TECHNIQUES AND DATA STRUCTURES FOR PARALLEL RESOURCE MANAGEMENT

Jit Biswas

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-42 November 1987

TECHNIQUES AND DATA STRUCTURES

FOR

PARALLEL RESOURCE MANAGEMENT

APPROVED BY

SUPERVISORY COMMITTEE

J. C. Browne

Aloysius Mok

Jeff Brumfield

To Baba and Ma

TECHNIQUES AND DATA STRUCTURES

FOR

PARALLEL RESOURCE MANAGEMENT

by

Jit Biswas, B.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 1987

Acknowledgements

I would like to gratefully acknowledge the constant encouragement, support and great leeway given to me by my supervisor Dr J. C. Browne. I am also indebted to other members of my committee for their helpful suggestions from time to time. Dr Al Mok and Dr Clement Leung have been sharp critics and I shall always be grateful to them for goading me into seeking nothing but the best in terms of the quality of my work. In addition I would especially like to thank Dr Vipin Kumar who has been a believer and a supporter. He has also been very generous with his time and financial support. Dr Carla Ellis (of Duke University) has been very helpful in pointing me toward relevant literature on concurrent data structures, especially in the initial stages of my work. I really appreciate her guidance.

To Pradeep Jain and Raghu Ramakrishnan I am especially indebted for countless discussion sessions over innumerable cups of coffee. Pradeep's painstaking reading of my dissertation has greatly improved its readability. I also wish to express my gratitude to my colleagues of the Parallel Programming Group, especially Paul Suhler, Steve Sobek, Ashok Adiga, S. J. Kim, Mary McShea, Steve Eubank, Bob O'Dell, Nicolas Graner, T. T. Ramgopal and many others who have suffered through my Wednesday afternoon presentations. I would also like to thank Sanjay Deshpande and V. N. Rao for their suggestions. Finally, I wish to thank my roommate Avi Saha for his support and encouragement during my travails as a graduate student.

This thesis is dedicated to my parents.

TECHNIQUES AND DATA STRUCTURES

FOR

PARALLEL RESOURCE MANAGEMENT

Publication No. ____________

Jit Biswas, Ph.D.

The University of Texas at Austin, 1987

Supervising Professor: James C. Browne

The problem of managing the resources of a highly parallel system is viewed as the problem of simultaneously updating data structures that hold system state. We approach this problem in an abstract data type framework. Simultaneous update may be attained in two ways: by decomposing abstract states into components and allowing operations to concurrently transform the state of these components in a controlled manner, and by weakening the specification of abstract data types in ways that are acceptable to entities using instances of abstract data types.

This thesis contributes to parallel resource management in both ways. First, we have considered management of system state for computation structures consisting of arrays of computations that differ only in indexing parameters. We have proposed simple decompositions of the externally visible state into simultaneously updatable

components. Second, we have considered the management of system state for weakened priority queues. The two priority structures proposed in this thesis, a concurrent heap and a software banyan, have been found to be efficient and effective.

We have, in addition, contributed in the area of language tools for computations that utilize predefined abstract data type implementations. A mechanism for abstract data type definition is presented. To promote simultaneity of update we have defined a significant extension of the linguistic construct of path expressions and used it as a basis for defining implementation of sequencing within abstract data types. The main advantage of using extended path expressions is that in addition to synchronization requirements, binding of activities to object decompositions may be specified, along with runtime consistency checking, while leaving the object implementation to the underlying system. We have developed algorithms for the automatic synthesis of sequencing and synchronization code.

A task level data flow language designed and implemented by us has provided a context and a testbed for ideas presented in this thesis.

Table of Contents

Chapter 1 - Introduction

1. The problem

This research addresses the problem of resource management, or the management of system state, for highly parallel large scale multiprocessor systems. The allocation of a resource to a computation, or alternatively the assignment of a computation to a resource, results in a change of state of the system. In order to perform correctly, a data structure that holds system state must obey consistency relations just as any other data structure. In executing the functionality of the state management one must avoid serializing bottlenecks resulting from update of system data structures, since this results in performance bottlenecks.

2. The solution approach

The obvious approach is to apply parallelism to resource management or the management of system state. The requirement is to decompose the data structures defining the state of the system so that the partitions of the decomposition can support parallel or simultaneous update, and to establish algorithms which execute simultaneous updates with maintenance of the consistency specifications for the system state data structures. We approach this problem in an *abstract data type* framework. We define semantics and consistency properties for the data structures which maintain system state such that these data structures can be partitioned and parallel algorithms defined on them. A fundamental guideline which is the basis for our work is the separation of the external specifications from the implementation specifications for *objects* (abstract data type instances) implementing system state management.

3. Results and contributions

We have targeted two particular problem domains of resource management. The first is management of system state for computation structures consisting of arrays of computations that differ only in indexing parameters. In this case we have proposed simple decompositions of the externally visible state into simultaneously

updatable components. The second is the management of priority structures. Parallel abstract data types, including data structures and parallel algorithms implementing priority queues supporting simultaneous update, are defined and characterized. We have also contributed in the area of language tools for computations that utilize predefined abstract data type implementations. An illustrative framework for an abstract data type definition facility is defined. We identify a class of computations which support implementation of the models of computation directly from current programming languages. To promote simultaneity of update we have defined a significant extension of the linguistic construct of path expressions and used it as a basis for defining implementation of sequencing within abstract data types. The main advantage of using extended path expressions is that in addition to synchronization requirements, binding of activities to object decompositions may be specified, along with runtime consistency checking, while leaving the object implementation to the underlying system. We have developed algorithms for the automatic synthesis of sequencing and synchronization code. The two priority structures proposed in this thesis, a concurrent heap and a software banyan, have been found to be efficient and effective. Variants of both algorithms can be easily implemented in specialized hardware, gaining additional performance.

Chapter 2 - Background

1. Highly Parallel Architectures

A highly parallel architecture is a MIMD [Flynn '66] multiprocessor architecture that has no inherent limitation upon the number of processors that it can have. These architectures have been traditionally divided into three broad classes. The first class consists of *shared memory multiprocessors* with high speed interconnection networks. Examples are the Butterfly [Butterfly '85], RP3 [Pfister '85], NYU Ultracomputer [Gottlieb '83b] and Cedar [Gajski '83]. The second class is that of distributed memory architectures, where no memory is shared by processors, and information must be sent from one processor to another across hardware channels. The hypercube machine first built at Caltech [Seitz '85] and commercialized by Intel [Intel '87] and other manufacturers are examples of this class. Finally, there are hybrid † architectures, that support memory sharing within clusters of processors and messaging passing, or *messaging* between clusters. Examples are C.mmp [Wulf '72], Cm* [Swan '77] and FLEX [Matelan '85].

The data structures and algorithms described in this thesis apply to resource management for highly parallel architectures in all three above categories, though our implementation and some of our illustrative examples deal with machines in the first category.

2. Guidelines for highly parallel resource management

In our view, the following guidelines are essential for doing resource management for highly parallel architectures.

i) *Multiple serialization points*. Traditional views of resource management have been centralized. This is because, a resource has always been modelled as a centralized entity with global critical sections. For example, a monitor

† Sometimes researchers prefer to classify some of the shared memory multiprocessors as being hybrid, because there is a dichotomy in the amount of time taken between local and nonlocal memory accesses.

[Hoare '74] guarantees that no two processes shall be inside it at the same time, thereby bringing about a global serialization point. Arvind [Arvind '77, '83] breaks away from this restriction with the use of dataflow resource managers, but these managers are defined within the framework of dynamic, instruction level dataflow. We would prefer to have a solution that applies in more general situations.

ii) *Multiple binding times*. Activities should be bound to the object of their computation only when such a binding is required. This can be at load time, if the object is frequently required, or at runtime, if the object is not frequently required or shared by several activities.

iii) *Patterns*. Frequently occurring patterns of object partitioning and mapping should be optimized and provided in libraries for programmers to use.

iv) *Database concurrency* for *multiprocessors*. The field of concurrency has been very well researched in distributed databases. Well known techniques exist for achieving simultaneous update for databases in a manner that preserves consistency and integrity constraints [Kung '80a]. These ideas may be adapted to achieve concurrency in data structures for resource management on highly parallel architectures.

In order to achieve the above guidelines we *partition* data structures that hold the state of computations. Partitioning obviously brings about multiple serialization points, since different activities may access different partitions at the same time in an overlapped manner. Binding activities to partitions of an object can be done at different times, to exploit performance advantages. To incorporate patterns, we develop control abstractions around *path expressions*, which are synchronization specification constructs. Often, it is beneficial to relegate non-urgent tasks of restoration of data structures, to *restructuring* activities, so that more immediate activities can proceed faster. This is a technique used in the preservation of database indexes. We demonstrate how this technique can be applied to resource management for highly parallel architectures.

Our work is thus centered around control and data abstractions. To put this work in perspective we now discuss the related literature.

3. Control and data abstractions

3.1. Specification of control dependencies

Programs written in sequential programming languages such as Fortran must be carefully analysed to extract dependency information [Kuck '81], to permit the generation of code that exploits features of the target machine. This paradigm of programming and code optimization is still the one with largest following among users of high performance computers.

An alternate paradigm, and one that makes the job of the compiler writer much simpler is to impose certain restrictions upon the constructs that a programmer is allowed to use, and thereby restrict the class of dependence graphs that will be generated at runtime. If these graphs have certain attractive properties, such as having only disciplined access to shared data structures, or having restrictions upon the kinds of cycles permitted, then it is straightforward to perform efficient resource management for them. Data flow languages, for example, VAL [Ackerman '79] Id [Arvind '78], and SISAL [McGraw '85], are so structured that once a program is written in these languages, the parallelism naturally falls out of the program. Another example of this paradigm is the SPMD (single program multiple data) programming style introduced in the programming of the RP3 multiprocessor [Darema '85]. In this approach, each program is written as a single Fortran program with special system calls for parallel constructs. Some of our work has been influenced by this approach and some of the tools that we propose are directly usable within this framework.

In addition, our work has focussed on an alternate paradigm, in that although we provide constructs to the programmer with which to compose programs, the schedulable units of computation are at the task level. The programmer programs in large grain units called tasks, (approximately at the level of subroutines in Fortran, or procedures in Pascal), that are the natural units of computation. We wish to exploit parallelism at the level of tens or hundreds of instructions, rather than that of a single instruction, as has been the case with traditional data flow. Many regular structured problems, can exploit parallelism at this level. For example, graph algorithms such as the maximum concurrent flow problem [Biswas '86] and problems from computational physics, such as the computation of trajectories of particle in motion in a plasma, [Biswas '87a], have identifiable parallelism at the level of blocks of instructions.

We have designed and implemented a simple graphical task level language called TDFL (Task-Level Data Flow Language) [Suhler '87]. The language provides a set of constructs that programmers are inclined to use frequently. Separation of concerns between the program's specification and its implementation is maintained, since the specification is a graph. Thus irrespective of the sequence in which the nodes of the graph fire, or compute, the outcome of a computation is always the same. Cycles in the program graphs must belong to a restricted class of allowable cycles. We have parallelized the scheduling of ready nodes, and other constructs in the language.

3.2. Specification of synchronization

The genealogy of path expressions start with regular path expressions [Campbell '74], which have *closed* semantics, in the sense that operations in an expression are assumed to execute in mutually exclusive mode, unless otherwise specified. These were succeeded by open path expressions, OPEs, [Campbell '77], that had unconstrained semantics. Thus an unrestricted number of parallel invocations of an operation on an object may be active simultaneously if no explicit restriction is mentioned in the associated OPE. It has been demonstrated [Oldehoeft '84] how to synthesize data flow resource managers of the same ilk as Arvind's, from OPEs, and [Headington '85] to incorporate predicates associated with these path expressions making them open predicate path expressions (OPPEs). The authors also demonstrate how to convert an OPPE into a network of controllers with data flow nodes implemented in a purely message based environment.

While writing and implementing programs for execution on multiprocessors, we see two levels at which there is the necessity for specifications. The first is at the level of user programming, i.e. when only what needs to be done is specified. The second is the specification of synchronization between user level programs; here we must specify the allowable sub-sequences of interleavings. For example the writing of a TDFL program is programming at the user level whereas programming the TDFL scheduler itself is at the second level. If we consider the writing of controllers or schedulers as a programming task (which it is), we immediately feel the necessity of automating this task. Synchronization specifications can be provided, to automate the task of code generation for controllers required for the language. Thus if the language has a provision for setting up mutual exclusion dependencies, then it should be possible to specify an appropriate mutual exclusion constraint at a very high level, by means of an appropriate language, and have code for the preservation of the mutual exclusion dependency, generated automatically. Automation of code generation can also be made to apply to other areas such as binding and consistency constraints. In this thesis we examine an extension to path expressions as a means of specifying synchronization, binding and consistency constraints to be satisfied by implementations.

3.3. Data abstraction

The guidelines of section 2 can be met by partitioning implementations of abstract data types (*adt* s) [Guttag '80]. There are two types of issues here, namely those related to formal properties of *adt* s and those related to *adt* implementations. Abstract data types are defined by giving them semantics and abstract properties which the implementation must then satisfy and preserve.

Data structures have always been thought of in sequential settings whereas databases have always been thought of in concurrent settings. The key difference is in the application. Most of the earlier work with abstract data types was centered around the abstract semantics of types and their operations. The application of objects such as stacks and queues were in strictly sequential domains. In contrast, databases are shared by definition, and different applications require different kinds of consistency or serializability requirements from their databases.

New application areas such as databases for computer aided design [Bancilhon

'85], atomic data types [Weihl '84], and distributed operating systems [Rashid '86] have introduced a class of objects that we call *simultaneously updatable objects*, with formal properties that are quite different from those of conventional *adt* s. Computations using these objects have consistency requirements that are based on the semantics of *adt* s and *adt* computations rather than syntactic serializability requirements that are to be found in traditional database concurrency control [Papa '86].

Various issues have emerged as being important in characterizing this new class of objects. These are atomicity properties [Weihl '84], object implementation states and correctness of implementations [Shasha '87], new notions on interleavings in computations concerning concurrent objects [Herlihy '87], semantics of conflict and commutativity in shared abstract data types [Schwarz '83] and implementation of shared abstract data types [Bottos '85].

Database concurrency control has furnished a rich body of results, which can be used to characterize formal properties and develop efficient implementations of simultaneously updatable objects. Among the most significant contributions from this area (in relation to our work), are contributions in locking protocols, and those in the concurrent manipulation of data structures used to maintain large database indexes. We briefly discuss the relevant results. The two phase locking protocol [Easwaran '76] is a simple protocol for locking data items in such a way that serializability is preserved. Tree locking protocols [Silberschatz '80], use a simple strategy of progressing down a tree-structured collection of data items, preserving serializability of the resulting computation without locking the entire tree. Concurrency can be enhanced [Korth '83] by granularizing a database in such a way that locks at higher levels of granularity may be released once those at lower levels are obtained.

The results mentioned above have found application in designing concurrent data structures for database indexes. Two techniques predominate, the link based technique and the lock coupling technique. The link based technique [Kung '80b] relies on the property that a very small atomic state change is all that is necessary to preserve the structure and semantics of a binary search tree, for an unrestricted set of input operations. The basic idea in lock based techniques [Bayer '77] is a direct application of the tree protocol, called lock coupling, i.e. the paradigm that a parent lock (i.e. the lock at a parent node), is released only when a child lock is obtained. These algorithms and their derivatives have been proposed for binary search trees [Manber '83], AVL trees [Ellis '80b] and B Trees [Lanin '86].

Most of the above work concerns software structures. The balanced cube object [Dally '86] is an attempt at reflecting the physical structure of a hardware architecture in a search structure. This object has been proposed as a concurrent object that scales up and down with the size of the hypercube upon which it is currently mapped. Concurrent smalltalk has been used by Dally as a programming language for his object implementation. In our work we demonstrate how general purpose programming languages can be used in similar situations.

The issue of efficient abstract data types has been analysed from the perspective of distributed systems in the work of Bastani *et al* [Bastani '87]. In addition to the notions of *periodic* and *concurrent* maintenance, the authors also use invariants to specify properties of implementations of an abstract data type. If the strongest (weakest) invariant is satisfied then the performance is best (worst). In our approach we also propose weakening of specifications to introduce performance gains in implementations of abstract data types.

Simultaneous update of data structures that maintain system state information is the focus of our work. We have considered two ways of obtaining simultaneous update, decomposition of externally visible or abstract states of an object, and weakening specifications of an object at external and internal levels. In chapter 3 we present our terminology and address the decomposition aspect of simultaneous update. We introduce an event based model consisting of objects and activities, where abstract specifications are provided for each object. Internal or object implementation states are characterized in terms of decomposition functions upon abstract states. Based upon these definitions we develop the notion of correct external histories and internal histories. In chapter 4 we develop tools for obtaining simultaneous update. We propose an abstract data type definition facility and illustrate its use with examples. The augmented synchronization specification language of extended path expressions is presented in this chapter and code synthesis algorithms provided.

The issue of weakened specifications is addressed in the remaining chapters of this thesis. We identify a specific object class namely priority structures, and characterise under what conditions such structures may be weakened for performance gains. We outline serial and concurrent algorithms for managing priority structures, demonstrating such weakening at external and internal levels. The proposed structures are analysed in chapter 6.

The granularization approach taken in this thesis was inspired by related research in concurrent management of index structures for databases [Lanin '86, Kung '80b, Ellis '85, '83, '80a, '80b]. We know of no existing parallel algorithms for priority structures for general purpose multiprocessors. Specialized hardware has been proposed for priority applications and dictionary machines [Leiserson '79]. This topic is discussed further in section 4 of chapter 6 where we develop systolic structures for priority applications and contrast these with those existing in the literature.

Chapter 3 - Terminology and a model for simultaneous update

1. Introduction

Informally two activities are said to *simultaneously update* an object if their operations upon that object are overlapped in time. For example two activities simultaneously update a file object if their updates on different records overlap in time. Thus simultaneous update is achievable by partitioning an object into smaller granularity units (records, for example) and overlapping updates to these units in conformance with consistency specifications for updates of records.

In this chapter we define simultaneous update using terminology from a model based upon objects and activities. We are mainly concerned with the definition of a model of computation which separates external specifications from internal specifications so that simultaneous update can be tolerated in the external specifications. We consider two ways of achieving simultaneity of update. The first is *partitioning* or decomposition of the externally visible state of an object so that the components may be concurrently updated. The second is the introduction of *weakened specifications* at both external and internal levels. In this chapter and the next we look at the former. In chapter 5 we address the issue of weakened specifications.

First we present in section 2, an analogy using a bank account example to provide an intuitive feel for what we mean by partitioning. In the succeeding section we characterize simultaneous update of objects in terms of a *system model* for objects and their implementations. We define object specifications using a state machine model with partial functions and annotations using a standard event based model [Weihl '84]. Then we define object implementation states in a similar manner as that introduced by Shasha [Shasha '87] in the context of search structures. We show how an abstract state can be decomposed into a set of components at the implementation level such that these components are concurrently updatable. We generalize the composition of these constituent components back to form the abstract state that they implement. Internal histories are also characterized in section 3 and it is shown how an external history may be obtained from an internal history. We compare our system model with the Karp and Miller model [Karp '69].

In section 4 we illustrate our model by the bank account example and two more examples and examine their performance in section 5. We examine the correctness of internal histories and address the role of a scheduler for an object in the process of witnessing and permitting overlaps. To provide a concrete illustration of our ideas we briefly discuss a construct of a task level data flow language and demonstrate how computations resulting from programs written in this language perform simultaneous update. Issues that are related to partitioning are *time of update*, to take advantage of performance tradeoffs on parallel architectures and *restructuring*, to perform asynchronous maintenance of object implementations. We characterize *multiple binding times* and *restructuring* using the terminology developed.

2. An analogy - The joint account

Let us say that Mary and Bob hold a joint account with savings and checking components (see Fig 2.1). We assume that both accounts are protected by an automatic overdraft scheme whereby money is transferred automatically from one to the other in case of inadequate funds in the latter. Let us assume that the bank has been informed that Bob's transactions will always be associated with the checking account and Mary's with the savings account. Initially the checking account contains $ 0 and the savings account contains $ 0. The following history is possible: (Events occur chronologically with the i^{th} event occurring at time t_i.)

t_1 Bob requests withdrawal of $ 50 from the joint account.
t_2 Mary requests deposit of $ 50 into the joint account.
t_3 A bank employee transfers $ 50 from savings to checking.
t_4 Bob gets $ 50.
t_5 Mary gets acknowledgement.

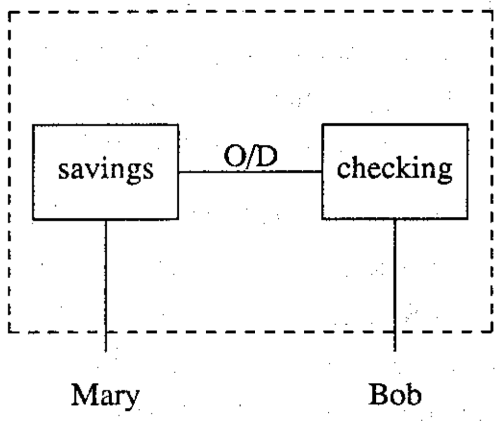


Fig 2.1: The joint account object

Notice that at the time Bob requested withdrawal there was no money in either component of the joint account. However at the time the bank employee performed the transfer, Mary had already requested a deposit and it had been internally recorded. Admittedly the manner in which partitioning is done in the above example is somewhat contrived. However, this example illustrates three points. Firstly the joint account is an object which is simultaneously updated at an abstract level, although internally each update is channelized to a different partition of the object. Within a partition updates are not simultaneous. Secondly the *binding* of account holders to account components is statically performed, i.e. at creation time of the joint account object. We may have a dynamic or *runtime* binding of account holders by randomly picking an account component for every request. This illustrates time of binding. Finally we have a maintenance activity namely the "transfer" operation carried out by the bank employee. Such maintenance activities carry out *restructuring* of the internal state of a bank account in order to support the correct behaviour of the implementation of an account with respect to its specification. Informally this means a client must be able to withdraw money if sufficient funds are in either component of her (his) account.

3. System Model

Our system consists of activities and objects that interact through events. Objects are instances of abstract data types that have internal state and have external specifications that allow for simultaneous update. (The type definitions themselves are in objects called *managers* that support the *create* operation. We shall see an example of this in the application domain of work management in the next chapter.) A computation is a history of events involving activities and objects.

Formally, a *system* consists of a quadruple $< OB, AC, \sum, C >$, where $OB = \{x, y, \cdots\}$ is a finite set of objects, $AC = \{A, B, \cdots\}$ is a finite set of activities, $\sum =$ RES $\cup$ INV $\cup$ PARMS (described below) is the set of symbols that cause state transitions, and $C = \{c_1, c_2, \cdots\}$ is a finite set of components that are analogous to memory locations in the sense that they hold the state of object implementations. For each component $c_i \varepsilon C$ there is a domain D_i or set of possible values that the component can have.

3.1 Events, operations and histories

Activities are analogous to processes in that they are threads of control of finite duration that invoke operations on objects. We do not specify activities any further. The behaviour of our system is captured by a sequence of *events* that occur between activities and objects, called a *history*. An event may be an invocation or a termination event. An *invocation event* is a four tuple $< invok, args, act, obj >$ and a *termination event* is a four tuple $< resp, results, act, obj >$ where $act \, \varepsilon AC$, $obj \, \varepsilon OB$, $invok \, \varepsilon$ INV (the set of invocation symbols), $resp \, \varepsilon RES$ (the set of termination symbols), and $args$, $results \, \varepsilon$ PARMS (the set of parameter values that may be elementary values such as booleans and integers or structured values such as pairs of elementary values). If x is an object and A an activity of a particular event e, then e is said to *involve* A and x.

A *complete history* is a sequence of events of a system such that for every invocation event there is exactly one termination event that is matching, i.e. agreeing in the activity and object fields, the termination event appears after the invocation event, there are no other events involving the same activity between these two events[†], and there are no unmatched events.

[†] The requirement that there is no more than one outstanding invocation event at a time from a single activity arises from our application domain.

3.2 The serial specification of an object

The serial specification of an object is a state machine that describes the behaviour of that object in the absence of concurrency. Formally, for each object $x \in OB$ is associated a quadruple $< S_x, I_x, O_x, T_x >$, called the *serial specification* of x, where S_x is a set of states, I_x is a designated initial state, O_x is a set of operations and T_x is a transition function. $T_x : S_x \text{ X } O_x \rightarrow_p S_x$ is a partial function describing how the state changes with each permissible operation. The partial nature of T_x is specified with the help of preconditions discussed below. We shall now characterize the set of operations O_x.

An operation is a pair of events $<< invok, args, A, x >, < resp, results, A, x >>$, where the first event is an invocation event and the second event is a matching termination event. *Preconditions* in the specification of the transition function T_x are predicates associating the state, the arguments and the results in the form of a boolean valued function which must be satisfied for the transition to occur.

Example 3.1. Serial specification of the two element set object

Let us assume that we have an object $x \in OB$ that behaves as a set of values with the operations "put" and "get", where the universe consists of two elements "a" and "b". The serial specification of this object is as follows:

$S_x = \{ s_1, s_2, s_3, s_4 \}$
where $s_1 = \{\}, s_2 = \{a\}, s_3 = \{b\}, s_4 = \{a, b\}$

$I_x = s_1$

$O_x = \{ o_1, o_2, o_3, o_4, o_5 \}$
where $o_1 = << \text{get, NULL}, A, x >, < \text{ok, a}, A, x >>$
$o_2 = << \text{get, NULL}, A, x >, < \text{ok, b}, A, x >>$
$o_3 = << \text{get, NULL}, A, x >, < \text{sorry, NULL}, A, x >>$
$o_4 = << \text{put, a}, A, x >, < \text{ok, NULL}, A, x >>$
$o_5 = << \text{put, b}, A, x >, < \text{ok, NULL}, A, x >>$
and $A \in AC$

The partial function T_x is depicted in Fig 3.1. Not every operation is defined at every state in this diagram. Thus it represents a partial function. For the two element set object it is possible to enumerate the states and the operations exhaustively (except for the activity field). For more complex objects such exhaustive enumeration is infeasible and we shall introduce variable names and predicates on these variable names to denote entire sets of states and classes of operations.

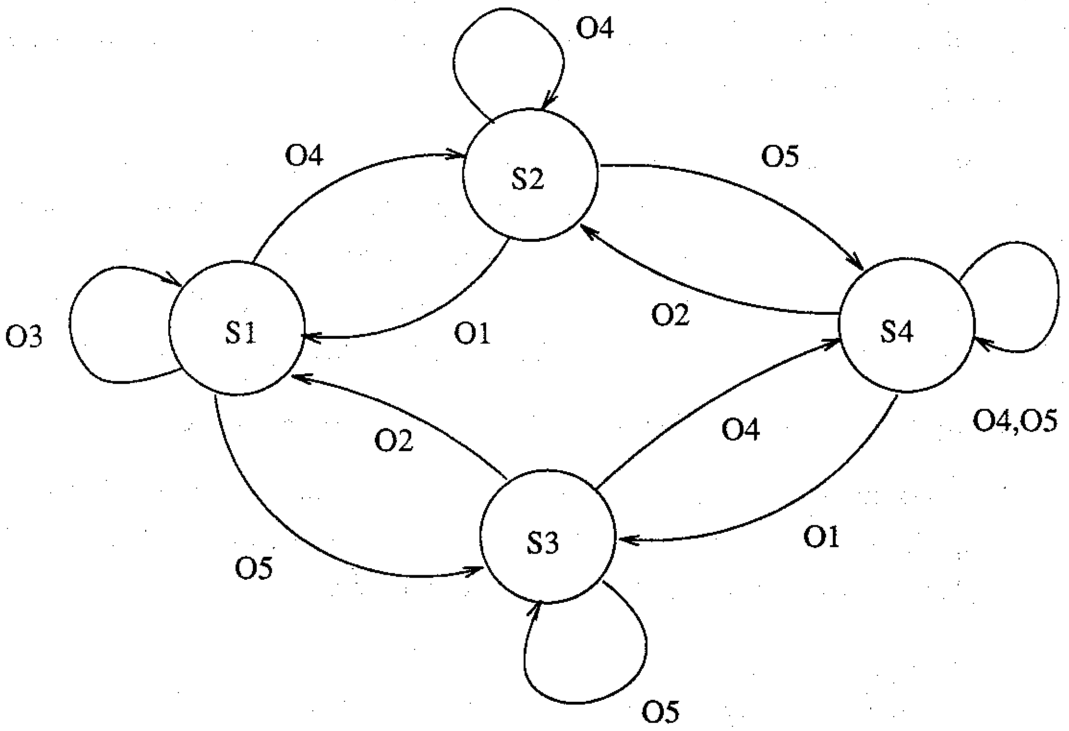


Fig 3.1: T_x for the two element set object x

3.3 Sequences of operations

The serial specification of an object may be extended to apply to sequences of operations in the usual way as follows: If *Seq* denotes a sequence of operations with *Last* (*Seq*) denoting the last (most recent) operation in the sequence and *Past* (*Seq*) denoting the previous sequence of operations before the last operation then for a given $s \in S_x$: T_x $(s, Seq) = T_x$ $(T_x$ $(s, Past(Seq)), Last(Seq))$ if T_x $(x,$

Past (*Seq*)) is defined, and is undefined otherwise.

Given a history H, the *projection* of H onto activity A (denoted H | A) is the sub-sequence of H restricted to only and all events involving activity A. Similarly the projection of H onto object x (denoted H | x) is the sub-sequence of H restricted to only and all events involving object x. If H is a history we use $<_H$ to denote the (total) ordering on its events.

Two operations $<inv_1, res_1>$ and $<inv_2, res_2>$ on an object x are said to *overlap* at x in history H, if *either* $inv_1 <_H inv_2 <_H res_1$ *or* $inv_2 <_H inv_1 <_H res_2$.

Simultaneous Update. Given activities A and B, and complete history H, we say A and B *simultaneously update* object x in history H, if there are two operations $o_1 = <inv_1, res_1>$ and $o_2 = <inv_2, res_2>$ that overlap in H | x, such that o_1 involves activity A and o_2 involves activity B.

3.4 Correct histories

The consistency constraint of systems using objects is defined as a predicate upon the states of all the objects. In the most general case [Kung '83], such a constraint is a subset of the cartesian product of all the object state domains. We focus on a class of such consistency constraints, that can be put in conjunctive normal form [Korth '85], where each conjunct involves only a single object.

Thus, given objects $\{x_1, x_2, \cdots, x_n\}$ our consistency constraint is of the form CC = $\{cc_1 \wedge cc_2 \wedge \cdots \wedge cc_n\}$ where cc_i involves only object x_i. Each conjunctive term is stated as an invariant on the corresponding object state, which must be maintained through each state transition in the serial specification of the object. A history involving several objects is equivalent to an arbitrary interleaving of the set of projections of the history upon each object. For example, let us suppose that Bob and Mary have two joint accounts, with Bob requesting transactions before Mary in some (correct) history from both accounts. A history, in which Bob precedes Mary in one and succeeds Mary in the other, is deemed to be an equivalent and correct history. The benefit of restricting ourselves to this class of computations is that we do not have to worry about preservation of consistency *across* objects.

Since we are not interested in preserving consistency across multiple objects, henceforth we shall restrict our attention to histories that involve a single object. Thus unless otherwise specified, whenever we mention a history H it will be understood that this history involves a single object x, and possibly multiple activities A, B and so on.

Given a history H a *reordering transformation* of H is a new history H$'$, that is obtainable from H by permuting the events in H, in such a way that

a) the ordering of events in each operation in H is preserved in H$'$,

b) there are no simultaneously updating operations in H$'$ and

c) if a termination event e_1 precedes an invocation event e_2 in H, it does the same in H$'$.

Since H$'$ does not contain any simultaneously updating operations the sequence of events in H$'$ may be transformed into a sequence of operations, *Seq*, by considering successive invocation - response pairs.

A complete history H involving x is *correct* if there is a history H$'$ such that H$'$ is a reordering transformation of H, and the sequence of operations *Seq* composed of the sequence of successive pairs of events in H$'$ is consistent with the serial specification of x. Thus a complete history H involving an object x is correct if a reordering can be found for H that satisfies the serial specification of x. The notion of reordering is also present in the work of Herlihy and Wing [Herlihy '87] who call such histories *linearizable*.

3.5 Object implementation states and internal histories

An object's *implementation* realizes its abstract specification [Shasha '87]. Thus if the object's transition function maps state s to state s' with a certain set of result values for a given operation, then the implementation maps some underlying representation of state s to some underlying representation of state s' with the same result values. The issue of partitioning primarily deals with an object's implementation.

It is beneficial (as we saw in the joint accounts example) to visualize objects as being coherent entities, while allowing simultaneous update in their implementations. Accordingly, we define an object implementation state to include the notion of *components* of an abstract state of an object. Components may be thought of as constituents of a *decomposition* of the externally visible state of an object that may be simultaneously updated, in a manner that will still preserve the specifications of the object. Components may also be thought of as granules of data that must be treated as unit entities, or sets of unit entities for synchronization purposes.

Formally, corresponding to each state $s \varepsilon S_x$ for some object x is a quadruple $< Dec, Cval, Dec^{-1}, Crel >$, called the *object implementation state* of s.

- * $Dec : S_x \rightarrow 2^C$ is a function assigning to each state s in S_x a subset of components from C, that is called the *decomposition* of the state s, denoted $Dec\ (s)$.
- * Let $Dec\ (s) = \{\ c_{i_1}, c_{i_2}, \cdots c_{i_k}\ \}$ for some $s\ \varepsilon\ S_x$. Then the *component value* function $Cval\ (\ c_{i_1}, c_{i_2}, \cdots c_{i_k}\ ) = (\ r_{i_1}, r_{i_2}, \cdots r_{i_k}\ )$ where $r_{i_j}\ \varepsilon\ D_{i_j}$: $1 \le j \le k$. r_m is also denoted as $Cval\ (c_m)$ for $m\ \varepsilon\ \{\ i_1, i_2, \cdots, i_k\ \}$. Thus $Cval$ is an assignment of values to components.
- * If $(\ r_{i_1}, r_{i_2}, \cdots r_{i_k}\ )$ is the tuple of values of components $(\ c_{i_1}, c_{i_2}, \cdots c_{i_k}\ )$ then $Dec^{-1}\ (\ r_{i_1}, r_{i_2}, \cdots r_{i_k}\ ) = s$. Thus $Dec^{-1} : \mathrm{X}_{j=1}^{k}\ D_{i_j} \rightarrow S_x$ is an *inverse decomposition* function that collapses the values of the components back into the abstract state that the components constitute.
- * $Crel$ is a set of functions $\{\ f_1, f_2, \cdots, f_{N(x)}\ \}$ where each function is a one to one association identifying a pair of components. Thus $f_i : C \rightarrow C$: $1 \le i \le N(x)$ are a set of binary *component relationship* functions. The component relationships capture some relation, logical or structural, between components in a state decomposition. $N(x)$ is a non-negative integer that represents the number of component relationship functions for a given object x. Information provided by the component relation functions (for example Parent, Left_child, Right_child of a component representing a node of a tree structured object) is used by restructuring activities (see below) to perform

changes of object implementation states.

An *internal event* is similar to our earlier notion of an event which applied to *external event* or events at the abstract level. The differences between internal and external events are:

i) An internal event may involve either a *restructuring activity* or a user activity whereas an external event may not involve restructuring activities.

ii) The set of state transitions caused by pairs of internal events that constitute an internal operation includes those caused by restructuring activities. Thus in addition to the invocation and response symbols and parameter lists for external operations, internal events may also involve a new set of symbols and parameter values, namely those corresponding to restructuring operations.

iii) An internal operation (a matching pair of internal events) may involve a set of components of an object implementation state, whereas an external event may involve only a single object. For a given activity and a given internal operation at a given component there is associated a set of components called the *window* of that invocation. The window is often expressed in terms of the component relationship functions in *Crel*. It need not be a function of all three parameters indicated above.

For the sake of clarity we shall not introduce additional notation to represent these new operations, activities and transition symbols. It will be clear from the context whether a history, event, operation, activity etc. is internal or external.

Each internal event is an internal invocation event or an internal termination event. Internal events are four tuples where the first three fields are similar to the corresponding fields in external events (with the provision for internal activities and operation symbols and parameter values). The fourth field in an internal event is a set of components that constitutes the *window* of the corresponding operation invocation.

3.6 Correctness of internal histories

A definition for correct external histories was given in section 3.4. In this section we consider internal histories and relate them to external histories in terms of correctness. We characterize internal histories, define when an internal history is correct, demonstrate how an internal history can be transformed into an equivalent external history, and thereby how to verify if a given internal history is a correct implementation of a given external history. Our development hinges around a notion of *atomicity* of window updates. Atomicity is the absence of simultaneous update or overlap. (This restriction will be relaxed in chapter 4).

An *internal history* is a sequence of internal events where each internal termination event is preceded by a matching internal invocation event. A *complete internal history* is an internal history with no unmatched events.

Atomicity assumption. We shall assume that there are no simultaneous updates at the internal level. Thus each pair of matched internal events constitutes an atomic operation that transforms an object implementation state in mutual exclusion. Realization of the atomicity assumption amounts to implementation of mutual exclusion by all operations at each component.

The atomicity assumption allows us to transform a sequence of events into a sequence of operations very easily. Given an internal history h, we simply consider each operation as taking effect at the time of its termination event. Thus the subsequence of termination events in h gives the sequence of operations *Opseq* (h) in that history, such that if *Opseq* (h) is applied to the initial object implementation state (*ois*) of the history, the system will move through precisely the same sequence of object implementation states as it did with h.

Associated with each component is a consistency constraint or an invariant for that component. A *correct internal history* is an internal history that satisfies the atomicity assumption and causes a state transition if and only if the consistency constraint of each component in the window of each operation is preserved as an invariant. Intuitively an internal operation should not be allowed if it invalidates the consistency constraint at a component. If each operation in an internal history preserves the consistency constraint at each component, the overall consistency constraint of the object must be preserved. Therefore such an internal history is correct.

3.7 Relating internal histories to external histories

Given a correct internal history h, we say h is an *implementation* of an external history H if the subsequence of *Opseq* (h) consisting of *Opseq* (h) with all restructuring operations removed and with the object name replacing the component names in each operation in *Opseq* (h) is a valid reordering of H.

In example 4.1 in the following section we demonstrate external and internal histories of the joint account object. The latter can be shown to be a correct implementation of the former.

Given the above notation for object implementation states, we can discuss partitioning, binding and restructuring in more concrete terms.

3.8 Partitioning, binding and restructuring

Informally, partitioning is described as the task of creating state decompositions or the task of designing an object implementation that consists of *components* that are simultaneously updatable. We shall not attempt to characterize partitioning more formally, since this intuitive notion is adequate for our purposes.

Given a set of activities, A, a state s of an object x, and an object implementation state of s given by the quadruple $< Dec, Cval, Dec^{-1}, Crel >$, B is a *binding* if it is a many to many mapping from A to *Dec* (s), the set of components in the decomposition of s.

Binding is thus the association between activities and components of an object. Such an association may take place statically, at time of creation of the object, or dynamically, at runtime. A binding of an activity A to an object x is *static* if *B(A)* is fixed at the time of creation of x, and this binding never changes. Similarly, a binding is *dynamic* if it is different at different times in the same history. For example in the case of the joint account object of section 2, B(Bob) = "checking" and B(Mary) = "savings", and this is a static binding.

Partitioning can also take place at different times. For example, an implementation of Balanced Cube object on a Hypercube [Dally '86] dynamically alters the partitioning as the size of the structure grows and shrinks, thus resulting in dynamic partitioning, or the alteration of the number of components of an object implementation state in the course of a computation. We shall see examples of dynamic binding in chapter 4.

Informally, restructuring is the task of changing an object's implementation state without changing the externally visible state of the object. Formally, *restructuring* is defined as a transformation in the object implementation state from ois to ois' in such a way that the inverse decomposition of both object implementation states are equal to the same object state, s. Note that we do not specify how the transformation is to take place. It could be brought about by a change in the decomposition function or a change in the component value function, or possibly even a change in the component relationships.

3.9 The scheduler of an object and motivation for path expressions

We shall informally define the scheduler for an object as an active encapsulation (because of internal activities such as restructuring activities) that allows certain sequences of events while disallowing others. The scheduler permits internal events to occur by issuing *permissions* to access partitions of an object's implementation state and receiving *acknowledgements* that signify the conclusion of such access. Simultaneous updates refer to overlapping event pairs witnessed by an external observer. An overlap witnessed externally need not correspond to an overlap permitted internally by the scheduler of an object. In fact, we have stipulated (by means of the atomicity assumption) that the only overlaps that occur internally are those that act on different components. Consider an object which may be simultaneously read but must be updated atomically. A valid history for this object is shown in Fig 3.2 (where the subscripts *PERM* and *ACK* denote the internal events *permission* and *acknowledgement* above).

By considering semantics of operations such as those of "read" and "write" we find that we can relax the atomicity assumption in certain cases. In chapter 4 we introduce the language of *extended path expressions* for specifying when a scheduler may allow concurrent access at an object partition or component. The motivation for path expressions has traditionally come from a desire to specify a different set of concerns from those that motivate dependency relations in a programming language. Path expressions have been used for specifying the synchronization properties of objects (resources) that are in general shared by multiple activities, stating what operations can commute and what operations must be done in serial order. We use path expressions to capture sequencing restrictions in partitions of object implementation states. Our path expressions go beyond specification of synchronization and commutativity to including the capability to partition the set of activities.

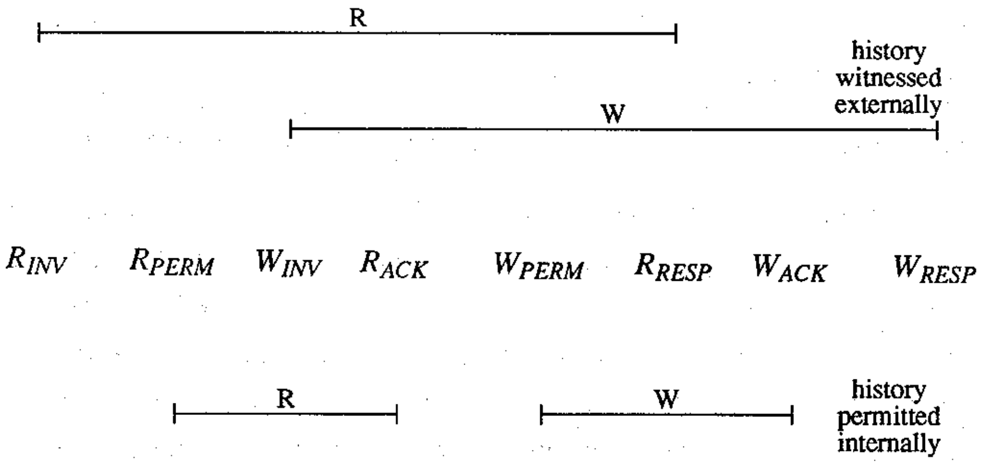


Fig 3.2 Valid history for multiple reader single writer synchronization

3.10 Comparison with the Karp and Miller model

At an abstract level our model of an object is similar to that of an instance of an *abstract data type* created by a Type manager [Browne '84] for the type of that object. The *adt* implementation details remain hidden. In our model we not only hide the implementation, we also guarantee that the implementation will meet the property of simultaneous update. With respect to the object implementation states we would like to compare our model with that of Karp and Miller.

Karp and Miller [Karp '69] have presented a model to represent a general class of *determinate* computations. This model provides for computational state to be simultaneously updated by introducing memory locations. A parallel program *schema* is a triple $< M , A , C >$, where M is a set of memory locations, A is a set of operations and C is a control defined by means of a state machine. Operations may asynchronously read and update memory locations as long as they preserve the property that two or more concurrent executions of an operation terminate in the order of their initiation. An operation is initiated by means of an initiation symbol, and upon termination produces one of a finite set of symbols denoting the set of possible outcomes for that operation. The values that are to be written into memory and the particular choice of the outcome symbol are functionally determined by the values read from memory. Thus regardless of the relative speeds of execution (times of completion of operations) the outcome of each computation (a string of symbols accepted by this schema) is unique, in the sense that every memory cell contains a single possible sequence of values. This property is known as *determinacy*.

Determinacy has been further characterized [Weng '75, Kahn '74] as the property that given an arbitrary directed graph with a function g_i at each node i of p nodes, and some initial conditions, the sequence of values on all arcs is uniquely determined. The streams (sequences of values) in the arcs is precisely the fixed point of the system of equations $Y_i = g_i (X_i) : 1 \leq i \leq p$, where Y_i and X_i represent respectively the output and input stream tuples corresponding to node i. The notion of histories or streams (lacking in the Karp and Miller model) is a notion that we have found very useful.

In contrasting our model of computation with the Karp and Miller model we make the following observations:

* Our internal operations are analogous to Karp and Miller's operations; however, at any given time there may be only one outstanding initiation of an operation on a component in our model. This is necessitated by the atomicity assumption. This requirement is relaxed through the use of ***extended path expressions*** in chapter 4. We do not assume that operations terminate in the

order of their initiation.

- Memory locations in the Karp and Miller model are similar to components in our model.
- Karp and Miller have a single global control that characterizes the state transitions taking place over the entire schema. In our case the state machines are distributed in the serial specifications of different objects and there is no attempt to unite these state machines into a global state machine.
- Since their model is an uninterpreted (based on syntax rather than semantics of the operations) the state transition function in the Karp and Miller model is a total function on all possible outcome symbols. In our model since we have added interpretation and semantics of the abstract data types for our objects we have undefined transitions.

4. Examples

Example 4.1. The joint account object

The joint account can be expressed in terms of the notation presented in section 3. The state domain is the set of non-negative integers, with the initial state being zero. The operation invocation symbols are "deposit" and "withdraw", each with an integer argument. The response symbols are "ok" and "sorry". The system may contain several objects and activities but we are only interested in the object x which is the joint account object, and the external or user activities Bob, Mary, and the restructuring activity Bank_Employee. We assume that the Bank_Employee can detect the state when a client is waiting for a withdrawal and there is not enough money. In such a case the client will receive a response "sorry".

Serial specification:

S_x : { i | i is an integer $\geq$ 0 }

$I_x : 0$

O_x : { $\ll$ withdraw, v, A, x >, < ok, $v', A, x \gg$,

$\ll \text{withdraw}, v, A, x >, < \text{sorry}, \text{NULL}, A, x \gg$,
$\ll \text{deposit}, v, A, x >, < \text{ok}, \text{NULL}, A, x \gg$
where v, v' are integers and $A \; \varepsilon \; \{\text{Bob, Mary}\}$ }

$T_x \; (s, \ll \text{withdraw}, v, A, x >, < \alpha, v', A, x \gg) =$
$s - v \;$ if $(\; s \geq v \;\; \Lambda \;\; v' = v \;\; \Lambda \;\; \alpha = \text{"ok"} \;)$
s if $(\; s < v \;\; \Lambda \;\; v' = \text{NULL} \;\; \Lambda \;\; \alpha = \text{"sorry"} \;)$
$undefined$ otherwise

$T_x \; (s, \ll \text{deposit}, v, A, x >, < \text{ok}, \text{NULL}, A, x \gg) =$
$s + v \;$ if$(\; v \geq 0 \;)$
$undefined$ otherwise

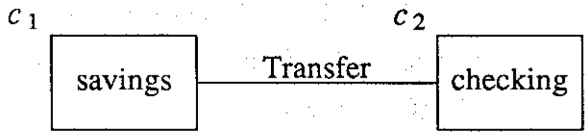


Fig 4.1: An *ois* for the joint account object

An object implementation state (Fig 4.1):

We show an object implementation state for which s, the abstract state is 50, and the internal state has 50 in c_1, which is the savings component, and 0 in c_2, which is the checking component.

In this case the decomposition function never changes. Thus for all $s \; \varepsilon \; S_x$:
$Dec \;\; (s) = \{c_1, c_2\}$ with $D_1 = D_2 = \{\text{i} : \text{i} \geq 0 \; \}$
Thus $Dec \;\; (50) = \;\; \{c_1, c_2\}$
$Cval \;\; (< c_1, c_2 >) = < 50, 0 >$
The inverse decomposition Dec^{-1} is the addition function. Thus:
$Dec^{-1} \; (< Cval \;\; (c_1), Cval \;\; (c_2) >) = Cval \;\; (c_1) + Cval \;\; (c_2) = s \; = 50$
Crel = $\{ \; (c_1, c_2), (c_2, c_1) \; \}$

$window$ (Mary, op) $= c_1$, $window$ (Bob, op) $= c_2$ and *window* (Bank_Employee, *op*) = $\{c_1, c_2\}$ where *op* ε {withdraw, deposit}

Shown below is an internal history *h* in which Mary makes a deposit of 50 dollars into the savings account, which is transferred to checking to cover Bob's withdrawal.

internal history *h*
< deposit, 50, Mary, c_1 >
< ok, NULL, Mary, c_1 >
< withdraw, 50, Bank_Employee, c_1 >
< ok, NULL, Bank_Employee, c_1 >
< deposit, 50, Bank_Employee, c_2 >
< ok, NULL, Bank_Employee, c_2 >
< withdraw, 50, Bob, c_2 >
< ok, 50, Bob, c_2 >

The sequence of operations *Opseq* (*h*) of the above history is shown below.

Opseq (*h*)
<<deposit, 50, Mary, c_1 >, <ok, NULL, Mary, c_1>>
<< withdraw, 50, Bank_Employee, c_1 >, < ok, NULL, Bank_Employee, c_1 >>
<< deposit, 50, Bank_Employee, c_2 >, < ok, NULL, Bank_Employee, c_2 >>
<< withdraw, 50, Bob, c_2 >, < ok, 50, Bob, c_2 >>

The reduced operation sequence of *Opseq* (*h*) with the internal restructuring operations removed is shown below.

reduced *Opseq* (*h*)
<<deposit, 50, Mary, c_1 >, <ok, NULL, Mary, c_1>>
<<withdraw, 50, Bob, c_2 >, <ok, 50, Bob, c_2>>

The operation sequence of H′, a valid reordering of the external history H is shown below.

Seq (H′)
<<deposit, 50, Mary, x >, <ok, NULL, Mary, x>>
<<withdraw, 50, Bob, x >, <ok, 50, Bob, x>>

Finally we show the external history H and a possible internal history h that is correct implementation of H. The events are shown in chronological order.

External history H	Internal history h
< withdraw, 50, Bob, x >	
< deposit, 50, Mary, x >	
	< deposit, 50, Mary, c_1 >
	< ok, NULL, Mary, c_1 >
	< withdraw, 50, Bank_Employee, c_1 >
	< ok, NULL, Bank_Employee, c_1 >
	< deposit, 50, Bank_Employee, c_2 >
	< ok, NULL, Bank_Employee, c_2 >
	< withdraw, 50, Bob, c_2 >
	< ok, 50, Bob, c_2 >
< ok, 50, Bob, x >	
< ok, NULL, Mary, x >	

Example 4.2. The index server object

Often, a set object consists of an ordered set of values, such as a sequence of integers over a given range (or an array holding elements of a set, in such a manner that a sequence of integers represents indices into the array). We introduce a certain kind of object called an *index server object*, abbreviated *iso*, that returns integers from a given range.

Say we have an index server object for the integer interval [lo, hi]. Let Z = { x | lo ≤ x ≤ hi : x, lo and hi are integers } be an ordered set of integer values. The object domain of the index server object is the powerset of Z, since it may contain any of the $2^{|Z|}$ possible subsets of Z. The initial state is the set Z. The only operation of interest is an index acquisition. The transition function defines the effect of removing an index from an *iso*. The transition function is interpreted as follows: If for some v $\{v \ \varepsilon\ s\}$, then the precondition that holds at an object state s, of the *iso*, then a *get* operation upon the *iso* returns the value v, modifying the object state s accordingly.

The index server object could be implemented with n [†] components $\{ c_1, c_2, \cdots, c_n \}$, each component initialized to hold a partition of Z. Let the i^{th} component, c_i, $1 \leq i \leq n$, have associated state variables lo_i and hi_i, such that component c_i contains the integers lo_i to hi_i.
Initially $Dec^{-1} < Cval\ (c_1), Cval\ (c_2) \cdots, Cval\ (c_n) > \ = \ Z$. Thus each component is suitably initialized to "contain" a subset of Z. Taken together the components implement the entire set.

Serial specification:
$S_x : 2^Z$, the powerset of Z where Z = { i | lo ≤ i ≤ hi : i, lo and hi are integers }
$I_x : Z$

O_x : { ≪ get, NULL, A , x >, < ok, v', A , x ≫,
≪ get, NULL, A , x >, < sorry, NULL, A , x ≫,
where v' is an integer between lo and hi and $A \ \varepsilon\ AC$ }

T_x (s, ≪ get, NULL, A , x >, < α, v', A , x ≫) =
$s - \{v'\}$ if ($v' \varepsilon\ s \ \Lambda\ \alpha =$ "ok")
s if ($s = \{\} \ \Lambda\ v' =$ NULL $\Lambda\ \alpha =$ "sorry")
undefined otherwise

An object implementation state: (s is the initial state)

$Dec\ (s) = \{ c_i : 1 \leq i \leq n \}$

$Cval\ (c_i) = <lo_i, hi_i> : 1 \leq i \leq n$

$$\text{where } lo_i = (i - 1) * \frac{IndexLimit}{n} + 1;\ hi_i = i * \frac{IndexLimit}{n}$$

$Dec^{-1} (< Cval\ (c_1), Cval\ (c_2), \cdots, Cval\ (c_{n)} >)$

[†] n is a parameter to be tuned for performance.

$$= Cval\ (c_1) \cup Cval\ (c_2) \cdots \cup Cval\ (c_n) = \mathrm{Z}$$

The above *ois* applies to the initial state. In general the inverse decomposition function gives the external state of the object as follows:

$$Dec^{-1}\ (<Cval\ (c_1), Cval\ (c_2), \cdots Cval\ (c_n)>) =$$
$$\{\ lo_1, \cdots, hi_1, lo_2, \cdots, hi_2, \cdots\ lo_n, \cdots, hi_n\ \}\ =\ s$$

$$\mathrm{Crel} = \{\ (\ c_i, c_{i+1}\ ) : 1 \le i < n\ \} \cup (\ c_n, c_1\ )$$

The window function associates an activity doing a particular operation upon the index server object with a single component, i.e. the one to which the activity is currently bound. The component relationships represent a circular list. Note that our implementation will be capable of yielding only a subset of possible sequences from the set Z. In this sense it does not implement the full behaviour of the set. However it does satisfy the serial specification of the set since the serial specification does not require that all possible sequences should be obtainable.

Example 4.3. The software combining tree

A replace add interconnection network can be regarded as a collection of *fan-in trees*, or inverted trees with memory cells at the root and processing elements at the leaves. A fan-in tree may get requests much faster than the requests are satisfied, thereby causing the tree to get congested. Congested trees of this nature are severe bottlenecks because they impede the flow of traffic that is otherwise unrelated. Yew *et al* [Yew '87] propose to reduce such contention for a given memory cell by distributing accesses to it across several memory cells, and imposing a global structure across these cells. This structure can be simultaneously updated at several points, thereby relieving contention at the bottleneck. The structure selected by the authors is a k*–ary* tree called a *software combining tree.* However, there is no reason why it should not be some other structure (such as that of the last example). It is clear that in order to reduce contention the authors have introduced structure into the object, and then introduced partitioning. Let d be the depth of the software combining tree, k the arity and L the largest value (assuming the initial value is 1). N denotes the number of processors. Taking the values of L, N, k, and d from the example in [Yew '87], we have $L=N=1000$, $d=2$ and $k=10$, we have 111 memory cells in the tree, with 100 cells at the leaf level. We partition the processors into 100 sets of ten processors each, with each set sharing a leaf level memory cell. When the last processor in each set decrements the contents of its memory cell to zero, it then decrements the value in the parent cell. Thus there are 111 bottlenecks, each with 10 accesses. Fig 4.2 illustrates this software combining tree. We illustrate the *ois* with a simplified version of this tree with three leaf components and one root component (see Fig 4.3).

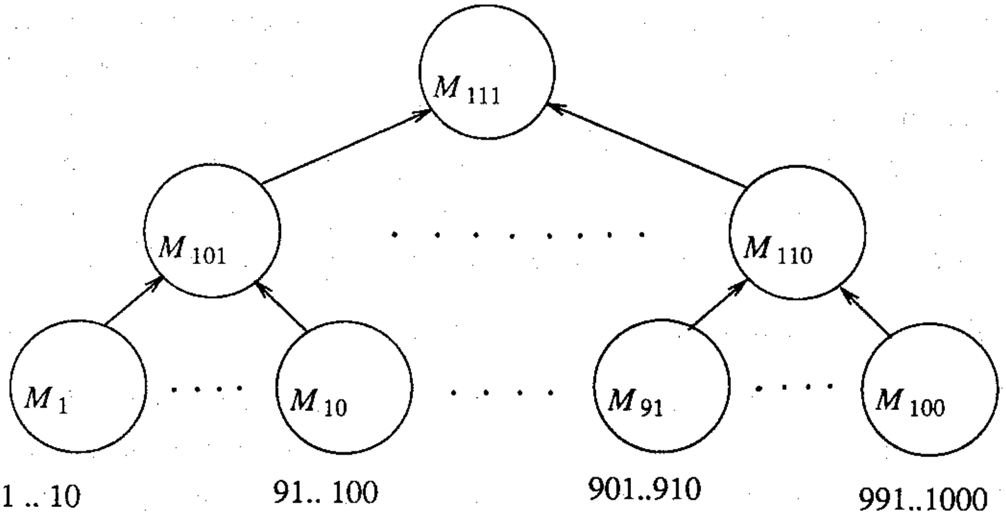


Fig 4.2 A software Combining tree with N = 1000, k = 10 and d = 2

Serial specification. The serial specification are the same as those of the *iso*, and therefore not being repeated here.

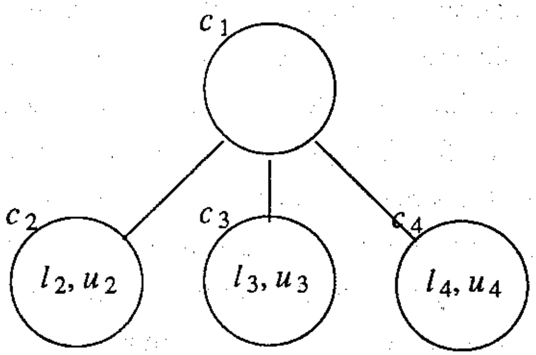


Fig 4.3: An *ois* of the software combining tree

An object implementation state (Fig 4.3):

$Dec\ (s) = \{\ c_1,\ \cdots\ , c_4\ \}$

$Cval\ (c_k) = < l_k, u_k > : 2 \leq \mathrm{k} \leq 4,\ \ Cval\ (c_1) = \mathrm{NULL}$

Dec^{-1} is the union function which is the same as the inverse decomposition function of the index server object.

$\mathrm{Crel} = \{\ <c_1, c_2>, <c_1, c_3>, <c_1, c_4>\ \}$

5. Performance tradeoffs in partitioning, binding and restructuring

What is unique about the index server object and software combining tree? At an abstract level, both these objects behave exactly as a *set* data type with the operation *get* (in the case of a combining network this operation is done by the *RepAdd* or "replace-add" primitive instruction) being analogous to the operation of removing an element from a set. Thus, if an activity invokes a *get* operation, and there is no element (i.e. index) in the partition to which it is bound, the operation must not return failure without making sure that there is indeed no index available in any partition of the object. In the case of the *iso* the linked list of partitions may, if desired, be traversed by a request until a non-empty partition is found, or all partitions are seen to be empty in which case the request returns with a failure.

In the case of a software combining tree, a request that finds a partition empty will continue to be an outstanding request until all indices are exhausted from all partitions. At this time the request will return with a failure, thus retaining consistency with the abstract specifications. A tree of barriers[†] is used to implement the synchronization necessary for all requesting activities to terminate at the same time. At each internal node of this tree only one activity is pre-selected to propagate a termination request up, and confirmation down the tree. Thus even though a large number of activities may be actually waiting at a barrier, at a given time only few of them are actively waiting at a specific memory location. Thus memory contention is greatly reduced. This is the performance advantage of the software combining tree.

The main questions are:

i) Under what conditions are such partitioning and binding strategies useful?

ii) What exactly is gained by having multiple binding times?

These questions are addressed by means of simple performance models.

Say the index server object is mapped onto a distributed memory machine, where the activities are organized as a ring. A natural way to map the circular list of components of an *ois* is as a ring upon this ring. Let us assume that the number of partitions equals the number of processors, n. We assume that activities are statically bound to partitions, with one activity being assigned to each processor, and that termination is signalled by a token that propagates through the linked list of components (processors). The time taken to complete the consumption of all indices, called the completion time of the index server object, is given by:

$$T_{iso} = Max_{i=1}^{n} \left\{ Tpart_i \right\} + n * T_{message}$$

where $Tpart_i$ is the time to complete consumption of all indices in partition i, and $T_{message}$ is the time for propagation of a message down one link of the ring. The maximum is taken over all partitions. T_{iso} is obviously dependent upon the distribution of computation times at partitions. Skewed distributions of computation

[†] A *barrier* is a shared memory location used to implement a synchronization point for a set of activities.

times at partitions can seriously affect the overall completion time. We therefore propose an alternate implementation using compile time partitioning initially, but with a provision for sharing work at runtime through dynamic binding of activities to partitions. Assuming that k% of the requests are directed at other partitions where k (a small positive integer less than n) is the distance propagated by a request increases as we near completion of the computation. Thus the distance propagated by the last 1% is larger than that propagated by the last but one 1% and so on. This scheme can capture skewed distributions by distinguishing between the values of k at different components. The analysis of this scheme is a little more complex. Let us assume that the computation time for an index T_{index}, is much less than the communication time $T_{message}$. Thus once communication starts, the time is dominated by the communication time. We also assume that the i^{th} component directs k_i% of its requests at other partitions. Thus skewed index distributions are captured by skewing the distribution of values given to k_i. We assume 100 requests are made at each partition. $T_{message}$ is the average time for a message to travel a single hop in the presence of other messages.

$$T_{iso} = Max_{i=1}^{n} \left\{ (100 - k_i) * T_{index} + \left[n + (n-1) + \cdots + (n - k_i + 1) \right] * T_{message} \right\}$$

$$= Max_{i=1}^{n} \left\{ (100 - k_i) * T_{index} + \left[\frac{n\ (n+1)}{2} - \frac{k_i\ (k_i + 1)}{2} \right] * T_{message} \right\}$$

5.1 Restructuring

The above analysis assumes one activity per component. There could be many activities per component, exemplified by several processes on a single processor. Indeed, it would be very inefficient if in the presence of skewed data, as above, each activity were to incur the messaging overhead shown above. Instead, we propose restructuring activities that dynamically alter the bounds of the implementation of partitions of the index server object. The analysis of restructuring is as follows. At each processor we have a fixed number of activities and a restructuring activity. Activities are statically bound. Termination is signalled by messages between restructuring activities. Assuming the computation is equally distributed by the restructuring, the time for the index server object is the time for R rounds of message exchanging to accomplish a uniform distribution of indices.

$$T_{iso} = \frac{Number\ of\ Indices}{n} * T_{index} + n * R * T_{message}$$

5.2. Degrees of simultaneity and contention

Two other performance measures by which simultaneously updatable objects can be rated are the degrees of simultaneity and the degree of contention. The degree of simultaneity of an object is the number of operations that it allows to proceed concurrently, or in an overlapped fashion. The degree of contention is the number of activities that may potentially access an object partition. For example, the degree of simultaneity of an index server object implementation, is precisely the number of partitions, n. The degree of contention is 1 in this case. The degree of simultaneity for the software combining tree with arity k and depth d is the number of leaves in the tree, i.e. k^d. By dividing L, the largest value, into k^d memory cells, and binding a different set of processors to each memory cell, we have guaranteed that the maximum number of contending requests at a single cell is $\frac{\min(L, N)}{k^d}$. This is the degree of contention of the software combining tree. We shall use these performance measures once again in chapter 5 when we characterize the performance of the concurrent heap.

6. The Doall node of TDFL

The Task-Level Data Flow Language (TDFL) is a graph oriented language for converting existing sequential programs to run in parallel, or writing entirely new programs. Computations in this language are expressed as static or dynamic directed graphs. Each node contains a subroutine-sized task and each arc holds data tokens that cause nodes to fire and emerge from nodes that have fired (Fig 6.1). The task functions are written in standard high-level languages. At any given time in the execution of a TDFL program, several nodes may be in execution, having consumed tokens from their input arcs and in due course, providing tokens on their output arcs. This language has been used to write application programs for a shared memory multiprocessor machine, with good speedups over a range of applications and close to linear speedup in one application [Biswas '87a]. Further details of this language and its implementation appear in [Suhler '87].

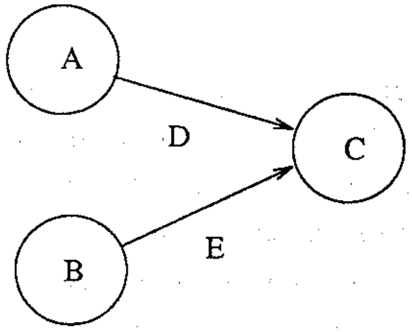


Fig 6.1

We are interested in looking for efficient implementations of the Doall node of this language. The Doall node receives a token (an array or a structure) on one of its inputs and a non-negative integer value on another. The latter determines the number of parallel iterates. The computation encapsulated in the Doall is repeated once for each iterate, incrementally computing output tokens. When all iterates have finished executing, the Doall node is said to have completed its firing. At this time tokens on the output arcs of the Doall node are ready for consumption.

One way to implement *iterate acquisition* for the Doall node is with a single memory cell that holds the state of the shared integer object. This scheme is inefficient since it relies upon a single shared object. On a highly parallel architecture, such a scheme presents an unacceptable bottleneck. It is necessary to decompose the single variable that represents the range of indices for simultaneous update. We can carry out this decomposition by viewing the input arcs of a Doall node as delivering decomposable objects that are instances of the *index server object* as introduced in section 4 of this chapter. Index server objects may be implemented efficiently on parallel machines, either by hardware primitives (such as the *RepAdd*) or by software (such as the *index server object*, presented in chapter 4). An activity doing a Dequeue upon an input arc of a Doall node removes an index from an index server object. The Dequeue operation does not rely on global serialization points, since the implementation of an *iso* object does not rely on global serialization points.

7. Conclusions

It is desirable for performance reasons, to partition an object implementation state so that simultaneous update can take place at different partitions or components at the same time. By early binding of different activities to different components, run-time contention and traffic in general can be reduced on highly parallel architectures. There is evidence to show that such objects would indeed prove useful on parallel machines. The *software combining tree* is a case in point. In order to provide external views of simultaneous update, it is desirable to perform internal restructuring asynchronously, provided that the consistency constraints of the application permit such restructuring.

We have introduced in this chapter the notion of simultaneous update through a model of abstract data types. The approach outlined in this chapter, namely characterizing the abstract behaviour of an object at an external level and then designing its implementation specifications at an internal level, can be applied repeatedly in a hierarchical fashion. For example, each component of an object may be thought of as an object and can have a decomposition. For practical reasons we have stopped at one level. We emphasize that we do not present a methodology to go from serial specification in the abstract to the concrete object implementation. Specifying the object implementation states is a problem left to the user, and it is a part of the "art" of good programming to be able to design appropriate decompositions and component value functions for a given object to be mapped onto a given architecture.

A correct computation is one that satisfies consistency constraints specified. We focus on a class of computations that only require consistency constraints on individual objects. To guarantee preservation of correctness we have introduced a strong requirement called the atomicity assumption. Since operations are treated in a very general and uninterpreted manner, such a strong assumption is necessary. A possible application of simultaneously updatable objects is in the TDFL Doall node, which can potentially use the index server object or the software combining tree objects introduced in this chapter.

By incorporating semantics of operations through the use of linguistic constructs called path expressions it is possible to allow simultaneous update even at the component level. In the following chapter we introduce work container objects, a generalization of the index server object, and characterize the class of computations that are generated by the use of these objects. We also discuss an implementation of a particular form of path expressions.

Chapter 4 - Language tools to support simultaneous update

1. Introduction

In chapter 3 objects have been characterized by their serial specifications and their external and internal histories. An internal history involves sets of components of an object implementation state (also called *object components* or *partitions*). By assuming that updates to sets of components will be atomic, we have quite straightforwardly characterized the correctness of internal histories with respect to external histories.

In this chapter we sketch an implementation scheme for creation of simultaneously updatable objects and their use on current multiprocessor machines. We focus on an object known as a *work container object*, abbreviated *wco*. The serial specification of this object is very similar to that of a multi-set or a bag, except for the fact that operations have been introduced for programming convenience. We provide a structuring concept, the *work manager*, similar to monitors [Hoare '74] for creating work container objects and setting up computations centered around these objects. We also relax the assumption that updates to object components must be atomic, by allowing in work managers, the use of *extended path expressions* (abbreviated *epe* s).

Path expressions are languages for specification of synchronization properties of objects. A path expression specifies the permissible sequences of operations at an object (or a partition). We have extended path expressions in two ways: a) consistency properties at an object partition may be specified, and, b) activity bindings to object partitions may be specified. The former makes it possible to implement correct internal histories and the latter permits the specification of patterns of activity bindings that may be tailored to various multiple processor architectures. Thus *epe* s are a concise notational tool for expressing mapping and consistency properties of components of an object in addition to specifying synchronization behaviour.

Epe s allow simultaneous update at the component level. Thus histories of operations at the internal level are now partially ordered instead of being totally ordered. We characterize how our earlier notion of correct internal histories must be altered to accommodate this change.

The use of work container objects is illustrated through a few examples. We also present in this chapter, algorithms for synthesizing code for the implementation of extended path expressions. Various implementations of work container objects are possible including several proposed in the literature, e.g., the semi-queue [Weihl '84], the weakly FIFO queue [Schwarz '83], the tuple-space [Gelernter '86], and distributed open lists [Quinn '84, Rao '87a]. These objects differ significantly in their usage and performance, illustrating the potential diversity of work container computations. The balanced cube [Dally '86] and concurrent search structures [Shasha '87] are similar to *wco* s in that they contain sets of items; however, they are different in that they also support query operations, such as *member*, and allow insertion and removal by key.

In the following chapter we introduce priority structures, simultaneously updatable objects whose serial specification is that of a weak priority queue in the sense the item returned is not the largest value but *one of* the largest values. As expected the performance of these objects is much better than that of strict priority queues. With further weakening at an internal level it is possible to obtain implementations with further improvements in terms of simultaneous update. Performance tradeoffs for these objects are analysed in chapter 6.

2. Work container objects

A *work container object* (abbreviated *wco*) has the abstract semantics of a multiset or bag with the operations *put*, *get*, *init*, *clear* and *sense*. The form of these operations is unrestricted (as long as they preserve the properties of a bag). Work container objects are used to store *workpieces*, or abstract units of data used by activities to do computation. A *workpiece* is an entity that keeps activities busy. The most important operations are *put* and *get*. The *put* operation inserts a workpiece, and a *get* operation removes a workpiece.

2.1 Serial specification of work container objects

Let D be a set of values denoting the domain of a multi-set.

S_x : The set of all multi-sets of items from domain D.

I_x : { }, the empty multi-set.

O_x : $\{ \ll \text{put}, v, A, x >, < \text{ok}, \text{NULL}, A, x \gg,$
$\ll \text{get}, \text{NULL}, A, x >, < \text{ok}, v', A, x \gg,$
$\ll \text{get}, \text{NULL}, A, x >, < \text{sorry}, \text{NULL}, A, x \gg,$
$\ll \text{init}, \text{NULL}, A, x >, < \text{ok}, \text{NULL}, A, x \gg,$
$\ll \text{clear}, \text{NULL}, A, x >, < \text{ok}, \text{NULL}, A, x \gg,$
$\ll \text{sense}, \text{NULL}, A, x >, < \text{ok}, \text{NULL}, A, x \gg,$
$\ll \text{sense}, \text{NULL}, A, x >, < \text{sorry}, \text{NULL}, A, x \gg,$
where $v, v' \varepsilon D$, and $A \ \varepsilon AC$ }

$T_x\ (s, \ll \text{put}, v, A, x >, < \text{ok}, \text{NULL}, A, x \gg) =$
$s \cup_b \{v\}$ if ($v \ \varepsilon \ D$) where $\cup_b$ denotes bag union

$T_x\ (s, \ll \text{get}, \text{NULL}, A, x >, < \alpha, v', A, x \gg) =$
$s \ -_b \{v'\}$ if ($v' \varepsilon s \ \wedge \ \alpha = \text{"ok"}$) where $-_b$ denotes bag difference
s if ($s = \{\} \wedge \ \alpha = \text{"sorry"} \ \wedge \ v' = \text{NULL}$)
undefined otherwise

$T_x\ (s, \ll \text{init}, \text{NULL}, A, x >, < \text{ok}, \text{NULL}, A, x \gg) =$
{} if (x does not exist)

$T_x\ (s, \ll \text{clear}, \text{NULL}, A, x >, < \text{ok}, \text{NULL}, A, x \gg) =$
{} if (x exists)

$T_x\ (s, \ll \text{sense}, \text{NULL}, A, x >, < \alpha, \text{NULL}, A, x \gg) =$
s if (x exists $\wedge$ ($\alpha = \text{"ok"} \vee \alpha = \text{"sorry"}$))

Intuitively *sense* returns "sorry" when the bound partition of the object is empty. This is further explained below.

2.2 Sense and sense-lessness

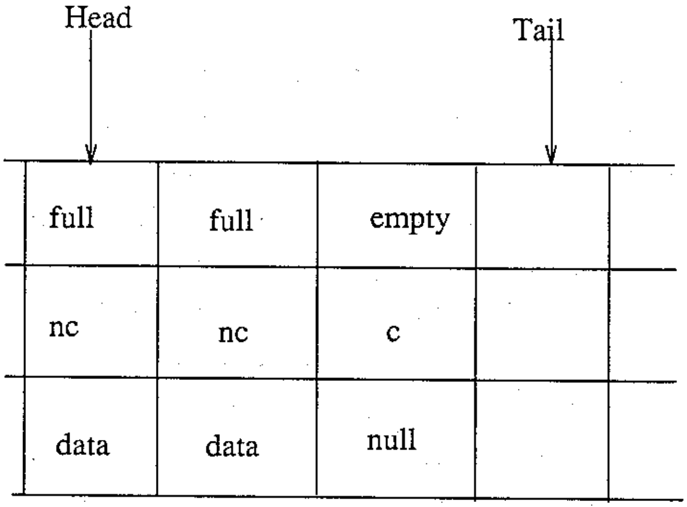
As explained in chapter 3 an activity may be bound to one or more components of the decomposition of the abstract state of an object. The *sense* operator is a means of determining if the currently bound partitions are empty. All we can assert after a *sense* operation returns with a success (failure) indicated by "ok" ("sorry") is that a set of bound partitions is non-empty (empty).

The *sense* operator is important from an implementation standpoint. However the abstract properties of this operator are not very interesting. A *sense* returning success indicates that the local partition, in other words, the object component to which an activity is bound, was non-empty at the time of the invocation. A *sense* returning failure only indicates that the local partition is (or was) empty. Absence of work at a partition does not say anything about absence (or presence) of work globally. An activity may attempt to acquire a new binding and retry work acquisition.

Thus a computation in which *sense* appears, does not inherently have stronger properties than a *sense* -less computation. It can easily be shown that for every computation consisting of a set of objects and a set of activities in which *sense* operations appear, there exists a *sense* -less computation that is equivalent, in that it returns exactly the same values to every operation invocation and it puts the set of objects in the same final state. The proof consists of taking a projection of a history including *sense* operations, such that all *sense* operations are removed. The resulting computation is a valid *sense* -less computation. Since *sense* does not alter the state of the object, the two computations, namely the original and the projection of it, must be equivalent. Thus the *sense* operator is useful only from a performance standpoint. A computation using index server objects with dynamic bindings is an example of a work container computation that uses *sense*. Another example is the semi-queue object implementation discussed below.

2.3 Implementation of the semi-queue

An interesting implementation is proposed by Weihl [Weihl '84], for an object called the *semi–queue* that has the same serial specification as the work container object. In this scheme recovery is an important issue, thus an item is not deemed to be enqueued until its corresponding activity has committed. Weihl's implementation is roughly as follows. The main object is an array of cells (Fig 2.1), each with a boolean flag indicating whether the cell is full or empty. There are in addition, a head and a tail pointer for facilitating dequeueing and enqueueing respectively. A non-empty cell contains an enqueued value. An insertion consists of atomically incrementing the tail pointer of the queue, inserting (i.e. updating the cell), and then setting the flag to full. A deletion consists of two stages, first a search through the array from the head towards the tail until a non-empty cell is found or the tail is reached, whichever is earlier. The second phase consists of atomically removing the available item. (There is a possibility of failure at this stage due to a race condition). Resetting the head pointer is done by restructuring processes (garbage collectors) that push back the head pointer past garbage (empty) cells.



c - committed

nc - not committed

Fig 2.1. Implementation of the Semi-queue

Each cell is obviously a partition. All activities have access to all partitions, and a partition contains at most a single element. Each activity does a *sense* upon a partition before issuing a *get*. Thus an item acquisition may be regarded as a sequence of *sense* operations followed by a single *get*. If we ignore recovery, this object is empty only when head and tail pointers coincide.

2.4 Work Container Computations

Work Container Computations are computations that involve activities centered around work container objects. The major distinguishing feature of such computations is that they should satisfy consistency constraints as outlined in section 3.4 of chapter 3. In other words, there are no interactions across work container objects and no integrity constraints to be preserved across work container objects. All that is required is that items should not *languish* [Schwarz '83] inside work container objects for ever; in other words, an item inserted must *eventually* be deleted, given that new insertions are suspended.

2.5 The need for canned type definitions

It is generally agreed that if an abstract data type (e.g. stack) and its operations (e.g. pop and push) are supported on two machines (with possibly different architectures), programs using the data type can be easily ported between the machines. Also, if an abstract data type is available in a library it can be used by multiple programs that are written for the same machine.

Work container objects should be supported on multiple types of parallel machines. If this is done, parallel programs using canned definitions of such objects can exploit the same two benefits, namely portability and reusability, as sequential programs have exploited by using conventional abstract data types on sequential machines. To facilitate the setting up of work container computations we introduce canned definitions of work container objects. Such definitions, called *work managers*, allow instantiation of these objects at various times depending upon when the binding of key parameters of the object takes place. In order to support work container objects certain auxiliary operations that are transparent to user programs are necessary. Foremost among these is the *create* procedure that permits instantiation and binding of a named *wco*. Binding of activities to partitions of an object may be performed at creation time, and may be made to apply to subsets of activities by the use of extended path expressions (discussed below). As shown in the schematic in Fig 2.2, user programs contain declarations of work container objects, which are instantiations of work managers, which in turn contain *epe* s.

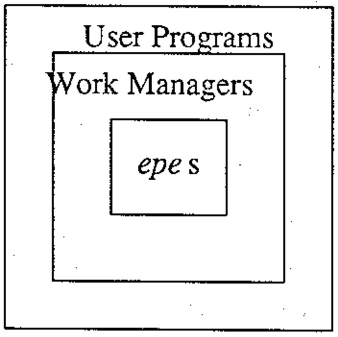


Fig 2.2.

3. Extended path expressions

The notion of a path expression was first proposed by Campbell and Habermann [Campbell '74, Habermann '75] who used a simple language of regular expressions to express commonly occurring synchronization patterns. In time path expressions evolved into a family of languages that were more powerful but required context free grammars. Some of these proposals are *predicate path expressions* [Andler '79], *open path expressions* [Campbell '77], *resource expressions* [Jayaraman '81], and *open predicate path expressions* [Headington '85, Oldehoeft '84].

In our work, we have felt the need for a concise notation to express possible interleavings of operations at object partitions or components. Path expressions are an obvious choice. We associate a path expression with each component of an object implementation state. In addition to the information specified in the path expression, we also provide notations for expressing: a) consistency constraints that must be preserved as invariants by each operation accessing the component, and b) ranges or sets of activity identifiers such that only invocations made by these activities will be allowed access to the component. The overall expression is called an extended path expression and is explained in more detail below.

3.1 Syntax of extended path expressions

An *extended path expression* (abbreviated *epe*) is an expression in a very high level language that specifies sequencing, synchronization and consistency constraints that must be preserved by operations accessing a component of an object. A *target language* is a high level language into which an *epe* is translated. An *epe* consists of three portions : a synchronization portion, an activity binding portion, and a consistency constraints portion. The first of these is a recursively defined list of sequences, where each element of a sequence called an item, consists of a synchronization part (mandatory) and an optional predicate part. Predicates involve certain predefined event counters to implement synchronization constraints. These are *invok*, or the count of invocations of an operation, *perm*, the count of permissions granted for a particular operation, *ack*, the count of acknowledgements received against the permissions granted, and *rep* the count of replies sent to callers or invokers. The activity binding portion of an *epe* consists of invoker identifications, or names of activities that are allowed to invoke operations on an object partition. The consistency constraints allow specification of consistency properties that must hold at an object component. The syntax [†] of *epe* s is given below.

```
<epe > ::= path <synchronization_portion> <invoker_portion>
                <consistency_portion> end
<synchronization_portion> ::= <list>
<list> ::= <sequence> | <sequence> , <list>
<sequence> ::= <item > | <item> ; <sequence>
<item> ::= <unsigned-integer> : ( <list> )
    | { <list> }
    | <item> [ <predicate> ]
    | ( <list> )
```

[†] Our notation for expressing the synchronization portion is Headington's notation [Headington '85] with the addition of an event counter for replies to invocations.

```
        | <operation-id>
<invoker-portion> ::= <invoker_term> |
                      <invoker_term> <invoker_portion>
<invoker-term> ::= invoker in <range> | <set>
<consistency-portion> ::= ccbegin <targ-lang-bool-exp> ccend
```

(The remaining details of the *epe* syntax are in Appendix 1).

Some terms in the above syntax need to be explained. Operation identifiers ("operation-id") are the atoms from which synchronization portion of an *epe* are created. The "predicate" as mentioned previously, is built up of four event counters associated with each operation and may involve several operations. It is defined in the usual way as a linear relational expression.

We assume that each activity (invoker) is given a unique number similar to a subscript from an integer valued "range", or that it bears a unique name from a "set" of predefined names (such as "Bob" and "Mary" in the joint accounts example). Partitioning of a range or a set can be specified by the use of a suitably defined arithmetic expression or set valued expressions. The phrase "invoker in <range> | <set>" specifies that if the identification of an accessing activity is in the range "range" or the set "set", then it may issue operation invocations upon the object partition. Different object partitions have different ranges and thus different activity bindings. By separating activities into groups in this manner, accesses to the object are distributed, and runtime contention is reduced.

The phrase "ccbegin <targ-lang-bool-exp> ccend" specifies that an operation may access an object component only if it satisfies the consistency constraint specified by the target language boolean expression. A consistency constraint is an arbitrary boolean expression on the state of the component preserved by the underlying system as an invariant through state changes. These expressions are written in the syntax of the target language, and naturally involve data types and operators in the target language. The invoker and consistency portions may have null bodies, indicated by an asterisk and "nil" respectively.

3.2 Semantics of the synchronization portion

The semantics of the synchronization portion are standard [Headington '85]. The comma is a separator between elements of a list. Unless otherwise constrained, there may be an unrestricted number of concurrent executions of each expression that is the element of a list. The restriction operator which is an unsigned integer *k* followed by colon and a list within parentheses, allows at most *k* executions of elements within the parentheses. Thus 20 : (A, B) allows the total number of concurrent executions of A and B to be at most 20. The derestriction operator (chain braces) allows an unlimited number of concurrent executions of the expressions within the braces, with the restriction that the number of initiations counted by *perm* (the number of permissions of the expression) must be strictly greater than the number of terminations counted by *ack* (the number of terminations of the expression). As soon as these two counts become equal, no further initiations are allowed in the current instance of the subpath. For example the expression 1:({R}, W) denotes that at any time *either* a subpath consisting of a single W operation *or* a "burst" of R operations can be in progress. The burst is deemed to have terminated as soon as the event counts *perm* and *ack* are equal. Interpreting R to mean readers and W to mean writers, this expression specifies multiple reader, single writer mutual exclusion with possible writer starvation. The sequencing operator, denoted by a semicolon, makes sure that operations (or expressions) execute in sequential order. Thus at any given time it ensures that the number of initiations of B is at most as much as the number of terminations of A, if A; B is the expression being implemented.

3.3 Semantics and correctness of internal histories

An *epe* history h is a sequence of events at a component that are allowed by the *epe* associated with the component. Since *epe* s allow simultaneous update there may be overlapping pairs of internal operations in h. Thus the set of operations in h is a partially ordered set. Let $(Op(h), <)$ denote the partial order associated with the set of operations $Op(h)$ in h.

The semantics of an *epe* are provided as in [Andler '79] by translating it into an equivalent program in the programming language of Dijkstra's guarded commands, augmented with *cobegin* and *coend* constructs for collateral execution (the "," operator). Given an internal history h, and a partial order on its operations $(Op(h), <)$, h is said to *obey* the extended path expression e, if and only if there is a history of execution of operations in C(e), the translation of e into guarded commands, such that the partial order of operations in C(e) is $(Op(h), <')$ and $< \subseteq <'$.

An internal history is *correct* if each operation in the history individually preserves the consistency constraints or invariant at each component that appears in the history. Note that since we have relaxed the atomicity assumption through the introduction of path expressions we no longer have to bind an operation to the set of components that it accesses. A history must be correct at each component in the sense that it must satisfy the consistency properties and it must be acceptable by the *epe* associated with each component accessed in the history.

Equivalence of internal and external histories: To obtain an equivalent external history from an internal history of operations we impose a total order consistent with the partial order (topological sort) of operations in the internal history. Since we do not know which total order represents the actual sequence of operation completions, all possible total orders must produce equivalent histories (reorderings of) the externally observed history.

4. Work managers and their use

Work managers are data abstraction mechanisms (type definitions) for work container objects. A work manager consists of a type definition, instantiations of which are work container objects. It also supports the *create* operation with the syntax: < *name* >.*create* where *name* is the identifier associated with the created object. A created work container object has the operations put, get, init, clear, and sense. One or more of these operations could have null bodies, i.e., the operations may be left undefined since they do not have any significance in an application.

In this section, we demonstrate the use of work managers. We first briefly discuss the style in which our user programs (i.e., programs that use predefined work managers) are written. This style of programming ensures that the same instruction in the same instruction stream of two activities will be directed at different object partitions. Next, we present a user program and work manager for a canonical example. Following this, another example is provided, demonstrating how a task bag can be set up using a work container object. In our examples, we restrict ourselves to Pascal as a generic example of Algol like languages, with block structuring and dynamic allocation of memory. In section 5 we discuss implementation issues in mapping and executing such user programs on parallel machines. We have not actually implemented work managers and path expressions, thus the implementation ideas presented here are to be viewed as a feasibility demonstration.

4.1 The single program multiple data (SPMD) programming style

A particular style of parallel programming has become quite popular [Darema '85], [Karp '87]. The assumption is that programs written in this style will be run as a number of parallel threads of execution of the same program. Our current implementation of TDFL is also written in this fashion. The program is written in one of several source languages, such as Fortran, Pascal and C, augmented with some constructs that are directives to a preprocessor for generating appropriate code. Branching based upon values of local or shared variables, enables each thread to execute a different part of the program asynchronously. We shall adopt this programming style for the following reasons:

i) It allows us to construct *activities* as unit executions of the same program. These unit executions may be viewed as histories. Each unit execution of an SPMD program is an activity. Any program variable that is accessible by more than one activity is an object.

ii) SPMD programs are suited for shared memory as well as message based architectures (as shown in [Karp '87]).

iii) Objects such as barriers, that have already been developed by others for SPMD programming, may be used as long as they do not violate the consistency properties of our computations.

For the remainder of this thesis, we shall use "user program" to stand for a program that has been written in the SPMD style and references work managers. The specific SPMD constructs we have in mind are *serial sections* and *barriers*. Barriers have been explained in section 5 of chapter 3. A serial section is a portion of a program that must be executed by a single activity and must be completed before any of the remaining activities may proceed beyond the serial section. Its main use is in initialization but it is also used for synchronization. In addition user programs may have *wco* declarations and operation invocations. Activities may be given program relative identifications (such as arrays of activity identifiers), and the actual mapping of these identifications to physical processors is kept transparent from user programs.

4.2 A canonical example - the Prime Number Solver

Let us assume we want to find all prime numbers between 1 and *maxnum*. One solution would be as follows. We introduce a single integer variable called *NextValue*, protected by a lock called *Lock*, that is used to store the value of the next integer in the range 1, $\cdots$, *maxnum*. *NextValue* is accessed and incremented in mutually exclusive mode by invoking the 'lock' operation upon *Lock* for the duration of access.

Assuming $k \ll maxnum$ (where k is the number of activities), each activity executes the code to determine if a number is prime. Each activity then follows the algorithm sketched in Program 4.1, which consists of a simple non-recursive procedure, *Print_If_Prime*, to determine by repeated division if a given number is prime. The program is written in the syntax of Pascal augmented with commands such as *@lock* and *@unlock* for converting it into an SPMD program.

```
var NextValue : shared_integer; Lock : shared_lock; (* globals *)
procedure prime;
var done : boolean; index : integer; (* locals *)
begin
  @serial
    NextValue := 0;
  @end_serial
  done := false;
  while (not done) do begin

    @lock ( Lock );
    if ( NextValue > maxnum ) then
        done := true
    else begin
      NextValue := NextValue + 1;
      index := NextValue;
    end;
    @unlock ( Lock );

    if (not done)
     then Print_If_Prime (index);
  end;
end;
```

Program 4.1

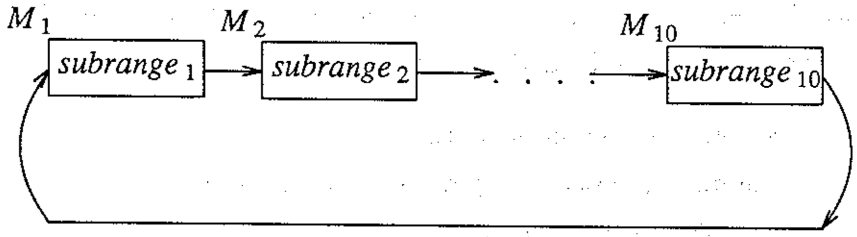


Fig 4.1. A linked list of memory cells for holding different ranges of values.

4.2.1 Solution using an index server object

It is obvious that program 4.1 does not allow simultaneous update of *NextValue*. Using an index server object, it is possible to simultaneously update *NextValue*. We set up a work container object for a set of values from 1 to *maxnum*. The primary purpose of this object is to act as a server of indices from a set (hence the name index server object). The associated work manager is of predefined type index-manager.

We concurrently start a number of activities (*nacts*), each of which executes the code to determine if a number is prime. The object implementation state consists of a circularly linked list of *npartitions* nodes, each of which contains a value from a subrange of $\{1, \cdots, maxnum\}$. Fig 4.1 demonstrates the circular list for *npartitions* = 10. Different sets of activities may be bound to different memory cells statically or dynamically. This has been discussed in section 5 of chapter 3.

Say *nacts* is larger than and a multiple of *npartitions*. Then chunk = (*nacts* / *npartitions*) activities could be bound to each component. This is specified by the subexpression "invoker in ((*j* - 1) * chunk + 1, *j* * chunk)", which holds for the j^{th} partition or component. Consistency constraints are expressed in the syntax of the target language, which in this case is C.

The name of the instance of the index server object is *iso*. As before, each processor uses a simple non-recursive algorithm to determine if a number is prime. The computation terminates as soon as a get operation upon *iso* returns a null condition, signified by a flag. Although in this example, the linked list structure has not been used, it is easy to see how processors can be made to traverse this structure, so that the binding is not static but dynamic.

```
procedure prime (npartitions, maxnum );
var npartitions, maxnum : integer; (* number of partitions *)
  over : boolean; index : integer;
  @declare_wco
      wco_name      : example_wco;
      work_manager : index_manager;
      parameters   : npartitions = 10;
                   maxnum = 1000;
  @end_wco
  @declare_wco
      wco_name      : iso;
      work_manager : index_manager;
  @end_wco
begin
  @serial
        iso.create (npartitions, maxnum);
  @end_serial

  over := false;
  while (not over) do begin
       iso.get (index);
       over := (index = -1);
       if (not over) then Print_If_Prime (index);
    end;
end;
```

Program 4.2. User Program for finding primes between 1 and *maxnum*

The object instantiation "*example_wco*" declared in the declaration section of Program 4.2 demonstrates how the creation of a *wco*, and hence activity binding, can be specified to be performed at load time. Program 4.3 shows the work manager for *iso*, called *index manager*. Note that certain variable bindings are available from the load time environment. These are *myid* and *nacts*. Other variable bindings are available from the create-time environment. *npartitions, maxnum*, etc. are examples. Creation can be forced at load time, by binding the formal parameters of the work manager definitions in the definitions themselves. The load time parameter *myid* essentially provides the activity with a knowledge of its own identification. For certain applications, it may be appropriate to provide the activity with other load time bindings, such as the identifications of neighbouring processors (activities).

```
@wm_begin
@name : index_manager;
@formal_parameters : npartitions,
            maxnum
@epe : **path** 1 : ( get, init, clear ) **invoker in** (( j - 1) * chunk + 1, j * chunk)
```

```
        **ccbegin** lo < hi **ccend end**

@operations :
procedure put;
begin
end;
procedure init;
begin
    hi := ichunk * j;
    lo := (j-1) * ichunk;
end;
procedure get (var index : integer);
begin
    if ( lo < hi ) then
      begin
        lo := lo + 1;
        index := lo;
      end
    else
      index := -1;
end;
procedure clear;
begin
   lo := hi;
end;
procedure sense;
begin
end;
@where
begin
    chunk := nacts / npartitions;
    ichunk := maxnum / npartitions;
    j := 1  . .  npartitions;
```

```
        end
        @wm_end
```

Program 4.3. Work Manager for *iso* .

The syntax for work manager definitions is given in Appendix 2. The key points to note here are:

i) The flexibility of creating objects at load / runtime.

ii) The specification of "create-time" parameters in a separate *where* clause.

iii) Provision of consistency constraints in *epe* s.

iv) Use of the shorthand "$j := 1 \cdots npartitions$" to generate patterns of activity bindings. Linear lists, matrices, arrays e.t.c. can be easily constructed.

4.2.2 The Maze Problem

This example illustrates the use of a mixed binding strategy, using two objects: a task bag that is a work container object, and a two dimensional grid, in solving a problem that is inherently asynchronous. A maze (shown in Fig 4.2) can be represented by a two dimensional array of square elements, where each element has a vector OBS of four booleans indicating whether or not there is an obstruction to the north, south, east and west respectively. The task bag, *Pathpool*, is implemented in such a way that the grid elements are bound dynamically at runtime. The problem is to design an algorithm that finds a path from a start element (x_s, y_s), to a finish element (x_f, y_f); where the tuple (x_i, y_i) denotes the element that is x_i units along the horizontal axis and y_i units along the vertical axis. In addition to the boolean vector, we shall assume that each element has a boolean VISITED, signifying whether or not it has been visited, and a lock LOCK, for exclusive access.

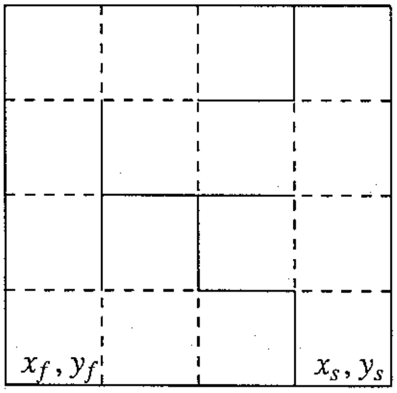


Fig 4.2. A maze

OBS is a three dimensional array of booleans, the third being the direction as described above. On a shared memory machine, the arrays OBS, VISITED and LOCK are simultaneously updatable objects, if they are simultaneously accessible by activities running on different processors.

In the course of our algorithm, workpieces that are candidate path segments starting at (x_s, y_s) are added to and removed from a task bag, Pathpool. Each activity gets a path segment from Pathpool, traverses an element in each direction where there is no obstruction, and puts the new paths thus generated into Pathpool. Pathpool is implemented in a similar way as *iso* in the canonical example presented earlier, the difference being that Pathpool supports both insertions and deletions. The algorithm for the program is shown below (only the essential portions have been shown).

```
begin
  ...

  Pathpool.get (path);
  while (not done) do begin

  let x, y represent coordinates of last element of path.
  if x, y = $x_f, y_f$ then begin
          print path
          Pathpool.clear
```

```
            (* this sets done to true *)
         exit
   end;

   b := false;
   lock (LOCK [x,y]);
     if not VISITED [x,y] then begin
          VISITED [x,y] := true;
          b := true;
        end;
   unlock (LOCK [x,y]);

   if b then begin
     for each direction d
       if not OBS [x, y, d] then
          append to path coordinates of neighbouring element
          in direction d, and put new path in Pathpool
   end;
  end;
end;
```

Some parts of the problem description, such as the array OBS, are shared by all activities in read mode; however, elements in the VISITED array are accessed in exclusive mode. The section of the program that updates VISITED is a critical section. By assuming a lock for each element we have pushed the problem of lock contention to a lower level of granularity than that of an entire array.

5. Implementation of work managers and extended path expressions

5.1 The overall schematic

Fig 5.1 shows a schematic of the various phases in the transformation, translation and execution of a user program. The preprocessor, which like the canned work managers is language and architecture specific, scans the user program and injects source code at appropriate places, for work container object creation, operations, synchronization, consistency constraints, and SPMD constructs.

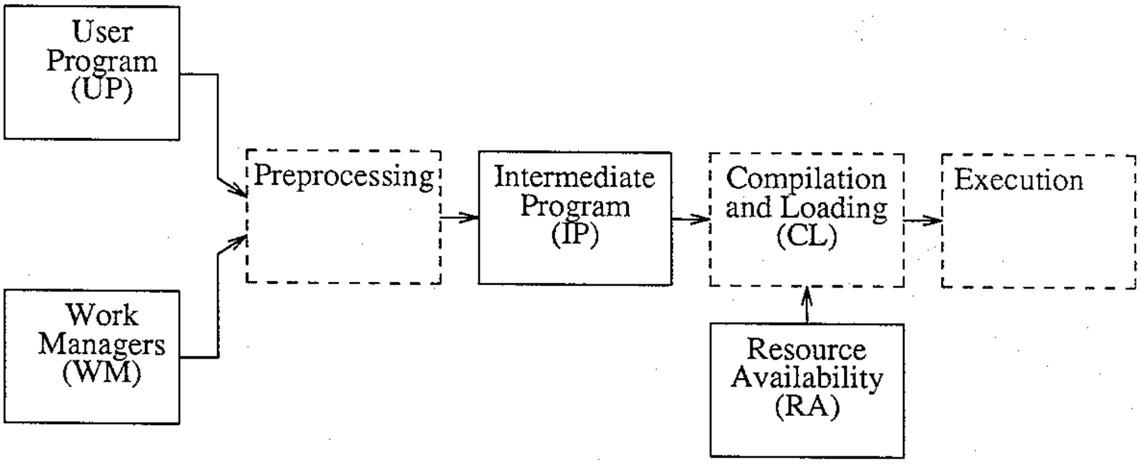

User Program (UP)
Work Managers (WM)
Preprocessing
Intermediate Program (IP)
Compilation and Loading (CL)
Execution
Resource Availability (RA)

Fig 5.1. Schematic of various phases in the execution of a user program

The intermediate program (IP) is in the source language, and is generated by the preprocessor following algorithms that are given later in this section. This program is then compiled and loaded onto the parallel machine. The parameters *nacts* and *myid* get bound at this point. There are *nacts* invocations of IP, bearing identifications from 1 through *nacts*, and a proportional number of invisible restructuring activities (if needed). The resource availability is a table of free processors available, possibly accompanied by some indication of how to map activities onto processors. The most significant task performed during execution, from our perspective, is creation of partitioned objects. At this time the object is bound to subsets of activities, according to the specification in the *epe* portion of each work manager used by the user program. The algorithm for the Preprocessing phase is given in Algorithm 5.1. This algorithm calls a procedure to emit code for synchronization from the *epe* s. This algorithm is presented in Algorithm 5.2.

5.2 The preprocessor

(The following algorithm is aimed at a target language that supports dynamic memory allocation.)

Algorithm 5.1.

Scan the user program (UP) from left to right. For each work container object (*wco*) that has been declared in the *var* section in UP perform steps [1] through [9].

[1] If *wco* is followed by actual parameters in the declaration itself (see example_wco in Algorithm 4.2) then

> generate a create call with the name and actuals from the declaration portion. This call is embedded at the start of the generated IP.

else

> enter *wco* in list of objects to be (possibly) created at runtime.

[2] Say *wco* is of type *wman*. Locate *wman* in WM.

[3] Generate a create procedure for objects of type *wman* as follows †:

> For each create-time-assignment in the create-time-bindings section of *wman*, if the r.h.s. is an integer expression, then generate code for the assignment.
>
> Otherwise, generate code to evaluate the limits of the range, i.e., the start and limit. Generate a "for" loop using a new subscript variable. Since nesting of for loops is from left to right, no lookahead is required at this stage.

† Note: The code generated by this portion of the Preprocessor algorithm will be executed by only one processor, by means of a serial section. The variable bindings thus generated at create time will be effective for all activities, throughout the lifetime of the object. In this algorithm, we have assumed shared memory for storing these variables. In a message based implementation these bindings must be broadcast to all activities through send / receive statements, before further execution.

Generate code to allocate a node for each component of the object implementation state (defined in section 3.5 of chapter 3) at each execution of the innermost "for" loop. The nodes are linked sequentially in the order of their generation††.

[4] Generate code for activity bindings as follows:
(* Each activity executes the code generated by this section *)

Generate code to create a list of node pointers called nlist. For each "invoker in" definition in the *epe*, generate a nested "for" loop structure, as in [3]. In the innermost loop, generate code to do the following:

Advance in the list of nodes, updating the nodeptr.

Evaluate limits of the range specified in the "invoker in" phrase, using values of loop parameters from the current iterations. If myid is within the range, then add nodeptr to nlist.

Note: The code generated by the above section has the capability to perform a create time binding between an activity and several nodes, based upon the specifications in the *epe*.

[5] Generate synchronization code - call Algorithm 5.2

[6] Generate code to check consistency constraints.
For each node generated, allocate a lock for mutual exclusion. Generate code to lock the node and evaluate the consistency constraint under mutual exclusion. Each operation must pass through this check before being allowed access to the node. If a constraint spans across more than one node, then lock each node in the order of their generation. Evaluate the constraint and release the locks. If the constraint is true then allow the operation. If it is false, deny the operation by sending back a *failure* reply.

[7] Generate remaining work manager procedures as they are in WM. The

†† If other component relations are desired, these must be programmed explicitly by the programmer (i.e. the work-manager-writer).

procedures are sense, get, put, init and clear.

[8] Generate code for the remaining SPMD constructs (such as locks, barriers etc.) if such constructs have been used.

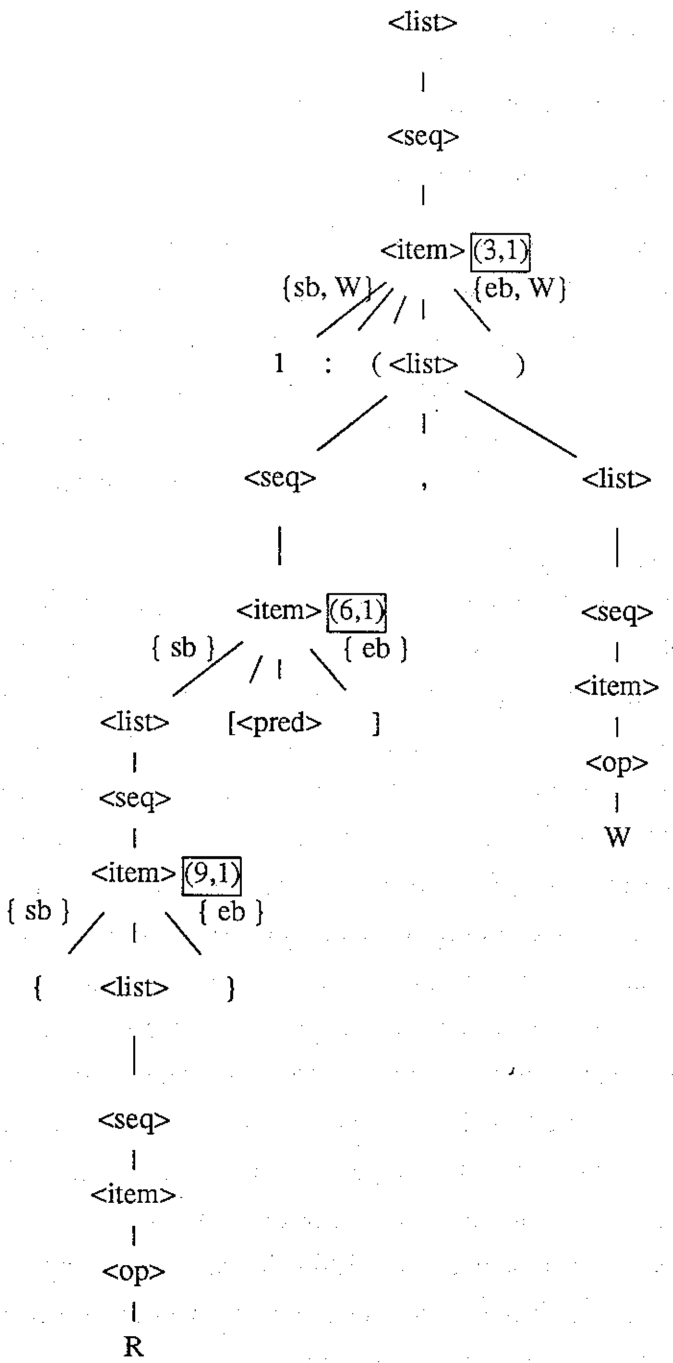
[9] Generate code for restructuring operations.

5.3 Synthesis of synchronization code

In this section we present an algorithm for automatically generating code for an open predicate path expression, i.e. the synchronization portion of an *epe*. This algorithm has been adapted from [Oldehoeft '84] and [Headington '85] to suit work managers and user programs written in the SPMD programming style. We discuss some aspects of the synchronization sub-expressions and their parse trees below, but for a more detailed account of this topic, the reader is referred to the above papers.

The synchronization portion of an *epe* is called an open predicate path expression, abbreviated *oppe*. Given an *oppe*, we can generate a parse tree by any parsing technique, following the rules of the grammar restricted to *oppe* s. Fig 5.2 shows an example of such a parse tree for the expression 1 : ({ R } [invok (W) - perm (W) = 0] , W). This expression denotes multiple reader, single writer synchronization with writers' priority. The basic idea behind our parsing and code synthesis algorithm is to generate controllers, or procedures, that implement the synchronization necessary for each of the four sequencing constructs in path expressions (derestriction or bursts, sequencing, predicates, and restriction). Once the controllers are generated, they are interconnected by properly placing calls in procedures that implement the basic operations. Except for the sequence controller, all other controllers are demonstrated by the example in Fig 5.2. Since we do not provide a full fledged object specification language, the consistency constraints are written as boolean expressions in the target language of the path expressions. We assume that an operation will not appear twice within the same path expression. We feel that this is a reasonable assumption for most synchronization problems (and one made by most other implementations that we know of).

Fig 5.2. Parse tree for 1:({R}[invok(W)-perm(W) = 0],W)



The predicate specifies writer priority. The left and right sets of the controller nodes have been shown as pre and post subscripts. The labels of the controllers are shown in boxes alongside. The predicate-event-count-set (pecs) operator-set (os) (definitions of these are given in Appendix 3) of the predicate controller node are as follows: pecs(n) = { invok(W) , perm(W) } and os(n) = { R }, where n is the predicate controller labelled (6,1).

Algorithm 5.2. Generation of Synchronization Code

Input: The synchronization portion of an *epe*, called *oppe*.
Output: Synchronization code to be embedded in the intermediate program (IP), for implementation of access control and synchronization at each node.

[1] Create the parse tree of the input by any parsing technique. Call this parse tree T.

[2] Let S denote the set of terminal symbols { "{", ";" , ":" , "]" }. For each internal node n in T, if $n' \in$ S, and n' is the child of n, then n is called a controller node. A controller node is called a *burst*, *restriction*, *predicate* or a *sequence* controller depending on whether it has a "{", ":", "[", or a ";" child, respectively. Label each controller node by a pair (d,p), where d is the depth and p is its position from the left. The labels of the controller nodes in the tree in Fig 5.2 are shown in boxes alongside the nodes. For example, the restriction controller is labelled (3,1).

[3] Find the left-set and right-set of each controller node. Intuitively, the left-set and right-set are the operations that a controller may allow as the very first and very last operations respectively, in its current control sequence. For each controller node n in T, $ls(n)$ and $rs(n)$ denote the left and right sets of the node respectively. Definitions for the left-set and right-set of parse tree nodes are given in the Appendix 3. The left and right sets of the controller nodes are shown in Fig 5.2, as pre and post subscripts of the node concerned. For example, the left-set of the controller labelled (3,1) is { sb, W }. This means that the operations that may execute first in the subpath enforced by this controller are the write or a start-burst pseudo-operation signifying the beginning of a burst of reads.

[4] Let n be a predicate controller node. The predicate-event-count-set of a predicate controller node is the set of signals or event counts that are specified in the predicate. The operator-set of a predicate controller node is the the set of operators in the leaves of the subtree of the parse tree rooted at the node. We shall use $pecs(n)$ and $os(n)$ to denote respectively the predicate-event-count-set and operator-set of n. $pecs(n)$ can easily be constructed at compile time by scanning the predicate represented by node n. $os(n)$ can be generated at compile time by traversing the parse tree.

[5] For each controller node in T, generate code for the controller according to the general forms of controllers shown in section 5.4 of this chapter. For each controller node in T, we thus have a label and a piece of code which will be embedded in IP. It remains to be seen how we must connect up these controllers, so that taken together, they implement the synchronization abstractly specified by the *oppe*.

[6] This is the *Sandwich* algorithm for connecting up the controllers. For each leaf node in T that is an operator, generate a procedure as follows.

i) Generate code for the access of the data.

ii) Traverse up the parent links of T from the operator towards the root until either a burst controller or the root node is encountered. Discontinue traversal at this point and go to the next leaf operator. During the traversal, generate calls to each controller encountered on the way (these controllers have been identified in step [5]). The calls are placed either before, or after, or both before and after the code accessing the data, thereby enveloping the accessing code in a *sandwich*. The calls are generated according to the following rules:

  a) If a sequence controller is encountered and the operator is in the left-set of the controller, then generate a call *after* the data access with parameter *ack*.

  b) If a sequence controller is encountered and the operator is in the right-set of the controller, then generate a call *before* the data access with parameter *invok*.

c) If a restriction controller is encountered and the operator is in the left-set of the controller, then generate a call *before* the data access with parameter *invok*.

d) If a restriction controller is encountered and the operator is in the right-set of the controller, then generate a call *after* the data access with parameter *ack*.

e) If a predicate controller is encountered (then the operator must be in the operator-set of the controller by the definition of operator-set), then generate a call *before* the data access with parameter *invok*.

f) If a burst controller is encountered, then generate a call *before* the data access with parameter *invok*, and a call *after* the data access with parameter *ack*. Also, discontinue the traversal at this point, since a burst controller has been reached.

iii) For each burst controller identified in step [5], execute step [6], creating a *sandwich* which is embedded in the burst procedure already generated in step [5].

iv) For each predicate controller, if a signal appears in the predicate-event-count-set (pecs) of the controller, then deal with it as follows:

- * if the signal is invok(X), then generate a call to the predicate controller from procedure X immediately upon entry.
- * if the signal is perm(X), then generate a call just before data access.
- * if the signal is ack(X), then generate a call just after data access.
- * if the signal is rep(X), then generate a call just before exit.

5.4 Code templates for controllers

The code templates presented here are for use in conjunction with Algorithm 5.2.

```
procedure restrict_contid (signal)
begin
 if signal.type = ack then
```

```
  begin
    nacks_contid := nacks_contid + 1;
    if ninvoks_contid - nacks_contid >= k then
      begin
        dequeue (contid, id);
        awaken (id)
      end;
  end;
if signal.type = invok then
  begin
    ninvoks_contid := ninvoks_contid + 1;
    if ninvoks_contid - nacks_contid > k then
      begin
        enqueue (contid, myid);
        sleep (myid)
      end;
  end;
end;
```

The template above is a generic template for a controller that implements a restriction operator. The subpath implemented is "k : (<list>)", where <list> is an arbitrary subpath. The *contid* field in various identifiers is filled at compile time with the label of the controller from the parse tree of the *oppe*. This subpath is depicted by the node n shown in the Fig 5.3.

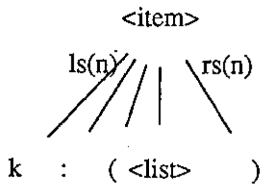


Fig 5.3. A restriction controller

The event counter invok_contid keeps count of the number of invocations operations so far. Similarly ack_contid keeps count of the number of acknowledgements so far. If an access request (operation invocation) is in ls(n) then it is channelized to the controller for n *before* access is permitted. Similarly, an acknowledgement of an operation in rs(n) is channelized to the controller for n *after* access is over. This channelizing is done by the sandwich algorithm.

```
procedure sequence_contid (signal)
begin
 if signal.type = ack then
   begin
     nacks_contid := nacks_contid + 1;
     if ninvoks_contid >= nacks_contid then
       begin
         dequeue (contid, id);
         awaken (id)
       end;
   end;
 if signal.type = invok then
   begin
     ninvoks_contid := ninvoks_contid + 1;
     if nacks_contid < ninvoks_contid > k then
       begin
         enqueue (contid, myid);
         sleep (myid)
       end;
   end;
end;
```

This template implements a sequence operator. The subpath implemented is "<item> ; <seq>", where <item> and <seq> are arbitrary subpaths. The subpath is shown in the Fig 5.4.

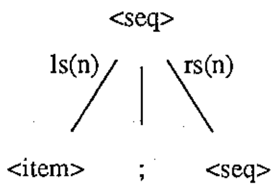


Fig 5.4. A sequence controller

If an access request (operation invocation) is in ls(n), which is also the rs (<item>), then it is channelized to the controller for n *before* access is permitted. Similarly, an acknowledgement of an operation in rs(n), which is also ls (<seq>), where <seq> is the sibling of <item>, is channelized to the controller for n *after* access is over. As before, the channelization is achieved by the sandwich algorithm.

```
procedure predicate_contid (signal)
begin
 if signal ε predicate-event-count-set (contid) then
   begin
      increment appropriate event count
      evaluate predicate
      if predicate is true then
       repeat
           dequeue (contid, id);
           if id <> nil then
              awaken (id)
       until (id = nil)
   end;
 if signal ε operator-set (contid) then
   begin
      evaluate predicate
      if predicate is false then
        begin
           enqueue (contid, myid);
```

```
            sleep (myid)
          end;
      end;
   end;
```

This template implements a predicate controller. The subpath implemented is "<pred>". Certain portions of this template have been specified in prose form. The reason for this is that the code depends upon the exact contents of pecs(n) and os(n), the predicate-event-count-set and the operator-set, are unknown until compile time. Depending upon the precise nature of these sets, appropriate code is generated. A hand-executed example shown in Appendix 4.

```
procedure burst_contid (signal)
begin
 if signal.type = invok then
   begin
     if burst_count_contid = 0 then      (* start of new burst       *)
       begin
           .
           .
         embedded invocations upon ancestor controllers
         following the sandwich algorithm. Note that all the
         queueing is internal to the ancestral controllers.
           .
           .
         burst_count_contid := 1;
       end
     else                    (* continuation of old burst *)
       burst_count_contid := burst_count_contid + 1;
   end;
 if signal.type = ack then
   begin
     if burst_count_contid = 1 then      (* end of a burst          *)
```

```
            begin
                .
                .
              embedded acknowledgements upon ancestor controllers
              following the sandwich algorithm.
                .
                .
              burst_count_contid := 0;
            end
          else                  (* continuation of a burst  *)
            burst_count_contid := burst_count_contid - 1;
      end;
    end;
```

We deal with a burst controller exactly as an operator with special pseudo events called start-burst and end-burst, which are used to synchronize events with ancestral controllers. As with the predicate controller, certain portions of the burst controller have been specified in prose form. Invocations upon and acknowledgements to ancestral controllers are embedded in these portions at compile time. These code segments cannot be determined earlier, since they depend upon the exact contents of the subpath expression enclosed in the braces, signifying a burst. The key point to note in the burst controller is the absence of queueing. Queueing, if necessary, will take place within ancestral controllers. A "hand-executed" output of algorithm 5.2 for the *oppe* in Fig 5.2 is shown in Appendix 4.

6. Conclusions

To summarize this chapter, we have defined a class of simultaneously updatable objects and provided a structuring concept called the work manager, which is useful in setting up a class of computations with relaxed consistency requirements. We have introduced the notations of extended path expressions to gain additional concurrency at the level of components of an object. Work managers are suitable for inclusion in global libraries on parallel machines, and their synchronization code can be automatically generated by means of simple algorithms that utilize parse trees of extended path expressions.

Chapter 5 - Weak Priority Queues

1. Weakened specifications

Besides partitioning, simultaneous update can also be introduced if we can accommodate the use of objects with weakened specifications (at external or internal levels). For example instead of using a strictly FIFO queue, some applications can do with objects that have weakened serial specifications such as the semi-queue object discussed in chapter 4 section 2.3.

The strict priority queue represented by a heap is an inherently serial object in that there is no scope for simultaneous update in its object implementation states. The objects of interest to us in this chapter and the next are weak priority queues. These are simultaneously updatable objects whose serial specification is that of a weak priority queue in the sense the item returned is not the largest value but *one of* the largest values. We address the problem of designing a parallel algorithm in which a system of concurrent activities simultaneously update such shared priority structures.

As expected the performance of these objects is much better than that of strict priority queues. With further weakening at an internal level it is possible to obtain implementations with more scope for simultaneous update. Depending upon what degree of weakening is acceptable by an application, different internal representations and algorithms with different degrees of simultaneity, may be developed.

We present two parallel algorithms for priority structures and prove their correctness. The first structure developed is the concurrent heap (CHEAP), and it is obtained by weakening the requirements of a regular heap. The weakened conditions are guaranteed by the implementation by means of a coloring scheme that is intrinsic to the working and proofs of our algorithms. A straightforward derivative of the concurrent heap is the software banyan (SBAN), which may be thought of as a forest of overlapping heaps. The internal specification of software banyans is weaker than that of concurrent heaps. Software banyans follow the same strategy for concurrent mutation of the structure as the concurrent heap but the specific algorithms for the SBAN are quite different from those of the CHEAP. By specifying a variable percentile level, our specifications may be made stronger or weaker. These structures are proposed as possible basic building blocks for implementation of resource allocation in operating systems.

In the remainder of this section we present the serial specification of the weak priority queue. In section 2 we present sequential heap algorithms and consistency properties based upon a coloring scheme that is useful in formulating the parallel solution. In sections 3 and 4 the concurrent heap and software banyan algorithms are developed. We also derive some structural properties of the software banyan and characterize its mapping. In section 5 we study and predict the performance of priority structures using queueing theoretic models. We end with some notes on related work and potential applications in section 6.

1.1 Serial specification of the weak priority queue

The weakening of a priority queue is captured in the notion of percentile values. Associated with each delete operation is an input argument called the *percentile* that specifies the relative magnitude of the deleted item required. The smaller the percentile value the stronger is the specification. Let D be a set of values in the domain of a multi-set. Let p denote an integer which represents the percentile level of a "delete" request. If a delete is invoked with a percentile $p = 1$, then the item returned in response to the invocation must belong in the top one percentile of the priority queue.

S_x : The set of all multi-sets of items from domain D.

I_x : { }, the empty multi-set.

O_x : { $\ll$ insert, v, A, x >, < ok, NULL, $A, x \gg$,
$\ll$ delete, p, A, x >, < ok, $v', A, x \gg$,
$\ll$ delete, p, A, x >, < sorry, NULL, $A, x \gg$
where $v, v' \varepsilon D$, $1 \le p \le 100$ and $A \ \varepsilon AC$ }

T_x (s, $\ll$ insert, v, A, x >, < ok, NULL, $A, x \gg$) =

$s \cup_b \{v\}$ if ($v \; \varepsilon \; D$) where $\cup_b$ denotes bag union

$T_x \, (s , \ll \text{delete}, p , A , x >, < \alpha, v', A , x \gg) =$

$s \; -_b \{v'\}$ if ($v' \varepsilon \, s \;\; \Lambda \;\; \alpha =$ "ok" $\Lambda \mid \{ \; y : y \; \varepsilon \, s$

$\Lambda \, y > v' \; \} \mid \; < \; p \, * \; \mid s \mid$)

s if (α = "sorry" $\Lambda \; s = \{ \}$)

where $-_b$ denotes bag difference

A weak priority queue is similar to a bag in that the exact order of arrival is not maintained. It is different from a bag in that a guarantee is made about the relative magnitude of a deleted item.

1.2 Simplifying assumption

For the remaining sections of this chapter we assume (for historical reasons) that delete requests arrive with a *wildcard* specification which leaves the percentile level unspecified (null). This is for convenience in implementation. We outline (in section 3.5.1) a trivial alteration of our algorithms that makes it possible to accommodate non-null percentiles.

2. The heap as a strict priority structure

A *priority queue* is a *"largest-in-first-out"* data structure. The most widely used implementation of priority queues is with a *heap* data structure [AHU '74], i.e. a complete $k\!-\!ary$ tree of depth d, with the property that the data at any node is not less than, in priority order, the data in each of its children. This property is known as the *heap property*.

If we use the array representation of a $k\!-\!ary$ tree, the root occupies array element 1, and the children of the node at array element i, (for convenience called node i), occupying array elements $k^* (i-1) + 2, \ldots, k^*i + 1$. The most studied special case of this is the binary heap, with k=2. In this case the children of node i are nodes $2i$ and $2i$+1, and the parent of node i is node i *div* 2. Let the nodes of the heap PQ be in array elements $1, \ldots, N$, where N is the last element in the bottom of the heap i.e. all elements to its right and below it are empty (Fig 2.1). Each node contains three fields: a data field, a color field and a lock field. The data field contains a value from a linearly ordered set. Two nodes may have the same value in the data field. The color field may be "red", "blue" or "yellow". The lock field of a node is used for obtaining writer exclusion to a node. Only one activity may lock a node at a time. Once an activity locks a node, then and only then, it may modify the data and color fields of the node.

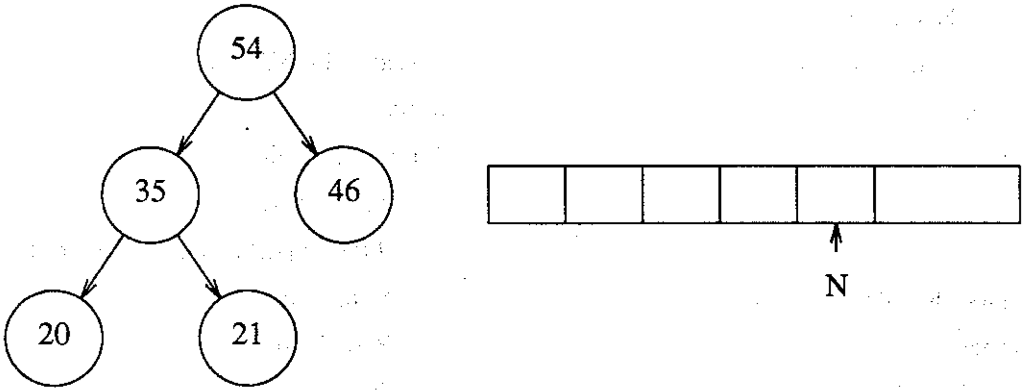


Fig 2.1. A Heap and its representation

The contents of the data, color and lock fields of a node in the i^{th} array element, are denoted by PQ[i].data, PQ[i].color and PQ[i].lock respectively. We shall assume that there is a value called a *null value* (denoted by *null*), that is smaller than any of the possible values in the data field. We shall sometimes use PQ[i] to refer to the data field of the i^{th} element of PQ, when it is obvious by the context that we are discussing the data field.

2.1 The sequential heap algorithms

We present the sequential heap insertion and deletion algorithms formulated in a recursive fashion.

(* Delete checks that there is at least one item *)

```
procedure Delete (item)
begin
  if (N = 0) then item := null
  else begin
     item := PQ [1].data;
     PQ [1].data := PQ [N].data;
     N := N - 1;
     PushDown (1);
  end;
end;
```

```
procedure PushDown (n)
begin
   if (n * 2 ≤ N) begin
      Let Max be larger child of n
      if PQ [n].data < PQ [Max].data) begin
         Swap (n, Max);
         PushDown (Max);
   end;
  end;
end;
```

```
procedure Insert (v)
begin
  N := N + 1;
  PQ [N].data := v;
  PushUp (N);
end;
```

```
procedure PushUp (n)
begin
   if (n > 1) begin
      p := n div 2;
      if (PQ [p].data < PQ [n].data) begin
      Swap (p,n);
      PushUp (p);
   end;
  end;
end;
```

2.2 Requirements for Correctness

The serial algorithms are *correct* if they always preserve consistency of the heap structure, and if every delete operation returns the largest item in the heap. We shall call these two properties the *heap consistency* (HC) and *delete consistency* (DC) properties, respectively. As said earlier, the *heap property* is preserved by a node if it is greater than or equal to its children. We cannot expect the heap property to hold at all nodes during an insert or a delete. We therefore stipulate the following correctness properties.

HC: The heap property is always satisfied at each node except for those holding items that are being inserted or deleted. For any such exception, the heap property holds between its parent and its children.

DC: A deletion operation always returns the highest value in PQ.

2.3 Introduction of colors

To be able to state HC and DC more precisely, we introduce a coloring scheme as follows. Initially, all elements in the array PQ are colored *blue* (to indicate that they satisfy the heap property), and are assigned the null value. The insertion algorithm colors the new item *red* and inserts it in the first available empty node, i.e. a node that contains a null value. It then pushes up the red value recursively. The deletion algorithm picks up the first non-null blue value (pointed to by N), colors this value *yellow*, puts it in the root node and pushes it down the tree recursively. A yellow value *stops* when it is larger than both its children, or it is in the bottom level. A red value stops when its parent is larger than itself, or it is in the root node. When a value stops, it turns blue. Note that the coloring scheme does not materially alter the flow of control of the programs, it merely makes assertions easier to specify. The serial HC and DC properties are given below:

First we must define a few predicates.

$$blue(x) \equiv PQ[x].color = blue$$

$$gib(x,y) \equiv PQ[y].color = blue \rightarrow PQ[x] \geq PQ[y]$$

This is the *greater if blue* predicate. It states that if the color of a child of x is blue, then the value at x is larger than the value at the child. Note, that the color of x is unspecified.

$$h(x) \equiv [2x \leq N \rightarrow gib(x,2x)] \wedge$$
$$[2x+1 \leq N \rightarrow gib(x,2x+1)]$$

This is the *heap* property. It states that the value at x is larger than the values in its blue children.

$$gh(x,y,z) \equiv 1 \leq x \leq N \wedge [\ 1 \leq y \leq N \rightarrow PQ\ [x] \geq PQ\ [y]]$$

$$\wedge [\, 1 \le z \le N \rightarrow PQ\ [x] \ge PQ\ [z]]$$

This is the *generalized heap* property. It is similar to the heap property except that x, y and z need not be related as parent and children.

$allblue \equiv \forall k : 1 \le k \le N : PQ\ [k].color = blue$

$abx(x) \equiv \forall k : 1 \le k \le N \wedge k \ne x : PQ\ [k].color = blue$

This is the *all blue except* property. It states that all nodes in the heap except for x are blue.

SHC: (Serial Heap Consistency)

$$\{N \ge 0 \wedge \forall k : 1 \le k \le N : blue\ (k) \rightarrow h\ (k)$$
$$\wedge \forall k : 2 \le k \le N : \neg\, blue\ (k) \rightarrow gh\ (k\ div\ 2, 2k, 2k{+}1)\}$$

If a node in the heap is blue, then it satisfies the heap property. If it is not blue, then its parent, (if it exists), satisfies the generalized heap property with respect to its children.

SDC: (Serial Delete Consistency)

$\forall k : 1 \le k \le N : blue\ (k) \rightarrow item \ge PQ\ [k]$ at the termination of a delete.

The *termination* of a delete is the point when control returns to the calling program that had invoked the delete operation upon the priority queue. The Serial Delete Consistency property claims that at the time of termination, the value returned in the variable *item*, is the largest value in the heap.

For convenience in proving correctness, the PushDown and PushUp routines may be made iterative and incorporated inside the body of the Delete and Insert routines respectively. The revised algorithms are given below. Note that the Swap routine is assumed to swap both color and value fields. The proofs these serial algorithms (i.e. proofs of SHC and SDC) are straightforward, and omitted from this thesis.

```
procedure Delete (item)
begin
 if N = 0 item := null
```

```
procedure Insert (item)
begin
 N := N + 1;
```

```
  else begin
     item := PQ[1].data;
     PQ[1].color := "yellow";
     PQ[1].data := PQ[N].data;
     N := N - 1;
     x := 1;
     while $\neg h(x)$ do begin
        if $gh(2x, x, 2x+1)$ begin
           lc := 2x;
           sc := 2x+1;
        end;
        else begin
           lc := 2x+1;
           sc := 2x;
        end;
      swap (PQ[x], PQ[lc]);
      x := lc;
     end;
    end;
    PQ[x].color := "blue";
end;

   PQ[N].color := "red";
   PQ[N].data := item;
   x := N;
   p := x div 2;
   while (PQ[p].data < PQ[x].data) do begin
     swap (PQ[p], PQ[x]);
     x := p;
     p := p div 2;
   end;
   PQ[x].color := "blue";
end;
```

3. Development of the concurrent heap algorithm

The only component in the decomposition of the heap is the entire PQ array, since this array cannot be simultaneously updated. Any attempt to remove this requirement will result in a structure that is not a strict priority structure in the sense that the item returned by a delete may not be the largest values in the heap. Let us assume that we have an application that can tolerate the use of a weakened priority structure in that it does not matter if the value returned by a delete is not the highest value as long as some guarantee can be made about its relative magnitude. With this relaxed specification we decompose the heap array into its individual components (array elements) and design a parallel algorithm to maintain this structure (called a concurrent heap or CHEAP) concurrently. The development is outlined below. First we state a fundamental lemma about deadlock avoidance. Sometimes (when it is clear from the context) we shall use node or component to mean the component value function (*Cval*) associated with a given component.

Let O = $\{ o_1, o_2, \ldots, o_n \}$ be a set of n shared components, each of which can be locked in writer exclusive mode, and A = $\{ a_1, a_2, \ldots, a_k \}$ be a set of k activities that have access to the components in O. We say that the activities are *deadlocked* if there are $k \leq n$ components, which we shall call $o_{j_1}, o_{j_2}, \ldots, o_{j_k}$ such that $\forall$ $i : 1 \leq i \leq k$, activity a_i has locked component o_{j_i} and is waiting to lock $o_{j_{i+1}}$; and activity a_k has locked component o_{j_k} and is waiting to lock component o_{j_1}.

Lemma 4.1.: Let O be an *ordered* set of components, $\{o_1, o_2, \ldots, o_n\}$, each of which can be locked with a writer exclusive lock. Then, given a set of activities, each of which locks and unlocks subsets of O *in the order that they appear in* A, with no lock following an unlock in a given activity, these activities are free from deadlock.

Proof : Let us assume that there are m deadlocked activities $a_1, a_2, \ldots, a_m$ such that activity a_i has currently locked the set of component S_i and is waiting for a component from set $S_{(i \ mod \ m + 1)} : 1 \leq i \leq m$. Without loss of generality, assume that set S_1 has the smallest component among all these sets. Since activity a_1 is deadlocked, all the elements in S_1 must be ordered less than some element in S_2. Let C_2 be that element. Similarly, since p_2 is deadlocked, all the elements in S_2 must be ordered less than some element, say C_3, in S_3. Carrying on this argument, all the elements in S_m must be ordered less than some element, C_1 in S_1. This gives us a cycle of elements $C_i : 1 \leq i \leq m$ such that each element is ordered less than its successor in the cycle, and C_m is ordered less than C_1. This is impossible from the manner in which set O was constructed. Thus, by contradiction, there cannot be a deadlocked set of activities.

3.1 The parallel algorithm

3.1.1 The Work Queue

An auxiliary data structure called a Work Queue (WQ) is need by the algorithm. This work queue consists of an array of work nodes[††]. Each work node contains two fields; an index field and a color field. The index field contains a natural number, representing an index (a pointer) into the Priority Queue; the color field may be "red" or "yellow". For the rest of this thesis we shall assume that activities can invoke the operations *enqueue* and *dequeue* upon the Work Queue. In other words we shall not specify the exact behaviour of these operations as they are fairly standard.

3.1.2 Concurrent manipulation of the heap

Three types of operations simultaneously mutate the data structures. These are the Insertion, Deletion and Restructuring operations. *Insertion* is the operation of adding a data value to the heap. *Deletion* is the operation of removing the highest priority value from the heap. *Restructuring* is the operation of interchanging the values of a group of heap nodes when these values are observed to be violating the heap property.

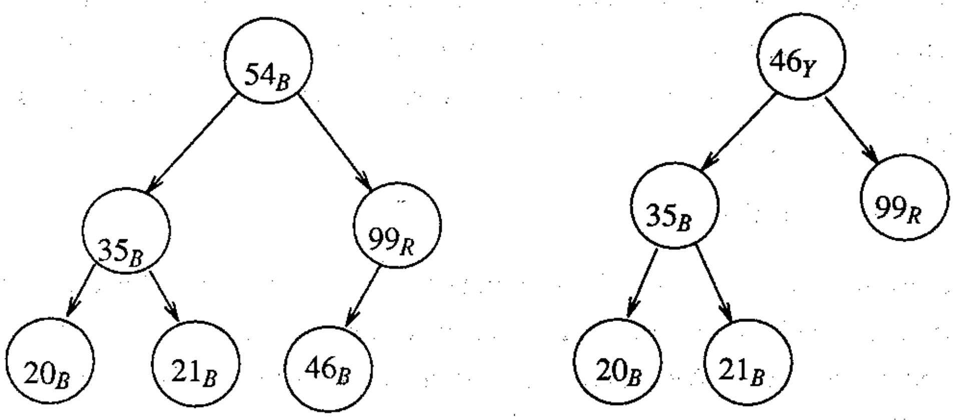
The interpretation of the sequential heap algorithms using colored values, provides the intuition for the invariant for the concurrent priority structure. Red and yellow values may violate the heap property, but a blue value must satisfy this property with respect to all other blue values that appear in the subtree rooted at the node in which it appears. Our algorithms are guided by this invariant. In particular, the algorithms make *atomic* changes to groups of nodes, all of which are locked, to guarantee freedom from interference. Each atomic change moves the priority structure closer to a heap structure with all values colored blue. To avoid deadlock, we make sure that the variables and the nodes in the tree are always locked in a specific linear order, and thereby exploit the above lemma. Specifically N is ordered (and locked) before the elements of the PQ array, and the ordering among the array elements is given by the natural ordering of their indices.

[††] The term *work* is used in the same sense as in the work container objects of chapter 4 section 2.

3.1.3 A naive approach

Our first attempt at making the sequential algorithms parallel is one in which the activity performing the insertion and deletion are also the ones that restructure the heap. An inserting (deleting) activity will have to prevent access to the inserted (substituted) red (yellow) value until this value has become blue. By locking a group of nodes consisting of a parent node and its children, updating the contents of these nodes in an appropriate manner, and then releasing the locks, we can guarantee that the heap property is preserved with each update of heap nodes. Also, since modifications can only occur when the appropriate nodes are locked, an activity cannot "see" the incomplete results of another activity. However, since decisions may be taken only with respect to *blue* values, this approach may lead to livelock. We take *livelock* to mean a situation when a set of activities are able to lock protected data, but are unable to do any useful computation, (i.e. make progress), once they do so, because of semantic constraints on the data. Consider the case of the data in Fig 3.1. The steps in an execution sequence may appear as below. Note that $\mathrm{PQ}[k] = V_c$ means that the k^{th} element of PQ is assigned the value V and the color c. Some intermediate steps have been omitted for clarity.

Step.	a_1	a_2
1.	Insert (99)	
2.	PQ [6] = 99_R	
3.	PushUp (6, "red")	
4.	Swap (99_R, 46_B)	
5.	PushUp (3, "red")	
6.		delete (item)
7.		item = 54
8.		PQ [1] = 46_Y
9.		PushDown (1, "yellow")



Before step 6 After step 9

Fig 3.1. States showing development of livelock

By our naive algorithm, a_1 (charged with inserting and pushing up 99_R) must wait until the parent of the node containing 99_R turns blue, repeatedly locking and unlocking the two nodes involved. Similarly, a_2 (charged with deleting an item and pushing down 46_Y), must wait until both the children of the node containing 46_Y become blue. Both a_1 and a_2 continuously encounter a value that is not blue (see Fig 3.1). Thus neither can make progress.

In general, say A, a parent node contains a yellow value being pushed down by activity a_1, and B, a child node of A, contains a red value being pushed up by activity a_2. Naturally, both a_1 and a_2 cannot wait indefinitely for the appropriate child / parent node to become blue, since this will amount to livelock. We will have to devise a protocol, whereby in such a situation, a given activity "wins". Without loss of generality, let us assume that a_1 has precedence over a_2. Let us also assume that for this particular example, the red value is greater than the yellow value, and both are greater than the sibling (or the siblings in case $k>2$) of the red value. Thus the red and yellow values should be swapped. If this swap were carried out, a_2 returning later, would discover that B no longer contained a red value, or to be precise, B no longer contained *the* red value that it (a_2) was charged with "pushing up". Furthermore, there would be no way for a_2 to *know* where its red value was, without some mode of communication between a_1 and a_2. This is unacceptable to us, since we do not wish to add to the complexity of the system, by adding communications channels between all pairs of potentially inserting and deleting activities.

3.1.4 A better algorithm

Inserting and deleting should be delinked from restructuring. Restructuring should be performed by several "maintenance activities" that concurrently manipulate (update) the heap. We regard the propagation of a red / yellow value by a single level, as a single unit of work. Units of work are contained in another object called the *Work Queue*, defined earlier. Items of WQ are tuples of the form *(a,b)* where *a* is an index into PQ and *b* is a color. The work represented by a tuple is as follows. If *b* is "red", then try to push up the red value at the a^{th} node of PQ up by one level, if it can be; if *b* is "yellow", then try to push down the yellow value at *a* by one level. Pushing a node up by a level means that the contents (value and color fields) of it and its parent nodes are swapped. Similarly pushing down by a level entails a swap of a parent node with one of its children.

3.2 Informal discussion of the concurrent activities

The concurrent heap system has one or more activities that may perform deletion and insertion and one or more activities that do only restructuring.

An insert proceeds as follows. Under mutual exclusion, N is incremented thus yielding a new array element, which is filled with the new item and colored "red". Then a restructuring workpiece, i.e. a pair of the form (i, "red"), where i is the array index, is enqueued in the work queue, WQ. The inserting activity exits the system at this point. The *window* of an insertion, i.e. the resources that an activity must possess in order to preserve consistency during an insert consists of N and PQ [N+1].

A delete proceeds as follows. Under mutual exclusion, the topmost element of the heap is examined to check if its color is non-yellow. If it is yellow, the delete activity releases its lock, thereby allowing some other activity to mutate the node.

In case the topmost element is non-yellow, delete removes this element, and puts in its place, the N^{th} element in the heap, regardless of the color of this element. It colors this element yellow, and enqueues a restructuring workpiece of the form (1, "yellow") in WQ. The deleting activity exits at this point. The window of a deletion consists of N, PQ [1] and PQ [N]. Upon deletion, the N is decremented after setting the deleted element to null.

A restructuring activity continuously removes workpieces from WQ and attempts to carry out the restructuring indicated by these workpieces. The window of restructuring depends on whether it is an insert restructuring operation (Fig 3.2a) or delete restructuring operation (Fig 3.2b). Essentially, restructuring consists of attempting to establish the heap property. Fig 3.3 illustrates an exhaustive enumeration of the possible restructuring scenarios.

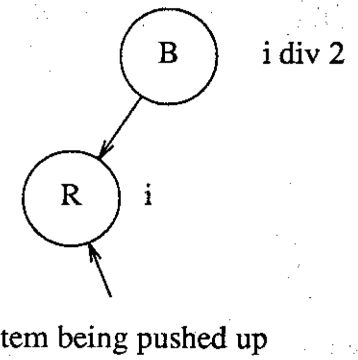


Fig 3.2a. Window of an insert restructuring operation. Restructuring workpiece is (i,red).

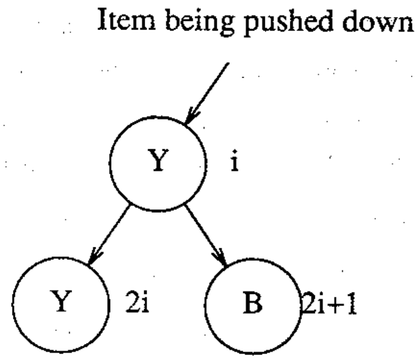
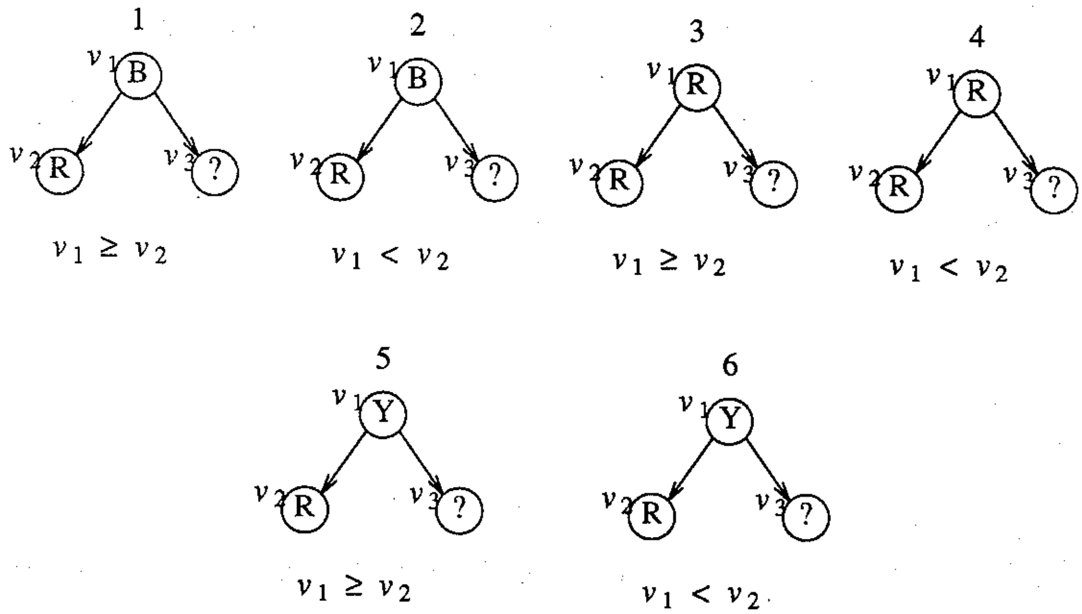
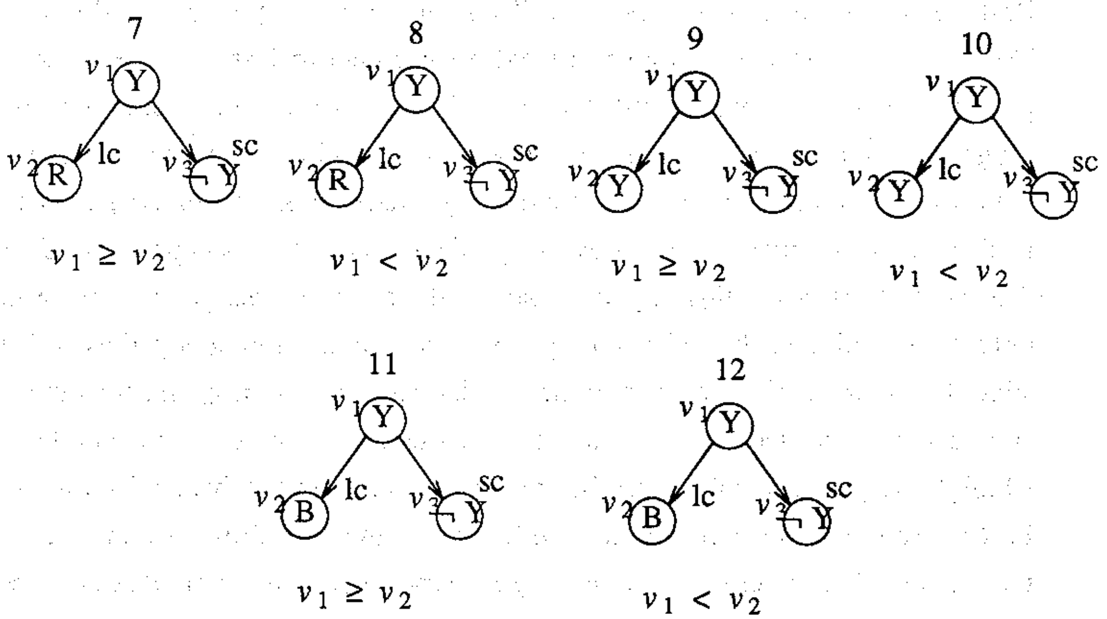


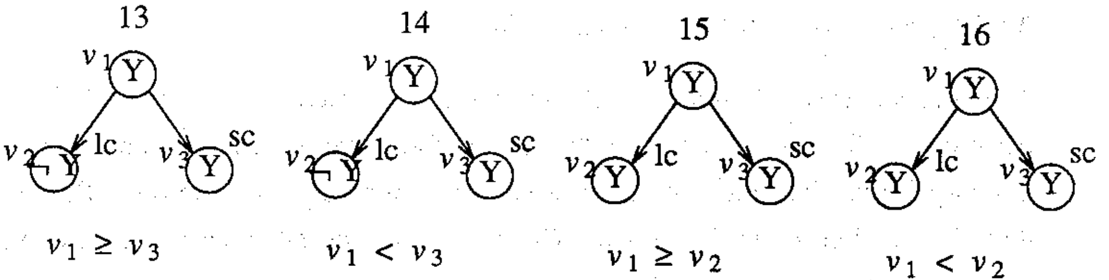
Fig 3.2b. Window of an delete restructuring operation. Restructuring workpiece is (i,yellow).



Cases 1 through 6 are for Insert restructuring operations. v_1 and v_2 denote the values of the parent and child nodes respectively. v_3 denotes the value of the other child (that is not examined by the inserting activity).



Cases 7 through 16 are for delete restructuring operations. Of these, cases 7 through 12 are for those cases in which the smaller child is not yellow. Note: *lc* and *sc* denote *larger child* and *smaller child* respectively.



Cases 13 and 14 are for the case where the larger child is not yellow and the smaller child is yellow. v_3 denotes the value of the smaller child. Cases 15 and 16 are for the cases where both children are yellow.

Fig 3.3 Exhaustive list of cases within the Restructuring activity.

3.2.1 Cases 1 through 6 of restructuring

The program statements corresponding to the cases in Fig 3.3 appear in procedures PushUp and PushDown, called by procedure Restructure (see algorithm below).

PushUp consists of the first six cases. These apply to an upward bound red value. The value in a red child may be ordered greater than or less than the value in its parent, which itself may be red, blue or yellow. There are three scenarios where progress may be temporarily stalled. When a red child being pushed up is smaller than a red parent, no progress is possible. Similarly when a yellow parent being pushed down is larger than a yellow (larger) child, no progress is possible. Finally, if a red child is larger than a yellow parent, the child and parent may be swapped only if the sibling of the child is not yellow. In case the sibling is yellow, it is possible that a larger blue value is lurking underneath this sibling, and thus the swap should not be carried out. Additional sub-cases are possible, (for example if the sibling is larger or smaller etc.), but rather than provide for these sub-cases in the PushUp routines, we prefer to deal with them in the PushDown routines. Thus, in the interest of simplicity, we do nothing if, during insert, a red child is observed to be greater than a yellow parent. In other words, in case of the above kind of 'conflict', PushDown has precedence over PushUp. The decision may seem a little arbitrary, but after repeated attempts this seems the most "natural" choice. We claim that this will lead to neither deadlock nor starvation (see proofs in appendix).

3.1.2 Cases 7 through 16 of restructuring

The first six cases (7 through 12), in PushDown, have to do with relationships between a yellow parent and its larger child, given that the other (i.e. smaller) child is non-yellow. The remaining four cases (13 through 16), have to do with the case that the smaller child is yellow. There are four possible cases, depending on whether the larger child is yellow or non-yellow.

We reiterate that the critical point in the restructuring algorithm is that we can say nothing at all, about the descendants of a yellow value. The invariant PHC' (stated later), mentions only *ancestors* of yellow values. Thus in our algorithms we must be very careful not to swap a non-yellow child with a yellow parent when siblings of the non-yellow child are yellow. If k, the arity of the tree is larger than 2, such a swap is permissible only when none of the siblings of the child node are yellow. The parallel algorithms are given below.

3.3 Additional notations

We use $\equiv$ to denote multiple assignment and Λ to denote boolean conjunction. We also assume that LAST denotes the lock for the memory location that contains the index of the last element of PQ. Unless otherwise specified p denotes the index of a parent node and c denotes the index of a child node. red(p) stands for the predicate (PQ[p].color = "red"). Similarly with blue(p) and yellow(p). Swap(c,p) is shorthand for PQ[c].data, PQ[c].color, PQ[p].data, PQ[p].color, $\equiv$ PQ[p].data, PQ[p].color, PQ[c].data, PQ[c].color; (p>c) is shorthand for the predicate (PQ[p].data > PQ[c].data), similarly (p<c) is shorthand for the predicate (PQ[p].data < PQ[c].data). For simplicity we assume that there are no duplicate values. predicate → begin statements end; is shorthand for: if "predicate" is true then perform "statements". We assume that the procedure will be exited immediately after the performance of "statements". @Lock (arg_list) and @UnLock (arg_list) are shorthand for the corresponding lock and unlock commands for each element of the argument list, in the order that they appear in the argument list.

Please note that the *Work Queue* must be locked and updated atomically for each Enqueue and Dequeue operation. The corresponding lock and unlock instructions are assumed to be inside the routines Enqueue and Dequeue. We omit definitions and declarations where obvious.

```
procedure Delete (item)
begin
 filter (item); (* filter out the trivial cases *)
 if (item <> null) begin (* at least one item *)
   if (item = undefined) begin  (* at least two items *)
       item := PQ [1].data;
       PQ [1].color, PQ [1].data, PQ [N].data, PQ [N].color, N
       ≡ "yellow", PQ [N].data, null, "blue", N-1;
       @UnLock (LAST, PQ [1].lock; PQ [N+1].lock);
       EnQueue (WQ, (1, "yellow"));
   end;
   print ("the item dequeued from the heap is", item);
  end;
  else begin
     print ("there are no items in the heap");
     end;
end;

procedure Insert (item)
begin
(* put item in first available null node and color it red *)
    @Lock (LAST, PQ [N+1].lock);
    begin
       N, PQ [N + 1].data, PQ [N + 1].color
                      ≡ N + 1, item, "red";
    end;
    @UnLock (LAST, PQ [N].lock);
    EnQueue (WQ, (N, "red"));
    print ( item, "has been enqueued in the heap");
end;
```

```
procedure filter (item)   (* check the trivial cases *)
begin
 done := false;
 item := undefined;
 repeat begin
      @Lock (LAST); (* check if there are no items *)
      if (N < 1) begin
             item := null;
             @UnLock (LAST);
             done := true;
           end;
      elseif (N = 1) begin (* check if exactly one item *)
             @Lock (PQ [1].lock);
             item := PQ [1].data;
             N := 0;
             @UnLock (LAST, PQ [1].lock);
             done := true;
           end;
      else     begin  (* two or more items *)
             @Lock (PQ [1].lock, PQ [N].lock);
             if (PQ [1].color <> "yellow")
               begin done := true; end;
             else begin  (* try once more *)
                 @UnLock (LAST, PQ [1].lock, PQ [N].lock);
             end;
           end;
  end;
  until (done);
end;
```

```
procedure Restructure
begin
  *repeat forever*
    begin DeQueue (WQ, (a, b));
    if (b = "red") begin
      p := parent (a);
      @Lock (p, a);
      PushUp (a);
      @UnLock (p, a);
     end;
    else begin (* b is "yellow" *)
      @Lock (a, lchild(a), rchild(a));
      PushDown (a);
      @UnLock (a, lchild(a), rchild(a));
    end;
   end;
end;

procedure PushUp (c);
begin
  p := parent(c);
  if (p = 0) PQ [c].color := blue; exit; (* must be the root *)
  not (red(c)) ⟶ begin (* exit this procedure *) end;
  blue(p) Λ (p ≥ c)  ⟶ begin PQ[c].color := blue; end;
  blue(p) Λ (p < c)  ⟶ begin Swap (p,c);
                              EnQueue (WQ, (p, "red")); end;
  red(p) Λ (p ≥ c)   ⟶ begin EnQueue (WQ, (c, "red")); end;
  red(p) Λ (p < c)   ⟶ begin Swap (p,c);
                              EnQueue (WQ, (c, "red")); end;
  yellow(p) Λ (p ≥ c) ⟶ begin PQ[c].color := blue; end;
  yellow(p) Λ (p < c) ⟶ begin EnQueue (WQ, (c, "red")); end;
end;
```

```
procedure PushDown (p);
begin
  Let Larger and Smaller be the indices
  of the larger and smaller children of p
  $\neg$ yellow(p) $\rightarrow$ begin exit this procedure end;
  $\neg$ yellow (Smaller) $\Lambda$ red(Larger) $\Lambda$ (p $\geq$ Larger)   $\rightarrow$ begin PQ[Larger].color := blue;
                          PQ[p].color := blue; end;
  $\neg$ yellow (Smaller) $\Lambda$ red(Larger) $\Lambda$ (p < Larger)   $\rightarrow$ begin Swap (p,Larger);
                          EnQueue (WQ, (Larger, "yellow"));
                          EnQueue (WQ, (p, "red")); end;
  $\neg$ yellow (Smaller) $\Lambda$ yellow(Larger) $\Lambda$ (p $\geq$ Larger) $\rightarrow$ begin
                          EnQueue (WQ, (p, "yellow")); end;
  $\neg$ yellow (Smaller) $\Lambda$ yellow(Larger) $\Lambda$ (p < Larger) $\rightarrow$ begin Swap (p, Larger);
                          EnQueue (WQ, (p, "yellow")); end;
  $\neg$ yellow (Smaller) $\Lambda$ blue(Larger) $\Lambda$ (p $\geq$ Larger)   $\rightarrow$ begin PQ[p].color := blue;
                          PQ [Smaller].color := "blue"; end;
  $\neg$ yellow (Smaller) $\Lambda$ blue(Larger) $\Lambda$ (p < Larger)   $\rightarrow$ begin Swap (p,Larger);
                          EnQueue (WQ, (Larger, "yellow"));
                          PQ [Smaller].color := "blue"; end;
  yellow (Smaller) $\Lambda$ $\neg$ yellow(Larger) $\Lambda$ (p $\geq$ Larger) $\rightarrow$ begin
                          EnQueue (WQ, (p, "yellow")); end;
  yellow (Smaller) $\Lambda$ $\neg$ yellow(Larger) $\Lambda$ (p < Larger) $\rightarrow$ begin Swap (p,Smaller);
                          EnQueue (WQ, (p, "yellow")); end;
  yellow (Smaller) $\Lambda$  yellow(Larger) $\Lambda$ (p $\geq$ Larger) $\rightarrow$ begin
                          EnQueue (WQ, (p, "yellow")); end;
  yellow (Smaller) $\Lambda$  yellow(Larger) $\Lambda$ (p < Larger) $\rightarrow$ begin Swap (p,Smaller);
                          EnQueue (WQ, (p, "yellow")); end;

end;
```

To take care of the trivial cases, Delete calls a filter procedure. When control returns to Delete, if *item* is still undefined, i.e. not *null* and not a legitimate value, then there must have been at least two items in the heap. In this case, both the 1^{st}

and N^{th} elements of the heap are locked and returned if and only if the 1^{st} node is non-yellow. (If the 1^{st} element is "red" then it can be considered "blue").

3.4 Definitions for correctness

A *concurrent priority system* consists of a priority queue, a work queue, and a finite number of activities that have access to both, and simultaneously carry out Insertion, Deletion and Restructuring.

A concurrent priority system is *correct* if it satisfies the following properties:

PHC (Parallel Heap Consistency): The heap property is preserved at all times at all nodes except those at which an insert / delete restructuring operation is currently in progress. For any such exception x, the heap property holds between all ancestors and all descendants of x that are not exceptions.

PDC (Parallel Delete Consistency): A deletion operation always returns the highest value in the priority queue, from among those nodes where an insert / delete restructuring operation is not currently in progress.

To these two we add the following two requirements, that only apply to parallel systems.

DA (Deadlock Avoidance): The system is free from deadlock. This directly follows from Lemma 4.1, and the fact that there is an implicit ordering (i.e. the variable N followed by the elements of the array PQ ordered from 1 to N), among the set of globally shared variables (objects). Subsets of these variables called 'windows' are accessed strictly in this order.

SA (Starvation Avoidance): The system is free from starvation, if we assume a fair scheduling policy.

If a deleting activity is never able to return an item from the heap in spite of the heap being non-empty, then we say that the deleting activity starves. Our delete algorithm is not strictly starvation free. For instance, one can imagine a system in which a given deleter is always beaten to the root node by another deleter, and thus persistently gets to see a yellow value at the root node. The fair scheduling policy assumption states that a deleting activity will eventually be able to access a blue node at the root given that structure remains non-empty.

EP (Eventual Progress): The system never enters a state in which values in those nodes of the heap where an insert / delete restructuring operation is currently in progress; are permanently unable to make such progress. Every state the system enters, is one from which progress is possible.

We make the following assumptions.

A1: PHC and PDC are initially true. This axiom is easily established by suitably initializing the data structure and associated variables.

A2: The work queue (WQ) access and update algorithms are correct.

We define correctness as the preservation of the above five properties. Only two of these properties need to be proved. PDC is a direct consequence of PHC. DA directly follows from Lemma 4.1 and the ordering of resources (program variables) followed by all our parallel algorithms. SA is obvious by its construction and could have been stated as an axiom. In what follows, we rigorously state PHC. The detailed proof of PHC and EP are given in Appendix 5.

$$PHC \equiv \forall i, j : 1 \le i < j \le N : \quad ancestor(i,j) \wedge blue(i) \wedge blue(j) \rightarrow PQ[i] \ge PQ[j]$$

where $ancestor(i,j) \equiv \exists\, n : 1 \le n \le \log_2 N : j \; div \; 2^n = i$

is preserved by the set of activities on the heap. We shall do this by first proving a stronger assertion

$$PHC' \equiv \forall i, j : 1 \le i < j \le N : ancestor(i,j) \wedge \neg yellow(i) \wedge \neg red(j) \rightarrow PQ[i] \ge PQ[j]$$

is preserved by the set of activities. In other words, all red and blue nodes are ordered above their yellow and blue descendants, where "above" and "below" mean

$\geq$ and $\leq$ respectively. The detailed proofs are in Appendix 5.

3.5 Object implementation state for the concurrent heap

Given a state s of the CHEAP, we outline various functions that describe the *ois* of s. Let D denote the domain of possible values in a heap node (for example D could be the set of pairs consisting of a priority value and a capability to a system object).

$Dec\ (s) = \{c_1, c_2, \cdots, c_N\}$

$Cval\ (<c_1, c_2, \cdots, c_N>) = <r_1, r_2, \cdots, r_N>$
such that $r_i \, \varepsilon \, \mathrm{D} : 1 \leq \mathrm{i} \leq \mathrm{N}$.

$Dec^{-1}\ (< Cval\ (c_1), Cval\ (c_2), \cdots, Cval\ (c_N)>) = < r_1, r_2, \cdots, r_N > = s$

$Crel$ = { Parent(i) = i / 2 : 2 $\leq$ i $\leq$ N,
Left_child(i) = 2 * i : 1 $\leq$ i $\leq$ N / 2,
Right_child(i) = 2 * i + 1 : 1 $\leq$ i $\leq$ N / 2 }

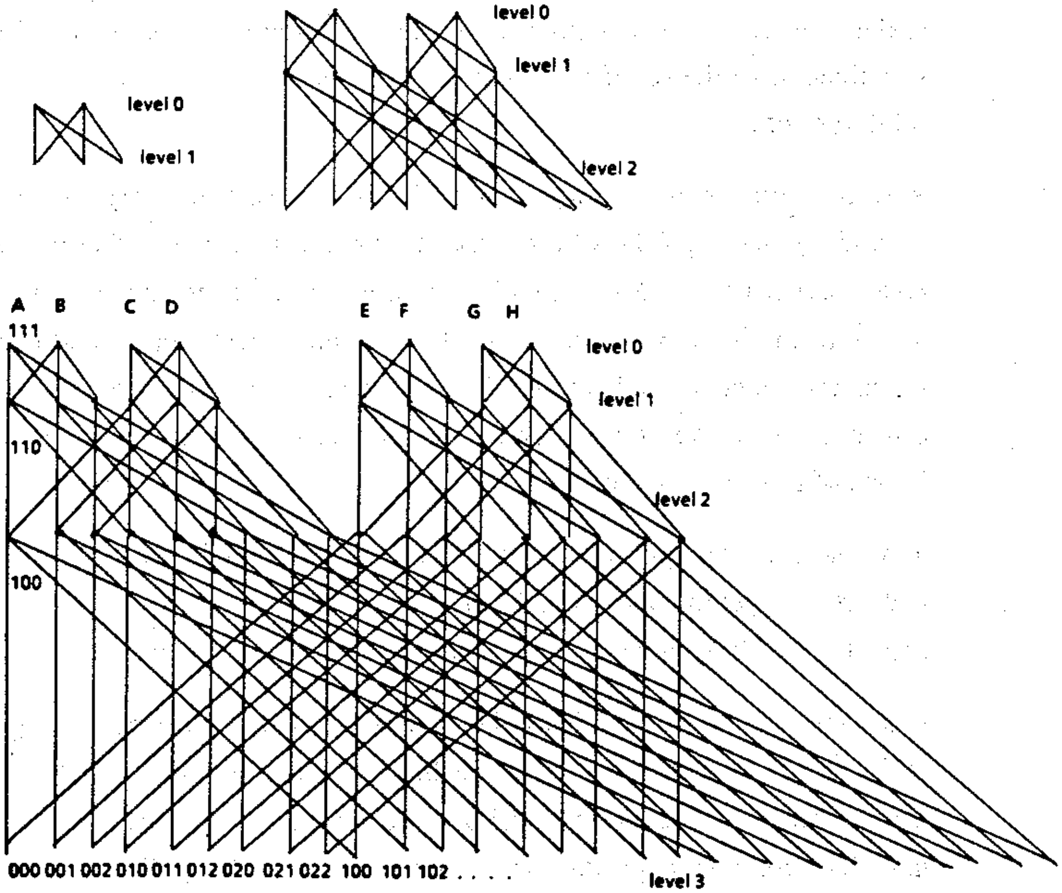
$Window\ (A$, Insert) = { c_N }
$Window\ (A$, Delete) = { c_1, c_N }
$Window\ (R$, PushUp, i) = { $c_i, c_{Parent(i)}$ }
$Window\ (R$, PushDown, i) = { $c_i, c_{Left\_child(i)}, c_{Right\_child(i)}$ }
where $R, A \ \varepsilon \, AC$ (R is a restructuring activity and A is a user activity.)

3.5.1 Incorporating non-null percentiles

The algorithms presented above for the concurrent heap assume that the percentile parameter is null. These algorithms may be easily modified for non-null percentiles as follows. A count of red values is maintained. This count is atomically incremented with every insert and decremented every time a red value changes color. Thus at any given time the total number of red values inside the structure is known. This gives us a means for obtaining the maximum possible percentile level a deletable value in the root node, such that even if all the red nodes were larger than this value, it would still be in the top p percentile.

4. The Banyan as a simultaneously updatable data structure.

The heap algorithms can be extended to more general directed acyclic graphs (*DAGs*). A *software banyan* (SBAN) is a regular multilayered DAG structure [Goke '73], excellently adaptable to a concurrent priority structure. More interestingly, such a structure affords removal (deletion) at a spectrum of percentile levels at an associated spectrum of costs. Figs 4.3.1 - 4.3.3 show a number of regular SW banyans (see Appendix 6 for definitions). Henceforth we shall use the term *banyan* to refer to software banyans.



Figs 4.3.1 - 4.3.3 Scaling between different sizes of a banyan with spread (s) = 2, fanout (f) = 3

4.1. Algorithms for the Software Banyan (SBAN) and their correctness proofs

Insertion into a banyan is achieved identically to insertion in the concurrent heap. *Deletion* too is done in an identical fashion, except for the fact that more than one apex nodes may need to be examined before the final dequeue operation takes place†. The restructuring algorithm for the SBAN is different from the corresponding algorithm for the CHEAP, because of the presence of several parent nodes for each child. The algorithm makes use of the notion of a restructuring window.

Given a restructuring workpiece (i,c) with color c at index i, if it is an insert (delete) restructuring workpiece, then the *restructuring window* for the workpiece is defined to be the set of nodes consisting of the siblings of i along with all parents (children) of i.

A restructuring window may be regarded as two sets of array elements or nodes, S_1 and S_2 (see Fig 4.1), whose structural properties are such that every node in S_1 *covers* every node in S_2. Also, $|S_1| = s$, the *spread* of the banyan, and $|S_2| = f$, the *fanout* of the banyan. As with the CHEAP, the critical idea behind restructuring is the maintenance of the invariant PHC' presented earlier. Once a restructuring window has been locked for prevention of interference, the state of the window may be in one of three possible classes:

a) S_1 has a yellow node and S_2 has a yellow node (Fig 4.2a).
b) S_1 has no yellow node (Fig 4.2b).
c) S_1 has a yellow node and S_2 has no yellow node (Fig 4.2c).

If the state of the window is a), the only swaps possible are between yellow values in S_1 and S_2. Any other swap must be prohibited because of lack of information about values covered by yellow nodes in S_2. Once in state a) the *iterative_minimax* algorithm is invoked to order the yellow values. A simple minded sort algorithm is incorrect, since yellow nodes must preserve consistency

† In an earlier version [Biswas '87b] of this algorithm, we had reported the use of null values for the concurrent insertion and deletion of values from the software banyan. In the current version we have eliminated null values, and present a more elegant algorithm.

with respect to non-yellow ancestors. We now present the algorithm itself and its proof of correctness.

```
[1] repeat forever
  dequeue a restructuring workpiece (i,c)
  if color of i has changed, go to [1]
  if c is RED then begin
     $S_1$ := parents (i)
     $S_2$ := siblings (i)
  else
     $S_1$ := spouses (i)
     $S_2$ := children (i)
  lock all nodes of $S_1 \cup S_2$
    call Mutate_Window (i, $S_1$, $S_2$)
  unlock all nodes of $S_1 \cup S_2$
  go to [1]
```

Algorithm 4.1

As it must be obvious, restructurers do a lot of work for each other. It would be pointless to attempt to do a restructuring which has already been done. Therefore, as a runtime optimization, we first check to see if the color of a node has changed from what it was supposed to be according to the restructuring workpiece retrieved from the work queue. If the color has changed, we simply ignore the workpiece and go back for another workpiece. This examination can be done in read-only mode, i.e. without locking any of the nodes and thus does not interfere with concurrent locking or writing. This optimization has been incorporated into the SBAN restructuring algorithm shown in Algorithm 4.1.

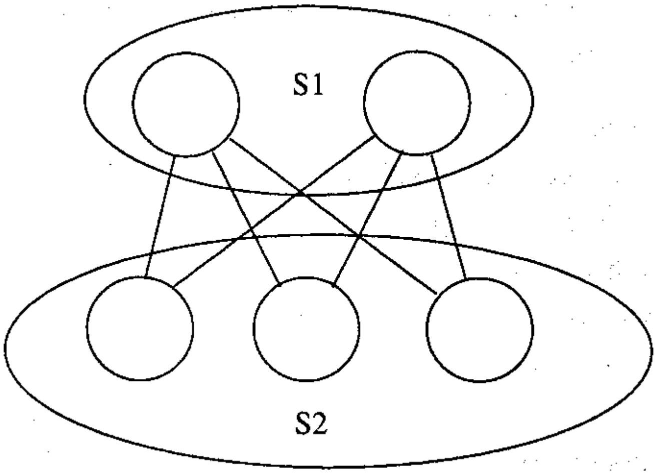


Fig 4.1

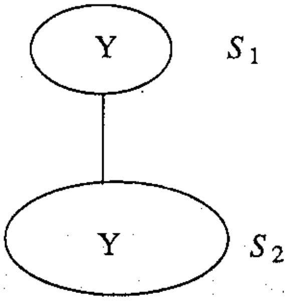


Fig 4.2a

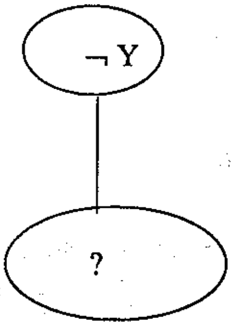


Fig 4.2b

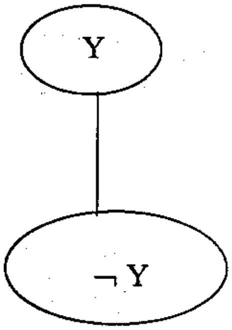


Fig 4.2c

4.1.1. Algorithm *Mutate_Window*

This algorithm is invoked with i, the index of a restructuring workpiece, and S_1 and S_2, the sets of parent and child nodes respectively, that together constitute the restructuring window for i. It maintains a list called TouchList, of nodes that have been touched during reorganization, so that these nodes may later be put on the Work Queue.

```
[1] Add i to TouchList. Initialize booleans $b_1$ and $b_2$ to false.
[2] Set $b_1$ to true if $S_1$ has a yellow node.
    Set $b_2$ to true if $S_2$ has a yellow node.
```

```
 if ( $b_1$ and $\neg b_2$ ) then go to [3]  (* window is in state c *)
else if ( $b_1$ and $b_2$ ) then go to [5]  (* window is in state a *)
else go to [4]  (* window is in state  b *)

[3] Algorithm PushDown
   Let $u$ be the smallest yellow node in $S_1$.
   Let $v$ be the largest (obviously non-yellow) node in $S_2$.
   if ( $u < v$ )
       change all yellow nodes in $S_1$ to blue
       change all red nodes in $S_2$ to blue
   else
       swap $u$ with $v$ and add both $u$ and $v$ to TouchList
   go to [6]

[4] Algorithm Pushup
For each red node $r$ in $S_2$,
     Attempt to push up $r$ by swapping it with its minimum (obviously non-yellow) parent.
     If swap was successful, add swapped nodes to TouchList.
If there are no red nodes in $S_1$ then change all red nodes in $S_2$ to blue.
go to [6].

[5] Algorithm Iterative_Minimax
   Let $u$ be smallest yellow node in $S_1$
   Let $v$ be largest yellow node in $S_2$
   if ( $u < v$ ) then
     swap $u$ and $v$
     add swapped nodes to TouchList
     go to [5]

[6] For each node $u$ in TouchList, if $u$ is not blue, then enqueue it along with its color in the WorkQueue.
```

4.1.2. Proof of correctness

Here we merely give a brief illustration of how the proof of correctness of the SBAN algorithms would proceed. The same proof techniques that have been developed and applied to the CHEAP algorithms in Appendix 5 could be adapted to banyans in the following way. We assume that the state of the SBAN is consistent before invocation of the *Mutate_Window* algorithm. Let us consider in turn, the three cases a, b and c shown in Figs 4.2.

Case a: Algorithm *Iterative_minimax* swaps yellow nodes in S_1 with yellow nodes in S_2. Thus we need not be concerned with descendants of S_1 and / or S_2, since PHC' is trivially satisfied with respect to these nodes. Let u be a node in S_1 that has undergone a swap with v, a node in S_2. Ancestor nodes of u are by definition, ancestor nodes of v. Thus, for any non-yellow ancestor c of u, the swap is between two values, each of which is smaller than c. Ancestors of v see a smaller yellow value after the swap. Thus the swap leaves the invariant true at all ancestors of both nodes involved.

Case b: Algorithm *Pushup* attempts to push up a red value, say v, from S_2 by swapping them with a non-yellow value, say u, from S_1. u can be either red or blue, (by our choice of cases a, b and c). In each case, it is easily seen how PHC' is preserved through the swap at ancestor nodes of u and descendant nodes of v. Thus the swap preserves the invariant.

Case c: Algorithm *Pushdown* attempts to swap yellow nodes from S_1 and S_2. We need only be concerned about ancestor nodes since nothing can be asserted about descendant nodes of a yellow node. Say u from S_1 is to be swapped with v from S_2. All non-yellow ancestors of v see a smaller value at v after the swap, and all non-yellow ancestors of u see a value at u that was already known to ordered lower than themselves. Thus the swap preserves PHC'.

4.2. Percentile levels

When a banyan gets full, it must grow, and conversely when there are too few elements in it, it must shrink. For the period of time a banyan is being scaled no access to it must be permitted. We now investigate the structural properties of a banyan that make it attractive for priority applications.

A *full* banyan is a banyan that has a non-null value in each node, and in addition satisfies the heap property at each node. The key advantage of a software banyan over a heap is that simultaneous deletes are possible, at different percentile levels. Given a full banyan, if a request for delete investigates a fraction of the *apex* nodes, it is guaranteed a corresponding percentile level of the item returned. If it investigates all of the apexes, it is guaranteed to be returned the highest priority value, (100 percentile) at the cost of having to place a lock on all apexes.

An item A removed (deleted) from a banyan is at *percentile* p if at least p % of the items in the structure are ordered less than or equal to A at the time of the removal, ignoring insertions in progress. (In other words, treating items in progress as if they have not yet entered the structure). Definitions for spread (s) fanout (f) and depth (L) are given in Appendix 6.

Lemma 4.1.: Given a full banyan, if p apex nodes are investigated by a deleting operation returning the value A, that is the maximum among these nodes, then, the *expected percentile*, of A, written $ep(A)$, is given by:

$$ep(A) = [\gamma\ (p, s, f, L)\ / \sum_{i=0}^{L} s^{L-i}\ f^{i}] * 100 \qquad (1)$$

where $\gamma\ (p, s, f, L) =$

$$\begin{cases} \sum_{i=0}^{L} f^{i} & : p = 1 \\ \sum_{i=0}^{L} f^{i} + \sum_{i=1}^{q-1} (s^{i} - s^{i-1}) \sum_{k=0}^{L-i} f^{k} + (p - s^{q-1}) \sum_{k=0}^{L-q} f^{k} & \\ \qquad : p = s^{q-1} + 1, \cdots, s^{q} \ :\ 1 \le q \le L\ : p \ne 1 & \end{cases}$$

Proof:

If node B is reachable from node A, by traversing down parent - child links in a banyan, we say A *covers* B. The total number of nodes in a (s, f, L) SW banyan is the sum of the number of nodes at each level. This is the denominator of eqn (1). This can be rewritten as eqn (2), where *sizeof* (s, f, L) denotes the number of nodes in an (s, f, L) banyan.

$$sizeof\ (s, f, L) = \sum_{i=0}^{L} f^{i} + \sum_{i=1}^{L} \left[(s^{i} - s^{i-1}) \sum_{k=0}^{L-i} f^{k} \right] \qquad (2)$$

The first term, the number of nodes in an f *–ary* tree of depth L corresponds to the number of items covered when we examine (after locking) the first apex node. Of the remaining nodes, exactly $s^{1} - s^{1-1}$, i.e. $s-1$ apex nodes may be found that cover independent f *–ary* trees of depth $L-1$, none of which is covered by the first apex; similarly , for $2 \le i \le L$, $s^{i} - s^{i-1}$ such trees may be found, each of depth $L-i$. The only way to cover all the nodes is to investigate all the apex nodes. $\gamma\ (p, s, f, L)$ is a restatement of equation (2), varying the parameter p between successive values in the range s^{q-1} through s^{q}, $1 \le q \le L$. □

Table 4.1 shows percentile values versus p, the number of nodes locked for (2,3,5) and (3,4,5) banyans. From this table we see that by locking only a small fraction of the apex nodes, an item with a high percentile level can be retrieved from the priority structure. The percentile values naturally depend on the salient parameters of the banyan.

(2,3,5) banyan (665 nodes)		(3,4,5) banyan (3367 nodes)	
p	percentile	p	percentile
1	54.7	1	40.5
2	72.9	3	60.8
4	85.0	9	75.9
8	92.8	27	87.2
16	97.6	81	95.2
32	100.0	243	100.0

Table 4.1. Percentiles for various values of p (the number of apexes examined).

4.2.1 Weakened specifications for the software banyan

For the strict priority queue if a delete returns value p then p is the highest value in the structure. For the concurrent heap the weakened specification is that if a delete returns p then p is the highest *blue* value in the structure. The software banyan has still weaker specifications than a concurrent heap, but at an internal level. That is to say, these specifications must be explicitly stated in terms of the object implementation states of the two implementations. The weakened specification is as follows: "if a delete returns p then p is the largest blue value in the set of trees rooted at the set of apex nodes examined by the delete operation". More formally, assuming that identical delete operations are carried out (with identical windows) on the CHEAP and SBAN structures, and assuming that the percentile specification is the wildcard (null) in both cases, then the guaranteed percentile level for the CHEAP will be higher than the guaranteed percentile level for the SBAN.

4.2.2 Coverage

In order to cover the maximum number of nodes with the fewest possible locks, a locking schedule must be enforced. To get maximum coverage from apex nodes, a deleting operation must lock items in a particular sequence known as a *locking schedule*. For a given (s, f, L) banyan, the locking schedules are fixed. For example, in the (2,3,3) banyan of Fig 4.3.3, a locking schedule starting at apex A is any member of the class of schedules generated by the schema A-E-{CG}-{BDFH}, where parenthesis are used to indicate that the nodes inside may be accessed in any sequence, but must be all accessed before moving ahead in the sequence. Starting at other nodes it is possible to arrive at similar schedule schemas. Different entering operations may follow different locking schedules, (modulo deadlock). Note that the complete locking schedule need not be enforced if smaller percentiles (than 100) are desired.

4.3. Mapping

4.3.1. Mapping onto linear address spaces

When mapping the banyan we use linear memory address for successive nodes at the same level. This is not necessarily optimal, but it obviates the necessity for storing pointers explicitly and makes for an easier implementation. Every vertex in an (s, f, L) regular SW banyan, $(s \leq f)$, may be labelled by an L digit number in base f, such that vertices in the same level have distinct labels, and there is an arc from a vertex in level *i, (0* $\leq$ *i* $\leq$ *L),* to a vertex in level $i+1$ if their labels differ at most in the i^{th} digit [Premkumar '81]. In Fig 4.3.3 we show a possible labelling of the nodes of our (2,3,3) banyan.

Memory can be allocated in such a way that given a level and a label, we may arrive at a unique address for any node in an (s, f, L) banyan. This raises two points:

i) Wasted memory. If the entire memory for the banyan is allocated as a single block, for an arbitrary (s, f, L) banyan the memory wasted could be arbitrarily large. If pointers are used roughly 50% of the memory is "wasted". However, for small s and f (between 2 and 4) both schemes, i.e. with and without pointers have approximately the same amount of wasted memory.

ii) Parent / Children addresses. In calculating the address of a parent given that of a child and vice versa, the fundamental operation needed is conversion between binary and *base f* numbers. Given a binary number, to convert it to its *base f* representation and vice versa takes $O(k)$ multiplications and additions, where k is the length of the label in the original representation. This price must be paid once for each restructuring operation. The operations needed are the computation of the addresses of all children given that of a parent, and conversely the addresses of all parents, given that of a child. In both cases, if the address of the leftmost parent (child) is calculated, by zeroing the most significant digit, the addresses of the remaining parents (children) can be calculated by repeatedly adding a fixed offset (which is different for each level). The alternative is to store the labels explicitly in each node, which is similar to building the entire structure as a linked structure with pointers.

For a banyan to be useful as a priority structure it must be conical in shape. Rectangular banyans are ruled out since they have too many apex nodes. For small fanouts banyans using linear memory implementation are not too space inefficient. However as the fanout increases the relative amount of unused memory also increases dramatically. Therefore for fanouts greater than 3, we recommend the use of a linked list mapping. The linked mapping proceeds roughly as follows. Each node contains three pointers, a pointer to its first parent, a pointer to its next sibling and a pointer to its first child. These pointers can be set up at the time of scaling or initialization. Following these pointers it is fairly straightforward to construct the sets of parents, spouses and children discussed in the Banyan algorithms.

4.3.2. Hypercube mappings

Premkumar [Premkumar '81] has demonstrated how to map a rectangular (d, L) SW banyan (definitions are in Appendix 6) onto a (d, L) hypercube by homomorphically† reducing the banyan to the hypercube. He used alternating sequences of labelled nodes to construct partitions of the cube, which correspond to data trees in the original banyan.

The partitioning problem as defined by Premkumar is of secondary concern to us. Our first problem is to find an efficient mapping of a regular (s, f, L) SW banyan onto a binary hypercube. Our next problem is to find an efficient algorithm to realize this mapping. Binary hypercubes are selected because of their popularity and current abundance.

4.3.2.1. The mapping and its optimality

We shall refer to a rectangular (d, L) SW banyan simply as a (d, L) banyan, and a regular (s, f, L) SW banyan simply as a (s, f, L) banyan. Our end goal is to map an arbitrary (s, f, L) banyan onto a binary hypercube. In the special case of a (d, L) banyan where d is a power of 2, we demonstrate that an alternating sequence labelling yields an optimal mapping. In the more general case where $d = max(s, f)$ is a power of 2, we use the optimal mapping as a basis for mapping

† A homomorphism is a one to one mapping. An isomorphism is a one to one as well as onto mapping.

of the (s, f, L) banyan. The mapping is done in the following steps:

[1] Let $d = \max(s, f)$ such that d is a power of 2. Homomorphically reduce a (d, L) banyan to a (d, L) hypercube.

[2] Let $d = 2^k$. Find an optimal mapping from a (d, L) hypercube to a $(2, kL)$, i.e. binary, hypercube. The optimality criterion is the minimum cumulative edge dilation criterion (see definition below).

[3] Remove the unwanted nodes and edges in the (d, L) banyan to obtain the desired (s, f, L) banyan.

Step 1 maps an edge (i.e. two adjacent nodes) of a (d, L) banyan onto an edge (i.e. two adjacent nodes) of a (d, L) hypercube. Step 1 therefore satisfies the optimality criterion. For rectangular banyans, Step 3 does not add to the cumulative edge dilation. Thus if Step 2 is optimal, the entire procedure is optimal. For nonrectangular banyans however, we are not guaranteed optimality. In other words, depending on the values of s and f, it may be possible to find a banyan that has a smaller cumulative edge dilation than that obtained by going through steps 1 - 3.

If $G = (V, E)$ is a graph and $e = (v_1, v_2) \varepsilon E$, then v_1 is called the *source* of e and v_2 is called the *destination* of e.

Definition: Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs such that $|V_1| = |V_2|$ and $|E_1| \geq |E_2|$. Let $\tau: V_1 \rightarrow V_2$ be a one to one and onto mapping. We say τ is *edge preserving* from G_1 to G_2, if for every edge $e = (v_1, v_2)$ in E_1, there is a sequence of edges $S = \{e_1, e_2, \cdots, e_k\}$ in G_2, such that the source of e_1 is $\tau(v_1)$ and the destination of e_k is $\tau(v_2)$ and the source of e_i corresponds with the destination of e_{i-1} for $2 \leq i \leq k$. S is called a *path* from $\tau(v_1)$ to $\tau(v_2)$ and k is the *length* of the path S.

If S is the shortest path from $\tau(v_1)$ to $\tau(v_2)$ then the *edge dilation* of e is $k-2$. Thus the edge dilation is simply the number of edges by which an edge has been stretched after having gone through the mapping τ. If τ maps an edge in G_1 onto an edge in G_2 then the edge dilation of e under τ is 0.

Theorem 4.1. : Given a (d, L) hypercube G_1, where $d = 2^k, k \geq 1$, with a (d, L) alternating sequence labelling S_1 of its nodes, and a $(2, kL)$ hypercube G_2, with a $(2, kL)$ alternating sequence labelling S_2 of its nodes, the mapping τ that maps the node from G_1 corresponding to the i^{th} label in S_1 to the node in G_2 corresponding to the i^{th} label in S_2, is an optimal mapping.

Proof : To prove this theorem we shall prove the following lemmas.

Lemma 4.2. : There are $d^{(L-1)} * L$ d-cliques in a (d, L) hypercube.

A d-clique is a complete graph on d nodes. We shall refer to a d-clique simply as a clique. This lemma is proved by induction on L. It is obviously true for L=1. Assume it is true for arbitrary L. When the hypercube is grown in size by one, from L to L+1, we have $d^{(L-1)} * L$ cliques from each of d components of the smaller size. In addition, each node in a smaller cube belongs to exactly one additional clique in the larger (composite) cube. There are d^L nodes in a smaller cube (see Fig 4.4). Thus the total number of cliques in the composite cube is $d^{(L-1)*}L + d^L = d^L * (L+1)$, which establishes the claim.

To prove that the cliques must be edge-disjoint, we simply observe that the total number of edges in a (d, L) hypercube is $(d^L * L * (d-1)) / 2$, i.e. half the sum of the degrees of the nodes, which is precisely the same as the edges in $d^{(L-1)} * L$ cliques in a (d, L) hypercube. □

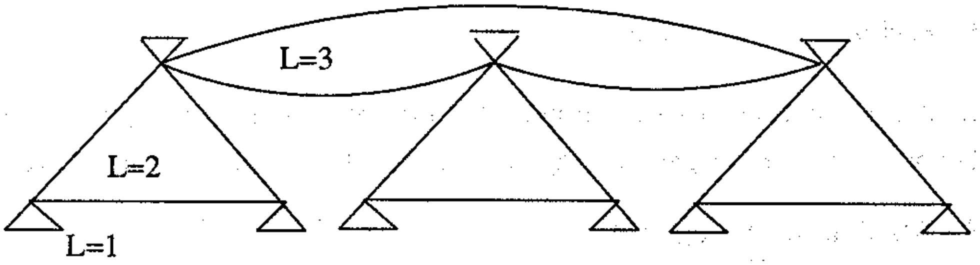


Fig 4.4 Cubes and cliques

Lemma 4.3. : A $(d, 1)$ hypercube (i.e. a d-clique) with nodes labelled $0, \cdots, d-1$ maps optimally onto a $(2, k)$ hypercube if the i^{th} node in the $(d, 1)$ hypercube is mapped onto the node with the i^{th} label in the binary alternating sequence labelling of the $(2, k)$ hypercube.

Proof: The proof is trivial. Since all possible edges must be realized by the $(2, k)$ hypercube, all labellings must be optimal. In particular, the labelling $0, \cdots, d-1$ must also be optimal. This is in a sense the base case of the above theorem. □

Lemma 4.4. : Every d-clique in G_1 maps onto a binary subcube of size k in G_2.

Proof: Since d is a power of 2, a d*-ary* label in S_1 can be converted into a corresponding (under the mapping τ) binary label in S_2 by simply converting the former label digit by digit into their binary labels, and concatenating the binary labels thus obtained.

Consider a set of d nodes that form a clique in G_1. Obviously they must all differ in the same digit position. Let this be the n^{th} digit position, $1 \le n \le L$. Thus these nodes vary in only k consecutive binary bit positions in S_2. Thus they form a subcube of size k in G_2. □

We have shown that τ maps every d-clique in G_1 optimally onto a corresponding k subcube in G_2. Thus, every edge is optimally 'stretched', in the sense that there is no other mapping that can produce a smaller cumulative edge dilation for the entire structure. Thus τ maps the entire structure optimally. This completes the proof of the theorem. □

5. Performance estimates

In this section we derive formulae for the maximum throughput and an upper bound on mean propagation delay in concurrent heaps and software banyans.

5.1 Maximum Throughput

By *maximum throughput* we mean the maximum degree of simultaneity or the maximum number of simultaneous updates that a data structure affords. We shall draw an analogy from a simpler example, namely that of an array of N elements each of which can be locked. Assuming that this array is in shared memory and that each operation that mutates it requires only a single element, the array can sustain a maximum of N simultaneous updates. If instead, each operation carries out an atomic change on *two* array elements, the number of simultaneous updates that the array affords is now halved to N/2.

Assume now, that there are two types of operations, type A requiring 2 elements and type B requiring 3 elements for an atomic update. Let the fraction of operations of type A be f_A and that of type B be f_B, such that $f_A + f_B = 1$. Then the maximum number of simultaneous updates afforded by the array is $\frac{N}{2f_A + 3f_B}$.

To apply the above analogy to our priority structures, we observe that insert restructuring is a type A operation and delete restructuring is a type B operation. Thus for a complete heap of depth L (i.e. with 2^L - 1 nodes) the *maximum restructuring operation throughput*, $T_{max\_rest\_op}$, that can be sustained by a heap of depth L is given by:

$$T_{max\_rest\_op} = \frac{2^L - 1}{2f_{ir} + 3f_{dr}}$$

where f_{ir} and f_{dr} denote fractions of insert and delete restructuring operations respectively. (Note $f_{ir} + f_{dr} = 1$).

Now we shall extend this formula from restructuring operations to regular operations of the concurrent heap, ie. inserts and deletes. Let r_i and r_d denote respectively the rates of insertion, and deletion operations upon the heap. Let n_i and n_d denote respectively the average number of levels propagated by a red (inserted) and yellow (deleted) value respectively. Then:

$f_i = \frac{r_i}{r_i + r_d}$ is the fraction of insert operations. Similarly, $f_d = \frac{r_d}{r_i + r_d}$ is the fraction of delete operations.

The *maximum operation throughput*, T_{max_op}, afforded by a concurrent heap of depth L is:

$$T_{max\_op} = \frac{2^L - 1}{(f_i\, n_i + f_d\, n_d)\ (2f_i + 3f_d\,)}$$

The first term in the denominator is the mean number of levels propagated by a value, to complete an operation and all the restructuring operations generated by it. The second term is the mean number of nodes locked for an operation.

Please note:

i) The above formula makes use of the assumption that $f_i = f_{ir}$ and $f_d = f_{dr}$. In other words, we assume that the fractions of insert and delete operations is equal to the fractions of insert and delete restructuring operations they generate.

ii) For stability reasons, we also assume that L is large, so that there are no random fluctuations caused by changes in the depth of the heap.

5.2 Upper bound on mean propagation delay

The mean propagation delay is the mean time for a red (yellow) value to turn blue in the insert (delete) algorithms.

Let $\lambda = r_i + r_d$. Let R_i (R_d) denote respectively the probability that a restructuring is required because of an inserted (deleted) item. Then, $R_i = \frac{n_i - 1}{n_i}$ and $R_d = \frac{n_d - 1}{n_d}$.

Thus the probability that a serviced request will be reinserted in the work queue, is:

$$p = f_i\, R_i + f_d\, R_d = \frac{r_i\, R_i}{\lambda} + \frac{r_d\, R_d}{\lambda}$$

In the worst case, $n_i = n_d = L$, i.e. all inserts and deletes propagate L levels. In this case, the worst case probability of reinsertion ($p_{\max}$) is given by:

$$p_{\max} = \frac{(r_i + r_d\,)}{\lambda} R_d = R_d = \frac{L - 1}{L}$$

Therefore, $\lambda_{\max}$ = maximum rate of restructuring requests is given by:

$$\lambda_{\max} = \frac{\lambda}{1 - p_{\max}} = L\, \lambda$$

Assuming that the work queue is served by an **M/M/c** queueing system, which is valid as long as we assume collision freedom. Let W_{ub} denote the upper bound on the mean propagation delay through the work queue. This is the mean time that a red / yellow restructuring request spends in the work queue, assuming maximum rate of restructuring requests.

$$W_{ub} = \frac{1}{\mu} + \frac{\frac{(\frac{L\,\lambda}{\mu})^c}{c\,!}}{\frac{(\frac{L\,\lambda}{\mu})^c}{c\,!} + (1-\rho)\sum_{n=0}^{c-1}\frac{(\frac{L\,\lambda}{\mu})^n}{n\,!}} * \frac{\frac{1}{\mu}}{c\,(1-\rho)}$$

where

μ is the individual service rate of each restructuring activity,

c is the number of restructuring activities,

ρ is the server utilization, i.e. $\frac{L\,\lambda}{c\,\mu}$

The first term $\frac{1}{\mu}$ above stands for the expected service time for a single server. The second term is the mean queueing time, and makes use of Erlang's c formula, the probability that all c servers are busy, so that an arriving request has to wait. The M/M/c formula gives the basis for the calculating T_{ub}, the upper bound of the mean time for a red or a yellow value to turn blue, i.e. an upper bound on the mean propagation delay.

$$T_{ub} = L\;W_{ub}$$

In the next chapter we investigate the accuracy of these models and predictions.

6. Related work and potential applications.

The granularization approach taken by us was inspired by related research in concurrent management of index structures for databases [Lanin '86, Kung '80, Ellis '85, '80a, '80b, Manber '83]. The deadlock avoidance and consistency maintenance schemes are based on extensions of two phase locking protocols [Easwaran

'76], namely the *tree* and *dag* protocols [Silberschatz '80]. These protocols have been directly applied in both our structures. Lock coupling based schemes as proposed in [Bayer '77] are not directly applicable to concurrent heap and software banyan algorithms because of the potential of deadlock as demonstrated in section 3.1.3 of this chapter. However, variants of our algorithms might be able to utilize lock coupling schemes as discussed in iii) below.

One might ask are there any other concurrent structures that may be used as priority structures, and if so what are the benefits of using the priority structures proposed in this chapter? One structure that immediately comes to mind is the binary search tree. Binary search trees use a linked implementation to represent a total order. The main drawbacks of binary search trees are:

a) They have a single serialization point namely the root of the structure. The simple manner in which we extended the concurrent heap into the software banyan structure is not applicable to a binary search tree.

b) They require rebalancing. The amount of effort spent in rebalancing a binary search tree is not needed for a partially ordered structure such as a heap. Thus concurrent maintenance (balancing) of binary search is an expensive operation with many more operations within a critical section, as compared to concurrent maintenance of concurrent priority structures.

We know of no existing parallel algorithms (that do not use special purpose hardware, such as systolic arrays) for manipulating heaps and structures based on the heap, such as the software banyan. In chapter 6 we contrast our approach to that of introducing special purpose hardware such as systolic arrays. A potential use for the CHEAP is in parallel resource management. The resource management component of an operating system performs the mapping of requests for resources upon the configuration of resources available to the execution environment, in conformance with the allocation policies of the installation. The traditional approach has been to reflect policy in terms of priority for access to resources, and to implement the mapping of requests via priority queues of requests for resources. A workload executing on a high performance multiprocessor execution environment can be expected to generate requests for resources at a very high rate.

The CHEAP and SBAN are generic priority structures. Priority for access to any resource or combination of resources can be implemented. It is possible, for example, to use a CHEAP for allocation of resources in scheduling coordinated blocks [Ousterhout '82]. In related work Suhler [Suhler '87b] analyses scheduling of loop iterates in the context of TDFL. Weak priority structures and index server objects are both usable in this context. Thus our algorithms are proposed as a basic building blocks for operating systems for multiprocessor architectures.

Some further details are worthy of note.

i) Locking modes. A related issue is the mode of locking. We feel the main reason for accessing a prioritized structure is to add and delete items. For this reason, we see no use for a more permissive locking strategy than the one presented in this thesis, for example, *read* locks. Writer exclusive locks themselves can be used for unimpeded searching (read only) of the data structure, which may return stale information. In some cases such stale information may be acceptable. However, our algorithms do not utilize the read only option.

ii) Parallelizing the work queue. There are various ways of parallelizing the work queue. One way is as we have suggested above, i.e. to have multiple servers serve a common work queue. On some current multiprocessors, this form of queue is very efficiently supported, by making use of some primitive instructions [Gottlieb '83], that enable combining Enqueueing or Dequeueing requests as these requests travel in the machine.

iii) A very simple algorithm for the concurrent heap that is promising, is to have both inserts and deletes propagate downwards after entering at the root of the structure. We abandoned this scheme because of the presence of null values that caused "holes" to appear in the heap, thereby distorting its structure. Although the presence of holes is theoretically inelegant, in practice performance of hole based schemes may be attractive. This scheme is currently being analysed by our colleagues [Rao '87b] for its performance on AI search algorithms being implemented on parallel machines.

7. Conclusions:

In this chapter we have presented algorithms for maintaining concurrent heap and software banyan structures. Although our formulation assumes a shared memory multiprocessor, with writer exclusive locking capability at each memory word the structures themselves may be mapped onto distributed memory machines. One such mapping has been analysed and conditions for optimal mapping have been derived. The algorithms remove the serialization bottleneck by reducing the size of critical sections. A drawback of these algorithms is that global mutual exclusion is still required albeit for a very small duration. To evaluate the performance of these priority structures we have performed detailed implementations and simulations, results of which are presented in the next chapter.

Chapter 6 - Performance measurements

1. Introduction

In this chapter we discuss performance of concurrent priority structures. The performance measurements are included in two phases. The parallel execution environments available to us had only a small degree of parallelism and so the implementation of the CHEAP was done (on a 10 processor Sequent Balance 8000) primarily to gain a detailed understanding of the execution behaviour of the CHEAP and to obtain parameterization and validation for the simulation model. The simulation model applies to execution environments with large numbers of processors.

As might be expected, the behaviour of CHEAP and RHEAP (the regular heap algorithm) on the implementation on a small number of processors was comparable. The performance of the CHEAP and the SBAN on larger systems was as expected, much more favourable to CHEAP and SBAN. Section 4 deals with direct implementation of priority structures in hardware. We demonstrate how systolic structures may be constructed for the priority structures proposed.

2. Implementation of the concurrent heap

In this section we discuss the implementation and performance of the CHEAP. In section 3 we discuss simulation results of the CHEAP and SBAN.

2.1. The activities

Two types of activities, *Inserters,* and *Deleters* perform repetitive cycles consisting of user computation (the time spent in doing this is called *compute time*), followed respectively by inserting into and deleting from a shared concurrent heap (see Fig 2.1). The concurrent heap is maintained by means of a set of activities called *Restructuring* activities, that mutate the structure while preserving its consistency. An auxiliary data structure, the Work Queue is also used. An activity corresponds to a process in our implementation.

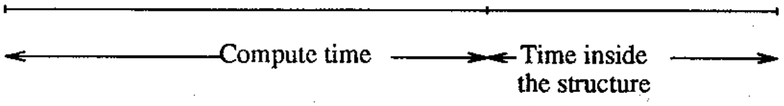


Fig 2.1. Time spent by each activity

2.2. Hardware description

The implementation was carried out on a quiescent Sequent multiprocessor machine with 10 processors. The machine has 64 Kbytes of atomic lock memory (ALM) on its Multibus adapter board. A lock is synthesized from a 32 bit word, the least significant bit of which indicates the status of the lock. This gives a total of 16k locks. The lock is implemented with a simple test and set mechanism in an indivisible memory cycle. There is also a shadow locking scheme to offload the system bus. To give the abstraction of an unlimited number of locks, Sequent has multiplexed the use of each hardware lock through software, thereby deteriorating the performance of programs that use locks frequently (such as our program). A pair of instructions consisting of a lock followed by an unlock takes approximately 52 microseconds as a result of shadowing and multiplexing. For measurements we had access to a microsecond timer board with a free running register. The overhead of accessing (reading) this was 25.5 microseconds.

2.3. Workload

The workload was synthetically generated, and consisted of a variable number of increments of a counter. This is a little simplifying, since in real application programs, demands on memory made during the computation stage will surely conflict with demands on memory made by the concurrent data structure. The effects of these conflicts have not been studied.

2.4. CHEAP vs RHEAP

For comparison with the best known existing algorithm for priority queues, we implemented a parallel heap algorithm (RHEAP for regular heap). The regular heap is a conventional heap in which inserters and deleters update the structure in a single global critical section. Additional overhead due to procedure calls and other sundry instructions caused the minimum achievable workload granularity to be restricted to 1400 microseconds.

In order to get an idea of the relative magnitudes of the critical sections for the two structures, we provide a number of time charts. The time components in Fig 2.1. may be further subdivided as shown in Fig 2.2. The individual patterns of instructions that appear in the C - language source programs for the the CHEAP and RHEAP inserters and deleters are shown in Fig 2.3. "I" denotes an instruction (conditional branch, assignment, arithmetic operation etc). Instructions can take between 10 and 50 µsecs. *jsr* and *rtn* denote, as their names suggest, a procedure call and a return from a procedure call respectively. The call and return take on the average 70.3 µsecs together. Branches (if statements) have been depicted as forks in the timing diagrams. A repetition (while statement) is depicted by a feedback loop. Branching probabilities are key parameters of the program. Data collected on these probabilities serve as input to a more extensive simulation study. In gathering this data we have taken care to avoid counting instructions that concern time measurement or instructions that deal with gathering statistics. Such instructions are present in both algorithms and thus can be ignored.

In summary, time is spent within the structures in two ways, namely in *waiting for locks* and in *computing*. The former depends upon the degree of *contention* in the system. Contention may be altered by changing *compute granularities* and the *number of activities*. The effects of contention may therefore be studied by varying these two parameters. The latter depends upon the time to do *arithmetic* and *memory access* operations. The use of virtual memory brings about unpredictable delays in memory access time, giving rise to unexpected behaviour of the CHEAP and RHEAP algorithms (see plot in Appendix 7). A page fault may be *minor* (the page resides in main memory and is waiting to be flushed out), or *major* (the page is not in main memory and must be copied from the disk). Both types of page faults dominate the performance for large structures.

Finally, there are *algorithm−intrinsic* delays, such as the time spent by a CHEAP deleter in waiting for the topmost element of the structure to become non yellow. This is represented by the feedback loop in Fig 2.3.1, showing what happens when the color of the root node is yellow. Our measurements were taken with the minimum RSS (resident set size) set at 16, or 16 pages of size 1 Kbyte each; and the maximum RSS set at 750. Increasing these may reduce the anomalies observed.

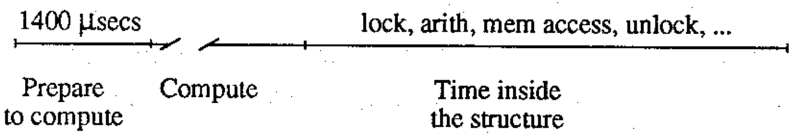

1400 μsecs
lock, arith, mem access, unlock, ...
Prepare to compute
Compute
Time inside the structure

Fig 2.2. Detailed breakup of Fig 2.1

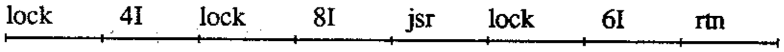


Fig 2.3.1 CHEAP insert

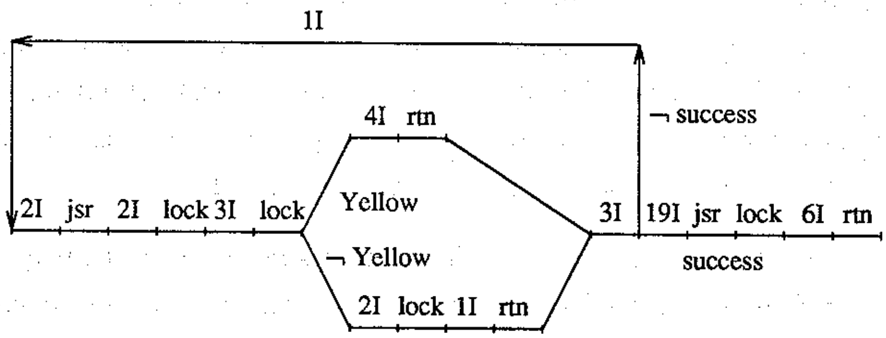


Fig 2.3.2 CHEAP delete

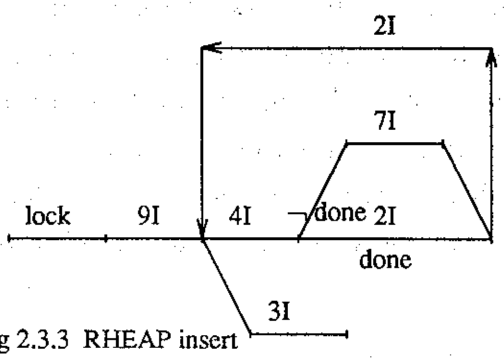


Fig 2.3.3 RHEAP insert

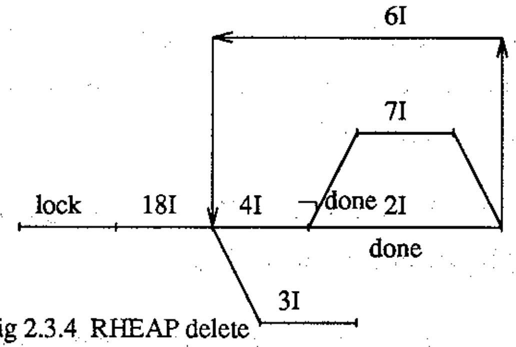


Fig 2.3.4 RHEAP delete

2.5. Quiescence

The purpose of our experiments, among other things, is to test the validity of formulae for thruput and delay derived in chapter 5 section 5. Since these equations had been derived for quiescent conditions we had to first establish quiescence in our structure before taking measurements. To achieve quiescence we tried a number of schemes and then settled on the following.

A serial algorithm is used to insert a number of items initially into the structure, coloring each item blue once it satisfies the heap property. Next a number of inserts, deletes and restructures are carried out upon the structure, so as to introduce colored values in the CHEAP as well as in the work queue. At this point a reference clock is started and computation and restructuring activities are started by forking a process for each activity. A fixed amount of computation is now performed, with a fixed number of inserts and deletes upon the priority structure. All measurements are taken with respect to the reference clock.

2.6. Measurements

The overall time to completion, the time spent inside the CHEAP and the time spent outside it were measured for each run. In addition, counts of the number of restructurings, the number of attempted work acquisitions were also gathered. We also measured the times spent outside and inside the CHEAP and the average time for red/yellow values to turn blue. Data collected during a run was fed through a filter which extracted statistics. The results confirmed our derived formulae (see Appendix 8).

2.7. Implementation results

Appendix 7 shows the performance of CHEAP and RHEAP varying the parameter size (depth) of the structure (Fig 1.1). The key points that emerged from this implementation are as follows:

1. CHEAP vs RHEAP: Overall, RHEAP performed better than CHEAP although CHEAP did better in a number of cases.

2. Ratio of restructuring rates: If r_i and r_d denote rates of insertion and deletion respectively, and r_{ir} and r_{dr} denote rates of insert and delete restructuring respectively, then we had assumed in chapter 5 section 5 (and Appendix 8) that the ratio $(r_i \ / \ r_r)$ is equal to the ratio $(r_{ir} \ / \ r_{rr})$. This assumption was proved wrong. For large structures (> 10 levels deep), a yellow item that is randomly picked has to travel almost the entire depth of the structure (recall that a binary tree has half its nodes at the base level). This conclusion was apparent from our observation that a yellow item makes approximately k trips through the work queue, where k is the depth of the structure. In contrast, a red item only makes between 1 and 4 trips on the average, through the work queue (smaller as the depth increases).

   Thus an inserted (red) item doesn't have very far to go to reach its final position and turn blue. However a deleted item gives rise to a yellow item that must travel most of the levels before finding its proper place in the structure. Other reasons for red values changing color faster than yellow values are, a) yellow items are requeued more often in the restructuring activity, and b) a number of red items turn yellow almost immediately upon insertion because of a concurrent deletion.

3. Deadlocks: The simple deadlock avoidance scheme of ordering the resources, is adequate for the CHEAP.

4. Number of work queues: The number of work queues did not significantly affect the measured performance of the CHEAP. Insufficient work in the work queue may be a reason for this, although it does not fully explain it.

5. Yellow chains: The formation of yellow chains is indicated by the fact that a yellow node makes many trips through the work queue before turning blue. On occasion the number of such trips were found to be greater than the depth of the structure. This indicates the formation of *chains* of yellow nodes, which is a major source of inefficiency in the deletion operation.

6. Granularity: As granularity of computation (compute time) increases, so does the propensity for there to be insufficient work in the work queue. We measured the number of futile work (restructuring) acquisition attempts from the work queue (WQ). For compute granularities above 100 millisecs, this was significant. Further observations are made in Appendix 8.

7. Locking granularity: It may be possible that CHEAP algorithms that use locking at an intermediate locking granularity, i.e. somewhere between the entire structure and each individual node; will prove useful even on a small processor configuration. Algorithms using medium locking granularities will be significantly different from our current fine grained lock algorithm. Their proofs will also need to be significantly different. For these reasons we abandoned experimenting with schemes that dealt with intermediate locking granularities.

2.8. Overall conclusions from the implementation

The most positive result of this batch of experiments was that CHEAP compares favorably with the most popular algorithm (RHEAP) for manipulating priority structures concurrently. In fact CHEAP did better in some cases, even though there was very little scope for parallelism (considering the small number of processors available) and higher overhead in the CHEAP. In general, with the small number of processors on the Sequent it is not possible to obtain interesting levels of parallelism for which structures such as the CHEAP will be useful. Such levels of parallelism may be introduced by increasing the number of processors or reducing the work granularity. The former is severely restricted because of the machine. We expect that the multiple work queue option will prove useful on larger machines. On our implementation it did not prove beneficial; in fact there was a deterioration in performance with multiple work queues.

A disadvantage with the Sequent machine that the tuning parameters make it almost impossible to totally avoid paging, context switching, daemons in the background, remote file transfers and so on. While taking highly sensitive measurements in the order of tens of microseconds, these random factors influence the readings to a great extent.

If we were to repeat this exercise we would have the same activity doing both insert operations and delete operations, rather than separating inserting activities from deleting activities. We feel this will lead to better utilization of the processors and distribution of values. We would also try to minimize on overhead for measuring time, and introduce in-line expansion of work queue accessing function calls, to reduce overhead. We would also prefer to use a machine in which each memory cell is lockable.

3. A hybrid simulator for the CHEAP and SBAN

This section is concerned with the design of a hybrid simulator for the CHEAP and SBAN structures.

3.1. Objective of the simulation

There are two reasons why this simulation was performed. The major reason was to generate sufficient workload for testing the priority structure algorithms. With our experiments on a 10 processor machine, it was not possible to get heavy workloads with reasonable compute granularities. Another reason was that we wished to eliminate artificial delays caused by virtual memory and other limitations on our test machine and have a uniform basis to simulate our algorithms over larger machines and different mixes of instruction times.

3.2. A hybrid simulator

A hybrid simulator is a simulator in which the actual control structure of the system under study is retained while the timing measurements are taken with respect to a simulated clock. Thus the logical properties of the system and the interactions between system components are faithfully reproduced while the timing properties are made independent of real execution time (which may be affected by extraneous factors such as slow virtual memory, restrictions on the number of processors online, locks and so on).

We started with the implementation of the CHEAP algorithm as discussed in section 1. To convert this implementation to a hybrid simulation, we superimposed upon this algorithm, an event-driven simulator, and provided markov chains for each type of activity. Each activity was initialized to be in a particular node in its markov chain. The memory nodes of the system were initialized at a quiescent state and thus steady state was possible from the start of the simulation. The same principles were used to simulate parallel execution of the RHEAP (the regular heap algorithm, which is the most popular alternative), and the SBAN.

3.2.1. Simulation assumptions and results

Appendix 9 shows the markov chains for the different activities that were simulated for the CHEAP, RHEAP and SBAN algorithms. For the first two, the markov chains were synthesized from the timing diagrams shown in Figs 2.3.1 - 2.3.4. For instance, the markov chain in Fig 1.1 in Appendix 9 is interpreted as follows: an inserting activity spends a certain amount of time known as the *Compute Time* in the node 1. It then moves to the state "attempting to lock PQ" in node 2. The *mean* time spent in this node per visit is naturally a very small fraction of the time spent in node 1. Similarly node 3 is a compute node, with a few normalized instructions. Node 4 is a lock node, signifying the locking of the last element of the CHEAP before insertion, and node 5 represents the final few instructions that must be completed before the activity can return to its compute node. We have assumed exponentially distributed service time distributions at each state.

We have collapsed the sequence of lock nodes in the markov chain for restructuring in a CHEAP algorithm, into a single node in corresponding markov chain of the SBAN algorithm. This is because a banyan may have a variable number of such nodes depending upon its spread and fanout. The chains for SBAN are shown in Figs 1.6 - 1.8 in Appendix 9.

The timings for the various nodes are parameters that depend upon the memory access time, lock acquisition time and similar parameters of the target architecture. We have assumed times shown in the chart on Fig 2 (Appendix 9). These timings are obtained by taking instruction counts from our implementation and some gross estimates of instruction execution time. These timings are applicable to a large scale multiprocessor such as the RP3 which has a uniform capability to lock each memory cell.

For the main data structures being studied, each attempt at obtaining a lock is modelled by a suitable state transition in the corresponding markov chain. However this was not done for the auxiliary data structure, namely the work queue, since the auxiliary data structure was not the main focus of our attention. Thus we could restrict ourselves to the simple markov chains shown in the Appendix 9.

Results of the simulation are presented in Appendix 10.

3.3. Conclusions drawn from the simulation

We got a substantially better performance for the CHEAP and SBAN than for the RHEAP, with less than 2.5% of the number of inserting and deleting processors devoted to restructuring. Fig 3 in Appendix 9 shows the breakeven point beyond which adding more restructuring processors does not help, because there is not enough work to go around. An analytical way to arrive at this breakeven point is to calculate the maximum throughput of restructurings required and to ensure that there are at least as many restructuring processors to deliver that throughput. We assumed compute granularity of 1 sec, with an equal number of inserters and deleters. We feel that an investment of 2.5% of the processor resources is not unreasonable, especially when the benefits are so visible (see Appendix 10 Fig 1.1).

As expected, the CHEAP and SBAN algorithms show almost constant times over a large range of sizes of the structure (Appendix 10 Fig 1.2). The falling trend in the cost of insertion and deletion is accounted for by reduced contention as we go to larger sizes of the CHEAP and the SBAN. The RHEAP algorithm pays a heavy penalty in the delete phase, that grows linearly with the size of the structure. This penalty is also paid in the case of RHEAP inserts, where the distribution of inputs is skewed, with insertions of very large values into an already constructed heap.

Compute granularities must be moderately large to warrant effective use of any concurrent data structure, including the CHEAP, SBAN and RHEAP. Computations with average compute granularities smaller than 1 sec pay a steep price in terms of contention. This price decreases dramatically as we increase the compute granularity (Appendix 10 Fig 1.3.) The outstanding performance benefit of a SBAN over a CHEAP is that it performs better at smaller compute granularities. This is a significant improvement, since it becomes more and more promising as we increase the number of processors.

Finally, the crucial time in the restructuring algorithm is the cost of doing a compare. It may conceivably be required to organize a priority structure on a comparison between complex structures with several fields. In this case the RHEAP algorithm falls woefully short, because of the large critical section. The CHEAP algorithm is clearly better as we go to higher and higher compare costs (Appendix 10 Fig 1.4).

3.4 Domain of Characterization

Our experiments and simulations involved five parameters:

i) Size of structure (the number of values inside the structure).

ii) Number of Processors (doing insertions and deletions).

iii) Compute Granularity (in millisecs).

iv) Number of Restructurers.

v) Comparison Time.

The domain of characterization of these parameters is discussed in this section. The first three parameters are the most significant. The fourth parameter i.e. *number of restructurers* is linked to the rate of inserts and deletes by formulae derived in chapter 5 and these formula have been validated (Appendix 8). We assume the presence of adequate restructuring resources (either in hardware or software) and hence this parameter is not discussed any further.

As regards the last parameter i.e. *compare time*, our charts and diagrams have been plotted considering the smallest possible compare time. If compare time is increased all algorithms degrade in performance, however the degradation in performance of the concurrent heap is far smaller than that of the regular heap. Thus the concurrent heap (and software banyan) are increasingly favorable over the regular heap with increase in compare time.

3.5 Estimation of surface coordinates.

Based upon the results presented in the appendix we arrive at the three surfaces shown in Figs 3.1 - 3.3. These surfaces have been constructed by estimation based on the shape of known curves and the assumption that the projections of each surface normal to both its parameter axes are parallel to the known curves. For example in Fig 3.1 we assume that a plane normal to the *size* axis will intersect the RHEAP and CHEAP surfaces in lines that are parallel to the known curves of Appendix 10, Fig 1.3. Similarly planes normal to the *granularity* axis will intersect the two surfaces in lines parallel to the plots of Appendix 10, Fig 1.1.

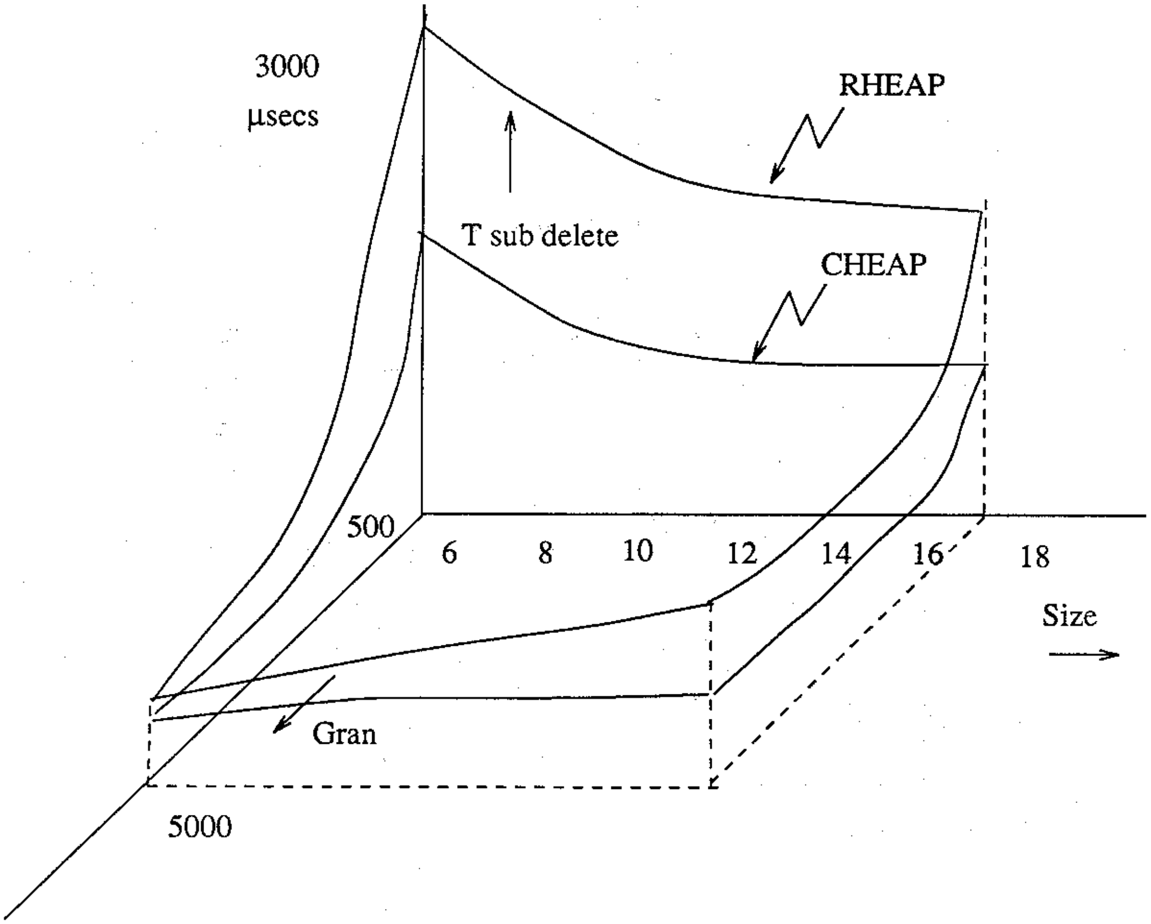


Fig 3.1 Estimated surfaces showing dependence of delete time upon *size* and *granularity*, keeping *number of processors* constant at 500.

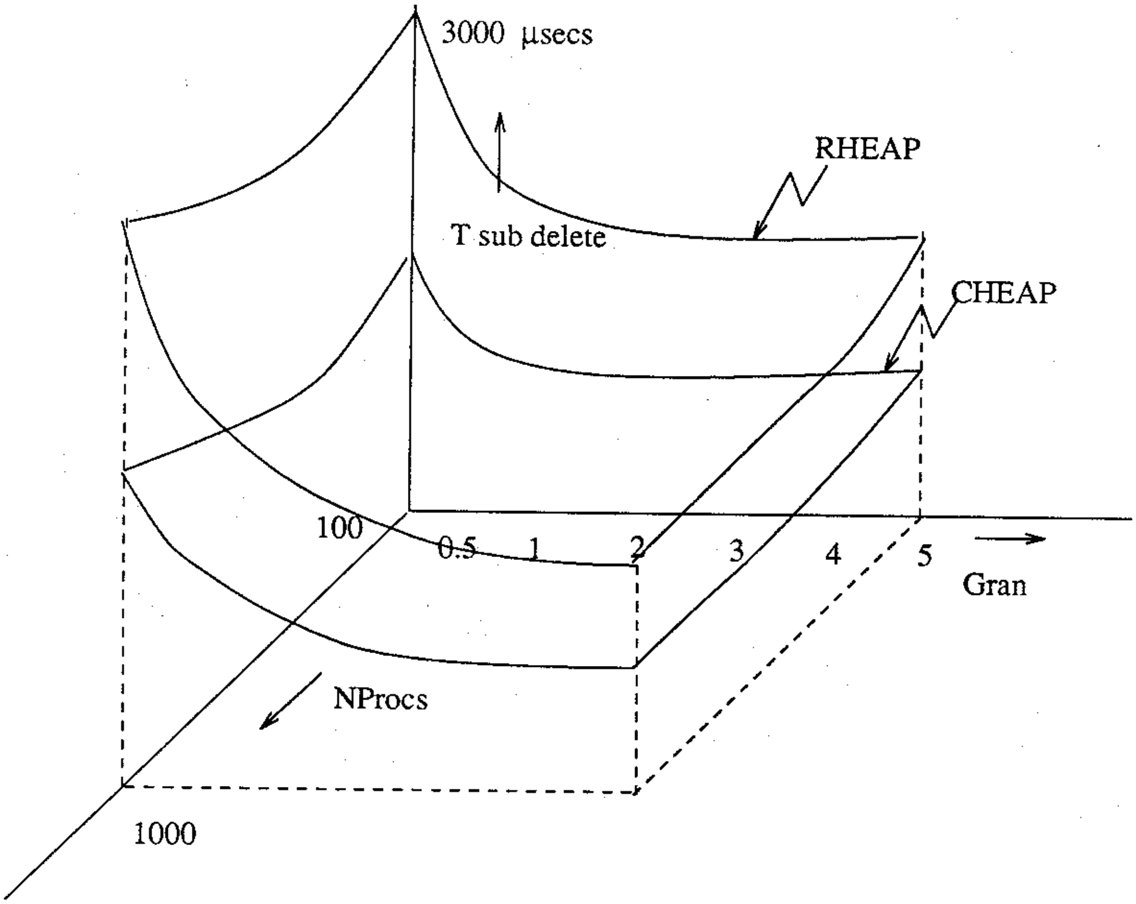


Fig 3.2 Estimated surfaces showing dependence of delete time upon *granularity* and *number of processors*, keeping *size* constant at 12 levels (4096 nodes).

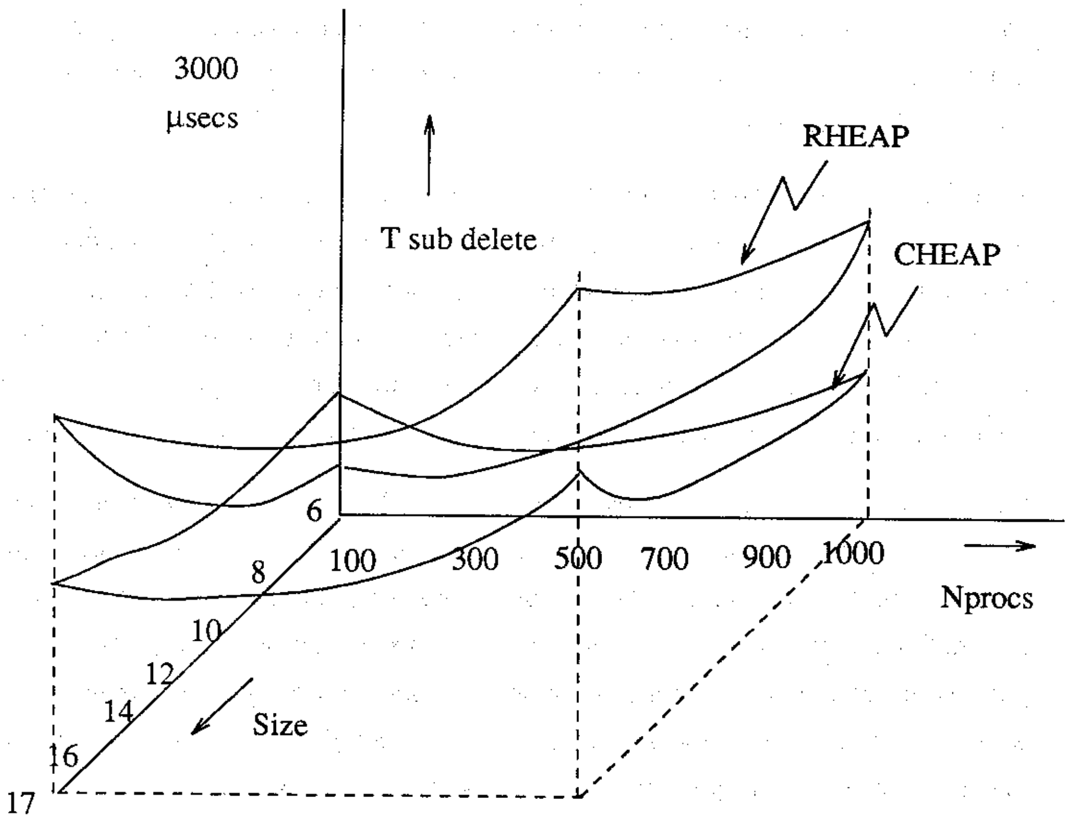


Fig 3.3 Estimated surfaces showing dependence of delete time upon *number of processors* and *size*, keeping *granularity* constant at 1 sec.

4. Specialized Hardware for priority structures

4.1 Related work

Existing algorithms for systolic priority queues are all based on one of three designs: a) *array based* designs [Leiserson '79, Guibas '82], where values propagate down an array and stop when they have found their proper position, and b) *tree based* designs called *dictionary machines* [Ottmann '82, Atallah '85, Somani '85], where the data is organized in a sorted / unsorted manner and mapped upon a tree of processors; the edges of the tree being used for communication between the processors, and c) Varman's fault tolerant design [Varman '83] that embeds a depth first search tree on a two dimensional mesh working around faulty processors. All of these designs provide only a single I/O (input / output) port for data transfer from / to a host machine, and use one processor per data item (or a small number of data items).

A serial bottleneck such as the I/O port of existing designs for systolic priority queues, may fail to utilize the performance of these specialized machines. We believe that the need for high performance priority queues will be increasingly felt in conjunction with multiple processor systems, where there may be tens or hundreds of "hosts" that may need access to the same priority queue. In such usage configurations it is not hard to imagine the need for multiple input/output ports.

Given current and forecasted technology it will be very expensive (see Appendix 11) to construct linear arrays or trees of processors for large priority queues (upwards of 10^6 data items). When cycle time is not the most critical parameter it is beneficial to try alternate designs that use a small number (O(log N)) of general purpose processors each with a fair amount of local memory, rather than large networks of simple processors. Fisher ([Fisher '84]) has proposed a dictionary machine that is not "processor profligate", in that it uses as many processors as the size of the longest key and a systolic *trie* structure to support search operations on an ordered set. Carey *et al* [Carey '84] have proposed a dictionary machine with a small number of O(log N) processors that uses a variation of the 2-3 tree known as the 2-3-4 tree to maintain a systolic index into a large search table. However both these designs lack the ability to exploit multiple I/O ports.

In this paper we present systolic priority queues of both designs: i.e. the PPN or "processor per node" design which is an idealized design with O(log N) input ports, and the PPLM or "processor per level with multiple input ports" design. PPLM machines use O(log N) processors with M input ports, where N >> M > 1. A systolic banyan architecture is proposed as a third alternative. This architecture is interesting in that it has multiple output ports as well as input ports.

The only internal ordering maintained inside these structures is one that is consistent with the *heap property*, i.e. the value at a parent node should be greater than or equal to the value at each of its children. The algorithms presented are variants of similar algorithms that we have developed in the context of general purpose multiprocessor [Biswas '87b]. The ideas are also related to similar work on concurrent data structures [Kung '80b, Ellis '80a, Quinn '84] and work on distributed implementations of *balanced* search structures on hypercube architectures [Dally '86]. We assume familiarity with the literature.

4.2 Systolic priority queues based on binary trees

The most popular sequential algorithm for a priority queue uses a heap † data structure [AHU '74]. In this section we present a systolic algorithm for a priority queue using an *unbalanced* heap, in other words a binary tree in which leaves may appear at any level as long as the the heap property is satisfied at every node. Our algorithm assumes a machine in which a processor is assigned to each internal node of a fixed size complete binary tree. Unbalanced heaps are dynamically formed in this machine by values occupying the processors. In the latter half of this section we present a simple mapping scheme that reduces the number of processors to a small number, each with a fair amount of memory. The requirement that deletes go through a single output port is relaxed in section 3 through the introduction of a banyan structure. The banyan is more complex than a complete binary tree, but a simple mapping onto a few processors is again possible.

4.2.1 PPN machines

In a PPN machine N processors are arranged as a complete binary tree of size N (we assume for convenience that $N = 2^n - 1$, for some integer n). The root has an output port and each leaf has an input port. Each processor contains a register called its data register which is initialized with a null value that is smaller than all possible data values. Each processor has the capacity to store exactly one value, or node of a complete binary tree and hence the name processor per node.

† A *heap* is a binary tree in which leaves appear only in the two lowest levels, and the leaves in the lowest level appear in the left-most places (i.e. they are left-justified). In addition the values in the nodes of a heap must satisfy the heap property.

4.2.2 Rewrite rule

A processor operates in cycles, each cycle being broken up into two phases, the *parent* phase and the *child* phase. All processors in the same level must be in the same phase and adjacent levels are always in different phases. With the exception of the root, when a processor is in the child phase it is passive and allows its data register to be examined (and possibly mutated) by its parent processor. Similarly except for leaf processors, all other processors are active in their parent phase and may mutate their own data registers and those of their children. Since phases alternate, a processor that is in the parent phase in a given half-cycle must be in the child phase in the subsequent half-cycle.

The rewrite rule for this machine is simple. Let p, lc, and rc denote values in the data registers of a parent processor (i.e. a processor in the parent phase) its left child and its right child *prior* to an application of the rule. Similarly let p', lc', and rc' denote values in these registers *after* application of the rule. The rewrite rule is stated as follows (we use the multiple assignment statement):

$$\text{if } (lc > p \ \Lambda \ lc \geq rc) \text{ then } \quad p', lc', rc' := lc, p, rc$$
$$\text{else if } (rc > p \ \Lambda \ rc \geq lc) \text{ then } \quad p', lc', rc' := rc, lc, p$$

An application of this rewrite rule is demonstrated in Fig 4.1.

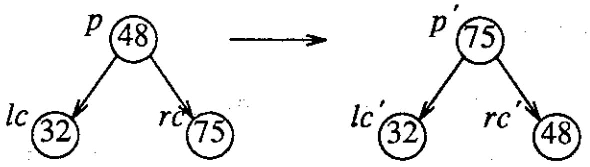


Fig 4.1. Example application of the rewrite rule

Intuitively the rewrite rule corresponds to the "pushup" phase of the well known sequential heap algorithm. Obviously this rule does not apply to the parent phase of a leaf or the child phase of the root. Each leaf processor allows an insert (input) operation in its parent phase. Similarly the root processor allows a delete operation in its child phase. A delete operation removes a value from the root and puts a null value in its place. A delete fails if the root contains a null value at the time of deletion. An insert fails if the leaf involved contains a valid data value (not a null value) at the time of insertion. Failure is appropriately signalled to the host generating the request.

An inserted value is said to be propagating if it makes progress to a new level at every cycle following the insertion. A value v inserted at clock cycle t stops propagating within log N + 1 cycles of t. This is obvious, since there are only log N levels in the tree. If v is greater than any other value in the tree then it must rise to the root in log N cycles. Seven iterations of this algorithm are illustrated in Fig 4.2.

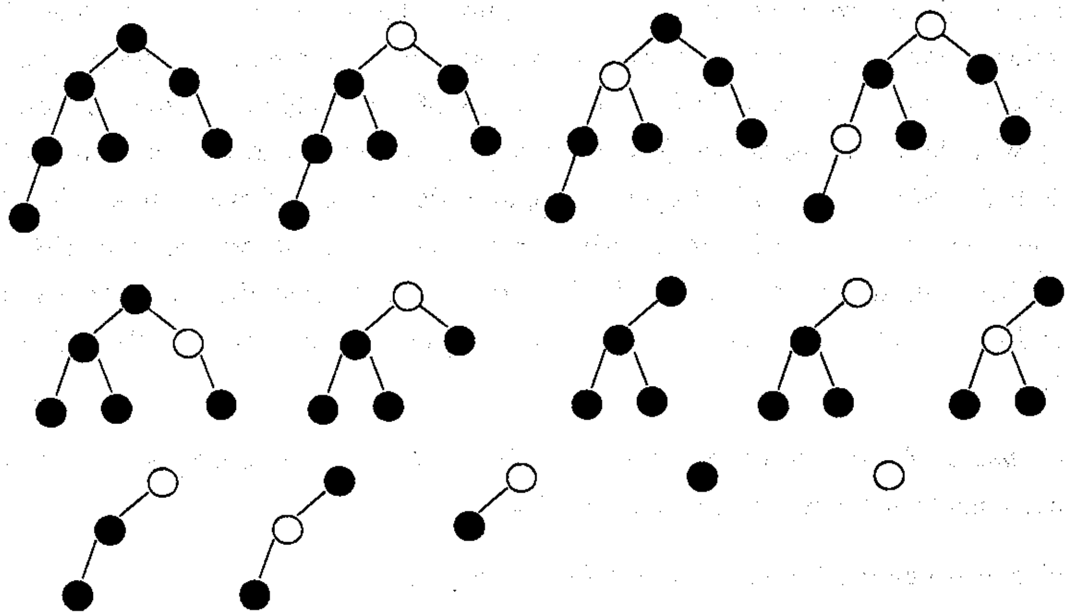


Fig 4.2. A sequence of deletes upon an unbalanced heap

4.2.3 Failure performance of deletes and inserts

a) Delete failures.

Given that the structure has k values for a quiescent period (no I/O) of log N cycles we claim that there will not be a delete failure until at least k deletes have taken place (for example in Fig 4.2, k = 7). This is stated more formally as the following lemma.

Lemma 1: If a PPN machine has k processors occupied by values for a quiescent period of log N cycles, then the next delete failure can occur only after k deletes have taken place.

At the end of the quiescent period the k values in the PPN machine must be organized in the form of an unbalanced heap rooted at the root processor. This is obvious since every value has had log N cycles within which it must have reached either the root or been stopped by a value that is larger. Fig 4.2 depicts an unbalanced heap with 7 values at the end of the quiescent period.

To prove Lemma 1 we first introduce some terminology. We shall use *heap* to mean *unbalanced heap*. A processor is said to be *inside* the heap if *either* its value is non-null *or* it has a descendant whose value is non-null. This property of a processor is known as its *inside−ness*. Levels of the machine are labelled from 0 up with the label 0 being assigned to the root. *Even* levels start at 0 and *odd* levels at 1. An *odd cycle* is a half-cycle (see earlier description of cycle) at which only processors at odd levels are in their parent phase (i.e. the rewrite rule may be applied). Similarly for *even cycles*. Thus the first half-cycle after quiescence is an odd cycle and deletes can occur only in odd cycles. A null value is also called a *hole*.

Lemma 2: At the end of the i^{th} even cycle, $i < k$, the only processors *inside* the heap that have holes in them are at *odd* levels.

Note that Lemma 1 directly follows from Lemma 2.

Proof of Lemma 2:

For $k = 1$ the lemma is trivially true. Assuming $k > 1$, we use induction on the number of completed even cycles.

Base case:

At the end of the 1^{st} even cycle, by inspection the only hole *inside* the heap *may* be (it need not be, for example if the root has only one descendant along a particular branch) at level 1.

Inductive hypothesis:

After the i^{th} even cycle ($i < k$) the only holes *inside* the heap are at odd levels.

We must show that after the $i+1^{th}$ even cycle the only holes *inside* the heap will be at odd levels. Let $a_1, a_2, \cdots, a_j$ be the holes *inside* the heap at the end of the i^{th} even cycle. Let us consider one of these holes, say a_m. Let a_{m_1} and a_{m_2} be the children of a_m. Then *either* a_{m_1} or a_{m_2} is *inside* the heap since it must lie on the path from a_m to a descendant of a_m that is non-null (by definition of *inside−ness*). Thus either a_{m_1} or a_{m_2} must be non-null by the inductive hypothesis.

We have demonstrated that *every* hole *inside* the heap at the end of the i^{th} even cycle has at least one non-null child. At the end of the $i+1^{th}$ odd cycle (the half-cycle immediately following the i^{th} even cycle) there will therefore be a value to fill each hole *inside* the heap and each hole will propagate down by one level into an even level (and in some cases exit the heap). Thus at the end of the $i+1^{th}$ odd cycle the only holes *inside* the heap are at even levels.

We now use precisely the same argument as above once more, to show that at the end of the next half-cycle, which is the $i+1^{th}$ even cycle the only holes *inside* the heap are at odd levels. □

b) Insert failures.

In the best case values will be distributed during insertion in such a manner that the tree is filled uniformly and it is completely full at the time of the first insert failure. In the worst case, assuming that a sequence of inserts are carried out at the input port of the same leaf processor there will be an insert failure within log N + 1 cycles, even though the rest of the tree may be sparse. Thus the worst case performance of the PPN machine for insert failures is unfavorable. The average case performance of this structure is yet to be investigated.

The large number of input ports (one per processor) of the idealized PPN machine is infeasible due to practical restrictions. To alleviate this problem, input channels may be shared between a number of input ports. Sets of leaf level processors may be mapped onto the same chip in such a way that a single input channel is sharable by all processors on the chip. Similarly all chips on a board may be made to share the same input channel (bus) with all chips on another board. We shall not address this issue any further since it has to do with specific details of an implementation. We now address the issue of cost reduction (by means of reducing the number of processors).

4.2.4 Mapping onto a few processors

We use an approach similar to that adopted by Carey *et al* ([Carey '84]) in reducing the number of processors from N to O(log N). The key idea is to associate a single level of the PPN machine with one processor (or a constant number of processors in case of multiple input ports). First we consider precisely log (N + 1) processors organized in a pipeline (see Fig 4.3a). The root processor has an output port and one word of memory (where a word is as many bits as required to store a priority field and an uninterpreted pointer field). Similarly the second processor has two words of memory and the i^{th} processor has 2^{i-1} words of memory. Unlike the PPN machine a processor is responsible for the data values in all the nodes in the corresponding level of the complete binary tree. At every cycle at most one data value may pass through any level of the structure, thus the processors need not interchange vectors of values. It is enough for a processor in the child phase to indicate to its parent the address (array index) and value of the mutated node.

This machine is still systolic in the sense that it has a linear array of processors which pump values from one end to the other, however each processor is more complex than the processor of the PPN machine especially in that it must be able to handle random access memory and communicate addresses. Instead of N/2 input ports there is only one such port (associated with the last processor in the array) and as before only one output port (associated with the first processor).

4.2.5 PPLM machines

We can provide room for M (for some small integer M) input ports for simultaneous insertion, by splitting the bottom row of the complete binary tree into M partitions and assigning them to M processors, each with a single input port. We assume for convenience that M is a power of 2. Thus at the lowest level each processor has (N+1)/(2*M) nodes. This gives rise to log (N+1) + 2*M - 1 - (log M + 1) = log

(N+1) + 2*M - log M - 2 processors. Thus this machine has O(log N) processors since M is a small constant unrelated to N. The resulting machine is shaped like a much smaller binary tree (depth log M) (Fig 4.3b) surmounted with a linear array of processors. Such a machine is called a PPLM machine, meaning "processor per level with multiple input ports".

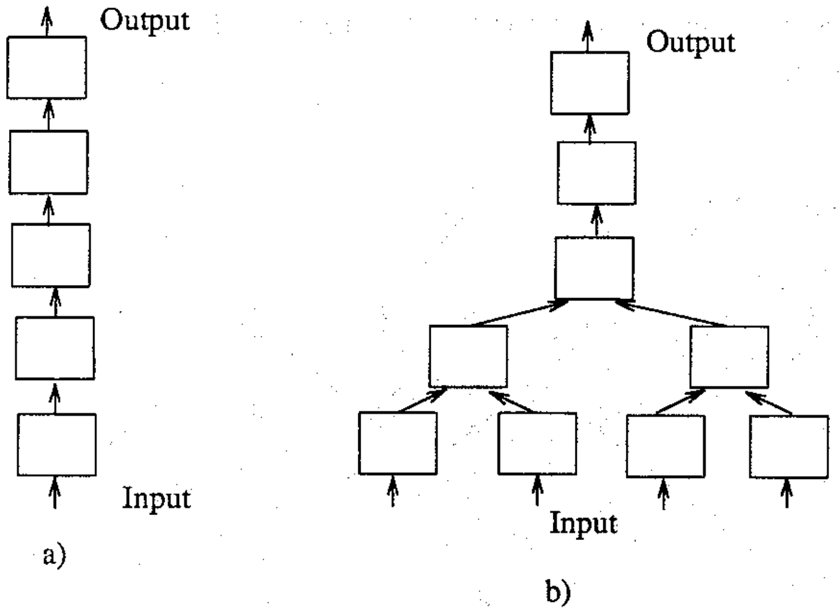


Fig 4.3. PPLM machines using O(log N) processors with N = 31 and a) M = 1 b) M = 4

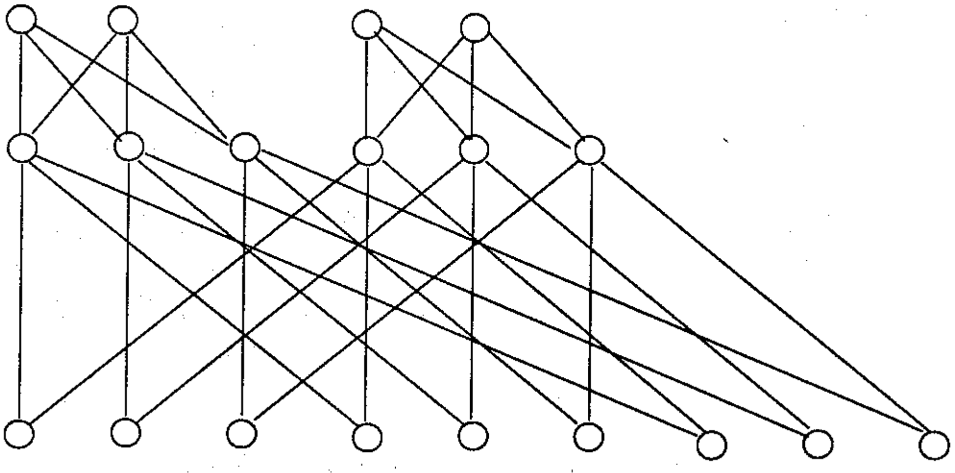
4.2.6 Synchronization

Unlike some existing designs our processors are all locally synchronized. This means that there are no global control signals. The processors compute at their own speed within the constraint that their parent and children phases alternate and a parent and its two children exchange local handshaking messages to synchronize their data transfers and their phases.

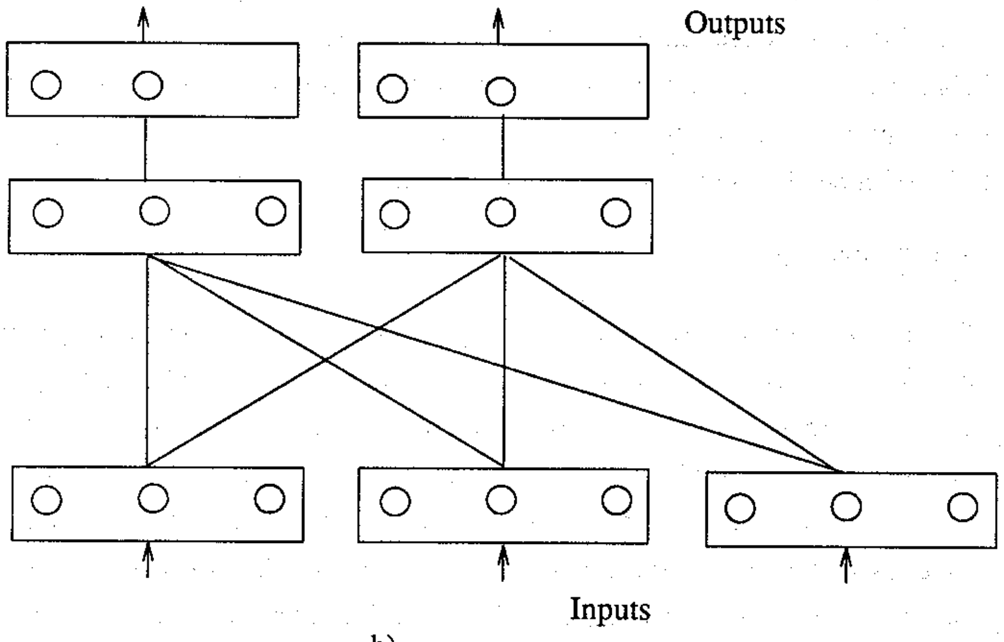
4.3 The systolic banyan

The chief problem with tree implementations of systolic priority queues is that it has only one output port. We can alleviate this problem by means of additional hardware as we have done for the inserts. The idea is very similar to the that of building a software banyan [Biswas '87], except that in this case the banyan structure is partly in hardware and partly in software.

Fig 4.4 demonstrates a systolic banyan. The basic algorithm of cycles consisting of parent and child phases remains in this case. The only difference is that in this case a *set* of parent nodes are associated with a *set* of children nodes. The rewrite rule reorders the values in such a way that all parent nodes are ordered above all children nodes.



a)



b)

Fig 4.4. A banyan and its mapping onto a systolic structure consisting of a few processors: a) shows the logical relationships and b) shows the physical connections between the processors.

An item deleted from a systolic banyan need not be the highest value in the machine. We have derived bounds in chapter 5 to show percentile figures for the relative magnitude of a value that is at the root node of a banyan. The mapping result in chapter 5 section 4.3.2, that demonstrates a straightforward optimal mapping of $k-ary$ hypercubes onto binary hypercubes where k is a power of 2, is also relevant in this context. Unlike purely software based algorithms in banyan algorithms we cannot "place a lock upon an apex node and examine a sequence of apex nodes looking for the maximum value". This scheme runs contrary to the basic nature of systolic computing. Instead we introduce the notion of threshold based deletes. We assume that a delete request arrives with a threshold specification approximately as follows: "return a value if and only if it is larger than X". Now we pipeline the request from apex node to apex node in the banyan until it is satisfied or it has investigated all the apex nodes. At this point the request leaves the machine indicating failure. One problem remains: how do we resolve between two deletes contending for the same apex node? One can devise prioritization rules regarding this as one chooses. A similar approach can be used to perform load balancing at the base level nodes of the systolic tree and banyan. In an approach that is quite similar to ours, Schwartz *et al* [Schwartz '87] have analyzed the suitability of cube class networks for dictionary machines.

4.4 Conclusions

We have presented systolic designs for priority queue machines that have multiple ports for I/O. Versions of these machines that use a few general purpose processors (rather than many specialized processors) are discussed, and it is argued that these machines have favorable cost/performance ratios. A host (or set of hosts) sharing a systolic priority queue through single I/O port may be unable to sustain the rate of requests necessary to run the machine at full speed. Thus if there are bandwidth restrictions on the I/O the cheaper PPLM machines may perform just as well as the more expensive dictionary machines proposed in the literature. In this regard the idea of increasing the number of I/O ports is attractive, because it provides us with a way to increase input and output rates.

Average case failure of the PPLM machine has yet to be characterized. Besides this, two more issues have to be addressed, namely stability and load balancing. If systolic trees are to be useful there must be a balance between the rates of insertion and deletion. If the structure is full the rate of inserts will come down to the rate of deletes. If, on the other hand, the rate of deletes is much higher than that of inserts the structure might lie unused for most of the time. The problem of unbalanced delete and insert rates is partially solved in the systolic banyan structure where deletions are done on a percentage basis.

The second issue is that of load balancing. If insert patterns are skewed parts of the machine may become congested and may even overflow while other parts are lightly loaded. Some systematic way of dealing with load balancing is needed.

Finally, since our machines use fairly general purpose processors it would be worthwhile to look for alternate applications / algorithms for such machines. For example there have been proposals for building dictionary machines with a few processors [Carey '84, Fisher '84]. Our machines could be augmented with additional instruction sets and could be made to work in two modes viz. dictionary machine mode or priority queue mode.

5. Hardware / Software tradeoffs

The following table summarizes the results of the experimental, simulated and analytical performance estimates of hardware and software schemes for concurrent heap structures.

Machine or Algorithm	Cycle Time	Comments (basis)
PPN	250 nsecs	board to board signal delay
PPLM	1000 nsecs	2 memory cycles
CHEAP	828 μsecs	Simulation & Implementation (600 Pc)
RHEAP	1328 μsecs	same as above
SBAN	798 μsecs	same as above

Appendix 11 provides details of predicted costs of the PPN and PPLM designs. It is easy to see that the cost performance ratio of the PPLM design is better than that of the PPN design. The purely software alternatives are attractive since they can be realized on general purpose MIMD machines.

Chapter 7 - Contributions and future directions

1. Contributions

The goal of this dissertation research was the development of concepts and algorithms for resource management on highly parallel computer systems. The conceptual foundations we have explored are extensions of abstract data type models including preparation of external and internal specification and the notion of selecting external specifications which allow valid parallel internal specifications. The specific algorithms developed include an index server object and a class of priority queues which support simultaneous update. Development of the algorithms also engendered extension of mechanisms used to express parallelism, including an extended version of open predicate path expressions.

The index server object is an instance of the *set* abstract data type and may be used by computations that differ only in indexing parameters. The implementation of this object depends upon guidelines for decomposing externally visible states of objects into simultaneously updatable components. Linguistic support is provided for these objects in the form of a mechanism (a type definition facility for abstract data types). The mechanism includes an extension to path expressions that makes it convenient to express patterns of activity bindings and consistency properties to be satisfied at partitions of objects.

Our second major contribution is a class of data structures and algorithms for weakened priority queues. The weakened properties of these objects are expressed at an external level in terms of percentiles and the implementation guarantees that deleted items will satisfy the percentile value requested. The priority structures proposed in this thesis, a concurrent heap and a software banyan, have been found to be efficient and effective. The software banyan was derived from the concurrent heap but differs greatly in implementation. It is a weaker structure and thus has a higher degree of concurrency. The major drawback of a software banyan is that it must undergo periodic copying when it is grown or shrunk in size. In contrast, the concurrent heap (because of its direct mapping onto linear memory) grows and shrinks in size arbitrarily without any overhead. Variants of both algorithms can be easily implemented in specialized hardware, gaining additional performance.

Both contributions are unified under the umbrella of a model for simultaneous update. This model describes an object's execution behaviour in terms of histories of events, or interactions between objects and activities. Abstract or external specifications and implementation states of objects are characterized within this model. It is also stated under what conditions a correct internal history is an implementation of a correct external history.

2. Significance

How significant are these results? The class of work container computations (computations covered by index server objects) is quite frequently encountered in resource management of highly parallel machines. Thus we feel that the index server object has good potential for application on current parallel machines. Priority queues are intrinsic to resource management. Their general utility has been eloquently expressed in [Leiserson '79]. The two weakened priority queues presented in this thesis may be used as building blocks in a large variety of resource management applications where getting the absolutely highest value is not as important as getting one of the highest values. One application is in the management of nodes to be expanded in the course of exploration of a search tree in an AI search algorithm. It may be the case that the node yielding the final solution is not the largest (most promising) current node but one of the largest. Thus any one of the largest nodes is good enough for immediate expansion.

3. Future work and possible directions for extending this work

This work may be extended in several directions. A study of the performance of index server objects over various architectures would be useful. The algorithms for implementation of work managers and synthesis of code from extended path expressions remain to be implemented. The implementation is straightforward and uses well understood software engineering techniques and compiler technology.

More generally, we believe that there may be weakened forms of other structures that are yet to be to be discovered. The challenge is not so much in discovering them as in discovering their use. To illustrate this point, an area within resource management that has been recognized but defied solution for sometime now is *coordinated scheduling*. Coordinated scheduling has to do with the simultaneous scheduling of related tasks for execution at the same time to save on redundant context switching and other reasons. The pioneering work on this was done by Ousterhout [Ousterhout '82] but he made the assumption that task blocks were identified by the user. A more general mechanism for identification of ready groups of tasks at runtime is needed. This is still an open problem.

Other implementations of weakened priority queues may be proposed. As outlined in chapter 6, the idea that insertions and deletions should both propagate downwards seems to be promising from a performance standpoint although the structure now contains holes and is not as elegant.

An area within weakened priority queues that needs further investigation is the use of intermediate grain locking. Algorithms that we have presented here all lock either the entire structure or only the smallest lockable window. This approach have led to simple algorithms that were easy to reason about and implement. We feel that algorithms based upon intermediate grain locking (i.e. algorithms that use larger window sizes) will perform better. These algorithms will be quite different from those presented in this thesis although their design approach should be similar.

We have probed distributed implementations of our weakened priority queues in our mapping results in chapter 5. By building distributed structures such as these in the kernel of certain time-stamp based protocols one might be able to improve the performance of distributed algorithms that use strict priority queues, but could be made to tolerate weakened ones. For what classes of distributed algorithms this approach is suited is something that is yet to be characterized.

REFERENCES

[Ackerman '79] Ackerman W. B. and Dennis J. B. *"A Value-Oriented Algorithmic Language: Preliminary Reference Manual"*, Tecnhical Report MIT/LCS/TR-218, MIT, June 1979

[AHU '74] Aho A. V., Hopcroft J. E., and Ullman J. D. *"The Design and Analysis of Computer Algorithms"*, Addison-Wesley, Reading, Massachusetts, 1974.

[Andler '79] Andler S. *"Predicate Path Expressions: A High-level Synchronization Mechanism"* August 1979, Ph.D. Dissertation, Carnegie Mellow University.

[Arvind '77] Arvind, Gostelow K. P. and Plouffe W. *"Indeterminacy Monitors and Dataflow"*, Proceedings of the 6th ACM Symposium on Operating System Principles, pp 159-169, Nov 1977

[Arvind '78] Arvind, Gostelow K. P. and Plouffe W. *"An Asynchronous Programming Language and Computing Mahine"*, Technical Report, University of California at Irvine, Deptt. of Computer Science, Dec 1978

[Arvind '83] Arvind and Brock J. D. *"Resource Managers in Functional Programming"*, Manuscript version of the paper that appears in the Journal of Parallel and Distributed Computing.

[Atallah '85] Atallah M. J. and Kosaraju S. R. *"A Generalized Dictionary Machines for VLSI"*, IEEE Transactions on Computers, Vol C-34, No 2, Feb '85 pp 151-155

[Bancilhon '85] Bancilhon F., Kim W. and Korth H. F. *"A model for CAD transactions"*, TR 85-06, April '85, UT Austin.

[Bayer '77] Bayer R. and Schkolnick M. *"Concurrency of Operations on B-Trees"* Acta Informatica, Vol 9, 1977, pp 1-21

[Berger '85] Berger M. J. and Bokhari S. H., *"A Partitioning Strategy for PDEs across Multiprocessors"*, ICPP '85, pp 166-170.

[Biswas '86] Biswas J. and Matula D. W. *"Two Flow Routing Algorithms for the Maximum Concurrent Flow Problem"*, Proceedings of the 1986 Fall Joint Computer Conference, Dallas, Nov 2 - 6, 1986, pp 629 - 636.

[Biswas '87a] Biswas J., Leonard L. and Horton W. *"Parallel processing particle trajectory code for anomalous transport studies"* Technical Report IFSR#265, Institute of Fusion Studies, UT Austin

[Biswas '87b] Biswas, J. and Browne, J. C. *"Simultaneous Update of Priority Structures"* Proceedings of the 1987 ICPP.

[Browne '84] Browne J. C., Dutton J. E., Fernandes V. and Palmer A. *"Zeus: An Object-Oriented Distributed Operating System for Reliable Applications"*, ACM National Conference, 1984.

[Bitner '85] Bitner, J. *"Course notes in CS385, Theory of Algorithms."* University of Texas at Austin, Spring '85.

[Bottos '85] Bottos B. *"The Use of Shared Data in Parallel Programs"*, Thesis proposal: Rephrased and Abridged, 18th Nov, 1985.

[Butterfly '87] Bolt Beranek and Newman Inc. Product literature on the Butterfly Parallel Processor. Available from BBN Laboratories Ltd. Boston Massachussetts.

[Campbell '74] Campbell R. H., and Habermann A. N. *"The specification of process synchronization by path expressions"*, Lecture Notes in Computer Science, 16. 1974, pp 89-102.

[Campbell '77] Campbell R. H. *"Path expressions: A technique for specifying process synchronization"*, Dept. Comp. Sci., Univ Illinois at Urbana-Champaign, Report UIUCDCS-R-77-863, May 1977.

[Carey '84] Carey M.J. and Thompson C. D. *"An Efficient Implementation of Search Trees on* log *N + 1 Processors"*, IEEE Transactions on Computers, Vol C-33, No 11, Nov '84 pp 1038-1041

[Dally '86] Dally W. J., *"A VLSI architecture for Concurrent Data Structures"*, Ph.D. Thesis, California Institute of Technology, March 3rd 1986.

[Darema '85] Darema-Rogers F., Norton V. A. and Pfister G. F., *"Using a Single-Program-Multiple-Data Computational Model for Parallel Execution of Scientific Applications"* Technical Report RC 11552 (#51726) 11/19/85, Computer Sciences Department, IBM T. J. Watson Research Center. Yorktown Heights, NY.

[Edler '85] Edler J., Gottlieb A., and Lipkis J. *"Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and* IBM RP3." Technical Report - Ultracomputer Note #91, Courant Institute of Mathematical Sciences., December '85.

[Ellis '80a] Ellis C. A. *"Concurrent Search and Insertion in 2-3 Trees."* Acta Informatica, Vol 14, pp 63 - 86, 1980.

[Ellis '80b] Ellis C. A. *"Concurrent Search and Insertion in AVL Trees."* IEEE Transactions on Computers, Vol C-29, No 9, Sept 1980, pp 811-817.

[Ellis '85] Ellis C.S., *"Distributed Data Structures, A Case Study"*, IEEE Transactions on Computers, December 1985, Vol C-34, No: 12, pp 1178-1185.

[Easwaran '76] Easwaran K. P., Gray J. N., Lorie R. A. and Traiger I. L. *"The notions of consistency and predicate locks in a database system"*, CACM, Vol 19 No 11, Nov '76 pp 624-633

[Fisher '84] Fisher A. L. *"Dictionary Machines with a Small Number of Processors"*, Proceedings of the 11th Annual International Symposium on Computer Architecture, June '84, pp 151-156.

[Flynn '66] Flynn, M. J. *"Very High-Speed Computing Systems"*, Proc. IEEE Vol 54, 1966 pp 1901-1909.

[Gajski '83] Gajski D., Kuck D., Lawrie D. and Sameh A. *"Construction of a Large Scale Multiprocessor"* Technical report # UIUCDCS-R-83-1123 (UILU-ENG 83 1704) Cedar Document No. 5 Computer Science Department, University of Illinois, Feb' 1983

[Gelernter '86] Gelernter D., and Carriero N. *"Distributed data structures in Linda"* Proceedings of the 1986 ACM Symposium on the Principles of Programming Languages, (POPL '86), pp 255-263, Jan 1986.

[Gottlieb '83a] Gottlieb A., Lubachevsky B. D. and Rudolph L., *"Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors."* ACM Transactions on Programming Languages and Systems, Vol 5., No 2., April 1983, Pages 164-189.

[Gottlieb '83b] Gottlieb A. et al *"The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer."* IEEE Transactions on Computers, Vol C32., No 2., Feb 1983.

[Greif '86] Greif I., Seliger R. and Weihl W. "Atomic Data Abstractions in a Distributed Collaborative Editing Environment", 13th ACM Conference on the Principles of Programming Languages (POPL), Jan '86 pp 150-160

[Guibas '82] Guibas L. J. and Liang F. M. *"Systolic Stacks, Queues and Counters"*, Proceedings of th 1982 Conference on Advanced Research in VLSI, MIT, Jan '27 '82

[Guttag '80] Guttag J. V., *"Notes on Type Abstraction (version 2)"*, IEEE Transactions on Software Engineering, Jan 1980, pp 13-23.

[Habermann '75] Habermann, A. N. *"Path expressions"* Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, June 1975.

[Headington '85] Headington, M. R. and Oldehoeft, A. E. *"Open Predicate Path Expressions and their implementation in highly parallel computing environments"* Proceedings of the ICPP '85 pp 239-246.

[Herlihy '87] Herlihy M. P. and Wing J. M. *"Axioms for Concurrent Objects"*, Proceedings of the Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), Jan 1987.

[Hoare '74] Hoare C. A. R., *"Monitors, An operating system structuring concept"*, Vol 17, No 10, Oct 1974, CACM 1974, 549-557.

[Intel '87] Intel Scientific Computers, Product literature on iPSC computers. Available from Intel Scientific Computers, 15201 NW Greenbrier Parkway, Beaverton, OR 97006

[Jayaraman '81] Jayaraman B. and Keller R. M. *"Resource expressions for applicative languages"* Manuscript, April 3 1981, Department of Computer Science, University of Utah.

[Kahn '74] Kahn G. *"The semantics of a simple language for parallel computing"*, In J. N. Rosenfeld (editor), *Information Processing 74, Proceedings of the IFIP Congress '74*, pp. 471-475 August 1974.

[Karp '69] Karp R. M. and Miller R. E. *"Parallel Program Schemata"*, Journal of Computer and System Sciences: Vol 3, pp 147-195 (1969)

[Karp '87] Karp A. H. *"Programming for Parallelism"*, IEEE Computer, May '87, pp 43 - 57.

[Korth '83] Korth H. F. *"Locking Primitives in a Database System"*, JACM, Vol 30 No 1, Jan '83 pp 55-79

[Korth '85] Korth H. F. and Kim W. *"A concurrency control scheme for CAD transactions"*, TR-85-34, UT Austin, Dec '85.

[Kuck '81] Kuck D. J., Kuhn R. H., Padua D. A., Leasure B. and Wolfe M. *"Dependence graphs and compiler optimizations"*, Proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL) Williamsburg, VA., Jan 1981, pp 207-218.

[Kung '80a] Kung H. T. *"The Structure of Parallel Algorithms"*, in David K Hsiao ed. Advances in Computers, Vol 19. Copyright ACM Press, 1980 pp 65-112

[Kung '80b] Kung H. T., and Lehman P. L. *"Concurrent Manipulation of Binary Search Trees."* ACM TODS, Vol 5, No 3., pp 339-353, 1980.

[Kung '83] Kung H. T. and Papadimitriou C. *"An Optimality Theory of Database*

Concurrency Control" Acta Informatica, Vol 19 No 1, 1983, pp 1-11.

[Lanin '86] Lanin V. and Shasha D. *"A Symmetric Concurrent B-Tree Algorithm."* Proceedings, Fall Joint Computer Conference '86, Nov. '86.

[Leiserson '79] Leiserson C. E. *"Systoloc Priority Queues",* Proceedings of the Caltech Conference on VLSI, January 1979.

[Manber '83] Manber U. and Ladner R. E. *"Concurrency Control in a Dynamic Search Structure",* ACM TODS Vol 9, No 3. Sept 1984, pp 439-455.

[Matelan '85] Matelan N. *"The architecture and implementation of the FLEX/32 MultiComputer",* Proceedings of the National Computer Conference, 1985.

[McGraw '85] McGraw J., et al *"SISAL - Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2",* March 1, 1985, Lawrence Livermore National Laboratories.

[Nievergelt '83] Nievergelt J. and Hinterberger H., *An adaptable, symmetric, multikey file structure"* ACM Transactions on Database Systems, Vol 9, No. 1, March '84 pp 38-71.

[Oldehoeft '84] Oldehoeft A. E., and Jennings S. F. *"Dataflow resource managers and their synthesis from open path expressions"* IEEE Transactions on Software Engineering, Vol. 10, No. 3, pp. 244-256, May 1984.

[Ottomann '82] Ottomann T. A., Rosenberg A. L. and Stockmeyer L. J. *"A Dictionary Machine (for VLSI)",* IEEE Transactions on Computers, Vol C-31 No 9, Sept '82 pp 892-897

[Ousterhout '82] Ousterhout J. K. *"Scheduling techniques for concurrent systems."* Proceedings of the Third International Conference on Distributed Systems, Oct 18-22, 1982, pp 22-30.

[Papa '86] Papadimitriou, C. *"The theory of Database Concurrency Control"* Computer Science Press, 1986.

[Pfister '85] Pfister G. F. et al. *"The IBM Research Parallel Processor Prototype*

(RP3); Introduction and Architecture." Proceedings of the 1985 ICPP, August '85, pp 764-771

[Premkumar '81] Premkumar U. *"A Theoretical Basis for the Analysis and Partitioning of Regular SW Banyans"*, Ph.D. dissertation, 1981, UT Austin.

[Quinn '84] Quinn M.J. and Yoo Y.B. *"Data Structures for the Efficient Solution of Graph Theoretic Problems on Tightly-coupled MIMD Computers"* Proceedings of the ICPP '84 pp 431-438.

[Rao '87a] Rao V. N., Kumar V. and Ramesh K. *"Parallel Heuristic Search on a Shared Memory Multiprocessor"* AI TR87-45 Jan '87, Artificial Intelligence Laboratory, UT Austin.

[Rao '87b] Rao V. N. *Personal discussions, UT Austin.*

[Rashid '86] Rashid R. F. *"From RIG to Accend to Mach: The Evolution of A Network Operating System"*, Proceedings of the 1986 Fall Joint Computer Conference, Dallas, Nov 2 - 6, 1986, pp 1128 - 1137.

[Schmeck '85] Schmeck H. and Schroder H. *"Dictionary Machines for Different Models of VLSI"*, IEEE Transactions on Computers, Vol C-34, No 5, May '85 pp 472-475

[Schwartz '87] Schwartz A. M. and Loui M. C. *"Dictionary Machines on Cube-Class Networks"*, IEEE Transactions on Computers, Vol C-36, No 1, Jan '87 pp 100-105

[Schwarz '83] Schwarz, P. and Spector A., *"Synchronizing Shared Abstract Types"*, ACM Transactions on Computer Systems, August 1984, pp. 223-250

[Seitz '85] Seitz C. L. *"The cosmic cube"*, CACM Vol 28, No 1, pp 22-33, Jan '85.

[Shasha '87] Shasha, D. and Goodman, N. *"Concurrent Search Structure Algorithms"*, ACM Transactions on Database Systems (to appear), 1987.

[Silberschatz '80] Silberschatz A. and Kedem Z. *"Consistency in Hierarchical Database Systems"*, JACM Vol 27 No 1, Jan '80 pp 72-80

[Somani '85] Somani A. K. and Agarwal V. K. *"An Efficient Unsorted VLSI Dictionary Machine"*, IEEE Transactions on Computers, Vol C-34 No 9, Sept '85 pp 841-852

[Suhler '87] Suhler P. and Biswas J., *"The Task-Level Data Flow Language (TDFL)"*, Working paper. Oct '87.

[Suhler '87] Suhler P. *"Performance tuning of Task-Level Data Flow Programs"*, Dissertation in preparation, Dept. of Electrical Engineering, UT Austin, Oct '87.

[Swan '77] Swan, R. J. et al. *"Cm* - A Modular Multi-Microprocessor"*, Proceedings of the National Computer Conference, Dallas, Tx. June 1977.

[Varman '83] Varman P. J. and Fussell D. S. *"Robust (VLSI) Data Structures"*, Proceedings of the Seventeenth Annual Conference on Information Sciences and Systems, John Hopkins University, Baltimore, March 23 - 25 '83.

[Weihl '84] Weihl, W. *"Specification and Implementation of Atomic Data Types"*, Ph. D. Thesis, MIT/LCS/TR-314, Lab for Computer Science, MIT, March 1984.

[Weng '75] Weng, K.-S. *"Stream oriented computation in recursive data flow schemes"*, Technical Report, MIT/LCS/TM-68, Lab for Computer Science, MIT, Oct 1975.

[Wulf '72] Wulf W. A, and Bell C. G. *"C.mmp - A Multi Mini Processor"*, Proceedings of the AFIPS Fall Joint Computer Conference, Montvale, NJ 1972.

[Yew '87] Yew P-C., Tzeng N.-F. and Lawrie D. H. *"Distributing Hot-Spot Addressing in Large-Scale Multiprocessors."*, IEEE Transactions on Computers, Vol. C-36, No. 4, pp. 388-395, April 1987.

Glossary of Abbreviations

Abbreviation	Expansion	Page no
abx	all blue except	80
adt	abstract data type	7
CHEAP	concurrent heap	74
DC	delete consistency	79
epe	extended path expression	47
gh	generalized heap	79
gib	greater if blue	79
iso	index server object	29
HC	heap consistency	79
oppe	open predicate path expression	63
ois	object implementation state	18
PDC	parallel delete consistency	97
PHC	parallel heap consistency	97
PQ	priority queue	76
PPLM	processor per level with multiple input ports	140
PPN	processor per node	135
RHEAP	regular heap	119
SBAN	software banyan	99
SDC	serial delete consistency	80
SHC	serial heap consistency	80
SPMD	single program multiple data	51
TDFL	task-level data flow language	36
wco	work container object	41
wman	work manager	50
WQ	work queue	83

Appendices

Appendix 1. BNF grammar of Extended Path Expressions (continued)

<predicate> ::= <bool-term> | <bool-term> <boolean-op> <bool-term>
<bool-term> ::= <event-count-term> | ¬ <event-count-term>
<event-count-term> ::= <term> | <term> <arith-op> <event-count-term>
<term> ::= <unsigned-integer> | <type-of-count> (<operation-id>)
<type-of-count> ::= **invok** | **perm** | **ack** | **rep**
<range> ::= * | (<int-expr> , <int-expr>)
<set> ::= { <set-of-identifiers> }
<int-expr> ::= <int-term> | <int-term> <arith-op> <int-term>
<int-term> ::= <unsigned-integer> | <int-var>
<int-var> ::= <target-lang-id>
<set-of-identifiers> ::= <target-lang-id> | <target-lang-id> , <set-of-identifiers>
<targ-lang-bool-exp> ::= **nil** | <targ-lang-bool> | <targ-lang-bool> <boolean-op> <targ-lang-bool>
<targ-lang-bool> ::= <relational-exp> | ¬ <relational-exp>
<relational-exp> ::= <integer-expr> <rel-op> <integer-expr>
<integer-expr> ::= <integer-term> | <integer-term> <arith-op> <integer-term>
<integer-term> ::= <unsigned-integer> | <int-var> | <subscripted-identifier>
<subscripted-identifier> ::= <target-lang-id> [<int-expr>]
<arith-op> ::= + | - | * | / | **
<boolean-op> ::= ∧ | ∨
<rel-op> ::= < | > | = | <= | >= | <>

The above BNF is a continuation of the syntax of *epe* s presented in section 3.1 of chapter 4. Certain obvious definitions, such as <unsigned-integer> have been omitted. This BNF provides for a bare minimum, and could be extended for programmer convenience by adding the ability to parenthesize sub-expressions, and by adding a wide variety of data types.

Appendix 2. Syntax of Work Managers.

Assumption about parameters. In our development, we have used only one type of parameters, namely integers. On a real implementation, a richer variety of types would no doubt be an attractive feature, but having additional types would not contribute materially to the concepts being discussed here. Hence we use only integer parameters in our work managers.

<work_manager> ::= **@wm_begin** <body> **@wm_end**

<body> ::= <header> <formal_parms> <pexp> <operations> <create_time_bindings>

<header> ::= **@name** <identifier>

<formal_parms> ::= **@formals** | **@formals** <formal_list>

<formal_list> ::= <identifier> | <identifier> , <formal_list>

<pexp> ::= **@epe** <*epe*[†] >

<operations> ::= **@operations** <user_operations> <other_operations>

<create_time_bindings> ::= **@where begin** <create_time_assignment> **end**

<create_time_assignment> ::= <unit> | <unit> ; <create_time_assignment>

<unit> ::= <ident> := <int_expr> | <identifier> := <range>

<range> ::= <int-expr> . . <int-expr>

[†] The BNF for <*epe* > has been defined in the text.

Appendix 3. Grammar for the generation of left-set and right-set of internal nodes of a parse tree.

Intuitively, the left-set and the right-set of a node in a parse tree of an *oppe* are sets of operations that are allowed to execute first and last respectively, by the subpath represented by the section of the tree rooted at the node.

More formally, given a parse tree T, the left-set and right-set of a node are recursively defined by the following grammar:

Let *list*, *seq* and *item* denote a node that is a respectively a list, sequence and item node. Furthermore, let a subscript of O, L and R denote an only internal child, a left internal child and a right internal child node respectively. (Note that the *epe* grammar does not allow any node with more that two non-terminal children.)

left-set ($<list>$) ::= left-set ($<seq_O>$) | left-set ($<seq_L>$) $\cup$ left-set ($<list_R>$)

right-set ($<list>$) ::= right-set ($<seq_O>$) | right-set ($<seq_L>$) $\cup$ right-set ($<list_R>$)

left-set ($<seq>$) ::= left-set ($<item_O>$) | right-set ($<item_L>$)

right-set ($<seq>$) ::= right-set ($<item_O>$) | left-set ($<seq_R>$)

left-set ($<term>$) ::= left-set ($<list_O>$) | left-set ($<op_O>$) | { **sb** }

right-set ($<term>$) ::= right-set ($<list_O>$) | right-set ($<op_O>$) | { **eb** }

The special symbols **sb** and **eb** need some explanation. A burst is recognizable by the system and is converted into two special signals, **start-burst,** abbreviated **sb** and **end-burst,** abbreviated **eb.** If a term expands into a burst, then the corresponding controller must recognize the start-burst and end-burst signals.

Appendix 4 - Hand executed output of Algorithm 5.2

The generated source code shown below is an execution of the algorithm 4.2, that translates open predicate path expressions. The expression in this case is as follows: 1 : ({ R } [invok (W) - perm(W) = 0] , W).

```
procedure R
begin
   predicate_6_1 (invok);
   burst_9_1 (invok);

   (* access (i.e. read) the data *)

   burst_9_1 (ack);
end;

procedure W
begin
   predicate_6_1 (invok(W));
   restrict_3_1 (invok);
   predicate_6_1 (perm(W));

   (* access (i.e. write ) the data *)

   ack++
   restrict_3_1 (ack);
end;

procedure burst_9_1 (signal)
begin
if signal.type = invok then
   begin
      if burst_count_9_1 = 0 then      (* start of new burst      *)
         begin
            restrict_3_1 (invok);
            burst_count_9_1 := 1;
      end
    else                  (* continuation of old burst *)
      burst_count_9_1 := burst_count_9_1 + 1;
  end;
 if signal.type = ack then
  begin
    if burst_count_9_1 = 1 then      (* end of a burst            *)
      begin
        restrict_3_1 (ack);
        burst_count_9_1 := 0;
      end
    else                  (* continuation of a burst  *)
      burst_count_9_1 := burst_count_9_1 - 1;
  end;
end;

procedure predicate_6_1 (signal)
begin
 if (signal = invok(W)) or (signal = perm(W)) then
  begin
    if signal = invok(W) then invok(W) := invok(W) + 1;
    if signal = perm(W) then perm(W) := perm(W) + 1;
    if ( invok(W) - perm(W) = 0 ) then
     repeat
        dequeue_6_1 (id);
        if id <> nil then
          awaken (id)
     until (id = nil)
  end;
if signal = R then
  begin
    if not ( invok(W) - perm(W) = 0 ) then
```

```
          begin
             enqueue_6_1 (myid);
             sleep (myid)
          end
      end;
   end;

   procedure restrict_3_1 (signal)
   begin
    if signal.type = ack then
      begin
         nacks_3_1 := nacks_3_1 + 1;
         if ninvoks_3_1 - nacks_3_1 >= 1 then
           begin
              dequeue_3_1 (id);
              awaken (id)
           end;
      end;
    if signal.type = invok then
      begin
         ninvoks_3_1 := ninvoks_3_1 + 1;
         if ninvoks_3_1 - nacks_3_1 > 1 then
           begin
              enqueue_3_1 (myid);
              sleep (myid)
           end;
      end;
   end;
```

Each read invocation first goes to the predicate controller, and proceeds to the burst controller only after being cleared. Once the predicate becomes true, all the operations (reads) waiting at the controller are released. Thus predicate controllers are typically used to specify some kind of priority between operations. An important point to note about a predicate controller, is that it enforces predicates involving non-local operations, i.e. the signals enabling the predicates are not generated from within the subtree, but are from the external environment. This is an unpleasant feature, and causes problems in implementation. Since it is not known what will happen if the predicate is violated, every operation in the subpath must go through this controller, and potentially be blocked in the predicate controller until the predicate is true.

Appendix 5

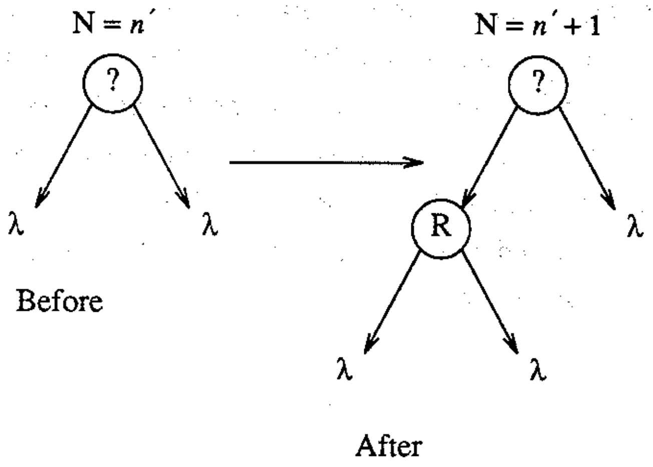
Proof of *PHC′*

We shall examine each state transition upon the heap. Since locks guarantee non-interference, every state transition may be thought of as taking place at the instant an activity releases its last lock. Thus we have to show that every state transition preserves *PHC′*. The proof is by exhaustive enumeration of the state transitions.

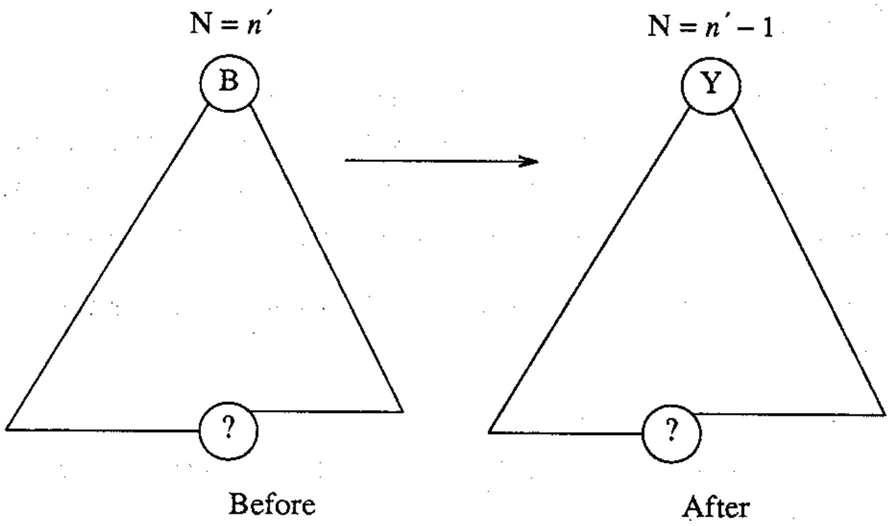
State transitions. The shared data structure is mutated by the following kinds of state transitions:

1. The color of a node is changed.
2. The contents of a parent and child nodes including their colors, are swapped.
3. An item is added.
4. An item is deleted.

Swapping can be looked at as two color changes, accompanied by a transposition of values. Insertion is the birth of a new node at the lowest level. Since *PHC′* is only concerned with descendants of a red node, insertion trivially preserves *PHC′* (see picture below).



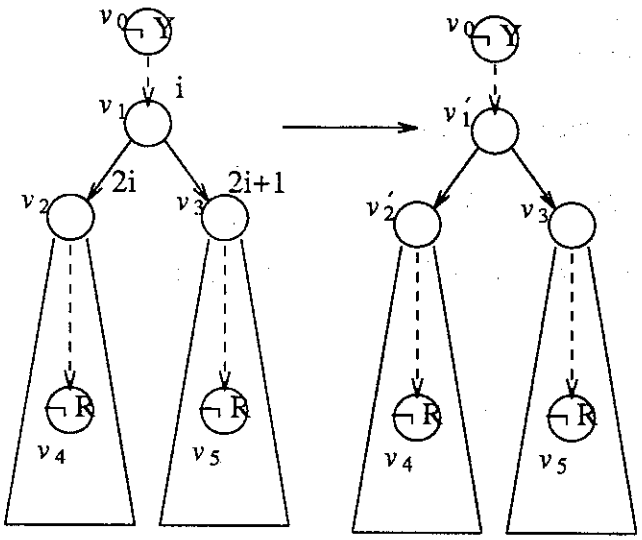
Deletion is the removal of a node from the lowest level in the heap, accompanied by a color change of the root node to yellow. Since the root node has no ancestors and *PHC′* is only concerned with ancestors of a yellow node, deletion trivially preserves *PHC′* (see picture below).



We shall therefore concentrate only on state transitions caused by restructuring operations.

State transitions caused by restructuring operations.

As shown in Fig 4.3, there are 16 cases, numbered 1 through 16. The first six are due to restructuring workpieces of the form (i, "red"). The remaining ten are for delete operations, i.e. restructuring workpieces of the form (i, "yellow"). We use the following notation (see picture below). v_0 denotes the value of a non yellow ancestor. v_1, v_2, and v_3 are the values in the parent (index i), left child (2i) and right child (2i+1) of a restructuring window, prior to a state transition. v_1', v_2', and v_3' denote the values in the same nodes immediately after the transition. v_4 and v_5 denote values at non-red nodes in the subtree rooted at the left and right children of the parent node, respectively.



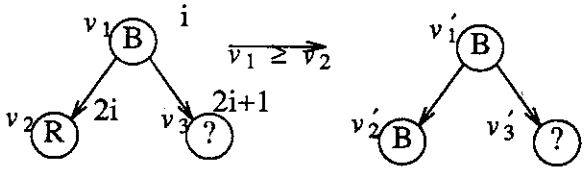
Before transition After transition

If v_0, v_4 or v_5 do not exist, *PHC'* is trivially true at the nodes where we make the assertion.

Let *NoChange* $\equiv v_1' = v_1 \wedge v_2' = v_2 \wedge v_3' = v_3$

Let *Swap* $\equiv v_1' = v_2 \wedge v_2' = v_1 \wedge v_3' = v_3$

Case 1:



There is one color change, namely that at node 2i. We must examine the ancestors and descendants of this node, to see if *PHC'* has been invalidated.

Ancestors of 2i

$v_0 \geq v_1 \wedge v_1 \geq v_2 \wedge \mathit{NoChange} \therefore v_0 \geq v_2'$
also, $v_1 \geq v_2 \wedge \mathit{NoChange} \therefore v_1' \geq v_2'$

Thus PHC' is preserved at all ancestors of the new blue node (2i).

Descendants of 2i

$v_2 \geq v_4 \wedge NoChange \therefore v_2' \geq v_4$

Thus PHC' is preserved at all descendants of the new blue node.

The remaining cases (cases 2 - 16) are similar and the proofs are left to the reader.
□

Proof of EP (Eventual Progress):

Progress is said to be made whenever either a node changes color to blue, or a parent and child nodes are swapped. In most cases, progress is made by a red / yellow value with each restructuring operation and inserts and deletes. Cases 3, 6, 9, 13 and 15 (see Fig 4.3) are the only cases where such progress is not possible. We argue that such lack of progress cannot continue indefinitely, and eventually the structure will enter one of the states from which progress is possible.

The five cases identified may be summarized into three categories: a) when a red parent and child are properly ordered, b) when a yellow parent and child are properly ordered and c) when a red child is greater than a yellow parent, but has a yellow sibling.

Consider a yellow value that is unable to make progress. By b), it must have a yellow child that is in the same situation. Since the tree is not infinite, there must therefore exist a cycle of ordered yellow values. This is impossible in a tree structure. Thus a yellow value will eventually be able to make progress.

Now consider a red value that is unable to make progress. By a) and c), this means that it must have either i) a red parent that is greater, and is similarly stagnant; or ii) a yellow stagnant parent that is smaller, and a yellow sibling. i) implies the existence of a cycle of ordered red values, which by our previous argument, is impossible in a tree structure. ii) is not possible, since we have just shown that yellow values are not stagnant. Thus the red value will eventually be able to make progress. □

Appendix 6 - Banyan definitions and formulae

A *banyan* is a Hasse diagram of a partial ordering (i.e. a partial ordering under transitive reduction), in which there is only one path from any *apex* to any *base*. An apex (base) is any vertex with no arcs incident into (out of) it. A vertex that is neither an apex nor a base vertex is called an *intermediate* vertex.

An *L level banyan* is a banyan in which every path from any apex to any base is of length L. An L level banyan is an $(L+1)$ partite graph in which the partitions may be linearly ordered from 0 through L such that the arcs exist only from the i^{th} partition to the $(i+1)^{th}$ partition $(0 \le i \le L)$. The set of vertices in the i^{th} partition are called the vertices in level i.

An *regular (s, f, L) banyan* is an L level banyan in which the in degree of every vertex except the apex vertices is s, and the out degree of every vertex except the base vertices is f. s is called the *spread* and f is called the *fanout* of the banyan.

An (s, f, L) banyan has $s^{L-i} f^{i}$ vertices at level i, $0 \le i \le L$. Thus an L level banyan has s^L apexes and f^L bases. The definitions and properties above are due to [Premkumar '81] except for the fact that we have interchanged the terms base and apex.

If there is an edge between two vertices, u in level i and v in level i+1, we shall say u is a *parent* of v, and v is a *child* of u.

A *regular (s, f, L) SW banyan* is a regular (s, f, L) banyan with the additional properties that a) two vertices at an intermediate level i, have either no or all common parents at level $i-1$, and b) two vertices at intermediate level i have either no or all common children at level $i+1$.

A *rectangular (s, f, L) SW banyan* is a regular (s, f, L) SW banyan with $s = f = d$.

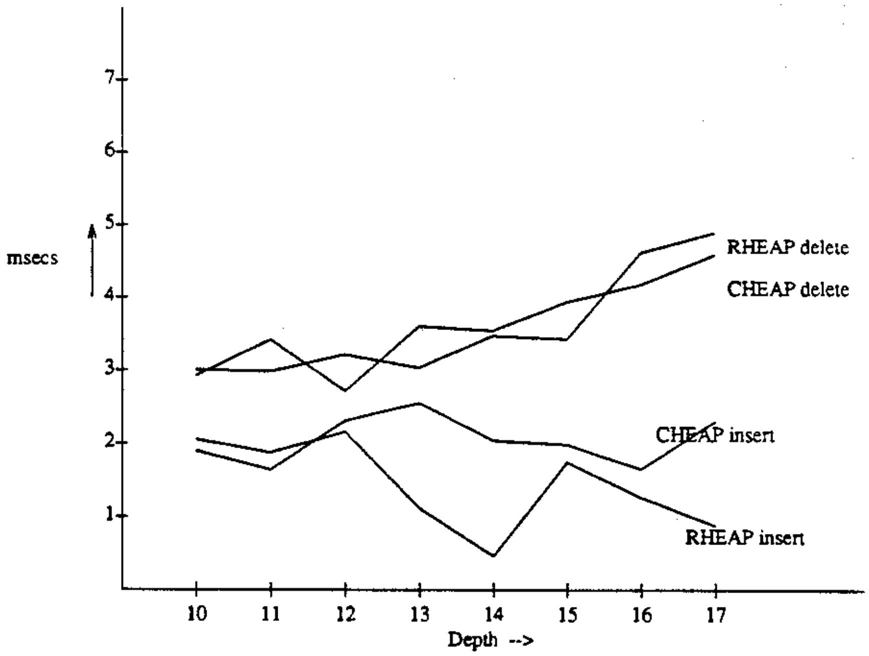
A (d, L) *hypercube* $(d>1, L\ge 0)$, is an undirected graph with d^L nodes and an L digit label in base d assigned to each node, such that an edge exists between two nodes if and only if their labels differ at exactly one digit position. Since there are $d-1$ ways of modifying each digit prosition, and there are L digit positions, each node in a (d, L) hupercube must have exactly $(d-1) * L$ neighbours.

A (d, L) *alternating sequence* S, on L-digit numbers in base d is recursively defined as follows: a $(d, 1)$ alternating sequence is $0, 1, \cdots, d-1$ and $S = 0.S_1, ..., d-1.S_{d-1}$, where S_1 is a $(d, L-1)$ alternating sequence such that $S_{i+1} = S_{i'}$ and $S_{i'}$ is the reflection of S_i. The period denotes concatenation of the digit to its left with each element of the sequence to its right.

Appendix 7 - Performance of CHEAP vs RHEAP

Fig 1.1. Varying size of the structure (the number of nodes in the heap)

		CHEAP		RHEAP	
Depth (*n*)	2^n	Avg time per Insert (μsecs)	Avg time per Delete (μsecs)	Avg time per Insert (μsecs)	Avg time per Delete (μsecs)
17	131072	2308	4595	882	4899
16	65536	1654	4180	1271	4622
15	32768	1987	3940	1748	3429
14	16384	2045	3548	464	3473
13	8192	2555	3608	1101	3040
12	4096	2310	3718	2159	3216
11	2048	1647	3420	1875	2985
10	1024	1905	2929	2062	3008



Appendix 8. Validation of thruput and delay derivations

i) Thruputs.

In chapter 5 section 5.1 the maximum thruput or simultaneity of restructuring operations in a CHEAP was derived to be:

$$T_{max\_rest\_op} = \frac{2^L - 1}{2f_{ir} + 3f_{dr}}$$

where f_{ir} and f_{dr} denote fractions of insert and delete restructuring activities respectively, and L is the depth of the structure. Also, the maximum thruput of operations was derived to be:

$$T_{max\_op} = \frac{2^L - 1}{(f_i\, n_i + f_d\, n_d)\ (2f_i + 3f_d\ )}$$

where f_i and f_d denote fractions of insert and delete operations respectively, and L is as before, the depth of the structure.

The above derivation assumed an unlimited number of processors, so that a processing element could be devoted to each node of the CHEAP. However, in our shared memory implementation of the CHEAP, the maximum number of processors is 10, and some of these 10 processors must be devoted to doing inserts and deletes. We therefore have to adjust the formula for our application.

For a particular size of the structure, i.e. with depth 16, the time spent in doing a restructure is on the average (measured at) 385 μsecs (weighted average of 216.7 μsecs for red values and 344.4 μsecs for yellow values, with an additional procedure call overhead of 70 μsecs). Assuming there are NR restructuring activities. The revised expected maximum operation thruput[1] is:

[1] This is different from T_{max_op} in chapter 5. t_{max_op} is the maximum operation thruput with a limited number of processors, whereas T_{max_op} is the maximum possible simultaneity or overlap among operations, given an unlimited number of processors.

$$t_{max\_op} \;=\; \left[NR \,*\, \frac{10^6}{385} \right]$$

In our experiments NR was 4. This gives t_{max_op} = 10388 restructurings per sec. On an average, red values made 3.3 trips and yellow values made 15.2 trips through the work queue. The total number of restructurings available in 50 secs (the time taken to finish the experiment), was 10388 * 50 = 519400. The total number actually performed was 8000 * 3.3 + 4000 * 15.2 = 87200. This shows that there are far more restructurings available than are actually used, making the restructurers idle for most of the time. This was generally true of large structures with moderate compute time (granularity 10 millisecs).

Upper bound on mean propagation delay.

T_{ub}, the upper bound of the mean time for a red or a yellow value to turn blue, i.e. an upper bound on the mean propagation delay is given by:

$$T_{ub} = L\; W'_{ub}$$

where W'_{ub} is as given in chapter 5 section 5.2.

We now plug in the measured values for various terms in these equations:

$$\lambda = (8000 + 4000) / 50 = 240$$

$$L\,\lambda = 16 * 240 = 3840$$

$$c = NR = 4$$

$$c\,\mu = 4\,(\frac{10^6}{385}) = 10388$$

$$\text{Thus}\quad \rho = (\frac{L\,\lambda}{c\,\mu}) = 0.37$$

$$\text{and}\quad (L\,\frac{\lambda}{\mu}) = 1.48$$

$$W_{ub} = (\frac{1}{\mu}) + (\frac{.2}{.2 + 2.59})\,\frac{315}{2.52} = 324$$

$$T_{ub} = L\;\; W_{ub} = 16 * 324 = 5184$$

As is apparent from the results, the deletes came close to this figure (4312) but the inserts took much less time than this on the average (1304).

Appendix 9 - Markov chains for CHEAP, RHEAP and SBAN algorithms.

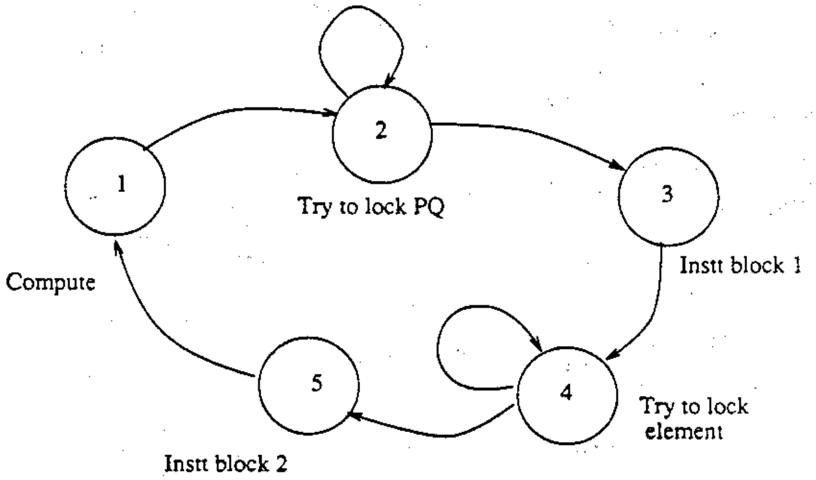


Fig 1.1 Markov chain for inserting activities

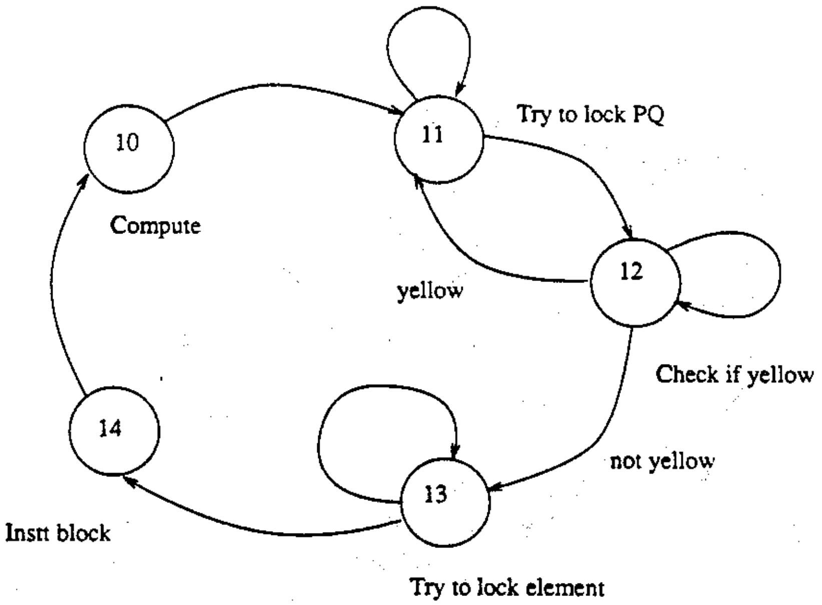


Fig 1.2 Markov chain for delete activities

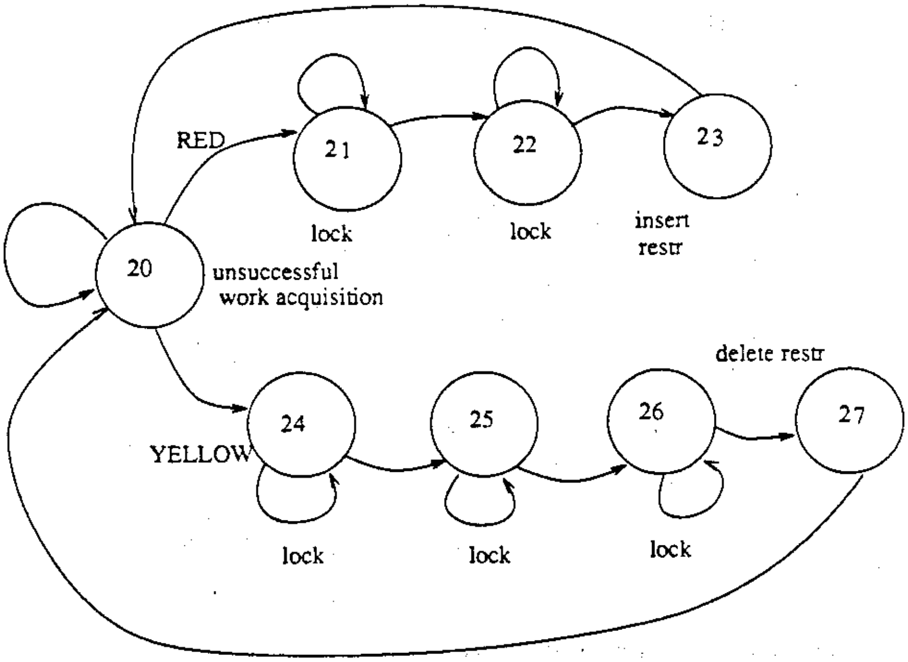


Fig 1.3 Markov Chain for Restructuring activities

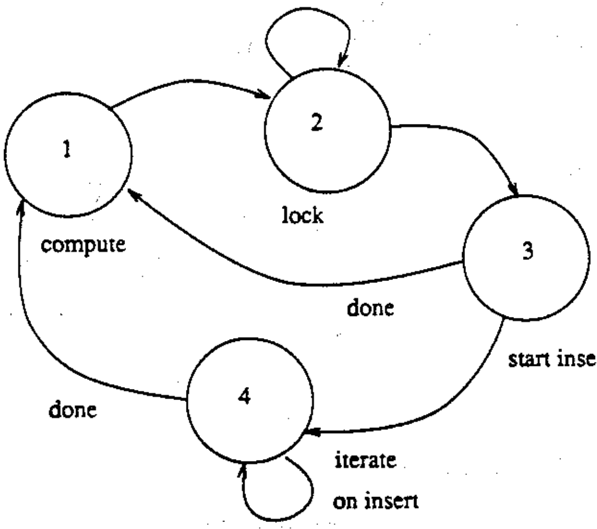


Fig 1.4 RHEAP insert

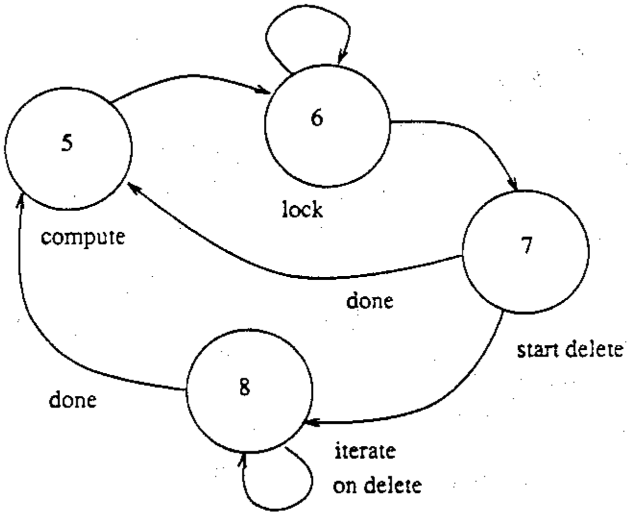


Fig 1.5 RHEAP delete

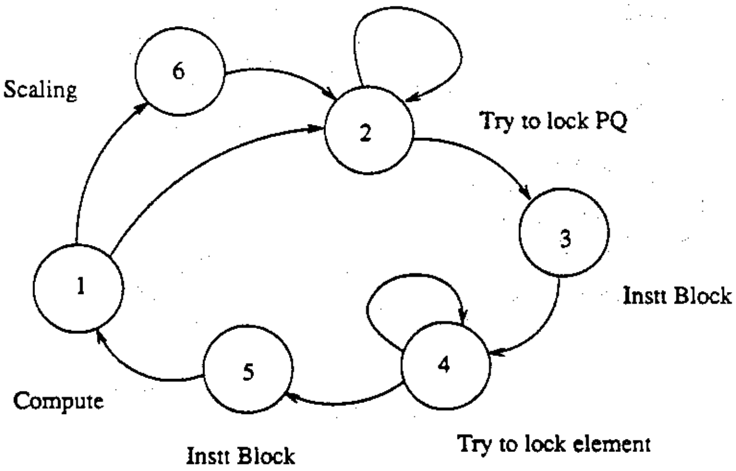


Fig 1.6 Markov chain for Banyan insert

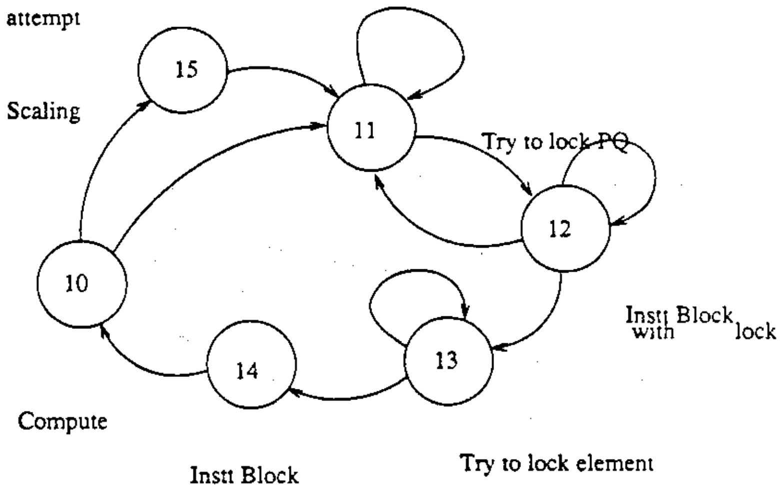


Fig 1.7 Markov chain for Banyan delete

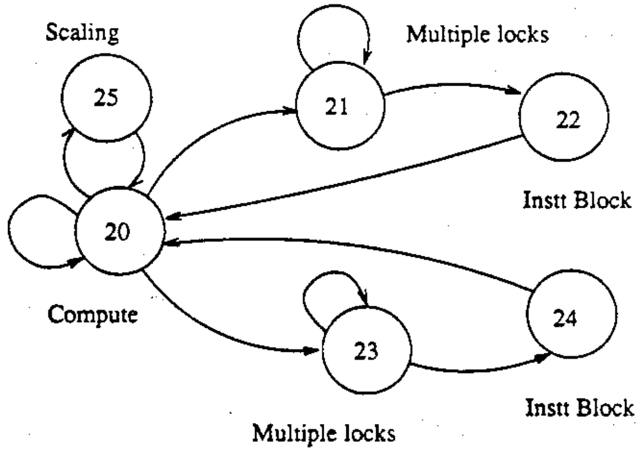


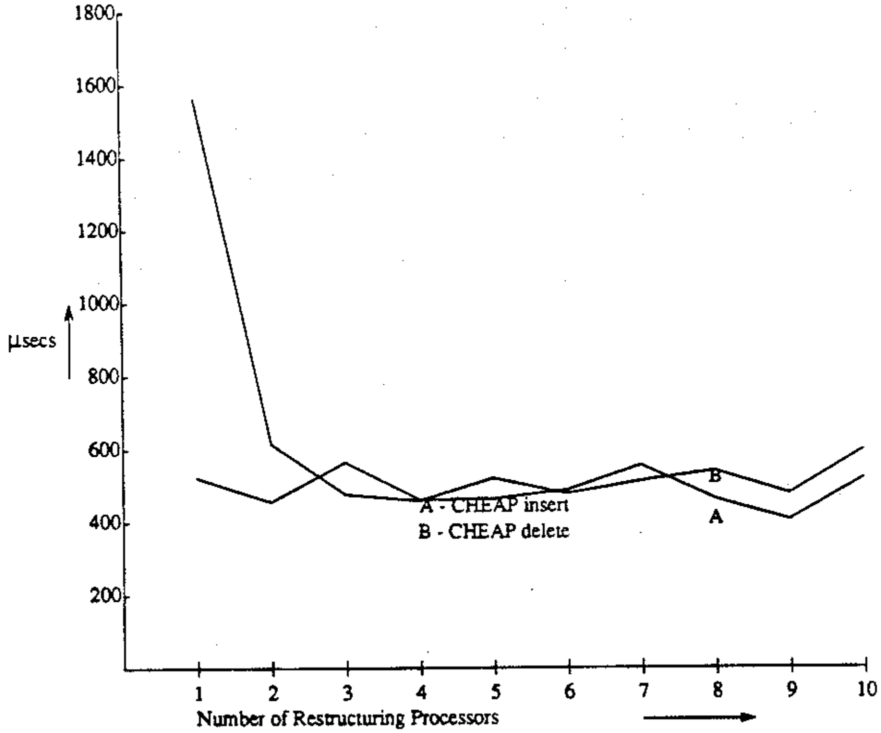
Fig 1.8 Markov chain for banyan restructure

Node	avg time (µsecs)	Node	avg time (µsecs)
1	500000	14	300
2	100	20	200
3	50	21	50
4	50	22	50
5	250	23	200
10	500000	24	50
11	100	25	50
12	50	26	50
13	50	27	200

Fig 2. Typical average times for the markov chain nodes of Fig 1.

Fig 3. Sensitivity to the number of restructuring processors. The number of inserters and deleters are fixed at 20 each.

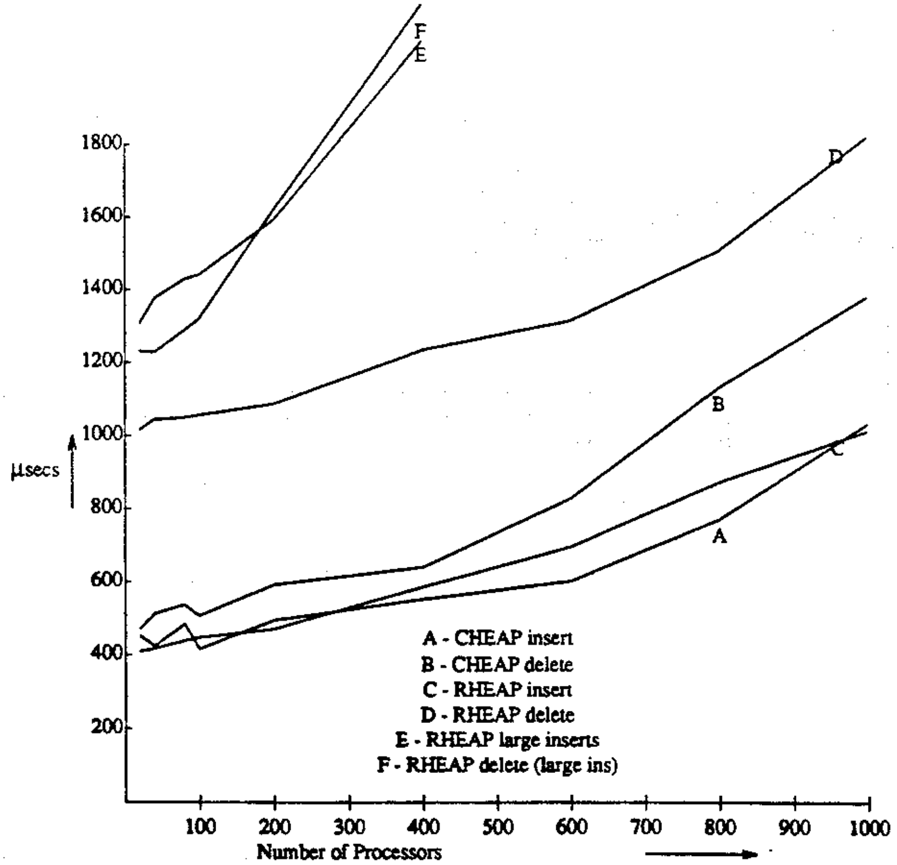
SBAN		
No. of restructuring procs	Avg time per Insert	Avg time per Delete
1	524	1565
2	457	617
3	566	477
4	463	459
5	464	521
6	488	478
7	556	512
8	463	540
9	405	477
10	520	598



Appendix 10 - Simulated performance of CHEAP vs RHEAP

Fig 1.1. Varying the number of processors, keeping granularity fixed at 1 sec and the depth fixed at 12 (4096 nodes).

	CHEAP		RHEAP			
No. of procs	Avg time per Insert	Avg time per Delete	Avg time per Insert	Avg time per Delete	Avg time per Insert (large)	Avg time per Delete (large ins)
20	445	470	409	1017	1307	1233
40	424	513	418	1043	1376	1229
80	484	537	438	1049	1426	1287
100	416	507	447	1056	1439	1319
200	493	591	470	1086	1593	1619
400	543	639	585	1234	2085	2183
600	602	828	695	1315	3802	3611
800	769	1130	871	1505	23644	23229
1000	1030	1378	1011	1817	13175	12052



SBAN		
No. of procs	Avg time per Insert	Avg time per Delete
20	504	564
40	544	405
80	475	497
100	434	599
200	491	543
400	514	413
600	559	798
800	664	1003
1000	1262	935

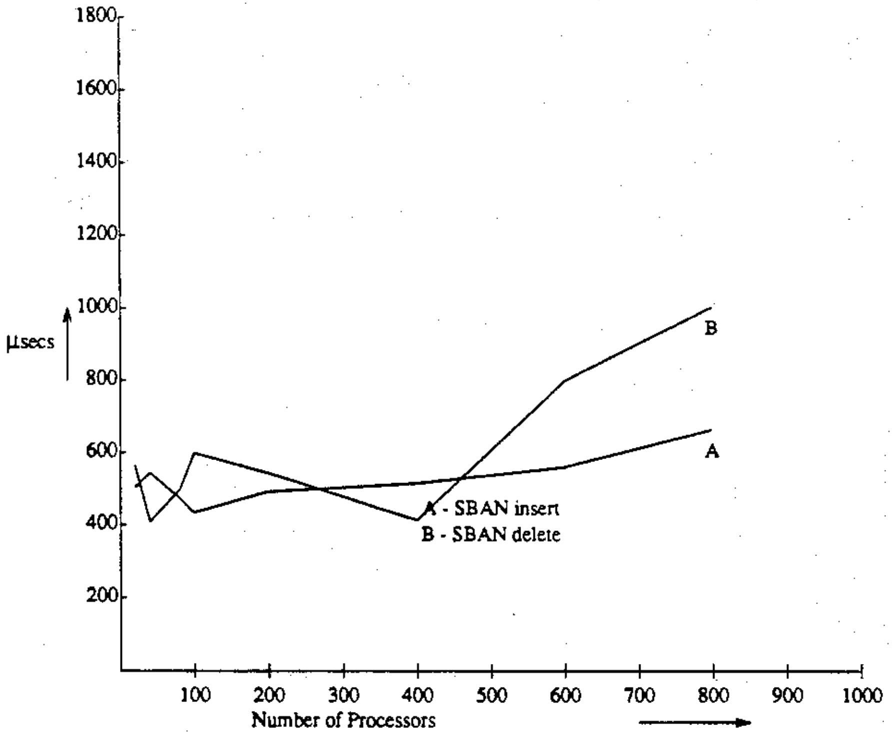
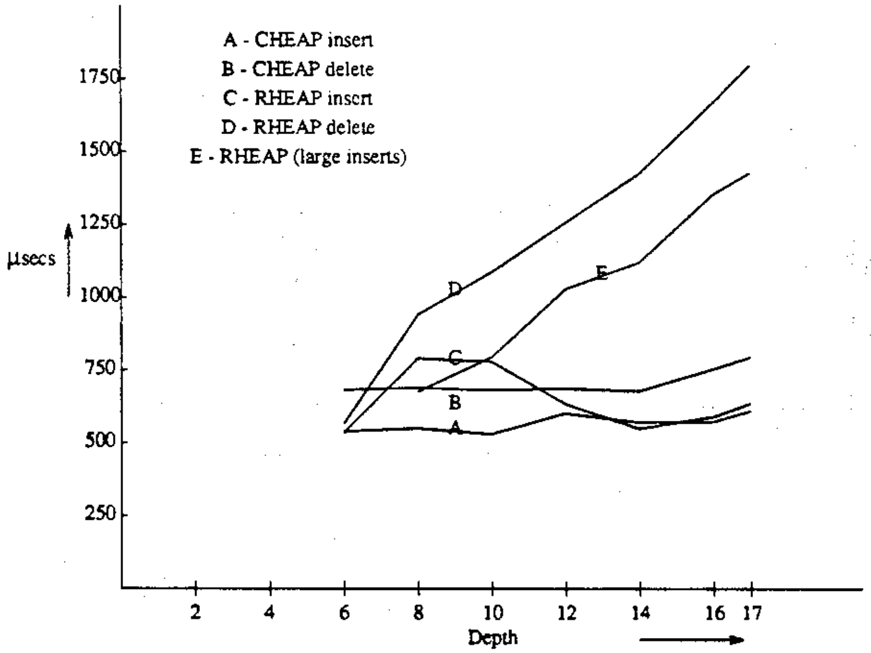


Fig 1.2. Varying size of the structure (the number of nodes in the heap) keeping the number of processors fixed at 500 and 40 (for large inserts).

	CHEAP		RHEAP		
	Avg time per		Avg time per		
Depth (n)	Insert	Delete	Insert	Delete	Large insert
17	539	682	635	1794	1426
16	550	689	589	1666	1352
14	530	681	549	1423	1119
12	601	687	633	1255	1028
10	571	678	779	1086	795
8	573	753	791	941	675
6	609	794	536	566	-



SBAN (spread=2, fanout=3)		
Depth (n)	Avg time per Insert	Avg time per Delete
8	492	602
7	548	509
6	568	682
5	625	666
4	568	693

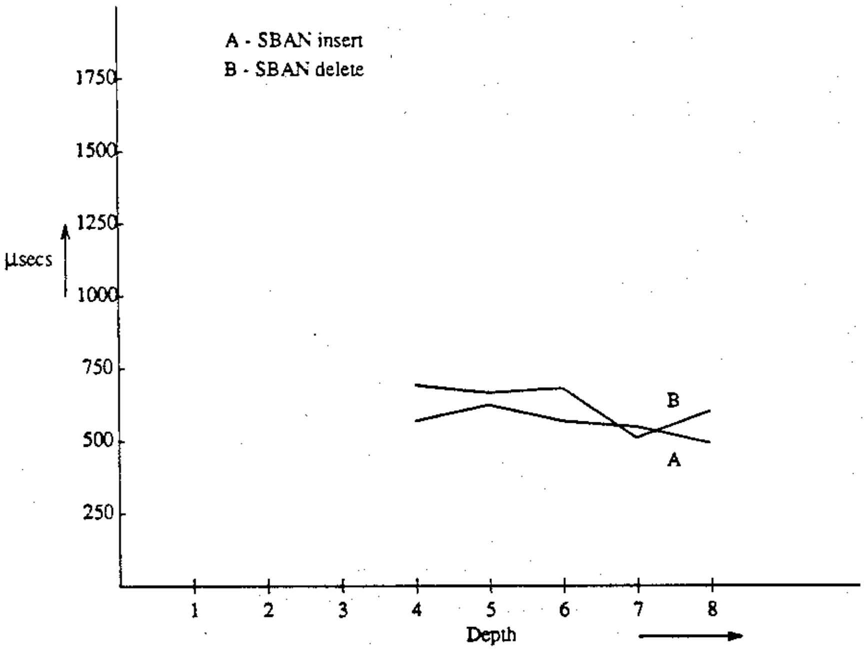
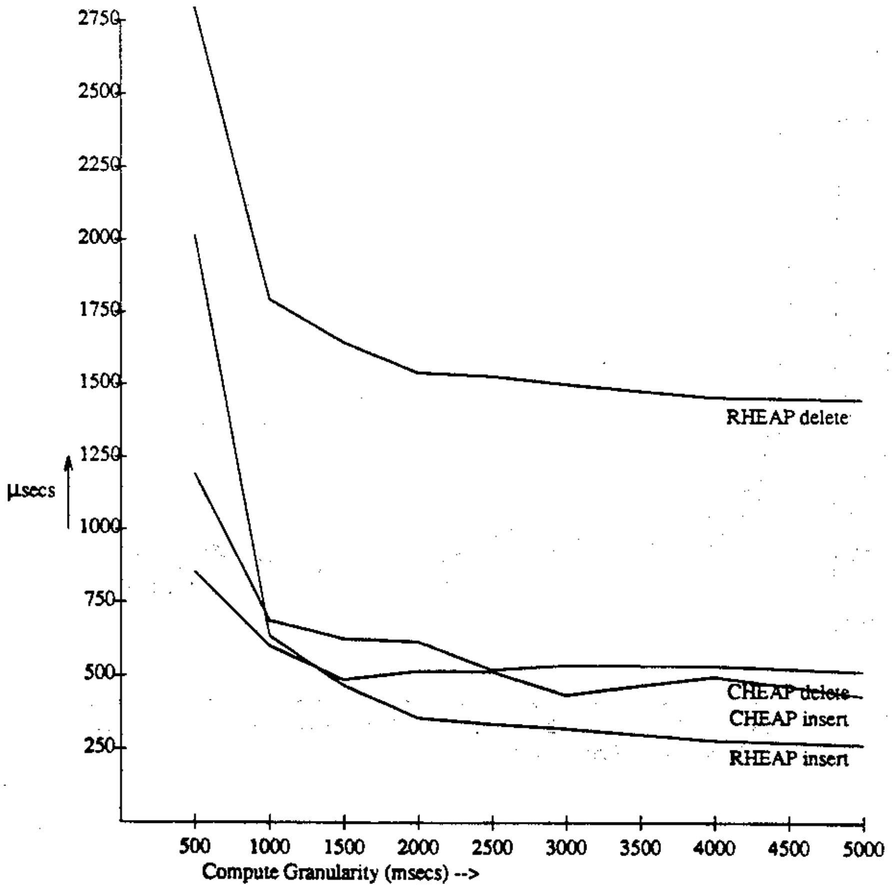


Fig 1.3. Varying the Granularity (compute time), with the depth of the structure being fixed at 17 (131072 nodes). In the case of the 2,3,8 banyan, the number of nodes in the structure is roughly the size of a 2,3,8 banyan, i.e. 19171 nodes.

	CHEAP		RHEAP	
Compute time (msecs)	Avg time per Insert (μsecs)	Avg time per Delete (μsecs)	Avg time per Insert (μsecs)	Avg time per Delete (μsecs)
500	852	1193	2015	2798
1000	601	687	635	1794
1500	485	625	465	1645
2000	515	615	355	1541
2500	514	520	335	1529
3000	434	535	320	1502
4000	496	533	279	1457
5000	431	515	265	1449



SBAN		
Compute time (msecs)	Avg time per Insert (µsecs)	Avg time per Delete (µsecs)
300	1580	1387
500	701	760
1000	454	580
1500	444	500
2000	535	634
2500	409	512
3000	411	407
4000	489	484
5000	532	404

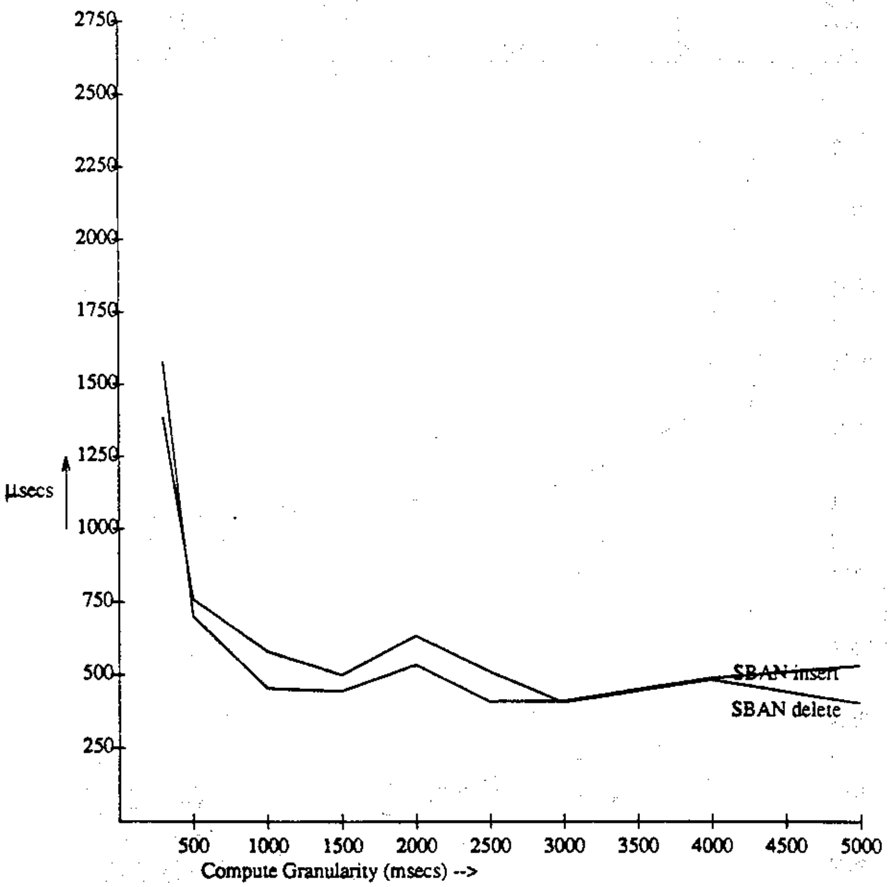


Fig 1.4. Varying the granularity of work done by the restructuring processor, (compare time).

CHEAP			RHEAP	
Compare time (µsecs)	Avg time per Insert (µsecs)	Avg time per Delete (µsecs)	Avg time per Insert (µsecs)	Avg time per Delete (µsecs)
100	561	678	335	1808
200	452	989	742	3684
500	2887	3848	3358	10874
700	13348	10061	9176	18514
1000	8476	15605	26172	40896

Appendix 11

Using figures of forecasted technological parameters from [Fisher '84] it is easy to convince oneself that a PPLM (processor per level with M input ports) machine should be cheaper to construct than a PPN (processor per node) machine, even if we ignore the problem of I/O and assume that the tree algorithm is executed from the root processor for both inserts and deletes (i.e. as in dictionary machines, a single port for all I/O is assumed). The calculations and assumptions are summarized below.

Assumptions common to PPN and PPLM machines:

Maximum size of tree = N = 10^6.

Size of each data item = 4 bytes of priority information + 4 uninterpreted bytes of pointer information = 8 bytes.

PPN machines:

Number of processors per chip = 32.

Number of chips per board = 100.

Therefore number of boards = 10^6 / (100 * 32) = 300 (approximately).

It has been argued [Schmeck '85] that grid implementations of binary trees are able to achieve better packing densities with more uniform delay characteristics, however there is no general consensus on this issue. Note that linear pipelines (eg. the Leiserson priority queue design) are easier to pack on grids but we do not consider this option since it does not allow multiple ports.

Cycle time of PPN = 250 nsecs. (Most of this is board to board signal propagation delay).

Thus PPN machines are made up of about 300 boards and have a cycle time of 250 nsecs (i.e. every 250 nsecs a new operation can begin).

PPLM machines:

Memory cycle time = 500 nsecs.

Number of input ports = M = 8 (for example).

Number of processor boards = log (N + 1) + M - 1 = 27.

Average memory required per board = $10^6 * 2^3 / 20$ = .5 Megabyte (approximately).

Number of memory boards = 10. (We have assumed that smaller memory configurations than those at the bottom half of the tree may be accommodated on the same board as the processor on that level).

Therefore, the total number of boards = 37.

Operation cycle time of PPLM machines = time for 2 memory cycles = 1 µsec.

The cycle time of one microsecond is arrived at on the basis of its very simple rewrite rule. With each cycle a processor has to read and write a memory location. This takes two memory cycles and dominates the operation cycle time. Note that the systolic banyan (see text) is a more complicated structure than the systolic tree and will have a larger cycle time because of additional synchronization and exchange of data values between processors at adjacent levels.

Thus for the particular choice of parameters we have considered, the PPN machine is 4 times faster in terms of cycle time than the PPLM machine, but about 8 times as expensive in terms of the number of boards, for a tree of size one million. Furthermore, the benefit of PPLM machines is more visible as we go to larger sized trees, since the number of processors required grows logarithmically in contrast with linear growth in PPN machines. The cycle time is constant with increase in size until the capacity of the local memory is exceeded. For trees of size in the range 10^6 to 10^8 it appears that PPLM machines will be very favorable with respect to their cost performance ratio. Beyond this range secondary memory is required to support large storage requirements at each processor. This slows down the cycle time. Trees with more than 10^8 values are outside the scope of our analysis.

VITA

Jit Biswas was born in Shillong, India, on June 23 1956, the son of Roma Biswas and Arun Biswas. After completing high school at St. Edmunds' School, Shillong he entered Birla Institute of Technology and Science, Pilani, India where he obtained a Bachelor's degree in Electrical and Electronics Engineering in June 1977. He went on to obtain a professional degree in Industrial Engineering, after which he was employed for two years with TELCO, a large engineering company in India as an engineer. In June 1983 he obtained an MS degree in Computer Science from Southern Methodist University, Dallas, USA.

Permanent address: C/O Mrs R. Biswas
H/O Shri K. K. Mishra
10-C Inder Rd.
Dehra Dun 248001, India

This thesis was typed by Jit Biswas using the troff text processor with pic, tbl and eqn preprocessors (in that order!).