc := 0;
    for i := 1 to mutatorCount do
        if mutatorStatus[i] = lgcDone) then      { count mutators that have done at least one lgc }
            noOfMutatorsDoneLgc :=  noOfMutatorsDoneLgc + 1;
        if (noOfMutatorsDoneLgc >= thresholdLevel) then
            changePePhase()                              { change system phase }
end;

procedure changePePhase();
    masterPhase = nextColour(masterPhase);
    for i := 1 to mutatorCount do
        sendMsg(changePhase, i)                          { send phaseChange msgs }
end;
```

Figure 4.4 Master Algorithm.

4.4.5 Mutator Algorithm

Mutators are in one of two states, mutation or local garbage collection. Messages

from other processors periodically interrupt these activities. Fig. 4.5 illustrates the mutator algorithm. The mutation operations are illustrated in Fig. 4.6. Mutators maintain a local phase colour register (localColour), four outstanding message counters (*paintNewLinkOut, refreshLinkOut, lgcStartOut* and *lgcDoneOut*), a pointer to the local root of the graph (localRoot) and a pointer to the free list.

When a node is allocated, its colour (*node.nodeColour*) is set to white. When a local edge is made, the colour of the edge (node.edge[x].edgeColour) is set to white. When a remote edge is constructed, the edge colour is set to the mutator's current colour (localColour), a paintNewLink message containing the current colour (localColour) is sent to the destination mutator, and the local counter – paintNewLinkOut – is incremented. New mutation phases block until the paintNewLinkOut message counter is zero.

```
algorithm mutator
    while (true)                                        { begin an infinite loop }
        if (freeList is not empty) then
            mutation;                                   { mutation }
        else
            localGarbageCollection                      { garbage collection }
end;                                                    { end while loop }

procedure mutation();
    wait (paintNewLinkOut = zero)       { previous mutation messages unacknowledged }
                                        { delay starting of  mutation phase }
    mutate                              { do mutation }
end;

procedure localGarbageCollection();
    wait (lgcDoneOut = zero and refreshLinkOut = zero)
                                        { previous lgc messages unacknowledged }
                                        { delay starting of local garbage collection }
    localGarbageCollect                 { do a local garbage collection }
end;
```

Figure 4.5 Mutator algorithm.

```
algorithm mutate

procedure allocateNode();                          { allocate a node }
begin
    if freeList is empty then                      { if free list is empty, start an lgc }
        localGarbageCollection();
    else
        begin
            node := remove first node from freeList;   { remove a node from free list }
            node.nodeColour := colour.white            { paint node white }
        end
end;

procedure makeLink(sourceNode, destinationNode);
begin
    if destinationNode is in local memory then     { destination node is in the same }
                                                   { memory as source node }
        makeLocalLink(sourceNode, destinationNode);
    else
        begin
            makeRemoteLink(sourceNode, destinationNode);
            sendMsg(newLinkPaint, destinationProcessor)
        end                                        { send a new link paint message }
end;

procedure breakLink(sourceNode, destinationNode);
begin
    sourceNode.linkAddress := null                 { replace link address by a null value }
end;
```

Figure 4.6 Mutation Operations.

```
algorithm localGarbageCollection

procedure localGarbageCollection();
begin
    sendMsg(lgcStart, master);              { send start lgc message to master }
    markPhase();                            { do marking }
    remotelyOwnedCells();                   { mark remotely owned nodes in graph }
    sweepPhase();                           { sweep memory and reclaim unmarked cells }
    sendMsg(lgcDone, master)                { send lgc done message to master }
end;

procedure markPhase();
begin
    for i := graphRoot to allNodes do
        node.markBit := clear;                                  { reset all mark bits }
    for i := graphRoot to allNodes do
        if node.nodeColour = colour.white then                  { an unmarked local node }
            if node unmarked then
                node.markBit := true                            { set mark bit to true }
        else                                                    { send a refresh paint message }
            sendMsg(refreshPaint, destinationProcessor)
end;

procedure remotelyOwnedCells();
begin
    for i := graphRoot to allNodes do
        if (node.nodeColour not white) and ( node.nodeColour not equal to staleColour ) then
            mark from this node                 { node is reachable from remote memory }
end;                                            { mark from this node as root }

procedure sweepPhase();
begin
    for i := graphRoot to allNodes do
        if node is unmarked then
            insert node into freeList
end;
```

Figure 4.7 Local Garbage Collection Algorithm.

New *lgc* phases block until the counters lgcDoneOut and refreshLinkOut are zero. At the beginning of an *lgc* an lgcStart message is sent to the master. Almost any variant of non-compacting mark-sweep garbage collection can be adapted for the *lgc's*. We describe the simplest. 1) All markBit's are reset. 2) The graph is traversed from localRoot. The markBit's of all reachable local nodes are set. When the graph has an edge to another memory, if the colour of that edge is not equal to localColour, a refreshLink message is sent with the current localColour, refreshLinkOut is incremented, and the edge is marked with localColour. 3) The local memory is scanned for unmarked nodes with colour not equal to *eraseColour(localColour)*, and the mark procedure (2) is carried out from all such nodes. Nodes not reachable at all, and nodes reachable only from nodes of the eraseColour are not marked. (4) All unmarked nodes are gathered in a linked free list.

An lgcDone message is sent only when refreshLinkOut reverts to zero. This may occur after mutation has resumed. The lgcDone message carries the colour used in the *lgc*, even if the phase changes in the meantime. The local garbage collection algorithm is illustrated in Fig. 4.7.

4.4.6 Response to Messages

Mutator's respond to messages in the following manners: 1) *paintNewLink* and *refreshLink*: The specific local node is painted with the colour carried by the message. 2) *changePhase*: Mutators set their localColour register to *nextColour(localColour)*, unless they are engaged in an *lgc*, in which case they defer the reset of localColour until the *lgc* is complete. 3) *forceLgc*: Mutators begin an *lgc* if they have not completed one in the phase indicated by the colour carried by the message. Acknowledgements to all message types are generated and received as described earlier. Fig. 4.8 illustrates the response to various messages.

```
algorithm responseToMessages

on arrival of paintNewLink message                { receiving a paintNewLink message }
begin
    node.nodeColour := paintNewLink.colour;       { paint local node with message colour }
    sendMsg(ackPaintNewLInk, i)                   { send acknowledgement message }
end;

on arrival of refreshLink message                 { receiving a refreshLink message }
begin
    node.nodeColour := paintNewLink.colour;       { paint local node with message colour }
    sendMsg(ackPaintNewLInk, i)                   { send acknowledgement message }
end;

on arrival of changePhase message                 { receiving a changePhase message }
begin
    mutator.localColour := changePhase.colour;    { change mutator colour to msg colour }
    sendMsg(ackChangePhase, master)
end;

on arrival of forceLgc message                    { receiving a forceLgc message }
begin
    if noLgcStarted in currentPhase then          { if lgc not started in current phase }
    begin                                         { start an lgc }
        startLocalGarbageCollection;
        sendMsg(startLgc, master)
    end
end;
```

Figure 4.8 Response to Messages.

4.4.7 Initial Conditions

Initially, the distributed memory is loaded with the graph to be processed, all processors are given *red* as their initial colour (it could as well be any of the other phase colours) and all remote edges and all remotely reachable nodes painted red. All message acknowledgement counts are set to zero, and the mutatorStatus array is set to all

noLgcStart.

4.5 Arguments for Correctness of the Algorithm

In this section we present an argument for the correctness of the algorithm. We need to establish three properties:

1) no sub-graph that is reachable from any local root is collected as garbage
2) all sub-graphs that are unreachable from any root are eventually collected as garbage.
3) assuming fair delivery of messages, deadlock does not occur.

To prove the above properties, recall the nature of phase cycles. The phase cycles are characterized by:

i) The processors follow a loosely synchronized cycle of four colour phases.
ii) It is possible for adjacent phases to overlap in the sense that local garbage collections from two phases are active at once.
iii) It is not possible for non-adjacent phases to overlap.

To prove the first two properties: Each processor performs a local garbage collection when it runs out of local heap space and reclaims all garbage cells that are locally referenced. Remote nodes (i.e. nodes referenced from remote processors) are painted with phase colours. *refreshPaint* messages are sent to remote processors for repainting all active nodes with the colour of the current phase. Hence it is ensured that every active remote link is painted in a phase. The next mutation phase is not begun until all the acknowledgement messages for these repaint messages are received. This guarantees that no active remote nodes are collected as garbage.

Inactive remote nodes lose links with nodes in the graph. Their colour is not refreshed in ensuing phases. Any node that was last painted with the staleColour is

considered as garbage. These nodes and all nodes reachable only from them are reclaimed. Thus global cyclic structures having links spanning across memories are guaranteed to be reclaimed in two phases. Although adjacent phases can overlap, all of the repainting messages of a phase must have taken effect (i.e. painted the remote node) before the next phase begins.

A process in a concurrent system is said to be in a deadlocked state if is waiting for an event that can never occur. A situation may arise with two concurrent processes in which both can not proceed as each of them is waiting for a resource the other is holding. In a distributed algorithm deadlocks can be expensive or disastrous, an algorithm must be free from deadlocks. We can prove that the present algorithm is free from deadlock from these observations: local garbage collection blocks only due to unacknowledged paintNewLink messages or an unacknowledged lgcDone message of the previous *lgc*. Similarly starting a mutation phase blocks until all refreshLink messages are acknowledged. Phase change by the master blocks until all phase change messages and forceLgc messages are acknowledged. Assuming that the message deliver is fair (not necessarily FIFO) none of the messages get blocked preventing proper operation of the overall algorithm.

4.6 Analysis of the Algorithm

In this section we present an analytical model of our algorithm. First we present a model of a globally synchronous garbage collection algorithm. This model motivates and justifies our present algorithm. A cost benefit analysis of our loosely-synchronous algorithm in comparison with the cost of a global algorithm is presented. The analytical results derived here serve as a basis for our simulator model and simulation experiments that are presented in the next chapter.

4.6.1 Cost of Global GC in a Distributed System.

In globally synchronous garbage collection all processors collect garbage at once. Some processors may have run of space before the starting of a global garbage collection, these processors have to wait until a garbage collection is initiated. Other processors have remaining free space when the garbage collection begins. The times at which processors need to do a garbage collection can be assumed to be normally distributed as shown in the Fig. 4.9.

The vertical line TGC (Time of Garbage Collection) is the starting time of a global garbage collection. The area under the curve to the left of TGC is a measure of the number of processors exhausting their storage before the start of a global garbage collection. This represents CPU cycles lost because of idling while waiting for a garbage collection. The area to the right of TGC is a measure of the proportion of processors that do a premature garbage collection. These processors do a garbage collection in spite of having free space. The extra free cells collected may not be utilized until the next garbage collection. This results in the loss of useful cycles spent in collecting these extra cells. If TGC is moved to left, i.e. garbage collection initiated as soon as one or a few processors have run out of space, the CPU cycles lost by idling processors decreases and the cycles lost by processors that do a premature garbage collection increases. Conversely by moving TGC to the right (i.e. delaying garbage collection until most of the processors have run out of space), more CPU cycles are lost by idling processors and cycles lost by premature garbage collection decreases.

The cost of any global garbage collection depends on the following parameters:

CCPC – Cells Consumed Per Cycle
CRTC – Cycles required to reclaim a Cell
TGC – Time of GC
TOM_i – Time Out of Memory for *PE i*

MRTC – Messages Required to Reclaim a Cell

The total number of CPU cycles lost is the sum of the cycles lost by idling processors and the number of cycles lost because of premature garbage collection. The total number of CPU cycles lost is given by the following equation:

$$lost\ \ cycles\ = \sum_{PE}^{N1} (TGC\ -\ TOM_{PE}) + \sum_{PE}^{N2} (TOM_{PE}\ -\ TGC)\ *\ CCPC\ \ *CRTC$$

$N1$ = number of processors idling due to delayed garbage collection
$N2$ = number of processors performing premature garbage collections.

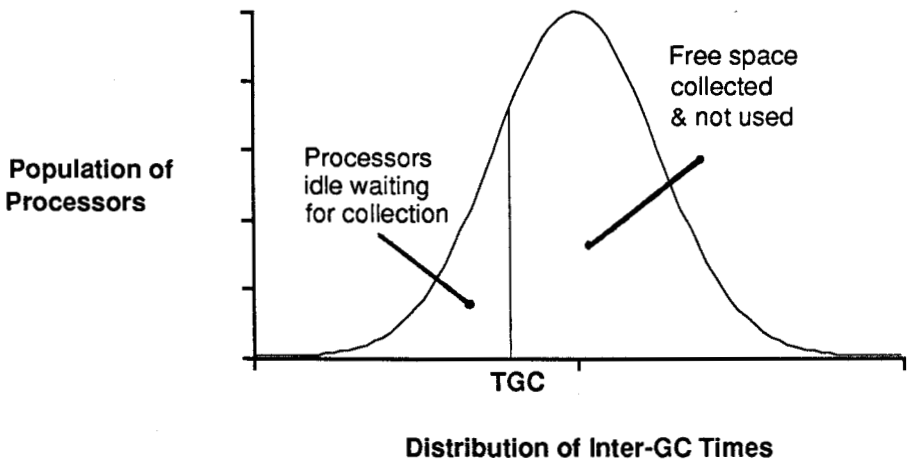


Figure 4.9 Distribution of cost in a global Garbage Collection scheme.

There is a cost associated with each garbage collection message that needs to be sent during garbage collection (due to remote edges). The cost of messages can be approximated by the following equation:

$$wasted\ \ messages\ = \sum_{PE}^{N2} (TOM_{PE}\ -\ TGC)\ *\ CCPC\ \ *\ MRTR$$

The total cost of garbage collection is the sum of the cost due to lost cycles as well

as the cost of wasted messages. An evaluation of the above model was conducted to observe the variation of the number of lost cycles with a varying TGC. The analysis was done on a system having a normal distribution (as shown in Fig. 4.9) with a mean time of 3000 units and a standard deviation of 1000 and 750. TGC was varied from 1500 units to 6500 units in increments of 50 units. The number of lost cycles for each value of TGC was recorded. Low (premature starting of GC) as well as high value of TGC (postponement of GC) increases the number of cycles lost during a global garbage collection. There is a range of TGC values for which the number of lost cycles is minimum. Irrespective of the time of starting a global collection a certain number of cycles is always lost.

4.6.2 Lost memory model

In this section we present a model to account for lost memory in our algorithm. As mentioned in the earlier discussions our algorithm is characterized by independent *lgc's* and global phases. Mutators perform a local garbage collection whenever they run out of local heap space. Assuming good locality in the program graph, mutators will recover most of the garbage during a local garbage collection. Local garbage is recovered independent of the phase changes.

Two phases are required to recover global garbage. The remote garbage generated accumulates through a two phase length period. Assuming a uniform generation of remote edges, some percentage of memory is always lost due to unreclaimed remote garbage.

A cost analysis of this algorithm is given below:

Let, m = size of memory in each mutator.
r = 0.5 = storage theoretically freeable in each *lgc*.
x = proportion of memory not reclaimable due to remote garbage not being discovered

l = memory lost till phase change.

Then memory reclaimed per phase is given by $(m - l) * 0.5 - l$

An evaluation of the above model was conducted to observe the behaviour of the algorithm for varying values of x and phase lengths. The results are shown in Fig. 4.10. When x is low the percentage of local to remote links is high (i.e. only a few remote links are created). Memory loss due to accumulating remote garbage is low. The accumulating losses remain low even for longer phases. As x increases, the accumulating memory losses increase rapidly with increasing length of phases.

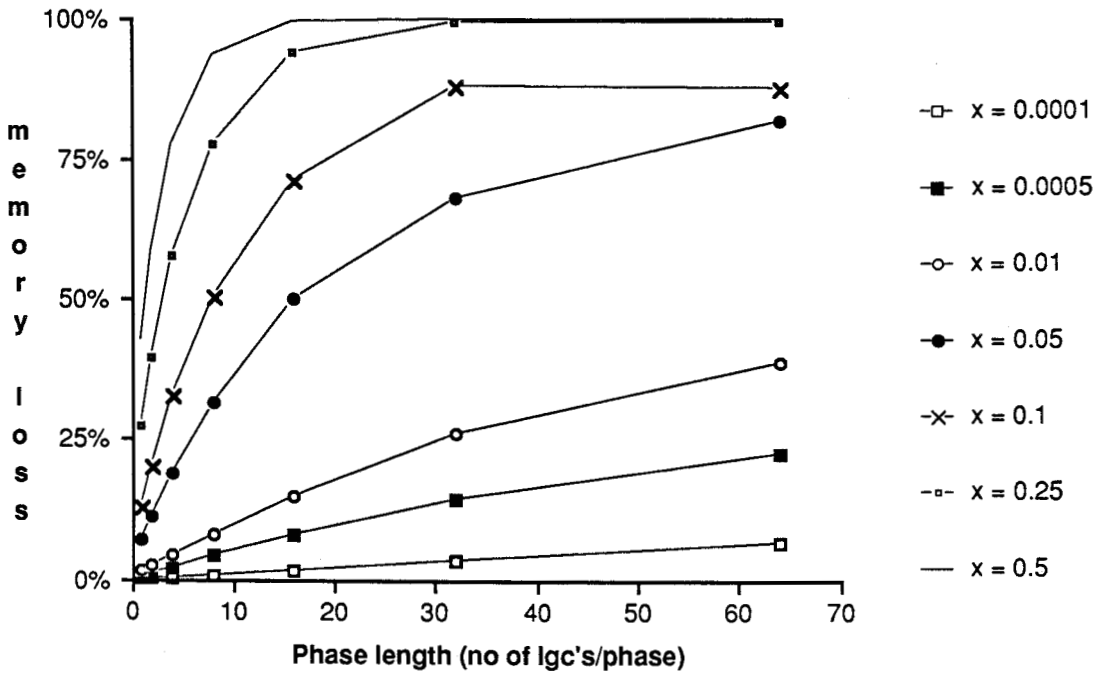


Figure 4.10 Variation of memory loss with phase length.

The cost of messages (newLinkPaint and refreshPaint) is estimated next. Each newly created remote edge requires a newLinkPaint message. Once in a phase we need to send refreshPaint message for every active remote edge. Let us define r as the ratio of

newly created remote edges to the sum of new and static remote edges, i.e.

$$r = \frac{newremote}{(newremote + static)}$$

As mentioned earlier refreshPaint messages are sent only once in a phase during the first *lgc*. After the first *lgc* the message traffic in a the system is only due to newLinkPaint messages. The number of newLinkPaint messages sent depends on the number of new remote links made. For a given value of *r*, the number of messages per *lgc* decreases with an increasing phase length. A lower value of r decreases the rate of generation of new remote edges. Hence the number of newLinkPaint messages decreases. The analysis was conducted on a system with a memory size of 10^6 cells, for different values of *r* (ranging from 0.09 to 0.33) and phase lengths (from 1 to 48 *lgc's* per phase) to observe the variation of message load with phase length. The simulation results obtained are shown in Fig. 4.11.

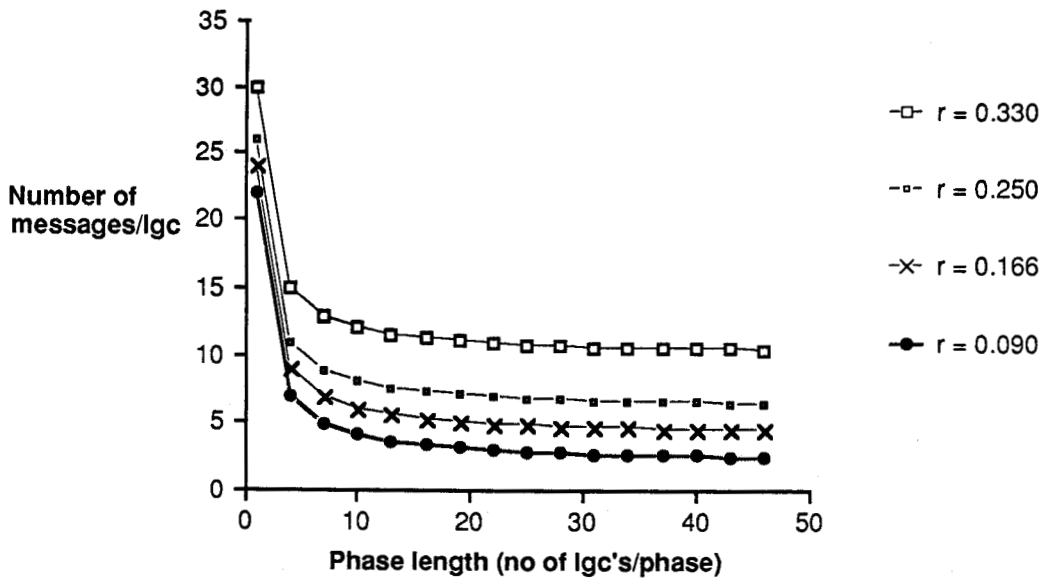


Figure 4.11 Variation of number of messages/lgc with phase length.

4.7 Conclusions

This chapter presented our algorithm for garbage collection in a parallel distributed system. This algorithm overcomes deficiencies present in the existing algorithms; it supports a generalized model of parallel graph reduction and guarantees to reclaim all global garbage including cyclic structures in two phases.

The other major strength of the algorithm is the flexibility provided to the system designer in optimizing the performance by tuning certain parameters such as the phase length and the ratio of local to remote links in the collection scheme.

A global garbage collection scheme has a cost associated with it in terms of the lost CPU cycles and wasted messages. A non-zero cost is always associated with this algorithm no matter when the global garbage collection is done.

The lost memory model of our algorithm shows the dependency of the overall cost of the algorithm on the phase length and the percentage of generated remote links. The percentage of remote links created depends on the locality in the partitioning of the program graph. The algorithm and the analytical model presented here is validated by simulation. The next chapter presents the simulator and the results obtained.

CHAPTER 5
Simulation

5.1 Introduction

In this chapter we present simulations of our algorithm. The objective of the simulator is to validate the working of the proposed algorithm and to analyze its performance. The simulator models a generalized parallel graph reduction system that uses the proposed garbage collection algorithm for memory reclamation. Experiments were conducted to verify the algorithm, and observe phase transitions and other performance characteristics. A description of the experiments and their analysis is presented.

The simulator was implemented in Turing-Plus, a concurrent programming language. Some key features of this language are presented in section 5.2. The design of the simulator is presented in section 5.3. Section 5.4 describes the implementation of the simulator. Section 5.5 presents the simulation experiments conducted and the results obtained.

5.2 Turing-Plus language

Turing-Plus is a concurrent programming language that is well-suited for writing operating systems, and low level kernel modules like device drivers. The key features of this language are its concise and expressive syntax, graceful and efficient treatment of errors, good software engineering features like modules and mathematically precise language definition. The language is free from various inconsistencies and insecurities that are prevalent in languages like Pascal and 'C'. The non-concurrent features of Turing Plus are similar to that of Pascal. Concurrency features include re-entrant procedures and functions, *monitors* (for *mutual exclusion*), *wait* and *signal* (for interprocess

communication) conditions.

Sometimes several processes make use of similar algorithms. Processes can share algorithms by calling the same procedure. All procedures and functions in Turing-Plus are *re-entrant*. Re-entrant procedures and functions are essential for software such as operating systems as they make the software easier to understand, and make the code smaller.

When processes need to update common data, the data may be corrupted if more than one update takes place in parallel. *Monitors* are provided in a concurrent programming language to guarantee mutually exclusive access to common data. A monitor can be considered to be a fence around the data; all code accessing the data is gathered into procedures and functions and moved inside the fence. Processes wishing to access the data do so by entering a *gate* or *entry* in the fence, to execute one of these procedures or functions. The monitor guarantees that only one process is active inside the fence at a given time.

Concurrent processes must synchronize their activities. A typical situation occurs when processes compete for shared resources. Once a resource is allocated to one process, another process needing the resource should be blocked until the first process releases it. A processor blocks itself by executing the *wait* statement if the resource is not available. The process that is inside the monitor after relinquishing the shared data executes the *signal* statement to *wake up* one waiting process (if there is one) before exiting the monitor. If there is no waiting process, the signaler just continues.

5.3 Design of the Simulator Model

In this section we present the simulator model of our algorithm. We model a parallel graph reduction system that uses our proposed garbage collection algorithm for

memory reclamation. The *mutator* and the master processors are represented as processes. Modelling a distributed algorithm is quite complex due to the parallel events that occur in a distributed system. The logical view of concurrent processes facilitated by the language makes the understanding of the simulator easier and expresses the underlying distributed system elegantly. The facility of modules simplifies the simulator design as it encourages good software engineering practices.

One of the key elements in a simulation study is the load used. The data used for simulation must be representative of the real-world environment. Generally this data is acquired from real systems over long periods of time. These include *reference strings* (trace of the memory addresses generated by an executing computer program), *process* and *work load* statistics. The advantage of using such loads is that the results will be more realistic. The major drawback of real data is that they consume a great deal of storage. For example a reference string may represent millions of addresses referenced in a second of real execution time, and each reference may require several bytes of storage to record the address and the type of reference. This can result in a very unwieldy body of information that is both time and space consuming.

Synthetic loads are generated from a model. They are generated *on the fly* hence do not require much storage. They have the advantage of being *flexible* (simple modifications to a few parameters can produce a load with different characteristics) and *reproducible* (the same load can be generated when needed by keeping the parameter settings the same). However, caution should be exercised before drawing conclusions based on these synthetic loads; their representativeness of real systems must be determined.

Parallel graph reduction as a way of implementing functional languages is relatively a new idea. This concept is still in the research phase, many underlying practical issues have not yet been fully understood. Moreover, there is no literature available on existing

parallel graph reduction systems. In our simulations we had the option to use programs written in a functional language (such as Lisp or Miranda) as a working load. Translating these programs to an equivalent parallel graph reduction system is not simple. Moreover the results would be dependent on specific loads. To overcome these problems we use a synthetic load. As mentioned above this approach gives flexible and reproducible results as well as generalizes the model.

5.4 Implementation of the Simulator

As mentioned earlier each mutator and the master are implemented as processes. Each mutator has a local colour, a *free list*, a pointer to a local graph, counters for outstanding messages, and a *local clock*. The *master process* has the *mutator status table* (to keep track of the *lgc* status of mutators in the current phase), a *local clock* and counters for outstanding messages. All shared data such as message buffers are kept in a *critical section* and buffer operations are done inside a monitor.

The simulator load is generated dynamically based on probability distributions. The *random mutator model* recognizes three basic graph reduction operations namely :

i) Allocate a node.
ii) Make a link
iii) Break a link.

Each of these random operations is associated with a probability of occurrence. There is a simulation time associated with each of these events required to execute it (relative times rather than absolute values). In the first operation of *allocating* a node, a cell is removed from the free list and inserted into the graph at some random location. The point of insertion in the graph is determined by a random walk through the graph. This is started from the root of the graph. The inserted cell is painted white. This random walk

ensures a uniformly grown graph.

In the second operation, a link is made between two nodes of the graph. This operation may be either *local* or *remote*. As in the allocate operation, a random search through the graph is done to identify a *source node*. To make a link locally, a random *destination node* is identified in the same subgraph. For a remote link operation the *destination node* is identified in a remote subgraph. A newly made link may result in a cyclic structure. Even self cycles (a node having a link to itself) are permitted.

In the case of making a remote link, a destination processor is randomly chosen. A random walk is done through the remote graph to identify a destination node. The destination processor address, destination node address, and the current phase colour is registered in the source node of the newly created remote link. A *newLinkPaint* message is sent from the source processor to the destination processor. The destination processor responds to this message by painting the destination node in its graph with the colour carried by the message and sends back an acknowledgement message.

In the third operation, a link is broken. This could be either a local or a remote operation. A link is randomly selected in the graph, the destination processor and the node address is replaced by a null value to indicate a deleted edge.

On executing a random event, a mutator's clock is advanced by the time associated with that event. Mutators (and the master processor) are scheduled based on their simulated timers. The scheduler compares the clocks of all processes and identifies a process with the lowest clock. This process is *signalled*, the signalled process *wakes up* and all other processes remain blocked in the *waiting* state. This scheduling imitates the parallel operations as the clock of a process is frozen until it is woken up and it advances only when it is being scheduled. Thus all processor clocks advance in a step-by-step

fashion.

In order to bypass initial transient behaviour, an initial graph is built in all mutators. This makes it possible to carry out random searches through the graph from the very beginning. This helps to bring the system to steady state more quickly.

5.5 Simulation Experiments

In this section we describe the simulation experiments that were conducted. The objective of the experiments were to:

i) Verify the working of the algorithm
ii) Determine the percentage of memory loss for varying phase lengths.
iii) Observe the variation in the number of messages per *lgc* with different phase lengths.
iv) Observe the effect of interprocessor communication delay on the cost of the algorithm.

In each simulation run, the following statistics are generated: number of cells marked, number of cells reclaimed, number of newLinkPaint messages, number of refreshLinkPaint messages, the wait time to start an lgc (*lgcWaitingTime*) and mutation (*mutationWaitingTime*). Starting of a mutation phase is blocked until all the outstanding refreshLinkPaint messages are acknowledged and starting of *lgc* is blocked until all the outstanding newLinkPaint messages are acknowledged. The details of the experiments are given below.

5.5.1 Verification of Phase Transitions

As said before, the simulator load is determined by the probability distribution of graph operations. The percentage of local to remote links determines the number of remote edges created which also accounts for the number of messages generated in the system. The duration of a phase is determined by the average number of *lgc's* per processor in a

phase.

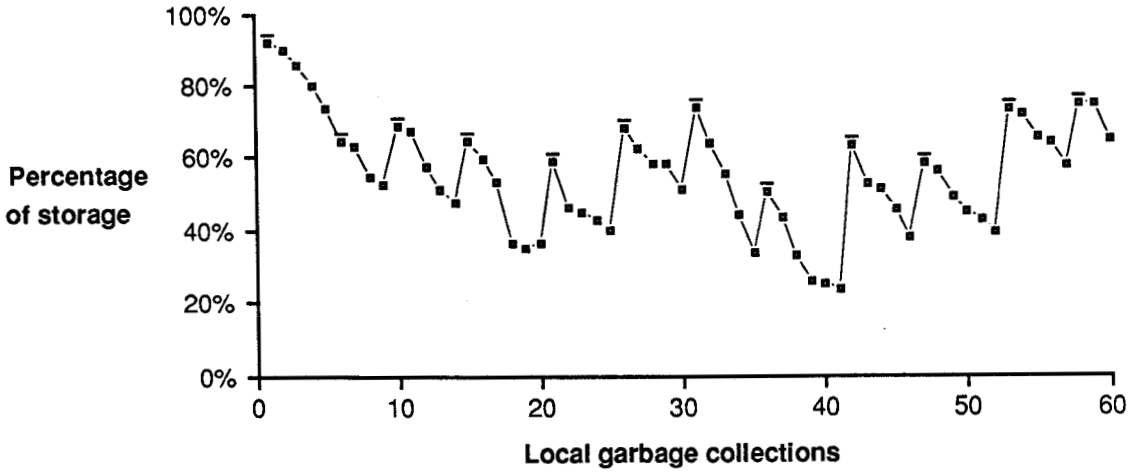


Figure 5.1 Illustration of phase changes (system with 16 processors, phase length of 5 *lgc's*/phase)

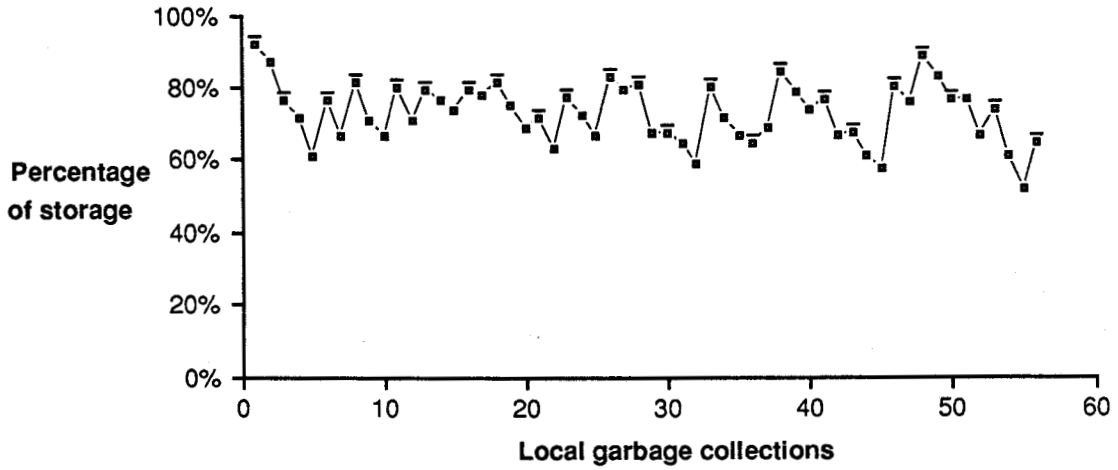


Figure 5.2 Illustration of phase changes (system with 16 processors, phase length of 2.5 *lgc's*/phase).

The simulator was run with a constant probability distributions and a fixed ratio of local to remote links for different varying phase lengths. The results obtained are illustrated in Fig. 5.1 and 5.2. The plots show the variation in memory reclamations with the phase transitions. Each point on the graph indicates completion of an *lgc*. The point with a bar

on top indicates a phase change. The number of cells reclaimed from *lgc* to *lgc* gradually declines in a phase. With a phase transition the number of reclaimed cells shoots up. This is due to the reclamation of global garbage. As mentioned earlier, remote garbage cells are reclaimed two phases after they are last painted. In the first phase transition there is no increase in the number of reclaimed cells. The number of reclaimed cells continues to decrease in the second phase since no global garbage is ready to be reclaimed in this phase.

With an increasing phase length, the percentage of memory loss increases. During each local garbage collection, all local garbage is reclaimed but the remote garbage is not reclaimed. The unreclaimed global garbage contributes to the memory loss. In the steady state the percentage of memory loss remains approximately constant for a given phase length. The percentage memory loss increases with increasing phase length. A decrease in the percentage of remote links lowers the memory loss. This saw tooth response is in accordance with the expected response from the algorithm. These results are in accordance with the analytical model shown in Fig. 4.7 in Chapter 4.

5.5.2 Variation of Message Load with Phase Length

One of the key characteristics of our algorithm is that the number of messages required for each newly made remote link is constant. Each remote link contributes to two messages per phase. A refreshLinkPaint message is sent for every active remote link during the first *lgc* in a phase. During the subsequent *lgc's* these messages are not sent. For every new remote link created a newLinkPaint message is sent during the mutation phase. As the number of *lgc's* per phase i.e. the phase length increases, the message load is contributed mainly by the newly created remote edges (by newLinkPaint messages). The contribution of refreshLinkPaint messages is restricted to the first *lgc*. Hence the number of messages per phase per processor decreases with the increasing phase length. The

converse of this statement is that the number of cells reclaimed per message increases with an increasing phase length.

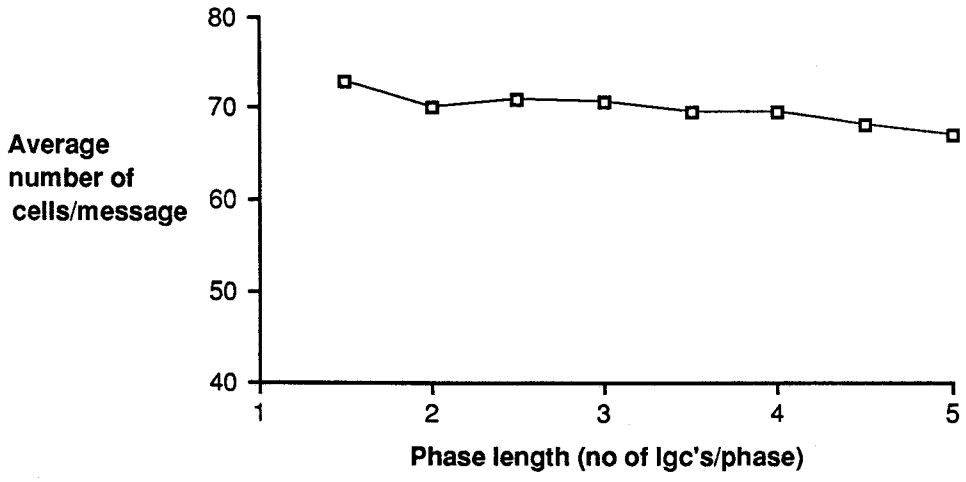


Figure 5.3 Variation of cells reclaimed per message with phase length.

Simulations were conducted to observe the variation of message load (in terms of the number of cells reclaimed per message in a phase) with varying phase length. The results obtained with phase length varying from 1.5 to 5 *lgc's* per phase is shown in the Fig. 5.3. The number of cells reclaimed per message remained approximately constant. As stated before we expected the average number of cells per message to increase with increasing phase length. The discrepancy in the results may possibly be attributed to the simplifications assumed in the simulator design as well as the synthetic load. Nevertheless our argument remains convincing, but more work is needed to substantiate this claim.

5.5.3 Effect of Network Delay on Waiting Time.

The message transmission delay is dependent on the inter-connection network in the system. Delay in the communication network is due to circuit switching, limited network bandwidth, and congested and broken links. The design of an appropriate inter-connection network is crucial to the performance of any loosely-coupled distributed

system.

In a parallel graph reduction system, message delay plays an important role. Mutators communicate with the master processor at the beginning and end of every local garbage collection. Also the master processor needs to communicate with the mutators during phase transitions. The mutator messages undergo master processing delay in addition to the communication delay. The message delay affects the waiting time of mutators. A mutator will not begin a new mutation phase until it receives acknowledgement messages for all previous newLinkPaint messages. Similarly a local garbage collection is not begun until all acknowledgement messages for previous refreshLinkPaint messages are received.

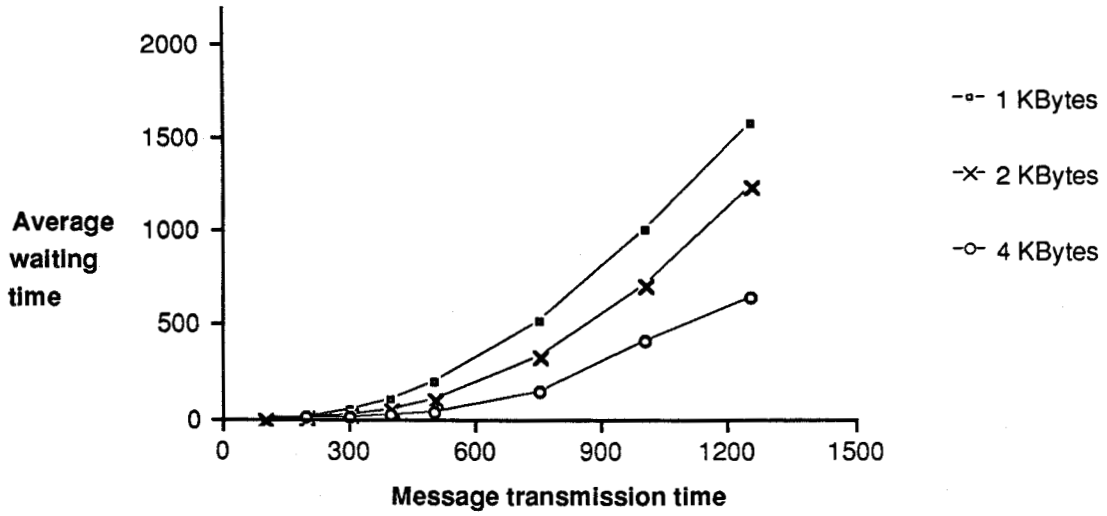


Figure 5.4 Variation of waiting time with message transmission time (system with 16 processors, phase length varying from 1 to 5 *lgc's* per phase).

An experiment was conducted to observe the effect of message transmission delay on the mutator waiting time. The simulator was run with a constant phase length with a varying message delay; the corresponding waiting times were observed. The experiment

was repeated for different memory sizes. The plots shown in Fig. 5.4 illustrate the results obtained. The waiting time increases non linearly with the message delay. Assuming good locality in the system we can say that most of the time mutators need to communicate with their neighbors so that the communication delay will be reduced significantly. The locality depends on the partitioning of program graph and is crucial to the performance of any parallel functional language. But the locality factor was not taken into consideration in our simulator model.

5.6 Conclusions

This chapter presented the design of the simulator model and discussed the experiments conducted. The simulation results are largely in accordance with the expected behaviour of the algorithm. The synthetic work load enabled us to verify the algorithm. The work load did not take into consideration the issues involved in a parallel graph reduction system such as partitioning of the program graph, locality and the nature of reduction scheduing.

The results in Fig. 5.1 and 5.2 show the phase transition behaviour, reclamation of global garbage, and the resultant memory loss with varying phase lengths. From the simulations conducted on systems with different number of processors (4, 8, 16 and 32 processors), memory sizes (1KBytes, 2KBytes, 4KBytes per processor), loads (different probability distributions) and phase lengths (from 1 to 5 *lgc's* per processor per phase) it seems that the algorithm is sound and robust. The experiments to observe the variation in the message load with increasing phase lengths did not give the results as expected (Fig. 5.3). The results shown in Fig. 5.4 depict the effect of inter-communication network delay on the overall cost of the algorithm in terms of the waiting delays. These simulation results provide insight into the working of a parallel graph reduction system and must be taken into consideration for optimal performance.

CHAPTER 6
Petri Net Modelling of the Algorithm

6.1 Introduction

This chapter presents a Petri Net model of the algorithm. Petri nets are a tool for the study of systems [43]. Modelling refers to the study of a phenomenon indirectly through a simplified abstract representation of the phenomenon. Petri nets have been found to be useful in modelling systems exhibiting *concurrency* and *parallelism*. They have been used in the modelling of diverse systems ranging from computers to social and legal systems. Distributed algorithms are quite complex to model and analyze mainly due to the concurrent activities in a distributed system. Petri nets have been used in the modelling of distributed algorithms [9]. The analysis techniques of Petri nets can be applied to these models to verify the correctness of algorithms. This is the motivating factor in our approach. We model the proposed distributed garbage collection algorithm in Petri nets and apply some proof techniques to verify some properties of the algorithm. The Petri nets help to better understand and debug the algorithm.

In Section 6.2 we describe the application of Petri net theory to modelling and analysis of systems. The definitions for the basic Petri net are presented in the Section 6.2.1. In order to enhance the modelling and decision power of Petri nets, many extensions and modifications have been suggested to the basic Petri net model. These enhancements are discussed in the Section 6.2.2. Once the system under investigation has been modelled as a Petri net, it is to be analyzed. The analysis problems of Petri nets are presented in Section 6.3.1. There are two well known analysis techniques namely *marked graphs* and the *linear algebraic method*. Section 6.3.2 describes these techniques. The linear algebraic representation using matrices is presented in Section 6.3.3. The concept of S and T invariants is introduced in Section 6.3.4. Section 6.5 describes the Petri net model of our algorithm and a proof based on the *S-invariants*.

6.2 Petri Net Theory and Modelling

Petri nets are abstract, formal models of information and control flow in systems exhibiting concurrency and asynchronous behaviour [41, 42, 47]. Petri net theory allows a system to be modelled by a Petri net, a mathematical representation of the system. The components of a system may exhibit *concurrency* or *parallelism*. Activities of one component may occur simultaneously with activities of other components. Petri nets are suited specifically to model systems with interacting concurrent components. Petri nets have been used in the modelling of systems in diverse fields such as computer hardware, computer software, operations research, chemical systems, biological systems, semantics for natural language representation, communication protocols, economics, political systems and social and legal systems.

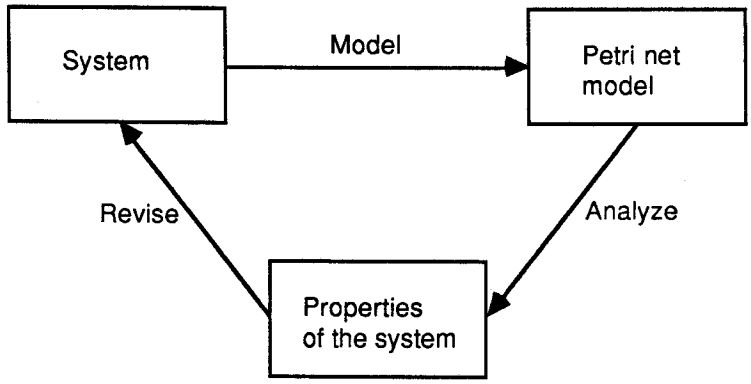


Figure 6.1 Use of Petri nets for the modelling and analysis of systems.

The application of Petri nets to the design and analysis of systems involves two basic steps: first the system under investigation is modelled as a Petri net, then this model is analyzed. The analysis of the Petri net may reveal important information about the structure and dynamic behaviour of the modelled system. Any problems encountered in the analysis points to flaws in the design. The design is modified to correct the flaws. The modified

design is then modelled and analyzed again. The modelling-analysis loop is repeated until the system meets the design specifications. The design cycle using Petri nets is shown in the Fig. 6.1.

6.2.1 Definition of Petri Nets

A Petri net may be defined as a bipartite, directed graph $N = (P,T,A)$ [2], where,

P	=	$\{p_1, p_2, \ldots p_n\}$	a set of *places*,
T	=	$\{t_1, t_2, \ldots t_m\}$	a set of *transitions*,
A	$\subseteq$	$\{P \times T\} \cup \{T \times P\}$,	a set of directed *arcs*,

A *marking*, M of a Petri net is a mapping from the set of places P to natural numbers N.

$M : P \rightarrow N$, where $M(p_i) = m_i$ for $1 \leq i \leq n$

M assigns *tokens* to each place in the net. A Petri net $N = (P, T, A)$ with marking M is called a *marked Petri net* $PN = (P, T, A, M)$. Marking is also known as the *state* of Petri net. The marking M can also be defined as an n-vector $M = (m_1, m_2, \ldots m_n)$, where $n = |P|$ and each $m_i \in N$, $i = 1,\ldots n$. The vector M gives for each place p_i in a Petri net the number of tokens in that place. The number of tokens in place p_i is m_i, $i = 1,\ldots,n$.

Pictorially, in a Petri net graph, places are represented by circles, transitions as bars, and tokens as small dots inside the circles (•). Fig. 6.2 illustrates an example of a Petri net.

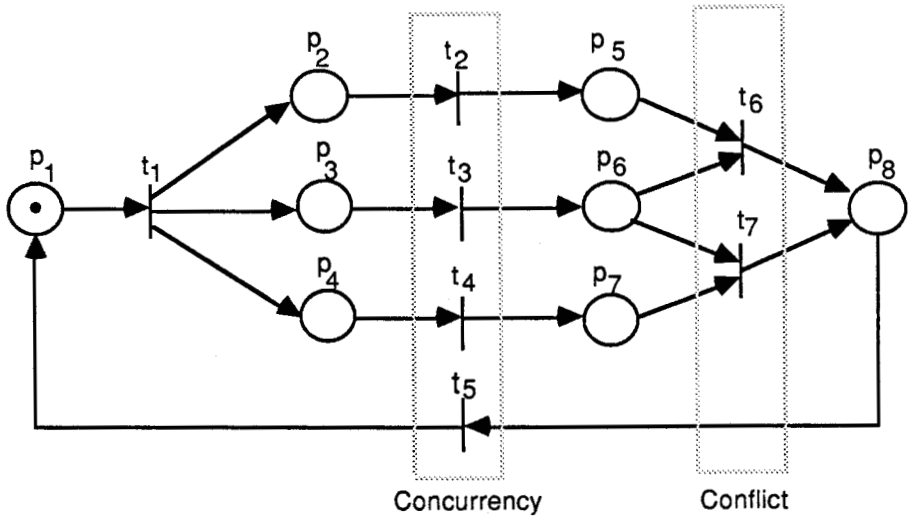


Figure 6.2. An example of a Petri net.

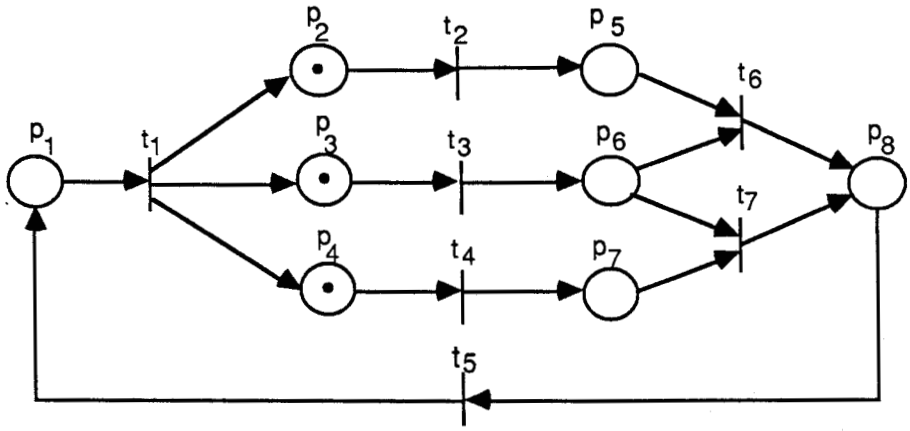


Figure 6.3 Network after the firing of transition t_1

Input and output functions relate *transitions* and *places*. The input function I is a mapping from a transition t_j to a collection of places $I(t_j)$, known as the input places of the transition. The output function O maps a transition t_j to a collection of places $O(t_j)$ known as the output places of the transition.

$$I(t_j) = \{p_i \mid (p_i,t_j) \in A\}$$

$$O(t_j) = \{p_i \mid (t_j,p_i) \in A\}$$

The execution of a Petri net is controlled by the number and distribution of tokens. Tokens reside in places and control the execution of transitions of the net. A Petri net executes by *firing* transitions. A transition is enabled for firing if each of its input places has at least as many tokens in it as arcs from the place to the transition. For example in the Petri net shown in Fig. 6.2, transition t_1 is enabled whenever there is one or more tokens in the place p_1. A transition fires by removing an enabling token from each of its input places and depositing one token into each of its output places. For instance in Fig. 6.2 the firing of transition t_1 removes a token from p_1 and puts one token into places p_2, p_3 and p_4. Firing a transition changes the marking M of the Petri net to a new marking M'.

In the above example, once the transition t_1 fires, the transitions t_2, t_3 and t_4 are enabled, and can fire *concurrently*. As these transitions complete their firings, places p_5, p_6 and p_7 receive a token each. Now both transition t_6 and t_7 are enabled, but the firing of one disables the other, for example if t_6 fires t_7 can not fire. This represents a *conflict* between two transitions. This ability to model both *concurrency* and *conflict* makes Petri nets a powerful modelling tool.

6.2.2 Restrictions, Extensions and Modifications of Petri nets

Extensions and modifications have been made to the basic Petri net model to overcome the two limitations, namely limitations on *modelling power* and *decision power*.

The fundamental extension to Petri net is to allow *zero testing* using *inhibitor arcs*. A Petri net with inhibitor arcs is illustrated in Fig. 6.4. An inhibitor arc from a place p_i to a transition t_j has a small circle rather than an arrowhead at the transition. The transition t_j can fire only if p_i is empty [1, 2]. It has been shown that a Petri net with inhibitor arcs has the modelling power of a Turing machine [1, 2].

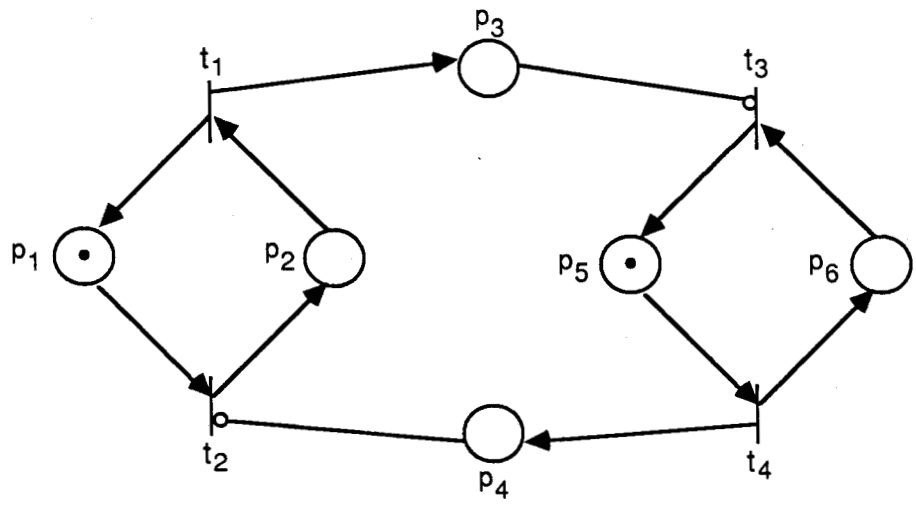


Figure 6.4 An extended Petri net with inhibitor arcs.

A Petri net is called an *ordinary Petri net* if no transition has more than one input arc from a place and more than one output arc to a place. In a *self-loop free Petri net* no place is allowed to be both input and output of a transition i.e. $I(t_i) \cap O(t_i) = \varnothing$.

Petri nets having *exclusive-OR transitions* [38, 39] and *switches* [6, 7] have been suggested. An *exclusive-OR transition* t_j with input $I(t_j)$ requires that one and only one of its inputs be nonzero to enable the transition. A *switch* is a special transition with a special input called the *switch input* and exactly two outputs, one labeled *e* for *empty* and another labeled *f* for *full*. A switch transition fires when it is enabled, ignoring the state of the switch input. When it fires, a token is put in the output labeled *e* if the switch input is empty, otherwise if the switch input has a token then a token is delivered to the *f* output. Thus, the firing of a switch transition results in only one of the two markings.

Probabilistic arcs from a transition to a set of output places $O(t_j)$ deposit a token in one and only one of the output places. The choice of which place receives the token is determined by the probability labeled on each arc [20]

A *counter arc* from a place to a transition is labeled with an integer value *k*. The

firing rule with a counter arc is changed such that a transition is enabled when tokens are present in its normal input places, and at least k tokens are present at the counter input place. When the transition fires, one token is removed from each of the normal input places and k tokens from the counter input place [20].

Another major extension of Petri nets is the association of *time*, either constant and probabilistic with transitions. In *timed Petri nets* a fixed firing time is associated with each transition [45, 46]. To overcome the fixed-time constraint in *timed Petri nets*, *Stochastic Petri nets* (*SPN*) were introduced [34, 35, 37]. In *SPN*s an exponential firing time distribution is associated with each transition of the Petri net.

6.3 Analysis of Petri Nets

Once a Petri net model is developed, it is to be analyzed to verify the properties of the system based on the model. This requires the understanding of various problems (or properties) of Petri nets. In the first section a brief mention of the various problems that need to be solved for Petri nets are presented. The second part concentrates on the analysis techniques to answer questions related to Petri net problems.

6.3.1 Analysis Problems of Petri Nets

Safeness : A place in a Petri net is *safe* if the number of tokens in that place never exceeds one. A Petri net is *safe* if all places in the net are *safe* [42].

Boundedness : A Petri net is said to be *k-bounded* if, for all possible markings, the maximum number of tokens in all places is less than or equal to k.

Conservativeness : A Petri net is said to be *conservative* if the total number of tokens in the net remains constant under all possible markings i.e. $\sum_{p_i \in P} M'(p_i) = \sum_{p_i \in P} M(p_i)$, M is an initial marking and M' is all possible markings from M. For *conservation*, the number of inputs to each transition must equal the number of outputs $|I(t_j)| = |O(t_j)|$.

Liveness : A *deadlock* in a Petri net is a transition (or a set of transitions) which can not fire. A transition is *live* if it is not deadlocked. A Petri net is *live* if every transition is *live*.

Reachability : The *reachability problem* is to find in a given Petri net with a marking M and a marking M', if M' is reachable from M by some sequence of transition firings.

Coverability : The *coverability problem* is : given a Petri net with initial marking M and a marking M', is there a reachable marking $M'' \in R(M)$, ($R(M)$ is the set of all reachable markings) such that M'' covers M i.e., M' is reachable from M and M'' is reachable from M'.

6.3.2 Analysis Techniques

Two major Petri net analysis techniques have been suggested. They are *reachability tree* and the *linear algebraic technique* involving matrix equations.

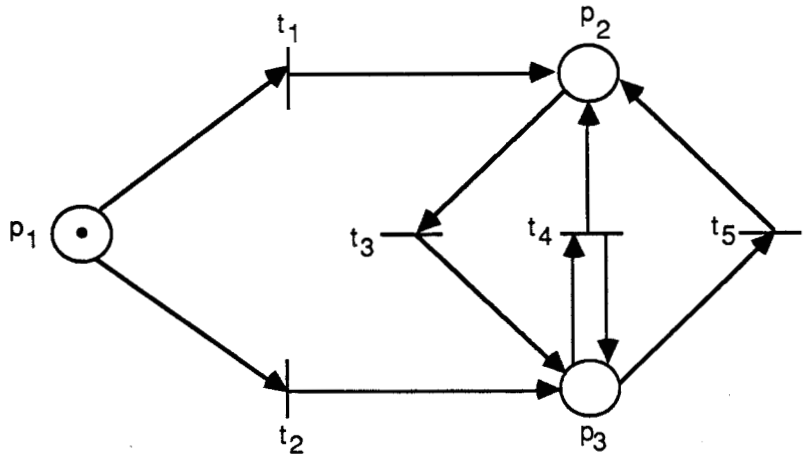


Figure 6.5 A marked Petri net for illustrating the construction of a reachability tree.

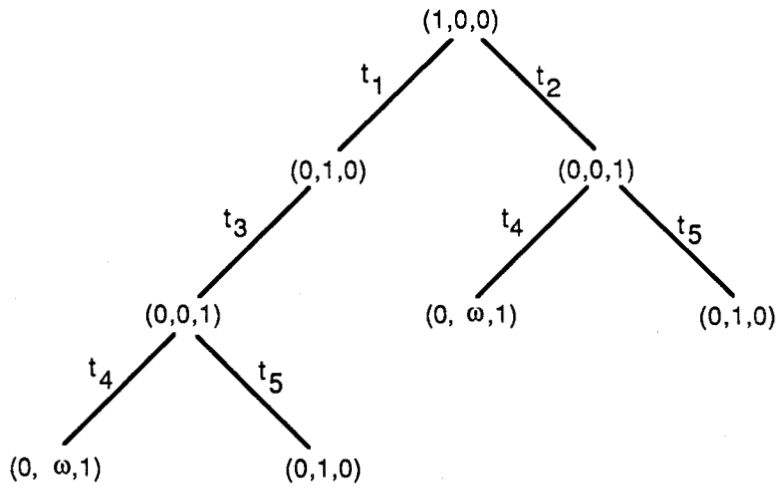


Figure 6.6 The reachability tree of the Petri net shown in Figure 6.5.

The *reachability tree* represents the *reachability set* of a Petri net. Consider the marked net of Fig. 6.5. The initial marking is (1,0,0) since there is a token in place p_1 and zero tokens in p_2 and p_3. In the initial marking two transitions are enabled: t_1 and t_2. We define two nodes in the reachability tree for the (reachable) markings which result from firing both transitions. An arc leads from the initial marking to each of the new

markings and is labelled by the transition fired. From the marking (0,1,0) (a token present in p_2 and no tokens in p_1 and p_3) only transition t_3 can fire. This leads to the marking (0,0,1). A token in p_3 enables two transitions : t_4 and t_5. For transition t_4 , p_3 is both an input place as well as an output place. The enabling of this transition leaves a token in p_2 and p_3. The transition t_4 may fire continuously resulting in infinite number of tokens in place p_2. This condition is represented by ω in the graph. If instead of transition t_4, t_5 were to fire we get a new marking (0,1,0). Thus a Petri net with an infinite reachability set is represented by a finite number of nodes in the reachability tree. The reachability tree of the Petri Net in Fig. 6.5 is illustrated in Fig. 6.6.

The reachability tree can be used to solve the *safeness*, *boundedness*, *conservation* and *coverability* problems.

A Petri net is *not safe* if the symbol ω appears in the reachability tree. The ω symbol in the reachability tree indicates the places that are unbounded. A Petri net is *bounded* if and only if the symbol ω never appears in its reachability tree.

A Petri net is *conservative* if it does not lose or gain tokens but merely moves them around. There is no one-to-one mapping between tokens and resources. Some tokens may represent several resources with one token. This condition arises due to the firing of transitions with more outputs than inputs. In general, we may define a *weighting* of tokens. Tokens in each place are assigned some weight. A *weighting vector* $w = (w_1, w_2, \ldots w_n)$ defines a weight w_i for each place $p_i \in P$. For conservativeness the weighted sum for all reachable markings should be constant. If any marking with nonzero weight is ω , the net is not conservative.

The *coverability problem* can be solved with the aid of the reachability tree.

Given an initial marking M, we construct the reachability tree. Then we search for any node x such than marking of x is greater than or equal to m' (*i.e.* $m[x] \geq m'$). If no node is found, the marking m' is not covered by any reachable marking; if such a node is found, $m[x]$ gives a reachable marking which covers m' .

In general the reachability tree cannot be used to solve the *reachability* or *liveness* problems or to define or determine which firing sequences are possible. Solutions to these problems are limited by the existence of ω symbol.

6.3.3 Linear Algebraic Representation

Reisig [47] presents a formal definition of a generalized Petri net with each place having a finite (or even infinite) capacity, and each arc having a weight (*counter arcs*) associated with it. The places in his notation are denoted by S and transitions by T, and F denotes the arcs between the places and transitions. A Petri net is defined as a 6 tuple $N = (S, T; F, K, M, W)$ where,

i) $(S, T; F)$ is a finite net, the elements of S and T are called places and transitions, respectively,

ii) $K : S \rightarrow N \cup \{\omega\}$, gives a (possibly unlimited) capacity for each place, (ω denotes infinite capacity of a place).

iii) $W : F \rightarrow N \setminus \{0\}$, attaches a weight to each arc of the net.

iv) $M : S \rightarrow N \cup \{\omega\}$ is the initial marking, respecting the capacities, i.e. $M(s) \leq K(s)$ for all $s \in S$.

The components of a net N are denoted by $S_N, T_N, F_N, K_N, W_N, M_N$ respectively.

A Petri net can also be represented using matrices. The matrix representation permits use of linear algebraic techniques for the analysis of Petri net problems. The linear algebraic representation of a Petri net is defined as:

i) For transitions $t \in T$, let the vector $\underline{t}: S \rightarrow Z$ be defined as

$$\underline{t}(s) = \left\{ \begin{array}{l} W(t,s) \; iff \; s \in O(t) \wedge s \notin I(t), \\ -W(t,s) \; iff \; s \in I(t) \wedge s \notin O(t), \\ W(t,s) - W(s,t) \; iff \; s \in I(t) \cap O(t), \\ 0 \; otherwise \end{array} \right\}$$

ii) Let the matrix $\underline{N}$: $S \times T \rightarrow Z$ be defined as $\underline{N}(s,t) = \underline{t}(s)$.

$W(t,s)$ is the weight of arc from a transition t to a place s. The component of vector $\underline{t}$ corresponding to a place s is equal to the weight of the arc between a transition t and s, if s is a member of the output function $O(t)$ but not a member of the input function $I(t)$. It is equal to the negation of the weight of the arc, if s is a member of the input function $I(t)$ but not a member of the output function $O(t)$. If s is a member of both input and output functions then it is equal to the difference between the weights of output and input arcs.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	M_N
s_1	−1	0	0	0	1	0	0	1
s_2	1	−1	0	0	0	0	0	0
s_3	1	0	−1	0	0	0	0	0
s_4	1	0	0	−1	0	0	0	0
s_5	0	1	0	0	0	−1	0	0
s_6	0	0	1	0	0	−1	−1	0
s_7	0	0	0	1	0	0	−1	0
s_8	0	0	0	0	−1	1	1	0

Figure 6.7 Matrix Representation of Petri Net shown in Figure 6.2.

Based on the above definition, the matrix representation of the Petri net in Fig. 6.2 is illustrated in Fig. 6.7. M_N denotes the initial marking of the net. It has the count of tokens present in each of the places initially. In the Petri net shown in Fig. 6.2, initially

one token is present in the place s_1, and no tokens in other places.

6.3.4 Net Invariants

This section briefly describes the concept of net invariants. First we consider sets of places whose token count do not change during transition firings. Knowledge about any such set of places helps in analyzing liveness and also allows us to investigate other properties of systems. Such sets of places are called *S-invariants*.

The derivation of S-invariants from the Petri net matrix $\underline{N}$ is given below. This derivation allows us to solve $\underline{N}$ to determine all the invariants of a given net. Let N be a Petri net with places denoted by S. We want to characterize sets of places, $S \subseteq S_N$ of N which do not change their joint token count when transitions fire. If the token count on $S \subseteq S_N$ does not change when a transition $t \in T_N$ fires then,

$$\sum_{s \in I(t) \cap S} W(s,t) = \sum_{s \in O(t) \cap S} W(s,t)$$

This condition is equivalent to,

$$\sum_{s \in I(t) \cap S} \underline{t}(S) = -\sum_{s \in O(t) \cap S} \underline{t}(S) \Rightarrow \sum_{s \in I(t) \cap S} \underline{t}(S) + \sum_{s \in O(t) \cap S} \underline{t}(S)$$

$$\Rightarrow \sum_{s \in S} \underline{t}(S) = 0$$

If we replace S by its characteristic vector c_S the condition becomes,

$$\sum_{s \in S_N} \underline{t}(S).c_s(s) = 0 \quad \text{or by vector multiplication} \quad \underline{t}.c_s = 0.$$

If the token count on $S \subseteq S_N$ never changes under arbitrary transition firings, the condition $\underline{t_i}.c_s = 0$ must be fulfilled for all transitions $t_i \in T_N$, hence

$\underline{N}'.\ c_s = 0$ ($\underline{N}'$ denotes the transpose of the matrix $\underline{N}$) must hold. Conversely, each solution c of $\underline{N}'.\ x = 0$ consisting of components from {0,1} is a characteristic vector of a set of places with a constant token count. So such sets are found by solving $\underline{N}'.\ x = 0$. For the Petri net shown in Fig. 6.2, we have one invariant i_1 that is shown in Fig. 6.8.

It can be shown that every Petri net which is *finite*, *live* and *bounded* is covered by S-invariants [47, 32]. Thus the verification of finiteness, liveness and boundedness problems of a Petri net reduces to the determination of all invariants of the net.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	i_1	M_N
s_1	−1	0	0	0	1	0	0	1	1
s_2	1	−1	0	0	0	0	0	0	0
s_3	1	0	−1	0	0	0	0	1	0
s_4	1	0	0	−1	0	0	0	0	0
s_5	0	1	0	0	0	−1	0	0	0
s_6	0	0	1	0	0	−1	−1	1	0
s_7	0	0	0	1	0	0	−1	0	0
s_8	0	0	0	0	−1	1	1	1	0

Figure 6.8 Illustration of S-invariants of Petri net shown in Fig. 6.2.

Similar to S-invariants, *T-invariants* are defined. Here we consider the sets of transitions ($v : T_N \rightarrow N$). The T-invariants are determined by the solutions of equations of the form $\underline{N}.\ x = 0$. A T-invariant indicates that if it is possible, starting from some marking M, to fire each transition t exactly $v(t)$ times, to again yield the marking M. It can be proved that every P/T-net which is finite, live and bounded is covered by T-invariants [47].

6.4 Modelling of the Algorithm

In this section the stepwise development of the Petri net model of the algorithm is presented. The development of this model is based on the examples given by Reisig [47]. First the algorithm is represented as a net with inscriptions in English. Then it is refined so that its structure corresponds to a Petri net and its behaviour to the firing rule.

The many processors are assumed to be in either of two states: mutation or local garbage collection (*lgc*). In mutation, graph operations (such as making and breaking links) are performed. Mutation is intermittently interrupted by incoming messages from other processors. These messages are processed and reply messages are sent. In addition for each newly created remote link a newLinkPaint message is sent. In the local garbage collection phase refreshLinkPaint messages are sent for every active remote edge. In addition to processing messages during mutation and *lgc*, processors communicate with the master processor at the time of starting and completion of an *lgc*. Master processor sends phaseChange messages to all processors at the time of system phase change.

The first step in the development of a Petri net model is the construction of the model as an inscribed net. The inscriptions in the transitions denote instructions, which are executed when the transition fires. The conditions written into places have to be fulfilled to allow the associated transitions to fire. The instructions on one transition form an *atomic* operation. This means that during the execution of the instructions of some transition, the entities involved may not be changed by the firing of other transitions. The inscribed net represents a high level model of the system. This model is refined in a step-by-step fashion to arrive at the final Petri net model. Fig. 6.9 shows the algorithm as an inscribed net. Messages from processors enter the system through the input place. Each message contains a processor identification and the type of message – it may be considered

a labelled token. A token in the place *INL* indicates that the processor is in the *lgc* state and a token in *INM* indicates that it is in the mutation phase. $\overline{INM}$ and $\overline{INL}$ are complements of *INM* and *INL* respectively. Introduction of complement $\bar{p}$ of a place p serves to test emptiness of p. The outstanding message lists are organized in a *first-in-first-out* principle, *first(M)* denotes the first element of M. An instruction $i \rightarrow M$ adds i to the end of the list M, *skip(i,M)* deletes i from the list M. Messages are sent out of the system through the output place.

The graph operations taking place in the mutation phase are internal operations and do not account for a change in the token count of the system (except when a remote link is made). Hence they are not explicitly shown in the model. Similarly the *mark* and sweep of the *lgc* state are not shown. The model concentrates on message operations and the changing of system phases. The number of outstanding messages of each message type is maintained. On receiving an acknowledgement message the corresponding count is decremented. In order to simplify the modelling of the algorithm we assume that the system sends a certain fixed number of messages during each state. The state of the system changes as soon as that many messages have been sent. In the Fig. 6.9 x and y keep track of the number of mutation and *lgc* messages sent. The system changes from mutation to *lgc* state as soon as m mutation messages have been sent. Similarly the system changes from *lgc* to mutation state as soon as n *lgc* messages have been sent. M and N are the outstanding message lists. M is the list of outstanding mutation messages and N is the list of outstanding lgc messages. On receiving an acknowledgement for a mutation message, the count M is decremented. Similarly N is decremented on receiving an acknowledgement for an *lgc* message.

For considerations of *liveness* and *boundedness*, the dependencies between x, y, m and n are crucial. Assuming that the transitions are invoked on satisfying the required conditions the network shown in the Fig. 6.9 can be simplified as shown in Fig.

6.10. This net is translated into a Petri net with complementary places for p and q. The self-loops in the system are decomposed and the simplified Petri net model is shown in Fig. 6.11. The matrix representation of this network is shown in Fig. 6.12.

Well known linear algebraic techniques can be used to determine all the minimal support invariants (invariants having non-zero values) of a net. If the net is large, this may require enormous CPU time [31]. We developed a new Prolog algorithm that generates all the minimum support invariants of any generalized Petri net represented in the matrix form. The main advantage of this algorithm is that the determination of invariants of any Petri net is fully automated. This in turn simplifies the design cycle using Petri nets. Using this algorithm we obtained all the S-invariants of the net. The Fig. 6.13 illustrates the S-invariants of the network. The properties of the system can be proved using these invariants.

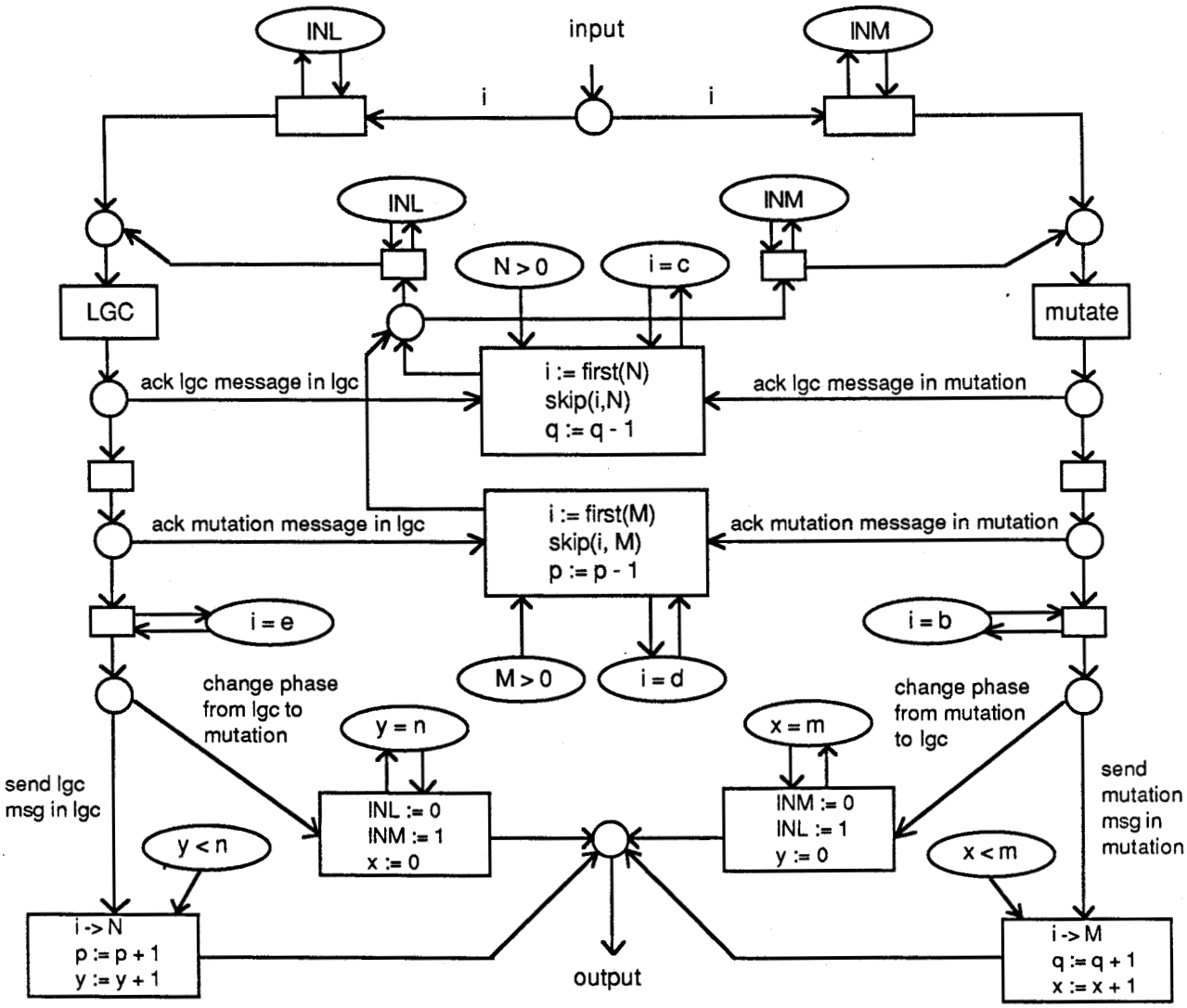


Figure 6.9 High level Petri Net Model of the Algorithm.

INL	–	*local garbage collection* (*lgc*) state.
INM	–	*mutation* state.
N	–	outstanding *lgc* messages list.
M	–	outstanding mutation messages list.
n	–	capacity of *lgc* message list.
m	–	capacity of mutation message list.
q	–	number of messages in the *lgc* message list
p	–	number of messages in the mutation message list.

y – total number of *lgc* messages sent so far in the current phase.
x – total number of mutation messages sent so far in the current phase.
i – incoming message token (acknowledgement or reply).
c – receiving an *lgc* message acknowledgement.
d – receiving a mutation message.
b – sending a mutation message.
e – sending a *lgc* message.

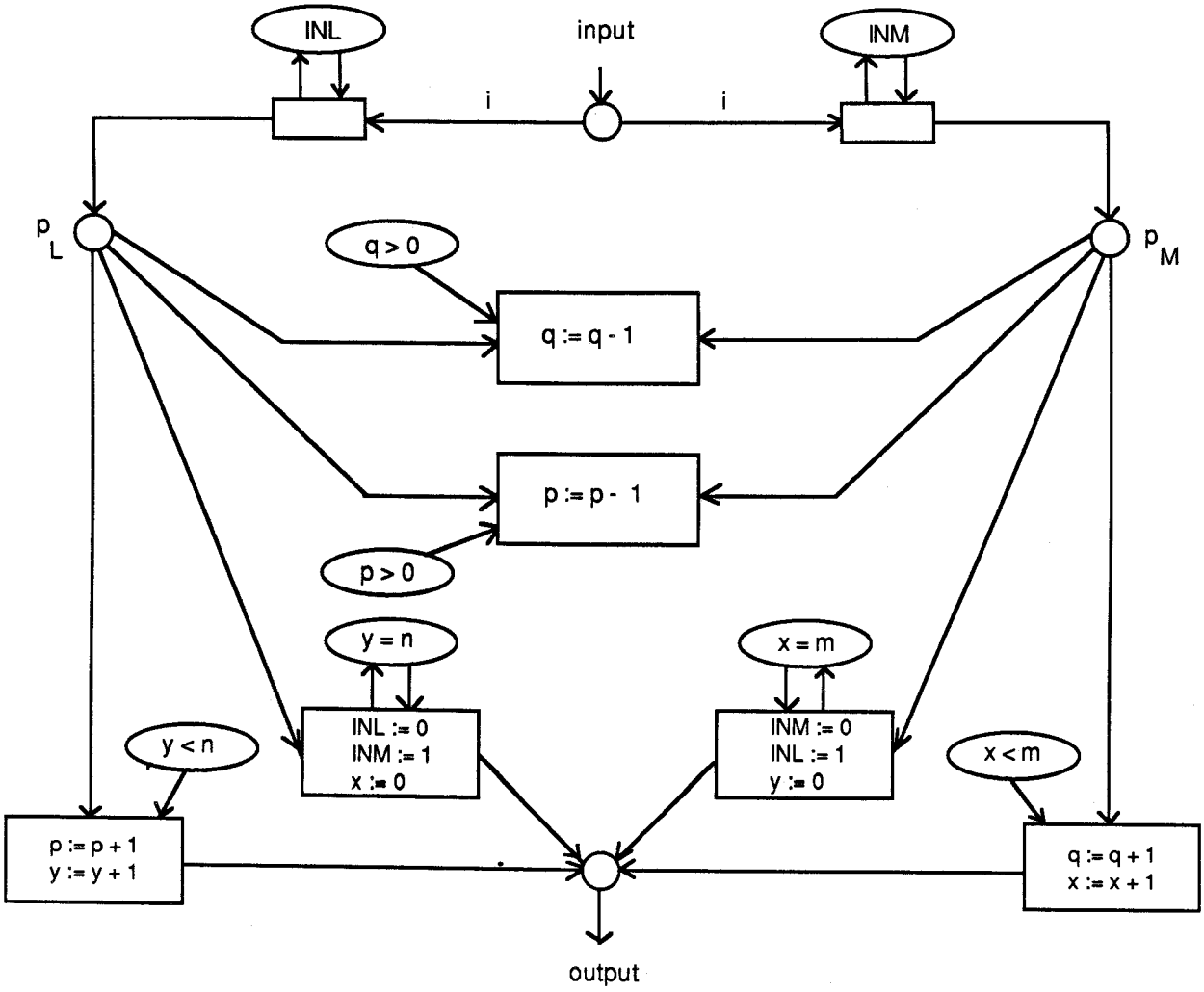


Figure 6.10 Simplified Petri Net Model.

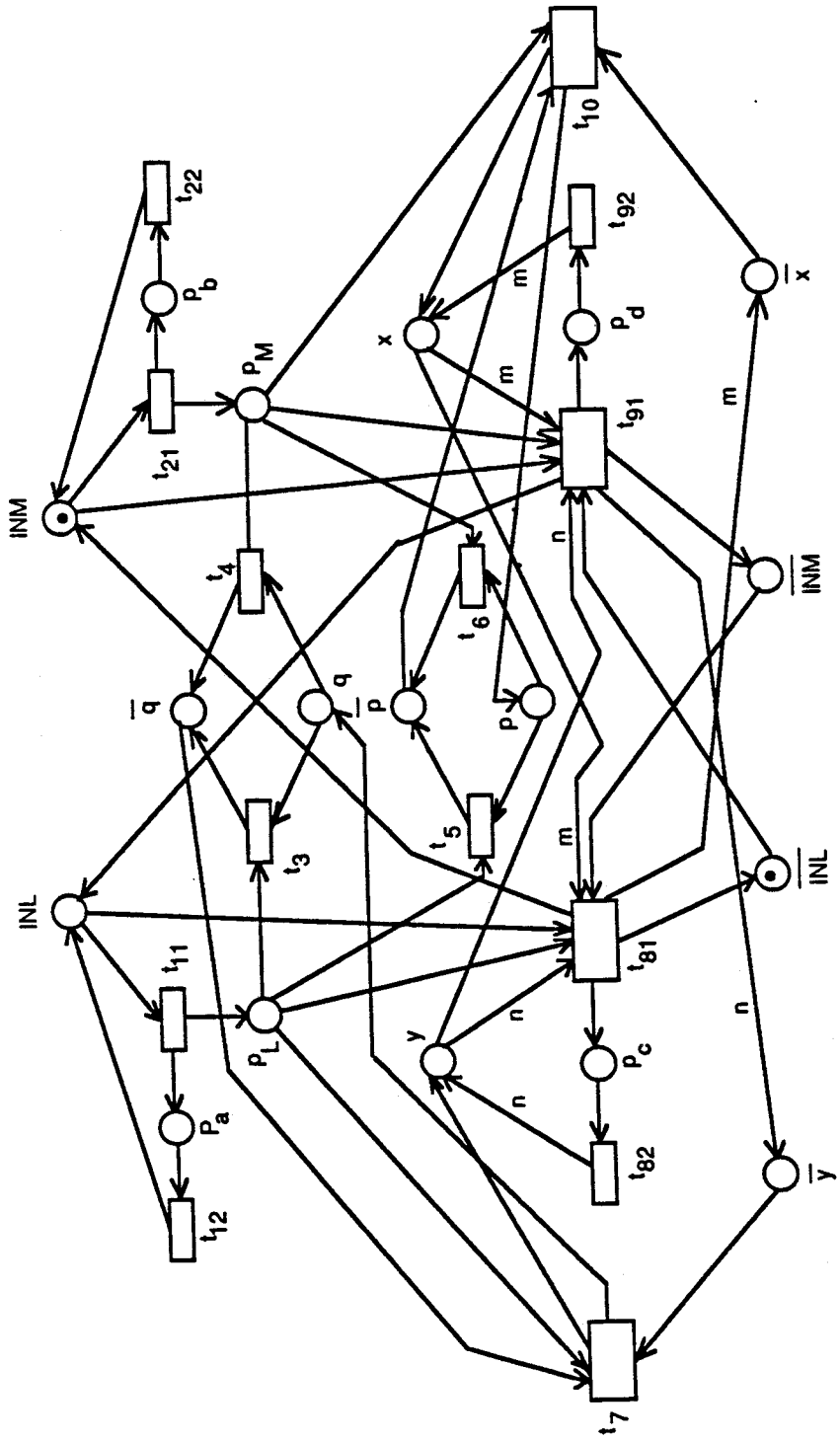


Figure 6.11 Petri Net Model of the Algorithm.

places \ Transitions	t_{11}	t_{12}	t_{21}	t_{22}	t_3	t_4	t_5	t_6	t_7	t_{81}	t_{82}	t_{91}	t_{92}	t_{10}
INL	−1	1	0	0	0	0	0	0	0	−1	0	1	0	0
$\overline{\text{INL}}$	0	0	0	0	0	0	0	0	0	1	0	−1	0	0
INM	0	0	−1	1	0	0	0	0	0	1	0	−1	0	0
$\overline{\text{INM}}$	0	0	0	0	0	0	0	0	0	−1	0	1	0	0
p_a	1	−1	0	0	0	0	0	0	0	0	0	0	0	0
p_b	0	0	1	−1	0	0	0	0	0	0	0	0	0	0
p_L	1	0	0	0	−1	0	−1	0	−1	−1	0	0	0	0
p_M	0	0	1	0	0	−1	0	−1	0	0	0	−1	0	−1
q	0	0	0	0	−1	−1	0	0	1	0	0	0	0	0
$\overline{q}$	0	0	0	0	1	1	0	0	−1	0	0	0	0	0
p	0	0	0	0	0	0	−1	−1	0	0	0	0	0	1
$\overline{p}$	0	0	0	0	0	0	1	1	0	0	0	0	0	−1
y	0	0	0	0	0	0	0	0	1	−n	n	−n	0	0
$\overline{y}$	0	0	0	0	0	0	0	0	−1	0	0	n	0	0
x	0	0	0	0	0	0	0	0	0	−m	0	−m	m	1
$\overline{x}$	0	0	0	0	0	0	0	0	0	m	0	0	0	−1
p_c	0	0	0	0	0	0	0	0	0	1	−1	0	0	0
p_d	0	0	0	0	0	0	0	0	0	0	0	1	−1	0

Figure 6.12 Matrix Representation of the Petri net model of the Algorithm.

places	M_N	S-invariants i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8
INL	0	1	1	0	0	0	0	0	0
$\overline{\text{INL}}$	1	0	1	0	0	0	0	1	0
INM	1	1	0	1	0	0	0	0	0
$\overline{\text{INM}}$	0	0	0	1	0	0	0	1	0
p_a	0	1	1	0	0	0	0	0	0
p_b	0	1	0	1	0	0	0	0	0
p_L	0	0	0	0	0	0	0	0	0
p_M	0	0	0	0	0	0	0	0	0
q	0	0	0	0	1	0	0	0	0
$\overline{q}$	n	0	0	0	1	0	0	0	0
p	0	0	0	0	0	1	0	0	0
$\overline{p}$	m	0	0	0	0	1	0	0	0
y	n	0	0	0	0	0	n	1	0
$\overline{y}$	0	0	0	0	0	0	n	1	0
x	0	0	0	0	0	0	0	0	1
$\overline{x}$	m	0	0	0	0	0	0	0	1
p_c	0	0	0	0	0	0	m	n	0
p_d	0	0	0	0	0	0	0	0	m

Figure 6.13 S-Invariants of the Petri Net Model.

6.4.1 Verification of Properties from the Model

As mentioned earlier any system that is finite, bounded and live must have positive invariants. From the S-invariants obtained from the model we can conclude that the model is bound, finite and live. Some of the properties of the system can be verified from these invariants:

i) Using the invariant i_2,

$$M(INL) + M(INM) + M(p_a) + M(p_b) =$$
$$M_N(INL) + M_N(INM) + M_N(p_a) + M_N(p_b) = 1$$

From this invariant it is seen that the total token count in all these four places always remains to be 1. This implies that processor either remains in mutation or *lgc* state.

ii) From the invariant i_4,

$$M(q) + M(\bar{q}) = M_N(q) + M_N(\bar{q}) = n$$

The total number of messages sent in *lgc* phase is constant and is equal to n.

iii) From i_5,

$$M(p) + M(\bar{p}) = M_N(p) + M_N(\bar{p}) = m$$

The total number of messages sent in the mutation phase is constant and is equal to m.

6.5 Conclusions

In this chapter we presented the application of Petri nets to the modelling and analysis of systems. The modelling and decision power of the Petri nets has been enhanced by extensions and modifications to the basic Petri net model. The *zero testing* which tests for the emptiness of a place is one of the most significant extensions. These modifications suggested have been motivated by the need to model diverse systems using Petri nets.

The flexibility and usefulness of analysis techniques is an equally important criterion for a modelling tool. In the present case, we have concentrated on using the existing analysis tools for the modelling and verification of properties. From the *reachability tree* we can solve the *safeness*, *boundedness*, *conservation* and *coverability* problems of a net. But reachability tree has the limitation that it can not be used to solve the *reachability* or *liveness* problems.

Linear algebra provides the other well known analysis technique. This technique promises to be interesting as the tools available in solving linear equations may be applied to the analysis of Petri nets. S-invariants of a net correspond to the sets of places whose joint token count do not change on transition firings. The properties of systems such as *liveness* and *boundedness* can be verified from these invariants. A *live* and *bounded* net must have positive invariants.

In this chapter we presented the Petri net modelling of the algorithm. The stepwise development of the model is shown. Some of the properties of the system have been proved using the S-invariants. The invariants obtained show that the model is sound. It is bounded, finite and live. Inability of the approach in modelling graph mutations and garbage collection operations limited the scope of the model. This also prevented us from completely proving the correctness of the algorithm. Inspite of this weakness the work enhanced our confidence in the algorithm.

CHAPTER 7
Summary and Conclusions

The main objective of this thesis was to develop an algorithm for garbage collection in a distributed multicomputer system. This chapter summarizes the work performed in this thesis and discusses its contributions. Conclusions and directions for future research are given.

A key feature of functional and declarative languages is their inherent parallelism. Functional languages seem especially suited to parallel architectures because they lack side effects. These languages have dynamic storage allocation. Evaluation of a program generates garbage cells. The performance of any functional language depends on the efficiency of its underlying garbage collection scheme. This strong dependency on garbage collection has motivated this study of distributed garbage collection algorithms.

In Chapter 1, the general motivation for the study of storage reclamation schemes was given. The techniques of implementing a functional language were presented. The concept of graph reduction as a way of implementing functional languages was introduced. The model of parallel graph reduction which we assumed for the development of our algorithm was presented.

Chapter 2 and 3 presented a review of garbage collection algorithms for uniprocessor and multicomputer systems. The distributed garbage collection algorithms are modifications of *mark and sweep* and *reference counting* algorithms. These vary from completely global to fully distributed real-time algorithms. Global algorithms are simple to implement but restrict parallelism as well as waste much computational power. The non-global algorithms are expensive to implement but they provide greater flexibility at

lower cost of computational power. Algorithms proposed by Hudak, Ali and Hughes were discussed. A comparison of these algorithms was presented.

Chapter 4 presented our algorithm. The first section of this chapter described the algorithm. Data structures and the low level algorithms were discussed. The second section presented an analysis of distributed garbage collection algorithms. Analytical models of a global algorithm and the proposed algorithm were presented.

Chapter 5 presented a simulator model of the algorithm. The design and implementation of the simulator model was discussed. The simulation experiments were designed to observe phase transitions, the effect of inter-connection network delay on the cost of the algorithm and the number of messages sent for different phase lengths. The results and analysis of these experiments were presented.

Chapter 6 discussed the Petri net modelling of the algorithm. The first section of this chapter discussed the Petri net theory and modelling. The analysis techniques of Petri nets, marked tree and invariants were introduced. The step-wise development of the Petri net model was presented. The scope of the model was limited as graph operations and garbage collection operations could not be modelled in Petri nets. Some properties of the algorithm were verified using S-invariants.

7.1 Contributions of the Thesis

We have presented a new algorithm for memory reclamation in a parallel distributed system that is superior to the existing solutions. This algorithm is characterized by independent local garbage collections to reclaim local garbage and four cyclic colour phases for reclaiming global garbage cells. The idea of independent *lgc's* is an adaptation of ideas proposed by Ali [3, 4] in his *local-global* and *distributed-local* algorithms. As in the case of Ali's *global* algorithms any variation of mark and sweep algorithm can be used for a

local garbage collection. Our algorithm introduces a new idea of global colour phases. A master algorithm co-ordinates the global phase changes. In the Hughes algorithm [29] the time required for the reclamation of global garbage can be arbitrarily long and it requires the co-operation of all the processors. In the proposed scheme the reclamation of global garbage is dependent only on the rate of phase changes and it is guaranteed that all garbage cells including distributed cyclic structures are reclaimed in two phases. This schme has much less message and space requirements compared to both Ali's and Hughes algorithms.

An analytical model of the algorithm was developed. The analysis of this model shows that this algorithm is superior to a global algorithm. A simulator model of a parallel graph reduction system that incorporates the proposed garbage collection algorithm was implemented. Simulation experiments verified the working of the algorithm. Experiments conducted on systems of different sizes and load distributions show that the algorithm is sound. The experimental results show that the algorithm is flexible, it permits the tuning of various system parameters like the phase length to suit the underlying architecture and the application program.

Petri net modelling provided insight into the modelling of the proposed algorithm. Also the modelling helped in debugging the algorithm. Some of the properties of the algorithm were verified from this model using S-invariants. A new algorithm was developed to find the invariants of a Petri net. This Prolog algorithm automates the determination of invariants of any Petri net represented in a matrix form. This algorithm simplifies the Petri net design as the analysis becomes simpler and less time consuming.

7.2 Extensions and Future Work

The work performed in this thesis has much scope for extensions. The simulation experiments presented in this thesis are limited due to large computational time required for

each run. These experiments can be repeated for extended ranges of parameters – the length of a phase, number of processors in the system and different probability distributions of graph operations. Simulations may be conducted using practical loads based on programs written in parallel functional languages, these results can then be used to develop better synthetic loads to test distributed algorithms of this nature. The Petri net modelling can be extended by developing models concentrating on a specific aspect of the algorithm such as graph mutations and local garbage collections. These sub-modules may then be merged to model the overall algorithm. The proof techniques used in this thesis may then be used to prove the overall correctness of the algorithm.

References

[1] Agerwala, T. "A Complete Model for Representing the Coordination of Asynchronous Processes", Hopkins Computer Research Report No. 32, Computer Science Program, John Hopkins University, July 1974.

[2] Agerwala, T. "Putting Petri Nets to Work", *Computer*, December 1979, pp. 85-94.

[3] Ali, K.A.M. "Object-Oriented Storage Management and Garbage Collection in Distributed Processing Systems", Ph.D. Dissertation, report TRITA-CS-8406, Royal Institute of Technology, Stockholm, Sweden, December 1984.

[4] Ali, K.A.M. and Haridi, S. "Global Garbage Collection for Distributed Heap Storage Systems", *International Journal of Parallel Programming*, Vol.15, No.5, October 1986.

[5] Augustsson, L. "A Compiler for lazy ML", *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, August 1984, pp. 218-127.

[6] Baer, J.L. "Modelling for Parallel Computation : A Case Study", *Proceedings 1973 Sagamore Computer Conference on Parallel Processing.*

[7] Baer, J.L. et.al., "The Two-Step Commitment Protocol : Modelling, Specification and Proof Methodology", *Proceedings of Fifth Conference on Software Engineering*, May 1980.

[8] Baker, H. "List Processing in real time on a serial computer", *Communications of ACM*, Vol. 21, No. 4, April 1978, pp. 280-294.

[9] Berthelot, G. et.al., "Petri Net Modelling and Reliability of Distributed Algorithms", *Net Theory and Applications*, Lecture Notes in Computer Science – 84, Springer-Verlag, 1980.

[10] Bhuiyan, S.H. "Development of a Software Tool to Simulate Stochastic Petri Nets", M.Sc. thesis, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, June 1987.

[11] Bobrow, D.G. "Managing reentrant structures using reference-counts", *ACM TOPLAS*, Vol. 2, No.3, March 1980, pp. 269-273.

[12] Brownbridge, D.R. "Cyclic Reference Counting for Combinator Machines", *Proceedings of the Functional Programming Languages and Computer Architecture Conference, Nancy, France*, LNCS 201, Springer-Verlag 1985, pp. 273-288.

[13] Brownbridge, D.R. "Recursive Structures in Computer Systems", Ph.D. Dissertation, Department of Computer Science, University of Newcastle Upon Tyne, UK, September 1984.

[14] Burstall, R.M. et al., "Hope: an experimental applicative language", CSR-62-80, Department of Computer Science, University of Edinburgh, UK, May 1980.

[15] Christopher, T.W. "Reference Count Garbage Collection", *Software Practice and Experience*, Vol. 14, No. 6, June 1984, pp. 503-507.

[16] Cohen, J. "Garbage Collection of Linked Data Structures", *Computing Surveys*, Vol. 13, No. 3, September 1981, pp. 341-367.

[17] Dawson, J.L. "Improved Effectiveness from a Real Time Lisp Garbage Collector", *Conference record of the ACM symposium on Lisp and Functional Programming (1982)*, pp. 159-167.

[18] Deutsch, L.P. and Bobrow, D.G. "An efficient incremental automatic garbage collector", Communications of ACM, Vol. 19, No. 9, September 1976, pp. 522-526.

[19] Dijkstra, E.W. et al., "On-the-Fly Garbage Collection: An exercise in co-operation", *Communications of ACM* , Vol. 21, No. 11, November 1978, pp. 966-975.

[20] Dugan, J.B. "Extended Stochastic Petri Nets : Application and Analysis", Ph.D. Dissertation, Department of Electrical Engineering, Duke University, 1984.

[21] Fairbairn, J. "Ponder and its type system", Technical Report 31, Computer Lab., Cambridge University, UK, November 1982.

[22] Fairbairn, J. "Design and implementation of a simple typed language based on

the lambda calculus", Technical Report 75, Computer Lab., Cambridge University, UK, May 1985.

[23] Gordon, H. et al., Edinburgh LCF, LNCS 78, Springer-Verlag 1979.

[24] Gottlieb, A. et al., "The NYU Ultra Computer - Designing an MIMD shared memory Parallel Computer", *IEEE Transactions on Computers*, Vol. 32, No. 2, February 1983, pp. 175-189.

[25] Holt, R.C. and Cordy, J.R. "The Turing Plus Report", University of Toronto, Computer Systems Research Institute, September 1987.

[26] Hudak, P. and Keller, R.M. "Garbage Collection and Task Deletion in Distributed Applicative Processing Systems", *Proceedings of the ACM symposium on Lisp and Functional Programming (1982)*, pp. 168-178.

[27] Hudak, P. "Distributed Task and Memory Management", *Proceedings of ACM Symposium on Principles of Distributed Computing (1983)*, pp. 277-289.

[28] Hughes, R.J.M. "Reference Counting with Circular Structures in Virtual Memory Applicative Systems", Oxford University (1983).

[29] Hughes, R.J.M. "A Distributed Garbage Collection Algorithm", *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, Nancy, France, LNCS 201, Springer-Verlag 1985, pp. 56-272.

[30] Lieberman , H. and Hewitt, C. "A Real Time Garbage Collector Based on the Lifetime of Objects", *Communications of ACM*, Vol. 26, No. 6, June 1983, pp. 419-429.

[31] Martinez, J. and Silva, M. "A Simple and Fast Algorithm to obtain all Invariants of a Generalized Petri Net", *Application and Theory of Petri Nets*, Informatik Fachbericht 52, Springer Publishing Company, 1982.

[32] Memmi, G. "Linear Algebra in Net Theory", *Net Theory and Applications*, Lecture Notes in Computer Science – 84, Springer-Verlag, 1980.

[33] Milutinovic, V.M. *Computer Architecture, Concepts and Systems*, Reading, North-Holland, 1988.

[34] Molloy, M.K. "On the Integration of Delay and Throughput Measures in

Distributed Processing Models", Ph.D. Dissertation, University of Los Angeles, 1981.

[35] Molloy, M.K. "Performance Analysis Using Stochastic Petri Nets", *IEEE Transactions on Computers*, Vol. C-31(9), September 1982, pp. 913-917.

[36] Moon, D.A. "Garbage Collection in a Large Lisp System", *ACM Symposium on Lisp and Functional Programming (1984)*, pp. 235-246.

[37] Natkin, S. "Reseaus de Petri Stochastiques", Ph.D. Dissertation, CNAM, Paris, June 1980.

[38] Noe, J.D. "A Petri Net Model of CDC6400", Technical Report 71-04-03, Department of Computer Science, University of Washington, Seattle, April 1971, 16-pages.

[39] Noe, J.D. "Application of Net Based Models", Lecture Notes in Parallel Systems 84, Springer-Verlag 1979.

[40] Nori, A.K. "A Storage Reclamation Scheme for Applicative Multiprocessor System", Master's thesis, Department of Computer Science, University of Utah, December 1979.

[41] Peterson, J.L. "Petri Nets", *Computing Surveys*, Vol. 9, No. 3, September 1977, pp. 223-252.

[42] Peterson, J.L. *Petri Net Theory and the Modeling of Systems*, Prentice-Hall Inc., Englewood Cliffs, N.J.07632 (1981)

[43] Petri, C.A. "Kommunikation mit Automaten", Schriften des Institutes fur Instrumentelle Mathematik, Bonn 1962.
(English Translation by Clifford F. Greene, Jr.)

[44] Peyton Jones, S.L. The Implementation of Functional Programming Languages, Prentice-Hall International Series in Computer Science, Englewood Cliffs, N.J. (1987).

[45] Ramchandani, C.V. "Analysis of Asynchronous Systems by Timed Petri Nets", Ph.D. Dissertation, MIT, Cambridge, September 1973.

[46] Ramchandani, C.V. and Ho, G.S. "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets", *IEEE Transactions on Software Engineering*, Vol. SE-6(5), September 1980, pp. 440-449.

[47] Reisig, W. *Petri Nets, An Introduction*, Springer-Verlag, Berlin 1985.

[48] Salkild, J.D. "Implementation and Analysis of two Cyclic Reference Counting Algorithms", M.Sc. thesis, University College, London, June 1985.

[49] Schorr, H., and Waite, W. "An efficient machine-independent procedure for garbage collection in various list structures", *Communications of ACM*, Vol. 10, No. 8, August 1967, pp. 501-506.

[50] Steele, G.L. Jr, "Multiprocessing Compactifying Garbage Collection", *Communications of ACM*, Vol. 18, No. 9, September 1975, pp. 495-508.

[51] Treleavan, P.C. et al., "Data Driven and Demand Driven Computer Architecture", *Computing Surveys*, Vol. 14, No. 1, March 1982, pp. 93-143.

[52] Turner, D.A. "The SASL Language Manual", University of Kent, UK, November 83.

[53] Turner, D.A. "Miranda - a non-strict functional language with polymorphic types", *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, Nancy, France, LNCS 201, Springer-Verlag 1985, pp. 1-16.

[54] Wadler, P. "Introduction to Orwell", Programming Research Group, University of Oxford, UK, 1985.

[55] Weng, K.S. "An Abstract Implementation for a Generalized Data Flow Language", Rep. TR-228, MIT Laboratory for Computer Science, 1979.

[56] Wise, D.S. "Design for a Multiprocessing heap with On-board Reference Counting", *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, Nancy, France, LNCS 201, Springer-Verlag 1985, pp. 289-303.

APPENDIX

The listing of simulator in Turing-Plus is given below.

```
/*************************************************************************
*************************************************************************
*       header file  'globals.h'                                        *
*       This file has global variables, constants and                   *
*       record definitions                                              *
*                                                                       *
*************************************************************************/

const noOfMutators := 16        % total number of mutators in the system
const lastAddr     := 4096      % size of memory in each mutator
const getBuf       := 1         % get buffer operation
const freeBuf      := 2         % release buffer operation
const null         := -1        % nil pointer

/*************************************************************************
*                                                                       *
*       Time constants mutator operations                               *
*                                                                       *
*************************************************************************/

const changePhaseTime           := 1    % change phase time
const allocateObjectTime        := 1    % allocate object time
const makeLocalLinkTime         := 1    % make a local link time
const breakLocalLinkTime        := 1    % break a local link time
const memoryTouchTime           := 1    % time for each memory operation
const receiveMsgTime            := 1    % time to receive a msg and process it
const sendMsgTime               := 10   % time for preparing a message
const msgAckTime                := 5    % time for sending a message ack
const processGcMsgTime          := 5    % time to process a GC message
var MessageFlightTime           : int   % message delay in the IC network

/*************************************************************************
*                                                                       *
*        Type definitions                                               *
*                                                                       *
*************************************************************************/

type peNumbers  : 0..noOfMutators
type memorySize : 1..lastAddr
type lgcStat    : enum(noStart,oneStart, oneDone, someDone, doneGoing)
type colour     : enum(white,red,blue,green,yellow)
type msgClass   : enum(newPleasePaint, lgcPleasePaint, newPaintAck,
                       lgcPaintAck, remoteRead, readReply, lgcStart,
                       lgcDone, ackLgcStart, ackLgcDone, phaseChange,
                       lgcForce)
type operation  : enum(allocateNode, makeLink, breakLink)

/*************************************************************************
*                                                                       *
*       Definition of messages                                          *
*                                                                       *
*************************************************************************/

var msgs: collection of forward msgtype       % the pool of messages

type msgtype: record
  class         : msgClass              % message type
  time          : int                   % time sent, then time of arrival
  destination   : int                   % who gets the msg
  source        : int                   % who sent the msg
  address       : int                   % address to read or paint
  phase         : colour                % colour of message
  nextMsg       : pointer to msgs       % next message in queue
end record
```

```
/*************************************************************************
*                                                                        *
*       definition of a node in the graph                                *
*                                                                        *
*************************************************************************/

type cell: record
  cellAddress          : int             % cell address
  processorNo          : int             % processor number
  markbit              : boolean         % mark bit
  phase                : colour          % phase in which last painted
  leftPhase,rightPhase: colour           % left cell phase
  leftPen,rightPen     : int             % left and right processors
  leftAddr,rightAddr   : int             % left and right cell addresses
end record

/*************************************************************************
*                                                                        *
*       Definition of mutator status record                              *
*                                                                        *
*************************************************************************/

type peState: record
  clock                  : int     % system clock at present
  newPpsOut              : int     % unacknowledged new pl-paint messages
  newPpsMsgsSent         : int     % no of new pl paint msgs sent
  lgcPpsOut              : int     % unacknowledged lgc pl-paint messages
  lgcPpsMsgsSent         : int     % no of lgc pl paint msgs sent
  readsOut               : int     % unacknowledged please-read messages
  firstMsg               : pointer to msgs   % header of message queue
  pePhase                : colour            % current phase of pe
  lgcStartMsgSent        : boolean % sent lgc start message
  lgcDoneMsgSent         : boolean % sent lgc done message
  receivedAckLgcStart    : boolean % received ack msg in reply to lgc start
  receivedAckLgcDone     : boolean % received ack msg in reply to lgc done
  noLgcsDoneInCurrentPhase : int   % no of lgcs done in current phase
  freeList               : int     % pointer to cells, freelist header
  root                   : int     % pointer to cells, local graph root
  memory                 : array memorySize of cell % memory in each mutator
end record

/*************************************************************************
*                                                                        *
*       Global variables                                                 *
*                                                                        *
*************************************************************************/

var state                : array peNumbers of peState
                                % complete state of each pe
var centralLgcRecords    : array peNumbers of lgcStat
                                % master's records of state of lgc's in phase
var masterPhase          : colour
                                % master's present colour
var forcesSent           : boolean := false
                                % if forces not sent forcesSent is false
var usingBuf             : boolean := false
                                % condition indicating some one using buffer
var noOfCellsMarked      : array 1..noOfMutators of int
                                % no of cells marked in each lgc
var noOfCellsRecovered   : array 1..noOfMutators of int
                                % no of cells recovered in each lgc
var noOfCellsAllocated   : array 1..noOfMutators of int
                                % no of cells allocated before one lgc
```

```
var noOfLocalMakeLinks   : array 1..noOfMutators of int
                             % no of local links made before one lgc
var noOfRemoteMakeLinks : array 1..noOfMutators of int
                             % no of remote links made before one lgc
var noOfLocalBreakLinks : array 1..noOfMutators of int
                             % no of local links broken
var noOfRemoteBreakLinks: array 1..noOfMutators of int
                             % no of remote links broken
var readyToStartLgc      : array 1..noOfMutators of boolean
                             % condition indicating if processor
                             % can start an lgc.
var readyToStartMutate   : array 1..noOfMutators of boolean
                             % condition indicating if processor
                             % can start mutation.
var lgcPpsMsgsSentInPhase    : array 1..noOfMutators of boolean
                             % condition indicating if lgc messages have
                             % been sent in the current phase
var nodeInsertionSuccessful : array 1..noOfMutators of boolean
                             % Is updating graph successful?
var updateGraphDepth         : array 1..noOfMutators of int
                             % level reached while updating graph
var lgcAttemptDone           : array 1..noOfMutators of boolean
                             % true if lgc already attempted
var lgcAttemptTime           : array 1..noOfMutators of int
                             % time of attempting to do an lgc
var lgcStartTime             : array 1..noOfMutators of int
                             % time at which lgc is started
var lgcCompletedTime         : array 1..noOfMutators of int
                             % time of completing an lgc
var mutateAttemptDone        : array 1..noOfMutators of boolean
                             % true if mutation is already attempted
var mutateAttemptTime        : array 1..noOfMutators of int
                             % mutate attempted time
var mutateStartTime          : array 1..noOfMutators of int
                             % mutation starting time
var mutateStartTimeEntered   : array 1..noOfMutators of boolean
                             % indicates if mutation start time is entered

/*************************************************************************
*                                                                        *
*       Variables to choose a random operation                           *
*                                                                        *
*************************************************************************/

var allocateNodeLowerLimit    : int
var allocateNodeUpperLimit    : int
var makeLinkLowerLimit        : int
var makeLinkUpperLimit        : int
var breakLinkLowerLimit       : int
var breakLinkUpperLimit       : int
var localOrRemoteLowerLimit   : int
var localOrRemoteUpperLimit   : int
var probabilityOfRemoteLinks : int
var averageNoOfLgcsLimit      : real     % average no. of lgc's per phase
```

```
/*************************************************************************
/*************************************************************************
*                                                                        *
*       Main file of the simulator 'nonstop.t'                           *
*       All modules are included into this file                          *
*                                                                        *
*************************************************************************/

grant
        (var msgs, var state, msgAckTime, MessageFlightTime, usingBuf,
         getBuf, null, freeBuf, msgClass, peNumbers, noOfMutators,
         mailBox, memoryTouchTime, allocateObjectTime, colour,
         makeLocalLinkTime, sendMsgTime, breakLocalLinkTime,
         changePhaseTime, memorySize, var centralLgcRecords, lgcStat,
         processGcMsgTime, masterPhase,mutator, lastAddr, cell,
         localGarbageCollection, initialSetup, allocateNodeLowerLimit,
         allocateNodeUpperLimit, makeLinkLowerLimit, makeLinkUpperLimit,
         breakLinkLowerLimit, breakLinkUpperLimit, noOfCellsMarked,
         noOfCellsRecovered, noOfCellsAllocated, noOfLocalMakeLinks,
         noOfRemoteMakeLinks, noOfLocalBreakLinks, noOfRemoteBreakLinks,
         var forcesSent, var readyToStartLgc, var nodeInsertionSuccessful,
         var updateGraphDepth, var lgcPpsMsgsSentInPhase,
         var lgcAttemptDone, var mutateAttemptDone, var readyToStartMutate,
         var lgcAttemptTime, var lgcStartTime, var lgcCompletedTime,
         var mutateAttemptTime,var mutateStartTime, var mutateStartTimeEntered,
         probabilityOfRemoteLinks, localOrRemoteLowerLimit,
         localOrRemoteUpperLimit, averageNoOfLgcsLimit, receiveMsgTime)

include "globals.h"             /* include global header file               */

child "messagehandler.t"        /* include message handling module          */

child "initialsetup.t"          /* initial set up module                    */

child "lgc.t"                   /* include local garbage collection module  */

child "mutator.t"               /* include mutator routines module          */

child "master.t"                /* include master module                    */


/*************************************************************************
*                                                                        *
*       Master Process program                                           *
*                                                                        *
*************************************************************************/

process monitorPhase
 const masterPe : int := 0
 bind var states to state(masterPe)
 var message    : pointer to msgs
 var tempMsg    : pointer to msgs
 states.pePhase := colour.red
 masterPhase    := states.pePhase

 loop
   states.pePhase := masterPhase
   if ((states.firstMsg not= nil(msgs)) and
      (states.clock >= msgs(states.firstMsg).time)) then
      tempMsg := msgs(states.firstMsg).nextMsg
      message := states.firstMsg
      states.firstMsg := tempMsg
   else
      message := nil(msgs)
```

```
   end if

   loop                                    /* loop forever through colours */
      exit when message = nil(msgs)
      assert ((msgs(message).class = msgClass.lgcStart) or
              (msgs(message).class = msgClass.lgcDone))
      case msgs(message).class of
         label msgClass.lgcStart : master.replyLgcStart(message)
         label msgClass.lgcDone  : master.replyLgcDone(message)
                                   master.checkPeStatus
      end case

      if (states.firstMsg not= nil(msgs)) and
         (states.clock >= msgs(states.firstMsg).time) then
          tempMsg := msgs(states.firstMsg).nextMsg
          message := states.firstMsg
          states.firstMsg := tempMsg
      else
          message := nil(msgs)
      end if
   end loop                                /* loop until current phase is over */
   states.clock += 1
   mailBox.scheduler(masterPe)
 end loop
end monitorPhase

/*************************************************************************
*                                                                        *
*       Mutator Process                                                  *
*                                                                        *
*************************************************************************/

process processor(pen : int)
  var sourcePen : int := pen
  var message   : pointer to msgs
  var tempMsg   : pointer to msgs
  var opCode    : operation
  var opChosen  : int
  var lgcStartMsgSent, lgcDoneMsgSent : boolean
  bind var states to state(sourcePen)

  initialSetup.initially(sourcePen)
  initialSetup.setupFreeList(sourcePen)
  initialSetup.setupInitialGraph(sourcePen)

  loop
    if ((states.firstMsg not= nil(msgs)) and
        (states.clock >= msgs(states.firstMsg).time)) then
        tempMsg := msgs(states.firstMsg).nextMsg
        message := states.firstMsg
        states.firstMsg := tempMsg
    else
        message := nil(msgs)
    end if

    if message not= nil(msgs) then
     loop
        case msgs(message).class of
          label msgClass.phaseChange    : mutator.changePePhase(sourcePen, message)
          label msgClass.lgcForce       : localGarbageCollection.forceLgc(sourcePen)
          label msgClass.newPleasePaint : mutator.sendNewPaintAck(message)
          label msgClass.lgcPleasePaint : mutator.sendLgcPaintAck(message)
          label msgClass.remoteRead     : mutator.sendReadReply(message)
          label msgClass.readReply      : mutator.receiveReadReply(message)
```

```
            label msgClass.newPaintAck     : mutator.receiveNewPaintAck(message)
            label msgClass.lgcPaintAck     : mutator.receiveLgcPaintAck(message)
            label msgClass.ackLgcStart     : mutator.receiveAckLgcStart(message)
            label msgClass.ackLgcDone      : mutator.receiveAckLgcDone(message)
         end case
         if (states.firstMsg not= nil(msgs)) and
            (states.clock >= msgs(states.firstMsg).time) then
            tempMsg := msgs(states.firstMsg).nextMsg
            message := states.firstMsg
            states.firstMsg := tempMsg
         else
            message := nil(msgs)
         end if
         exit when message = nil(msgs)
      end loop
   else
         if not(mutateAttemptDone(pen)) then
            mutateAttemptTime(pen) := states.clock
            mutateAttemptDone(pen) := true
         end if

         if (readyToStartMutate(pen)) then
            if not(mutateStartTimeEntered(pen)) then
               mutateStartTime(pen)        := states.clock
               mutateStartTimeEntered(pen) := true
            end if
            randint(opChosen, 1, 100)
            if (opChosen >= breakLinkLowerLimit and
               opChosen <= breakLinkUpperLimit) then
               opCode := operation.breakLink
            elsif (opChosen >= allocateNodeLowerLimit and
               opChosen <= allocateNodeUpperLimit) then
               opCode := operation.allocateNode
            elsif (opChosen >= makeLinkLowerLimit and
               opChosen <= makeLinkUpperLimit) then
               opCode := operation.makeLink
            end if

            assert (opCode = operation.allocateNode or
                    opCode = operation.makeLink  or
                    opCode = operation.breakLink)
            case opCode of
               label operation.allocateNode    : mutator.allocateNode(sourcePen)
               label operation.makeLink        : mutator.makeLink(sourcePen)
               label operation.breakLink       : mutator.breakLink(sourcePen)
            end case
         end if
      end if
      states.clock += 1                          /* advance processor clock */
      mailBox.scheduler(pen)
   end loop
end processor

/*************************************************************************
*                                                                        *
*               accept input parameters                                  *
*                                                                        *
*************************************************************************/

put "input the value of allocate node lower limit  "..
get allocateNodeLowerLimit
put "input the value of allocate node upper limit  "..
get allocateNodeUpperLimit
put "input the value of make link lower limit      "..
```

```
get makeLinkLowerLimit
put "input the value of make link upper limit      "..
get makeLinkUpperLimit
put "input the value of break link lower limit     "..
get breakLinkLowerLimit
put "input the value of break link upper limit     "..
get breakLinkUpperLimit
put "input the value of remote link probability 0  "..
get probabilityOfRemoteLinks
put "input value of average no of lgcs per phase   "..
get averageNoOfLgcsLimit
put "in'ut value of message delay time             "..
get MessageFlightTime

/*********************************************************************
*                                                                    *
*               fork all processeses                                 *
*                                                                    *
*********************************************************************/

for i: 1..noOfMutators
   fork processor(i)
end for
fork monitorPhase
```

```
/*********************************************************************
*********************************************************************
*       This module is in file 'initialsetup.t'                     *
*       Initial setup routines module                               *
*                                                                   *
*********************************************************************/

parent "nonstop.t"      /* This module is inserted in 'nonstop.t' file */

stub module initialSetup

        import(lastAddr, state, colour, null, msgs, allocateNodeLowerLimit,
                allocateNodeUpperLimit, makeLinkLowerLimit, makeLinkUpperLimit,
                breakLinkLowerLimit, breakLinkUpperLimit, centralLgcRecords,
                lgcStat, noOfMutators, noOfCellsMarked, noOfCellsRecovered,
                noOfCellsAllocated, noOfLocalMakeLinks,noOfRemoteMakeLinks,
                noOfLocalBreakLinks, noOfRemoteBreakLinks, cell,
                readyToStartLgc, nodeInsertionSuccessful, updateGraphDepth,
                lgcPpsMsgsSentInPhase, lgcAttemptDone, readyToStartMutate,
                mutateAttemptDone, lgcAttemptTime, lgcStartTime,
                lgcCompletedTime,  mutateAttemptTime, mutateStartTime,
                mutateStartTimeEntered, probabilityOfRemoteLinks,
                localOrRemoteLowerLimit, localOrRemoteUpperLimit,
                averageNoOfLgcsLimit, MessageFlightTime)

        export(initially, setupFreeList, setupInitialGraph, pickFirstCell,
               updateGraph, insertIntoFreeList)

        procedure initially(pen : int)
        procedure setupFreeList(pen : int)
        procedure setupInitialGraph(pen : int)
        function pickFirstCell(pen : int):int
        procedure updateGraph(var graphRoot:int, cellToBeInserted:int,pen:int)
        procedure insertIntoFreeList(pen:int,cellToBeInserted:int)

end initialSetup

body module initialSetup

        const sizeOfInitialGraph        := (lastAddr div 2)
        const left                      := 1
        const right                     := 2
        const maximumUpdateGraphDepth   := 200

/*********************************************************************
*                                                                    *
*       Initialization routine                                       *
*                                                                    *
*********************************************************************/

body procedure initially
  bind var states to state(pen)

  randomize
  for i : 1..lastAddr
    states.memory(i).cellAddress := i
    states.memory(i).processorNo := pen
    states.memory(i).markbit     := true
    states.memory(i).phase       := colour.white
    states.memory(i).leftPen     := null
    states.memory(i).rightPen    := null
    states.memory(i).leftAddr    := null
    states.memory(i).rightAddr   := null
  end for
```

```
states.clock                    := 0
states.newPpsOut                := 0
states.newPpsMsgsSent           := 0
states.lgcPpsOut                := 0
states.lgcPpsMsgsSent           := 0
states.readsOut                 := 0
states.firstMsg                 := nil(msgs)
states.pePhase                  := colour.red
states.lgcStartMsgSent          := false
states.lgcDoneMsgSent           := false
states.receivedAckLgcStart      := true     /* ready to send lgc done message */
states.receivedAckLgcDone       := true     /* ready to send lgc start message */
states.freeList                 := null
states.root                     := null
centralLgcRecords(pen)          := lgcStat.noStart
noOfCellsMarked(pen)            := 0
noOfCellsRecovered(pen)         := 0
noOfCellsAllocated(pen)         := 0
noOfLocalMakeLinks(pen)         := 0
noOfRemoteMakeLinks(pen)        := 0
noOfLocalBreakLinks(pen)        := 0
noOfRemoteBreakLinks(pen)       := 0
readyToStartLgc(pen)            := true
readyToStartMutate(pen)         := true
lgcPpsMsgsSentInPhase(pen)      := false
nodeInsertionSuccessful(pen)    := true
updateGraphDepth(pen)           := 0
lgcAttemptDone(pen)             := false
lgcAttemptTime(pen)             := 0
lgcStartTime(pen)               := 0
lgcCompletedTime(pen)           := 0
mutateAttemptDone(pen)          := false
mutateAttemptTime(pen)          := 0
mutateStartTime(pen)            := 0
mutateStartTimeEntered(pen)     := false

if (pen = 1) then
   localOrRemoteLowerLimit := 10
   localOrRemoteUpperLimit := (localOrRemoteLowerLimit +
                        round((noOfMutators * probabilityOfRemoteLinks) /
                        (noOfMutators - 1)))

   /*  print the banner for output file */

   put repeat(" ", 130) put repeat("*", 130)
   put repeat("*", 1), repeat(" ", 128), repeat("*",1)
   put repeat("*", 1), repeat(" ", 40), "NO OF MUTATORS          ",
      noOfMutators:6, repeat(" ",58), repeat("*",1)
   put repeat("*", 1), repeat(" ", 128), repeat("*",1)
   put repeat("*", 1), repeat(" ", 40), "NO OF NODES IN A GRAPH  ",
      lastAddr:6, repeat(" ",58), repeat("*",1)
   put repeat("*", 1), repeat(" ", 128), repeat("*",1)
   put repeat("*", 1), repeat(" ", 40), "PROBABILITY OF ALLOC    ",
      ((allocateNodeUpperLimit - allocateNodeLowerLimit)/100):6:3,
      repeat(" ",58), repeat("*",1)
   put repeat("*", 1), repeat(" ", 128), repeat("*",1)
   put repeat("*", 1), repeat(" ", 40), "PROBABILITY OF MAKE     ",
      ((makeLinkUpperLimit - makeLinkLowerLimit)/100):6:3,
      repeat(" ",58), repeat("*",1)
   put repeat("*", 1), repeat(" ", 128), repeat("*",1)
   put repeat("*", 1), repeat(" ", 40), "PROBABILITY OF BREAK    ",
      ((breakLinkUpperLimit - breakLinkLowerLimit)/100):6:3,
      repeat(" ",58), repeat("*",1)
```

```
    put repeat("*", 1), repeat(" ", 128), repeat("*",1)
    put repeat("*", 1), repeat(" ", 40), "PROB OF REMOTE MAKE      ",
       (probabilityOfRemoteLinks/100):6:3,repeat(" ",58),repeat("*",1)
    put repeat("*", 1), repeat(" ", 128), repeat("*",1)
    put repeat("*", 1), repeat(" ", 40), "AVERAGE LGCS PER PE      ",
       averageNoOfLgcsLimit:6:3, repeat(" ",58), repeat("*",1)
    put repeat("*", 1), repeat(" ", 128), repeat("*",1)
    put repeat("*", 1), repeat(" ", 40), "MESSAGE DELAY TIME       ",
       MessageFlightTime:6, repeat(" ",58), repeat("*",1)
    put repeat("*", 1), repeat(" ", 128), repeat("*",1)
    put repeat("*", 130) put repeat(" ", 130)

    put " PEN  "," PHASE ", "  RECLAIM "," LO_MAKE "," RE_MAKE ",
        "  LO_BREAK "," RE_BREAK "," ALLOC ", "  LGC_PPS ",
        "  NEW_PPS "," MU_W_TIME "," LGC_W_TIME "," LGC_TIME"
  end if

end initially

/*************************************************************************
*                                                                        *
*        Set up free list of a processor                                 *
*                                                                        *
*************************************************************************/

body procedure setupFreeList
  bind var states to state(pen)
  bind var freeListRoot to states.freeList
  freeListRoot := 1
  for cellAddress : 1..lastAddr
    states.memory(cellAddress).markbit     := false
    states.memory(cellAddress).phase       := colour.white
    states.memory(cellAddress).leftPhase   := colour.white
    states.memory(cellAddress).rightPhase  := colour.white
    states.memory(cellAddress).leftPen     := null
    states.memory(cellAddress).rightPen    := pen
    states.memory(cellAddress).leftAddr    := null

    if (cellAddress = lastAddr) then
      states.memory(cellAddress).rightAddr   := null
    else
      states.memory(cellAddress).rightAddr   := (cellAddress + 1)
    end if
  end for
end setupFreeList

/*************************************************************************
*                                                                        *
*        Set up initial graph                                            *
*                                                                        *
*************************************************************************/

body procedure setupInitialGraph
  bind var states to state(pen)
  var cellToBeInserted : int
  for cellAddress : 1..sizeOfInitialGraph
    cellToBeInserted                          := pickFirstCell(pen)
    states.memory(cellToBeInserted).phase     := colour.white
    states.memory(cellToBeInserted).processorNo := pen
    states.memory(cellToBeInserted).leftPen   := null
    states.memory(cellToBeInserted).rightPen  := null
    states.memory(cellToBeInserted).leftAddr  := null
    states.memory(cellToBeInserted).rightAddr := null
    updateGraphDepth(pen) := 0
```

```
    updateGraph(states.root, cellToBeInserted, pen)
    if not(nodeInsertionSuccessful(pen)) then
       insertIntoFreeList(pen,cellToBeInserted)
       nodeInsertionSuccessful(pen) := true
    end if
  end for
end setupInitialGraph

/*********************************************************************
*                                                                    *
*        Remove the first cell from a free list                      *
*                                                                    *
*********************************************************************/

body function pickFirstCell
  bind var states to state(pen)
  var currentPtr: int
  currentPtr    := states.freeList

  if (currentPtr  not= null)  then
    states.freeList := states.memory(currentPtr).rightAddr
    result currentPtr
  else
    result null
  end if
end pickFirstCell

/*********************************************************************
*                                                                    *
*        Insert a cell into free list                                *
*                                                                    *
*********************************************************************/

body procedure insertIntoFreeList
  bind var states to state(pen)
  var currentPtr, previousPtr : int
  currentPtr := states.freeList
  previousPtr := null

  if currentPtr = null then
     states.freeList := cellToBeInserted
  else
    loop
       exit when (states.memory(currentPtr).rightAddr = null)
       currentPtr := states.memory(currentPtr).rightAddr
    end loop
    states.memory(currentPtr).rightAddr := cellToBeInserted
  end if

end insertIntoFreeList

/*********************************************************************
*                                                                    *
*        Insert allocated cell into a graph                          *
*                                                                    *
*********************************************************************/

body procedure updateGraph
  bind var states to state(pen)
  var leftOrRight : int
  var currentPtr  : int

  updateGraphDepth(pen) += 1
```

```
  if (updateGraphDepth(pen) >= maximumUpdateGraphDepth) then
      nodeInsertionSuccessful(pen) := false
      return
  else
      if (graphRoot = null) then
         graphRoot := cellToBeInserted
      else
         currentPtr := graphRoot
         randint(leftOrRight, left, right)

         if (leftOrRight = left) then
            if (states.memory(currentPtr).leftAddr = null) then
               states.memory(currentPtr).leftPen  := pen
               states.memory(currentPtr).leftAddr := cellToBeInserted
            elsif (states.memory(currentPtr).leftPen = pen) and
                  (states.memory(currentPtr).leftPen not= null) then
                  updateGraph(states.memory(currentPtr).leftAddr,
                              cellToBeInserted,pen)
            elsif (states.memory(currentPtr).rightPen = pen) and
                  (states.memory(currentPtr).rightPen not= null) then
                  updateGraph(states.memory(currentPtr).rightAddr,
                              cellToBeInserted,pen)
            elsif (states.memory(currentPtr).rightAddr = null) then
                  states.memory(currentPtr).rightPen  := pen
                  states.memory(currentPtr).rightAddr := cellToBeInserted
            else
                  nodeInsertionSuccessful(pen) := false
                  return
            end if
         end if

         if (leftOrRight = right) then
            if (states.memory(currentPtr).rightAddr = null) then
               states.memory(currentPtr).rightPen  := pen
               states.memory(currentPtr).rightAddr := cellToBeInserted
            elsif (states.memory(currentPtr).rightPen = pen) and
                  (states.memory(currentPtr).rightPen not= null) then
                  updateGraph(states.memory(currentPtr).rightAddr,
                              cellToBeInserted,pen)
            elsif (states.memory(currentPtr).leftPen = pen) and
                  (states.memory(currentPtr).leftPen not= null) then
                  updateGraph(states.memory(currentPtr).leftAddr,
                              cellToBeInserted,pen)
            elsif (states.memory(currentPtr).leftAddr = null) then
                  states.memory(currentPtr).leftPen  := pen
                  states.memory(currentPtr).leftAddr := cellToBeInserted
            else
                  nodeInsertionSuccessful(pen) := false
                  return
            end if
         end if

      end if
  end if
end updateGraph

end initialSetup
```

```
/*************************************************************************
*************************************************************************
*       This module is in file 'messagehandler.t'                       *
*       Monitor module.  Has message transmission routines              *
*       and mutator status table.                                       *
*                                                                       *
*************************************************************************/

parent "nonstop.t"      /* this module is included in 'nonstop.t' file */

stub monitor mailBox
  import(var msgs, var state, MessageFlightTime, usingBuf, getBuf,
         freeBuf, msgClass, peNumbers, noOfMutators)
  export(bufferManager, send, checkPreviousLgcDoneAck,
         checkPreviousLgcStartAck, scheduler)

  procedure bufferManager(var inMsg:pointer to msgs, bufferOperation:int)
  procedure send(inMsg: pointer to msgs)
  procedure checkPreviousLgcDoneAck(pen : int)
  procedure checkPreviousLgcStartAck(pen : int)
  procedure scheduler(pen : int)

end mailBox

body monitor mailBox

var busyBuf                       : condition
                                    % event of some one using buffer
var previousLgcDoneAckReceived    : array 1..noOfMutators of condition deferred
                                    % if the previous lgc done ack received
var previousLgcStartAckReceived : array 1..noOfMutators of condition deferred
                                    % if the previous lgc start ack received
var wakeUpProcessor               : array 0..noOfMutators of condition
                                    %processor waiting for its turn, signalled by
                                    % the scheduler

/*************************************************************************
*                                                                        *
*       Allocate and deallocate buffer for messages                      *
*                                                                        *
*************************************************************************/

body procedure bufferManager
  if usingBuf = true then
    wait busyBuf
  end if
  usingBuf := true
  if bufferOperation = getBuf then
    new msgs, inMsg
  else
    free msgs, inMsg
  end if
  usingBuf := false
  signal busyBuf
end bufferManager

/*************************************************************************
*                                                                        *
*       Scheduler routine, wakes up one processor and blocks others      *
*                                                                        *
*************************************************************************/

body procedure scheduler
  var currentYoungestClock      : int
```

```
  var currentYoungestProcessor  : int

  currentYoungestClock          := state(0).clock
  currentYoungestProcessor      := 0
  for i : 0..(noOfMutators )
    if currentYoungestClock > state(i).clock then
       currentYoungestClock     := state(i).clock
       currentYoungestProcessor := i
    end if
  end for
  if (state(pen).clock > currentYoungestClock) then
    signal wakeUpProcessor(currentYoungestProcessor)
    wait wakeUpProcessor(pen)
  end if
end scheduler

/*************************************************************************
*                                                                        *
*       A processor wakes up another processor in case it has            *
*       to wait for previous lgc done acknowledgement message            *
*                                                                        *
*************************************************************************/

body procedure checkPreviousLgcDoneAck
bind var states to state(pen)
  var currentYoungestClock      : int := 0
  var currentYoungestProcessor  : int := 0
  var nextYoungestClock         : int := 0
  var nextYoungestProcessor     : int := 0

  if ((states.lgcDoneMsgSent) and not(states.receivedAckLgcDone)) then
     currentYoungestClock       := state(0).clock
     currentYoungestProcessor   := 0
     for i : 1..(noOfMutators)
        if currentYoungestClock >= state(i).clock then
           nextYoungestClock    := currentYoungestClock
           nextYoungestProcessor    := currentYoungestProcessor
           currentYoungestClock := state(i).clock
           currentYoungestProcessor     := i
        elsif (nextYoungestClock > state(i).clock) then
           nextYoungestClock    := state(i).clock
           nextYoungestProcessor        := i
        end if
     end for

     if ((state(pen).clock >= currentYoungestClock) and
         (pen not= currentYoungestProcessor)) then
          assert not(empty(wakeUpProcessor(currentYoungestProcessor))
          signal wakeUpProcessor(currentYoungestProcessor)
          wait previousLgcDoneAckReceived(pen)
     elsif ((state(pen).clock >= nextYoungestClock) and
            (pen not= nextYoungestProcessor)) then
          assert not(empty(wakeUpProcessor(nextYoungestProcessor))
          signal wakeUpProcessor(nextYoungestProcessor)
          wait previousLgcDoneAckReceived(pen)
     end if
  end if
end checkPreviousLgcDoneAck

/*************************************************************************
*                                                                        *
*       Check for the previous lgc start acknowledgement message         *
*                                                                        *
*************************************************************************/
```

```
body procedure checkPreviousLgcStartAck
bind var states to state(pen)
  if ((states.lgcStartMsgSent) and not(states.receivedAckLgcStart)) then
     wait previousLgcStartAckReceived(pen)
  end if
end checkPreviousLgcStartAck

/*********************************************************************
*                                                                    *
*       Routine to send a message from one processor to another.     *
*       Accepts a message and puts into the message queue of the     *
*       destination processor                                        *
*                                                                    *
*********************************************************************/

body procedure send
  var currentPtr, previousPtr : pointer to msgs
  var dest      := msgs(inMsg).destination
  var source    := msgs(inMsg).source
  var arrival   := msgs(inMsg).time + MessageFlightTime
  bind var states to state(dest)

  /* print statements for debugging purpose

   case msgs(inMsg).class of
      label msgClass.lgcStart           :   put "sending gc start message"..
      label msgClass.newPleasePaint     :   put "sending pl new paint message"..
      label msgClass.lgcPleasePaint     :   put "sending pl lgc paint message"..
      label msgClass.newPaintAck        :   put "sending new paint ack message"..
      label msgClass.lgcPaintAck        :   put "sending lgc paint ack message"..
      label msgClass.remoteRead         :   put "sending remote read message"..
      label msgClass.readReply          :   put "sending read reply message"..
      label msgClass.lgcDone            :   put "sending lgc done message"..
      label msgClass.phaseChange        :   put "sending phase change message "..
      label msgClass.lgcForce           :   put "sending lgc force message"..
      label msgClass.ackLgcStart        :   put "sending acknowledge lgc start message "..
      label msgClass.ackLgcDone         :   put "sending acknowledge lgc done message "..
  end case
  put "  from source proc  ", source,"  to dest proc  ", dest   */

  msgs(inMsg).nextMsg  := nil(msgs)
  msgs(inMsg).time     := arrival
  if states.firstMsg = nil(msgs) then                  /* Queue is empty */
    states.firstMsg := inMsg
  else
    if arrival <= msgs(states.firstMsg).time then      /* Insert at the head */
       msgs(inMsg).nextMsg := states.firstMsg
       states.firstMsg := inMsg
    else                                               /* Scan to find place */
       currentPtr  := states.firstMsg
       previousPtr := nil(msgs)
       loop
         previousPtr := currentPtr
         currentPtr  := msgs(currentPtr).nextMsg
         exit when (currentPtr = nil(msgs) or arrival <= msgs(currentPtr).time)
       end loop
       if (currentPtr = nil(msgs)) then                /* Insert at the end */
         msgs(previousPtr).nextMsg := inMsg
       else                                            /* Insert in the middle */
         msgs(inMsg).nextMsg        := currentPtr
         msgs(previousPtr).nextMsg  := inMsg
       end if
    end if
```

```
  end if

   case msgs(inMsg).class of
     label msgClass.ackLgcStart : signal previousLgcStartAckReceived(dest)
     label                      : /* do nothing  */
   end case

end send

end mailBox
```

```
/*********************************************************************
*********************************************************************
*        This module is in file 'mutator.t'                         *
*        Mutator routines module                                    *
*                                                                   *
*********************************************************************/

parent "nonstop.t"              /* include this module in 'nonstop.t' file  */

stub module mutator

        import(msgs, state, mailBox, freeBuf,getBuf, null,
               allocateObjectTime, colour, makeLocalLinkTime, msgClass,
               sendMsgTime, breakLocalLinkTime, changePhaseTime,
               lastAddr, cell, noOfMutators, localGarbageCollection,
               initialSetup, noOfCellsAllocated, noOfLocalMakeLinks,
               noOfRemoteMakeLinks, noOfLocalBreakLinks, noOfRemoteBreakLinks,
               readyToStartLgc, nodeInsertionSuccessful,receiveMsgTime,
               updateGraphDepth, lgcPpsMsgsSentInPhase, readyToStartMutate,
               localOrRemoteLowerLimit, localOrRemoteUpperLimit, msgAckTime)

        export(allocateNode, makeLocalLink,previousPhase,
               makeRemoteLink, makeLink, breakLink, changePePhase,
               sendNewPaintAck, sendLgcPaintAck,sendReadReply,
               receiveReadReply, receiveNewPaintAck,receiveLgcPaintAck,
               receiveAckLgcStart, receiveAckLgcDone, nextPhase)

        procedure allocateNode(pen:int)
        procedure makeLocalLink(pen:int)
        procedure makeRemoteLink(sourcePen:int, destPen:int)
        procedure makeLink(sourcePen: int)
        procedure breakLink(sourcePen:int)
        procedure changePePhase(pen:int, inMsg : pointer to msgs)
        procedure sendNewPaintAck(inMsg: pointer to msgs)
        procedure sendLgcPaintAck(inMsg: pointer to msgs)
        procedure sendReadReply(inMsg: pointer to msgs)
        procedure receiveReadReply(var inMsg: pointer to msgs)
        procedure receiveNewPaintAck(var inMsg: pointer to msgs)
        procedure receiveLgcPaintAck(var inMsg: pointer to msgs)
        procedure receiveAckLgcStart(var inMsg: pointer to msgs)
        procedure receiveAckLgcDone(var inMsg: pointer to msgs)
        function nextPhase(presentPhase:colour):colour
        function previousPhase(presentPhase:colour):colour
        function sourceRandomWalk(root : int, pen : int) : int
        function destRandomWalk(root : int, pen : int) : int

end mutator

body module mutator

        /* initially grown graph is half the full size */
        const sizeOfInitialGraph        := (lastAddr div 2)
        const left                      := 1
        const right                     := 2
        const remote                    := 99
        var sourceDepthLevel            : int
        var destDepthLevel              : int
        var destDepth : int             := 0
        var sourceDepth : int           := 0

/*********************************************************************
*                                                                   *
*      Routine to allocate a node                                   *
*                                                                   *
```

```
*************************************************************************/

body procedure allocateNode
  bind var states to state(pen)
  var pickNode  : int

  pickNode := initialSetup.pickFirstCell(pen)
  if (pickNode = null) then
    localGarbageCollection.lgc(pen)
  else
    states.memory(pickNode).phase      := colour.white
    states.memory(pickNode).leftPen    := null
    states.memory(pickNode).rightPen   := null
    states.memory(pickNode).leftAddr   := null
    states.memory(pickNode).rightAddr  := null
    noOfCellsAllocated(pen) += 1          % increment no. of cells allocated
    states.clock += allocateObjectTime    % advance processor's clock

    updateGraphDepth(pen) := 0
    initialSetup.updateGraph(states.root, pickNode, pen)
    if not (nodeInsertionSuccessful(pen)) then
       initialSetup.insertIntoFreeList(pen,pickNode)
       nodeInsertionSuccessful(pen) := true
       noOfCellsAllocated(pen) -= 1
    end if
  end if

end allocateNode

/*************************************************************************
*                                                                        *
*       Procedure to make a local link between two cells                 *
*                                                                        *
*************************************************************************/

body procedure makeLocalLink
  bind var states to state(pen)
  var leftOrRight : int
  var sourceNode, destNode : int

  randint(leftOrRight, left, right)
  sourceDepth    := 0
  sourceNode     := sourceRandomWalk(states.root, pen)
                                    /* identify a source node */
  destDepth      := 0
  destNode       := destRandomWalk(states.root, pen)
                                    /* identify a destination node */

  if ((sourceNode not= null) and
      ((states.memory(sourceNode).leftAddr = null) or
       (states.memory(sourceNode).rightAddr = null)) and
       (destNode not= null) and (sourceNode not= destNode)) then

     if (leftOrRight = left) then
       if (states.memory(sourceNode).leftAddr = null) then

           states.memory(sourceNode).leftPen   :=  pen
           states.memory(sourceNode).leftAddr  := destNode
           states.memory(sourceNode).leftPhase := colour.white
           noOfLocalMakeLinks(pen) += 1
           states.clock += makeLocalLinkTime      /* advance processor clock */

        elsif (states.memory(sourceNode).rightAddr = null) then
```

```
            states.memory(sourceNode).rightPen    := pen
            states.memory(sourceNode).rightAddr   := destNode
            states.memory(sourceNode).rightPhase  := colour.white
            noOfLocalMakeLinks(pen) += 1
            states.clock += makeLocalLinkTime      /* advance processor clock */

         end if
      end if

      if (leftOrRight = right) then
         if (states.memory(sourceNode).rightAddr = null) then

            states.memory(sourceNode).rightPen    :=  pen
            states.memory(sourceNode).rightAddr   := destNode
            states.memory(sourceNode).rightPhase  := colour.white
            noOfLocalMakeLinks(pen) += 1
            states.clock += makeLocalLinkTime      /* advance processor clock */

         elsif (states.memory(sourceNode).leftAddr = null) then

            states.memory(sourceNode).leftPen     := pen
            states.memory(sourceNode).leftAddr    := destNode
            states.memory(sourceNode).leftPhase   := colour.white
            noOfLocalMakeLinks(pen) += 1
            states.clock += makeLocalLinkTime      /* advance processor clock */

         end if
      end if

   end if
end makeLocalLink

/*********************************************************************
*                                                                    *
*       Make a remote link, send a newLinkPaint message              *
*       to the destination processor                                 *
*                                                                    *
**********:*********************************************************/

body procedure makeRemoteLink
   var pp : pointer to msgs
   bind far states to state(sourcePen)
   bind var Now to states.clock
   var sourceNode, destNode : int
   var leftOrRight : int

   randint(leftOrRight, left, right)
   sourceDepth  := 0
   sourceNode   := sourceRandomWalk(states.root, sourcePen)
   destDepth    := 0
   destNode     := destRandomWalk(states.root, destPen)
   if ((sourceNode not= null) and
       (((states.memory(sourceNode).leftAddr = null) and
         (states.memory(sourceNode).leftPen = null)) or
         ((states.memory(sourceNode).rightAddr = null) and
         (states.memory(sourceNode).rightPen = null))) and
         (destNode not= null)) then

      if (leftOrRight = left) then
        if (states.memory(sourceNode).leftAddr = null) then

            states.memory(sourceNode).leftPen    := destPen
            states.memory(sourceNode).leftAddr   := destNode
            states.memory(sourceNode).leftPhase  := states.pePhase
```

```
            noOfRemoteMakeLinks(sourcePen) += 1
            states.clock += makeLocalLinkTime

         elsif (states.memory(sourceNode).rightAddr = null) then

            states.memory(sourceNode).rightPen   := destPen
            states.memory(sourceNode).rightAddr  := destNode
            states.memory(sourceNode).rightPhase := states.pePhase
            noOfRemoteMakeLinks(sourcePen) += 1
            states.clock += makeLocalLinkTime

         end if
      end if

      if (leftOrRight = right) then
         if (states.memory(sourceNode).rightAddr = null) then

            states.memory(sourceNode).rightPen :=  destPen
            states.memory(sourceNode).rightAddr := destNode
            states.memory(sourceNode).rightPhase := states.pePhase
            noOfRemoteMakeLinks(sourcePen) += 1
            states.clock += makeLocalLinkTime

         elsif (states.memory(sourceNode).leftAddr = null) then

            states.memory(sourceNode).leftPen  := destPen
            states.memory(sourceNode).leftAddr := destNode
            states.memory(sourceNode).leftPhase := states.pePhase
            noOfRemoteMakeLinks(sourcePen) += 1
            states.clock += makeLocalLinkTime

         end if
      end if

      mailBox.bufferManager(pp,getBuf)
      state(destPen).memory(destNode).phase := states.pePhase
      states.newPpsOut      += 1
                          /* increment outstanding newLinkPaint message count */
      states.newPpsMsgsSent += 1
                          /* increment number of newLinkPaint messages sent   */
                          /* for statistics purpose */
      msgs(pp).time              := Now      /* Now is equal to present time */
      msgs(pp).class             := msgClass.newPleasePaint
      msgs(pp).destination       := destPen
      msgs(pp).source            := sourcePen
      msgs(pp).address           := destNode
      msgs(pp).phase             := states.pePhase
      mailBox.send(pp)
      states.clock += sendMsgTime  /* time for remote message sending */
   end if
end makeRemoteLink

/*********************************************************************
*                                                                    *
*      Make a link either remote or local                            *
*                                                                    *
*********************************************************************/

body procedure makeLink
   var destPen       : int
   var localOrRemote : int
   var sourceNode, destNode: int

   randint(localOrRemote, 1, 100)
```

```
  randint(destPen, 1, noOfMutators)
                                /* choose a random destination processor */
  randint(sourceDepthLevel,(round(ln(lastAddr)/ln(2.0)) div 3),
          round(ln(lastAddr)/ln(2.0)) + 2)
                        /* choose a random depth, at least three levels deep */

  if not((localOrRemote >= localOrRemoteLowerLimit) and
         (localOrRemote <= localOrRemoteUpperLimit) and
         (destPen not= sourcePen)) then
     destPen := sourcePen
  end if

  if sourcePen = destPen then
    makeLocalLink(sourcePen)            /* call local make link routine */
  else
    makeRemoteLink(sourcePen,destPen)   /* call remote make link routine */
  end if
end makeLink

/*********************************************************************
*                                                                    *
*       break link between two given cells                           *
*                                                                    *
*********************************************************************/

body procedure breakLink
  bind var states to state(sourcePen)
  var leftOrRight : int
  var sourceNode : int
  randint(sourceDepthLevel,(round(ln(lastAddr)/ln(2.0)) div 3),
          round(ln(lastAddr)/ln(2.0)) + 2)
                        /* choose a random depth, at least three levels deep */
  destDepth   := 0
  sourceNode  := destRandomWalk(states.root, sourcePen)
                        /* identify an edge for breaking */
  if ((sourceNode not= null) and
      (((states.memory(sourceNode).leftAddr not= null) and
      (states.memory(sourceNode).leftPen not= null)) or
      ((states.memory(sourceNode).rightAddr not= null) and
      (states.memory(sourceNode).leftPen not= null)))) then

     randint(leftOrRight, left, right)
     if (leftOrRight = left) then
        if (states.memory(sourceNode).leftAddr not= null) then

           if (states.memory(sourceNode).leftPen = sourcePen) then
              noOfLocalBreakLinks(sourcePen)  += 1
           else
              noOfRemoteBreakLinks(sourcePen) += 1
           end if

           states.memory(sourceNode).leftPen :=  null
           states.memory(sourceNode).leftAddr := null
           states.clock += breakLocalLinkTime     /* advance processor clock */

        elsif (states.memory(sourceNode).rightAddr not= null) then

           if (states.memory(sourceNode).rightPen = sourcePen) then
              noOfLocalBreakLinks(sourcePen)  += 1
           else
              noOfRemoteBreakLinks(sourcePen) += 1
           end if

           states.memory(sourceNode).rightPen  := null
```

```
            states.memory(sourceNode).rightAddr := null
            states.clock += breakLocalLinkTime     /* advance processor clock */
         end if
      end if

      if (leftOrRight = right) then
         if (states.memory(sourceNode).rightAddr not= null) then

            if (states.memory(sourceNode).rightPen = sourcePen) then
               noOfLocalBreakLinks(sourcePen)  += 1
            else
               noOfRemoteBreakLinks(sourcePen) += 1
            end if

            states.memory(sourceNode).rightPen  :=  null
            states.memory(sourceNode).rightAddr := null
            states.clock += breakLocalLinkTime     /* advance processor clock */

         elsif (states.memory(sourceNode).leftAddr not= null) then

            if (states.memory(sourceNode).leftPen = sourcePen) then
               noOfLocalBreakLinks(sourcePen)  += 1
            else
               noOfRemoteBreakLinks(sourcePen) += 1
            end if

            states.memory(sourceNode).leftPen  := null
            states.memory(sourceNode).leftAddr := null
            states.clock += breakLocalLinkTime     /* advance processor clock */
         end if
      end if
   end if
end breakLink

/*************************************************************************
*                                                                        *
*       Change processor phase                                           *
*                                                                        *
*************************************************************************/

body procedure changePePhase
   bind var states to state(pen)
   states.pePhase                  := nextPhase(states.pePhase)
   assert (states.pePhase = msgs(inMsg).phase)
   states.noLgcsDoneInCurrentPhase := 0
   lgcPpsMsgsSentInPhase(pen)      := false
end changePePhase

/*************************************************************************
*                                                                        *
*       Determine next phase colour                                      *
*                                                                        *
*************************************************************************/

body function nextPhase
   case presentPhase of
      label colour.red    : result colour.blue
      label colour.blue   : result colour.green
      label colour.green  : result colour.yellow
      label colour.yellow : result colour.red
   end case
end nextPhase

/*************************************************************************
```

```
*                                                                    *
*        Determine previous phase colour                             *
*                                                                    *
**********************************************************************/

body function previousPhase
  case presentPhase of
    label colour.red    : result colour.yellow
    label colour.blue   : result colour.red
    label colour.green  : result colour.blue
    label colour.yellow : result colour.green
  end case
end previousPhase

/**********************************************************************
*                                                                    *
*       Send acknowledgement for new please paint message            *
*                                                                    *
**********************************************************************/

body procedure sendNewPaintAck
  var tempSource, tempDest : int
  var reply  : pointer to msgs
  tempSource := msgs(inMsg).source
  tempDest   := msgs(inMsg).destination
  const pen  := msgs(inMsg).destination
  bind var states to state(pen)

  reply := inMsg
  states.clock             += msgAckTime
  msgs(reply).class        := msgClass.newPaintAck
  msgs(reply).destination  := tempSource
  msgs(reply).source       := tempDest
  msgs(reply).time         := states.clock
  mailBox.send(reply)
end sendNewPaintAck

/**********************************************************************
*                                                                    *
*       Send acknowledgement for lgc please paint message            *
*                                                                    *
**********************************************************************/

body procedure sendLgcPaintAck
  var tempSource, tempDest : int
  tempSource := msgs(inMsg).source
  tempDest   := msgs(inMsg).destination
  var node   := msgs(inMsg).address
  const pen  := msgs(inMsg).destination
  bind var states to state(pen)
  var reply  := inMsg

  if (states.memory(node).phase not= nextPhase(msgs(inMsg).phase)) then
     states.memory(node).phase  := msgs(inMsg).phase
  end if

  states.clock             += msgAckTime
  msgs(reply).class        := msgClass.lgcPaintAck
  msgs(reply).destination  := tempSource
  msgs(reply).source       := tempDest
  msgs(reply).time         := states.clock
  mailBox.send(reply)
end sendLgcPaintAck
```

```
/*************************************************************************
*                                                                        *
*       Send an acknowledgement for a read message                       *
*                                                                        *
*************************************************************************/

body procedure sendReadReply
  const pen := msgs(inMsg).destination
  bind var states to state(pen)
  var node := msgs(inMsg).address
  var reply := inMsg

  if (states.memory(node).phase not= nextPhase(msgs(inMsg).phase)) then
      states.memory(node).phase  := msgs(inMsg).phase
  end if

  states.clock             += msgAckTime
  msgs(reply).class        := msgClass.readReply
  msgs(reply).source       := msgs(inMsg).destination
  msgs(reply).destination  := msgs(inMsg).source
  msgs(reply).time := states.clock
  mailBox.send(reply)
end sendReadReply

/*************************************************************************
*                                                                        *
*       Receive an acknowledgement for a remote read message             *
*                                                                        *
*************************************************************************/

body procedure receiveReadReply
  var pen := msgs(inMsg).destination
  bind var states to state(pen)
  states.clock  += receiveMsgTime
  states.readsOut -= 1
  mailBox.bufferManager(inMsg,freeBuf)
end receiveReadReply

/*************************************************************************
*                                                                        *
*       Receive an acknowledgement for a new paint message               *
*                                                                        *
*************************************************************************/

body procedure receiveNewPaintAck
  var pen := msgs(inMsg).destination
  bind var states to state(pen)
  assert (states.newPpsOut > 0)
  states.clock  += receiveMsgTime
  states.newPpsOut -= 1
  if ((states.newPpsOut = 0) and not(readyToStartMutate(pen))) then
     readyToStartMutate(pen) := true
  end if
  mailBox.bufferManager(inMsg,freeBuf)
end receiveNewPaintAck

/*************************************************************************
*                                                                        *
*       Receive an acknowledgement for a lgc paint message               *
*                                                                        *
*************************************************************************/

body procedure receiveLgcPaintAck
  var pen := msgs(inMsg).destination
```

```
  bind var states to state(pen)
  assert (states.lgcPpsOut > 0)
  states.clock  += receiveMsgTime
  states.lgcPpsOut -= 1
  if (states.lgcPpsOut = 0) then
     readyToStartLgc(pen) := true
  end if
  mailBox.bufferManager(inMsg,freeBuf)
end receiveLgcPaintAck

/*************************************************************************
*                                                                        *
*       Receive an acknowledgement for a lgc start message               *
*                                                                        *
*************************************************************************/

body procedure receiveAckLgcStart
  var pen := msgs(inMsg).destination
  bind var states to state(pen)
  assert (states.lgcStartMsgSent)
  states.receivedAckLgcStart := true
  states.clock += receiveMsgTime
  mailBox.bufferManager(inMsg,freeBuf)
end receiveAckLgcStart

/*************************************************************************
*                                                                        *
*       Receive an acknowledgement for a lgc done message                *
*                                                                        *
*************************************************************************/

body procedure receiveAckLgcDone
  var pen := msgs(inMsg).destination
  bind var states to state(pen)
  assert (states.lgcDoneMsgSent)
  states.receivedAckLgcDone := true
  states.clock += receiveMsgTime
  mailBox.bufferManager(inMsg,freeBuf)
end receiveAckLgcDone

/*************************************************************************
*                                                                        *
*        Random walk through graph to identify a source node             *
*                                                                        *
*************************************************************************/

body function sourceRandomWalk
  bind var states to state(pen)
  var leftOrRight : int

   sourceDepth += 1

   if (root not= null) then
      if (sourceDepth > sourceDepthLevel) then
         result root
      else
         randint(leftOrRight, left, right)

         if (leftOrRight = left) then
            if (states.memory(root).leftAddr = null) then
               result (root)
            elsif (states.memory(root).leftPen = pen) and
                  (states.memory(root).leftPen not= null) then
                   result (sourceRandomWalk(states.memory(root).leftAddr, pen))
```

```
            elsif (states.memory(root).rightPen = pen) and
                  (states.memory(root).rightPen not= null) then
                   result (sourceRandomWalk(states.memory(root).rightAddr, pen))
            else
                   result null
            end if
         end if

         if (leftOrRight = right) then
            if (states.memory(root).rightAddr = null) then
               result (root)
            elsif (states.memory(root).rightPen = pen) and
                  (states.memory(root).rightPen not= null) then
                   result (sourceRandomWalk(states.memory(root).rightAddr, pen))
            elsif (states.memory(root).leftPen = pen) and
                  (states.memory(root).leftPen not= null) then
                   result (sourceRandomWalk(states.memory(root).leftAddr, pen))
            else
                   result null
            end if
         end if

      end if
   else
      result null
   end if
end sourceRandomWalk

/*************************************************************************
*                                                                        *
*        Random walk through graph to identify a destination node        *
*                                                                        *
*************************************************************************/

body function destRandomWalk
  bind var states to state(pen)
  var leftOrRight : int
  destDepth += 1

  if (root not= null) then
     if (destDepth > sourceDepthLevel) then
        result (root)
     else
        randint(leftOrRight, left, right)
        randint(destDepthLevel,(sourceDepthLevel div 8),
                (sourceDepthLevel div 1.3))

        if (leftOrRight = left) then
           if ((states.memory(root).leftAddr = null) and
               (destDepth > destDepthLevel)) then
                 result (root)
           elsif (states.memory(root).leftPen = pen) and
                 (states.memory(root).leftPen not= null) then
                  result (destRandomWalk(states.memory(root).leftAddr, pen))
           elsif (states.memory(root).rightPen = pen) and
                 (states.memory(root).rightPen not= null) then
                  result (destRandomWalk(states.memory(root).rightAddr, pen))
           else
                  result null
           end if
        end if

        if (leftOrRight = right) then
           if ((states.memory(root).rightAddr = null) and
```

```
                (destDepth > destDepthLevel)) then
                 result (root)
            elsif (states.memory(root).rightPen = pen) and
                  (states.memory(root).rightPen not= null) then
                   result (destRandomWalk(states.memory(root).rightAddr, pen))
            elsif (states.memory(root).leftPen = pen) and
                  (states.memory(root).leftPen not= null) then
                   result (destRandomWalk(states.memory(root).leftAddr, pen))
            else
                   result null
            end if
          end if

      end if
    else
       result null
    end if
end destRandomWalk

end mutator
```

```
/*********************************************************************
*********************************************************************
*       This module is in file 'master.t'                           *
*       Module for master processor                                 *
*                                                                   *
*********************************************************************/

parent "nonstop.t"              /* include this module in 'nonstop.t' file */

stub module master
        import(msgs, centralLgcRecords, lgcStat, state,
               colour, processGcMsgTime, msgClass, mailBox,
               noOfMutators, masterPhase, mutator, getBuf, forcesSent,
               averageNoOfLgcsLimit)
        export (replyLgcStart, replyLgcDone, checkPeStatus)

        procedure replyLgcStart(var inMsg: pointer to msgs)
        procedure replyLgcDone(var inMsg: pointer to msgs)
        procedure checkPeStatus

end master

body module master

        var nextForceTime           : int := 400

/*********************************************************************
*                                                                    *
*       Reply to an lgc start message                                *
*                                                                    *
*********************************************************************/

body procedure replyLgcStart
  var updatedLgcStatus : lgcStat
  var pen := msgs(inMsg).source
  var phase := msgs(inMsg).phase
  bind var masterPe to state(0)
  bind lgcRec to centralLgcRecords(pen)

  if not(phase = mutator.previousPhase(masterPe.pePhase)) then
     assert ((lgcRec = lgcStat.noStart) or (lgcRec = lgcStat.oneDone) or
             (lgcRec = lgcStat.someDone))
     case lgcRec of
       label lgcStat.noStart  : updatedLgcStatus := lgcStat.oneStart
       label lgcStat.oneDone  : updatedLgcStatus := lgcStat.doneGoing
       label lgcStat.someDone : updatedLgcStatus := lgcStat.doneGoing
     end case
     lgcRec := updatedLgcStatus
     masterPe.clock              += processGcMsgTime
     msgs(inMsg).class           := msgClass.ackLgcStart
     msgs(inMsg).source          := 0
     msgs(inMsg).destination     := pen
     msgs(inMsg).time            := masterPe.clock
     mailBox.send(inMsg)
  else
     /* ignore previous phase lgc start message */
     msgs(inMsg).class           := msgClass.ackLgcStart
     msgs(inMsg).source          := 0
     msgs(inMsg).destination     := pen
     msgs(inMsg).time            := masterPe.clock
     mailBox.send(inMsg)
  end if
end replyLgcStart
```

```
/*********************************************************************
*                                                                    *
*       Respond to a lgc done message                                *
*                                                                    *
*********************************************************************/

body procedure replyLgcDone
  var updatedLgcStatus : lgcStat
  var pen := msgs(inMsg).source
  var phase := msgs(inMsg).phase
  bind var masterPe to state(0)
  bind lgcRec to centralLgcRecords(pen)

  if not(phase = mutator.previousPhase(masterPe.pePhase)) then
     assert ((lgcRec = lgcStat.oneStart) or (lgcRec = lgcStat.doneGoing))
     case lgcRec of
       label lgcStat.oneStart  : updatedLgcStatus := lgcStat.oneDone
       label lgcStat.doneGoing : updatedLgcStatus := lgcStat.someDone
     end case
     lgcRec := updatedLgcStatus
     masterPe.clock             += processGcMsgTime
     msgs(inMsg).class          := msgClass.ackLgcDone
     msgs(inMsg).source         := 0
     msgs(inMsg).destination    := pen
     msgs(inMsg).time           := masterPe.clock
     mailBox.send(inMsg)
  else
     /* ignore previous phase lgc done message */
     msgs(inMsg).class          := msgClass.ackLgcDone
     msgs(inMsg).source         := 0
     msgs(inMsg).destination    := pen
     msgs(inMsg).time           := masterPe.clock
     mailBox.send(inMsg)
  end if
end replyLgcDone

/*********************************************************************
*                                                                    *
*       check processor's status                                     *
*                                                                    *
*********************************************************************/

body procedure checkPeStatus
  bind var masterPe to state(0)
  var forceMessage : pointer to msgs
  var phaseMessage : pointer to msgs
  var lgcsDone     : int := 0            % check if current phase is done */
  var averageNoOfLgcsInPhase : real
  var countOfLgcs  : int := 0

  for i:1..noOfMutators                  % examine mutator status records */
    bind var states to state(i)
    if (centralLgcRecords(i) = lgcStat.someDone) or
       (centralLgcRecords(i) = lgcStat.doneGoing) or
       (centralLgcRecords(i) = lgcStat.oneDone) then
       lgcsDone += 1
       countOfLgcs := countOfLgcs + states.noLgcsDoneInCurrentPhase
    end if
  end for

  averageNoOfLgcsInPhase := (countOfLgcs / noOfMutators)

  if (lgcsDone >= round(noOfMutators * 0.8) and (not forcesSent)
     and (averageNoOfLgcsInPhase  >= averageNoOfLgcsLimit)) then
```

```
    for i:1..noOfMutators
      if (centralLgcRecords(i) = lgcStat.noStart) then
         mailBox.bufferManager(forceMessage,getBuf)
         msgs(forceMessage).class              := msgClass.lgcForce
         msgs(forceMessage).time               := masterPe.clock
         msgs(forceMessage).destination        := i
         msgs(forceMessage).source             := 0
         msgs(forceMessage).phase              := masterPe.pePhase
         put "sending force lgc message to  ", i
         mailBox.send(forceMessage)
      end if
    end for
    forcesSent := true
  end if

  if ((lgcsDone = noOfMutators) and
     (averageNoOfLgcsInPhase  >= averageNoOfLgcsLimit)) then

    masterPhase                 := mutator.nextPhase(masterPe.pePhase)
    masterPe.pePhase            := masterPhase

    for i:1..noOfMutators
      centralLgcRecords(i)      := lgcStat.noStart
    end for

    for i:1..noOfMutators
      mailBox.bufferManager(phaseMessage, getBuf)
      msgs(phaseMessage).class          := msgClass.phaseChange
      msgs(phaseMessage).time           := masterPe.clock
      msgs(phaseMessage).destination    := i
      msgs(phaseMessage).source         := 0
      msgs(phaseMessage).phase          := masterPe.pePhase
      mailBox.send(phaseMessage)
    end for
    forcesSent := false
  end if
end checkPeStatus

end master
```