AN ALTERNATIVE METHOD FOR PARALLEL M-WAY TREE SEARCH ON DISTRIBUTED MEMORY ARCHITECTURES

By

TROY H. LARAMY

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1991

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
May, 1994

OKLAHOMA STATE UNIVERSITY

AN ALTERNATIVE METHOD FOR PARALLEL M-WAY TREE SEARCH ON DISTRIBUTED MEMORY ARCHITECTURES

Thesis Approved:

Huizhu Lu

Thesis Adviser

Blayne E Mayfield

Keith A Teague

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to Dr. Huizhu Lu, my adviser, for all her time, effort, patience, and invaluable suggestions and comments during the entire thesis process. I would also like to thank Dr. Blayne Mayfield for his comments and suggestions as well as for serving on my thesis committee. I wish to give special thanks to Dr. Keith Teague for the use of the Hypercube parallel processor as well as for providing me with a solid background and interest in parallel processing and parallel programming.

I also want to thank Jill, my wife, for all of her patience, love, and understanding, as well as for grading a lot of COMSC 3431 assignments while I worked on my thesis. Finally I wish to thank my parents. Without their emotional and financial support and encouragement, not only this thesis, but my entire education would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter I

Introduction

The efficient storage, access, and update of large amounts of data stored in the main memory of a computer has long been the subject of intense research. Some of the most efficient and general algorithms that have emerged as a result of this research are those that maintain the data in the form of a balanced search tree structure. Thus, it is not surprising that with the increased commercial availability of low cost parallel processing computers, much recent attention has been focused on the efficient implementation of balanced search tree structures on these parallel processing computers.

The majority of this research has been based on shared memory multiprocessor systems as opposed to distributed memory systems. While the shared memory schemes do offer an efficient implementation of balanced search tree structures on a multiprocessor computer, they are limited to a relatively low number of useable processors (20 to 30) by the memory contention and bus bottleneck of the very architecture on which they are implemented. Of the work that has been done on distributed memory systems, the majority is based on hypothetical special purpose database machines which are designed based on an actual tree-like structure. As such, these schemes do not efficiently map to a general purpose multiprocessor which usually consists of several identical processors connected by one of many various topologies each with a different degree of connectivity.

The development of efficient balanced tree maintenance algorithms for general purpose distributed memory multiprocessors has received only limited attention. As

these general purpose multiprocessors continue to become more prominent and support an increasing number of processors, attention should be given to the development of efficient search tree algorithms that can take advantage of the processing power and available interprocessor connectivity of these multiprocessors.

The objective of this research is to develop a scheme to efficiently maintain a balanced search tree on a general purpose distributed memory multiprocessor consisting of several identical processors connected by some arbitrary interconnection network. In particular, this work will focus on a scheme to maintain a balanced M-way search tree capable of concurrent search, insert, and delete operations as well as a periodic rebalance operation suitable for implementation on such a general purpose distributed memory multiprocessor.

In Chapter II, several methods from the literature for distributing and maintaining a search tree on a multiprocessor are presented. Emphasis is placed on pipelined schemes for use on distributed memory multiprocessors. In Chapter III, a new scheme for distributing and maintaining a balanced search tree on distributed memory is presented. The operations of insert, delete, and access are discussed as well as a detailed explanation and example of a parallel periodic rebalancing algorithm. An example of the new scheme as mapped to the hypercube parallel processing architecture is presented. Also, this new scheme is compared with the typical pipelined style of distributed memory search tree maintenance schemes for performance and efficiency. In Chapter IV, an empirical evaluation and comparison of these two methods as implemented on the hypercube multiprocessor architecture is presented. The performance results of each scheme as implemented and tested on the Intel iPSC/2 Hypercube distributed memory general purpose multiprocessor are

presented and discussed. Finally, in Chapter V the results of the research are summarized and discussed. Some contributions and advantages of the new scheme are given and some related areas for future research are presented.

Chapter II

Survey of Related Research

There are several schemes available in the literature for implementing and maintaining a balanced search tree in a parallel processing environment. Due to the earlier development and commercial availability of shared memory parallel processors, most of the earlier schemes are based on the shared memory model. In a shared memory system, the entire data structure is held in an area of common memory that is accessible to all processors, while in a distributed memory system the data structure is partitioned among the local memories of the individual processors.

Shared Memory Schemes

Shared memory schemes have been presented for most of the common tree structures. All of these schemes center around various types of locking protocols to ensure the consistency of the structure. A locking protocol to allow the concurrent manipulation of binary search trees was developed by Kung and Lehman[1]. Samdi[2] presented a basic locking protocol for the B-tree to avoid deadlocks. An improvement to this locking protocol was presented by Bayer and Schkolnick[3] to allow a greater degree of concurrency for the B-tree structure. Ellis[4] gave a locking protocol to allow concurrent search and insert on an AVL-tree, and another[5] to allow concurrent search and insert on a 2-3 tree.

Distributed Memory Schemes

The scope of this research will be limited to those schemes developed for distributed memory parallel processors. In particular, those schemes that can be

effectively mapped to a general purpose distributed memory multiprocessor. Much of the discussion of distributed memory search trees in the literature is based on hypothetical special purpose database machines [6, 7, 8, 9, 10, 11, 12]. In these machines, the processors are permanently configured in a tree topology and the hardware is specifically tuned and restricted to performing database dictionary operations only. These machines can handle a large variety of database operations executing concurrently. Each individual operation can complete in ***O(lg N)*** time, where N is the number of entries in the tree. Thus if there are several operations executing concurrently (such as in a pipeline), these machines are capable of completing an operation every ***O(1)*** time units. However, these machines also require the use of ***O(N)*** processors (processing elements), and more importantly, they are very special purpose machines useful only for one specific task. It is of more interest, and the specific focus of this thesis, to examine and expand upon the work done on implementing search trees that can be effectively used on general purpose distributed memory multiprocessors.

O'Gorman[13] presented a method for distributing the nodes of a binary tree among processors in a linear array such that the left child of a node N is at location 2N and the right child is at location 2N+1 in much the same way a binary tree is stored in an array in conventional data structures. This scheme allows concurrent searches using the nodes of the tree (processors in the array) as routers. While this scheme can be used on a general purpose multiprocessor (the simple vector processor) it still needs ***O(N)*** processors for an N element tree. Also, insertions and deletions are quite costly as they change the size of the tree (and hence the processor array) which causes a need for the inefficient task of rearranging the keys in the array.

Pipeline Schemes

Tanaka, Nozaka, and Masuyama[14] proposed a $\lceil \lg N + 1 \rceil$ processor search tree. They used a $\lceil \lg N + 1 \rceil$ processor pipeline to heapsort a stream of records and arrange the sorted stream into the form of a balanced binary search tree. This pipelined tree searching is just one component of a data flow database computer proposed by Tanaka, Nozaka, and Masuyama. The scheme is not of general interest as a tree maintenance method, as it does not even support insertions or deletions. The important contribution from this scheme is the concept of distributing the nodes of the search tree among the processors of the multiprocessor such that each level of the tree is stored and maintained on a separate processor. This distribution of the tree nodes allows search operations on the tree to be performed in a natural pipeline fashion on a linear array of processors. Also, this level-to-processor mapping guarantees that the number of processors required in the array to store an N element tree is of ***O(lg N)***, as opposed to the ***O(N)*** processors required by the previous schemes.

Similarly, Fisher[15] proposed a scheme for maintaining a trie tree on a pipeline of processors where the length of the processor pipeline is proportional to the length of the longest key. This scheme is also based on the level-to-processor partitioning of the tree. The tree is accessed and maintained by a level-parallel (pipelined) radix tree algorithm. This scheme showed improved performance for small key values. But more importantly, it showed that the ***O(N)*** processor tree machine schemes and their node-to-processor distributions are not necessarily the best method by which to improve performance. Unfortunately, this scheme (as well as the scheme of Tanaka, Nozaka, and Masuyama) relies on a highly specialized VLSI implementation to achieve good performance.

Carey and Thompson[16] developed a pipelined version of a 2-3-4 search tree based on the top-down node splitting algorithm of Guibas and Sedgewick[17]. This scheme also uses a linear array of $\lceil \lg N + 1 \rceil$ processors to maintain a balanced (2-3-4) tree of up to N items. As in the Tanaka, Nozaka, and Masuyama scheme, the tree is distributed among the processor array by levels, thus each processor takes care of one level of the tree. The last processor in the array ($P_{\lceil \lg N + 1 \rceil}$) holds all the keys in the tree; processors P_1 through $P_{\lceil \lg N \rceil}$ hold the index set. An example of a tree distributed on this type of architecture is shown in figure 1. Carey and Thompson's scheme is capable of performing insertions, deletions, exact match searches and range queries. The scheme is almost fully pipelined, so that as many as $\lceil \lg N + 1 \rceil / 2$ operations can be at varying stages of execution at any one point in time. This means that if the pipeline is full, it is possible for one operation to complete every ***O(1)*** message time units.

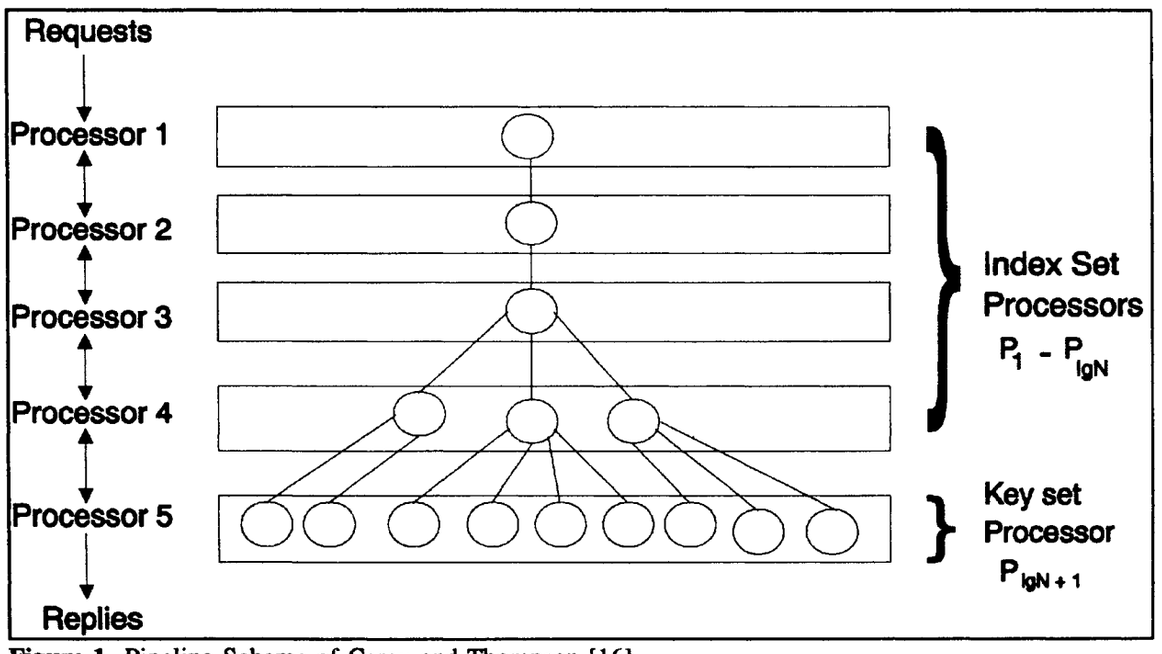


Figure 1 Pipeline Scheme of Carey and Thompson [16]

The work of Carey and Thompson has been extended by Colbrook and Smythe[18] to encompass the general case 2^{w-2}- 2^{w} tree (W>2). This scheme uses a generalized version of the 2-3-4 tree top-down node splitting algorithm of Guibas and Sedgewick. Analogous to the Carey and Thompson scheme, the Colbrook and Smythe scheme uses a linear array of up to $\lceil \lg N/(W-2) \rceil + 1$ processors and is capable of having up to $(\lceil \lg N/(W-2) \rceil + 1) / 2$ operations executing concurrently.

The operation of the Carey and Thompson style pipeline schemes will be presented and analyzed in detail in Chapter III.

Chapter III

An Alternative Search Tree Distribution And Maintenance Scheme

Explanation and Analysis of Carey and Thompson Style Pipeline Schemes

Explanation

Both the Carey and Thompson scheme, and its more generalized version, the Colbrook and Smythe scheme provide an acceptable implementation of general balanced search trees which can be mapped to general purpose distributed memory multiprocessors. However, both of these schemes rely on the pipeline concept for improved performance. As is shown in figure 1, requests come in at the root and propagate down the tree (making appropriate balance transformations at each level if the request is an insertion or deletion), until the last processor in the array is reached. At the last processor, the appropriate action is taken (insert, delete, or match) and an appropriate response is issued to the outside world (it is assumed by these schemes that P_1 and $P_{\lceil \lg N + 1 \rceil}$ have direct communication links with the outside world). If there are continuous requests waiting to be processed, then these schemes can complete an operation every ***O(1)*** message time units. Thus, as long as the pipeline is full (there is a series of requests waiting to be processed), these schemes provide a good improvement in throughput. However, the communication delay between the time a request is issued and the time the response to that particular request is received is ***O(lg N)*** message times. Also, the improvement in throughput is dependant upon the pipeline staying full.

The reason for the ***O(lg N)*** communication time delay can also be seen in figure 1. It is simply because the request must propagate all the way to the last processor before the response may be issued to the requestor. This means that if there are P processors in the processor array, then the request must pass through all P processors on the way to the last (leaf) level processor. For insertions / deletions, there is a node split / concatenation that may take place at each of these P processors (levels in the tree) as the request passes through. Each possible node adjustment (whether it actually occurs or not) requires two additional messages be passed between each processor in the array. One from P_i to P_{i+1} to give P_{i+1} the new key and indicate that an insert / delete transformation may be required to maintain the 2-3-4 tree invariants (message 1, right side of figure 2), and one from P_{i+1} back to P_i to give the new pointer and key arrangements resulting from an adjustment transformation if one was required, or a message indicating that no transformation was necessary otherwise (message 2, right side of figure 2). Finally, the operation is continued from P_i to P_{i+1} along these new pointers, if a transformation was required, or along the old pointers otherwise (message 3, right side of figure 2). The top down node splitting algorithm of Guibas and Sedgewick used by this scheme guarantees that

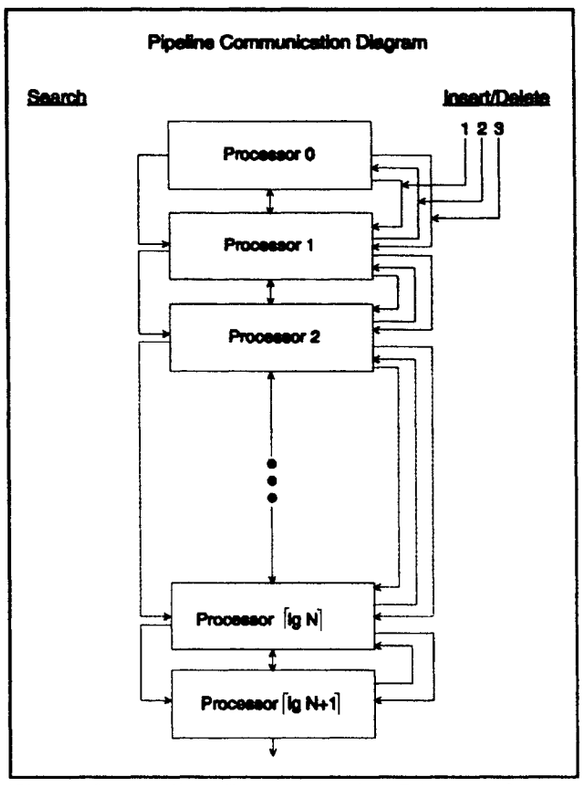


Figure 2 Pipeline Communication Diagram for Insert/Delete Operations (Right) and Search (Left)

node adjustments will not propagate back up the tree. Thus, the communication steps required per update (insert or delete) will always be exactly as just described.

Analysis

In the case of a simple search, one message must be passed between each processor (left side of figure 2), and for an insertion or deletion three messages must be exchanged between each processor (right side of figure 2). Since the processor array must be of length $\lceil \lg N + 1 \rceil$ for a tree of N elements in the Carey and Thompson scheme or $\lceil \lg N/(W-2) \rceil + 1$ for the Colbrook and Smythe scheme, the delay for a single search is equal to $\lceil \lg N + 1 \rceil$ or $\lceil \lg N/(W-2) \rceil + 1$ respectively. Similarly, the delay for a single insertion or deletion is $3(\lceil \lg N + 1 \rceil)$ for the Carey and Thompson scheme and $3(\lceil \lg N/(W-2) \rceil + 1)$ for the scheme of Colbrook and Smythe. In both cases, this delay is of $\boldsymbol{O(\lg N)}$. If the number of searches is roughly equal to the total number of insertions and deletions, this yields an average delay for any single operation of $2(\lceil \lg N + 1 \rceil)$ and $2(\lceil \lg N/(W-2) \rceil + 1)$ for the Carey and Thompson and Colbrook and Smythe schemes, respectively. Also notice that until any processor P_i receives a reply from processor P_{i+1} for an insert or delete operation, the keys and pointers in P_i may be incorrect. Thus, at any given point in time, half of the processors will be dealing with insertion and deletion transformations. Hence, for an arbitrary sequence of insert, delete and search operations, there may be a maximum of $\lceil \lg N + 1 \rceil / 2$ (Carey and Thompson) or $(\lceil \lg N/(W-2) \rceil + 1)/2$ (Colbrook and Smythe) operations executing concurrently at any given time.

As the number of processors on commercially available distributed memory parallel processing machines continues to increase, the response time delay for these

pipeline schemes will generally continue to increase as well. This is because of what is referred to as the throughput/response time trade-off. As more processors are added to the processor pipeline, there is an increase in throughput (if the pipeline stays full), but because the request and intermediate replies must now propagate through more processors, there is generally also a corresponding degradation in the response time for a single query. Furthermore, not only will an increase in the number of processors in the pipeline generally increase the response time delay, but it will also increase the number of requests for operations that must be available in order to fill the pipeline and keep it filled, which is required to get the increase in throughput.

It should also be pointed out that if the multiprocessor is a general purpose machine composed of identical processors, as most are, much of the memory in the processors that contain the upper levels of the tree will be wasted by the pipeline schemes since all the keys must be stored in the last (leaf level) processor. Also notice that the pipeline schemes use a linear array of processors which requires only one communication connection per processor. While this is certainly not a disadvantage, it does mean that these schemes do not take full advantage of modern multiprocessors which offer a higher degree of connectivity. These schemes also assume that the connections on the first processor P_1 and the last $P_{\lceil \lg N + 1 \rceil}$ may communicate directly with the outside world, which cannot be accepted to be a valid assumption for a general purpose multiprocessor. In light of these deficiencies, attention should be given to new ways to distribute, use, and maintain a balanced search tree structure on distributed memory multiprocessors. This new method should attempt to avoid the problems associated with the pipeline dependency of these other schemes.

In particular, any new scheme should attempt to minimize the response time delay for an individual request while still attempting to achieve increases in throughput for a series of requests. Also, since most commercially available general purpose multiprocessors have identical processors and a relatively rich interconnection network, a new scheme should attempt to make full use of the computational and storage capabilities available on every processor as well as the enhanced communication between processors. Of course, this new scheme should also be able to provide concurrent insertions, deletions, and accesses as well as provide a mechanism to maintain the global balance of the structure, all while ensuring the consistency of the search tree.

Alternative Search Tree Distribution and Maintenance Scheme

Description of Tree Distribution

Like the pipeline schemes, this new scheme will maintain an M-way search tree on a distributed memory parallel processor. The M-way tree will consist of nodes with M-1 keys and M pointers. A visual interpretation of the node structure can be seen in figure 3. Each key has an associated left pointer that points to either nothing or another node that contains values less than the key associated with the pointer. Each node also contains a count K of keys in the node and a right pointer. The right pointer always points to whatever is to the right of the K^{th} key in the node.

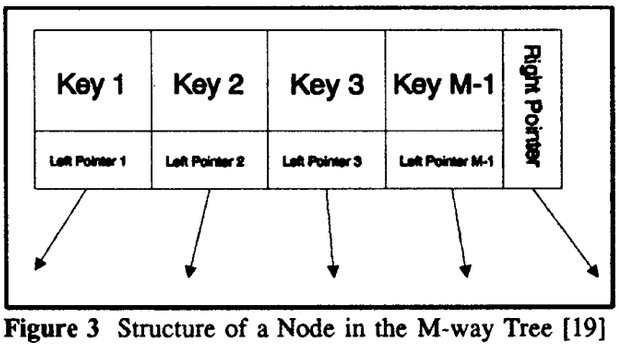


Figure 3 Structure of a Node in the M-way Tree [19]

In order to avoid the inefficient use of processors and connectivity and reduce the response time delay involved with earlier pipeline schemes, it is necessary to partition the tree among the processor's local memories in a different fashion than that used by the pipeline schemes. Recall from figure 1 that these schemes partition the tree among the processors by levels of the tree. This partitioning is inherently beneficial to the pipeline schemes for improvement in throughput and top-down balancing, however it is also inherently detrimental to the response time for a single query as well as efficient use of memory and processing power in the processors that contain the upper levels of the tree. In order to avoid these problems, this new scheme partitions the M-way tree among the processors of a multiprocessor by subtrees of the root rather than by levels of the tree.

An example of an arbitrary M-way tree partitioned by subtrees of the root is shown in figure 4. The *root* processor simply acts as a communication port for the tree. It routes incoming requests to the appropriate *subroot* processor and receives results form the *subroot* processors. Assuming bidirectional links between the processors and that any processor may communicate with any other processor (not necessarily by a direct link), the processors of the multiprocessor should be logically connected as shown in figure 4, where P_1 is the *root* processor and P_2 through P_8 are the *subroot* processors. Notice that the value of M for the M-way tree need not be equal to the number of *subroot* processors. Each *subroot* processor maintains its own local M-way subtree. The node in the *root* processor is a *pseudoroot* which contains P-2 keys and their associated left pointers and a right pointer, where P is the total number of processors available for use in the tree. Each of the *pseudoroot* pointers is

simply the address of a *subroot* processor which contains an M-way tree with keys that are less than or equal to the value of the of *pseudoroot* key associated with the pointer.

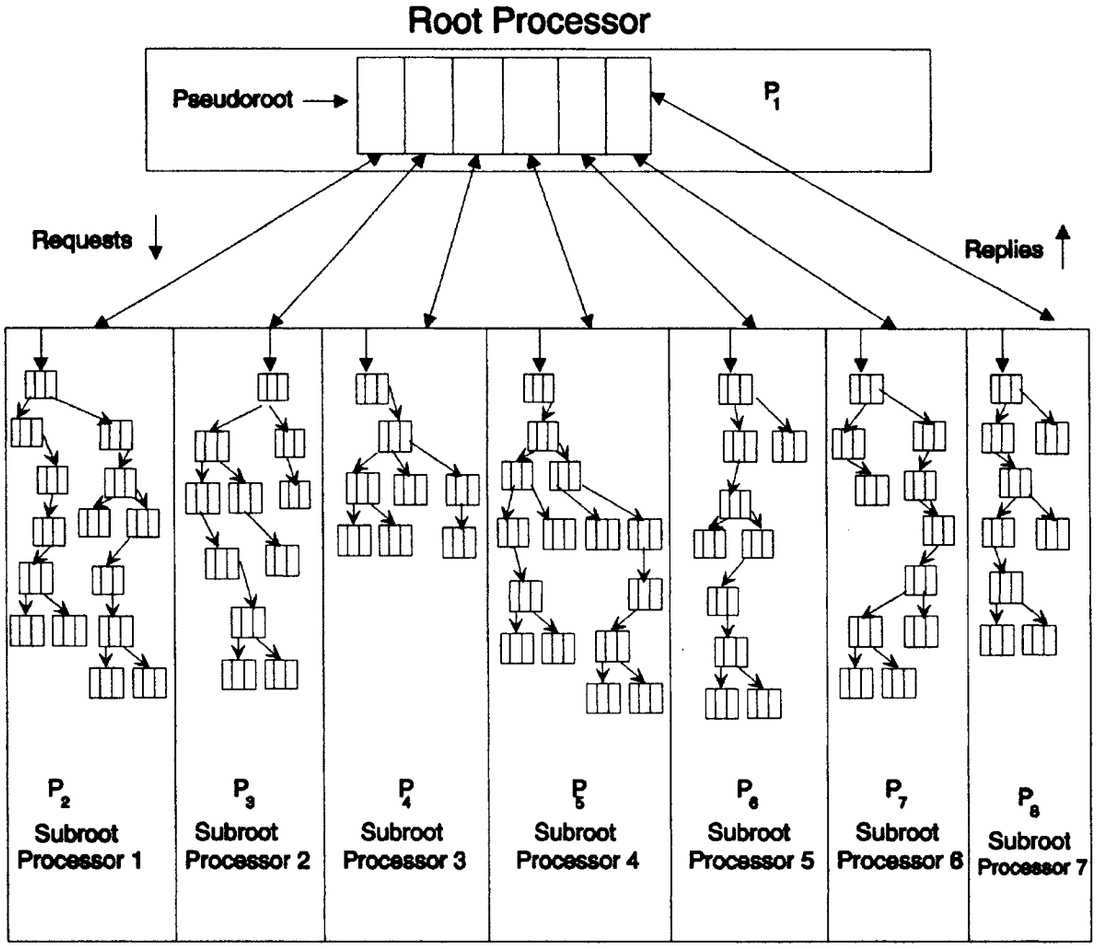


Figure 4 Example of Tree Partitioned by Subtrees of the Root

Description of Tree Operations

The dictionary operations on the tree are performed as follows. The key to be inserted, deleted, or accessed is compared left to right with the keys in the *pseudoroot* in the *root* processor. If and when the key is less than the i^{th} *pseudoroot* key, then the root passes the request on to the *subroot* processor whose address is associated with

the i^{th} *pseudoroot* key. If the key is greater than all the *pseudoroot* keys, then the request is sent to the *subroot* processor whose address is stored in the right pointer of the *pseudoroot*. Upon completion of the request at the *subroot* processor, a response is sent back to the *root* processor. This informs the *root* processor that the operation at that *subroot* processor is complete and it is ready to process another request. Thus, the *root* processor must dispatch requests to the appropriate *subroot* processors and poll for completion responses from these processors. The *subroot* processors must accept these requests, perform the requested operation, and send a completion response back to the *root* processor. Since each *subroot* processor may process requests completely independently of the others, up to P-1 operations can be going on concurrently at any given time. Notice that this scheme safely assumes only one processor (the *root* processor) has a direct communication link to the outside world.

Root and *Subroot* Processor Algorithms

A high-level outline of the *root* and *subroot* processor algorithms is given in figure 5. An explanation of the data structures and functions referenced by these algorithms follows. Done is a boolean variable that will remain FALSE as long as there are requests to process. New_Request is a compound variable with two required fields; namely Key and Operation. Key is the key used to place the item in the tree, and Operation is the operation to be performed (insert, delete, or access). The variable Request is of the same type as New_Request. The function Get_Incoming_Request() will check to see if a new request has arrived from the outside world, and if so return it in New_Request, otherwise it will set New_Request to NULL.

Root Processor Algorithm

```
While( !Done )
  Begin
      Get_Incoming_Request( New_Request )
      If( New_Request )
        Queue_Incoming_Request( Queues, New_Request, PseudoRoot )

      For( i=1 to Number_of_Processors - 1 )
        Begin
          If( Queues[i].Waiting )
            Begin
              If( Message_Done(Queues[i].Receive_id) )
                Queues[i].waiting = FALSE
            End
          If( !Queues[i].Waiting And !Queues[i].Empty )
            Begin
              Request = DeQueue( Queues[i] )
              Queues[i].Send_id = Send(Request,Queues[i].Subroot_Proc_id)
              Queues[i].Receive_id = Receive(Signal,Queues[i].Subroot_Proc_id)
              Queues[i].Waiting = TRUE
            End
        End
  End
```

Subroot Processor Algorithm

```
While( !Done )
  Begin
      Receive_id = Receive(Request,Subroot_Proc_id)
      Message_Wait( Receive_id )
      Switch( Request.Operation )
        Begin
          Case :  INSERT
                  Insert(Request)
          Case :  DELETE
                  Delete(Request)
          Case :  ACCESS
                  Search(Request)
        End
      Send_id = Send(Signal,Root_Proc_id)
  End
```

Figure 5 Outline of *Root* (top) and *Subroot (bottom) Processor Algorithms* for Insert, Delete and Access Communication Management

The function Queue_Incoming_Request() will place New_Request in the appropriate queue based on the value of the Key field as compared to the keys in the *pseudoroot*. The functions Send() and Receive() are asynchronous message passing functions. The Send() function sends the variable that is its first argument to the processor with the identification number that is its second argument. Similarly, the Receive() function places a message from a processor with the identification number of its second argument into the variable named as its first argument. Since both of these message passing functions are asynchronous, they both simply initiate the requested action and return immediately. They do not block until the message has been delivered or received. They both return a message identification number which can be used to check the status of the actual message (as described below).

Queues is a single dimensional array of queue heads. The elements stored in these queues are the incoming requests of type New_Request. There is a separate queue of requests maintained for each *subroot* processor. Each queue head contains the additional information fields Send_id, Receive_id, Waiting, Empty, and Subroot_Proc_id. Send_id and Receive_id are message identification numbers returned by the Send() and Receive() functions, respectively. These message identification numbers are used by the Message_Done() function to determine if a particular message has been received. Message_Done(), when supplied with a message id, returns TRUE if the message has been received, or FALSE otherwise. Waiting is a boolean variable set to TRUE when a request is sent to a *subroot* processor, and set back to FALSE when the reply from that processor is received. Empty is a boolean variable that indicates if the queue for a given processor has any requests waiting to be processed or not.

Root Processor Algorithm The *root* processor loops until there are no more requests to be processed. Inside the loop, it accepts new requests and places them in the queue for the appropriate *subroot* processor based on the keys in the *pseudoroot*. It then polls each of the queues and checks to see if each queue's associated processor is currently processing a request (i.e. if Queues[i].Waiting is TRUE). If it is, then the root processor calls Message_Done() with Queues[i].Receive_id to see if that processor has completed that request. If the processor has completed the request (Message_Done() returns TRUE), then Queues[i].Waiting will be set to FALSE indicating that this processor is ready for another request. If a processor is in this state (ready to accept a request) as indicated by Queues[i].Waiting being FALSE, and the processor's associated queue is not empty (Queues[i].Empty is FALSE), then a new request is taken from that processor's queue and sent to that processor. A receive buffer is then posted for the completion message that will be sent by this processor, and the processor's waiting flag (Queues[i].Waiting) is set to TRUE until this completion message arrives.

Subroot Processor Algorithm The outline of the algorithm that runs on the *subroot* processors (figure 5, bottom) is simple and straightforward. Each processor simply posts a Receive() and waits for a request to process. It waits for a request by calling the function Message_Wait() which will block until the message with the message id passed to it as an argument arrives. When a request does arrive, the processor checks the Operation field and calls the appropriate function (Insert(), Delete(), or Search()) to perform the requested operation on the tree. Upon completion of the operation (return from the function), a completion message is sent

back to the *root* processor to inform it that this *subroot* processor is now ready to process another request.

Analysis of Communication Costs

In this scheme, the response time delay is a function of the connectivity of the processors. This is assuming that the cost of sending a message between the processors is the dominant portion of the time spent processing a request, and thus ignoring for the moment the $\mathbf{O(\log_m N/(P-1))}$ tree search time involved at each *subroot* processor. If the *subroot* processor processing the request is directly connected to the *root* processor, then the response time will be equal to 2 message times. One to send the request to the *subroot* processor and one to send the response back to the *root* processor. In the worst case, the request would have to be routed through each of the *subroot* processors to reach its destination. Hence, the worst case performance would be of **O(P-1)** message times. This worst case would occur for processors linked by a linear array connectivity. The best case would be of **O(1)** message times, which would occur for a completely connected multiprocessor, while the average case for most current multiprocessors is somewhere in between depending on their degree of connectivity.

Example of Scheme Mapped to a Current Multiprocessor Architecture

As an example, consider the hypercube multiprocessor architecture. In this architecture, there are **lg P** connections per processor, where **P** is again the total number of processors allocated. Also, processors may only be allocated in quantities that are a power of 2. So, for the processor that is logically connected as shown in

figure 4 (redrawn in figure 6 to reflect the node numbering starting at zero on the hypercube), the actual physical connections in the hypercube are as shown in figure 7.

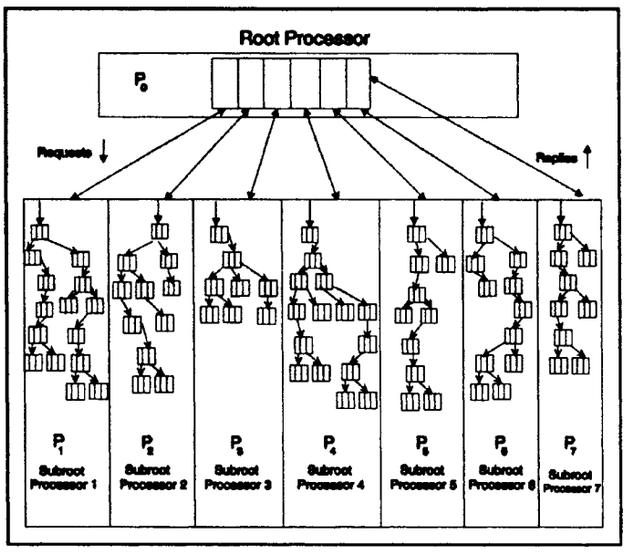


Figure 6 Figure 4 Redrawn with Processors Labeled Starting from 0 for Comparison with Figure 7

For requests that get routed to *subroot* processors 1, 2, or 4 the response time will be the best case 2 message times, while requests that get routed to *subroot* processors 3, 5 or 6 will have a response time of 4 message times. Requests that get routed to *subroot* processor 7 will take 6 message times to complete. In general, the maximum number of message times to go from any processor on a hypercube to any other processor on the hypercube is **lg P**. Thus, the response time delay for this new scheme on a hypercube architecture is **O(lg P)** as opposed to **O(lg N)** for the pipeline schemes.

Not only will this scheme make the response time delay a function of the number of processors and their connectivity rather than a function of the number of nodes in the tree, but it will also make much more efficient use of the available processors and memory because each processor will now store and perform operations on a complete subtree

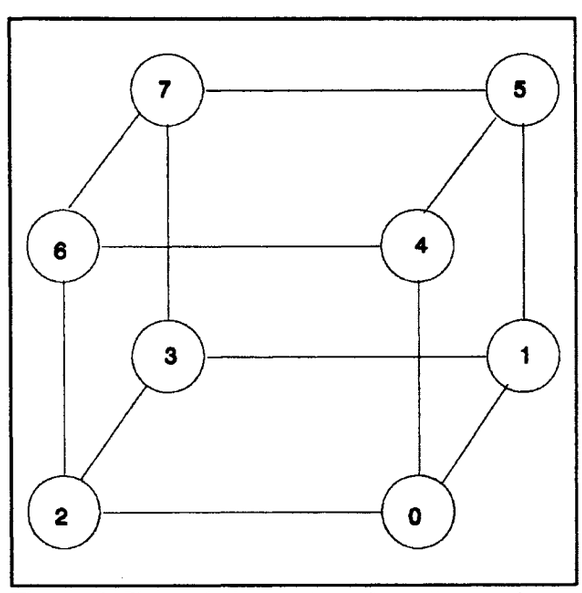


Figure 7 Physical Connections for a Hypercube Corresponding to Logical Connections of Figure 6

of the global search tree. Notice that as previously mentioned, this scheme will make it possible to have up to **P-1** operations executing concurrently. Recall that the *root* processor issues requests for operations to the appropriate *subroot* processor, then awaits a response from that processor indicating completion of the operation, at which point the next request may be sent to that processor. Note however that since the *root* processor need not wait for a response from any one *subroot* processor before sending a new request to a different *subroot* processor that is ready, it is possible to have an operation being processed at each of the **P-1** *subroot* processors at the same time (see the *root* processor algorithm in figure 5-top for details). In this way, this new scheme can be thought of as **P-1** pipelines all of varying length. Hence, this scheme provides good increased throughput while minimizing the response time as much as the connectivity of the architecture will allow.

Balance Method Used by New Scheme

So far this scheme has shown improvements in all the areas in which the earlier pipelined schemes were deficient, but there is one desirable attribute of the pipeline schemes that is not yet present in this scheme, namely that of maintaining the global balance of the search tree. Recall that both the pipelined schemes of Carey and Thompson and Colbrook and Smythe maintained search trees that were globally balanced across the processors. The level to processor partitioning of these schemes coupled with top-down node splitting was used to maintain a 2-3-4 tree and a 2^{W-2}-2^{W} tree respectively.

In this new scheme, the level to processor partitioning of the tree has been replaced with a subtree partitioning, the benefits of which have already been discussed.

This subtree partitioning does however make the use of traditional tree balancing techniques very costly. Any kind of on-the-fly balancing of the global tree structure could necessitate changes across several subtrees. This would require costly communication between the *subroot* processors as well as cause surrounding *subroot* processors to have to wait for completion of the balancing operations caused by an update on some other *subroot* processor before being able to service their next request. This would considerably reduce the amount of concurrency possible and thus the efficiency of the entire scheme. For these reasons, the new scheme will use a parallel version of the M-way search tree periodic balancing algorithm presented by W.F. Smyth[19]. The next section contains a detailed description of this balancing process, followed by an explanation of how it is used by this new tree search scheme.

Description of Serial Balancing Algorithm

The algorithm used is based on a periodic balancing technique that was originally presented for the binary tree by Stout and Warren[20] and later extended by Smyth to the M-way tree, M>2. A detailed explanation and example of the algorithm can be found in Appendix A. The following is an overview of the functionality of the algorithm and how it is used by this scheme to maintain the global balance of the parallel search tree. In general, Smyth's algorithm works in two stages.

1. Conversion of the existing tree to a *vine*
2. Conversion of the *vine* to a balanced tree

A *vine* is simply an M-way tree, consisting of nodes such as those depicted in figure 3, in which every left pointer is NULL and every right pointer points to another node, except the right pointer of the last node in the vine which points to NULL. Each vine node, except possibly the last (rightmost) node, is completely full (i.e. contains exactly

M-1 keys). So a vine is essentially a linked list of completely full tree nodes connected by right pointers.

The process of converting the M-way tree to a vine consists of repeated applications of the *rotation* shown in figure 8. These rotations start with the root node of an M-way tree as the current node (U) and rotate each left subtree (V) attached to this node to the right. When all the left subtrees of the current node (U) have been rotated, this current node is added to the vine and the new current node becomes the right child of this node. As each node is added to the vine, keys are shifted into the previously added vine node from the most recently added node to ensure that there are exactly M-1 keys in the previously added vine node. This assures that each node in the vine contains exactly M-1 keys. If this shifting causes the most recently added vine node to become empty, then that node is simply left out of the vine. This is then repeated for each node in the tree until the right child of the most recently added vine node is NULL. Notice that these rotations cause the keys in the vine to be stored in strict ascending order. Figure 11 (page 26) shows an example of a vine created by applying this process to the unbalanced M-way tree (M=4) containing randomly generated integer keys found in figure 9.

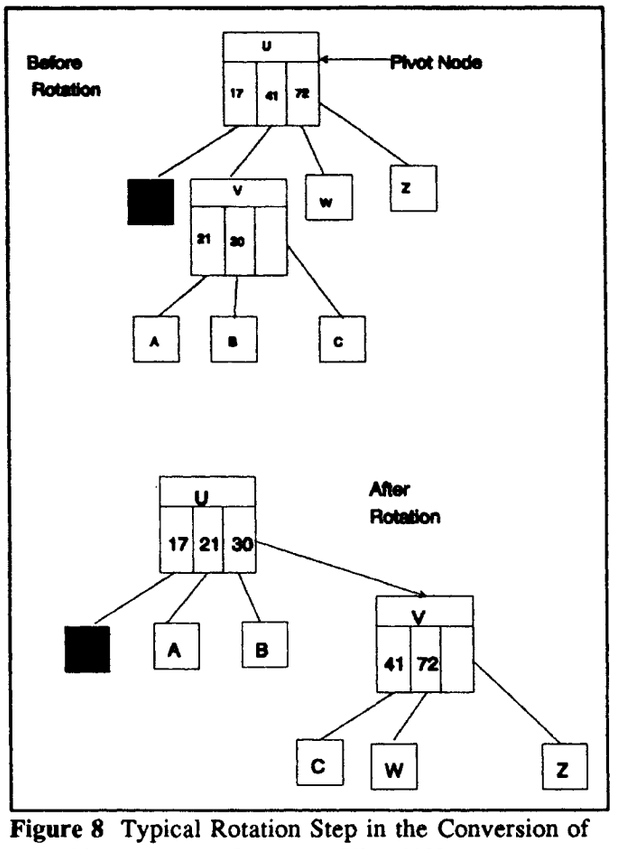


Figure 8 Typical Rotation Step in the Conversion of an Arbitrary M-way Tree to a Vine [19]

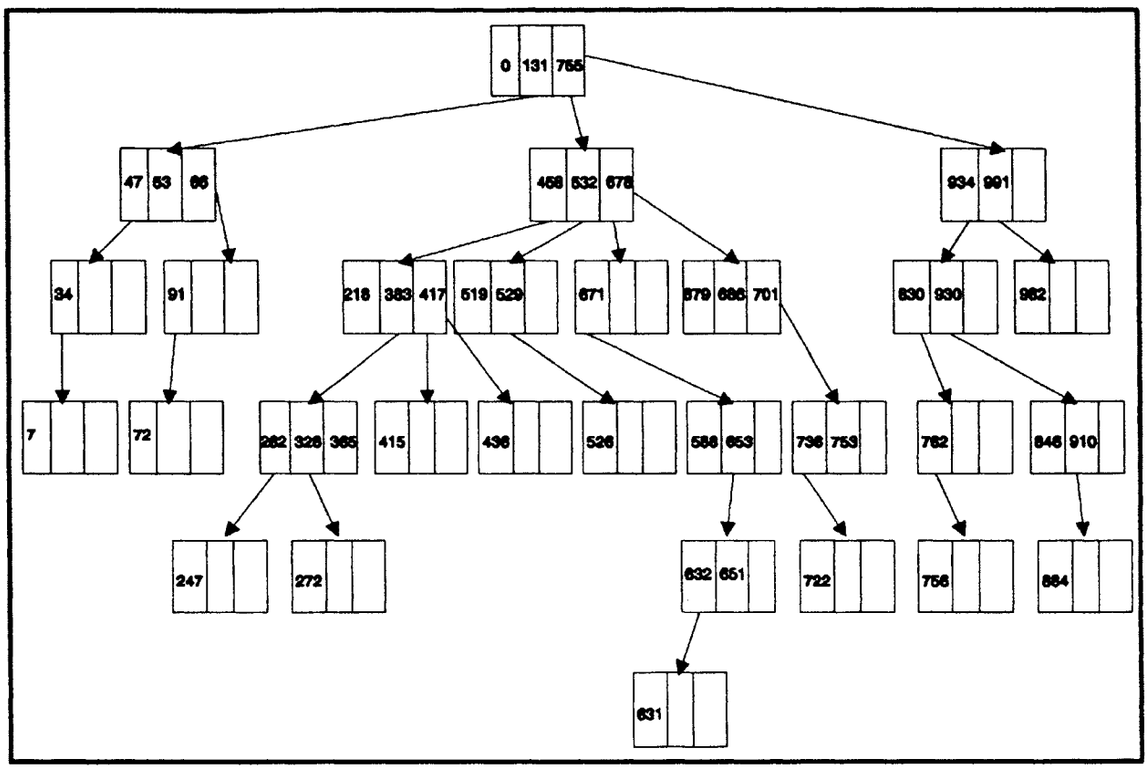


Figure 9 Arbitrary Unbalanced M-way Search Tree

The process of converting this newly created vine to a balanced M-way tree is accomplished by repeated applications of the *compression* shown in figure 10. The compressions perform the inverse of the rotations. They process the vine in groups of M nodes, starting with the root (U) and the three nodes to its right (V, W, and X). Each *compression* rotates the three nodes (U,V,W) around to be the left children of X, making pointer adjustments and

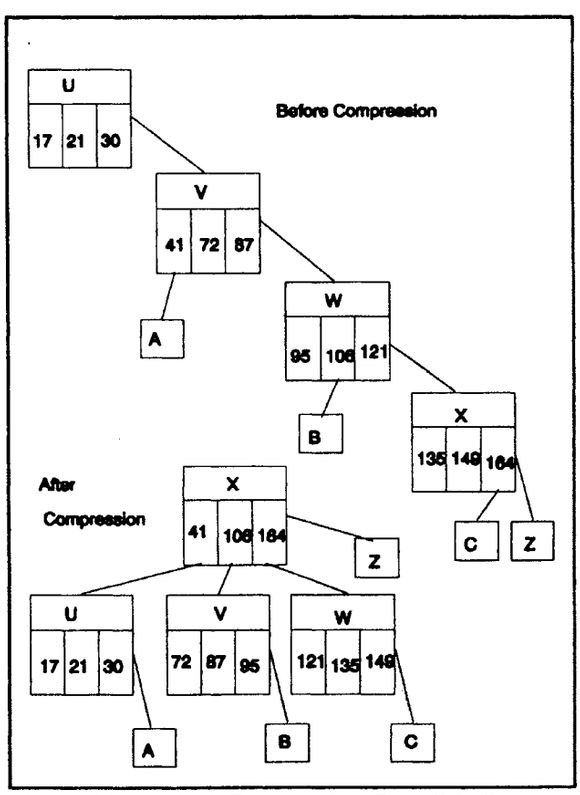


Figure 10 Typical Compression Step (M=4)

key exchanges as required. The node X then becomes the U-node for the next compression and so on until the vine has been completely processed (compressed). At this point, the vine will have been converted into a balanced M-way tree. An example of a balanced M-way tree created by this process from the vine in figure 11 is shown in figure 12.

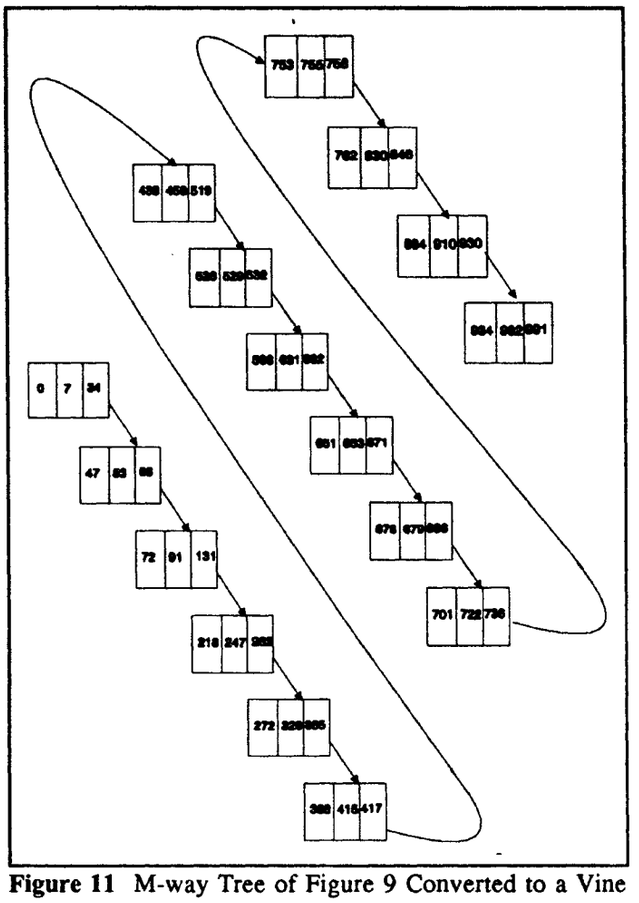


Figure 11 M-way Tree of Figure 9 Converted to a Vine

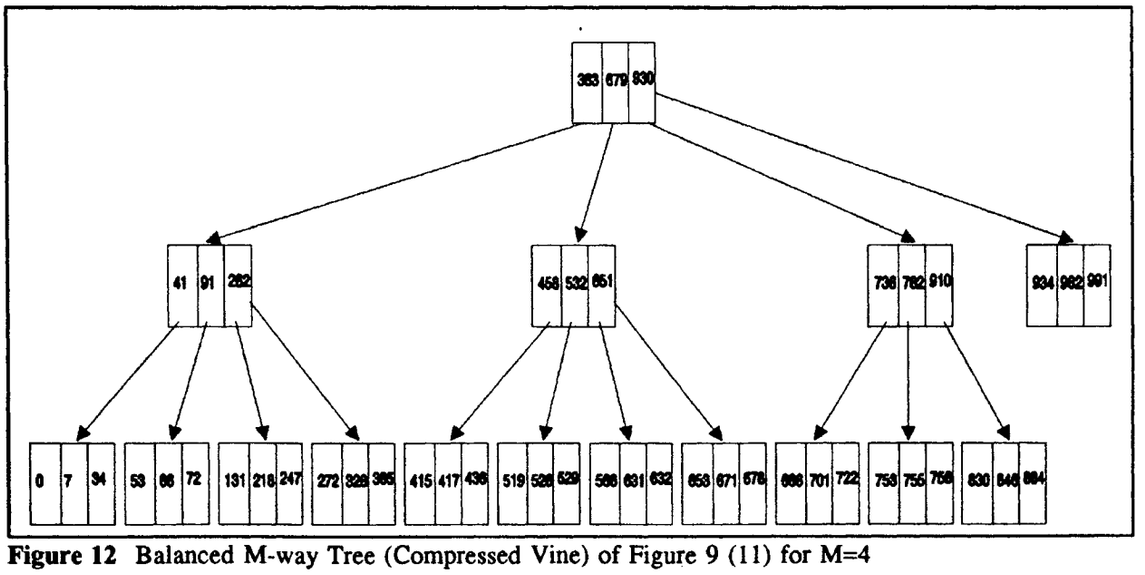


Figure 12 Balanced M-way Tree (Compressed Vine) of Figure 9 (11) for M=4

Description of Parallel Version of the Balancing Algorithm

To use this balancing algorithm efficiently in a parallel implementation, the basic steps of the algorithm are modified as shown below.

1. Each of the *subroot* processors performs the conversion of their local M-way subtree to a vine in parallel.
2. Nodes are exchanged between the *subroot* processors so that each processor will have an equal number of nodes.
3. Each *subroot* processor performs the conversion of its local vine to a balanced M-way tree in parallel.

Steps 1 and 3 are exactly the same as the serial case discussed above, except that they are carried out simultaneously by each of the *subroot* processors. Step 2 is required because we not only want to balance the individual M-way subtrees on each of the *subroot* processors, but also the global search tree structure as well. To balance the global search tree using this algorithm, each processor should have an equal number of tree nodes. This transfer of nodes is best done between the two stages of the original sequential balancing algorithm. This is the best time because when the M-way trees at each *subroot* processor are in the form of a vine, segments from the root (tail) of a neighboring processor's vine may simply be inserted at the tail (root) of the processor to the left (right).

This modified version of Smyth's algorithm is well suited for use in this new parallel tree search and maintenance scheme. While much of the actual data transfer involved in Step 2 must be done sequentially, Steps 1 and 3 are entirely independent and may be done wholly in parallel. Furthermore, since the transfer of these nodes in Step 2 is done by transferring sections of the tree in the form of a vine, the relinking involved at each *subroot* processor during Step 2 is trivial. An algorithm to perform this transfer of nodes (Step 2) is given in figure 13.

Each of the Following Variables is Calculated for Each Processor

Nodes_per_proc = Total_nodes / (P-1)

Give = number of nodes on processor over Nodes_per_proc

Requests = how many nodes needed on this processor to bring the node count up to Nodes_per_proc

Requests_from_left = how many nodes the processor to the left of this processor is requesting

Body of the Algorithm

```
FOR( i = leftmost subtree processor TO rightmost subtree processor )
  BEGIN
    IF( subroot processor[i].Give > 0 )
      BEGIN
        Send_right(subroot processor[i].Give)
              This sends subroot processor[i].Give nodes from subroot
              processor[i] to subroot processor[i+1]
        i = i + 1
      END
    ELSE
      BEGIN
        Current_processor = i
        WHILE( subroot_processor[Current_processor].Requests > 0 )
              Current_processor = Current_processor + 1
        Send_left(Current_processor,i)
              This sends the requested number of nodes
              (Requests_from_left) from Current_processor to the neighbor
              processor to the left. This left neighbor processor then sends
              the number of nodes requested of it (its Requests_from_left) to
              its left and so on until the requests on the original processor
              (processor[i]) have been satisfied
        i = Current_processor
      END
  END
```

Figure 13 Outline for Node Transfer Step of Parallel Balancing Algorithm

This algorithm is run by the *root* processor after receiving confirmation of completion of the conversion to a vine (Step 1) from each of the *subroot* processors. The calculation of the variables used by this algorithm (Requests, Give and Requests_from_left) are calculated in equations **1**, **2**, and **3** respectively. Again, a unique set of these variables is kept by each processor, so subscript notation is used in the formulae to indicate which processor the variables correspond to. For example, Processor[i].Left means the value of the variable Left for processor *i* as *i* goes from the leftmost (1) to the rightmost (P-1) processor.

Processor[i].Requests = ***(Nodes_per_proc + Processor[i].Requests_from_left) - (Processor[i].Nodes + Processor[i-1].Give)*** ***(1)***

Then based on the value of Requests (Processor[i].Requests) from ***(1)***, Give and Requests are reset as follows.

If (***Processor[i].Requests*** < ***0***) Then ***(2)***
    Processor[i].Give = ***-(Processor[i].Requests)***
    Processor[i].Requests = ***0***
Otherwise,
    Processor[i].Give = ***0***

And Finally,
Processor[i+1].Requests_from_left = ***Processor[i].Requests*** ***(3)***

After the calculation of these variables, the algorithm simply processes each of the *subroot* processors from left to right. If the i^{th} processor has more than the required nodes present, it sends the extra nodes to the processor to its right, if it has less than the required nodes, it requests what it needs from the processor to its right. A request keeps going down (right) through the processors, possibly accumulating new requests as it goes, until it finally reaches a processor which can satisfy all the requests accumulated so far (which it is guaranteed to be able to find). The required nodes are then sent left satisfying all requests accumulated to that point. The processing then

continues in the same manner from that point on to the right until the rightmost processor has been processed. At this point, all the node transfers (Step 2) will be complete and each *subroot* processor may convert its own local vine to a balanced tree simultaneously (Step 3).

Example of the Parallel Balancing Process

An example of this entire balancing process is now given. Consider again the unbalanced tree given in figure 9. Under this new scheme, each subtree of the root would be on a separate processor. Figure 14 shows the tree partitioned among the processors along with the corresponding values of **Give**, **Requests**, and **Requests_from_left** (R.F.L.) as well as the number of nodes present on each *subroot* processor (**Nodes)** after all the *subroot* processors have completed the conversion of their respective subtree to a vine (Step 1).

Following the algorithm of figure 13, the *root* calculates **Nodes_per_proc** to be 4 (total nodes/number of *subroot* processors), and then begins node distribution processing with the leftmost processor P_1. P_1 cannot have any requests from the left (R.F.L.) since it is the leftmost processor. Since P_1 has only one node, it must request 3 (to bring it up to the required 4 **Nodes_per_proc**) from the processor to its right (P_2). Hence, P_2's requests from left value (R.F.L.) is set to 3. P_2 has only 3 nodes, so it must request 1 node from the processor to its right (P_3) in addition to the 3 nodes requested from its left (from P_1). Thus P_2 must request a total of 4 nodes from P_3 which results in setting P_3's R.F.L to 4. P_3 has 10 nodes, but only needs 8 (4 to maintain the required **Nodes_per_proc** plus 4 which are requested from processors to its left). As a result, P_3 has 2 nodes to give to the processor to its right (P_4).

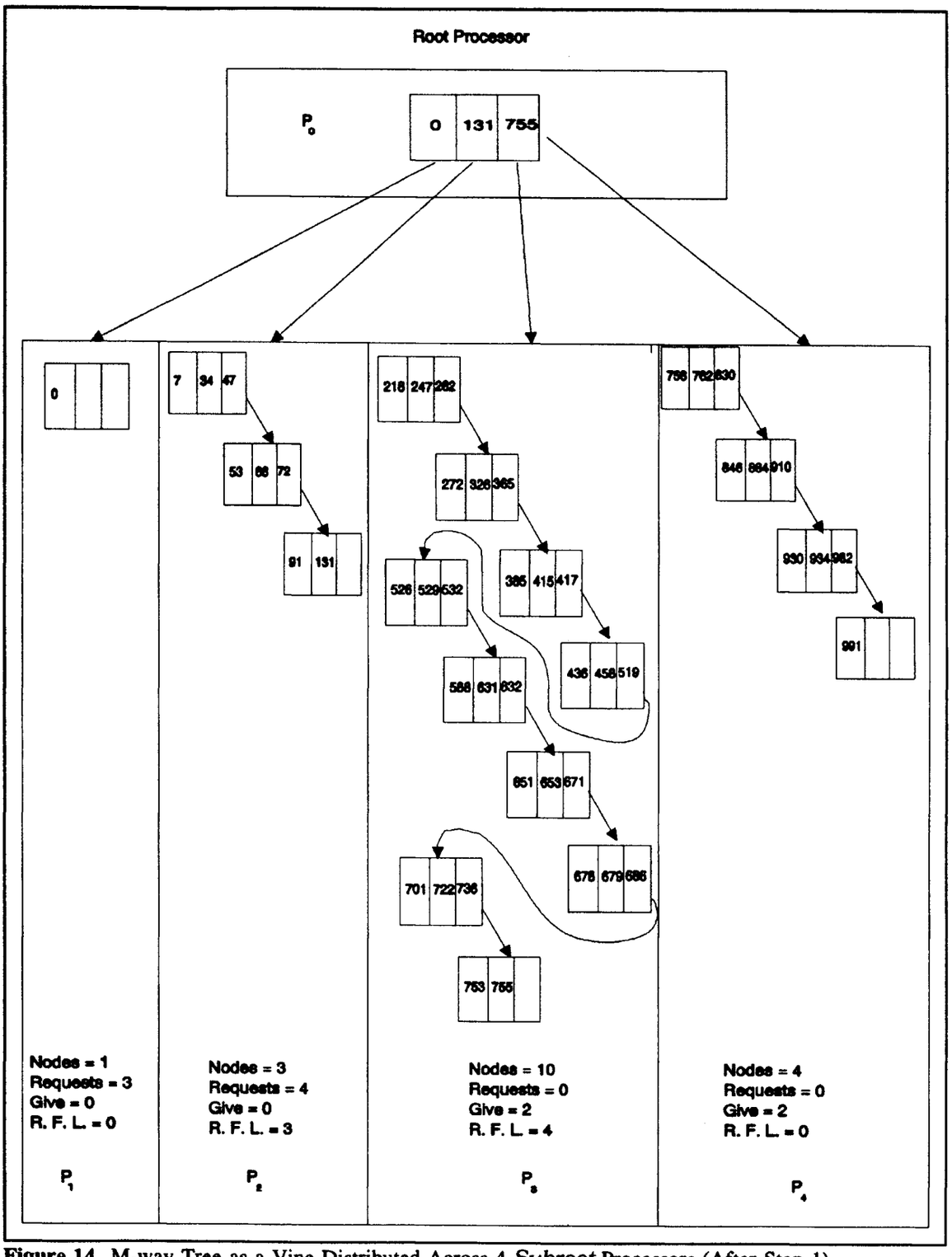


Figure 14 M-way Tree as a Vine Distributed Across 4 *Subroot* Processors (After Step 1)

So P_4 has no requests from the left (i.e. R.F.L. = 0). Since P_4 is the rightmost processor its **Give** value is never considered because there is no processor to its right.

But notice that if there was a processor to the right, P_4 would have 2 nodes to give to it. This is because it already has exactly the required 4 nodes plus an additional 2 nodes that it would receive from P_3.

Now that the *root* processor has calculated all the required values, it must simply examine each *subroot* processor from left to right, as described in figure 13, to see if it has any requests for nodes or nodes to give, and take the appropriate action for either case. In this example, P_1 has a request for 3 nodes, so the *root* processor searches right through the processors until it finds a processor with no requests. When a processor has no requests, it means this processor has sufficient nodes to satisfy its own needs as well as all the requests from its left. For this example, the first such node to the right of P_1 is P_3. So P_3 is instructed to send what is requested from it (its R.F.L.) to the processor to its left (P_2). This results in the adjustments shown by the dashed arrows in figure 15. P_2 now has 7 nodes so it can satisfy its R.F.L. and send 3 nodes to P_1. These adjustments are reflected by the dashed arrows in figure 16.

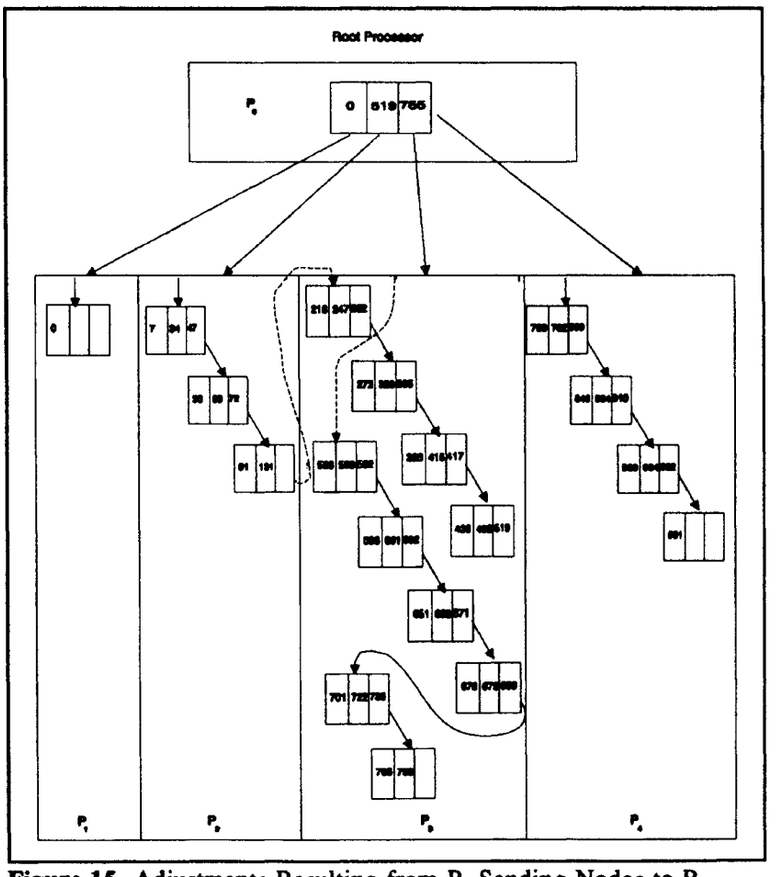


Figure 15 Adjustments Resulting from P_3 Sending Nodes to P_2

The adjustments shown in figure 17 need not be made since P_4 is the rightmost processor, but they are included to show how the vine is adjusted when a processor has excess nodes to give to the right, as P_3 has to give to P_4. Finally, the keys in the *pseudoroot* of the *root* processor are adjusted to reflect the adjusted vines and the node

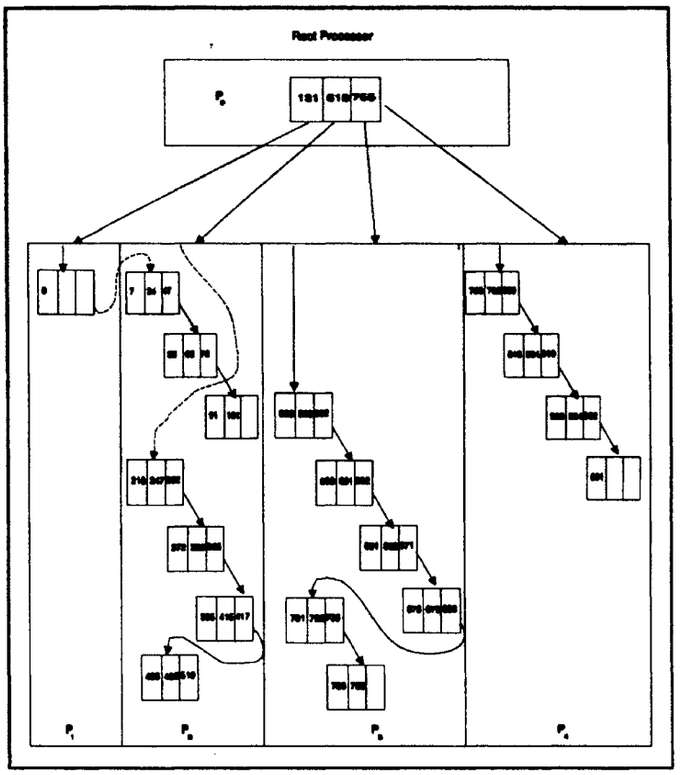
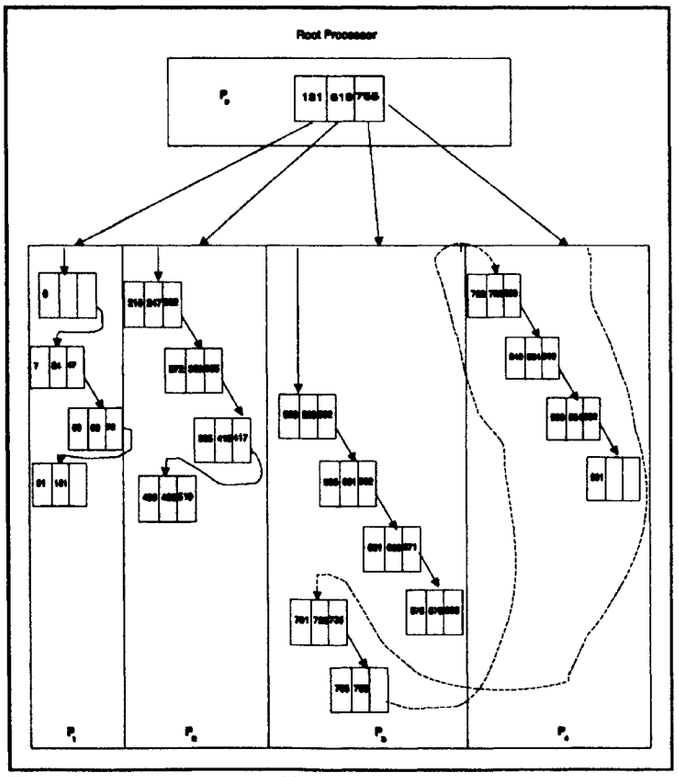


Figure 16 Adjustments for P_2 Sending Nodes to P_1



distribution process (Step 2) is complete. Now each processor may perform the conversion (compression) of its local vine to a balanced tree (Step 3) simultaneously and the entire balancing process will be complete.

Figure 17 Adjustments if P_3 Were to Send Nodes to P_4

Possible Out-of-Balance Detection Schemes

The decision of when a rebalance should take place can be based on a number of factors, depending on how the tree is being used. One possible method that would benefit this parallel implementation is based on the size of the request queues on the *root* processor. If any one processor's request queue becomes a certain percentage larger (i.e. contains a certain percentage more requests) than any other processor's queue, then this indicates that one processor is processing the majority of the requests while the others are relatively idle. This should trigger a rebalance in order to redistribute the keys among the processors to increase the degree of parallelism. Similarly, the decision could be based on the tree size at each processor. The *root* processor could keep a count of the number or keys or nodes at each processor and when one processor had a certain percentage more than any other processor, a rebalance would be triggered.

Alternatively, the decision could be based on space usage. The *root* could maintain a ratio of keys to nodes (keys/nodes) for each processor and when this ratio falls below a certain threshold, a rebalance would be triggered to compact the tree and free the wasted memory in partially filled nodes. The decision could also be based on factors external to the algorithm, such as available processing time or disk space. For example, the tree may be rebalanced during a time of low activity on the machine on which it is implemented (like at night or on weekends), or when the disk space on which it is stored reaches a certain maximum level. In any case, since all requests and replies are routed through the *root* processor, any desired statistics to be used in making this decision can be gathered and maintained there with no additional communication costs.

Integration of Balance Operation with New Tree Search Scheme

In order to implement this rebalancing algorithm as part of this new parallel search tree scheme, some changes must be made to the basic algorithm presented in figure 5. The *root* processor algorithm must be modified to be able to accept a request for a rebalance operation in addition to the insert, delete, or search operations already discussed. When the *root* receives a rebalance request, it must take the following actions.

1. Stop accepting new requests, and place the rebalance request on the tail of each *subroot* processor's queue.

2. Process all the requests that remain in the queues (empty the queues). This includes sending the rebalance request which was the last request queued.

3. While waiting for acknowledgement from each *subroot* processor of completion of the conversion of their respective subtrees to a vine (Step 1 of the rebalance algorithm), calculate the required statistics resulting from equations **1**, **2**, and **3** for each *subroot* processor.

4. Direct *subroot* processors to exchange vine segments as described by the node transfer algorithm of figure 13.

5. Upon completion of the node transfer algorithm, direct all the *subroot* processors to convert their modified vines to a balanced tree (Step 3 of the rebalance algorithm).

6. Wait for each *subroot* processor to acknowledge completion of the conversion of their respective vine to a balanced tree, and then resume normal request servicing.

Modified *Root* Processor Algorithm Figure 18 shows the *root* processor algorithm outline of figure 5 modified for servicing the rebalance request. Figure 19 shows the corresponding *subroot* processor algorithm. There are two new boolean variables introduced, namely Build_Queues and Serve_Queues. Both of these variables are initialized to TRUE.

Root Processor Algorithm

```
While( !Done )
  Begin

    If( Build_Queues )
      Begin
        Get_Incoming_Request( New_Request )
        If( New_Request )
          If( New_Request.Operation != REBALANCE )
            Queue_Incoming_Request( New_Request, PseudoRoot )
          Else
            Begin
              For( i=1 to Number_of_Processors - 1 )
                Queue_Rebalance_Request( Queues[i] )
              Build_Queues = FALSE
            End
      End

      Serve_Queues = FALSE
      For( i=1 to Number_of_Processors - 1 )
        Begin
          If( Queues[i].Waiting )
            Begin
              Serve_Queues = TRUE
              If( Message_Done(Queues[i].Receive_id) )
                Queues[i].waiting = FALSE
            End
          If( !Queues[i].Waiting And !Queues[i].Empty )
            Begin
              Serve_Queues = TRUE
              Request = DeQueue( Queues[i] )
              Queues[i].Send_id = Send(Request,Queues[i].Subroot_Proc_Id)
              Queues[i].Receive_id = Receive(Signal,Queues[i].Subroot_Proc_Id)
              Queues[i].Waiting = TRUE
            End
        End

      If( !Serve_Queues )
        Begin
          For( i=1 to Number_of_Processors - 1 )
            Queues[i].Receive_id = Receive(Signal,Queues[i].Subroot_Proc_Id)
          Calc_Node_Transfer_Stats( Queues )
          For( i=1 to Number_of_Processors - 1 )
            Message_Wait( Queues[i].Receive_id )
          Transfer_Nodes( Queues )
          For( i=1 to Number_of_Processors - 1 )
            Begin
              Queues[i].Send_id = Send(Trans_Complete,Queues[i].Subroot_Proc_Id)
              Queues[i].Receive_id = Receive(Signal,Queues[i].Subroot_Proc_Id)
              Message_Wait( Queues[i].Receive_id )
            End
          Serve_Queues = TRUE
          Build_Queues = TRUE
        End

  End
```

Figure 18 Outline of *Root* Processor Algorithm Modified for Rebalance Request Processing

The algorithm functions the same as described before (see figure 5) as long as Build_Queues and Serve_Queues are TRUE. However, notice that each New_Request is checked to see if it is a REBALANCE request. If it is not, then the request is queued just as before, but if it is a REBALANCE request, the function Queue_Rebalance_Request() is called for each *subroot* processor's queue to place the rebalance request at the tail of the queue. Also, Build_Queues is set to FALSE at this point so that no new requests will be added to the queues until the rebalance is complete.

Now the remaining requests in all of the queues must be processed before the rebalancing algorithm is executed. This is accomplished through the boolean variable Serve_Queues. Immediately before entering the loop that polls to see if any of the *subroot* processor's queues are waiting on a completion response or have a request in their queue that is ready to be sent, Serve_Queues is set to FALSE. If **any** *subroot* processor is found to be waiting on a completion response, or found to have a request waiting to be processed in its queue, then Serve_Queues is set back to TRUE before the loop is exited. In other words, if when the polling loop is exited Serve_Queues has the value of TRUE, then the queues are not yet all completely serviced and they must be checked again.

When the polling loop is exited and Serve_Queues has not been reset to TRUE, then all the requests in all the queues have been sent and completion responses have been received. This means that all of the queues have been flushed and the rebalancing algorithm may now be executed. Recall that the last request queued was the rebalance request. When the *subroot* processors receive this rebalance request,

they each perform the conversion their own local subtree to a vine (Step 1 of the rebalancing algorithm). Upon completion of this conversion, each *subroot* processor sends a signal to the *root* processor to inform the *root* that it is finished with Step 1. So the *root* posts a Receive() call for each of these completion messages. It then calls the function Calc_Node_Transfer_Stats() to calculate the required statistics in preparation for the node transfer step (Step 2) of the rebalancing algorithm. This allows the *root* to calculate all the node transfer statistics while the *subroot* processors perform the conversion to a vine. The *root* processor must now call Message_Wait() for each completion message to ensure that all *subroot* processors are finished with the conversion to a vine (Step 1) before starting the node transfer step (Step 2).

After receiving a completion signal from each of the *subroot* processors for Step 1, the *root* calls the function Transfer_Nodes() to perform the node transfer step (Step 2) of the rebalancing algorithm as described in figure 13. After the node transfer step is complete, the *root* sends a signal to each of the *subroot* processors to inform them that the node transfer step is complete and they may begin the conversion their local vine to a balanced M-way tree (Step 3). Then the *root* must again wait for a completion signal from each of the *subroot* processors by posting a Receive() call and calling Message_Wait() for each message id.

When the *root* receives a completion signal from each *subroot* processor for Step 3, it resets the value of Serve_Queues and Build_Queues to TRUE. This signals the end of the rebalancing process and allows normal request processing to resume once again.

Subroot Processor Algorithm

```
While( !Done )
   Begin
        Receive_id = Receive(Request,Subroot_Proc_id)
        Message_Wait( Receive_id )
        Switch( Request.Operation )
          Begin
            Case : INSERT
                   Insert(Request)
            Case : DELETE
                   Delete(Request)
            Case : ACCESS
                   Search(Request)
            Case : REBALANCE
                   Begin
                     Send_id = Send(Signal,Root_Proc_id)
                     Tree_to_Vine( )
                     Send_id = Send(Signal,Root_Proc_id)
                     Transfer_Nodes( )
                     Receive_id = Receive(Trans_Complete,Subroot_Proc_id)
                     Message_Wait( Receive_id )
                     Vine_to_Tree( )
                   End
          End
        Send_id = Send(Signal,Root_Proc_id)
   End
```

Figure 19 Outline of *Subroot* Processor Algorithm Modified for Rebalance Request Processing

Modified *Subroot* Processor Algorithm The modifications to the *subroot* processor algorithm to facilitate the rebalance request reflect the modifications in the *root* processor algorithm. When the *subroot* processor receives a REBALANCE request, it immediately responds by sending a signal back to the *root* processor to acknowledge receipt of the rebalance request. This signal allows the *root* processor to flush its queues and begin the rebalancing process on its end. The *subroot* processor then calls the function Tree_to_Vine() to convert its local tree to a vine. When the conversion to a vine is complete, the *subroot* processor sends another signal to the *root*

processor to inform the *root* of this. At this point, the *subroot* processor calls the function Transfer_Nodes() to exchange nodes with its neighbor processors as directed by the Transfer_Nodes() function running on the *root* processor. After the *subroot* processor has completed its transfer of nodes, it posts a Receive() call and waits (by using Message_Wait()) for a signal from the *root* processor that all the *subroot* processors have completed the node transfer step. When the *subroot* processor receives this signal from the *root*, it calls Vine_to_Tree() to convert its vine to a balanced M-way tree. Finally, after the conversion of its vine to a balanced tree is complete, the *subroot* processor sends a final signal to the *root* processor which informs the *root* that the rebalancing process on the *subroot* processor is complete and the *subroot* processor is now ready to accept requests once again.

Chapter IV

Results

To evaluate and compare the performance of this new subtree partitioned tree search and maintenance scheme on a current multiprocessor architecture, both the subtree scheme and a Carey and Thompson style pipeline scheme were implemented on a hypercube multiprocessor architecture. Specifically, both schemes were implemented on the Intel iPSC/2™ Hypercube running under the NX/2 operating system. This machine is typical of currently available general purpose distributed memory multiprocessors.

Description of the Hypercube Hardware

The particular iPSC/2 machine on which these two schemes were implemented consists of 32 Intel 80386 processors running at 33 MHz. Each processor has between 4 and 8 megabytes of RAM memory. The processors are connected by bidirectional communication links in a hypercube configuration. This results in each processor having **lg P** connections to **lg P** other processors, where **P** is the total number of processors in the hypercube. An example of 8 processors connected in this fashion is found in figure 7 of Chapter III (page 20). Notice that in order to be connected in a hypercube configuration, the quantity of processors to be used must always be a power of two (2, 4, 8, 16, or 32 in this case). The node labeling in the hypercube is exactly as shown in figure 7 as well. The nodes are labeled such that any two neighbor nodes (nodes connected by a direct link) differ by *exactly* one binary digit (bit) in the binary representation of their numeric label. The links between the nodes are labeled in a similar manner. A link is labeled such that its label is the exclusive-or of the two

nodes it directly connects. To illustrate this, an example of the cube in figure 7 is redrawn in figure 20 with the link labels shown and the node labels expressed as binary digits rather than decimal digits.

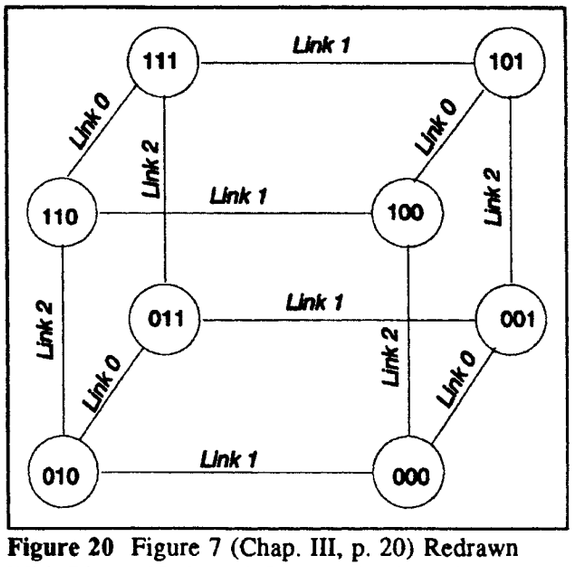


Figure 20 Figure 7 (Chap. III, p. 20) Redrawn With Binary Node and Communication Link Labels

Since the hypercube is a multiple instruction, multiple data (MIMD) machine, each processor executes its own set of instructions on its own set of data completely independently of all the other processors. The actual hypercube machine is front ended by a smaller single processor machine referred to as the *host.* The host is responsible for allocating and loading the programs onto the processors in the hypercube as well as receiving output from the hypercube. While any hypercube processor may send a message to the host, only node 0 has a direct link. All other messages must pass through node 0 and then on to the host.

It is important to realize that each processor in the hypercube may communicate with any other processor in the hypercube, even if they are not connected by a direct link. That is, if node 0 (000) in figure 20 wished to send a message to node 5 (101), it would be able to do so. The message would go from node 0 to node 1 across link 0 and then from node 1 to node 5 across link 2. The path for this message (as well as all other messages) on the hypercube is determined by the NX/2 operating system by using the following rules.

Message Path Determination Rules (Physical)

1. Take the exclusive-or of the sending and receiving node.
   Ex. Send a message from node 0 to node 5. Step 1 gives:

   ```
       000
   XOR 101
       101
   ```

2. The bit positions corresponding to the 1's in the result of the XOR from least-significant-bit to most-significant-bit (MSB $\leftarrow$ LSB) determine the links to follow.
   Ex. Using the XOR result above, Step 2 gives:
   (101) corresponds to links (l_2,l_1,l_0), which means that links l_0 and l_2 (in that order) are to be followed from the sending node (node 0) to the receiving node (node 5), since their corresponding values in the XOR result are 1's.

3. Follow the links in the prescribed order from each node along the path starting with the sending node, and terminating with the receiving node.
   Ex. Following links l_0 and l_2 as prescribed by Step 2 gives:
   Start from node 0, follow link 0 (l_0) to node 1. From node 1, follow link 2 (l_2) to node 5, which is the destination or receiving node.

Since this message routing is done by the NX/2 operating system, it is not of concern to any program running on the iPSC/2 hypercube. All the program must do is request that a message be sent from node *X* to node *Y*. It is important, however, to notice that this routing does guarantee that any node may send or receive a message from any other node on the hypercube, and that this message will only have to cross at most **lg P** links (where **P** is again the number of processors being used).

Message Passing Functions (Software)

Communication between any of the processors is by message passing only. The NX/2 operating system offers both synchronous and asynchronous message passing through calls to C library functions. These messages are passed by one processor issuing a send (by calling a send() function) of a message to some other processor which receives (by calling a recv() function) the message. If the

synchronous receive function (crecv()) is called, program execution will block at this statement until a message arrives. However, if the asynchronous receive function (irecv()) is called, a buffer is established to receive a message and program execution continues immediately. The asynchronous receive function returns a message identification number (*id*) which can be used by the library functions msgdone(*id*) and msgwait(*id*) to determine if the message has arrived yet or block until it does arrive, respectively.

Description of Scheme Implementations

Subtree Scheme Implementation

The new subtree scheme is implemented exactly as described in Chapter III. The *root* processor algorithm is implemented as a C language program which is loaded and run on processor 0 of the hypercube. The *subroot* processor algorithm is also implemented as a C language program and is loaded and run on processors 1 through P-1 (where P is the number of processors in the hypercube). All of the receive function calls in the subtree scheme programs are of the asynchronous type. In the few cases where execution must wait for a message to arrive (such as when a *subroot* processor is waiting for a new request, or during the rebalance process), the msgwait() function is used. Appendix B contains the source code for these programs.

Pipeline Scheme Implementation

The implementation of the Carey and Thompson style pipeline scheme is only a communication model simulation used to gather performance statistics for comparison to the new subtree scheme. No actual tree is maintained in this implementation.

However, since the number, size and synchronization of the messages that must be passed using this scheme are completely determined by the type of operation that is being performed (insert, delete, or access), it is possible to exactly simulate the communication required by this scheme without actually maintaining a tree. As discussed in Chapter III, this scheme is very communication bound. The only processing required at each processor is the search of a single tree node, and perhaps a node split or merge if the operation being performed is an insert or delete. Also, due to the inefficient use of memory, if a tree were actually maintained the scheme would run out of memory after a relatively small number of keys had been inserted. For these reasons, this implementation of the pipeline scheme is only a communication model simulation. These simplifications will only cause the pipeline performance statistics obtained from this implementation to appear an insignificant amount better than they would if an actual tree were maintained. Appendix C contains the source code for these programs.

The pipeline scheme is also implemented as a set of C programs that perform the message passing as described in Chapter III (see figure 2, page 9). It should be mentioned that the hypercube can be (and is for the implementation of this scheme) configured as a linear array of processors such that each processor in the array has a direct connection to its neighbor processor on both sides. This is accomplished by ordering the processors according to a *binary reflective gray code.* A binary reflective gray code will give an ordering of the processors such that for any given label, the previous label and the succeeding label will differ from it in exactly one bit position (thus generating a labeling for a linear array of directly connected processors). See appendix D for how to generate binary reflective gray codes.

Recall from Chapter III (figure 2, page 9) that the communication requirements of the first processor in the linear array (P_0), the next to last processor in the array ($P_{lg\,N}$), and the last processor in the array ($P_{lg\,N+1}$) are slightly different than those for the rest of the processors (P_1 through $P_{lg\,N-1}$). Thus, there is one C program loaded and run on the gray code equivalent of processor 0, another loaded and run on the gray code equivalents of processors 1 through *lgN - 1*, another on the gray code labeling for processor *lgN* and finally a slightly different one on the processor with a gray code label corresponding to *lgN + 1*.

Description of Evaluation Run Test Data

Each scheme was run using random integer keys in the range of 1 to 1,000,000 generated by the random number generator of Park and Miller[21]. Each run was timed (in milliseconds) for both schemes. The runs consisted of the request operation mixes (random orderings of insert, delete, and access requests) shown below.

1. 50% inserts, 50% deletes, 0% accesses (all updates)
2. 0% inserts, 0% deletes, 100% accesses (all accesses)
3. 25% inserts, 25% deletes, 50% accesses (even mix)

Each of these instruction mixes was run with a problem size of 1K, 10K, 20K, 30K, 40K, 50K, 60K, 70K, 80K, 90K, and 100K keys, on 4, 8, 16, and 32 processors. For each run, a tree of the indicated size (1K-100K) is built, and then the same number of operations (for each of the three operation mixes above) are performed on that tree. The resulting execution times of these operations performed after the tree of each given size is built are shown in the following graphs. The results for each instruction mix are presented and then analyzed and explained.

The results for the subtree scheme are presented two ways for each instruction mix. Once as just the time involved in performing the requested operations, and once with the time of the rebalance included as well.

Evaluation Run Results

Results for Operation Mix Number 1 (All Updates)

The graphs shown in figures 21 and 22 represent the execution times for each scheme run with random keys and an operation request mix that consists of all update operations (instruction mix number 1 from page 45). Figure 21 shows the execution times for this mix with each problem size when run with 4 (top) and 8 (bottom) processors. Figure 22 shows the same except as run on 16 (top) and 32 (bottom) processors. The times shown for the subtree scheme in both figures 21 and 22 do **not** include any rebalance time. Figures 23 and 24 show the same information as figures 21 and 22 except that the execution time shown in figures 23 and 24 for the subtree scheme **does** includes the time involved in rebalancing the tree after all of the operations had completed.

As would be expected, the new subtree scheme had considerably better performance for this operation mix. Recall from Chapter III (figure 2), that the pipeline scheme must pass 3 messages between each of the **P** processors in the processor array when performing an update operation (insert or delete), while the subtree scheme must pass only 2 messages (the request and the corresponding reply) between at most **lg P** processors.

Table 1 shows the performance increase (% faster) of the subtree scheme over the pipeline scheme for this operation mix as a percentage resulting from equation 4.

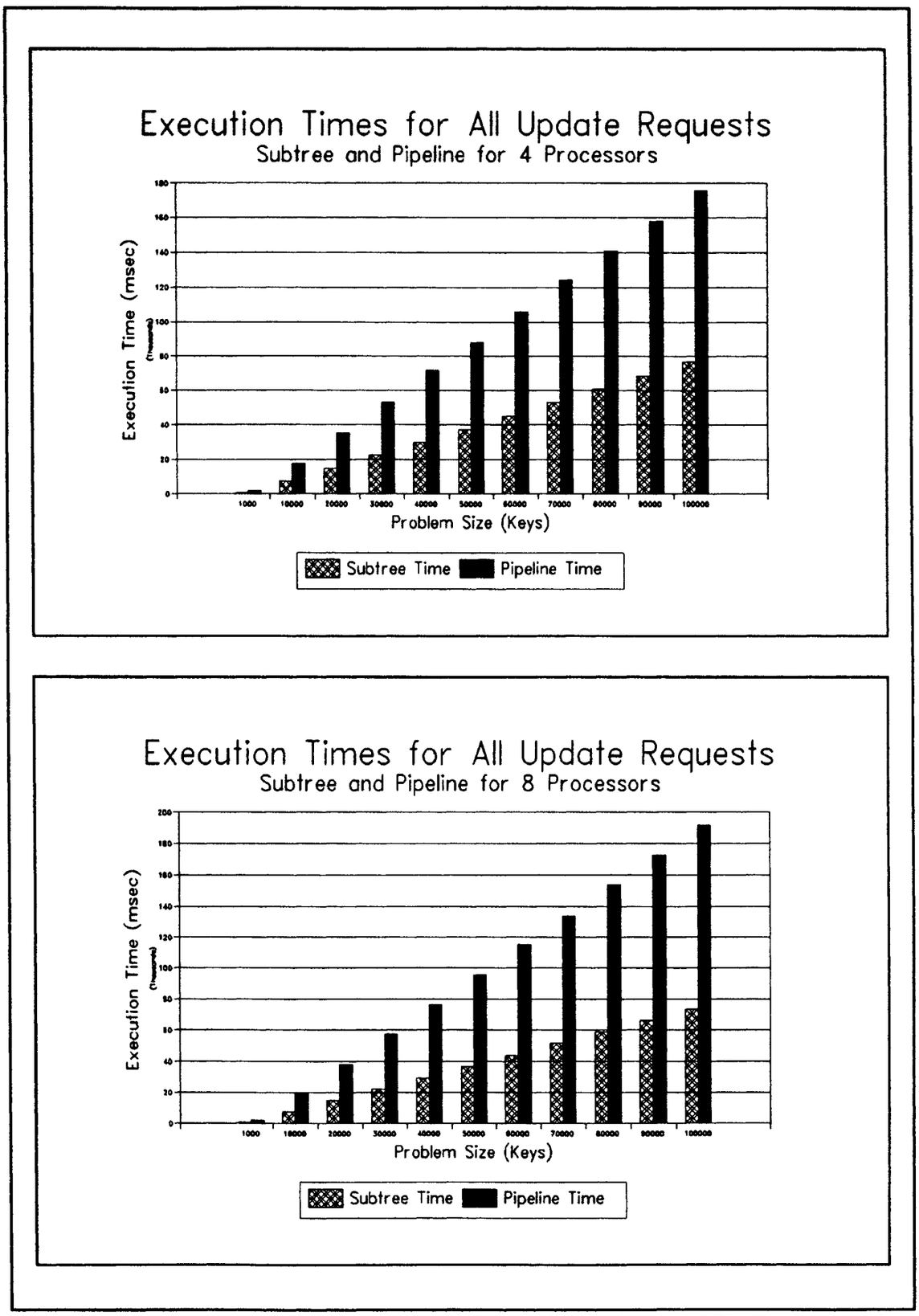


Figure 21 Execution Time vs. Problem Size for the Subtree (no rebalance) and Pipeline Schemes for a Request Mix of 50% Inserts, and 50% Deletes (all updates) As Run With 4 and 8 Processors

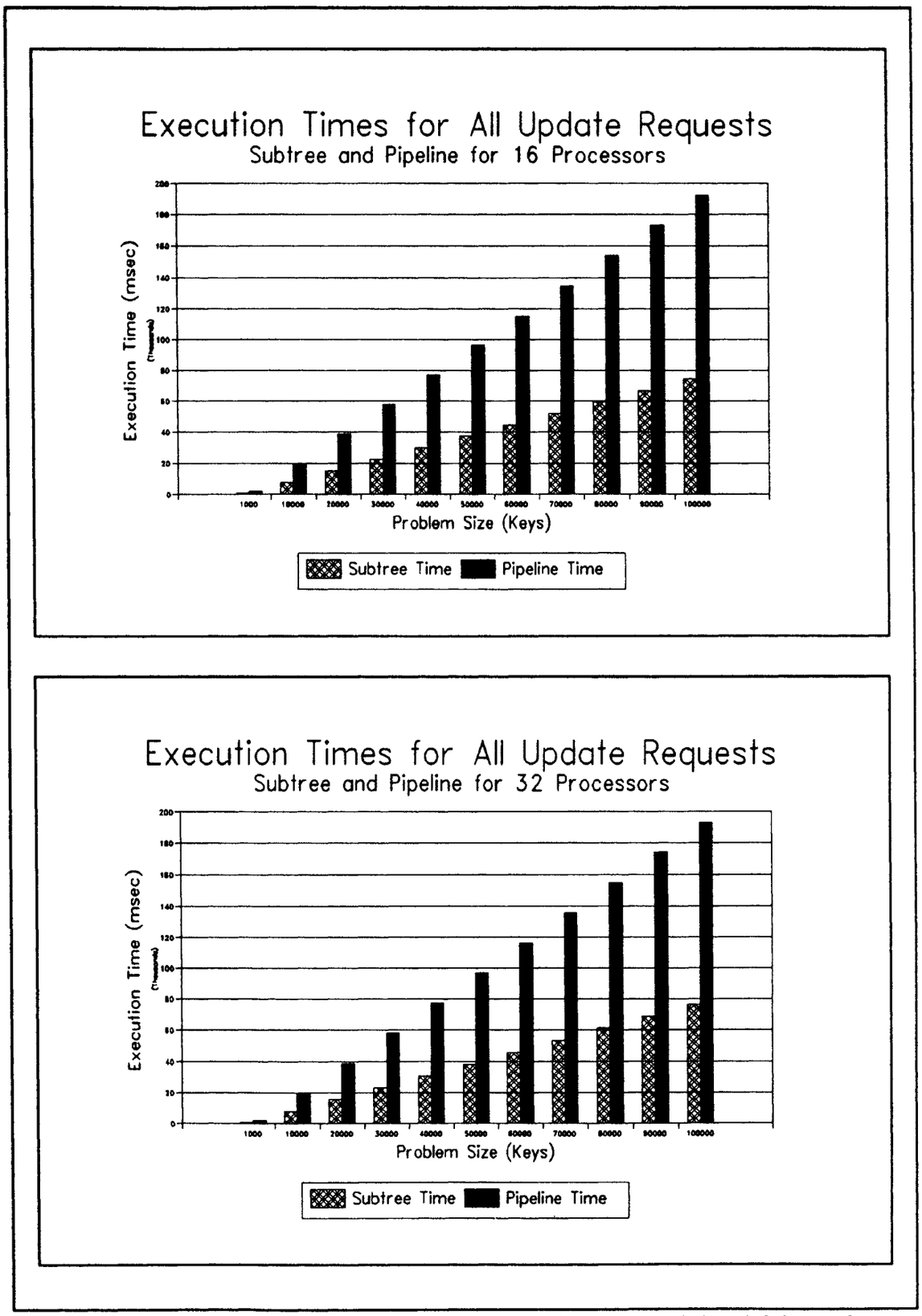


Figure 22 Execution Time vs. Problem Size for Subtree and Pipeline (no rebalance) Schemes for a Request Mix of 50% Inserts, and 50% Deletes (all updates) As Run With 16 and 32 Processors

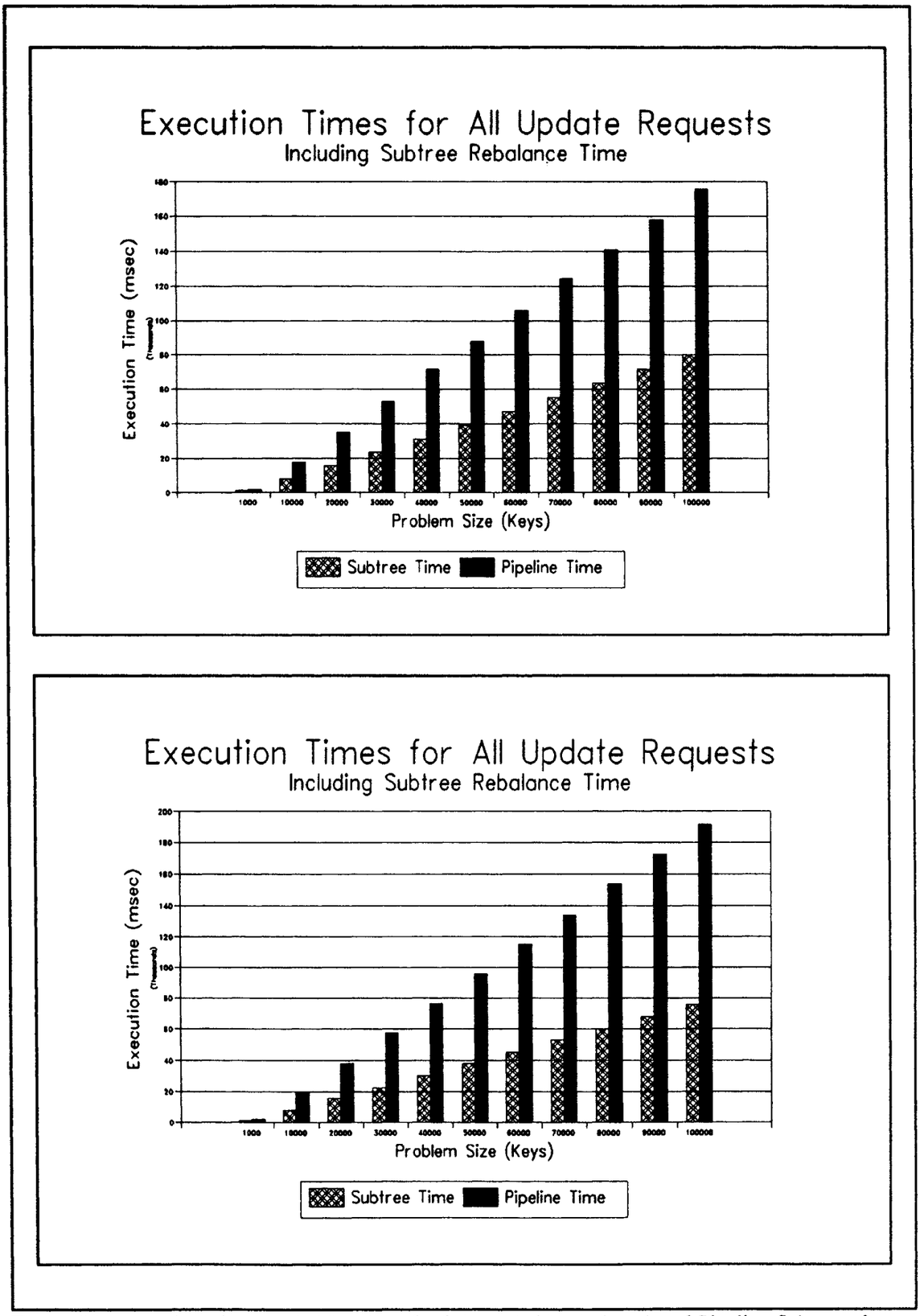


Figure 23 Execution Time vs. Problem Size for Subtree (with rebalance) and Pipeline Schemes for a Request Mix of 50% Inserts, and 50% Deletes (all updates) As Run With 4 and 8 Processors

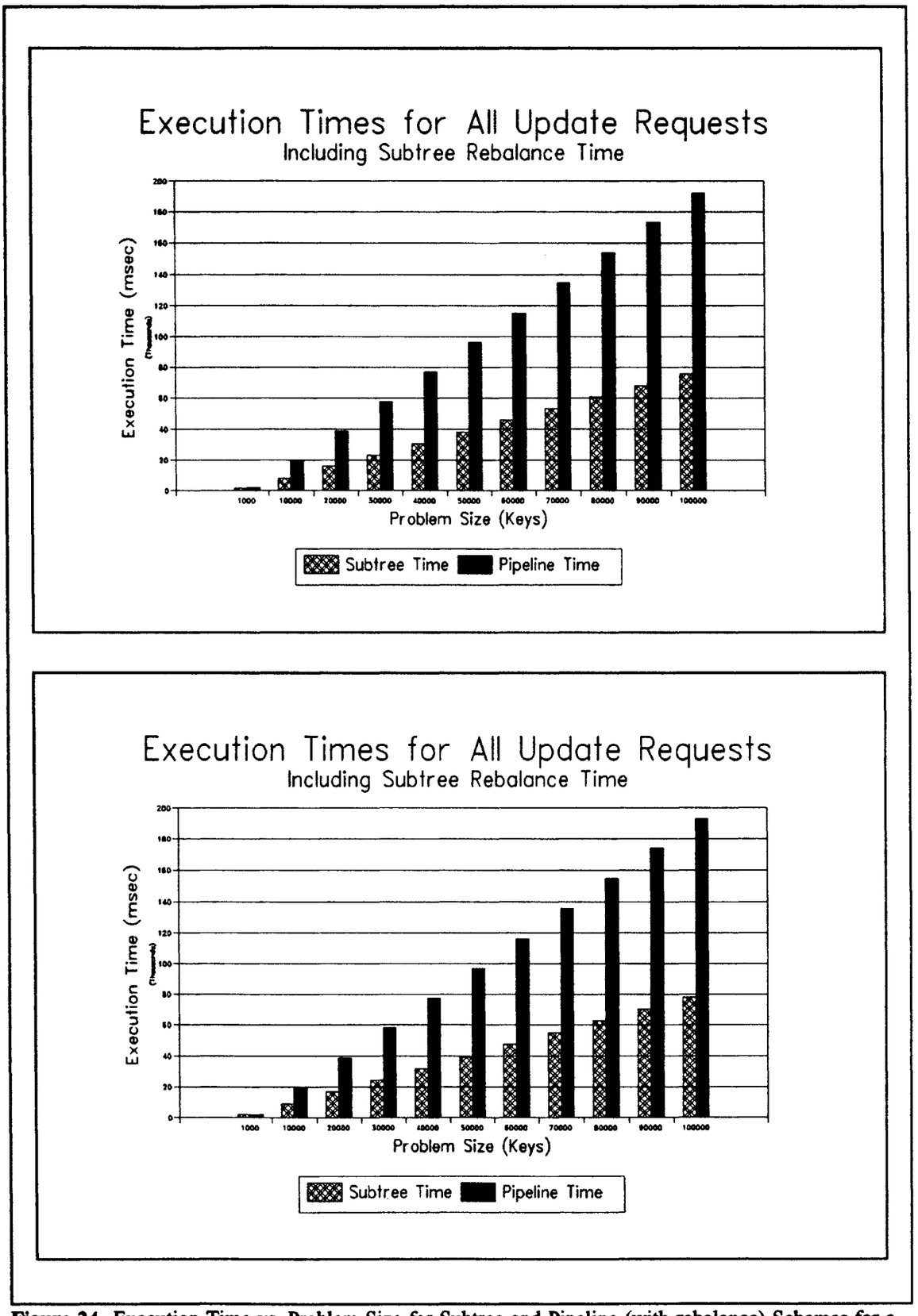


Figure 24 Execution Time vs. Problem Size for Subtree and Pipeline (with rebalance) Schemes for a Request Mix of 50% Inserts, and 50% Deletes (all updates) As Run With 16 and 32 Processors

$$Performance\ Increase = \frac{Execution\ Time\ of\ Pipeline\ Scheme - Execution\ Time\ of\ Subtree\ Scheme}{Execution\ Time\ of\ Pipeline\ Scheme} * 100 \quad (4)$$

The values shown in table 1 are the average of the result of equation 4 for all the problem sizes as run on 4, 8, 16, and 32 processors. These values are shown for the execution times of the subtree scheme with and without rebalancing time included.

Performance Summary for All Update Operation Request Mix

Number of Processors	Percent Faster Not Including Rebalance	Percent Faster Including Rebalance
4	57.7%	55.5%
8	61.7%	60.5%
16	61.4%	60.1%
32	60.5%	58.4%

Table 1 Performance Summary for 50% Insert, 50% Delete Operation Request Mix

Results for Operation Mix Number 2 (All Accesses)

Figures 25 and 26 show the execution times in the same fashion for the operation mix consisting of all access operations and no update operations (operation mix 2 from page 45). Again figures 25 and 26 show the execution times for this mix run on 4 (top) and 8 (bottom) processors, and 16 (top) and 32 (bottom) processors respectively. Since only access operations are performed in this operation mix, the structure of the tree will never be altered and so there is no need to rebalance the tree.

As can be seen from the graphs, the subtree scheme actually has somewhat worse performance (slightly larger execution times) than the pipeline scheme for this operation mix. Recall again from Chapter III that the pipeline scheme must pass only one message between each processor in the processor array for an access request.

This means that for a large series of access operations (such as operation mix 2), the pipeline scheme is capable of performing **P** operations concurrently in a pipeline of **P** processors. However, the message passing required by the subtree scheme for an access operation is exactly the same as for an update (2 messages, one to send the request and one for the corresponding reply). Also, the subtree scheme must devote one processor (the *root* processor) solely to perform the task of queue maintenance and request serving. Thus, the subtree scheme is capable of **P-1** concurrent operations (regardless of whether they are inserts, deletes, or accesses) rather than the **P** concurrent operations possible for the pipeline scheme (with a long string of access requests). Moreover, whether or not the subtree scheme will operate at the full **P-1** degree of concurrency is somewhat dependant on the randomness of the key values being inserted into the tree, while the degree of concurrency attained by the pipeline scheme is strictly dependant on the type of operation being performed, regardless of the key value. For these reasons, the pipeline scheme is actually able to outperform the subtree scheme for very long strings of uninterrupted access requests (such as operation mix 2).

Table 2 shows the performance decrease (percent slower) for the subtree scheme as compared to the pipeline scheme for this operation mix as a percentage resulting from equation 5.

$$Performance\ Decrease = \frac{Execution\ Time\ of\ Subtree\ Scheme - Execution\ Time\ of\ Pipeline\ Scheme}{Execution\ Time\ of\ Subtree\ Scheme} * 100 \qquad (5)$$

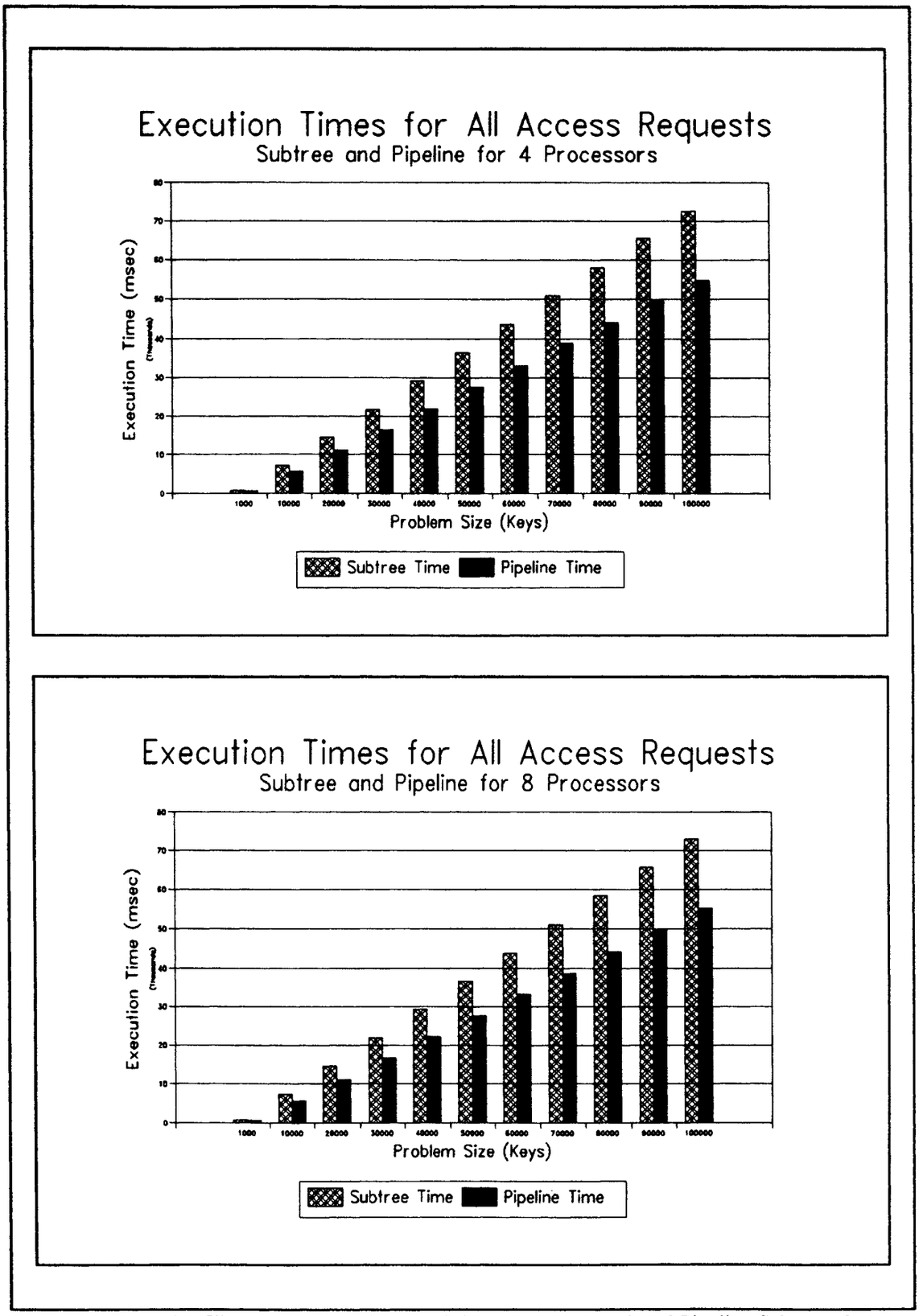


Figure 25 Execution Time vs. Problem Size for Subtree (no rebalance) and Pipeline Schemes for a Request Mix of 100% Accesses, 0% Inserts and Deletes (no updates) As Run On 4 and 8 Processors

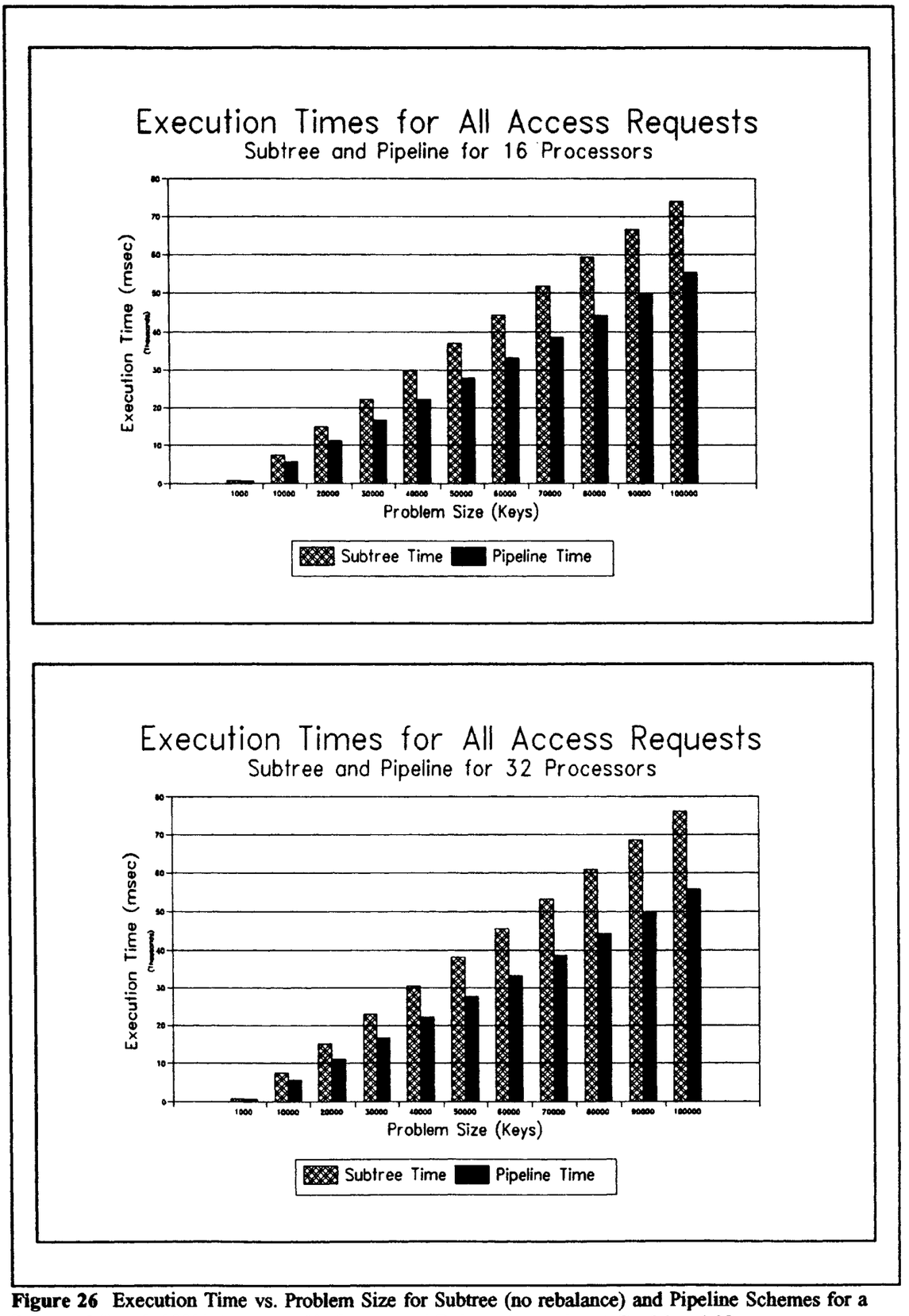


Figure 26 Execution Time vs. Problem Size for Subtree (no rebalance) and Pipeline Schemes for a Request Mix of 100% Accesses, 0% Inserts and Deletes (no updates) On 16 and 32 processors

Performance Summary for All Access Operation Request Mix

Number of Processors	Percent Slower Not Including Rebalance	Percent Slower Including Rebalance
4	24.1%	N/A
8	24.4%	N/A
16	25.1%	N/A
32	27.4%	N/A

Table 2 Performance Summary for 100% Access Operation Request Mix

So far, results have been presented for both extremes of possible request operation mixes. For the case of all update requests (mix 1 from page 45), the subtree scheme yields better performance. However, for the case of all access requests (mix 2 from page 45), the pipeline scheme gives better results. It is of more practical interest to examine an operation request mix composed of an even combination of these two extremes. Operation mix 3 from page 45 is exactly such a combination. It consists of half access operations and half update operations (25% inserts and 25% deletes). This request mix is much more similar to the type of request mix that would be found in real world situations.

Results for Operation Mix Number 3 (Half Updates/Half Accesses)

Figures 27 and 28 show the execution times for this operation request mix without the rebalance time included in the subtree execution times. The subtree execution times shown in figures 29 and 30 include this rebalancing time. The operation requests executed in these runs were generated in a random order. In other words, the runs did not consist of 50% accesses followed by 25% inserts followed by 25% deletes, but rather by a sequence of arbitrary combinations of these operations.

While this sequence of operations is arbitrary, it is guaranteed that is composed of 50% accesses, 25% inserts, and 25% deletes.

As would be expected from the results shown for the other two operation mixes, the performance of the subtree scheme is better than the performance of the pipeline scheme for this operation mix. Recall that the subtree scheme performed considerably better than the pipeline scheme for the update operations, but only slightly worse than the pipeline scheme for the access operations. Thus, for an even mix of these operations it should be expected that the subtree scheme would have better performance than the pipeline scheme. This is verified by the execution time graphs found in figures 27, 28, 29, and 30.

Table 3 shows this performance increase as a percentage (percent faster). As in table 1, the values in table 3 are taken from the average of the result of equation 4 for all the problem sizes on each of the 4, 8, 16, and 32 processor configurations.

Performance Summary for Mixed Update and Access Operation Request Mix

Number of Processors	Percent Faster Not Including Rebalance	Percent Faster Including Rebalance
4	35.8%	33.6%
8	38.8%	37.4%
16	38.7%	36.9%
32	37.4%	34.2%

Table 3 Performance Summary for 50% Access, 25% Insert, 25% Delete Operation Request Mix

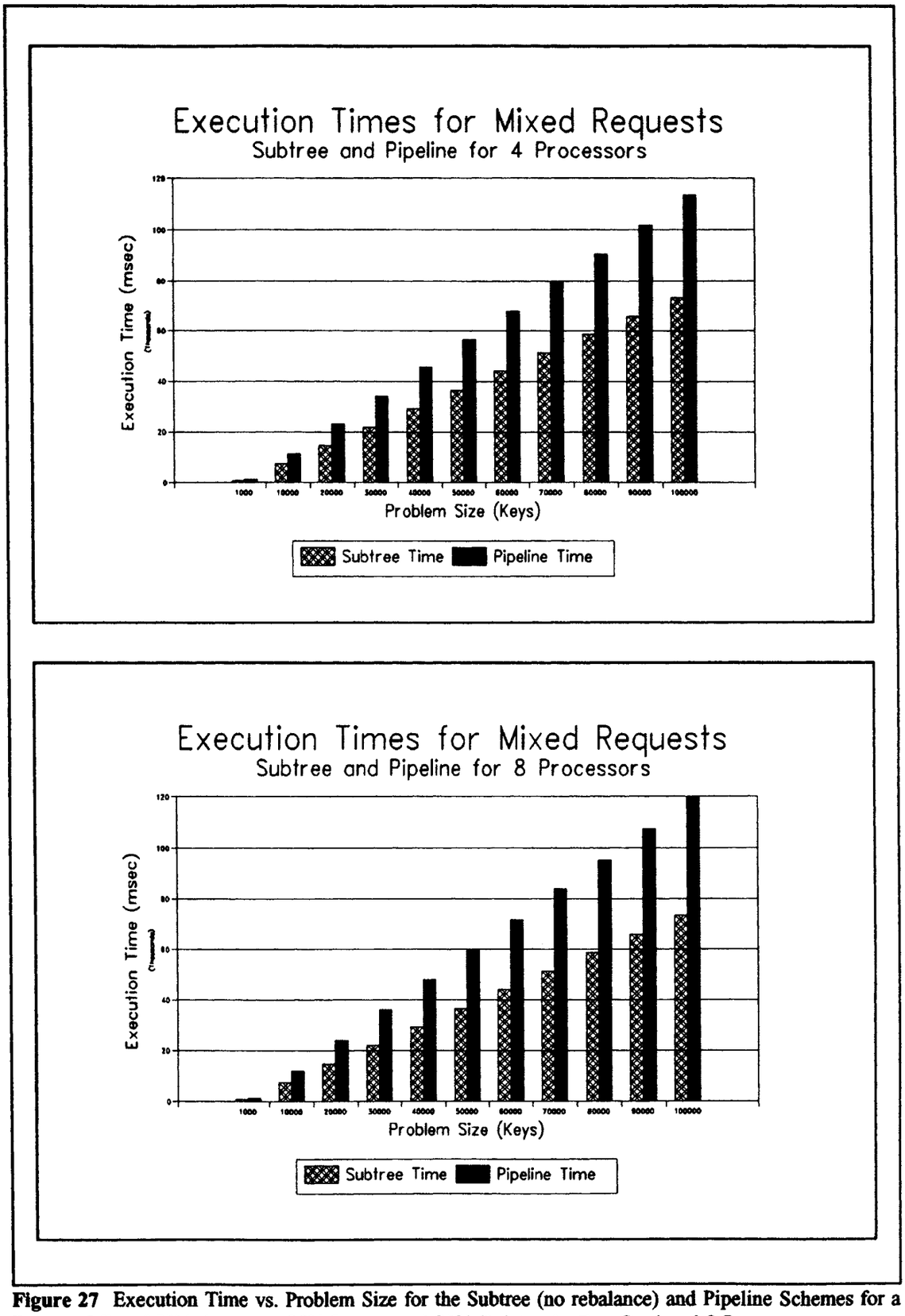


Figure 27 Execution Time vs. Problem Size for the Subtree (no rebalance) and Pipeline Schemes for a Request Mix of 50% Accesses, 25% Inserts, and 25% Deletes As Run On 4 and 8 Processors

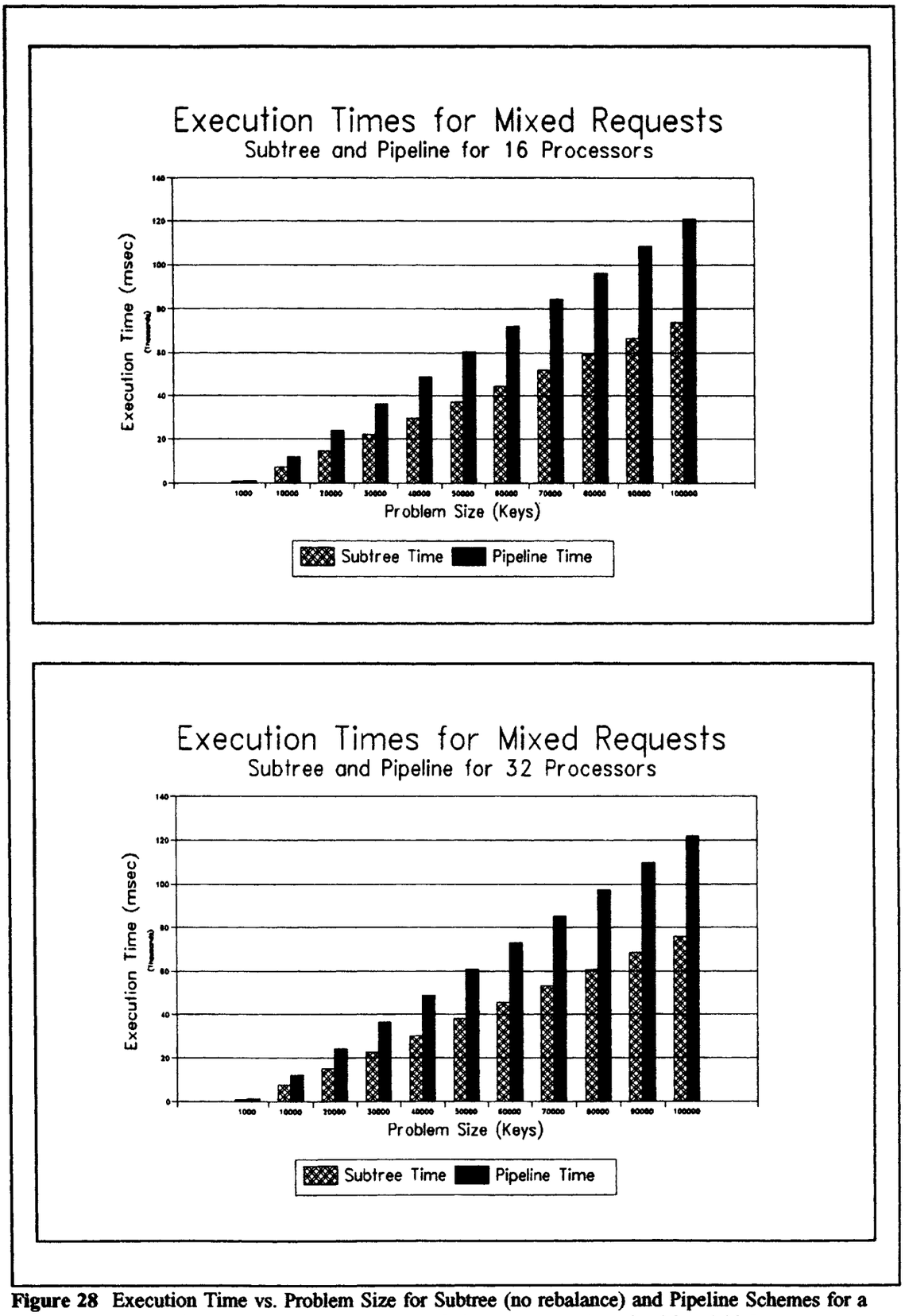


Figure 28 Execution Time vs. Problem Size for Subtree (no rebalance) and Pipeline Schemes for a Request Mix of 50% Accesses, 25% Inserts, and 25% Deletes As Run On 16 and 32 Processors

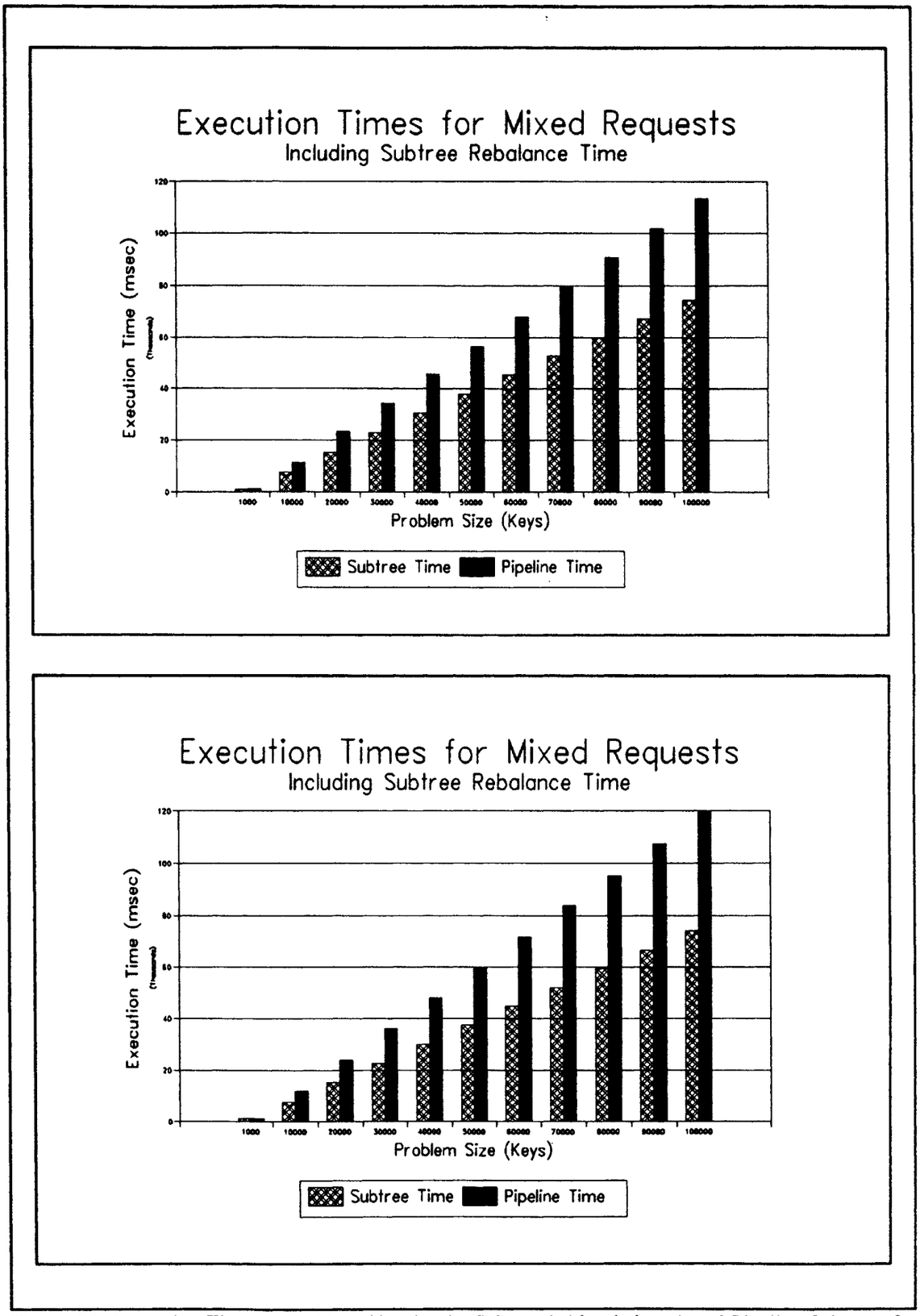


Figure 29 Execution Time vs. Problem Size for the Subtree (with rebalance) and Pipeline Schemes for a Request Mix of 50% Accesses, 25% Inserts, and 25% Deletes As Run On 4 and 8 processors

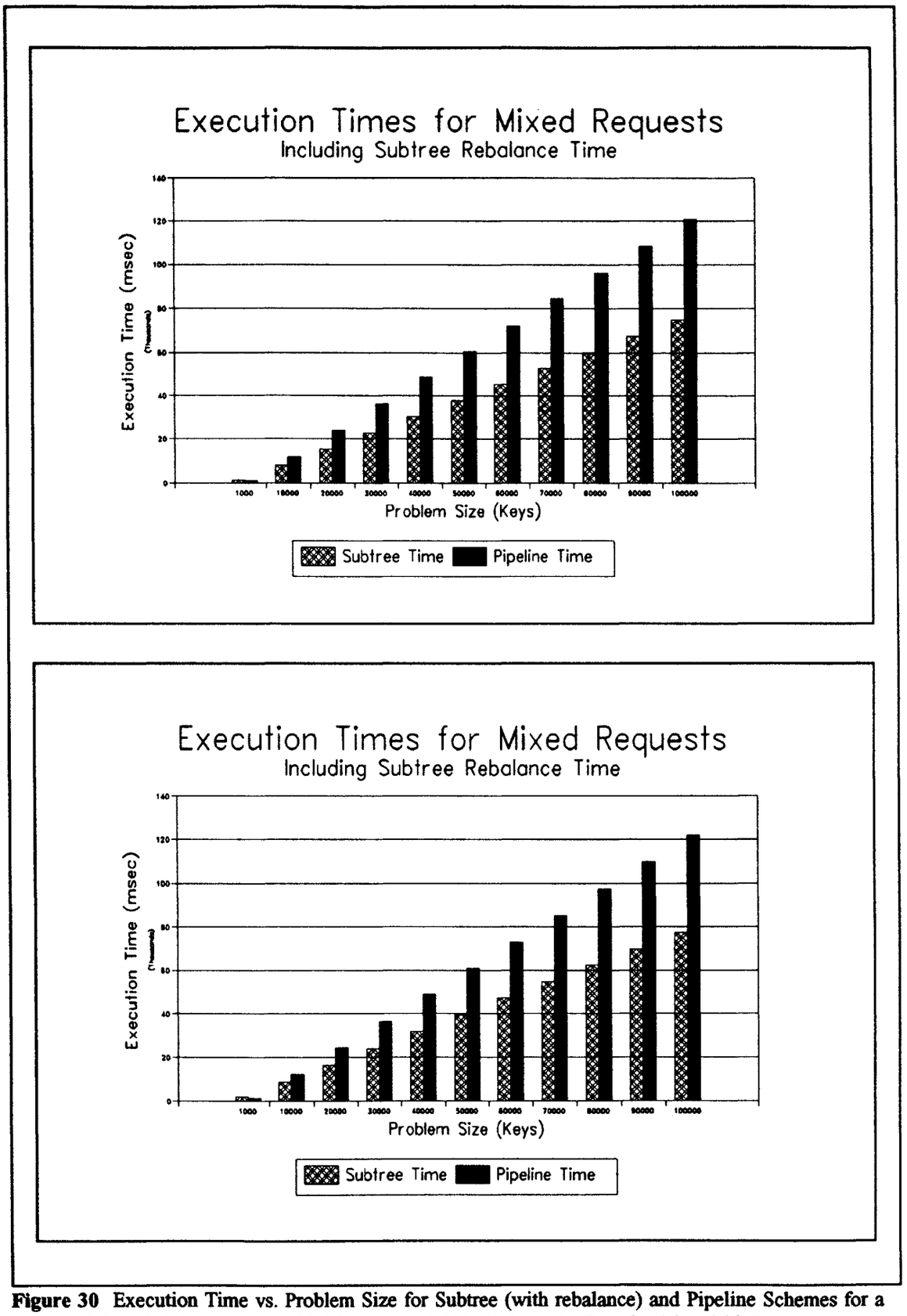


Figure 30 Execution Time vs. Problem Size for Subtree (with rebalance) and Pipeline Schemes for a Request Mix of 50% Accesses, 25% Inserts, and 25% Deletes As Run On 16 and 32 Processors

Discussion of Results

In general, these results show that the pipeline scheme has better performance for request operation mixes that contain long strings of access operations, while the subtree scheme is superior for update operations. For a realistic mix of access and update operations (50% update, 50% access), the subtree scheme remains considerably superior to the pipeline scheme (see table 3).

Notice that the subtree scheme has superior performance over the pipeline scheme even when the time to perform a rebalance is included. In fact, for all the execution times shown, the average time involved in the rebalance operation for each run was only 2% of the total run execution time. Table 4 shows the average time required for rebalancing as a percentage of the total execution time for all the problem sizes and each of the 3 instruction mixes, as run on 4, 8, 16, and 32 processors.

Summary of Time Required for Rebalance Operation For Subtree Scheme

Number of Processors	All Update Request Mix	All Access Request Mix	Mixed Operation Request Mix
4	2.2%	N/A	2.2%
8	1.2%	N/A	1.4%
16	1.3%	N/A	1.8%
32	2.1%	N/A	3.2%

Table 4 Summary of Time Required by the Rebalancing Operation (As a Percentage of Total Execution Time)

The reason for this efficient rebalancing performance is twofold. As previously discussed in Chapter III, the rebalancing algorithm of Smyth is naturally efficient in a parallel environment. Recall that the first stage of the parallel version of the rebalancing algorithm (the conversion of the tree to a vine) and the third stage (the

conversion of the vine to a balanced tree) are done completely in parallel. The second stage (the transfer of vine segments between the processors) is the only sequential portion of the process. The time required by this middle stage is directly proportional to the amount of data that must be transferred between the *subroot* processors. This is determined by how far out of balance the tree has become. Since the keys used in these runs were generated by a random number generator, they are fairly evenly distributed among the *subroot* processors. Thus it should be expected that the performance of the rebalancing operation for these runs would be good, because the runs consist of random data, which naturally helps to control the balance.

Not only is the performance of just the rebalancing portion of the subtree scheme optimal when used with random data, but also the performance of the entire scheme benefits from random data. This is because the performance of the scheme is directly related to the number of operations that are being performed concurrently. Recall that incoming requests are sent to a particular *subroot* processor based on the request's key value as compared to the values in the pseudoroot on the *root* processor. Thus, if the incoming request's key values are random, they will naturally be evenly distributed among the *subroot* processors, which will yield a high degree of concurrency for the scheme.

This does not mean that the subtree scheme will not perform well for data that is not perfectly random. It simply means that the less random (more clustered) the incoming requests are, the more frequently the rebalancing operations will need to occur. Each time a rebalancing operation is performed, not only will the tree be evenly distributed among the *subroot* processors, but also the keys in the pseudoroot

on the *root* processor will be adjusted to reflect this new distribution. Thus, if the incoming data is clustered around certain points, a rebalance should be performed when each cluster is encountered. A cluster could be detected (and a rebalance operation initiated) by any of the out-of-balance detection schemes discussed in Chapter III (page 33).

These results also show that both the subtree scheme and the pipeline scheme take very slightly longer to execute the same problem sizes on the 16 and 32 processor configurations than they do on the 4 and 8 processor configurations. The subtree scheme actually shows a slight improvement in performance (decrease in execution time) in going from 4 to 8 processors, but the pipeline scheme performance peaks with a maximum of 4 processors. Generally, the execution time varied almost insignificantly for both schemes with an increase in the number of processors. This suggests that this machine and architecture is as loosely coupled a system on which either of these schemes could be usefully implemented.

Chapter V

Conclusions

This thesis has described a parallel M-way tree search and maintenance scheme suitable for implementation on distributed memory multiprocessors. This scheme is capable of performing concurrent inserts, deletes, and accesses on an M-way tree distributed among the local memories of an arbitrary number of interconnected independent processors. This scheme also includes a facility to efficiently maintain the global balance of this distributed M-way search tree. This scheme distributes the tree among the processors of a multiprocessor by subtrees of a special root node.

Summary of Subtree and Pipeline Scheme Features

This subtree based distribution allows one processor (the *root* processor) to queue and distribute the incoming requests to the remaining processors (the *subroot* processors). Each of these remaining processors must accept the requests sent to it and perform the requested operations on its local M-way subtree. Thus, the communication required for any given request is only two messages; one to send the request to the appropriate *subroot* processor, and one for that *subroot* processor to acknowledge that the operation is complete and that it is ready to accept a new request. Since each of the *subroot* processors maintains its local M-way subtree completely independently of the others (except during the rebalance operation), the scheme is capable of performing **P-1** operations concurrently, where **P** is the total number of processors being used.

The subtree scheme was compared with the typical pipeline style distributed memory tree search schemes based on the work of Carey and Thompson[16]. In these

schemes, an M-way tree is distributed among the processors in a linear array of processors such that each level of the tree is stored on a separate processor. A top down balancing algorithm is employed by these schemes to maintain the global balance of the tree. A request must pass through all the processors (levels of the tree) being used before the request is complete. All the actual data stored by these schemes is stored in the last (leaf level) processor in the processor array. The advantage of these schemes is that the operations may be partially pipelined. In fact, the access (search) operation may be fully pipelined, allowing up to **P** accesses to be at varying stages of execution at any one time. However, in order to maintain the balance of the tree, additional balance information must be passed between each processor for an update operation (insert or delete), which reduces the average degree of concurrency possible for a typical random mix of operations to **P/2**.

Summary of Subtree Scheme Advantages

The new subtree scheme was found to compare favorably to the pipeline style schemes in several areas. While the pipeline schemes require only one connection per processor (a linear array), they are also incapable of using additional connections if they should exist on the machine on which the scheme is implemented. The subtree scheme, on the other hand, will fully exploit the connectivity of any machine on which it is implemented. Also, since all the data stored in the tree must be stored in the last processor in the array when using the pipeline schemes, the useable memory in the processors that contain the upper levels of the tree will be significantly reduced at each level closer to the root. The subtree scheme, by virtue of its efficient rebalancing algorithm coupled with its subtree based partitioning, is able to fully utilize all the

available memory on the processors. Since most current commercially available general purpose multiprocessors consist of a collection of identical processors connected by an interconnection network with a considerably higher degree of connectivity than a linear array, these advantages of the subtree scheme are very important features.

Summary of Performance Comparison

Most importantly, the subtree scheme offers improved performance over the pipeline style schemes when implemented on a common distributed memory multiprocessor and evaluated with realistic data. Both schemes were implemented and evaluated on the Intel iPSC/2 Hypercube parallel processing computer. The average performance difference as a percentage for three different combinations of insert, delete and access operations and randomly generated keys is shown in table 5 below. The numbers not enclosed by parentheses indicate the performance increase of the subtree scheme over the pipeline scheme; those in parentheses indicate the performance increase of the pipeline scheme over the subtree scheme.

Performance Gain/Loss Summary
For Subtree Scheme As Compared to Pipeline Scheme

Combination of Operations	Performance Gain/Loss No Subtree Rebalance Time Included	Performance Gain/Loss With Subtree Rebalance Time Included
100% Updates, No Accesses	60.3%	58.6%
100% Accesses, No Updates	(25.3)%	N/A
50% Updates, 50% Accesses	37.7%	35.5%

Table 5 Performance Gain (Loss) for Subtree Scheme as Compared to Pipeline scheme

This shows that for the operation combination mix of all updates, the performance of the subtree scheme was far better than that of the pipeline scheme, while the performance of the pipeline scheme was somewhat better than the performance of the subtree scheme for the operation combination mix of all access operations. For a typical realistic operation mix of half updates and half accesses, these results show the subtree scheme to have considerably better performance than the pipeline scheme, even when the time required to do a global rebalancing of the tree is included.

Conclusions

Both schemes ran in $O(\lg N)$ time and exhibited a relatively insignificant performance change when implemented on 4, 8, 16, or 32 processors. Thus, neither of these schemes should be used to attempt to improve the response time of a single query, but rather as a method by which to efficiently manage an extremely large amount of data stored in a tree structure by employing the use of additional processors. Of these two methods, the subtree scheme has been shown to be superior in performance for a typical combination of operation requests with random keys.

The subtree scheme presented in this thesis provides an improved alternative method for distributed memory based parallel M-way tree search and maintenance over the typical pipeline style schemes for a realistic typical combination of operations with random data. Also, this new subtree scheme is capable of fully and efficiently utilizing the available processing power, memory, and interprocessor connectivity of the machine on which it is implemented. This scheme would be an excellent choice for the storage, retrieval, and maintenance of a massive collection of data to be managed on a typical general purpose distributed memory multiprocessor.

Future Work Recommendations

The following is a list of areas in which future work is needed in order to further investigate and improve the new subtree based M-way tree search and maintenance scheme.

1. Develop a scheme to allow more than one access (search) to take place concurrently on individual *subroot* processors.

2. Evaluate and test various out-of-balance detection schemes (such as those discussed in Chapter III).

3. Investigate more efficient queuing and pseudoroot search techniques to run on the *root* processor.

4. Develop a more efficient node transfer step for the rebalancing algorithm to allow some of the transfers to be done in parallel.

REFERENCES

[1] Kung, H.T., Lehman, P.L, "Concurrent manipulation of binary search trees." *ACM Transactions on Database Systems*, 5, 3 (September 1980), 354-382.

[2] Samadi, B., "B-Trees in a system with multiple users." *Information Processing Letters,* 5, 4 (1976), 107-112.

[3] Bayer, R., Schkolnick, M., "Concurrency of operation on B-trees." *Acta Informatica*, (1977), 9, 1-21.

[4] Ellis, C.S., "Concurrent search and insertion in AVL trees." *IEEE Transactions,* C-29, 9 (1980), 811-817.

[5] Ellis, C.S., "Concurrent search and insertion in 2-3 trees.", *Acta Informatica,* (1980), 14, 63-86.

[6] Bently, J.L., Kung, H.T, "A tree machine for searching problems.", *Proceedings of the International Conference on Parallel Processing, IEEE, New York,* 1979.

[7] Song, S.W., "A highly concurrent tree machine for database applications." *Proceedings of the International Conference on Parallel Processing, IEEE, New York,* 1980.

[8] Ottman, T.A., Rosenberg, A.L., Stockmeyer, L.J., "A dictionary machine." *IEEE Transactions*, C-31, 9 (1984), 892-897.

[9] Atallah, M.J, Kosaraju, S.R., "A generalized dictionary machine for VLSI", *IEEE Transactions*, C-34, 2 (1985), 151-155.

[10] Chang, J.H., Ibarra, O.H., Chung, M.J., Rao, K.K., "Systolic tree implementation of data structures." *IEEE Transactions,* C-37, 6 (1988), 727-735.

[11] Bonuccelli, M.A., Lodi, E., Lucio, F., Maestrini, P., Pagli, L., "A VLSI tree machine for relational databases." *Proceeding of the 10th annual ACM International Symposium on Computer Architecture,* (June, 1983), 67-73.

[12] Somani, A.K, Agarwal, V.K., "An unsorted dictionary machine for VLSI.", VLSI design lab, McGill University, Montreal, Canada, 1983.

[13] O'Gorman, R. "The RPA - making the array approach acceptable." Major Advances in Parallel Processing, 130-146.

[14] Tanaka, Y., Nozaka, Y., Masuyama, A., "Pipeline sorting and searching modules as components of a data flow database computer.", *Proceedings of the International Federation for Information Processing,* New-Holland, Amsterdam, 1980, 427-432.

[15] Fisher, A.L., "Dictionary machines with a small number of processors.", *Proceedings of the 11th annual International Symposium on Computer Architecture, IEEE, New York,* 1984, 151-156.

[16] Carey, M.J., Thompson, C.D., "An efficient implementation of search trees on [LgN + 1] processors." *IEEE Transactions,* C-33, 11 (1984), 1038-1041.

[17] Guibas, L.J., Sedgewick, R., "A dichromatic framework for balanced trees." *Proceedings of the 19th annual IEEE Computer Society Symposium on Foundations of Computer Science,* October 1978, 8-21.

[18] Colbrook, A., Smythe, C., "Efficient implementations of search trees on parallel distributed memory architectures." *IEE Proceedings,* 137, 5 (Part E) (Sept. 1990), 394-400.

[19] Smyth, W.F., "Mu-balancing M-way search trees.", *The Computer Journal,* 34, 5 (1991), 406-414.

[20] Stout, Q.F., Warren, B.L., "Tree rebalancing in optimal time and space.", *Communications of the ACM,* 29, 9 (Sept. 1986), 902-908.

[21] Park, S., Miller, K., "Random number generators - good ones are hard to find.", *Communications of the ACM, 31, 5* (May 1988), 857-864.

APPENDIXES

APPENDIX A

The following appendix is a detailed example of the Sequential M-way tree rebalancing algorithm presented by W.F. Smyth [19]. It contains a description of the operations performed by the algorithm and a corresponding example.

1. General Approach of the Rebalancing Algorithm

Several of the special terms that will be used in the following sections are defined below:

N	N is simply the number of nodes in the tree.
M	M is the number of child pointers contained in each node.
MU	MU (μ) is a defined parameter (<M-1) that is the number of keys that will be in each node of the tree after rebalancing.
vine	A vine is an M-way tree or subtree in which every left pointer is NULL.
μ-full	A level of depth *d* of an M-way tree is said to be μ-full if it contains $(\mu+1)^d$ nodes.
μ-balanced	An M-way tree is said to be μ-balanced if and only if level $\lambda+1$ is empty and levels $0,\ldots,\lambda-1$ are μ-full, where $\lambda = \lfloor \log_{\mu+1} \mu N \rfloor$.

The basic idea of the rebalancing algorithm is summarized below.

1. The algorithm takes an ordinary M-way tree and converts it into a vine such that each node in the vine (with the exception of the last (rightmost) node) contains exactly MU keys.

2. This vine created in step 1 is then converted back to an M-way search tree which will be MU-balanced.

Step 1 is accomplished by a procedure called **TREETOVINE** and step 2 is carried out by a procedure called **VINETOTREE**. Each of these procedures (as presented by Smyth), along with their respective support procedures, will be discussed in the following two sections Each will then be explained by making use of an example of its operation. Note that it is only natural to assume that MU should be M-1 (thus each node is utilized as completely as possible), but it is not required. The only requirement is that MU be *less than* **or** *equal* to M-1. In the examples and discussion that follow, however, MU is assumed to be equal to M-1. This section is concluded

```
struct NODE {
            int K;
            struct CHILD {
                int KEY;
                struct node *LEFT;
              } C [M-1];
            struct node *RIGHT;
        };
```

Figure 1. Node Structure to be Used

with a description the node structure that will be used by the discussion and examples that follow.

Figure 1 shows a C-like implementation of the node structure to be used. K is the count of keys in the node. C is an array of M-1 KEY and LEFT pointer pairs. KEY is a numeric key value and LEFT and RIGHT are pointers to child nodes. The pointers C[1].LEFT through C[K].LEFT are pointers to nodes in the tree that contain key values that are less than the value in their own corresponding C[x].KEY field. RIGHT is the right pointer associated with key C[K].KEY.

2.Conversion of a Tree to a Vine

The procedure TREETOVINE is responsible for converting an M-way tree into a vine such that each node in the vine has exactly MU keys. Recall that a vine is simply an M-way tree in which all left pointers (in this case C[1].LEFT through C[M-1].LEFT) are NULL. TREETOVINE accomplishes this conversion by repeatedly performing the *rotation* described by Figure 2. Each time the rotation is performed on a given node U, its leftmost non-null subtree, headed by node V, is moved so that it is pointed to by U's right pointer (U.RIGHT). V's right pointer (V.RIGHT) is then set to point to whatever U's right pointer used to point to (in this example, Z). Then the left pointer in U that used to point to V is set to what V's right pointer pointed to before it was set to point to Z (in this example, C[2].left is set to point to node C). Finally, the left children (keys and corresponding left pointers) of node V (if any exist) are reinserted

into node U between the keys that were originally to the left and right of node V. In this example, this would be to insert the key-pointer pairs from V associated with the keys 21 and 30 into U between the keys 17 and 72. Note that in order for this to work, the keys must be allowed to flow out of node U and down into node V. Conceptually, the child arrays of each node (U and V) can be thought of as one continuous array into which values are deleted and reinserted in the manner just described (each key simply drags its left pointer along with it). Note in Figure 2 that the left pointer associated with 41 (originally the leftmost non-null pointer in U) is reassociated to point to node C before the insertions of the children of V into U take place. While the code to actually perform this rotation is quite complicated, the net effect of the rotation is rather easy to follow. The example in Figure 2 is for a 4-Way tree (M=4), so MU will be M-1 or 3.

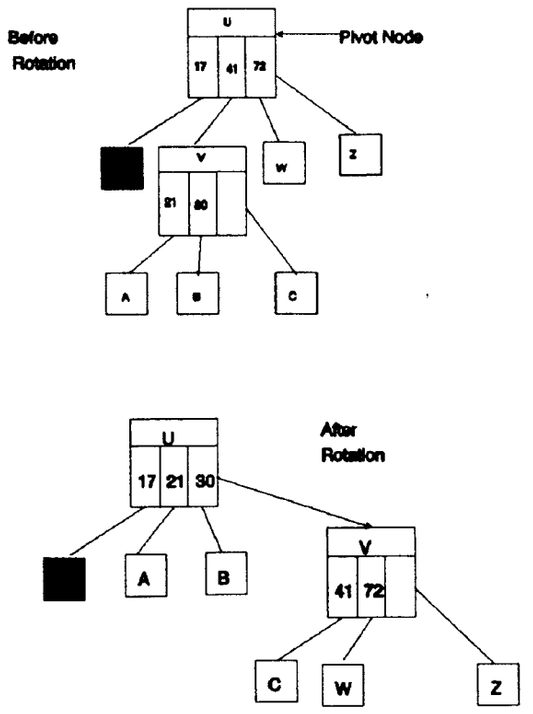
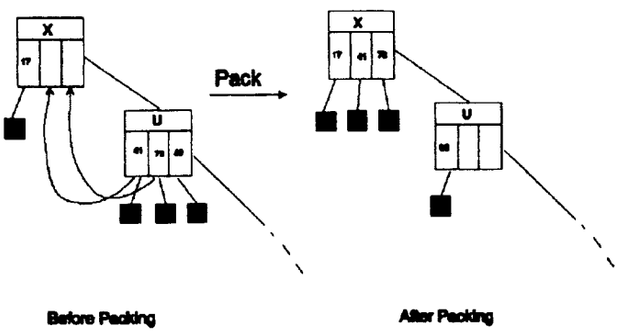


Figure 2 Typical Rotation Step

TREETOVINE continues to apply this rotation to the node U until every left subtree of U has been moved to the right and, as such, every left pointer in U will be NULL. When every left pointer in U has been made NULL by these rotations, the node U is added to the vine (at which point it is designated as node X) and U's right child is designated as the next U node to which rotations will be applied. This is continued until the right pointer of the current U node is NULL. At this point the procedure is finished and the M-way tree has been converted into a vine.

The repeated rotations will in fact create a vine from the nodes in an M-way tree. However, recall that TREETOVINE is responsible for creating this vine in such a way that every node in the vine (except possibly the rightmost) contains exactly MU keys. The rotations do not guarantee this condition. In order to guarantee this condition, the procedure TREETOVINE calls a support procedure PACK just before each new U-node that is ready to be added to the vine is actually added. The procedure PACK then checks the node that was most recently added to the vine (designated X) to see if it contains exactly MU keys. If it does, then PACK does nothing, otherwise PACK will move the required number of keys from the node designated as U into the node designated as X. If this movement of keys causes U to become empty, then U is eliminated, otherwise its count of keys (U->K) is reset and it is added to the vine. An example of the actions of PACK is shown in Figure 3 below.



To further illustrate the conversion of an M-way tree into a vine of nodes with MU keys per node, the following example is presented. In this example, the procedure TREETOVINE is applied to the 4-way tree shown below. Each rotation, and possibly packing, of the nodes is shown until the given tree is completely transformed into the specified vine.

NOTE: Again for this example, MU is assigned to be M-1, so MU=3. A NULL pointer from any nodes key or right side is indicated simply by the absence of a linking line.

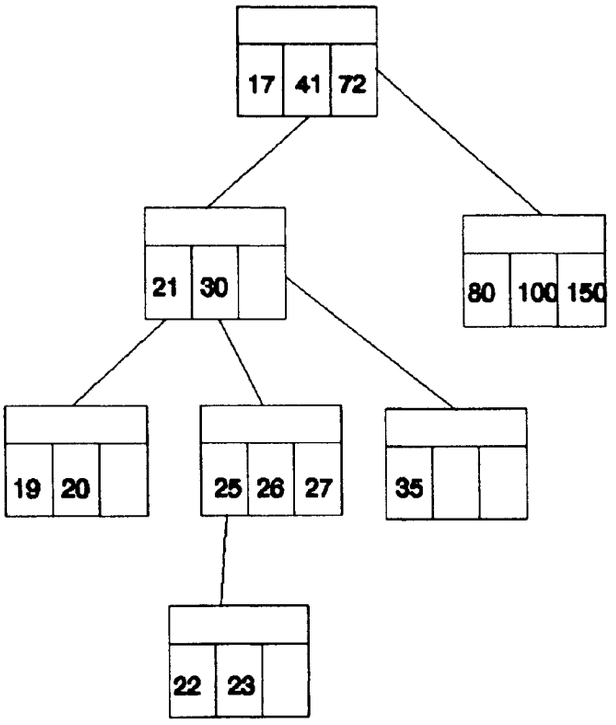
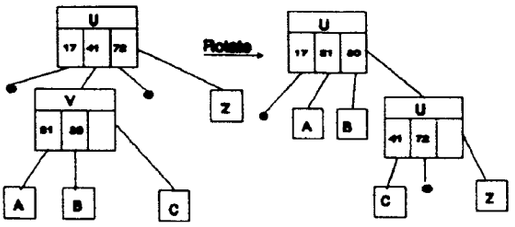
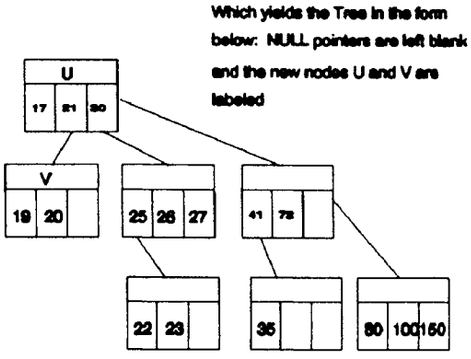


Figure 4 Original Tree to be used in the Example

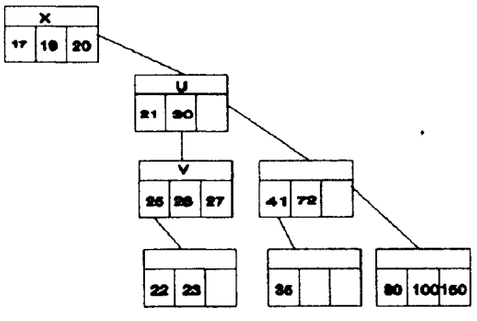
The root and its leftmost non-null child then participate in the original rotation of the

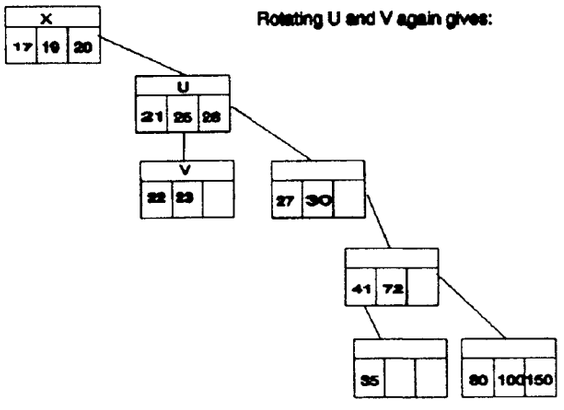
conversion to a vine. This rotation and the resulting tree are shown below.



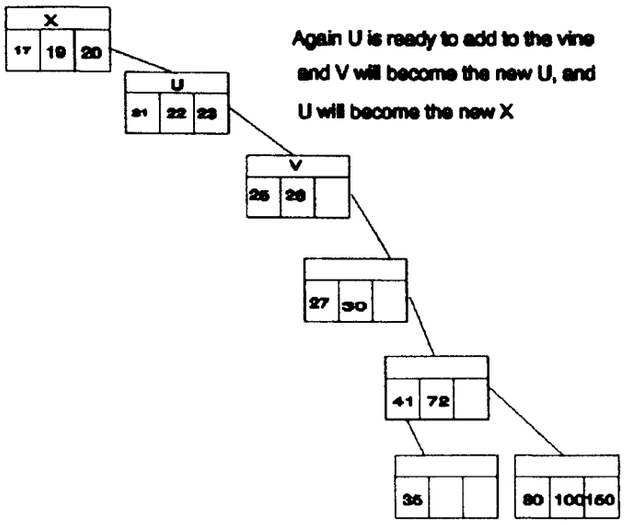


The rotations continue acting on the nodes labeled U and V. Notice that the next rotation makes all of U's left pointers NULL so that it is added to the vine, its right child becomes the next U node and the old U is labeled X. As always, U's leftmost non-null child is labeled V.

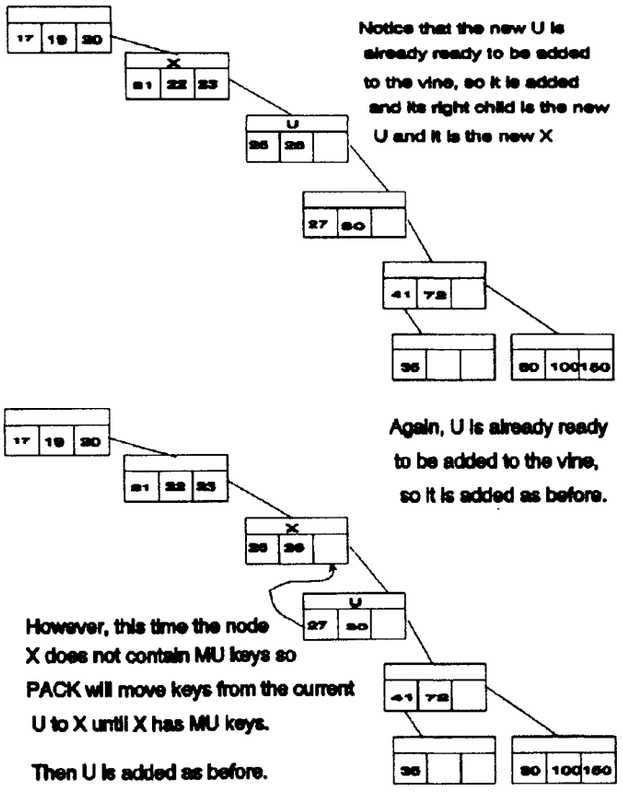




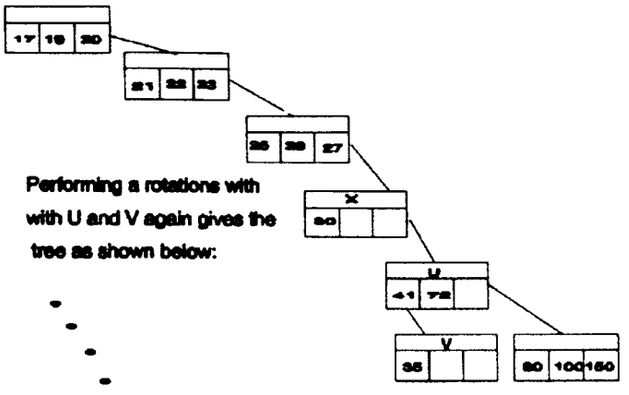
The U-V rotations continue as shown.

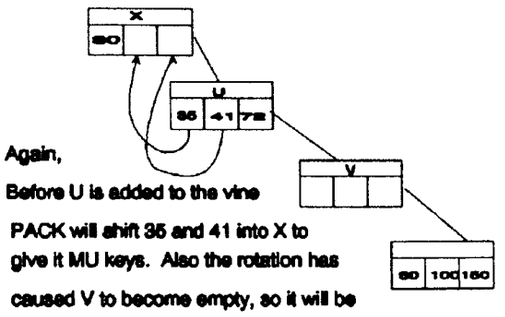


Since the new U already has no left subtrees, it can be added to the vine immediately, as shown below.

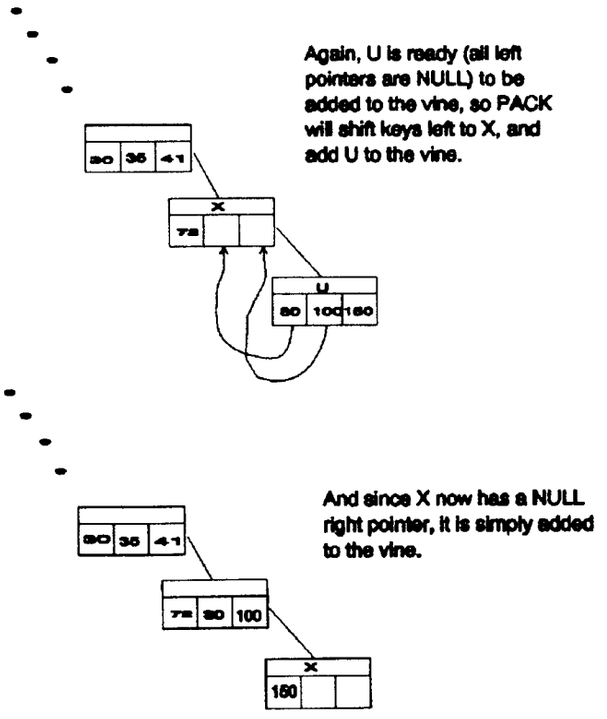


After X is filled to MU keys, the rotations with the new U and V nodes continue as follows.

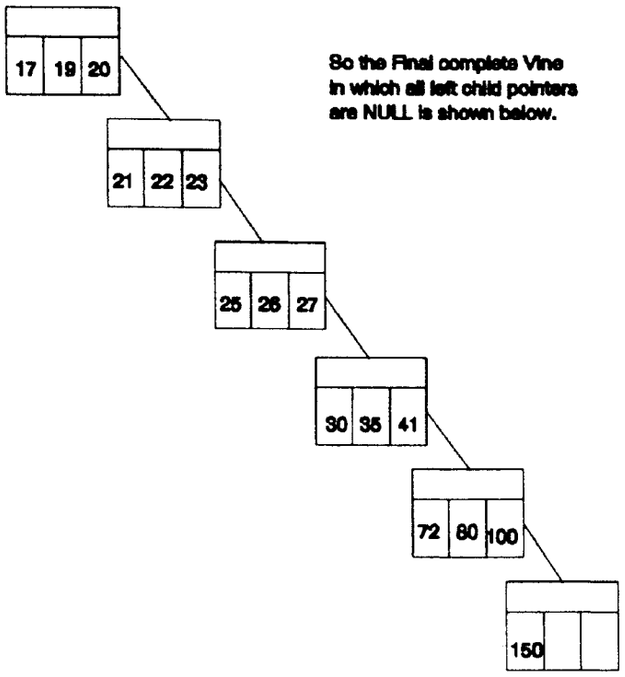

Performing a rotations with
with U and V again gives the
tree as shown below:


Again,
Before U is added to the vine
PACK will shift 35 and 41 into X to
give it MU keys. Also the rotation has
caused V to become empty, so it will be
dropped from the tree (vine).

Continuing:


Again, U is ready (all left
pointers are NULL) to be
added to the vine, so PACK
will shift keys left to X, and
add U to the vine.
And since X now has a NULL
right pointer, it is simply added
to the vine.

And so finally the last node (with NULL right pointer) is added to the vine, and the procedure is complete.



This is now the prescribed vine such that it is an M-way tree in which each node (except possibly the rightmost) has exactly MU keys and all NULL left child pointers. This example was created by simply tracing through the given code for TREETOVINE which can be found in Appendix A.

Note: It should be mentioned at this point that if for some special case, (none of which I can imagine) fewer than M-1 keys/node (MU<M-1) is desired in the final balanced tree, it can be accomplished by a *parceling* of the keys in any node with more than MU keys into itself and a new node. This would be done as each U node is added to the vine. If the current U node to be added has more than MU keys, a new node is allocated into which the extra (U->K-MU) keys are placed. Then, since both nodes will contain only keys with NULL left child pointers (because they came from a U node that was about to be added to the vine) both U and the new node are added to the vine. This is accomplished by procedure PARCEL (see Appendix A), and would only be needed if MU was set to some value less than M-1.

3.Conversion of a Vine back to a MU-balanced Tree

The conversion of the vine back to a MU-balanced tree is accomplished by the procedure VINETOTREE. While the actual code to implement this conversion is much more complicated than that for TREETOVINE, the conceptual actions that result from its execution are actually quite simple and perhaps even easier to follow. VINETOTREE takes the vine that was created by the procedure TREETOVINE and converts it to a MU-balanced M-way tree. The basic idea of VINETOTREE is to take a series of MU+1 nodes in the vine and perform a *compression* on them. The result of this compression is that only one of the MU+1 nodes will remain in the *spine.* A *spine* is simply what was once the vine, but is no longer qualified to be the vine because the compression step has made it such that some nodes now have left pointers that are not NULL. Recall that strictly speaking, a vine consists of nodes such that **every** node has all NULL left pointers.

The actions taken by a compression step of MU+1 nodes in a 4-way tree (MU=3) are shown in Figure 5. Keys are extracted from nodes V, W, and X such that the key from node V is taken from the $(k=1)^{th}$ position, the key from node W is taken from the $(k+1)^{th}$ position and so on until the key in position k=MU (3) is taken from the $(MU+1)^{th}$ node involved in the compression (X). The holes left by these extractions are filled in from the nodes below by shifting the keys, and their corresponding left pointers, left by one for each extraction (just like in the PACK function). Notice that keys are NOT shifted up into the $(MU+1)^{th}$ node (X) since the node below it does not participate in this compression step. After the extraction and related shifting of keys is complete, the last node in the chain (X), which will be the

$(MU+1)^{th}$ node, will be empty. The MU keys that were extracted are then placed, in order, into this empty node, X. The left child pointer of each key placed into X is set to point to the parent node of the node from which it was extracted, and that parent node's right pointer is set to point to whatever the corresponding extracted key's pointer originally pointed to before the compression.

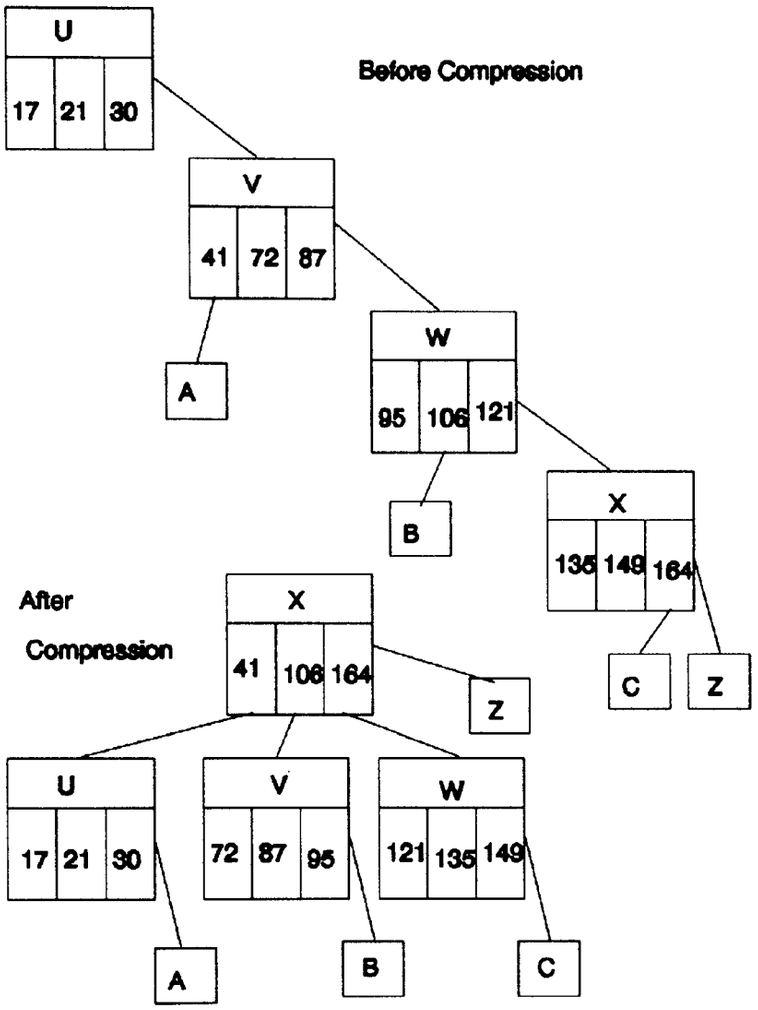


Figure 5 Typical Compression Step

The compression step of Figure 5 is what Smyth refers to as an *ordinary* compression. An *ordinary* compression is simply a compression that consists of MU+1 nodes.

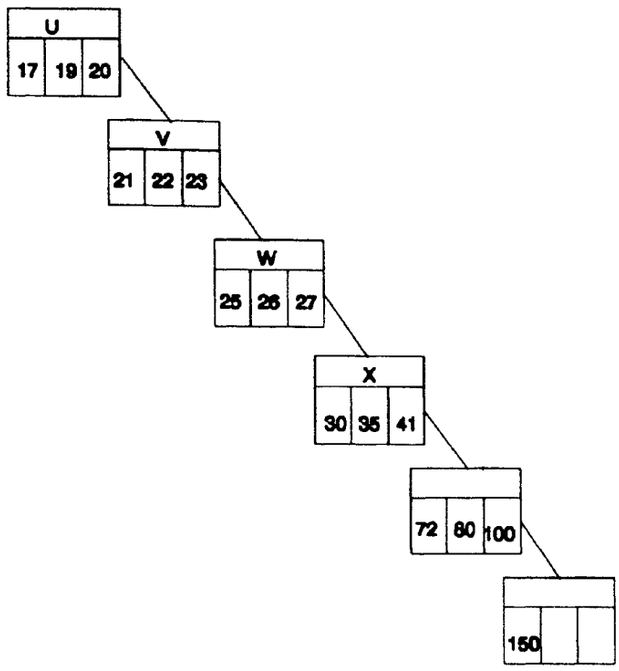
Notice that the nodes compressed in the first compressions of the vine will become the leaf nodes of the lowest level of the final MU-balanced tree. Recall that in order for the final M-way tree to be MU-balanced, every level except this lowest level must be MU-full. In order to guarantee this, the first stage of compressions in VINETOTREE must leave exactly enough nodes for the subsequent compressions to completely fill the upper levels of the tree. Smyth gives two simple equations (of not so simple derivation) to determine (1) the number of 'ordinary' compressions to be carried out, followed by (2) a final 'special' compression of a possibly different number of nodes. Thus this initial stage of compressions in VINETOTREE actually consists of the two distinct parts (1) and (2) shown below:

(1) N_λ ***div*** μ 'ordinary' compressions of $\mu+1$ nodes each, followed by
(2) a 'special' compression of N_λ ***mod*** $\mu+1$ nodes

where $N_\lambda = N - [(\mu+1)^\lambda - 1]/\mu$ [19].

After this initial stage of zero or more 'ordinary' compressions followed by zero or more 'special' compressions, all of the remaining nodes (if any) in the vine may be processed by 'ordinary' compressions like the one shown in figure 5. Finally, when all the compressions are complete, a procedure is called to pull any remaining keys from the rightmost non-root node up into the parent of this node (there will be remaining keys in this node when the last node in the vine is not full, so this will only involve the moving of a maximum of MU-1 keys)

Now VINETOTREE will be applied to the vine created by the example in the previous section to yield a MU-balanced M-way tree (M=4, MU=3). Recall that the final vine was as shown at the top of the next page.



First, VINETOTREE must calculate the number of ordinary compressions it will perform and the number of nodes to be involved in the special compression for the initial stage of compressions. This is done by plugging into equations (1) and (2) where N=6 and MU=3.

Then,

$$\lambda = \lfloor \log_{\mu+1} \mu N \rfloor = ln(\mu N) / ln(\mu+1) = ln(18) / ln(4) = 2$$

so,

$$N_\lambda = N-[(\mu+1)^\lambda-1]/\mu = 6-[(3+1)^2-1]/3 = 1$$

hence, the number of ordinary compressions of $\mu+1$ (or 4) nodes is:

$$\mathbf{N_\lambda \text{ div } \mu = 1 \text{ div } 3 = 0}$$

and the number of nodes in the final special compression is:

$$\mathbf{N_\lambda \text{ mod } \mu+1 = 1 \text{ mod } 4 = 1}$$

Thus the initial stage of compressions consists of no ordinary compressions (since this vine is so small), and a special compression which involves 1 node.

This special compression results in the changes to the original vine shown in Figure 6.

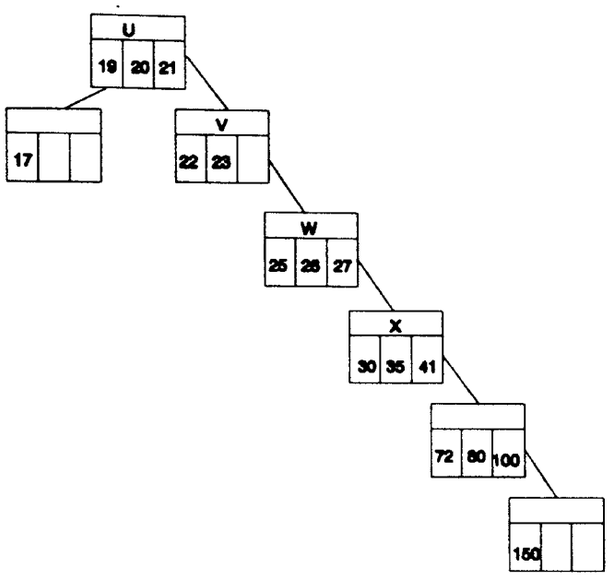


Figure 6 Special Compression Step

The remaining nodes in the spine are then compressed by 'ordinary' compressions just as shown in figure 5. These 'ordinary' compressions continue until there are no longer (MU+1) uncompressed nodes left in the spine to compress. At this point the tree that remains will be MU-balanced. Since there are only 6 nodes in the spine of figure 6, only one 'ordinary' compression of MU+1 (4) nodes will performed. The results of this compression (performed exactly as shown in figure 5, shifting keys up from the left as needed) is shown in below.

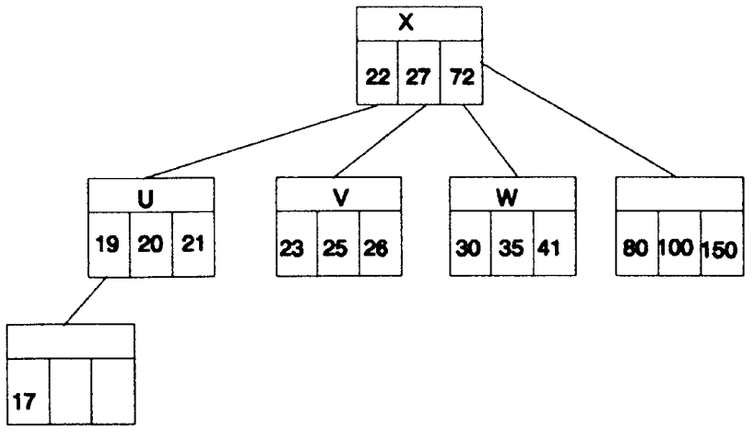


Figure 7 Final Ordinary Compression

This tree is then by definition, a MU-balanced M-way tree (M=4, MU=3) since every level, except the last one, is μ-full.

If there had been more nodes in the vine, the compressions would have continued in like manner. The current node labeled X in the last figure would have become the next U, its right child would become the next V, and this new V's right child would become the next W and so on for the number of nodes calculated to be in the particular compression. VINETOTREE would continue these compressions until there were no longer MU+1 uncompressed nodes in the spine. At that point the final MU-balanced M-way tree would have been created.

APPENDIX B

Subtree Scheme Source Code

Host Machine Program

```
/*****************************************************************************/
/*                                                                           */
/*                          Subtree Scheme Host Source Code                  */
/*                                  March 20, 1994                           */
/*                                                                           */
/*  This is the code that runs on the host computer to drive the subtree scheme.  */
/*  It allocates the cube, loads the nodes with the node code, and deallocates   */
/*  the cube.                                                                */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <cube.h>

#include "locdefs.h"
#include "locstructs.h"

#define UDNUM 2

main()
{
int i,k,j,signal;
int done = FALSE;
int num_procs,prob_size;
struct cnfg config;
char response[30];
char cube_size[30];

 for(k=4; k<=32; k=k*2)
  {
  for(j=1000; j<=100000; j=j+10000)
   {
    if( j == 11000 )
      j = 10000;
    num_procs = k;
    prob_size = j;
    sprintf(cube_size,"%d",(num_procs));
    strcat(cube_size,"sx");
    printf("\n\nWaiting for cube ==> %s prob_size ==> %d\n",cube_size,prob_size);
    done = FALSE;

    while(_getcube("Barney",cube_size,NULL,0,0));

    printf("Cube Allocated ====> %s\n\n\n",cube_size);

    setpid(HOST_PID);
    load("qserv",ROOT,QSERVER);
    for(i=1; i<=num_procs-1; i++)
     load("node",i,SUB_PID);

   printf("*****************************************************************\n");
   printf("*          All nodes Loaded  ====  Execution Begins\n");
   printf("*****************************************************************\n\n");

 crecv(READY,&signal,sizeof(int));
 config.procs = num_procs;
 config.base = prob_size;
 config.udsize = prob_size;
 config.udnum = UDNUM;
 csend(CONFIG,&config,sizeof(config),ROOT,QSERVER);

 while( !done )
  {
   crecv(SIGNAL,&signal,sizeof(int));
   if( signal == EXIT )
    { done = TRUE;
      printf("\nRoot Received EXIT Message from Nodes...Terminating Cube\n\n");
    }
   else if( signal == REBAL )
    printf("\nRebalance Completed ==> No Errors\n\n");
  }

  killcube(ALL_NODES,ALL_PIDS);
 relcube("Barney");
 }
}

}
```

Root Processor Program

```
/*****************************************************************************/
/*                                                                           */
/*                    Subtree Scheme Root Processor Source Code              */
/*                                March 20, 1994                             */
/*                                                                           */
/*  This is the code that runs on the root processor.                        */
/*  This code accepts new requests, queues them to the appropriate subroot   */
/*  processor, and polls to servie these queues.  This operation is explained*/
/*  in Chapters 3 & 4.                                                       */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <cube.h>

#include "locdefs.h"
#include "locstructs.h"

struct q_req *new_q_item();
struct q_head *pick_proc_q();
void enqueue();
void redist();
void dist_stats();
void send_right();
void send_left();
struct req dequeue();
void init_q();
int search_pseudo();
void print_stats();
double rndm();
void rebal();
void print_times();
void print_tot_times();
void mem_error();
void get_request();

struct q_head *Queues;
long int *pseudo_root;
int num_procs;

double seed = 1.0;

unsigned long start_time, stop_time, tot_start_time, tot_stop_time;

int ins_count, del_count, acc_count;

int BASE,UDNUM,UDSIZE;

FILE *qout;

void main()
{
int i,j,root_count,sent,recd,q_count,build,q_min;
int build_queues, serve_queues, signal;
long int rec_id, snd_id, rec_count, req_rep=1;
double range;
struct cnfg config;
struct req new_request;
struct q_head *proc_q;

root_count = 0;
j = 0;
q_count = 0;
sent = 0;
recd = 0;
rec_count = 0;
build = TRUE;
build_queues = TRUE;
serve_queues = TRUE;

csend(READY,&signal,sizeof(signal),myhost(),HOST_PID);

crecv(CONFIG,&config,sizeof(config));
  num_procs = config.procs;
  BASE = config.base;
  UDNUM = config.udnum;
  UDSIZE = config.udsize;

printf("Executing cube ==> %dsx   Problem Size ==> %d at %d, %d\n",num_procs,
```

```
        UDNUM,BASE,UDSIZE);

Queues = (struct q_head *)calloc( num_procs,sizeof(struct q_head) );
pseudo_root = (long int *)calloc( num_procs-1,sizeof( long int ) );

ins_count = del_count = acc_count = 0;

q_min = num_procs;

init_q( Queues );

for(i=1; i<=PRIME; i++)
 rndm();

for(i=1; i<=num_procs-1; i++)
  crecv(READY,&signal,sizeof(signal));

for( ; ; )
{

if( (build_queues)  )
{
 get_request( &new_request, &rec_count, &build );

 if( new_request.op == SPECIAL )
  {
   build_queues = FALSE;
   if( new_request.key == REBAL )
    {
     new_request.op = REBAL;
     for(i=1; i<=num_procs-1; i++)
       enqueue( &Queues[i],new_request );
    }
   else if( new_request.key == EXIT )
     new_request.op = EXIT;
  }
 else
  {
   proc_q = pick_proc_q( new_request,&root_count );
   if( proc_q != NULL )
    { enqueue( proc_q,new_request );
      q_count++;
    }
   else
    {
      /* printf("Error on Enqueue for key = %d  op = %d\n",new_request.key,
              new_request.op); */
    }
  }
}

if( (q_count >= q_min) || (!build_queues) )
{
serve_queues = FALSE;
for(i=1; i<=num_procs-1; i++)
 {
    if( Queues[i].waiting )
     {
       Queues[i].waiting = (((msgdone(Queues[i].rid)) == TRUE) ? FALSE : TRUE);
       if( !(Queues[i].waiting) )
             { msgdone(Queues[i].sid);
               recd++;
             }
       serve_queues = TRUE;
     }
    if( (!(Queues[i].waiting)) && (!(Queues[i].empty)) )
     {
       Queues[i].act_req = dequeue( &Queues[i] );
       Queues[i].sid = isend(Queues[i].proc_req,&Queues[i].act_req,
                             sizeof(struct req),Queues[i].proc_num,SUB_PID);
       Queues[i].rid = irecv(Queues[i].proc_rep,&Queues[i].act_rep,
                             sizeof(int));
       Queues[i].waiting = TRUE;
       sent++;
       q_count--;
       serve_queues = TRUE;
     }
 }
}
```

```
if( !build_queues && !serve_queues )
 {
 stop_time = mclock();
 printf("\nUpdate %d Complete\n",j);
  if( new_request.op == REBAL )
   {
    print_times( j, "Operations" );
    /* printf("Inserts = %d  Deletes = %d  Accesses = %d\n",
           ins_count,del_count,acc_count); */
     ins_count = del_count = acc_count = 0;
    /* for(i=1; i<=num_procs-1; i++)
      { printf("Queue %d has max_count = %d\n",i,Queues[i].max_count);
        Queues[i].max_count = 0;
      } */
    redist(&root_count);
    for(i=1; i<=num_procs-1; i++)
     {
       rec_id = irecv(100+i,&req_rep,sizeof(req_rep));
       msgwait( rec_id );
     }
    /* printf("q_count = %d     sent = %d    recd = %d\n",q_count,sent,recd); */
    tot_stop_time = mclock();
    print_tot_times( j, "Total Time" );
    sent = 0;
    q_count = 0;
    recd = 0;
    j++;
    signal = REBAL;
    snd_id = isend(SIGNAL,&signal,sizeof(int),myhost(),HOST_PID);
    build_queues = TRUE;
    msgdone( snd_id );
   }
  else if( new_request.op == EXIT )
   {
    signal = EXIT;
    csend(SIGNAL,&signal,sizeof(int),myhost(),HOST_PID);
    exit(0);
   }
 }

}   /* Close Endless For  */

}  /* End Main  */


struct q_head *pick_proc_q( new_request,root_count )
struct req new_request;
int *root_count;
{
int search_val,pos;

   search_val = search_pseudo( new_request.key,&pos,*root_count );

   if( search_val == LEFT )
     return( &Queues[pos] );
   if( search_val == FOUND )
       return( &Queues[pos] );
   if( search_val == HERE )
     {
       pseudo_root[pos] = new_request.key;
       (*root_count)++;
       return( &Queues[pos] );
     }
   else  /* if( search_val == RIGHT ) */
     return( &Queues[num_procs-1] );

}

void enqueue( proc_q,new_request )
struct q_head *proc_q;
struct req new_request;
{

   if( proc_q->empty )
    {
      proc_q->front = new_q_item();
      proc_q->front->request = new_request;
      proc_q->front->next = NULL;
      proc_q->tail = proc_q->front;
      proc_q->empty = FALSE;
    }
```

```
   else
    {
     proc_q->tail->next = new_q_item();
     proc_q->tail->next->request = new_request;
     proc_q->tail->next->next = NULL;
     proc_q->tail = proc_q->tail->next;
    }
  (proc_q->count)++;

  if( proc_q->count > proc_q->max_count )
    proc_q->max_count = proc_q->count;
}


struct req dequeue( proc_q )
struct q_head *proc_q;
{
struct q_req *temp;
struct req act_req;

temp = proc_q->front;
proc_q->front = temp->next;
act_req = temp->request;
(proc_q->count)--;

if( proc_q->count == 0 )
  proc_q->empty = TRUE;

free( temp );

return( act_req );

}

void init_q( Queues )
struct q_head *Queues;
{
int i;

  for(i=0; i<num_procs; i++)
    {
      Queues[i].empty = TRUE;
      Queues[i].waiting = FALSE;
      Queues[i].front = NULL;
      Queues[i].tail = NULL;
      Queues[i].count = 0;
      Queues[i].max_count = 0;
      Queues[i].proc_num = i;
      Queues[i].proc_req = 100 + i;
      Queues[i].proc_rep = 100 + i;
    }

}

struct q_req *new_q_item()
{
struct q_req *local_req;

local_req = (struct q_req *)malloc( sizeof(struct q_req) );
if( !local_req )
 {  printf("Memory allocation error in new_q_item function\n");
    exit(1);
 }
return( local_req );
}


int search_pseudo( key,pos,root_count )
long int key;
int *pos, root_count;
{
int temp;

temp = (root_count/2) + 1;
if( key < pseudo_root[temp] )
  temp = 1;

 for(*pos=temp; *pos<=root_count; (*pos)++)
  {
   if( key < pseudo_root[*pos] )
     return( LEFT );
   if( key == pseudo_root[*pos] )
```

```
    return( FOUND );
  }
 if( root_count < num_procs-2 )
   return( HERE );
 else
   return( RIGHT );
}


void redist(root_count)
int *root_count;
{
int i,nodes_per_proc,total_nodes,cp;
struct t_head *tree_heads;
struct t_head temp;

total_nodes = 0;

tree_heads = (struct t_head *)calloc(num_procs,sizeof(struct t_head));

for(i=1; i<=num_procs-1; i++)
 {
  crecv(BAL_STATS,&temp,sizeof(temp));
  tree_heads[(temp.proc_num)] = temp;
  total_nodes = total_nodes + (temp.nodes);
 }

nodes_per_proc = total_nodes / (num_procs-1);

for(i=1; i<=num_procs-1; i++)
  tree_heads[i].trans_info.nodes_per_proc = nodes_per_proc;

dist_stats(tree_heads,nodes_per_proc);

if( STATS )
{
for(i=1; i<=num_procs-1; i++)
   print_stats(&tree_heads[i],i);
}

for(i=1; i<num_procs-1; )
 {
  if( tree_heads[i].give > 0 )
   {
    send_right(tree_heads,i,nodes_per_proc);
    i++;
   }
  else
   {
    cp = i;
    while( tree_heads[cp].requests > 0 )
      cp++;
    send_left(tree_heads,cp,i,nodes_per_proc);
    if( i == cp )
      cp++;
    i = cp;
   }
 }
temp.trans_info.op = EXIT;

for(i=1; i<=num_procs-1; i++)
  csend(BAL_STATS,&temp,sizeof(struct t_head),i,SUB_PID);

(*root_count) = 0;
for(i=1; i<=num_procs-2; i++)
  { pseudo_root[i] = tree_heads[i].max_key;
    if( tree_heads[i].nodes > 0 )
       (*root_count)++;
    /* printf("psuedo_root[%d] = %ld\n",i,tree_heads[i].max_key); */
  }
free( tree_heads );
}


void send_right(tree_heads,from,nodes_per_proc)
struct t_head *tree_heads;
int from,nodes_per_proc;
{
int i,to;
struct trans trans_info;
struct t_head from_proc, to_proc;
```

```
from_proc = tree_heads[from];
to_proc = tree_heads[from+1];
to = from + 1;

trans_info.from = from;
trans_info.to = from + 1;
trans_info.direction = RIGHT;
trans_info.nodes_per_proc = nodes_per_proc;

trans_info.op = ACCEPT;
tree_heads[to].trans_info = trans_info;
csend(BAL_STATS,&tree_heads[to],sizeof(struct t_head),to,SUB_PID);

trans_info.op = GIVE;
tree_heads[from].trans_info = trans_info;
csend(BAL_STATS,&tree_heads[from],sizeof(struct t_head),from,SUB_PID);

crecv(BAL_SND,&tree_heads[from],sizeof(struct t_head));

crecv(BAL_REC,&tree_heads[to],sizeof(struct t_head));

}



void send_left(tree_heads,from,to,nodes_per_proc)
struct t_head *tree_heads;
int from,to,nodes_per_proc;
{
int i,inter_to;
struct trans trans_info;
struct t_head from_proc, to_proc;

while( from > to )
 {
  inter_to = from - 1;

   from_proc = tree_heads[from];
   to_proc = tree_heads[inter_to];

   trans_info.from = from;
   trans_info.to = inter_to;
   trans_info.nodes_per_proc = nodes_per_proc;
   trans_info.direction = LEFT;

   trans_info.op = ACCEPT;
   tree_heads[inter_to].trans_info = trans_info;
  csend(BAL_STATS,&tree_heads[inter_to],sizeof(struct t_head),inter_to,SUB_PID);

   trans_info.op = GIVE;
   tree_heads[from].trans_info = trans_info;
   csend(BAL_STATS,&tree_heads[from],sizeof(struct t_head),from,SUB_PID);

   crecv(BAL_SND,&tree_heads[from],sizeof(struct t_head));

   crecv(BAL_REC,&tree_heads[inter_to],sizeof(struct t_head));

 from = from - 1;
 }
}

void dist_stats(tree_heads,nodes_per_proc)
struct t_head *tree_heads;
int nodes_per_proc;
{
int i=1, given_from_left;

tree_heads[i].req_from_left = 0;

for(i=1; i<num_procs-1; i++)
{
   given_from_left = ((i > 1) ? tree_heads[i-1].give : 0);
   tree_heads[i].given_from_left = given_from_left;
   tree_heads[i].requests = ((nodes_per_proc + tree_heads[i].req_from_left) -
                             (tree_heads[i].nodes + given_from_left) );
   if( (tree_heads[i].requests) < 0 )
    {
     tree_heads[i].give = (-(tree_heads[i].requests));
     tree_heads[i].requests = 0;
    }
   else
     tree_heads[i].give = 0;
```

```
   tree_heads[i+1].req_from_left = tree_heads[i].requests;
}


given_from_left = tree_heads[num_procs-2].give;
tree_heads[num_procs-1].given_from_left = given_from_left;
tree_heads[num_procs-1].requests =
  ( (nodes_per_proc + tree_heads[num_procs-1].req_from_left) -
    (tree_heads[num_procs-1].nodes + given_from_left) );
  if( (tree_heads[num_procs-1].requests) < 0 )
   {
    tree_heads[num_procs-1].give = (-(tree_heads[num_procs-1].requests));
    tree_heads[num_procs-1].requests = 0;
   }
  else
    tree_heads[num_procs-1].give = 0;
}


void print_stats( tree_head,id )
struct t_head *tree_head;
int id;
{
 char fname[30];

 sprintf(fname,"stats.%d",mynode());

 if( tree_head )
 {
  qout = fopen(fname,"a");
  fprintf(qout,"\n***********************************************\n");
  fprintf(qout,"              Qserv Update %d \n",id);
  fprintf(qout,"***********************************************\n");
  fprintf(qout,"   Maximum Key ====> %ld\n",tree_head->max_key);
  fprintf(qout,"tree_head->nodes ==========> %d\n",tree_head->nodes);
  fprintf(qout,"tree_head->extra ==========> %d\n",tree_head->extra);
  fprintf(qout,"tree_head->requests =======> %d\n",tree_head->requests);
  fprintf(qout,"tree_head->req_from_left => %d\n",tree_head->req_from_left);
  fprintf(qout,"tree_head->given_left ====> %d\n",tree_head->given_from_left);
  fprintf(qout,"tree_head->give ===========> %d\n",tree_head->give);
  fprintf(qout,"trans_info.from =====> %d\n",tree_head->trans_info.from);
  fprintf(qout,"trans_info.to =======> %d\n",tree_head->trans_info.to);
  fprintf(qout,"trans_info.op =======> %d\n",tree_head->trans_info.op);
  fprintf(qout,"trans_info.direction > %d\n",tree_head->trans_info.direction);
  fprintf(qout,"trans_info.npp ======> %d\n\n",tree_head->trans_info.nodes_per_proc);
  /* fprintf(qout,"         tree_head->root\n");
  print_node( tree_head->root );
  fprintf(qout,"         tree_head->tail\n");
  print_node( tree_head->tail ); */
  fflush( qout );
  fclose( qout );
 }
}



void get_request( new_request, rec_count, build )
struct req *new_request;
long int *rec_count;
int *build;
{
static int count = 0;
static double range = 1000000.0;
double key_val;
int limit;

if( (*build) )
 {

 if( (*rec_count) == 0 )
   start_time = tot_start_time = mclock();

  if( (*rec_count) < BASE )
  {

   new_request->key = rndm() * range;
   new_request->op = INSERT;

   ins_count++;
   (*rec_count)++;
  }
  else
```

```
  {
   rebal( new_request,&count );
   *rec_count = 0;
   *build = FALSE;
  }
 }
else
 {
 if( ((*rec_count) < UDSIZE) && (count <= UDNUM)  )
 {
  if( (*rec_count) == 0 )
   start_time = tot_start_time = mclock();

  key_val = rndm();
   if( (key_val < 0.5) )
   {
    new_request->op = ACCESS;
    new_request->key = rndm() * range;
    ins_count++;
   }
   else
   {
    new_request->op = ((key_val > 0.75) ? DELETE : INSERT);
    if( (new_request->op == DELETE) )
     {
      new_request->key = rndm() * range;
      del_count++;
     }
    else
     {
      new_request->key = rndm() * range;
      ins_count++;
     }
   }
 (*rec_count)++;
 }
 else
 {
 rebal( new_request,&count );
 *rec_count = 0;
 }
}

}


void rebal( new_request,count )
struct req *new_request;
int *count;
{

if( (*count) <= UDNUM )
{
 new_request->key = REBAL;
 new_request->op = SPECIAL;
 (*count)++;
}
else
{
 new_request->key = EXIT;
 new_request->op = SPECIAL;
}

}


double rndm()
{
 double a,m,q,r,lo,hi,test;

 a = 16807.0;
 m = 2147483647.0;
 q = 127773.0;
 r = 2836.0;

 hi = (int)(seed/q);
 lo = seed - q*hi;
 test = a*lo - r*hi;
   if( test > 0.0 )
    seed = test;
   else
    seed = test + m;
```

```
 return seed/m;

}


void print_times(id, desc)
int id;
char *desc;
{
FILE *tms;
char fname[30];
unsigned long elapsed;

sprintf(fname,"%d-%d.%d",num_procs,BASE,id);

tms = fopen(fname,"w");

elapsed = stop_time - start_time;
/* fprintf(tms,"Start Time =======> %u\n",start_time);
fprintf(tms,"Stop Time  =======> %u\n",stop_time); */
fprintf(tms,"%d     %d      %u\n",num_procs,UDSIZE,elapsed);

fflush( tms );
fclose( tms );
}


void print_tot_times(id, desc)
int id;
char *desc;
{
FILE *tms;
char fname[30];
unsigned long elapsed;

sprintf(fname,"t-%d-%d.%d",num_procs,BASE,id);

tms = fopen(fname,"w");

elapsed = tot_stop_time - tot_start_time;
/* fprintf(tms,"Start Time =======> %u\n",tot_start_time);
fprintf(tms,"Stop Time  =======> %u\n",tot_stop_time); */
fprintf(tms,"%d     %d      %u\n",num_procs,UDSIZE,elapsed);

fflush( tms );
fclose( tms );
}


void mem_error( routine )
char *routine;
{
printf("\n\n ****************** MEMORY ERROR *********************\n");
printf("                      %s\n\n",routine);
exit(0);
}
```

Subroot Processor Program

```
/*****************************************************************************/
/*                                                                           */
/*                    Subtree Scheme Subroot Processor Source Code           */
/*                                  March 20, 1994                           */
/*                                                                           */
/*  This is the code that runs on the subroot processor processors.          */
/*  This program waits for a request to arraive and processes that request in the */
/*  appropriate manner for an M-way tree.  The communication involved is as  */
/*  explained in Chapters 3 & 4.                                             */
/*                                                                           */
/*****************************************************************************/


#include <cube.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#include "locdefs.h"
#include "locstructs.h"

#define DEBUG 0

void rebalance();

void tree_to_vine();
void vine_to_tree();

int  leftmost();
void pack();
void parcel();
void newnode();
void rotate();
void compress();
void movekey();
void move();
void leftshift();
void rightshift();
void tidyup();

struct t_head *create_tree();
int insert();
int ins();
void del();
void del_opt();
void del_in_node();
void del_pred();
struct node *blank_node();
int srch();
void add_new();
void insert_key();
void ins_at_root();
int search_node();
void fillspine();
void redist();
void accept();
void give();
void clean_up();
void delte();

void print_tree();
void prt_tree();
void print_vine();
void inord();
void print_node();
void print_vnode();
void inorder();
void print_stats();
void mem_error();


FILE *dlout, *out, *iocheck, *dbout;

int ins_count, del_count, dup_count, rec_count, acc_count;


void main()
{
int proc_req, proc_rep, repl, signal, status;
```

```
int first = TRUE;
int count = 0;
char filename[20];
long int rec_id, snd_id;
struct req act_req;
struct t_head *tree_head;

proc_req = 100 + mynode();
proc_rep = 100 + mynode();

ins_count = 0;
del_count = 0;
acc_count = 0;
dup_count = 0;
rec_count = 0;

tree_head = create_tree();

csend(READY,&signal,sizeof(signal),ROOT,QSERVER);

for( ; ; )
{

rec_id = irecv(proc_req,&act_req,sizeof(act_req));
if( !first )
  msgdone( snd_id );
msgwait( rec_id );

rec_count++;

 switch( act_req.op ) {
  case INSERT : status = insert( tree_head,act_req );
                         break;
  case DELETE : delte( tree_head,act_req.key );
                         break;
  case ACCESS : status = srch( tree_head->root,act_req.key );
                         break;
  case REBAL  : snd_id = isend(proc_rep,&repl,sizeof(repl),ROOT,QSERVER);
                rebalance( count,tree_head );
                /* printf("Node %d made it back from rebalance\n",mynode()); */
                msgdone(snd_id);
                         count++;
                         break;
  default     : printf("No match for %d\n",act_req.op);
                         dup_count++;
                         break;
        }

repl = status;
snd_id = isend(proc_rep,&repl,sizeof(repl),ROOT,QSERVER);
first = FALSE;

}    /* Close of Endless For Loop */

}


int insert( tree_head,act_req )
struct t_head *tree_head;
struct req act_req;
{
int result, pos;

 result = 0;

   if( !tree_head->root )
     ins_at_root( tree_head,act_req.key );
   else
     result = ins( tree_head->root,tree_head->root,act_req.key,pos );

   if( result == FOUND )
     return( DUPLICATE );
   else
     return( 0 );
}


void ins_at_root( tree_head,key )
struct t_head *tree_head;
long int key;
{
struct node *local;
```

```
local = blank_node();

local->k = 1;
local->c[1].key = key;

tree_head->root = local;

ins_count++;
}


int ins(prev,curr,key,pos)
struct node *prev, *curr;
long int key;
int pos;
{
int search_val;

 if( !curr )
    add_new( prev,curr,key,pos );
 else
   {
    search_val = search_node( curr,key,&pos );
    if( search_val == LEFT )
      ins( curr,curr->c[pos].left,key,pos );
    else if( search_val == RIGHT )
      ins( curr,curr->right,key,pos );
    else if( search_val == HERE )
      insert_key( curr,key,pos );
    else if( search_val == FOUND )
     { dup_count++;
       return( FOUND );
     }
   }
}


void insert_key( curr,key,pos )
struct node *curr;
long int key;
int pos;
{
 curr->k = curr->k + 1;
 curr->c[pos].key = key;
 ins_count++;
}


void add_new( prev,curr,key,pos )
struct node *prev, *curr;
long int key;
int pos;
{
  curr = blank_node();

    if( prev->k >= M-1 && key > prev->c[prev->k].key )
     prev->right = curr;
    else
     prev->c[pos].left = curr;

  curr->c[1].key = key;
  curr->k = 1;
  ins_count++;
}


int srch(curr,key)
struct node *curr;
long int key;
{
int search_val, pos;

 if( !curr )
   { dup_count++;
     return( NOT_FOUND );
   }
 else
   {
    search_val = search_node( curr,key,&pos );
    if( search_val == LEFT )
      srch( curr->c[pos].left,key );
    else if( search_val == RIGHT )
```

```
      srch( curr->right,key );
    else if( search_val == HERE )
      { dup_count++;
        return( NOT_FOUND );
      }
    else if( search_val == FOUND )
      { acc_count++;
        return( FOUND );
      }
   }
}


void delte( tree_head, key )
struct t_head *tree_head;
long int key;
{
int search_val,pos;

 search_val = search_node( tree_head->root,key,&pos );

 if( (search_val==FOUND) && (tree_head->root->k==1) && (!(tree_head->root->c[pos].left)) )
  {
   tree_head->root = NULL;
   del_count++;
  }
 else
   del( NULL,tree_head->root,key );

}


void del( prev, curr, key )
struct node *prev, *curr;
long int key;
{
int search_val, pos;
struct node *del_node;

  search_val = search_node( curr,key,&pos );

    if( (search_val == LEFT) && (curr->c[pos].left) )
      del( curr,curr->c[pos].left,key );
    else if( (search_val == RIGHT) && (curr->right) )
      del( curr,curr->right,key );
    else if( search_val == HERE )
     { /* printf("Key %ld not found in tree (Duplicate)\n",key); */
       dup_count++;
     }
    else if( search_val == FOUND )
      {
        if( curr->c[pos].left != NULL )
           del_opt( curr, pos, key );
        else
          {  del_in_node( curr,key );
             if( curr->k <= 0 )
              {
               search_val = search_node( prev,key,&pos );
               if( search_val == LEFT )
                  prev->c[pos].left = curr->right;
               else if( search_val == RIGHT )
                  prev->right = curr->right;
               free( curr );
              }
          }
      }
   else
     { /* printf("\nKey %ld is a duplicate\n",key); */
       dup_count++;
     }
}

void del_opt( curr,pos,key )
struct node *curr;
int pos;
long int key;
{
int deleted = FALSE;

 if( (curr->k > pos) && (pos < MU) )
  {
    if( !curr->c[pos+1].left )
```

```
      {
        curr->c[pos].key = curr->c[pos+1].key;
        curr->c[pos+1].key = key;
        del_in_node( curr,key );
        deleted = TRUE;
      }
  }
 if( !deleted )
   del_pred( curr,pos,key );

}


void del_in_node( curr,key )
struct node *curr;
long int key;
{
int i;

  for(i=1; i<=curr->k && curr->c[i].key!=key; i++);
  for(i=i; i<curr->k; i++)
    { curr->c[i].key = curr->c[i+1].key;
      curr->c[i].left = curr->c[i+1].left;
    }
  curr->c[curr->k].key = -1;
  curr->c[curr->k].left = NULL;
  curr->k = curr->k - 1;

  del_count++;
}


void del_pred( curr,pos,key )
struct node *curr;
int pos;
long int key;
{
struct node *temp, *prev, *save;
int search_val;

   temp = curr->c[pos].left;

   prev = curr;
     while( temp->right )
        {  prev = temp;
           temp = temp->right;
        }

   curr->c[pos].key = temp->c[temp->k].key;
   temp->c[temp->k].key = key;
   save = temp->c[temp->k].left;
   del_in_node( temp,key );
   if( temp->k > 0 )
     temp->right = save;
   else
    {
     search_val = search_node( prev,curr->c[pos].key,&pos );
     if( search_val == FOUND || search_val == LEFT )
       prev->c[pos].left = save;
     else if( search_val == HERE )
       printf("Oops -- I didn't reset anything\n");
     else if( search_val == RIGHT )
       prev->right = save;
     free( temp );
    }
}


void rebalance(id,tree_head)
int id;
struct t_head *tree_head;
{
struct node *pseudo_root;
int size;
char filename[20], delfname[20];

sprintf(delfname,"delstats.%d",mynode());
dlout = fopen( delfname,"a" );

 if( INORIGINAL )
  {
  sprintf(filename,"orig-%d.%d",id,mynode());
```

```
  /* printf("*******************************************\n");
  printf("The origional tree after update #%d on %d\n",id,mynode());
  printf("*******************************************\n\n"); */
  inorder( tree_head,filename );
  }

  pseudo_root = blank_node();
  pseudo_root->right = tree_head->root;

  tree_to_vine(pseudo_root,id,tree_head);


   redist(id,tree_head);
   pseudo_root->right = tree_head->root;
   clean_up(tree_head);

if( VINES )
 {
  sprintf(filename,"muout.%d",mynode());
  dbout = fopen(filename,"a");
    fprintf(dbout,"\n\nThe Tree #%d as a Vine\n",id);
    print_vine( tree_head->root );
  fflush( dbout );
  fclose( dbout );
 }

if( INVINES )
{
sprintf(filename,"vines-%d.%d",id,mynode());
/* printf("*******************************************\n");
printf("A printout of the vines after redistribution\n");
printf("*******************************************\n\n"); */
inorder( tree_head,filename );
}

if( STATS )
 print_stats(tree_head,id);

   vine_to_tree(pseudo_root,tree_head->nodes,tree_head->extra);
   tree_head->root = pseudo_root->right;

if( INFINAL )
 {
 sprintf(filename,"final-%d.%d",id,mynode());
 fprintf(dlout,"===========================================\n");
 fprintf(dlout,"Total Inserted keys for update #%d ====> %d\n",id,ins_count);
 fprintf(dlout,"Total Deleted keys for update #%d =====> %d\n",id,del_count);
 fprintf(dlout,"Total Accessed keys for update #%d ====> %d\n",id,acc_count);
 fprintf(dlout,"Total Duplicate keys for update #%d ===> %d\n",id,dup_count);
 fprintf(dlout,"Total Received keys for update #%d ====> %d\n",id,rec_count);
 fprintf(dlout,"===========================================\n\n");
 fflush( dlout );
 /* printf("*******************************************\n");
 printf("The Re-balanced tree after update #%d\n",id);
 printf("*******************************************\n\n"); */
 inorder( tree_head,filename );
 }


del_count = 0;
ins_count = 0;
acc_count = 0;
dup_count = 0;
rec_count = 0;

fflush( dlout );
fclose( dlout );
}


void redist(id,tree_head)
int id;
struct t_head *tree_head;
{
int done = FALSE;
struct t_head temp;
char filename[30];

csend(BAL_STATS,tree_head,sizeof(struct t_head),ROOT,QSERVER);

while( !done )
{
```

```
crecv(BAL_STATS,&temp,sizeof(struct t_head));
if( STATS && (temp.trans_info.op != EXIT) )
  print_stats(&temp,id);

switch( temp.trans_info.op ) {
  case ACCEPT : (*tree_head) = temp;
                accept(tree_head);
                if( STATS )
                  print_stats(tree_head,id);
                break;
  case GIVE   : (*tree_head) = temp;
                give(tree_head);
                if( STATS )
                  print_stats(tree_head,id);
                break;
  case EXIT   : done = TRUE;
                break;
  default     : break;
        };
}

}


void accept(tree_head)
struct t_head *tree_head;
{
int rec_buff_size,i;
struct node *rec_buffer, *temp, *hold, *old_root;

if( tree_head->trans_info.direction == RIGHT )
{
 rec_buff_size = ((tree_head->given_from_left)*(sizeof(struct node)));
 rec_buffer = (struct node *)malloc( rec_buff_size );
 if( !rec_buffer )
   mem_error( "rec_buffer" );
 crecv(BAL_SND,rec_buffer,rec_buff_size);

 temp = (struct node *)malloc(sizeof(struct node));
 if( !temp )
   mem_error( "temp in accept - right" );
 hold = temp;
 (*temp) = rec_buffer[0];
 old_root = tree_head->root;
 tree_head->root = temp;
 for(i=1; i<(tree_head->given_from_left)-1; i++)
  {
   temp = (struct node *)malloc(sizeof(struct node));
   if( !temp )
     mem_error( "temp in accept - right" );
   hold->right = temp;
   hold = temp;
   (*temp) = rec_buffer[i];
  }
 if( tree_head->given_from_left > 1 )
 {
  temp = (struct node *)malloc(sizeof(struct node));
  if( !temp )
   mem_error( "temp in accept - right" );
  (*temp) = rec_buffer[(tree_head->given_from_left)-1];
  hold->right = temp;
 }

 if( old_root )
  temp->right = old_root;
 else
  { tree_head->tail = temp;
    tree_head->tail->right = NULL;
  }

 tree_head->max_key = tree_head->tail->c[tree_head->tail->k].key;
 tree_head->nodes = tree_head->nodes + tree_head->given_from_left;
 tree_head->requests = ((tree_head->trans_info.nodes_per_proc +
                         tree_head->req_from_left) - (tree_head->nodes) );
 if( tree_head->requests < 0 )
  { tree_head->give = -(tree_head->requests);
    tree_head->requests = 0;
  }
 else
  tree_head->give = 0;
}
else if( tree_head->trans_info.direction == LEFT )
```

```
{
 rec_buff_size = ((tree_head->requests)*(sizeof(struct node)));
 rec_buffer = (struct node *)malloc( rec_buff_size );
 if( !rec_buffer )
   mem_error( "rec_buffer" );
 crecv(BAL_SND,rec_buffer,rec_buff_size);

 temp = (struct node *)malloc(sizeof(struct node));
 if( !temp )
  mem_error( "temp in accept - left" );
 hold = temp;
 (*temp) = rec_buffer[0];
 old_root = temp;
 for(i=1; i<(tree_head->requests)-1; i++)
  {
   temp = (struct node *)malloc(sizeof(struct node));
   if( !temp )
    mem_error( "temp in accept - left" );
   hold->right = temp;
   hold = temp;
   (*temp) = rec_buffer[i];
  }
 if( tree_head->requests > 1 )
 {
  temp = (struct node *)malloc(sizeof(struct node));
  if( !temp )
   mem_error( "temp in accept - left" );
  (*temp) = rec_buffer[(tree_head->requests)-1];
  hold->right = temp;
 }

 if( tree_head->tail )
  tree_head->tail->right = old_root;
 else
  tree_head->root = old_root;
 tree_head->tail = temp;
 tree_head->tail->right = NULL;
 tree_head->max_key = tree_head->tail->c[tree_head->tail->k].key;
 tree_head->extra = MU - (tree_head->tail->k);
 tree_head->nodes = tree_head->nodes + tree_head->requests;
 tree_head->requests = (tree_head->trans_info.nodes_per_proc +
                        tree_head->req_from_left) - tree_head->nodes;
 if( tree_head->requests < 0 )
  { tree_head->give = -(tree_head->requests);
    tree_head->requests = 0;
  }
 else
  tree_head->give = 0;
}
free( rec_buffer );
csend(BAL_REC,tree_head,sizeof(struct t_head),ROOT,QSERVER);
}


void give(tree_head)
struct t_head *tree_head;
{
struct node *new_tail,*temp, *disp;
int snd_buff_size,i;
struct node *snd_buff;

if( tree_head->trans_info.direction == RIGHT )
{
 snd_buff_size = ((tree_head->give)*(sizeof(struct node)));
 snd_buff = (struct node *)malloc( snd_buff_size );
 if( !snd_buff )
  mem_error( "snd_buff" );
 new_tail = tree_head->root;
 for(i=1; i<(tree_head->nodes)-(tree_head->give); i++)
   new_tail = new_tail->right;
 temp = new_tail->right;
 i = 0;
 while( temp && (i < snd_buff_size) )
  {
   snd_buff[i] = (*temp);
   disp = temp;
   temp = temp->right;
   free( disp );
   i++;
  }
 csend(BAL_SND,snd_buff,snd_buff_size,tree_head->trans_info.to,SUB_PID);
```

```
new_tail->right = NULL;
tree_head->tail = new_tail;
tree_head->extra = MU - (tree_head->tail->k);
tree_head->max_key = tree_head->tail->c[tree_head->tail->k].key;
tree_head->nodes = tree_head->nodes - tree_head->give;
tree_head->give = 0;
}
else if( tree_head->trans_info.direction == LEFT )
{
snd_buff_size = ((tree_head->req_from_left)*(sizeof(struct node)));
snd_buff = (struct node *)malloc( snd_buff_size );
if( !snd_buff )
 mem_error( "snd_buff" );
temp = tree_head->root;
for(i=0; i<(tree_head->req_from_left); i++)
 {
   snd_buff[i] = (*temp);
   disp = temp;
   temp = temp->right;
   free(disp);
 }
tree_head->root = temp;
csend(BAL_SND,snd_buff,snd_buff_size,tree_head->trans_info.to,SUB_PID);

tree_head->max_key = tree_head->tail->c[tree_head->tail->k].key;
tree_head->nodes = tree_head->nodes - tree_head->req_from_left;
tree_head->req_from_left = 0;
tree_head->requests = tree_head->trans_info.nodes_per_proc - tree_head->nodes;
if( tree_head->requests < 0 )
 { tree_head->give = -(tree_head->requests);
   tree_head->requests = 0;
 }
else
 tree_head->give = 0;
}
csend(BAL_SND,tree_head,sizeof(struct t_head),ROOT,QSERVER);
free( snd_buff );
}


void tree_to_vine(x,tree_num,tree)
struct node *x;
int tree_num;
struct t_head *tree;
{
struct node *u, *v;
int n,j,extra;

if( !x->right )
 {
   tree->nodes = 0;
   tree->tail = NULL;
   return;
 }

u = x->right;
n = 0;
j = 1;
  while( u )
    {
      /* Locate leftmost non-nil pointer in U. */
        j = leftmost( u );
        if( j > u->k )
          {
           if( n > 0 )
             pack( x,&u );
           if( u == NULL )
             newnode( &x,&u,0,&n);
           else if( u->k <= MU )
             newnode( &x,&u,1,&n);
           else
             parcel( x,u,&n );
          }
         else
          {
            v = u->c[j].left;
            rotate( u,v,j );
          }
    }
  extra = MU - x->k;

  tree->max_key = x->c[x->k].key;
```

```
  tree->nodes = n;
  tree->extra = extra;
  tree->num_keys = (n*MU) - extra;
  tree->tail = x;
}


void vine_to_tree(x,n,extra)
struct node *x;
int n,extra;
{
struct node *x1;
int lambda, ncomp, ncomp1, excess, i;
int ordinary = FALSE;

lambda = floor( log((float)MU*n) / log((float)MU+1) );

ncomp = (pow( (float)MU+1,(float)lambda ) - 1) / MU;

excess = n - ncomp;
 if( excess > MU*ncomp )
   ordinary = TRUE;
 if( ordinary == TRUE )
   excess = excess - 1;

ncomp1 = excess / MU;

excess = excess % MU;

x1 = x;
for(i=1; i<=ncomp1; i++)
 {
  if( (i < ncomp1) || (excess > 0) )
   compress(&x,(int)MU,(int)MU,(int)0,(int)TRUE);
  else
   compress(&x,MU,MU,extra,ordinary);
 }
if( excess > 0 )
  compress( &x,MU,excess,extra,ordinary );
ncomp = ncomp / (MU+1);
while( ncomp > 0 )
 {
  x = x1;
  for(i=1; i<=ncomp; i++)
   compress( &x,MU,MU,0,TRUE );
  ncomp = ncomp / (MU+1);
 }

if( (extra > 0) && ( n > 1) )
   {
       tidyup( &x );
   }
}


void compress( x,locmu,m1,extra,ordinary)
struct node **x;
int locmu,m1,extra,ordinary;
{
struct node *u, *v;
int i,j,offset;

u = (*x)->right;
(*x)->right = blank_node();
*x = (*x)->right;
(*x)->k = 0;

if( m1 == 1 )
  rightshift( u,*x,extra+(u->k-locmu));
for(i=1; i<=m1; i++)
 {
  v = u->right;
  j = (*x)->k;
  if( (i < m1) || (ordinary == TRUE) )
   {
    leftshift( u,&v,locmu-u->k );
    leftshift( *x,&v,1 );
   }
  else
   {
    offset = extra + (u->k-locmu);
    if( offset > 0 )
```

```
      rightshift( u,*x,offset );
    else
      leftshift( u,&v,-offset );
    leftshift( *x,&v,locmu-((*x)->k) );
   }

  u->right = (*x)->c[j+1].left;
  if( ml == 1 )
    j = 0;
  (*x)->c[j+1].left = u;

  u = v;
 }

 if( u->k == 0 )
   {
    (*x)->right = u->right;
    free( u );
   }
 else
   (*x)->right = u;

}


void leftshift( u,v,count )
struct node *u, **v;
int count;
{
int i;
struct node *temp;

 if( count > (*v)->k )
  {
   for(i=1; i<=(*v)->k; i++)
      u->c[u->k+i] = (*v)->c[i];
   u->k = u->k + (*v)->k;
   count = count - (*v)->k;
   temp = *v;
   (*v) = (*v)->right;
   free( temp );
  }
 for(i=1; i<=count; i++)
   u->c[u->k+i] = (*v)->c[i];
 if( count > 0 )
  {
   u->k = u->k + count;
   (*v)->k = (*v)->k - count;
   for(i=1; i<=(*v)->k; i++)
       (*v)->c[i] = (*v)->c[count+i];
  }
}

void rightshift( u,v,count )
struct node *u, *v;
int count;
{
int i;

  for(i=1; i<=count; i++)
    v->c[v->k+i] = u->c[(u->k)-count+i];
  if( count > 0 )
    {
     u->k = u->k - count;
     v->k = v->k + count;
    }
}



void tidyup( x )
struct node **x;
{
struct node *xprev;

  xprev = NULL;
  while( (*x)->right )
   {
     xprev = *x;
     *x = (*x)->right;
   }
```

```
if( xprev )
  {
   leftshift(xprev,x,MU-(xprev->k));
   if( (*x)->k == 0 )
    {
      free( xprev->right );
      xprev->right = NULL;          /* This line added  November 8, 1993 */
    }
  }
}


int leftmost( u )
struct node *u;
{
int i;

for(i=1; i<=u->k && u->c[i].left==NULL; i++);

return(i);

}

void parcel( x,u,n )
struct node *x, *u;
int *n;
{
int d, i;

printf("I called parcel, this is bad!!!\n");
exit(0);

d = 0;

do {
     x->right = blank_node();
     x = x->right;
     x->k = MU;
     for(i=1; i<=MU; i++)
       x->c[i] = u->c[i+d];
     d = d+MU;
       } while( u->k > d+MU );
x->right = u;

u->k = u->k - d;
for(i=1; i<=u->k; i++)
  u->c[i] = u->c[i+d];

newnode( &x,&u,d/MU+1,n );
}


void newnode( x,u,count,n )
struct node **x, **u;
int count,*n;
{
struct node *temp;

 if( count > 0 )
  {
   temp = *u;
   *u = (*u)->right;
   *x = temp;
   (*n) = (*n) + count;
  }
 else
  {
   *u = (*x)->right;
  }
}

void rotate( u,v,j )
struct node *u, *v;
int j;
{
int i, k, l;
struct child save;

u->c[j].left = v->right;
v->right = u->right;
u->right = v;
```

```
for(k=j,i=1; k<=u->k; i++,k++)
 {
  save = u->c[k];
  u->c[k] = v->c[1];
   for(l=1; l<v->k; l++)
    v->c[l] = v->c[l+1];
   v->c[l] = save;
 }
}


void pack( x,u )
struct node *x, **u;
{
int i, k, ksum;

  if( x->k < MU )
   {
     for(k=x->k+1; k<=MU; k++)
      {
        x->c[k] = (*u)->c[1];
         for(i=1; i<(*u)->k; i++)
           (*u)->c[i] = (*u)->c[i+1];
         (*u)->c[i].key = -1;
         (*u)->c[i].left = NULL;
      }
   }
  ksum = x->k + (*u)->k;
  if( ksum > MU )
   {
    x->k = MU;
    (*u)->k = ksum-MU;
   }
  else
   {
    x->k = ksum;
    x->right = (*u)->right;
    free( *u );
    *u = NULL;
   }
}


struct t_head *create_tree()
{
struct t_head *local;

local = (struct t_head *)malloc( sizeof(struct t_head) );
if( !local )
 mem_error( "create tree" );

local->root = NULL;
local->tail = NULL;
local->proc_num = mynode();
local->extra = 0;
local->num_keys = 0;
local->requests = 0;
local->req_from_left = 0;
local->give = 0;
local->max_key = -1;

local->req_count = 0;
local->run_time = 0;
local->comp_time = 0;

return( local );

}


struct node *blank_node()
{
 struct node *local_node;
 int i;

   local_node = (struct node *)malloc(sizeof(struct node));
    if( !local_node )
       mem_error( "blank_node" );
   local_node->k = 0;
   local_node->right = NULL;
    for(i=1; i<=MU; i++)
      {
```

```
        local_node->c[i].key = -1;
        local_node->c[i].left = NULL;
       }
    return( local_node );
}


int search_node( curr,key,pos )
struct node *curr;
long int key;
int *pos;
{

 for(*pos=1; *pos<=curr->k; (*pos)++)
  {
   if( key < curr->c[*pos].key )
     return( LEFT );
   if( key == curr->c[*pos].key )
     return( FOUND );
  }
 if( (curr->k < MU) && (!curr->right) )        /* line changed 11-11-93 */
   return( HERE );
 else
   return( RIGHT );
}


void clean_up(tree_head)
struct t_head *tree_head;
{
struct node *curr, *next, *rmvd;
int i,count;

curr = tree_head->root;
next = curr->right;

  while( next )
   {
    if( curr->k < MU )
     {
        count = ((next->k >= MU-curr->k) ? MU-curr->k : next->k);
        for(i=1; i<=count; i++)
          curr->c[curr->k+i] = next->c[i];
        curr->k = curr->k + count;
        next->k = next->k - count;
        if(next->k > 0)
         {
          for(i=1; i<=next->k; i++)
            next->c[i] = next->c[i+count];
         }
        else
         {
          curr->right = next->right;
          rmvd = next;
          free( next );
          next = curr->right;
          tree_head->nodes = tree_head->nodes - 1;
          if( rmvd == tree_head->tail )
              {
               tree_head->tail = curr;
              }
         }
       curr = ((curr->k == MU) ? next : curr);
       next = ((curr) ? curr->right : NULL);
     }
    else
     {
       curr = next;
       next = curr->right;
     }
   }
 tree_head->extra = MU - (tree_head->tail->k);
}


void inorder( tree_head,id )
struct t_head *tree_head;
char *id;
{
  iocheck = fopen( id,"w" );

  inord(tree_head->root);
```

```
   fflush( iocheck );
   fclose( iocheck );
}

void inord( curr )
struct node *curr;
{
int i;

     if(curr == NULL)
          return;
     else
      {
        for(i=1;i<=curr->k;i++) {
          inord(curr->c[i].left);
          fprintf(iocheck,"%ld\n",curr->c[i].key);
          }
        inord(curr->right);
      }
}


void print_tree( tree_head )
struct t_head *tree_head;
{

 prt_tree(tree_head->root);

}

void prt_tree( curr )
struct node *curr;
{
int i;

     if(curr == NULL)
          return;
     else
     {
      print_node( curr );
      for(i=1;i<=curr->k;i++)
        prt_tree(curr->c[i].left);
      prt_tree(curr->right);
     }
}

void print_vine( curr )
struct node *curr;
{
int i;

     if(curr == NULL)
          return;
     else
     {
      print_vnode( curr );
      for(i=1;i<=curr->k;i++)
        print_vine(curr->c[i].left);
      print_vine(curr->right);
     }
}

void print_node( curr )
struct node *curr;
{
int i;
  if( curr )
    {
      fprintf(out,"Node Address ==========> %d\n",curr);
      fprintf(out,"================================\n");
      fprintf(out,"Node Count   ==========> %d\n",curr->k);
      for(i=1; i<=curr->k; i++) {
      fprintf(out,"c[%d].key    ==========> %ld\n",i,curr->c[i].key);
      fprintf(out,"c[%d].left   ==========> %d\n",i,curr->c[i].left);
                               }
      fprintf(out,"Node right   ==========> %d\n",curr->right);
      fprintf(out,"\n\n");
    }
}

void print_vnode( curr )
struct node *curr;
```

```
{
int i;
  if( curr )
   {
     fprintf(dbout,"Node Address ==========> %d\n",curr);
     fprintf(dbout,"==================================\n");
     fprintf(dbout,"Node Count   ==========> %d\n",curr->k);
     for(i=1; i<=curr->k; i++) {
     fprintf(dbout,"c[%d].key    ==========> %ld\n",i,curr->c[i].key);
     fprintf(dbout,"c[%d].left   ==========> %d\n",i,curr->c[i].left);
                              }
     fprintf(dbout,"Node right   ==========> %d\n",curr->right);
     fprintf(dbout,"\n\n");
   }
}


void print_stats( tree_head,id )
struct t_head *tree_head;
int id;
{
 char fname[30];

 sprintf(fname,"stats.%d",mynode());

 if( tree_head )
 {
  out = fopen(fname,"a");
  fprintf(out,"\n************************************************\n");
  fprintf(out,"                    Update %d \n",id);
  fprintf(out,"************************************************\n");
  fprintf(out,"   Maximum Key ====> %ld\n",tree_head->max_key);
  fprintf(out,"tree_head->nodes ==========> %d\n",tree_head->nodes);
  fprintf(out,"tree_head->extra ==========> %d\n",tree_head->extra);
  fprintf(out,"tree_head->requests =======> %d\n",tree_head->requests);
  fprintf(out,"tree_head->req_from_left => %d\n",tree_head->req_from_left);
  fprintf(out,"tree_head->given_left ====> %d\n",tree_head->given_from_left);
  fprintf(out,"tree_head->give ===========> %d\n",tree_head->give);
  fprintf(out,"trans_info.from ======> %d\n",tree_head->trans_info.from);
  fprintf(out,"trans_info.to ========> %d\n",tree_head->trans_info.to);
  fprintf(out,"trans_info.op ========> %d\n",tree_head->trans_info.op);
  fprintf(out,"trans_info.direction > %d\n",tree_head->trans_info.direction);
  fprintf(out,"trans_info.npp =======> %d\n\n",tree_head->trans_info.nodes_per_proc);
  fprintf(out,"         tree_head->root\n");
  print_node( tree_head->root );
  fprintf(out,"         tree_head->tail\n");
  print_node( tree_head->tail );
  fflush( out );
  fclose( out );
 }
}


void mem_error( routine )
char *routine;
{
printf("\n\n ****************** MEMORY ERROR *********************\n");
printf("                     %s\n\n",routine);
printf(" ******************************************************\n");
fflush( stdout );
exit(0);
}
```

APPENDIX C

Pipeline Scheme Source Code

Host Program

```
/*****************************************************************************/
/*                                                                           */
/*                          Pipeline Scheme Host Source Code                 */
/*                                  March 22, 1994                           */
/*                                                                           */
/*  This is the code that runs on the host computer to drive the pipeline scheme.  */
/*  It allocates the cube, loads the nodes with the proper code, and deallocates   */
/*  the cube.                                                                */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <cube.h>
#include "simdefs.h"
#include "simstructs.h"

#define UDNUM 2

main()
{
int i,j,k,signal;
int done = FALSE;
int num_procs,prob_size;
struct cnfg config;
char response[30];
char cube_size[30];

for(k=4; k<=32; k=k*2)
 {
 for(j=1000; j<=100000; j=j+10000)
  {
    if( j == 11000 )
        j = 10000;
    num_procs = k;
    prob_size = j;
    sprintf(cube_size,"%d",(num_procs));
    strcat(cube_size,"sx");
    printf("\n\nWaiting for cube ==> %s  Prob Size ==> %d\n",cube_size,prob_size);
    done = FALSE;

    while(_getcube("Barney",cube_size,NULL,0,0));

    printf("Cube Allocated ====> %s\n\n\n",cube_size);

    setpid(HOST_PID);
    load("simroot",gray(ROOT),0);
    for(i=1; i<num_procs-2; i++)
     load("simnode",gray(i),0);
    load("simlgn",gray(num_procs-2),0);
    load("simlast",gray(num_procs-1),0);

   printf("*****************************************************************\n");
   printf("          All nodes Loaded  ====  Execution Begins\n");
   printf("*****************************************************************\n\n");

 crecv(READY,&signal,sizeof(signal));
 config.procs = num_procs;
 config.base = prob_size;
 config.udsize = prob_size;
 config.udnum = UDNUM;
 csend(CONFIG,&config,sizeof(config),ROOT,0);

 while( !done )
  {
   crecv(SIGNAL,&signal,sizeof(int));
   if( signal == EXIT )
    { done = TRUE;
      printf("\nRoot Received EXIT Message from Nodes...Terminating Cube\n\n");
    }
   else if( signal == UDATE )
    printf("\nUpdate Completed ==> No Errors\n\n");
  }

  killcube(ALL_NODES,ALL_PIDS);
 relcube("Barney");

 }
}
}
```

Root Processor (P_0- First Processor in the Array) Program

```
/*****************************************************************************/
/*                                                                           */
/*                    Pipeline Scheme Root Processor Source Code             */
/*                                  March 22, 1994                           */
/*                                                                           */
/*  This is the code that runs on the root processor node.                   */
/*  This program accepts requests and passes them on in the manner described in */
/*  Chapters 3 & 4 to the processor directly below it in the array.          */
/*  It also accepts replies from the last processor in the array.            */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <cube.h>

#include "simdefs.h"
#include "simstructs.h"


void mem_error();
void ins();
void del();
void srch();
struct node *search_node();
void print_stats();
void set_gray();
void print_times();
double rndm();

double seed = 1.0;

int NEXT, PREV, UDNUM, UDSIZE, BASE, NUM_PROCS;

long int ins_count, del_count, acc_count;

unsigned long start_time, stop_time;

main()
{
int i,j,status;
long int *del_array, *srch_array;
int count = 0;
double range = 1000000.0;
double key_val;
struct req_type new_request;
struct cnfg config;
int limit;

ins_count = del_count = acc_count = 0;

start_time = stop_time = 0;

csend(READY,&status,sizeof(status),myhost(),HOST_PID);

crecv(CONFIG,&config,sizeof(config));
  NUM_PROCS = config.procs;
  BASE = config.base;
  UDNUM = config.udnum;
  UDSIZE = config.udsize;

set_gray( (int) ROOT );

printf("Executing cube ==> %dsx  Problem Size ==> %d at %d, %d\n",NUM_PROCS,
       UDNUM,BASE,UDSIZE);

del_array = (long int *)calloc( (int)(BASE*0.25)+1,sizeof(long int) );
if( !del_array )
  mem_error( "del_array" );

srch_array = (long int *)calloc( (int)(BASE*0.5)+1,sizeof(long int) );
if( !srch_array )
  mem_error( "srch_array" );


for(i=1; i<=NUM_PROCS-1; i++)
  crecv(READY,&status,sizeof(status));
for(i=1; i<=NUM_PROCS-1; i++)
  csend(READY,&NUM_PROCS,sizeof(NUM_PROCS),i,0);
for(i=1; i<=NUM_PROCS-1; i++)
```

```
  crecv(READY,&status,sizeof(status));

start_time = mclock();

 for(i=1; i<=BASE; i++)
  {
   new_request.key =  rndm() * range;
   new_request.op = INSERT;

   switch( new_request.op ) {
     case INSERT : ins( new_request );
                   break;
     case DELETE : del( new_request );
                   break;
     case ACCESS : srch( new_request );
                   break;
     default     : printf("Unknown op on %d\n",mynode());
                   break;
           };

   if( count > NUM_PROCS )
     crecv(ACK,&status,sizeof(status));
   else
     count++;
  }
for(i=1; i<=count; i++)
  crecv(ACK,&status,sizeof(status));

stop_time = mclock();

print_times(0);

printf("Made it through the BASE loop\n");

for(j=1; j<=UDNUM; j++)
{
count = 0;

start_time = mclock();
 for(i=1; i<=UDSIZE; i++)
  {
   key_val = rndm();
   if( (key_val < 0.5) )
   {
    new_request.op = ACCESS;
    new_request.key = rndm() * range;
   }
   else
   {
    new_request.op = ((key_val > 0.75) ? DELETE : INSERT);
    if( (new_request.op == DELETE) )
     new_request.key = rndm() * range;
    else
      new_request.key = rndm() * range;
   }

   switch( new_request.op ) {
     case INSERT : ins( new_request );
                   break;
     case DELETE : del( new_request );
                   break;
     case ACCESS : srch( new_request );
                   break;
     default     : printf("Unknown op on %d\n",mynode());
                   break;
           };

   if( count > NUM_PROCS )
     crecv(ACK,&status,sizeof(status));
   else
     count++;
   }
 for(i=1; i<=count; i++)
   crecv(ACK,&status,sizeof(status));

stop_time = mclock();
print_times( j );

printf("Made it through an UPDATE loop\n");
 }
new_request.op = EXIT;
```

```
csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);
crecv(TRANS,&new_request,sizeof(new_request));
status = EXIT;
if( STATS)
  print_stats();
csend(SIGNAL,&status,sizeof(status),myhost(),HOST_PID);
}


void ins( new_request )
struct req_type new_request;
{
struct ins_trans_repl_type it_repl;
struct ins_type insert;
struct node curr;

ins_count++;

new_request.p_prime = search_node(curr,new_request.key);
csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);

crecv(IT_REPLY,&it_repl,sizeof(it_repl));
csend(INS,&insert,sizeof(insert),NEXT,PROC_PID);

}


void del( new_request )
struct req_type new_request;
{
struct del_trans_repl_type dt_repl;
struct del_type delte;
struct node curr;

del_count++;

new_request.p_prime = search_node(curr,new_request.key);
csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);

crecv(DT_REPLY,&dt_repl,sizeof(dt_repl));
csend(DEL,&delte,sizeof(delte),NEXT,PROC_PID);

}


void srch( new_request )
struct req_type new_request;
{
struct node curr;

new_request.p_prime = search_node(curr,new_request.key);

csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);

acc_count++;
}


double rndm()
{
 double a,m,q,r,lo,hi,test;

 a = 16807.0;
 m = 2147483647.0;
 q = 127773.0;
 r = 2836.0;

 hi = (int)(seed/q);
 lo = seed - q*hi;
 test = a*lo - r*hi;

   if( test > 0.0 )
    seed = test;
   else
    seed = test + m;
 return seed/m;

}
```

```
void set_gray(node_num)
int node_num;
{
int *gray_code,i;

gray_code = (int *)calloc(NUM_PROCS,sizeof(int));

for(i=0; i<NUM_PROCS; i++)
   gray_code[i] = gray(i);

for(i=0; i<NUM_PROCS; i++)
 {
  if( node_num == gray_code[i] )
    {
      NEXT = (i == (NUM_PROCS-1)) ? 0 : gray_code[i+1];
      PREV = (i == 0) ? 0 : gray_code[i-1];
    }
 }
free( gray_code );
}

void print_stats()
{
FILE *out;
char fname[30];

sprintf(fname,"stats.%d",mynode());

out = fopen(fname,"w");

fprintf(out,"Insert Count ====> %d\n",ins_count);
fprintf(out,"Delete Count ====> %d\n",del_count);
fprintf(out,"Access Count ====> %d\n",acc_count);

fflush( out );
fclose( out );
}


struct node *search_node( curr,key )
struct node curr;
long int key;
{
int i;

for(i=0; i<3 && (key != curr.n[i]); i++);

return( curr.child[i] );
}


void print_times(id)
int id;
{
FILE *tms;
char fname[30];
unsigned long elapsed;

sprintf(fname,"%d-%d.%d",NUM_PROCS,BASE,id);

tms = fopen(fname,"w");

elapsed = stop_time - start_time;
/* fprintf(tms,"Start Time =======> %u\n",start_time);
fprintf(tms,"Stop Time  =======> %u\n",stop_time); */
fprintf(tms,"%d     %d     %u\n",NUM_PROCS,UDSIZE,elapsed);

fflush( tms );
fclose( tms );
}


void mem_error( routine )
char *routine;
{
printf("\n\n ****************** MEMORY ERROR *********************\n");
printf("                        %s\n\n",routine);
exit(0);
}
```

Index Set Processor (P_1 through P_{N-1}) Program

```
/*****************************************************************************/
/*                                                                           */
/*                    Pipeline Scheme Middle Processor Source Code           */
/*                                  March 22, 1994                           */
/*                                                                           */
/*  This is the code that runs on the Middle processors in the processor array.  */
/*  This program exchanges messages with the processor directly above and below  */
/*  it in the processor array as describes in Chapters 2, 3 & 4.             */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <cube.h>

#include "simdefs.h"
#include "simstructs.h"

void ins();
void del();
void srch();
void ext();
struct node *search_node();
void set_gray();
void mem_error();


long int ins_count, del_count, acc_count;
int node_num;

int NEXT, PREV, NUM_PROCS;

main()
{
struct req_type new_request;
int done = FALSE;

ins_count = del_count = acc_count = 0;
node_num = mynode();

csend(READY,&done,sizeof(done),ROOT,PROC_PID);
crecv(READY,&NUM_PROCS,sizeof(NUM_PROCS));

set_gray( node_num );

csend(READY,&done,sizeof(done),ROOT,PROC_PID);

while( !done )
{
crecv(TRANS,&new_request,sizeof(new_request));

switch( new_request.op ) {
  case INSERT : ins( new_request );
                break;
  case DELETE : del( new_request );
                break;
  case ACCESS : srch( new_request );
                break;
  case EXIT   : ext( new_request );
                done = TRUE;
                break;
  default     : printf("Unknown op on %d\n",mynode());
                break;

       };
}

}


void ins( new_request )
struct req_type new_request;
{
struct ins_trans_repl_type it_repl;
struct ins_type insert;
struct node curr;

ins_count++;

csend(IT_REPLY,&it_repl,sizeof(it_repl),PREV,PROC_PID);
crecv(INS,&insert,sizeof(insert));
```

```
new_request.p_prime = search_node(curr,new_request.key);
csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);

crecv(IT_REPLY,&it_repl,sizeof(it_repl));

new_request.p_prime = search_node(curr,new_request.key);
csend(INS,&insert,sizeof(insert),NEXT,PROC_PID);

}


void del( new_request )
struct req_type new_request;
{
struct del_trans_repl_type dt_repl;
struct del_type delte;
struct node curr;

del_count++;

csend(DT_REPLY,&dt_repl,sizeof(dt_repl),PREV,PROC_PID);
crecv(DEL,&delte,sizeof(delte));

new_request.p_prime = search_node(curr,new_request.key);
csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);

crecv(DT_REPLY,&dt_repl,sizeof(dt_repl));

new_request.p_prime = search_node(curr,new_request.key);
csend(DEL,&delte,sizeof(delte),NEXT,PROC_PID);

}


void srch( new_request )
struct req_type new_request;
{
struct node curr;

acc_count++;

new_request.p_prime = search_node(curr,new_request.key);
csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);

}

void ext( new_request )
struct req_type new_request;
{
FILE *out;
char fname[30];

if( STATS )
{
sprintf(fname,"stats.%d",mynode());

out = fopen(fname,"w");

fprintf(out,"Insert Count ====> %d\n",ins_count);
fprintf(out,"Delete Count ====> %d\n",del_count);
fprintf(out,"Access Count ====> %d\n",acc_count);

fflush( out );
fclose( out );
}

csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);

}


void set_gray(node_num)
int node_num;
{
int *gray_code,i;

gray_code = (int *)calloc(NUM_PROCS,sizeof(int));

for(i=0; i<NUM_PROCS; i++)
   gray_code[i] = gray(i);

for(i=0; i<NUM_PROCS; i++)
```

```
 {
  if( node_num == gray_code[i] )
     {
       NEXT = (i == (NUM_PROCS-1)) ? 0 : gray_code[i+1];
       PREV = (i == 0) ? 0 : gray_code[i-1];
     }
 }
free( gray_code );
}


struct node *search_node( curr,key )
struct node curr;
long int key;
{
int i;

for(i=0; i<3 && (key != curr.n[i]); i++);

return( curr.child[i] );
}



void mem_error( routine )
char *routine;
{
printf("\n\n ****************** MEMORY ERROR ********************\n");
printf("                       %s\n\n",routine);
exit(0);
}
```

Last Index Set Processor (P_{LN}) Program

```
/*****************************************************************************/
/*                                                                           */
/*                    Pipeline Scheme Last Middle Processor Source Code      */
/*                                   March 22, 1994                          */
/*                                                                           */
/*  This is the code that runs on the last middle processor in the array.    */
/*  This program exchanges messages with the processor directly above and below */
/*  it in the processor array as described in chapters 2, 3 & 4.             */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <cube.h>

#include "simdefs.h"
#include "simstructs.h"

void ins();
void del();
void srch();
void ext();
struct node *search_node();
void set_gray();
void mem_error();


long int ins_count, del_count, acc_count;
int node_num;

int NEXT, PREV, NUM_PROCS;

main()
{
struct req_type new_request;
int done = FALSE;

ins_count = del_count = acc_count = 0;
node_num = mynode();

csend(READY,&done,sizeof(done),ROOT,PROC_PID);
crecv(READY,&NUM_PROCS,sizeof(NUM_PROCS));

set_gray( node_num );

csend(READY,&done,sizeof(done),ROOT,PROC_PID);

while( !done )
{
crecv(TRANS,&new_request,sizeof(new_request));

switch( new_request.op ) {
  case INSERT : ins( new_request );
                break;
  case DELETE : del( new_request );
                break;
  case ACCESS : srch( new_request );
                break;
  case EXIT   : ext( new_request );
                done = TRUE;
                break;
  default     : printf("Unknown op on %d\n",mynode());
                break;

       };
}

}


void ins( new_request )
struct req_type new_request;
{
struct ins_trans_repl_type it_repl;
struct ins_repl_type ins_repl;
struct ins_type insert;
struct node curr;

ins_count++;

csend(IT_REPLY,&it_repl,sizeof(it_repl),PREV,PROC_PID);
```

```
crecv(INS,&insert,sizeof(insert));

new_request.p_prime = search_node(curr,new_request.key);
csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);
crecv(INS_REPLY,&ins_repl,sizeof(ins_repl));
}


void del( new_request )
struct req_type new_request;
{
struct del_trans_repl_type dt_repl;
struct del_repl_type del_repl;
struct del_type delte;
struct node curr;

del_count++;

csend(DT_REPLY,&dt_repl,sizeof(dt_repl),PREV,PROC_PID);
crecv(DEL,&delte,sizeof(delte));

new_request.p_prime = search_node(curr,new_request.key);
csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);
crecv(DEL_REPLY,&del_repl,sizeof(del_repl));

}


void srch( new_request )
struct req_type new_request;
{
struct node curr;

acc_count++;

new_request.p_prime = search_node(curr,new_request.key);
csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);

}

void ext( new_request )
struct req_type new_request;
{
FILE *out;
char fname[30];

if( STATS )
{
sprintf(fname,"stats.%d",mynode());

out = fopen(fname,"w");

fprintf(out,"Insert Count ====> %d\n",ins_count);
fprintf(out,"Delete Count ====> %d\n",del_count);
fprintf(out,"Access Count ====> %d\n",acc_count);

fflush( out );
fclose( out );
}

csend(TRANS,&new_request,sizeof(new_request),NEXT,PROC_PID);

}


void set_gray(node_num)
int node_num;
{
int *gray_code,i;

gray_code = (int *)calloc(NUM_PROCS,sizeof(int));

for(i=0; i<NUM_PROCS; i++)
   gray_code[i] = gray(i);

for(i=0; i<NUM_PROCS; i++)
 {
  if( node_num == gray_code[i] )
    {
      NEXT = (i == (NUM_PROCS-1)) ? 0 : gray_code[i+1];
      PREV = (i == 0) ? 0 : gray_code[i-1];
    }
```

```
 }
free( gray_code );
}


struct node *search_node( curr,key )
struct node curr;
long int key;
{
int i;

for(i=0; i<3 && (key != curr.n[i]); i++);

return( curr.child[i] );
}


void mem_error( routine )
char *routine;
{
printf("\n\n ****************** MEMORY ERROR ********************\n");
printf("                       %s\n\n",routine);
exit(0);
}
```

Key Set Processor (P_{lgN+1}- Last Processor in the Array) Program

```
/*****************************************************************************/
/*                                                                           */
/*                      Pipeline Scheme Last Processor Source Code           */
/*                                   March 22, 1994                          */
/*                                                                           */
/*  This is the code that runs on the last processor in the array.           */
/*  This program accepts requests and then sends an acknowledgement signal to the */
/*  root processor (first processor in the array).  Again the messages are as */
/*  described in Chapters 2, 3, & 4.                                         */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <cube.h>

#include "simdefs.h"
#include "simstructs.h"

void ins();
void del();
void srch();
void ext();
struct node *search_node();
void set_gray();
void mem_error();


long int ins_count, del_count, acc_count;
int node_num;

int NEXT, PREV, NUM_PROCS;

main()
{
struct req_type new_request;
int done = FALSE;

ins_count = del_count = acc_count = 0;
node_num = mynode();

csend(READY,&done,sizeof(done),ROOT,PROC_PID);
crecv(READY,&NUM_PROCS,sizeof(NUM_PROCS));

set_gray( node_num );

csend(READY,&done,sizeof(done),ROOT,PROC_PID);

while( !done )
{
crecv(TRANS,&new_request,sizeof(new_request));

switch( new_request.op ) {
  case INSERT : ins( new_request );
                break;
  case DELETE : del( new_request );
                break;
  case ACCESS : srch( new_request );
                break;
  case EXIT   : ext( new_request );
                done = TRUE;
                break;
  default     : printf("Unknown op on %d\n",mynode());
                break;

      };
}

}


void ins( new_request )
struct req_type new_request;
{
struct ins_repl_type ins_repl;
struct ack_type ack;
struct node curr;

ins_count++;

csend(INS_REPLY,&ins_repl,sizeof(ins_repl),PREV,PROC_PID);
```

```
new_request.p_prime = search_node(curr,new_request.key);
csend(ACK,&ack,sizeof(ack),ROOT,PROC_PID);

}


void del( new_request )
struct req_type new_request;
{
struct del_repl_type del_repl;
struct ack_type ack;
struct node curr;

del_count++;

csend(DEL_REPLY,&del_repl,sizeof(del_repl),PREV,PROC_PID);
new_request.p_prime = search_node(curr,new_request.key);
csend(ACK,&ack,sizeof(ack),ROOT,PROC_PID);

}


void srch( new_request )
struct req_type new_request;
{
struct ack_type ack;
struct node curr;

acc_count++;

new_request.p_prime = search_node(curr,new_request.key);
csend(ACK,&ack,sizeof(ack),ROOT,PROC_PID);

}

void ext( new_request )
struct req_type new_request;
{
FILE *out;
char fname[30];

if( STATS )
{
sprintf(fname,"stats.%d",mynode());

out = fopen(fname,"w");

fprintf(out,"Insert Count ====> %d\n",ins_count);
fprintf(out,"Delete Count ====> %d\n",del_count);
fprintf(out,"Access Count ====> %d\n",acc_count);

fflush( out );
fclose( out );
}

csend(TRANS,&new_request,sizeof(new_request),ROOT,PROC_PID);

}


void set_gray(node_num)
int node_num;
{
int *gray_code,i;

gray_code = (int *)calloc(NUM_PROCS,sizeof(int));

for(i=0; i<NUM_PROCS; i++)
   gray_code[i] = gray(i);

for(i=0; i<NUM_PROCS; i++)
 {
  if( node_num == gray_code[i] )
    {
      NEXT = (i == (NUM_PROCS-1)) ? 0 : gray_code[i+1];
      PREV = (i == 0) ? 0 : gray_code[i-1];
    }
 }
free( gray_code );
}
```

```
struct node *search_node( curr,key )
struct node curr;
long int key;
{
int i;

for(i=0; i<3 && (key != curr.n[i]); i++);

return( curr.child[i] );
}


void mem_error( routine )
char *routine;
{
printf("\n\n ****************** MEMORY ERROR ********************\n");
printf("                        %s\n\n",routine);
exit(0);
}
```

APPENDIX D

Gray Code Calculation

Gray Code Calculation

There is a simple trick to generate a gray code that can be used to give a chain addressing to nodes in a hypercube parallel computer.

For a cube of **2** nodes, the labels are given as:

0
1

Then, as the number of nodes is increased (by a power of 2, as is required by the hypercube architecture), simply copy the code from the previous size, invert it and append the inversion to the end, and finally add a most significant bit each line such that the first half of the lines get a 0 and the second half get a 1. So for a cube of **4** nodes the code is:

Binary gray codes	Decimal Equivalent
00	0
01	1
1*1*	3
1*0*	2

Where **0,1** are the added most significant bit, 0,1 are the gray code from the previous power of two, and *0,1* are the inverted code from the previous power of two.

Thus the "*chain*" addresses of the nodes are 0,1,3,2 decimal. This can be repeated for the next power of 2, (**8**), and so on for as many nodes as are in the cube.

Binary gray codes	Decimal Equivalent
000	0
001	1
011	3
010	2
1*10*	6
1*11*	7
1*01*	5
1*00*	4

So the gray code labeling of the nodes would be (0,1,3,2,6,7,5,4).

VITA 2

Troy H. Laramy

Candidate for the Degree of

Master of Science

Thesis: AN ALTERNATIVE METHOD FOR PARALLEL M-WAY TREE SEARCH ON DISTRIBUTED MEMORY ARCHITECTURES

Major Field: Computer Science

Biographical:

Personal Data: Born in Topeka, Kansas, On November 3, 1968, the son of Richard and Marilyn Laramy.

Education: Graduated from Ponca City High School, Ponca City, Oklahoma in May 1987; received Bachelor of Science degree in Computer Science from Oklahoma State University, Stillwater, Oklahoma in December 1991. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 1994.

Professional Experience: Teaching Assistant, Oklahoma State University, Department of Computer Science, August 1993 to May 1994.