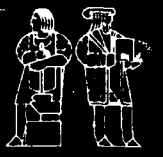
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-497

AN EVALUATION OF CONCURRENT PRIORITY QUEUE ALGORITHMS

Qin Huang

February 1991

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S): MIT/LCS/TR-49	5 MONITORING ORGANIZATION REPORT NUMBER(S): N00014-89-J-1988
6a NAME OF PERFORMING ORGANIZATION: MIT Lab for Computer Science	6b OFFICE SYMBOL (If applicable) / 7a NAME OF MONITORING ORGANIZATION: Office of Naval Research, Dept. of Navy
6c ADDRESS (City, State, and ZIP Code): 545 Technology Square, Cambridge, MA 02139	7b ADDRESS (City, State, and ZIP Code): Information Systems Program, Arlington, VA 22217
8a NAME OF FUNDING/SPONSORING ORGANIZATION: DARPA	8b OFFICE SYMBOL (If applicable) / 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code): 1400 Wilson Blvd, Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO
11 TITLE (Include Security Classification): An Implementation of Concurrent Priority Queue Algorithms	
12 PERSONAL AUTHOR(S): Nir Shavit	
13a TYPE OF REPORT: Technical / 13b TIME COVERED: FROM ____ TO ____	14 DATE OF REPORT (Year, Month, Day): February 1991 / 15 PAGE COUNT: 84
16 SUPPLEMENTARY NOTATION	
17 COSATI CODES: FIELD / GROUP / SUB-GROUP	18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The priority queue is a fundamental data structure that is used in a large variety of parallel algorithms, such as multiprocessor scheduling and parallel best-first search of state-space graphs. In these algorithms, each process performs an access-think cycle, in which the access is one of the insert, extract, decrease key, and delete operations on the priority queues. This thesis addresses the design and experimental evaluation of two new, concurrent priority queues: a parallel Fibonacci heap and a concurrent priority pool, and compares them with the concurrent binary heap.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL: Carl E. Landwehr	22b TELEPHONE (Include Area Code): (617) 253-5894 / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

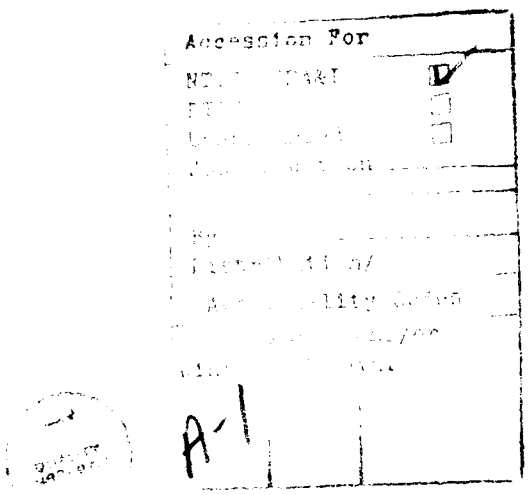
☆U.S. Government Printing Office: 1985—607-047

Unclassified

The parallel Fibonacci heap is based on the sequential Fibonacci heap, which is theoretically the most efficient data structure for sequential priority queues. This scheme not only preserves the efficient operation time bounds of its sequential counterpart, but also has very low contention by distributing locks over the entire data structure. The experimental results show its linearly scalable throughput and speedup up to as many processors as tested (currently 18). A concurrent access scheme for a doubly linked list is described as part of the implementation of the parallel Fibonacci heap.

The concurrent priority pool is based on the concurrent B-tree and the concurrent pool. The concurrent priority pool has the highest throughput among the priority queues studied. Like the parallel Fibonacci heap, the concurrent priority pool scales linearly up to as many processors as tested.

The three kinds of concurrent priority queues, parallel Fibonacci heaps, concurrent priority pools, and concurrent binary heaps, are evaluated in terms of throughput and speedup. Some applications of concurrent priority queues such as the vertex cover problem and the single source shortest path problem are tested.



An Evaluation of Concurrent Priority Queue Algorithms

by

Qin Huang

B.S., University of Science and Technology of China (1988)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

August 1990

Signature of Author ..... Q. Huang
Department of Electrical Engineering and Computer Science
1990

Certified by ..... William E. Weihl
William E. Weihl
Associate Professor of Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

An Evaluation of Concurrent Priority Queue Algorithms

by

Qin Huang

Submitted to the Department of Electrical Engineering and Computer Science
on 1996, in partial fulfillment of the
requirements for the degree of

Master of Science

in Electrical Engineering and Computer Science

Abstract

The priority queue is a fundamental data structure that is used in a large variety of parallel algorithms, such as multiprocessor scheduling and parallel best-first search of state-space graphs. This thesis addresses the design and experimental evaluation of two novel concurrent priority queues: a parallel Fibonacci heap and a concurrent priority pool, and compares them with the concurrent binary heap. The parallel Fibonacci heap is based on the sequential Fibonacci heap, which is theoretically the most efficient data structure for sequential priority queues. This scheme not only preserves the efficient operation time bounds of its sequential counterpart, but also has very low contention by distributing locks over the entire data structure. The experimental results show its linearly scalable throughput and speedup up to as many processors as tested (currently 18). A concurrent access scheme for a doubly linked list is described as part of the implementation of the parallel Fibonacci heap. The concurrent priority pool is based on the concurrent B-tree and the concurrent pool. The concurrent priority pool has the highest throughput among the priority queues studied. Like the parallel Fibonacci heap, the concurrent priority pool scales linearly up to as many processors as tested. The priority queues are evaluated in terms of throughput and speedup. Some applications of concurrent priority queues such as the vertex cover problem and the single source shortest path problem are tested.

Keywords: parallel, concurrent, algorithm, priority queue, pool, B-tree, Fibonacci heap, doubly linked list

Thesis Supervisor: William E. Weihl
Title: Associate Professor of Computer Science

醉翁之意不在酒，
在乎山水之间也。

---欧阳修

Acknowledgments

First, I would like to thank Professor William Weihl, my thesis supervisor, for his great supervision and helpful directions. Also, I would like to thank Professor Barbara Liskov for her insightful discussions at the early stage of the thesis, and her support when I came to MIT to begin my graduate studies. Professor Stephen Ward, my academic advisor, has kindly guided me through my study these years.

Wilson Hsieh and Anthony Joseph have read the thesis and given very helpful comments. Sanjay Ghemawat read the early chapters and provided useful suggestions. Paul Wang wrote the concurrent B-tree code which served as the basis for the concurrent priority pool programs. The members of the large scale parallel software group and programming methodology group make the fifth floor a fun place to work. My officemates Boaz Ben-Zvi and Rivka Ladin showed me around when I joined the group and put up with my messy desks later on. I would like to thank them all.

Lixia Zhang and Feng Zhao have given me advice and encouragement from time to time. Chao Yang Zhang explored many fine beaches and restaurants with me. I wish success to Professor Julian Stanley and his SMPY group at The Johns Hopkins University.

Some of the experiments have been done on an Encore Multimax at the Argonne National Lab Advanced Computing Research Facility(ACRF). I would like to thank them for providing the facilities and putting up with my tying up the machine.

Finally, my heartfelt thanks go to my family members, especially my parents, who have always supported me through my entire career and made my life more enjoyable.

黄沈

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, and in part by the National Science Foundation under grant CCR-8716884.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The priority queue is a fundamental data structure that is used in a large variety of parallel algorithms, such as multiprocessor scheduling and parallel best-first search of state-space graphs[Win84, Nil80, Pea84, KRR88]. In these algorithms, each process performs an access-think cycle. Every process works on its current node (thinking), then accesses the shared priority queue to *insert* nodes if it generated any, *extract* a high priority node to work on next, increase the priorities of some nodes by *decreasing* the keys[1], and *delete* some nodes from the priority queue if they no longer need to be worked on. Sequential priority queues are usually represented as binary heaps, Fibonacci heaps, or B-trees (see Chapter 2). Concurrent priority queues are used in a large number of parallel algorithms. An example is Seneff's speech recognition parser[Sen89], which maintains a priority queue of unparsed grammar nodes with associated priorities, and parses grammar nodes with higher priorities first.

We call the extract operation of a concurrent priority queue *strict* if it extracts the element with the highest priority in the queue. Strict extract operations require some kind of serialization of operations performed on a queue, which increases the contention on the queue. As discussed in section 3.1, most applications only need to extract promising elements that have high priority instead of the highest priority; this fact can be used to decrease contention on the priority queue. However, the promising quality of extracted nodes should be controlled to satisfy the requirements of different applications.

Biswas and Browne [BB87] present a scheme that allows parallel insertions and extractions in strict concurrent binary heaps, but it does not perform better than the serial access scheme even for heaps with 1,000 nodes. In the serial access scheme, each operation

[1] In this thesis, we use small keys to denote high priority.

locks the binary heap exclusively during the whole period of the operation. Rao and Kumar [RK88b] describe a concurrent binary heap algorithm for concurrent priority queues that has less overhead and provides strict extract operations. However, their scheme saturates when the number of processes accessing the priority queue is greater than about ten[2]. More recently, Kumar et al[KRR88] present several "distributed" formulations of priority queues based on binary heaps with relaxed strictness of priority.

This thesis presents the design and experimental evaluation of different implementations of concurrent priority queues. We present a novel concurrent priority queue mechanism based on the Fibonacci heap, which is theoretically the most efficient data structure for the sequential priority queue. This parallel Fibonacci heap provides operations that are theoretically and practically more efficient than the concurrent binary heap. A concurrent access scheme for a doubly linked list is described as part of the Fibonacci heap implementation. We also describe a new concurrent priority queue, the concurrent priority pool, that is based on concurrent B-trees [WW90][LY81][LSS87] and concurrent pools [KE89][Man86]. As shown in Chapter 5, this scheme has the highest throughput among all concurrent priority queues studied here. The performance of different concurrent priority queues is analyzed using the language Mul-T[KHM89] on an *Encore Multimax* shared memory multiprocessor. The performance indicates that both the parallel Fibonacci heap and the concurrent priority pool are linearly scalable and have larger throughput than the concurrent binary heap. The single source shortest path problem and the vertex cover problem are tested as applications of concurrent priority queues.

1.1 Parallel Fibonacci Heap

The parallel Fibonacci heap is based on the sequential Fibonacci heap, which is theoretically the most efficient data structure for sequential priority queues. The critical sections acquired by the operations on the parallel Fibonacci heap are small and distributed over the entire data structure. Therefore, the parallel Fibonacci heap has low contention. The insert operation takes constant time, the decrease key operation takes constant amortized time, and the extract and delete operations take logarithmic time. This scheme provides more scalable operations and higher throughput than current schemes such as the concurrent binary heap. An algorithm for concurrent access to doubly linked lists is

[2]This value depends on the length of the think time. Experimental results are shown in Chapter 5.

described as part of the implementation of parallel Fibonacci heaps.

1.2 Concurrent Priority Pool

Concurrent priority pools are based on concurrent B-trees and concurrent pools. Since the concurrent priority queue employs a distributed-data structure(the pool), the insert and extract operations do not share critical resources in most cases. As shown in Chapter 5, concurrent priority pools have the highest throughput among all concurrent priority queues investigated. Concurrent priority pools also allow tight control over the quality of extracted nodes. Insert operations run in logarithmic time, and extract operations take logarithmic time in the worst case.

1.3 Experimental Environment

I performed most of the experiments on two *Encore* shared memory multiprocessors. One of the Encore machines has 20 processors of which 18 processors can be used for running Mul-T. The concurrent priority queues were implemented in Mul-T, a Lisp-like programming language with futures and locking mechanisms.

1.4 Overview

Chapter 2 describes various implementations of sequential priority queues, such as binary heaps, binomial heaps, Fibonacci heaps, and B-trees.

Chapter 3 presents the data structure and concurrent access algorithms for the parallel Fibonacci heap. The concurrent operations on a doubly linked list are described as part of the implementation.

Chapter 4 presents the data structure of concurrent priority pools and concurrent operations on it.

Chapter 5 gives an experimental analysis of different implementations of concurrent priority queues.

Chapter 6 presents a summary of what has been accomplished and discusses some related research and directions for future research.

Chapter 2

Preliminaries of Sequential Priority Queues

A sequential priority queue is a data structure for maintaining a set of nodes each with an associated key value, and other satellite data. In this thesis, a small key value has higher priority than a larger key value. A priority queue usually supports the following operations:

1. Insert: insert a new node into the queue.
2. Extract: delete the node from the queue whose key is minimum, returning a pointer to the node.
3. Decrease key: make a node in the queue have a new key value which is assumed to be less than its current key value.
4. Delete: delete a node from the queue.
5. Union: create and return a new queue that contains all the nodes of the two input queues.

There are many applications of priority queues. An example is scheduling processes on a shared computer, especially on large multiprocessor systems. The priority queue keeps track of the processes to be performed and their relative priorities. When a processor is idle, it selects a high priority process from the priority queue to work on. Another application is the state space search in many graph algorithms, such as Dijkstra's single source shortest path algorithm(SSSP) and the vertex cover problem(VCP)[CLR90]. In

Operation	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)	B-tree (worst-case)
INSERT	$\Theta(lg\ n)$	$\Theta(lg\ n)$	$\Theta(1)$	$\Theta(lg\ n)$
EXTRACT	$\Theta(lg\ n)$	$\Theta(lg\ n)$	$O(lg\ n)$	$\Theta(lg\ n)$
DECREASE	$\Theta(lg\ n)$	$\Theta(lg\ n)$	$\Theta(1)$	$\Theta(lg\ n)$
DELETE	$\Theta(lg\ n)$	$\Theta(lg\ n)$	$O(lg\ n)$	$\Theta(lg\ n)$
UNION	$\Theta(n)$	$\Theta(lg\ n)$	$\Theta(1)$	Not well supported

Table 2.1: Time bounds of operations on different sequential priority queue implementations

the SSSP algorithm, a priority queue is used to monitor the distance of each vertex from the source, and the algorithm always explores the "closest" vertex first. In the VCP, we use a priority queue to keep track of the state-space search graph.

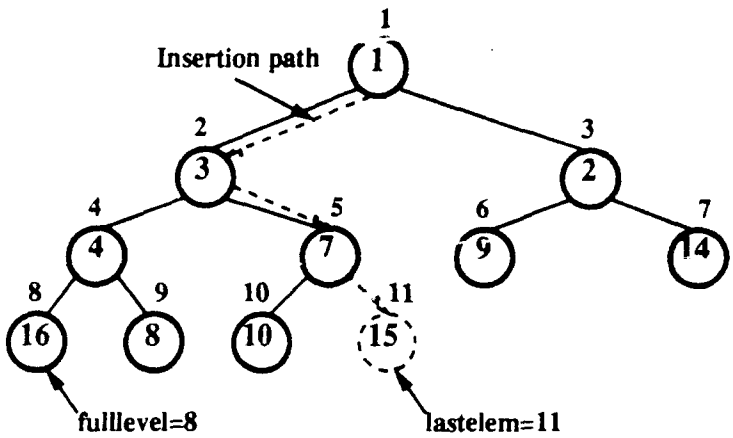
This chapter discusses different implementations of sequential priority queues, binary heaps, binomial heaps, Fibonacci heaps, and B-trees. We adopt the notation from the book *Introduction to Algorithms*[CLR90]. Table 2.1 shows the running times for operations on these four implementations of priority queues. The number of nodes in the heap at the time of an operation is denoted by n.

2.1 Binary Heap

2.1.1 Data Structure

The binary heap can be viewed as a complete binary tree, as shown in Figure 2.1(a), each node of which has a key. The heap satisfies the **heap property**: the value of a node is at least as big as the value of its parent. Thus, the node with the smallest key in a heap is stored at the root, and the subtrees rooted at a node contain larger values than the node. The tree is completely filled on all levels except possibly the bottom level, which is completely filled from the left up to a point.

Before presenting the access schemes for a binary heap, we first briefly describe an efficient representation of a binary heap using an array, as shown in Figure 2.1(b). Each node of the tree corresponds to an element of the array. The root occupies location 1



(a)

1	2	3	4	5	6	7	8	9	10	11	
1	3	2	4	7	9	14	16	8	10	15	...

(b)

Figure 2.1: A binary heap (a) viewed as a binary tree (b) represented as an array. The number within the circle representing a node in the tree is the value stored at that node. The number next to a node is the corresponding index in the array.

---

and node i occupies location i. The left child of node i, $LCHILD(i)$, occupies location $2i$ and its right child, $RCHILD(i)$, occupies location $2i + 1$. The parent of node i is at $\lfloor \frac{i}{2} \rfloor$. Associated with the heap are data fields *lastelem* and *fulllevel*, in which *lastelem* is the index of the last non-empty node of the heap and *fulllevel* is the index of the first node at the bottom level of the heap that contains at least one non-empty node. For an empty heap, *lastelem* = *fulllevel* = 0. An empty node has a special key called $MAXINT$ whose value is ∞. Figure 2.1 shows a heap with 11 keys, and the values of *lastelem* and *fulllevel*.

2.1.2 Operations on a Binary Heap

The operations usually performed on a binary heap are insertion and extraction. Here we show the algorithms [RK88a] for doing insertions and deletions; both proceed from the root to the bottom of a binary tree.

The insert operation adds a node into the binary heap. Let *target* be the first empty node in the heap; this will be the last non-empty node after the insertion. The **insertion path** is the path between the root and target. Figure 2.1(a) shows a ten node heap, to which the eleventh node is being added. The insertion path can be traversed starting from the root as follows. Let I be the displacement of target at the bottom level(i.e., $I = lastelem - fulllevel$) and P be the length of the insertion path. If we view I as a P bit binary number, the bits of the binary representation of I (from the most significant to the least significant) tell us whether to go right (if 1) or left (if 0) when we go from the root downward. In the example in Figure 2.1(a), $fulllevel = 8$, $target = 11$, so $I = 3 = (011)$ in binary representation. This means that we can go from the root to the target by following left, right, and right branches at successive levels. The algorithm is given in Figure 2.2.

Figure 2.3 shows the pseudocode for the delete operation. It removes the root of the heap and places the key of the last non-empty node of the heap at the root. The heap property may now be violated at the root of the heap. Reheapification is performed by repeatedly pushing this key downward until the heap property is satisfied.

Since a heap of n nodes is based on a complete binary tree, its height is $\Theta(lg\ n)$. The insert and extract operations run in time at most proportional to the height of the tree; thus, these operations take $O(lg\ n)$ time.

2.2 Fibonacci Heap

Fibonacci heaps were introduced by Fredman and Tarjan[FT87]. The Fibonacci heap has the best amortized time bound for all operations among the implementations listed in Table 2.1. From a theoretical point of view, Fibonacci heaps are especially desirable when the number of extract-min and delete operations is small relative to the number of other operations performed. This situation arises in many applications, such as computing minimum spanning trees[CLR90] and Dijkstra's algorithm for finding single source shortest paths[CLR90]. From a practical standpoint, the Fibonacci heap is generally regarded as being only of theoretical interest because of its code complexity and constant overhead. However, for parallel applications, the time spent on acquiring critical resources, like locking and waiting, can be dominant over the constant overhead. In fact, the experimental results in chapter 5 show that the parallel Fibonacci heap is more scalable and efficient than the concurrent binary heap whose code is much shorter. We first examine a simpler data structure, the *binomial heap*, which is the basis for the Fi-

```
    proc insert(heap, nkey)
    % insert a new nkey into heap

 1  lastelem := lastelem + 1
 2  target := lastelem
 3  if (lastelem ≥ fulllevel*2) then
 4          fulllevel := lastelem
 5  end
 6  i := target - fulllevel   % i is the displacement of target
 7  j := fulllevel/2      % j = 2^(length of insertion path - 1)
 8  p := 1    % p is the current position in the insertion path

    % Reheapification loop
 9  while (j ≠ 0)
10      if (key[p] > nkey) then
11          Exchange(nkey, key[p])
12      end
13      if (i ≥ j) then
14          p := rchild(p)
15          i := i - j
16      else
17          p := lchild(p)
18      end
19      j := j/2
20  end
21  key[p] := nkey
22  end insert
```

Figure 2.2: Insert operation on binary heap

bonacci heap. We then present an analysis of the data structure and the operations on the Fibonacci heap.

2.2.1 Binomial Heap

A binomial heap is a collection of binomial trees. The **binomial tree** B_k is defined recursively. The binomial tree B_0 consists of a single node. The binomial tree B_k consists of two binomial trees B_{k-1} that are linked together: the root of one tree is the leftmost child of the root of the other. The binomial tree B_k has the following properties,

1. There are 2^k nodes,

2. The height of the tree is k,

```
    proc delete(heap)
 1  if (lastelem = 0) then
 2          return nil
 3  end
 4  least := key[1]      % root resides at the location 1 of the array
 5  i := 1
 6  j := lastelem
 7  lastelem := lastelem - 1
 8  if (lastelem < fulllevel) then
 9      fulllevel := fulllevel/2
10  end
11  if (j = 1)
12      key[1] := MAXINT
13      return least
14  end
15  key[1] := key[j]
16  key[j] := MAXINT

    % Reheapification loop
    % let min-son(i) be the index of the son of i which has smaller key
17  while (key[i] > key[min-son(i)]) do
18      Exchange(key[i], key[min-son(i)])
19      i := min-son(i)
20  end
21  return least
22  end delete
```

Figure 2.3: Delete operation on binary heap

---

3. The root has degree k, which is greater than that of any other node; if the children of the root are numbered from left to right by $k-1, k-2, ..., 0$, child i is the root of a subtree B_i.

A **binomial heap** h is a set of binomial trees that satisfies the following **binomial-heap properties.**

1. Each binomial tree in h is **heap-ordered**: the key of a node is greater than or equal to the key of its parent.

2. There is at most one binomial tree in h whose root has a given degree.

The first property tells us that the root of a heap-ordered tree contains the smallest key in the tree. The second property implies that an n-node binomial heap h consists of at most $\lfloor lg\ n \rfloor + 1$ binomial trees.

The insert operation on binomial heaps creates a new tree on its own of degree 0. This may now violate the bionomial heap property 2 above, since there may be another tree of degree 0. If there is another tree of degree 0, the two degree 0 trees are merged into a single tree of degree 1 by making one tree a child of the other according to the heap-order rule (i.e., the root of the tree with the larger key is made a child of the root of the tree with the smaller key). This may again violate the bionomial heap property; if so, we continue merging in recursive fashion. Thus, the insertion operation runs in time at most proportional to the number of binomial trees, which is $\Theta(lg\ n)$.

The extract operation is very similar to the insert operation, and also takes time $O(lg\ n)$. The worst-case time bounds for the binomial heap are shown in Table 2.1. In particular, the *Union* operation takes only $O(lg\ n)$ time to merge two binomial heaps with a total of n elements, which is better than the $O(n)$ time for the binary heap.

2.2.2 Structure of Fibonacci Heap

Like a binomial heap, a **Fibonacci heap** is a collection of trees. However, a Fibonacci heap is a more "relaxed" data structure than a binomial heap: the trees in a Fibonacci heap are not constrained to be as those in a binomial heap, in that there may be many trees of a given degree as opposed to only one for a given degree in a binomial heap. Furthermore, an interior node of a tree may lose at most one child after it becomes an interior node and a root node may lose multiple children. This more relaxed structure allows for improved operation time bounds by delaying work that maintains the structure until it is convenient to perform.

As Figure 2.4 shows, a Fibonacci heap is a collection of trees whose roots are linked in a circular, doubly linked list called the **root list**; the heap is accessed through a *min* pointer to the root of the tree containing a minimum key. An empty heap has a *nil* min pointer. Each node x in a tree contains a pointer $p[x]$ to its parent and a pointer $child[x]$ to any one of its children. The children of x are linked together in a circular, doubly linked list called the **child list** of x. Each child y in a child list has pointers $left[y]$ and $right[y]$ that point to y's left and right siblings, respectively. The number of children in the child list of node x is stored in $degree[x]$. The boolean-valued field $mark[x]$ indicates whether node x has lost a child since the last time x was made the child of another node. The *mark* field is used only in decrease and delete operations.

Circular, doubly linked lists(DLL) have two advantages for use in Fibonacci heaps. First, we can remove a node from a circular, doubly linked list in $O(1)$ time. Second, given two such lists, we can concatenate them into one circular, doubly linked list in $O(1)$

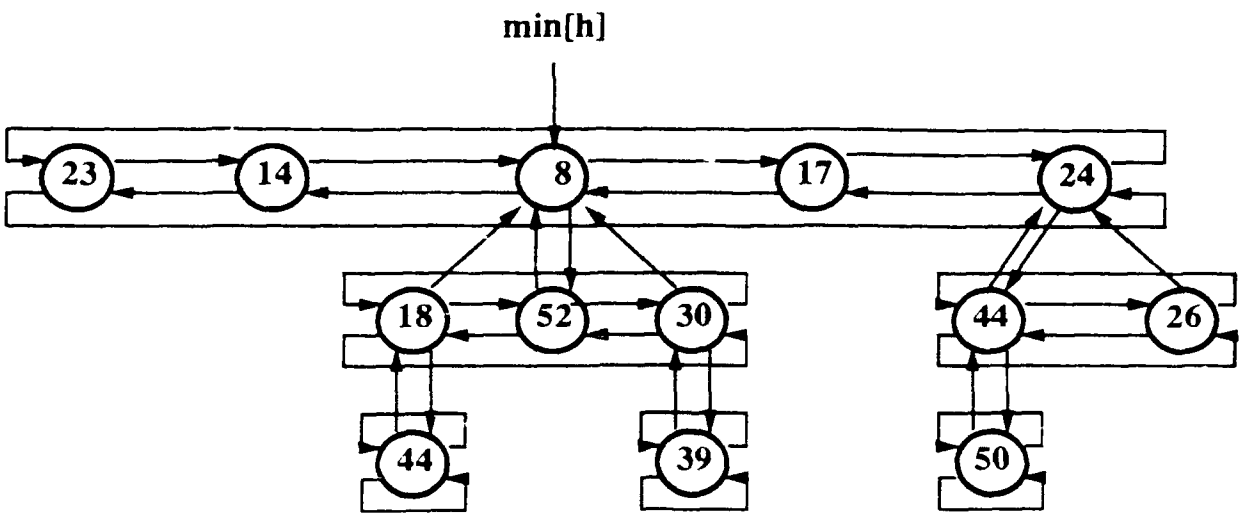


Figure 2.4: An example of Fibonacci heap

time. I have designed a parallel access scheme for DLL, described in section 3.2, that preserves the above two advantages.

2.2.3 Insert Operation

To insert a node into a Fibonacci heap, we only need to insert the node into the root list of the heap and return a pointer to it. If the heap was empty, or the newly inserted node has a smaller key than that of the minimum node, *min* is changed to point to the new node. The insertion only takes constant time compared to $\Theta(lg\ n)$ in the binary heap and the binomial heap. Figure 2.5 shows the pseudo code for the insert operation.

2.2.4 Extract Operation

The process of extracting the minimum node consists of two steps. The first step, finding the minimum node and removing it from the heap, is not hard, since we have the *min* pointer to the minimum node. The pseudo code for extracting the minimum node is shown in Figure 2.6.

In the second step, as shown in Figure 2.7, we reduce the number of trees in the Fibonacci heap and find a new minimum node by **consolidating** the root list of the Fibonacci heap. Consolidating the root list consists of repeatedly executing the following

```
    proc insert(h, x)
    % insert new node x into heap h

1   Initialize node x by updating its degree, p, child,
2       left, right, and mark fields properly
3   Put x into root list of h
4   if (min[h] = nil) or (key[x] < key[min[h]]) then
5           min[h] := x
6   end
```

Figure 2.5: Insert operation of Fibonacci heap

steps until every root in the root list has a distinct *degree* value.

1. Find two roots x and y in the root list with the same degree, where $key[x] \leq key[y]$.

2. Link y to x: remove y from the root list, and make y a child of x.

In lines 16-23, the consolidation process finds the current minimum node in the root list. The amortized time taken by the extract operation is $O(lg\ n)$.

2.2.5 Decrease Key Operation

The decrease key operation for a Fibonacci heap is shown in Figure 2.8. To decrease the key of node x to a value k, we first replace x's key with k in lines 1-4. If the heap-order is violated(i.e., $k < key[y]$ where y is the parent of x), we *cut* x from y in line 7, and make x a root. From the Fibonacci heap constraints, an interior node can only lose one child; further **cascading cuts** are performed at line 8 to satisfy this constraint. The amortized cost of the decrease key operation is $O(1)$.

2.2.6 Delete Operation

Deleting a node x from a Fibonacci heap can be viewed as making node x the minimum node in the heap by decreasing its key to $-\infty$, then removing node x from the Fibonacci heap with the extract operation; this is shown in Figure 2.9.

The amortized time of delete is the sum of the $O(1)$ amortized time of decrease key and the $O(lg\ n)$ amortized time of extract.

```
    proc extract(h)

 1  z := min[h]
 2  if (z ≠ nil) then
 3        for each child x of z do
 4              add x to the root list of h
 5              p[x] := nil
 6        remove z from the root list of h
 7        if (z = right[z]) then
                % z is the only node in the heap
 8              min[h] := nil
 9        else
10              min[h] := right[z]
                % consolidate the heap and find next min
11              consolidate(h)
12        end
13  end
14  end extract
```

Figure 2.6: Extract operation of Fibonacci heap

The delete operation could be improved by directly removing the node from the heap instead of first putting it into root list and then taking it out. However, the amortized time bound would not improve.

2.3 B-Tree

B-trees [BS77][Com79] are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage devices. The guaranteed small search, insertion, and deletion time of B-trees makes them quite appealing for database applications. Nevertheless, we will see later on that the B^+-tree[MR85], a variant of the B-tree, could also serve as a priority queue. In this section, we briefly describe the B^+-tree that is well suited for use in a concurrent database system. More information can be found in [CLR90][LY81][Wed74]. For simplicity, we denote B^+-tree as B-tree in this thesis.

Figure 2.10 shows an example of B-tree internal and leaf nodes. A B-tree has the following major properties:

1. Each path from the root to any leaf has the same length, h.

2. Each node contains at most $2k + 1$ elements, in which k is a tree parameter. Each node contains at least one element. There are other variations of B-trees that

```
    proc consolidate(h)

    % initialize an array for compacting trees with the same degree
 1  for i := 0 to DEGREE-UPPER-BOUND do
 2        A[i] := nil
    % compact the trees with the same degree
 3  for each node w in the root list of h do
 4        x := w
 5        d := degree[x]
 6        while (A[d] ≠ nil) do
 7              y := A[d]
 8              if (key[x] > key[y]) then
 9                    Exchange(x, y)
10                    link(h, y, x)
11                    A[d] := nil
12                    d := d + 1
13              end
14        end
15        A[d] := x

    % find the next node with the minimum key
16  min[h] := nil
17  for i := 0 to DEGREE-UPPER-BOUND do
18        if (A[i] ≠ nil) then
19              add A[i] to the root list of h
20              if (min[h] = nil) or (key[A[i]] < key[min[h]]) then
21                    min[h] := A[i]
22              end
23        end
24  end consolidate


    proc link(h, y, x)

 1  remove y from the root list of h
 2  make y a child of x, incrementing degree[x]
 3  mark[y] := false
 4  end link
```

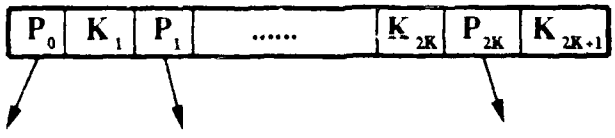
Figure 2.7: Consolidate operation of Fibonacci heap

```
    proc decrease(h, x, k)
    % decrease the key of x to k

 1  if (k > key[x]) then
 2        error "new key is greater than current key"
 3  end
 4  key[x] := k
 5  y := p[x]
 6  if (y ≠ nil) and (key[x] < key[y]) then
          % the heap order is violated
 7        cut(h, x, y)
 8        cascading-cut(h, y)
 9  end
10  if (key[x] < key[min[h]]) then
11        min[h] := x
12  end
13  end decrease


    proc cut(h, x, y)

 1  Remove x from the child list of y, decreasing degree[y]
 2  Add x to the root list of h
 3  p[x] := nil
 4  mark[x] := false
 5  end cut


    proc cascading-cut(h, y)

 1  z := p[y]
 2  if (z ≠ nil) then
 3        if (mark[y] = false) then
                % y has lost one child
 4              mark[y] := true
 5        else
                % y has lost two children
 6              cut(h, y, z)
 7              cascading-cut(h, z)
 8        end
 9  end
10  end cascading-cut
```

Figure 2.8: Decrease operation of Fibonacci heap

```
proc delete(h, x)
% delete node x from Fibonacci heap x

1  decrease(h, x, -∞)
2  extract(h)
3  end delete
```

Figure 2.9: Delete operation of Fibonacci heap



(a) B-tree internal node

(b) B-tree leaf node

Figure 2.10: Structure of B-tree

require each node to contain at least $k + 1$ elements.

3. The keys of all of the data in the B-tree are stored in the leaf nodes. Nonleaf nodes contain pointers and the key values to be used in following those pointers.

4. Within each node, the keys are in ascending order.

5. In nonleaf nodes, each pointer, P_i, points to a subtree T_i whose root is the node that P_i points to. The values stored in T_i are bounded by the two key values, K_i and K_{i+1}, to the "left" and "right" of P_i in the node(i.e., the set of values stored in subtree T_i is bounded by $K_i < v \leq K_{i+1}$).

B-trees have internal nodes that look like those shown in Figure 2.10(a). The K_i are instances of the key domain, and the P_i are pointers to other nodes. On the leaf level, B-tree nodes, as shown in Figure 2.10(b), contain keys and other information associated with them.

To insert a new node with key *newkey* into the B-tree, we start from the B-tree root and move downwards from each nonleaf level following the pointer P_i that has two neighbors K_i and K_{i+1} satisfying $K_i < newkey \leq K_{i+1}$. When a leaf is found, *newkey* is inserted if there is room; otherwise, the leaf is split, and the split may propagate back up the tree.

The delete operation first locates the leaf that stores the key *oldkey* to be deleted. The locating process is just like that in the insert operation. Once the leaf is found, *oldkey* is removed from it. If the leaf is then empty, it is merged with its neighbor, and the merge may propagate back up the tree.

To use a B-tree as a priority queue, the insert operation remains the same; the extract operation is implemented by deleting the smallest key from the leftmost leaf of the B-tree. In fact, if we maintain a direct pointer to the leftmost leaf of the B-tree, we can avoid the locating process used in the delete operation.

The insert operation takes time proportional to the height of the B-tree, $O(lg\ n)$, where n is the number of keys stored in the tree, and the extract operation takes time $O(lg\ n)$ including merging leaves and internal nodes.

Chapter 3

Parallel Fibonacci Heap and Concurrent Access Algorithms

In this chapter, we present our design for a parallel Fibonacci heap that is based on the sequential Fibonacci heap described in Chapter 2. The parallel Fibonacci heap maintains the advantages of its sequential counterpart, i.e., its asymptotically more efficient operations, and it also has linearly scalable throughput as shown in Chapter 5. The parallel Fibonacci heap reduces contention by weakening the semantics of the extract operation: an extract operation need not return the minimum element in the heap, instead it can return a promising element close to the minimum where the promising quality can be controlled. The non-strict semantics of the extract operation for the parallel Fibonacci heap is elaborated in Section 3.1. Section 3.2 presents a concurrent access algorithm for a doubly linked list. Section 3.3 gives a description of the data structure of the parallel Fibonacci heap. The concurrent access algorithms are presented in Section 3.4. Section 3.5 summarizes this chapter.

3.1 Semantics of Parallel Fibonacci Heap

The semantics of the *insert*, *decrease*, and *delete* operations on a parallel Fibonacci heap remain the same as on a sequential Fibonacci heap presented in Section 2.2, but the semantics of the *extract* operation are **non-strict**. The sequential Fibonacci heap has a strict extract operation in the sense that it always extracts the minimum node from the heap. However, for parallel Fibonacci heaps, since there are potentially many processes extracting nodes concurrently, strict semantics are undesirable for two reasons:

- In terms of correctness, strict semantics are not required in most, if not all, parallel priority queue applications. However, it is usually desirable to control the quality of extracted nodes to meet applications' requirement. The strict extract operation usually involves more contention, and doesn't extract more promising nodes overall. For example, suppose there are 5 processes concurrently trying to extract nodes from a priority queue that contains 5 highest priority nodes $n1$, $n2$, $n3$, $n4$, and $n5$. In the case of strict semantics, the extract operations have to be serialized and get $n1$ to $n5$ one at a time. This creates a bottleneck. If we adopt non-strict semantics, we potentially can extract $n1$ to $n5$ concurrently without blocking, and the extracted nodes $n1$ to $n5$ will be the same as those extracted with strict semantics, although the order in which they are extracted may differ. The concurrent access algorithms presented in Section 3.4 provide methods to control the promising extent of extracted nodes.

- Realizing strict semantics for parallel implementations is expensive, since we have to linearize all operations; this creates severe bottlenecks. There is a tradeoff between strictness and contention. The stricter the semantics, the greater the contention on a priority queue. The experiments in Chapter 5 show that a strict scheme for a concurrent binary heap saturates when the number of processes is more than about eight.

Instead of having a *min* pointer to the minimum node in the heap, our parallel Fibonacci heap has a *promising list* that is an array of pointers to some promising nodes in the root list. We will look into the extract operation in section 3.4.

3.2 Concurrent Operations on a Doubly Linked List

A **doubly linked list**(DLL) is a data structure in which the objects are arranged in linear order and every object has a *key* field and two other fields: *left* and *right*. Given an object x in a doubly linked list, *right[x]* points to its successor in the list, and *left[x]* points to its predecessor. The *insert* and *delete* operations take only constant time provided that we know where to insert an object and which object to delete. Searching an n-object list takes $O(n)$ time.

Concurrent insert and delete operations are more complicated than their sequential counterparts. Let's consider concurrent insertion, concurrent deletion, and concurrent insertion and deletion separately.

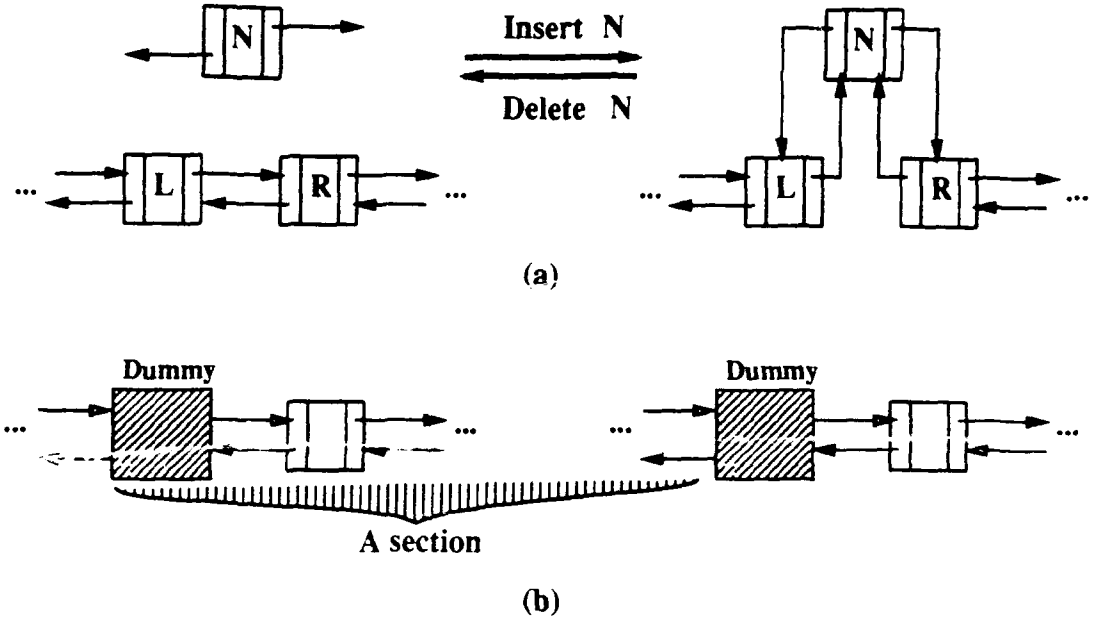


Figure 3.1: Concurrent insertion on DLL

3.2.1 Concurrent Insertion on DLL

Inserting a node N into DLL $LIST$, as shown in Figure 3.1(a), takes two steps:

1. Find two neighbor nodes L and R in $LIST$ to insert N between.
2. Modify the right field of L, the left and right fields of N, and the left field of R.

In the second step, we have to ensure that the fields are updated atomically. Doing so involves locking certain fields in some nodes (e.g., the right field of L). However, this could cause a bottleneck if there are many processes trying to insert new nodes between L and R, as they all have to lock the right field of L during insertion. Thus, it would be better to spread out insertions among the nodes in $LIST$, preferably as evenly as possible. One way to do this is to place a set of **dummy nodes** in $LIST$, as shown in Figure 3.1(b). Dummy nodes are similar to normal nodes in the DLL, except they are marked *dummy*, can be accessed directly[1], and remain in the DLL all the time. We define

[1]For example, we can have an array of pointers to the dummy nodes so that they can be accessed directly from the array.

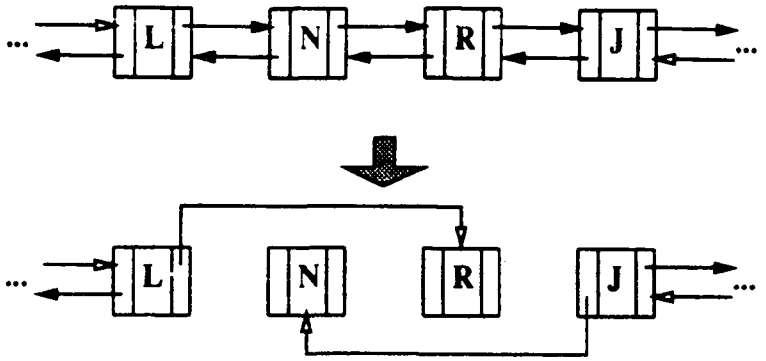


Figure 3.2: A scenario of concurrent deleting N and R of DLL without locking them

a *section* of DLL to be the sub-DLL between two dummy nodes as shown in Figure 3.1(b). The insert operation on $LIST$ is now the following:

1. Randomly choose a dummy node D. If D's right field is locked, we can try another dummy node; otherwise, lock D's right field.

2. Insert the new node to the right of D, and update the right field of D, the left and right fields of the newly inserted node, and the left field of D's old right neighbor.

The number of dummy nodes needed in $LIST$ depends on the access frequency and applications. We will see in the following section that the dummy nodes also help the delete operation.

3.2.2 Concurrent Deletion on DLL

Deleting node N from its two neighbors L and R, as shown in Figure 3.1(a), changes the right field of L and the left field of R. The left and right fields of N may also need to be changed. The right field of L and the left field of R have to be locked for proper deletion. Moreover, the left and right fields of N must be locked too. Otherwise, the following scenario may arise when deleting N and R concurrently, as shown Figure 3.2, which results in a broken list.

Delete N	**Delete** R
Lock $right[L]$	Lock $right[N]$
Lock $left[R]$	Lock $left[J]$
Set $right[L]$ pointing to R	Set $right[N]$ pointing to J
Set $left[R]$ pointing to L	Set $left[J]$ pointing to N
Clear $left[N]$, $right[N]$	Clear $left[R]$, $right[R]$

To avoid deadlock, we lock the fields in a particular order: first lock the right field of L, then the left and right fields of N, finally the left field of R. We could still deadlock if we did not have dummy nodes in the $LIST$. One example is to delete the only node N in a circular DLL. In this case, N itself is both its left and right neighbor, which will cause the locking process, described above, to deadlock. This problem could be avoided by keeping track of the number of nodes in the circular DLL, and treating deletion of the only node in a circular DLL as a special case. However, there is another situation that is similar to the dining philosophers problem and that can't be gracefully avoided without dummy nodes. Suppose there are n nodes in the circular DLL $LIST$ and n processes deleting nodes concurrently in a conspired way: each process is deleting a different node, and each process is executing the locking process synchronously. This will create a circular locking chain. Dummy nodes will prevent this form of deadlock chain.

Dummy nodes are not sufficient to prevent all locking problems. Consider the following situation: while deleting N, we have to lock the right field of L. We find L by using $left[N]$. But at the time of the lookup, $left[N]$ has not been locked, which means the field may be changed by another process. Although this problem can be overcome by using complex locking methods, the method described below using scavenger processes seems simpler and more elegant.

3.2.3 Concurrent Insertion and Deletion on DLL

The complexity of parallel operations on this relatively simple data structure is caused by allowing the concurrent removal of nodes from the list. We can get better performance if we disallow concurrent removals in the following way: deleting N only marks N as *dead*, and all dead nodes are actually removed from the DLL by scavenger process(es), which run as background or periodic foreground processes. Each scavenger process locks one section, and removes dead nodes from that section. Since the DLL is nicely divided by the dummy nodes into *sections*, we avoid deadlock and interference problems by allowing

```
    proc insert(obj, dll)
    % Insert an obj into doubly linked list dll

 1  Randomly find a unlocked dummy node d in dll, and lock right[d]
 2  Insert obj to the right of d
 3  Unlock d
 4  end insert


    proc delete(obj, dll)
    % Delete obj from doubly linked list dll

 1  Mark obj to be dead
 2  Occasionally do
 3        Randomly find a unlocked section s and lock it
 4        for every obj in s do
 5              if (obj is not the right neighbor of a dummy node) and
 6                  (obj is marked dead)  then
 7                     remove obj from dll
 8              end
 9        unlock s
10  end delete
```

Figure 3.3: Concurrent operations on doubly linked list

at most one scavenger process to operate on each section. This kind of distributed scavenging method alleviates the complex locking problem described in the last section.

Figure 3.3 gives the pseudocode for concurrent operations on a DLL. The insertion operation is the same as that described in Section 3.2.1. The delete operation occasionally locks a section, and removes dead nodes in it. With the help of dummy nodes, there is not much contention on the DLL. The insert and delete operations on a DLL still take constant time.

3.3 Data Structure of Parallel Fibonacci Heap

A parallel Fibonacci heap, as shown in Figure 3.4, is a collection of trees whose roots are linked in a circular DLL with dummy nodes as described in Section 3.2. Instead of having one *min* pointer to the root of the tree containing a minimum key, there is an array of pointers to the roots of the trees having promising keys. The array is called the **promising list**. For convenience, we use "node in promising list" to mean "node pointed to by some pointer in the promising list" in this thesis. There is a lock

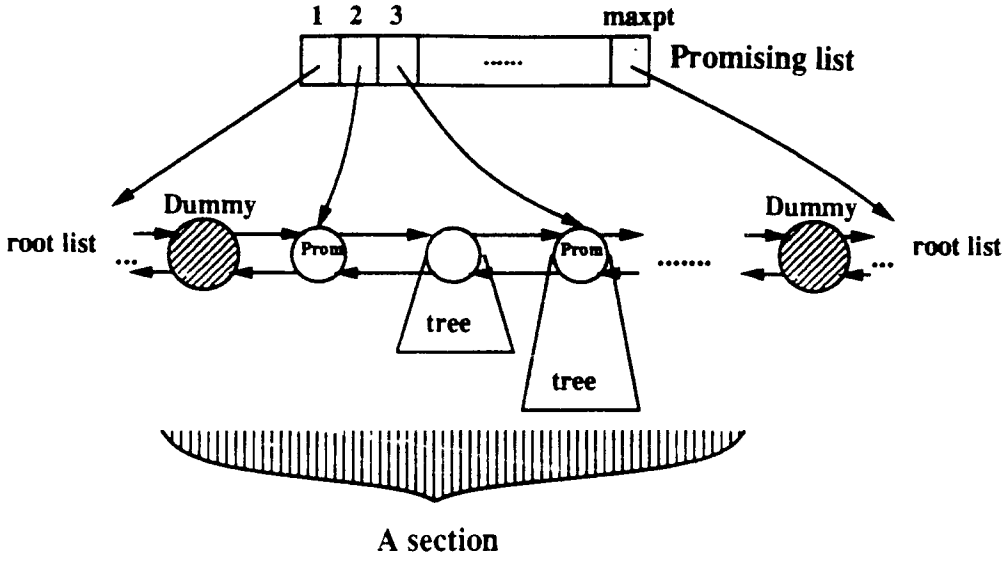


Figure 3.4: Structure of parallel Fibonacci heap

associated with each pointer in the promising list. The size of the promising list, *maxpt*, is a parameter that can be controlled in the algorithm. Besides having the fields of their sequential counterparts, such as left, right, parent, child, key, degree, and mark, the nodes in a parallel Fibonacci heap have some synchronization fields — there are three locks associated with the left, right, and key fields of a node, respectively. In addition, the mark of a node can be one of dummy, dead, promising, unmarked, and marked. Dummy means the node is a dummy node as described in section 3.2, dead means the node has been deleted, promising means the node is a promising node, and unmarked and marked are used in the same way as in the sequential algorithms to denote whether the node has lost a child since it became an interior node. As in the DLL, a section of a parallel Fibonacci heap contains the trees between two dummy nodes as shown in Figure 3.4.

3.4 Concurrent Access Algorithms

In this section, the concurrent access algorithms for the parallel Fibonacci heap are presented. In these algorithms, we use a method to minimize blocking time and enhance throughput called the *check-lock-verify* method. The check-lock-verify method is a high-level, efficient, non-blocking test&do atomic operation, which is described as the

"*cheating*" method in [Bir89]. Here is a comparison of test&do and check-lock-verify:

check-lock-verify	*test&do*
If (conditions are met) then	
Lock critical section	Lock critical section
If (verify conditions are met) then	If (test conditions are met) then
do things in critical section	do things in critical section
else	else
Unlock and exit	Unlock and exit
endif	endif
endif	

The check-lock-verify method asynchronously checks conditions before entering the critical section, while test&do enters the critical section first. In this way, the check-lock-verify method avoids some possible blocking time on the critical section, if the conditions are not met. However, the semantics of the check-lock-verify method are different from those of test&do in the sense that the latter is stricter. Test&do guarantees that the conditions are checked inside a critical section, while the check-lock-verify method first checks the conditions outside the critical section. Only when the conditions can be correctly atomically read[2], are the semantics of test&do and check-lock-verify the same. There are many places in the algorithm where the check-lock-verify method can be used. The check-lock-verify method makes programs look more complex and harder to understand, thus, it is normally not included in the pseudocode listings presented in this section.

3.4.1 Insert Operation

As shown in Figure 3.5, inserting a new key k into a parallel Fibonacci heap h is very similar to inserting a key into a DLL. First a new heap node n is created with key k, and the other fields are properly set. In lines 2-5, we randomly find a dummy node D in the root list, lock the right field of D, and insert the new node to the right of D. Actually, if we find that $right[D]$ has already been locked while trying to lock it at line 3, another dummy node can also be tried. The insert operation ensures that all nodes are inserted evenly among the dummy nodes in the root list.

In lines 6-8, we check whether the newly inserted node n with key k is promising; this is similar to checking whether the newly inserted node is better than *min* in the insert

[2]These features are often machine dependent. The programmer should always check these features before taking advantages of them.

```
    proc insert(id, h, k)
    % Insert key k into parallel Fibonacci heap h. id is an issued worker id

 1  Initialize a new node n with key k
 2  Randomly choose a dummy node D in the root list
 3  Lock right[D]
 4  Put n to the right of D
 5  Unlock right[D]
 6  if good(id, h, k) then
 7        check-promising(h, n)
 8  end
 9  return n
10  end insert


    proc good(id, h, k)
    % a heuristic function that tests whether k has a good chance to be promising

 1  if (k > last-extract[id] * strictness) then
 2        return NO
 3  else
 4        return YES
 5  end
 6  end good
```

Figure 3.5: Insert operation on parallel Fibonacci heap

operation on a sequential Fibonacci heap. In order to avoid checking some "obvious" non-promising nodes, a heuristic function *good* is designed to filter out most non-promising nodes. If the heuristic function says k is good, then we actually check whether node n is promising, as presented in the next section; otherwise, the node n still has a chance to be put into the promising list by the consolidation process described in section 3.4.4.

I have designed a simple "distributed" heuristic function as shown in Figure 3.5. Suppose there is a fixed number of workers doing operations concurrently on the parallel Fibonacci heap (see chapter 5); each worker is assigned an *id* to distinguish it from the others. If a given application doesn't fit this worker model, we can still map the operations performed by the application on the parallel Fibonacci heap to some number of virtual workers. Worker *id* keeps track of the key of the node it most recently extracted in $last - extract[id]$; this is used as a rough measure of whether a key k is good or not. If k is greater than $last - extract[id] \times strictness$, in which *strictness* is a tunable parameter(usually set to be around 1), then k is not treated as good. The heuristic function gives real promising nodes a chance to bypass the consolidation process and

```
    proc check-promising(h, n)
    % Check if node n is more promising than any already promising node prom-one, then
    %       replace prom-one with n in the promising list.

 1  for every pointer prom-pt in promising list do
 2        Lock prom-pt  % if prom-pt is locked, we can try next
 3        if prom-pt = nil then
 4              Lock key[n]
 5              if (mark[n] ≠ dead) and (mark[n] ≠ promising)
 6                  and (parent[n] = nil) then
 7                    mark[n] := promising
 8                    prom-pt := &n
 9                    Unlock key[n]
10                    Unlock prom-pt
11                    return YES
12              end
13              Unlock key[n]
14              Unlock prom-pt
15        else
16              prom-one := *prom-pt
17              Lock key[prom-one]
18              Lock key[n]
19              if ((mark[n] ≠ dead)
20                   and (mark[n] ≠ promising)
21                   and (parent[n] = nil)
22                   and ((mark[prom-one] = dead)
23                         or ((mark[prom-one] = promising)
24                               and (key[prom-one] > key[n])))) then
25                    mark[n] := promising
26                    if mark[prom-one] = promising then
27                          mark[prom-one] := unmarked
28                    end
29                    prom-pt := &n
30                    Unlock key[n]
31                    Unlock key[prom-one]
32                    Unlock prom-pt
33                    return YES
34              end
35              Unlock key[n]
36              Unlock key[prom-one]
37              Unlock prom-pt
38        end
39  return NO
40  end check-promising
```

Figure 3.6: Check whether a node is promising in parallel Fibonacci heap

directly be put into the promising list. We can tune *strictness* to control the quality of nodes in the promising list. The smaller the value of *strictness*, the better the nodes in the promising list, and possibly the longer it takes to find a promising node. So, there is a tradeoff here between *strictness* and contention on the queue. The experiments described in Chapter 5 show how the throughput varies with strictness. Moreover, *strictness* can be made adaptive depending on the feedback of check-promising: if check-promising always returns yes which means the heuristic function may be too strict, then *strictness* can be loosened to some degree; if check-promising always returns no, which means the heuristic function may be too loose, then *strictness* can be tightened a bit.

3.4.2 Check-Promising

Figure 3.6 shows how to check if node n is more promising than one of the already promising nodes in a parallel Fibonacci heap h. Basically, n is compared with every node in the promising list; if a *nil* pointer in the promising list or a promising node with key larger than $key[n]$ is found, then n is put in the promising list; otherwise n is simply left in the root list. In the pseudocode, lines 1-2 loop over all pointers in the promising list, and try to lock each one before checking. In fact, if the pointer *prom-pt* is found already locked in line 2, we can try other pointers in the promising list. If prom-pt is a *nil* pointer, lines 4-14 check if n is not dead or promising and n is in the root list, then put n into the promising list by changing prom-pt to point to n. If *prom-pt* is not nil, lines 16-39 test if n is more promising than node *prom-one* pointed by *prom-pt*, then replace *prom-one* with n. Lines 19-21, like lines 5-6, make sure that n is not dead, is not already promising, and is in the root list before making it promising.

The check-promising procedure is non-blocking in the sense that it does not block on a locked pointer in the promising list; instead it always tries to find a free promising pointer to lock. Also, since the heuristic function *good* filters out most non-promising nodes from being checked, there should not be much contention on the promising list. The time taken to check whether a node is promising is constant, $O(maxpt)$.

3.4.3 Extract Operation

Figure 3.7 shows how to extract a node from a parallel Fibonacci heap h. Since we already have the promising list, if it is not empty then we can randomly remove a promising node from it; otherwise, we find several promising nodes to put in the promising list by consolidating a section of the heap, and retry the extract operation.

Line 1 randomly chooses a pointer *prom-pt* from the promising list. Then we try to lock *prom-pt* in line 6; if it has already been locked, we try another pointer in the promising list. Line 7 checks if *prom-pt* is nil; if it is, we pick up another pointer from the promising list and repeat the process of locking and checking *prom-pt*. Otherwise, we lock the node *prom-one* pointed to by *prom-pt*. If *prom-one* is indeed a promising node, we put its children, if any, into the root list, and take *prom-one* out of the promising list by marking it *dead* in lines 14-21. If *prom-one* is not promising, we simply try other pointers in the promising list in lines 23-25. If after trying "enough" times, we still fail to find a promising node, then it is time to consolidate the heap in lines 3-5; that will compact trees together, and find some promising nodes to put in the promising list.

The promising list is implemented as an array in which each pointer can be directly accessed, and the size of the promising list can be controlled[3] to reduce contention. The extract operation never blocks on a locked pointer in the promising list; therefore, we do not expect much contention on grabbing a pointer from the promising list. The time taken to extract a promising node is constant, if we successfully find a promising node in the promising list. Otherwise, the extract time is the time spent consolidating a section of the parallel Fibonacci heap. This, we will see in next section, is logarithmic in the number of nodes in that section. Thus, the time taken to do extract operation is $O(lg\ |section|)$.

3.4.4 Consolidate the Parallel Fibonacci Heap

When a process performing an extract operation cannot find a promising node in the promising list after some number of probes, it consolidates the heap, actually a section of the heap, as described in Figure 3.8. The consolidate process randomly chooses a section that is not already being consolidated by another process and locks the section. The process then walks through the nodes in the root list of the section. If a root node is marked as dead, we remove it in lines 10-14. Since there is always at most one consolidation process in a section, there is at most one removal operation running in a section, so we don't have to lock a dead node's neighbors while removing it from the DLL. When a dead node and a dummy node are neighbors, between which there may be insertions going on, we just choose not to remove the dead node.

The consolidation process keeps track of several good nodes that are not already in the promising list by comparing all the non-promising and non-dead nodes in the root

[3]The size is usually chosen to be the number of processes accessing the heap.

list of the section, and puts them in buffer B. We can then "flood" B into the promising list by running *check-promising* on all the nodes in B after finishing the walk through the section in lines 19-20. Buffer B is implemented as a sorted array of fixed size, *buffersize*. A smaller *buffersize* means the nodes in the buffer tend to be more promising. We show the results of experiments that vary *buffersize* in Chapter 5.

The consolidate process also performs normal consolidation like its sequential counterpart. It merges trees of the same degree to reduce the number of trees in the root list. If the root node of a tree is dead or promising, then it won't be merged with other trees. When merging two trees rooted at x and y respectively, we have to lock $key[x]$ and $key[y]$ first. The reason for locking is that there may be delete and decrease operations going on that will interfere with the consolidate process.

The consolidation time for the parallel Fibonacci heap is basically the same as the time taken for the sequential consolidation, because there is only one consolidation process in each section, and the consolidation process only does a little more work than its sequential counterpart: it finds more promising candidates (*buffersize* per process), and there are some locks required when merging trees. These locks are used to prevent operations like delete and decrease key from getting in. The delete and decrease key operations can be operated on all nodes in the Fibonacci heap, not just nodes in the root list. In fact, most of these operations, like deleting some non-promising nodes and decreasing keys of some non-promising nodes, tend to happen to nodes not in the root list. Thus, we expect little contention on the locks the consolidate process acquires while merging trees. Overall, each consolidation process runs in time $O(lg\ |section|)$ time.

3.4.5 Controlling the Quality of Extracted Nodes

There are several parameters that control the promising quality of extracted nodes: *maxpt*, *buffersize*, and *strictness*. *Maxpt* is the size of the promising list, *buffersize* is the size of the buffer used during the consolidation process to gather candidates for the promising list, and *strictness* is used in the heuristic function *good*. We can see that a smaller value of *maxpt* means that the nodes in the promising list are more promising. The extreme case is that *maxpt* equals 1 — there is only one pointer as in the sequential Fibonacci heap. On the other hand, a smaller *maxpt* implies more contention on the promising list. A good value of *maxpt* might be the number of "workers" on the parallel Fibonacci heap.

In the consolidation process, the top *buffersize* number of non-promising nodes in the root list of a section are gathered in a buffer, and are checked if they are promising.

The smaller *buffersize* is, the better the nodes the buffer contains, the fewer candidate nodes there will be for the promising list, and the longer the time it takes to extract a promising node. On the other hand, larger *buffersize* incurs more traffic on the promising list, because there will be more *check-promising* processes trying to put nodes into the promising list.

The effect of the parameter *strictness* is explained in Section 3.4.1. Experiments that vary these parameters are presented in Chapter 5.

3.4.6 Decrease Key Operation

Figure 3.9 shows the pseudocode for decreasing the key of node x to k. Like the sequential decrease key operation discussed in Chapter 2, the idea of the concurrent decrease key operation is to check if k is smaller than x's old key, and then change x's key to k. After the key change, if the heap order property is violated, then cut x from its parent; if an internal node loses more than one child then perform cascading cuts. $Cut(h, x)$ will change x's parent link and its parent y's child link. Both x and y have to be locked during the operation. The order of locking is important here; the wrong locking order can cause deadlock. Consider the case of locking in bottom-up order where y is a promising node in the root list, x is one of y's children, and there is a decrease key operation that is trying to cut x from y. Suppose the decrease key operation has already locked x, and is trying to lock y. In the mean time, another process is doing an extract operation on y, having locked y, and is trying to put y's children, including x, into the root list. In the process of putting y's children into the root list, x's parent field will be updated. If we require locking x before updating its parent field, then this results in a deadlock. If we update x's parent link without locking it, it would be dangerous for the decrease process to read it.

Figure 3.9 shows a way to lock in a top-down order that avoids the problem described above. This locking order also makes the extract operation easier. When we put y's children into the root list in the extract operation as described above, we only need to lock y, because in the top down locking order, y's children won't be updated unless y has been locked. The decrease key operation works in two phases: Phase 1 locks x, locates its parent y if there is one, and unlocks x. Phase 2 locks y then x, verifies y is still x's parent, and does things as in the sequential case. If y is no longer x's parent in phase 2, we go back to phase 1 to locate x's parent again. In phase 1, lines 5-10 lock x, check whether x has a parent. If not, line 13 sets x's key; otherwise, line 17 sets the variable *has-parent?* to be *true* for use in phase 2. Phase 2 checks if variable *has-parent?* is true,

then locks y and x. After y and x have been locked, we verify if y is still x's parent, then change x's key and do the cut in lines 29-36 as in the sequential decrease key operation. Finally, cascading-cuts are done if needed in lines 43-44. If it turns out that y is no longer x's parent in phase 2, then we go back to phase 1 to find x's current parent, and repeat the whole process until x's true parent is found.

If there is no other operation updating x or y between phases 1 and 2, which is likely to be the common case, the parental relationship between y and x does not change between phases 1 and 2. Thus, in most cases, the decrease key operation succeeds without repeating phase 1 and 2. Also, the contention on x and y should be relatively small, since it should be rare that different workers are doing operations on the same x and y. The time taken to do the decrease operation is $O(1)$.

3.4.7 Delete Operation

The delete operation, as shown in Figure 3.10, is similar to the decrease key operation. Instead of cutting x and putting it in the root list as in the decrease key operation, we put x's children into the root list in lines 12 and 28, and mark x to be dead in line 13 if x is in the root list; or remove x in line 29 in case it is an interior node.

3.4.8 Algorithm Validation

We informally show that the algorithms for the parallel Fibonacci heap are deadlock-free as follows. Horizontally, the root list of the parallel Fibonacci heap is a DLL with dummy nodes, and we have shown that the operations on a DLL are deadlock-free in Section 3.2. Vertically, the parallel Fibonacci heap is a forest of trees, and we always lock nodes in a top-down order in the algorithms.

We also validated the correctness of operations experimentally: we occasionally ran a *verify-form* procedure to check the syntactic correctness of the heap (i.e., whether the number of nodes in the heap, the number of nodes in the root list, and the number of promising nodes are correct) and the semantic correctness of the heap (i.e., that the parallel Fibonacci heap is in correct heap-order, and satisfies the heap constraints).

3.5 Summary

The parallel Fibonacci heap presented in this chapter is based on the sequential Fibonacci heap described in Chapter 2. The parallel Fibonacci heap maintains the asymptotic time

bounds of its sequential counterpart, and it also achieves linearly scalable performance. The parallel Fibonacci heap has the following properties:

1. The locks each operation acquires are evenly distributed over the entire data structure and the time each operation takes while holding a lock is small. Assuming the size of the structure is relatively large compared with the number of processes accessing it, then there is very little contention on the structure and we expect linearly scalable throughput. This scalability is reflected in the performance analyses in Chapter 5.

2. Ignoring contention, the sequential operations' time bounds have been preserved: an *insert* operation takes only constant time, an *extract* operation takes $O(lg\ n)$ time, a *decrease key* operation takes constant amortized time, and a *delete* operation takes $O(lg\ n)$ time.

3. The priority queue is non-strict in the sense that an extract operation does not necessarily return the most promising node, but the promising quality can be controlled as described in Section 3.4.5. These non-strict semantics are compatible with most parallel applications, if not all, and they are also one of the reasons that the parallel Fibonacci heap has relatively low contention.

```
    proc extract(id, h)
    % extract a promising node from parallel Fibonacci heap h
    % id is a preassigned worker id

 1  Randomly choose a pointer prom-pt from the promising list
 2  (label#try)
 3  if we have tried "enough" times but still fail to find a promising node then
          % "enough" can be tuned here
 4        consolidate(id, h)
 5  end
 6  Lock prom-pt % if prom-pt is locked, we can try another
 7  if prom-pt = nil then
 8        Unlock prom-pt
 9        prom-pt := another pointer in the promising list
10        goto (label#try)
11  else
12        prom-one := *prom-pt
13        Lock key[prom-one]
14        if mark[prom-one] = promising then
15              if  prom-one has any children then
16                    put its children into the root list
17              end
18              mark[prom-one] := dead
19              Unlock key[prom-one]
20              Unlock prom-pt
21              return prom-one
22        else
23              Unlock prom-one
24              prom-pt := another pointer in the promising list
25              goto (label#try)
26        end
27  end
28  end extract
```

Figure 3.7: Extract operation on parallel Fibonacci heap

```
    proc consolidate(id, h)
    % Consolidate a section (or multiple sections) of the parallel Fibonacci heap h
    %     and find candidates for the promising list

1   Randomly find a section not being consolidated by other processes and lock it
2   for every node x in the section do
3         case mark[x]:
4               unmarked, marked:
5                     Merge trees like in the sequential consolidation. Don't merge dead
6                           or promising nodes.
7                     Maintain a buffer B of top buffersize number of
8                           candidate nodes(non-promising nodes) for the promising list.
                      % buffersize here is tunable parameter
9               dead:
10                    if x's left neighbor is not a dummy node then
11                          Lock key[x]
12                          Remove x from root list
13                          Unlock key[x]
14                    end
15              promising:
16              dummy:
17        end
18  Unlock section
19  for every node n in buffer B do
20        check-promising(h, n);
21  end consolidate
```

Figure 3.8: Consolidate process on parallel Fibonacci heap

```
    proc decrease-key(id, h, x, k)
    % Decrease the key value of x to k in parallel Fibonacci heap h
 1  done? := false % done? means whether the decrease operation has been accomplished or not
 2  has-parent? := false % has-parent? indicates whether node x has parent or not
 3  cascading-cut? := false % cascading-cut? indicates whether cascading-cut is needed
 4  Repeat %%%%%%%%%%%%%%%%%%%% Phase 1
 5        Lock key[x]
 6        if mark[x] = dead then
 7              Unlock key[x]
 8              return
 9        else
10              y := parent[x]
11              if y = nil then  % x doesn't have parent, it is in the root list
12                    if (k < key[x]) then
13                          key[x] := k
14                    end
15                    done? := true
16              else
17                    has-parent? := true
18              end
19        end
20        Unlock key[x]
    %%%%%%%%%%%%%%%%%%%% Phase 2
21        if has-parent? then
22              Lock key[y] % y was x's parent, but may not be now, which happens rarely
23              Lock key[x]
24              if (parent[x] = y) then
25                    if mark[x] = dead then
26                          Unlock key[x]
27                          return
28                    else
29                          if (k < key[x]) then
30                                key[x] := k
31                          end
32                          done? := true
33                          if (key[x] < key[y]) then % heap order has been violated
34                                cut(h, x)
35                                cascading-cut? := true
36                          end
37                    end
38              end
39              Unlock key[x]
40              Unlock key[y]
41        end
42  Until done?
43  If cascading-cut? then
44        cascading-cut(id, h, y)
45  end
46  end decrease-key
```

Figure 3.9: Decrease key operation on parallel Fibonacci heap

```
    proc delete(id, h, x)
    % Delete the node x from parallel Fibonacci heap h

 1  done? := false % done? means whether the delete operation has been accomplished or not
 2  has-parent? := false % has-parent? indicates whether node x has parent or not
 3  cascading-cut? := false % cascading-cut? indicates whether cascading-cut is needed
 4  Repeat %%%%%%%%%%%%%%%%%%% Phase 1
 5        Lock key[x]
 6        if mark[x] = dead then
 7              Unlock key[x]
 8              return
 9        else
10              y := parent[x]
11              if y = nil then % x doesn't have parent
12                    Put x's children into root list if there are any
13                    mark[x] := dead
14                    done? := true
15              else
16                    has-parent? := true
17              end
18        end
19        Unlock key[x]
    %%%%%%%%%%%%%%%%%%% Phase 2
20        if has-parent? then
21              Lock key[y] %y was x's parent, but may not be now, which happens rarely
22              Lock key[x]
23              if (parent[x] = y) then
24                    if mark[x] = dead then
25                          Unlock key[x]
26                          return
27                    else
28                          Put x's children into root list if there are any
29                          Remove x from y's children list
30                          cascading-cut? := true
31                          done? := true
32                    end
33              end
34              Unlock key[x]
35              Unlock key[y]
36        end
37  Until done?
38  If cascading-cut? then
39        cascading-cut(id, h, y)
40  end
41  end delete
```

Figure 3.10: Delete operation on parallel Fibonacci heap

Chapter 4

Concurrent Priority Pool

In this chapter we present another kind of concurrent priority queue, which is implemented as a combination of a concurrent B-tree and a concurrent pool. We call this priority queue a "concurrent priority pool". The concurrent priority pool supports insert and extract operations like the parallel Fibonacci heap. The extract operation is *non-strict*, as described in section 3.1, but there is a straightforward way of controlling the promising quality of extracted keys. The insert and extract operations do not share critical resources in most cases, so that the concurrent priority pool has the highest throughput among all the priority queues studied, as shown by the experimental results in Chapter 5. Section 4.1 briefly describes the concurrent B-tree. Section 4.2 gives an introduction to the concurrent pool. The concurrent priority pool and access algorithms are presented in section 4.3. Finally, Section 4.4 summarizes this chapter.

4.1 Concurrent B-Trees

The Concurrent B-Tree described here is mainly based on [Wan90, WW90, LS86, LY81]. This algorithm allows symmetric insertion and deletion in which each process locks at most one node at a time, except in rare cases.

4.1.1 Data Structure

The concurrent B-tree data structure is similar to the sequential B-tree described in Chapter 2. Figure 4.1 shows an example of a concurrent B-tree: A *B-link structure* is added into the sequential B-tree by connecting nodes on each level into a singly linked list. Each node has a right link that points to its right neighbor. Operations can go

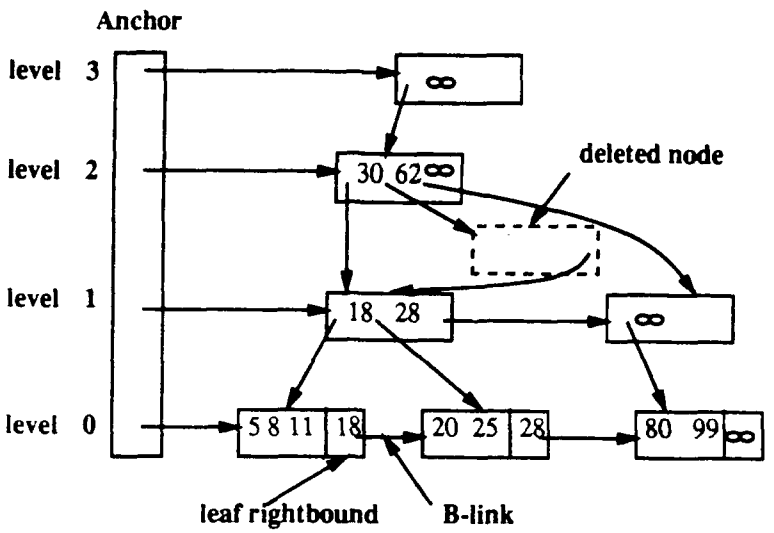


Figure 4.1: An example of a concurrent B-tree

across the linked list horizontally instead of vertically. An *anchor*, an array of pointers to the leftmost node on each level of the B-tree, is added into the sequential B-tree. With the anchor and the B-link structure, a node can be reached not only from its parent, but also from its left neighbors or the anchor.

4.1.2 Insert Operation

Inserting a new key k into a concurrent B-tree invokes two phases: the locate phase and the insert phase. The locate phase, which is similar to its sequential counterpart, traverses the B-tree from the root to the leaf level by following pointers P_i in the internal nodes that have two neighbors K_i and K_{i+1} satisfying $K_i < k \leq K_{i+1}$. In the locate phase, only one internal node is locked at a time. In fact, the nodes only need to be read locked, since the nodes are not changed. After a leaf node n is located, we insert key k into n. If n is full, we split n as shown in Figure 4.2. The split operation is done in two steps: a half-split as shown in Figure 4.2(b), followed by a complete-split as shown in Figure 4.2(c). *Half-split* creates a new node n', inserts n' to the right of n, and moves some data from n to n'. *Complete-split* goes up the tree, inserting a new $< left\ bound, pointer >$ into n's parent m. If m is full, then we split m in the same way as we split n. This split process can propagate from the leaf level up to the tree

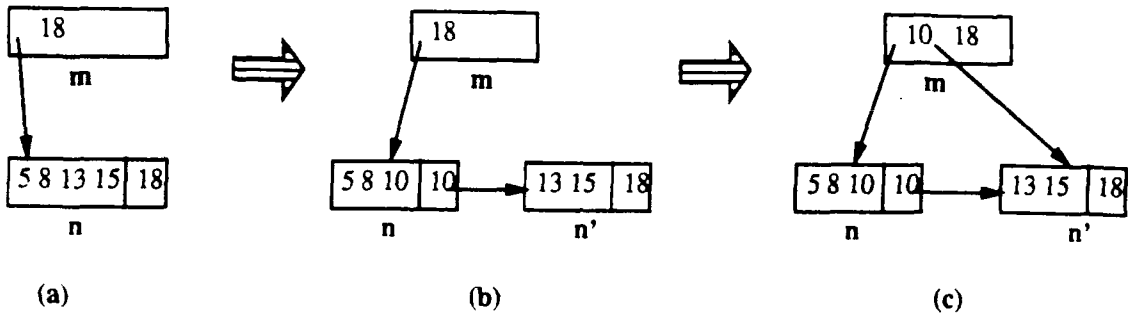


Figure 4.2: Split a concurrent B-tree node (a) Before inserting key 10 into n (b) Half split n (c) Complete split n

root, which might result in creating a new root, and increasing the B-tree height. In all situations, we write lock a node before updating it.

4.1.3 Delete Operation

The delete operation on a concurrent B-tree is symmetric to the insert operation. It consists of two phases: the locate phase and the delete phase. The locate phase is the same as that in the insert operation; it locates the node n containing the key k to be deleted. The delete phase removes k from n; if n is then empty, it merges n's right neighbor n' into n. The merge is also done in two steps: a half-merge as shown in Figure 4.3(b), and a complete-merge as shown in Figure 4.3(c). *Half-merge* first write locks n and n' and removes n' from its level's linked list. It then moves data from n' to n and sets the right link of n' to n before unlocking n and n'. Processes that try to find data in n' still can find them through its right pointer that forwards to n. Complete-merge removes a *<left bound, pointer>* pair from n's parent m. If m is then empty, we merge m with m's right neighbor. This merge process can propagate up to the tree root, which will possibly decrease the height of the tree. There is a special case when complete merging n and n': if n and n' do not have the same parent; this case is explained in [Wan90].

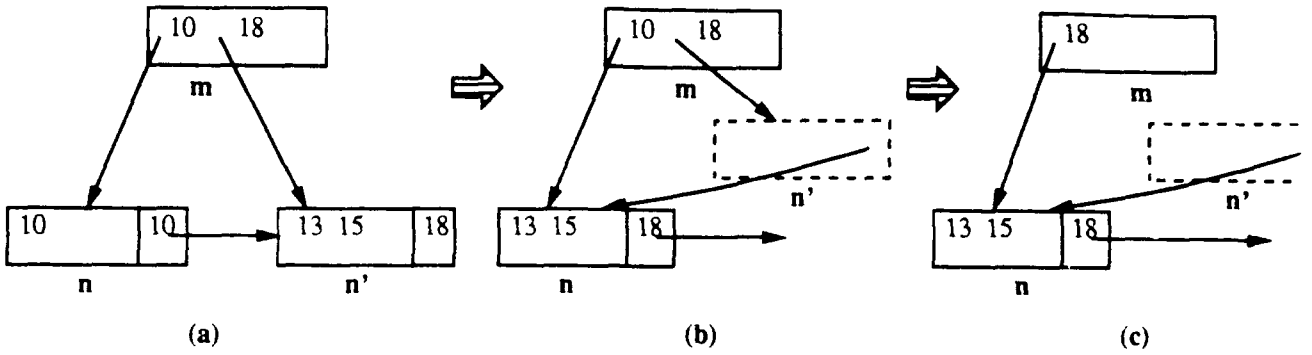


Figure 4.3: Merge two concurrent B-tree nodes (a) Before taking key 10 out of n (b) Half merge n (c) Complete merge n

4.2 Concurrent Pools

Concurrent pools[Man86][KE89] are largely used in the assignment of resources and tasks to processors in a distributed or parallel system that needs to balance the load on each processor. A **pool** is a collection of items that grows and shrinks with the demands of the processes. A process may add an element to the pool or request an element from the pool at any time; the element removed from the pool is chosen arbitrarily. A **concurrent pool** attempts to spread the elements out over the processors so that accesses are less likely to interfere with each other. The basic idea of the concurrent pool is to allow most operations to be done within the local components of the distributed data structure. When a request cannot be satisfied locally, it becomes necessary to access remotely stored components.

4.3 Concurrent Priority Pools

The concurrent priority pool is based on the concurrent B-tree and the concurrent pool. It is similar to the concurrent B-tree, except that the leaves of the B-tree are replaced with concurrent pool-like data structures. An insertion into the priority pool is like the insertion into the B-tree, which takes $O(lg\ n)$ time. The extract minimum operation on the priority pool is similar to the delete operation on the B-tree, but we always delete elements from the promising pools — the leftmost leaf in the B-tree.

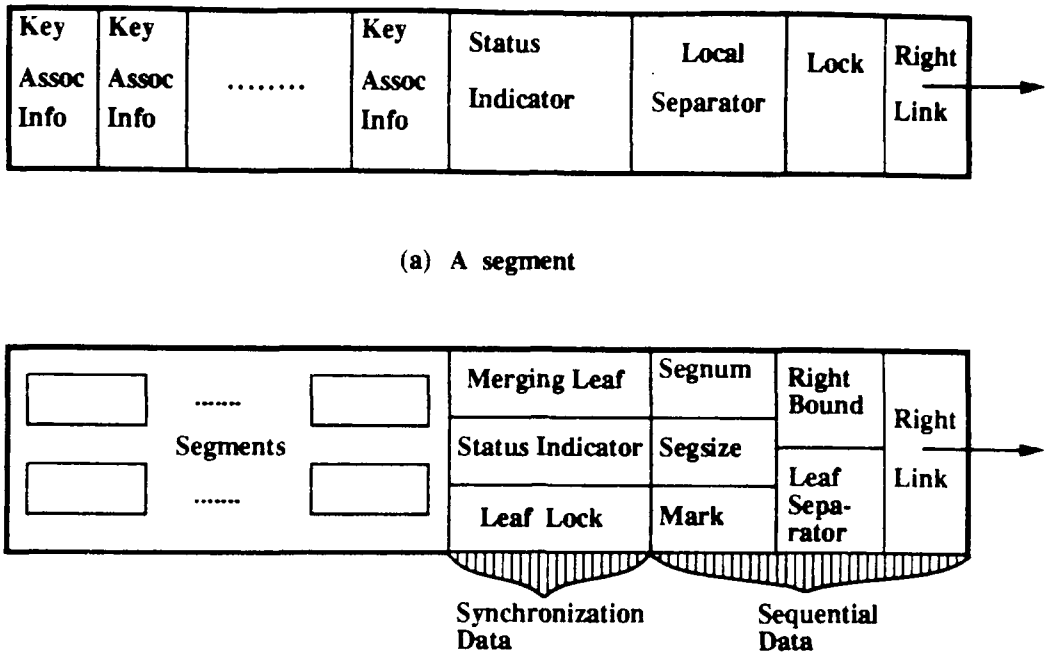


Figure 4.4: Data structure of concurrent priority pools

4.3.1 Data Structure

The concurrent B-tree is the basis for the concurrent priority pool. Each leaf of the priority pool is similar to a concurrent pool. A leaf contains *segnum* number of data segments. Each segment consists of *segsize* number of keys and associated data. The segment is the smallest unit that is locked during the insert and extract operations. Even when splits and merges happen, leaves are only locked briefly, as we will see in the next few sections. There can be different operations running concurrently on different segments in the same leaf.

As shown in Figure 4.4(a), a segment has an array of keys and associated data, a status indicator, a local separator, a lock and a local right link. The segment local separator is usually equal to the right bound of the leaf the segment is in, except in the middle of splitting or merging. The segment right link points to the leaf that contains keys equal to or larger than the segment separator; that is usually the right neighbor of the leaf containing the segment. The status indicator indicates whether the segment is in

normal mode or has been deleted. The segment can only be changed when the segment lock is acquired.

The keys in a segment are stored in an array that is ordered from largest to smallest. This simplifies extracting the smallest key: we only need to return the rightmost element of the array and decrease the array size by one. Keeping segments sorted also makes it easier to find a medium key in a segment, which is used in splitting the segment. On the other hand, it is more expensive to insert a key in a sorted segment and to merge two sorted segments.

A leaf has three major parts, as shown in Figure 4.4(b) : synchronization data, sequential data, and *segnum* number of segments. Sequential data consists of segnum, segsize, right bound, mark, right link, and separator. The right bound of the leaf is usually the largest key in the leaf. This is not true in two cases: when the leaf is being split, in which case there may be some larger keys that have not been moved to the right neighbor yet; or the when the leaf is being merged, in which case the right bound may be larger than all the keys in the leaf. The leaf mark is one of dead, orphan, dead-orphan, or nil: dead means the leaf has been deleted; orphan means that there is another leaf with the same right bound as this leaf, and the orphan leaves do not have parents as described in Section 4.3.2: dead-orphan means the leaf is both dead and an orphan.

Synchronization data consists of a leaf lock, a status indicator, and a merging-leaf field that points to the leaf, if any, that has been merged with this one. The status indicator is one of normal, split, merging, split-merging, and deleted: normal means the leaf is in normal mode, split means the leaf is being split, merging indicates that the leaf is now merging with another leaf, deleted indicates the leaf has been deleted, and split-merging means there is a split and a merge concurrently going on in the leaf. Figure 4.5 depicts the possible status transitions of a leaf. The leaf sequential data and synchronization data can be changed only when the leaf-lock is acquired.

4.3.2 Duplicate Keys

The concurrent B-tree, the basis for the concurrent priority pool, is changed to allow duplicate keys. On the leaf level of the B-tree, we allow multiple leaves with the same right bound; only one of the leaves can be directly reachable from internal nodes, and the rest of them are marked as orphans. Thus, there are no duplicate separators in internal nodes. The original concurrent B-tree algorithms are changed slightly:

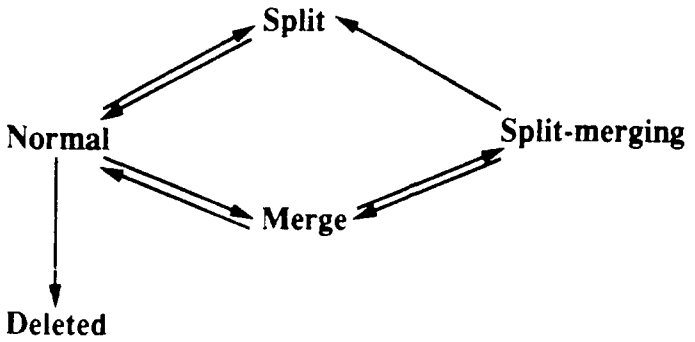


Figure 4.5: Concurrent priority pool leaf status transition graph

1. While doing "complete-split" as shown in Figure 4.2(b), which tries to add a $<$ *separator*, *pointer* $>$ pair into internal node m, if we find there already exists a *separator* in m, then instead of adding the pair in, we mark the leaf pointed to by *pointer* as an orphan.

2. While doing "complete-merge" as shown in Figure 4.3(b), which tries to delete a $<$ *separator*, *pointer* $>$ pair from an internal node, if we find the leaf pointed to by *pointer* is marked as an orphan, then we know the pair is not in an internal node. Thus, we can quit from complete-merge.

This method treats all leaves, whether orphan or not, quite uniformly while doing insert and extract operations. It also keeps the structure of internal nodes the same, so that the original concurrent B-tree algorithms on internal nodes are still applicable.

4.3.3 Insert Operation

Inserting a key into a priority pool invokes two steps: first, locating a leaf as in the concurrent B-tree algorithms; second, as described in this section, inserting the new key into the leaf, and performing split operations if necessary. Here we only present the algorithms on the leaves of the priority pool, since the algorithms on the internal nodes are the same as those for a concurrent B-tree. Figure 4.6 shows the pseudocode for inserting a *key* in leaf l of *tree*. We first randomly locate a segment s in leaf l, and lock it in lines 1-2. Line 3 checks whether segment s is the right one to insert *key* in — if *key* is larger than $separator[s]$, then we insert *key* into the leaf that is pointed to by $right[s]$.

```
    proc insert(l, key, tree)
    % insert a new key into leaf l of tree
 1  (label#0) Randomly locate a segment s in leaf l
 2  (label#1) Lock s
 3  if (key > separator[s])
 4        l :=  right[s]
 5        Uniock s
 6        goto (label#0)
 7  else
 8        case indicator[s]
 9        normal:
10        if s is not full then
11              insert key into segment s
12              Unlock s
13        else
14              if we have not tried twice(or some other number) then
15                    Unlock s
16                    s := another segment in leaf l
17                    goto (label#1)
18              else
19                    Lock l
20                    case indicator[l]:
21                          normal:
22                            Unlock s
23                            originate-split(l, key)
24                          split, split-merging:
25                            Unlock l
26                            split(s, l, l', separator[l], ...)
                              % l' is l's right neighbor; assume l' and separator[l]
                              %    are read before l is unlocked
27                            Unlock s
28                            s := another segment, goto (label#1)
29                          merging:
30                            Unlock s
31                            originate-split(l, key)
32                          deleted:
33                            Unlock s
34                            Unlock l
35                            insert(l', key, tree) % l' is pointed by the right link of l
36                    end
37              end
38        end
39        deleted:
40         insert(right[s], key, tree)
41        end
42  end
43  end insert
```

Figure 4.6: Insert operation on concurrent priority pool

We check *indicator*[*s*] in line 8: if *s* is deleted, then we insert *key* into the leaf pointed to by *right*[*s*] in lines 39-40. If *s* is in normal mode, we do "normal insertion" in lines 10-38. Line 10 checks whether *s* is full; if not, we directly insert *key* into *s*. Otherwise, we try to find other segments in leaf *l* to do the insertion in lines 15-17. If we still can not find a non-full segment in leaf *l* to insert *key* after some number of tries, we try to split leaf *l* in lines 19-30. Leaf *l* is locked in line 19 to check the indicator of *l*. In case *indicator*[*l*] is normal, the originate-split procedure is called to originate splitting leaf *l*. In case *indicator*[*l*] is split or split-merging, which means leaf *l* is already being split, we unlock *l* and help split segment *s* in lines 25-26. We try to insert again in line 28. In case leaf *l* is merging with another leaf, *l* is split by calling originate-split in line 31. In case *l* has been deleted, though *s* has not been deleted yet, we insert *key* in the leaf pointed to by *right*[*l*] in line 35.

Figure 4.7 shows the pseudocode of splitting a leaf of a concurrent priority pool. Procedure *originate-split* splits leaf *l* and inserts *k* into the priority pool. Procedure *split* splits a segment.

At the entry of *originate-split*, we assume *l* has been locked. Line 1 checks *indicator*[*l*] and changes it as depicted in Figure 4.5: if it is normal, then it is changed to split; if it is merging, it is changed to split-merging. Line 6 creates a new empty leaf *l'* with right bound, right link, segnum, segsize set to the same as those in leaf *l*. Line 7 chooses a separator for leaf *l*, and puts it in *separator*[*l*]. Line 8 unlocks *l*; note that the leaf lock is held for a relatively short time (lines 1-8). Lines 9-12 split all segments in *l*. While the originate-split process is splitting segments in *l*, there can be other processes helping split segments in *l* — see line 27 of the insert procedure in Figure 4.6. After all the segments are split, *l* is locked to change *indicator*[*l*] back as shown in Figure 4.5. Once again, the leaf is locked for only a brief time. In line 20, key *k* is inserted into *l* or *l'* depending on the chosen separator: if *k* is larger than *sep*, we insert *k* into *l'* and vice versa. Line 21 does complete-split by trying to add a new < *separator*[*l*], *l'* > pair in *l*'s parent.

Procedure *split* in Figure 4.7 splits segment *s* if it hasn't been split yet — *separator*[*s*] > *separator*[*l*], or it has been split — *separator*[*s*] = *separator*[*l*] and *s* is still full. In either case, we move some data from *s* to its right neighbor *l'*.

The time taken to insert a key into a concurrent priority pool is composed of the time taken to go from the tree root down to the leaf level, the time taken to insert the key into a leaf, and the time to do complete-split. We have seen that the leaf does not need to be locked if it is not split, and is only locked very briefly to change the indicator and link fields if a split happens. Thus, there is very little contention on inserting a

```
    proc originate-split(l, k)
    % Originate splitting leaf l, k is a key to be inserted.

 1  if (indicator[l] = normal)
 2        indicator[l] := split
 3  else
          %% indicator is merging
 4        indicator[l] := split-merging
 5  end
 6  Create a new empty leaf l' and link it to the right of l
 7  separator[l] := choose-separator(s, l)
 8  unlock l
 9  forall segment s in l do
10        lock s
11        split(s, l, l', separater[l], k)
12        unlock s
13  lock l
14  if (indicator[l] = split)
15        indicator[l] := normal
16  else
          %% indicator is split-merging here
17        indicator[l] := merging
18  end
19  unlock l
20  insert k depending on sep
21  Do complete-split as in the concurrent B-tree
22  end originate-split.


    proc split(s, l, l', sep, k)
    %% Assume s has been locked
    %% Split segment s in leaf l depending on separator sep.

 1  right[s] := &l'
 2  if ((separator[s] > sep) or
 3       ((separator[s] = sep) and
 5        full(s))) then
 6        separator[s] := sep
 7        move some data from l to l' using
 8             sep as a filter --- like the insert operation as futures
 9  end
10  end split
```

Figure 4.7: Split a leaf of concurrent priority pool

key into a leaf if there are enough segments in a leaf. The overall time taken to do the insert operation should be comparable to the time taken to do insertion in the concurrent B-tree, $O(lg\ N)$, where N is the number of keys in the priority pool.

4.3.4 Extract Operation

The leftmost leaf of a concurrent priority pool contains keys smaller than keys in other leaves. The extract operation on a concurrent priority pool always extracts a key from the leftmost leaf. Since the anchor contains direct pointers to the leftmost node on each level, we can locate the leftmost leaf without going down from the tree root. This decreases the traffic through the root.

The number of keys a leaf contains can be controlled, hence, the promising quality of extracted keys can be controlled — we can vary *segnum* and *segsize* to control the number of promising elements in the leftmost leaf. The extract operation always finds a key that is one of the $segnum * segsize$ smallest keys in the concurrent priority pool. In practise, the extracted key is usually better than the given bound, because the smallest key in a segment is extracted first.

Figure 4.8 shows the pseudocode for the extract operation. First, we randomly pick up a segment s in leaf l and lock it. We check whether s is in normal mode in line 3. If not, we go to the leaf pointed to by s's right link to do the extract operation in lines 38-40. Otherwise, we do "normal deletion" as following. If s is not empty then we extract the smallest key from s in line 5. If s is empty, then we can try other segments in lines 9-11. If we fail to find a non-empty segment in l after several tries, we merge l with its right neighbor in lines 13-34. We lock l to check $indicator[l]$ in line 14. In case it is normal, the originate-merge procedure is called to start merging. In case $indicator[l]$ is merging, we help merge some segments in leaf l by calling the help-merging procedure at line 27. In case leaf l has been deleted, we go to l's right neighbor to do the extract in lines 29-33. If leaf l is being split, we simply go back to try other segments, because we have not found a non-empty segment yet, so we can not help the split; if we find a non-empty segment, then the extract operation will be done.

Figure 4.9 shows the pseudocode of the procedure originate-merging, which merges two leaves in the concurrent priority pool. We assume leaf l is locked upon entrance. Line 1 finds l's right neighbor l' and locks it. Line 2 tests the indicator of l'. If it is normal, we merge l and l' in lines 4-21, do complete-merge as in the concurrent B-tree, and redo the extract operation in lines 22-23. The locks of leaves l and l' are acquired only to change their indicator and right fields in lines 4-8. Lines 11-14 merge all segments in l'

```
    proc extract(l)
    %% extract a key from leaf l in the concurrent priority pool

 1  (label#0) randomly pick up a segment s in l
 2  (label#1) lock s
 3  if indicator[s] = normal then
 4        if s is not empty then
 5              extract the smallest key from s
 6              unlock s
 7        else
 8              if we have not tried to delete enough times then
 9                    unlock s
10                    s := another segment in l
11                    goto (label#1)
12              else
                      %% do merge here
13                    lock l
14                    case indicator[l]:
15                     normal:
                            %% normal merge
16                          unlock s
17                          if l is not the rightmost leaf then
18                              originate-merge(l)
19                          end
20                     split:
21                          unlock l
22                          unlock s
23                          s := another segment; goto (label#1)
24                     merging:
25                          unlock s
26                          unlock l
27                          help-merging(l, merging-leaf[l], right[l])
                            % The merging-leaf and right fields of l should be
                            %    read before unlocking l
28                          goto (label#0)
29                     deleted:
30                          unlock s
31                          unlock l
32                          l := right[l] % right[l] should be read before unlocking l
33                          goto (label#0)
34                    end
35              end
36        end
37  else
          %% indicator[s]= deleted
38        l := right[s]
39        unlock s
40        goto (label#0)
41  end
42  end extract
```

Figure 4.8: Extract operation on concurrent priority pool

```
    proc originate-merging(l)
    %% try to merge l's right neighbor l' with l
    %% assume l is locked at entry

 1  (label#1) lock l' % l' is the right neighbor of l
 2  case indicator[l']
 3   normal:
 4        indicator[l] := merging
 5        indicator[l'] := deleted
 6        merging-leaf[l] := &l'
 7        right[l] := right[l']
 8        right[l'] := &l
 9        unlock l'
10        unlock l
11        forall segments s' in l' do
12             lock s'
13             match-merge(s', l, l', l")
               %% l" is the right neighbor of l and should be read before unlocking l
14             unlock s'
15        lock l
16        if (indicator[l] = merging)
17             indicator[l] := normal
18        else
               %% indicator is split-merging
19             indicator[l] := split
20        end
21        unlock l
22        Do complete merge like in the concurrent B-tree
23        extract(l)
24   split, split-merging:
25        unlock l'
26        unlock l
27        extract(l)
          %% This is a rarely happening loop. We cannot help split here,
          %% since l', the destination leaf, is unlocked and may be merged again.
28   merging:
29        unlock l'
30        unlock l
31        help-merging(l', merging-leaf[l'], right[l'])
          %% assume merge-leaf[l'] and right[l'] are read before unlocking l'
32        extract(l)
33   deleted: error
34  end
35  end originate-merge
```

Figure 4.9: Merge two leaves of concurrent priority pool

with segments in leaf l. The match-merge procedure, which is described later, is called to ensure that every segment in l will be updated. While the originate-merging process is merging the segments in l' into l, other processes can help to do the merge as shown in line 31. Lines 15-21 lock l to change its indicator back to normal or split. Once again, the leaf lock is held briefly. If *indicator*[l'] is split or split-merging, we just go back to extract again in line 27. If *indicator*[l'] is merging, we help merge some segments in lines 28-32. The indicator of l' cannot be deleted, because deleted leaves are moved out of the linked list — they cannot be l's right neighbor.

Procedure match-merge, as shown in Figure 4.10, merges segment s' of leaf l' with the corresponding segment s of leaf l. Because there are the same number of segments in every leaf, it is not hard to create a one-to-one correspondence between segments in two leaves. Leaf l'' was the right neighbor of l, but may be not now. Consider the example shown in Figure 4.11, in which leaf l is changed to the split-merging state from the merging state, and a new leaf *lnew* is created between l and l''. Segment $s1$ in l has been split, so $s1$'s right link points to *lnew*. Segment $s0$ in l has not been either split or merged yet, so its right link points to l'. Segment $s2$ in l has been merged but has not been split yet — its right link points to l''. The right links of segments in leaf l are set to point to l'' if the segments have not been split or merged; otherwise, the right links are left unchanged. The split process, concurrently going on with the match-merge, will change the right links of all segments in l to point to *lnew* as shown in line 1 of the split procedure in Figure 4.7. Thus, the match-merge procedure will change $s0$'s right link to point to l'' because it has not been either split or merged; $s1$'s right link will not be changed since it has been split; segment $s2$'s right link will be changed to point to *lnew* by the concurrent split process.

Figure 4.10 also shows the pseudocode for the help-merging procedure. This help-merging procedure randomly picks up a segment s' from leaf l', locks it, and calls match-merge to merge the segment if s' is in normal mode and non-empty, then unlocks it. Actually, we could help to merge more segments in the help-merging procedure.

Assume there are enough number of segments in a leaf, so that there is not much contention on grabbing a segment from the leaf. If the segment is not empty, then the extract operation takes only constant time — it can just take the smallest key in the segment. If the segment is empty and we cannot find a non-empty one after several tries, we need to merge the leftmost leaf with its neighbor, which takes $O(segnum * segsize)$ time. If we count in the time taken to do complete-merge, $O(lg\ N)$, the extract operation takes time $O(lg\ N)$.

4.4 Summary

This chapter presents another new concurrent priority queue called the concurrent priority pool, which is based on concurrent B-trees and concurrent pools. The concurrent priority pool supports insert and extract operations like the parallel Fibonacci heap. The structure of the concurrent priority pool is very similar to the concurrent B-tree, except the leaves are replaced with concurrent pool-like data structures. Each leaf of a concurrent priority pool consists of several segments, each of which contains a fixed number of keys. There can be different operations going on different segments in the same leaf. The lock granularity of normal insert and extract operations is pushed down to the level of segments instead of leaves. Even when splits and merges happen, the leaves are locked only briefly. The insert and extract operations do not share critical resources in most cases, which is one of the reasons why the concurrent priority pool has the largest throughput among all the priority queues studied, as shown by the experimental results in Chapter 5. Also, the concurrent priority pool provides a straightforward way of controlling the promising quality of extracted keys.

```
    proc match-merge(s', l, l', l")
    %% Match-merge  moves data from segment s' of leaf l' into the
    %%  corresponding segment in leaf l.

 1  Lock s    %% s is the segment in l corresponding to s' in l'
 2  if (right[s] = l') then
          %% s hasn't been either merged or splitted
 3        right[s] := &l"
 4        separator[s] := right-bound[l']
 5  end
 6  Transfer data from segment s' in l' to s.
    %% In this way, we are sure that every  segment in l is touched.
 7  If it does not all fit, insert the rest normally by calling insert procedure as futures.
 8  Unlock s
 9  indicator[s'] := deleted
10  end match-merge


    proc help-merging(l, l', l")

 1  Choose a segment s' in l'
 2  Lock s'
 3  if ((indicator[s'] = normal)
 4       and (not empty(s'))) then
 5         match-merge(s', l, l', l")
 6  end
 7  Unlock s'
 8  end help-merging
```

Figure 4.10: Match merge corresponding segments in two leaves on concurrent priority pool

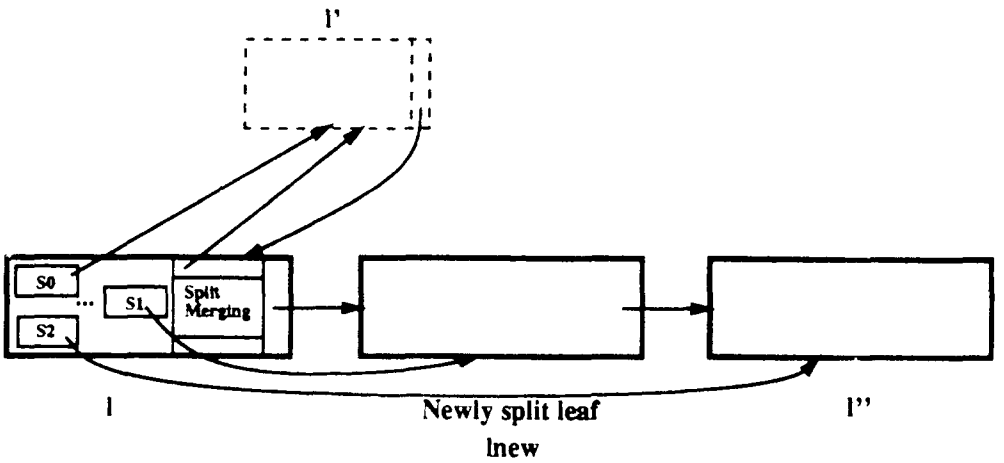


Figure 4.11: Match-merging two leaves l and l'

Chapter 5

Experimental Evaluation

In this chapter we present the experimental evaluation of the parallel Fibonacci heap and the concurrent priority pool and compare them with the concurrent binary heap. Section 5.1 describes the experimental environment and model. Section 5.2 shows the effects of different parameters on the parallel Fibonacci heap. Section 5.3 shows the the effects of different parameters on the concurrent priority pool. Section 5.4 compares different concurrent priority queues in terms of throughput. Section 5.5 presents two applications of concurrent priority queues: the single source shortest path problem(SSSP) and the vertex cover problem(VCP). Finally, Section 5.6 summarizes this chapter.

5.1 Experimental Environment

Experiments have been performed on Encore Multimaxes. The language used is Mul-T [KHM89], a Lisp-like programming language with futures and lock mechanisms. Two Encore machines have been used in the experiments: one with ten processors at LCS/MIT, where most of the debugging tests were done; one with twenty processors at the Argonne National Lab [1].

In most of the experiments, the master-worker model is used: a master spawns a fixed number of workers, each of which performs access-think cycles. An access can be an insert, extract, decrease key or delete on a concurrent priority queue. Think time is modeled by a simple delay in a loop; the number of iterations denotes the think time. Think = 0 means the workers do not think at all, and think = 1000 means think consists of 1000

[1] Only 18 processors can be used for running Mul-T. Due to some unknown errors, running Mul-T with large number of processors has caused the Encore at Argonne Lab to crash. Thus, we did not get all possible data up to 18 processors.

loop iterations. Since the decrease key and delete operations are not supported well on binary-heap-based concurrent priority queues and concurrent priority pools, we compare them by measuring only the insert and extract operations. In most trials described in this chapter, the following worker model is used unless otherwise stated: the number of workers is equal to the number of processors available; each worker performs access-think cycles on a heap initially containing 1000 keys[2], and access to the priority queue is composed of 55% inserts and 45% extracts. The keys inserted are randomly chosen from the range 0 to 10000. All workers are started at approximately the same time, and the first worker that finishes 1000 access-think cycles will stop other workers. The throughput is the total number of cycles performed by all the workers divided by the elapsed time. I used the timer facilities of Mul-T version 25 to collect data.

5.2 Parallel Fibonacci Heap

The parallel Fibonacci heap has three parameters: *maxpt*, *buffersize*, and *strictness* as described in section 3.4.5. We have tested different combinations of *buffersize* and *strictness*, with *maxpt* set to be the same as the number of processors. Figure 5.1 shows the throughput (cycles/second) vs. the number of processors, while the think time is 0. We can see that the throughput in the trials is linearly increasing with the number of processors, from around 70 with 2 processors to around 680 with 18 processors. We can roughly see from Figure 5.1 that all the curves are very close to each other, which indicates that the parameters *buffersize* and *strictness* do not affect the throughput too much. Trials with larger *buffersize* and *strictness* have a little larger throughput. However, *strictness* has more impact than *buffersize*. Note in Figure 5.1 that the throughput is quite good when *buffersize* = 1, and *strictness* = 1. *Buffersize* = 1 means only the least key in a parallel Fibonacci section is selected as a candidate for the promising list in the process of consolidation, and *strictness* = 1 means the promising list will only get better candidates from direct promise-checking since the *good* heuristic function filters out almost all keys worse than keys in the promising list.

Figure 5.2 shows the throughput vs. the number of processors when think = 1000, and different *buffersize* and *strictness*. The curves are quite similar to the case of think = 0, except the throughput is less due to the think time. Figure 5.2 also shows the trials with *strictness* equal to 1. It shows the throughput of the parallel Fibonacci heap does

[2]This avoids extracting from an empty priority queue.

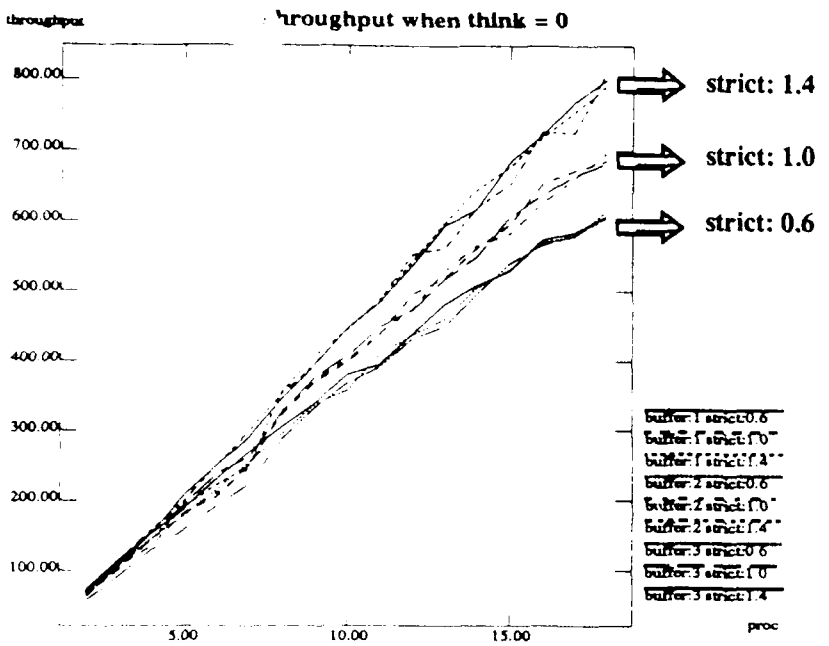


Figure 5.1: Parallel Fibonacci heap: Throughput (cycles/second) vs. number of processors while think =0, different values of parameters *buffer size* and *strictness*

not change too much with different *buffer* when *strictness* = 1.

5.3 Concurrent Priority Pool

The concurrent priority pool has two parameters: *segnum*, which is the number of segments in a leaf, and *segsize*, which is the number of keys contained in each segment and the number of < *pointer*, *bound* > pairs in an interior node. We have done some experiments on different values of *segnum* and *segsize*. In the experiments, ordinary blocking locks are used instead of read-write locks (see Section 4.1). Using read write locks should reduce the contention on interior nodes of the B-tree. Figure 5.3 shows the throughput vs. the number of processors when think = 0, *segsize* = 3, and different *segnum*. Figure 5.4 shows the throughput vs. the number of processors when think = 0, *segsize* = 5, and various *segnum*. Figure 5.5 shows the curves when think = 0, *segsize* = 7, and different *segnum*. These three graphs have one thing in common: the throughput are linearly increasing with the number of processors, and all the curves are close to each

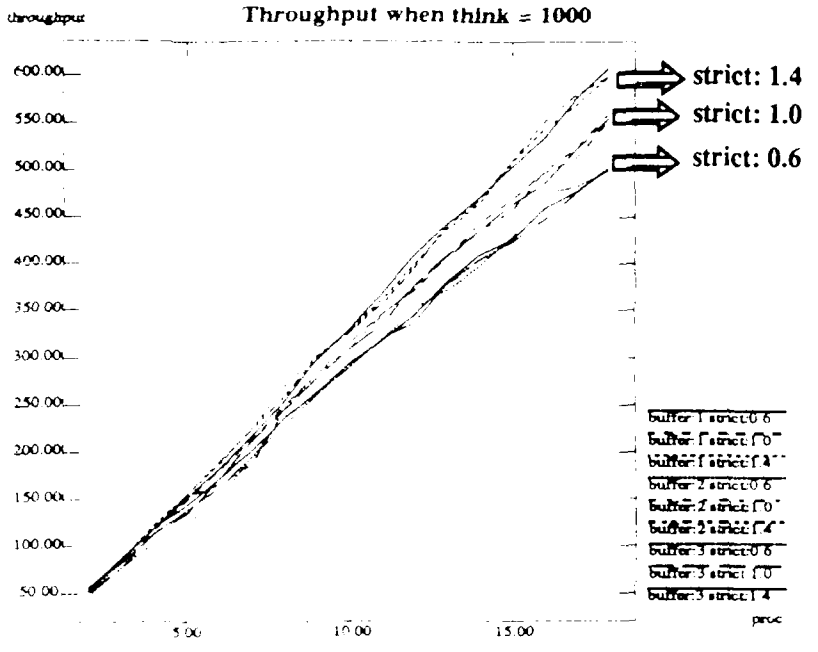


Figure 5.2: Parallel Fibonacci heap: Throughput (cycles/second) vs. number of processors while think = 1000, different values of parameters *buffersize* and *strictness*

other, which means the parameters do not affect the throughput too much.

5.4 Comparing Different Concurrent Priority Queues

We have seen how the parallel Fibonacci heap and the concurrent pool perform on different parameters. Here, we consider how they compare with each other, and how they compare with other kinds of concurrent priority queues, such as the concurrent binary heap. The concurrent binary heap compared here was developed by Rao and Kumar[RK88b]. They proposed a method of performing insert and delete operations concurrently in a top-down order on a balanced binary heap. The insert operation locks one node at a time, and the delete operation locks three nodes, a parent and two children, at a time. Their scheme has strict semantics for the extract operation, which means the extract operation always retrieves the most promising key. The problems with strict semantics have been discussed in Section 3.1.

Figure 5.6 shows a comparison of the throughput of different priority queues: the

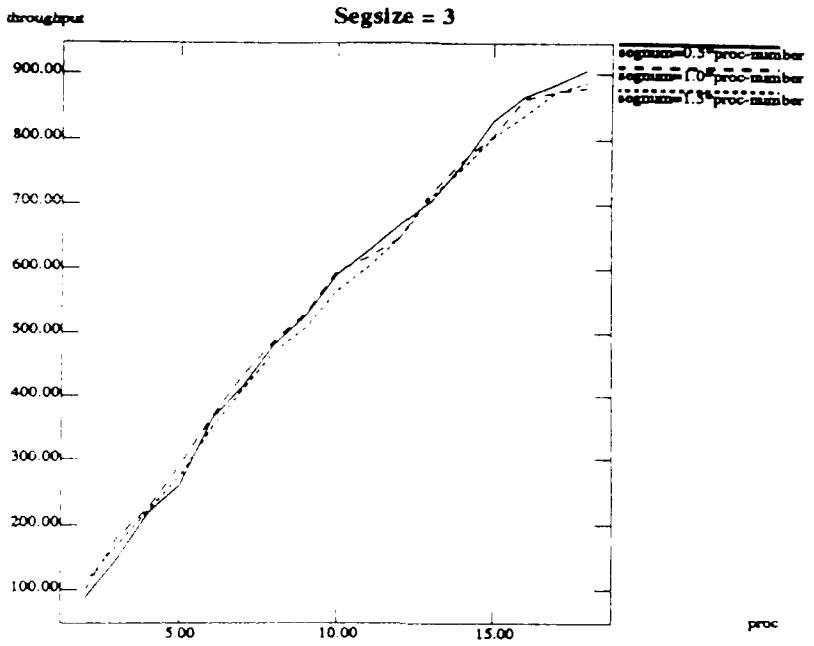


Figure 5.3: Concurrent priority pool: think = 0, *segsize* = 3, different *segnum*

sequential binary heap, the concurrent binary heap, the concurrent priority pool, and the parallel Fibonacci heap. Each operation on the sequential binary heap has an exclusive lock on the whole heap during the entire period of the operation. The parallel Fibonacci heap tested here is an average one, with *buffersize* and *strictness* both equal to one. The concurrent priority pool tested has *segnum* equal to the number of processors, and *segsize* equal to 5. The graph shows that the throughput of the parallel Fibonacci heap and the concurrent priority pool are both linearly scalable, and that the concurrent priority pool has the largest throughput among these four priority queues. The concurrent binary heap's throughput saturates when the number of workers is more than about eight. Since the sequential binary heap holds a lock on the entire heap during an operation, its throughput decreases as the number of processor increases. Because all the insert and extract operations of a concurrent binary heap both have to go through and lock the tree root, the tree root becomes a bottleneck when the number of processes accessing the concurrent binary heap increases. This bottleneck problem is reflected in Figure 5.6, which shows that the throughput of a concurrent binary heap saturates quickly. Overall, the concurrent binary heap is not as scalable and efficient as either the parallel Fibonacci

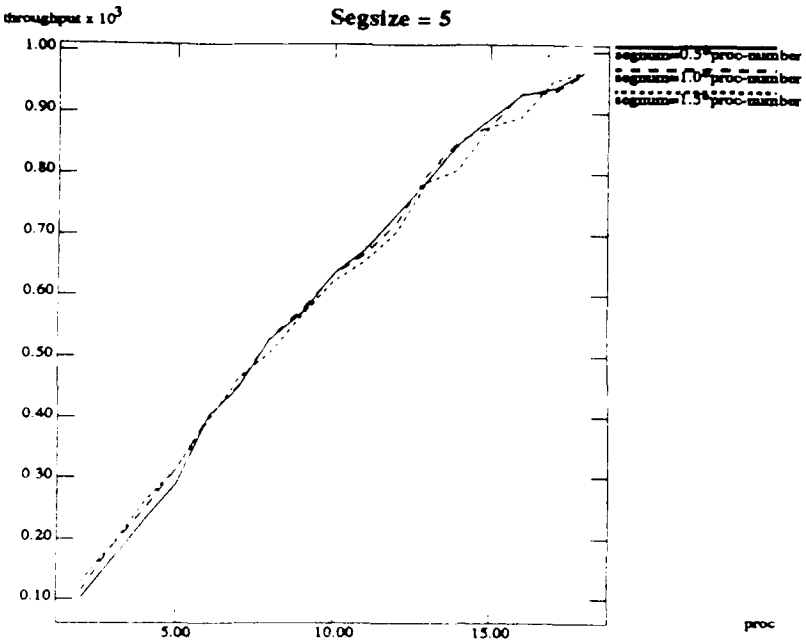


Figure 5.4: Concurrent priority pool: think = 0, *segsize* = 5, different *segnum*

heap or the concurrent priority pool.

Figure 5.7 shows the comparison when think = 1000. The contention on the priority queues is less than that of think = 0; this helps slow down the saturation of the less scalable priority queues.

5.5 Applications

Two kinds of applications of concurrent priority queues are presented in this section. One is the single source shortest path problem which is in the computational class *P*. The other one is the vertex cover problem which is in the computational class *NP-complete*.

5.5.1 Single Source Shortest Path Problem

The single source shortest path problem is as follows: given a source vertex s in a weighted graph $G =< V, E >$, find a path of minimum weight from s to every $v \in V$. We choose Dijkstra's algorithm as our basis[CLR90]. As shown in Figure 5.8, we keep a priority

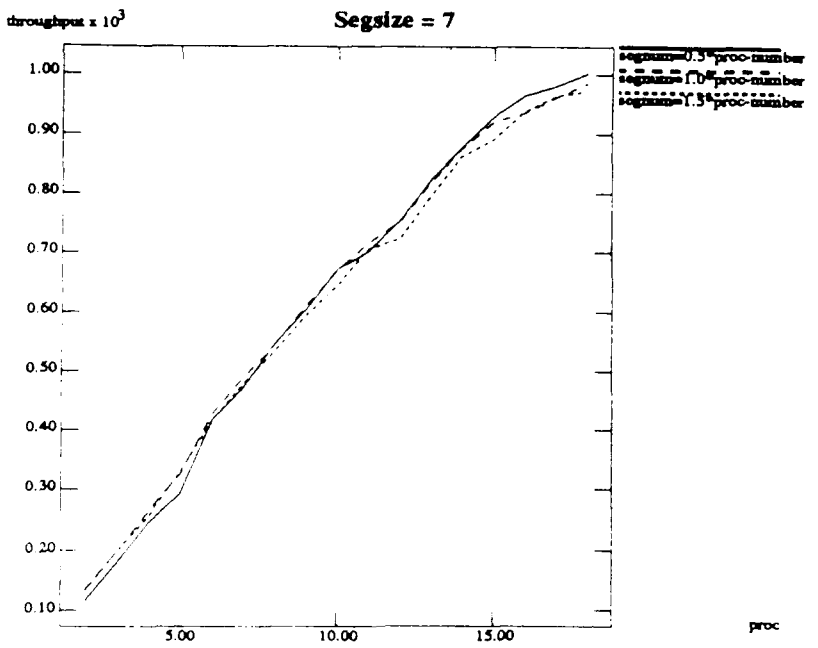


Figure 5.5: Concurrent priority pool: think = 0, *segsize* = 7, different *segnum*

queue Q of vertices in V. The priority of a vertex in Q is its distance from the source vertex s. The algorithm always chooses the vertex u that is the closest to s to add into S. For each vertex u's neighbor v, we check if a shorter path has been found: if so, we update $d[v]$ in line 11. Note that vertices are never added to Q, and each vertex is extracted from Q and added to S exactly once.

The parallel single source shortest algorithm is presented in Figure 5.9. Independent workers work on a concurrent priority queue. These workers perform the same job as their sequential counterparts: extract a close vertex n from the queue and check all n's neighbors to see if closer paths have been found. Unlike the sequential Dijkstra's algorithm, when we extract a vertex from the concurrent priority queue, the vertex does not necessarily have to be the closest one from the source. In this way, a node may be inserted into the queue several times if a better path is found later on. However, the experiments show that on average each node is inserted no more than 1.3 times. Similarly, more decrease key operations are performed.

This algorithm requires the use of the decrease key operation; since the decrease key operation cannot be effectively implemented on the concurrent binary heap, we only

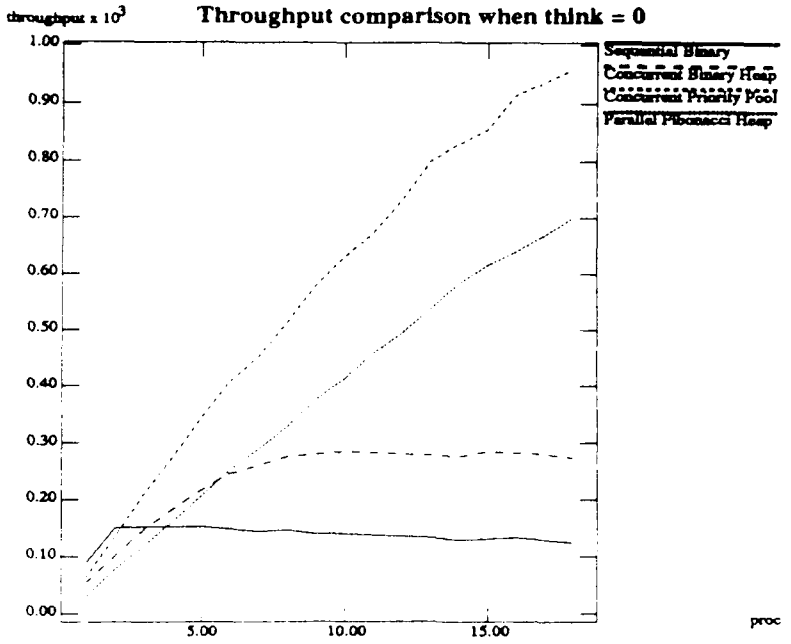


Figure 5.6: Comparing different priority queues: think = 0

compare the sequential binary heap, the parallel Fibonacci heap, and the concurrent priority pool. For the concurrent priority pool, the decrease key operation is implemented as a combination of delete and insert operations: first we delete the old key from the pool, then we insert the new key into the pool. In this way, a decrease key operation for the concurrent priority pool consists of two accesses whereas it is a simple operation with amortized constant cost for the parallel Fibonacci heap. In the implementations, we have kept track of where a key is in a priority queue to avoid searching when we do decrease key operations.

Figure 5.10 shows the speedup graph of the single source shortest path problem. The graph has 1000 vertices and the degree of each vertex is randomly chosen from 0 to either 10 or 50. The sequential binary heap is used to compute speedup. The sequential program is very efficient (it is in computational class P) and always finds the shortest path to any vertex in shorter steps as compared to the case of concurrent priority queues where we do some extra work such as inserting a vertex in the queue several times and decreasing the distance of a vertex more often. As expected, the speedup ranges from around 0.3 with one processor to about 4.5 with fifteen processors. The parallel Fibonacci

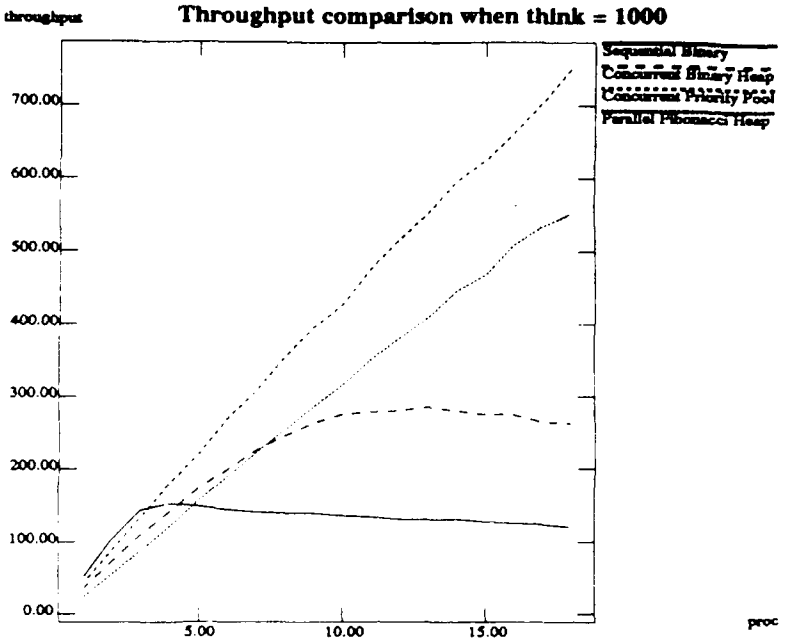


Figure 5.7: Comparing different priority queues: think = 1000

```
    proc Dijkstra(G, s)
    % Find the shortest path from source s

 1  for each vertex v in V[G]
 2        do  d[v] := ∞  % initialize distance to be $\infty$
 3  d[s] := 0
 4  S := ∅
 5  Q := V[G]
 6  while Q ≠ ∅  do
 7        u := extract-min(Q)
 8        S := S⋃{u}
 9        for each vertex v in Adj[u]  do  % relax edge (u, v)
10              if  d[v] >  d[u] + w[u, v]  then
11                    d[v] := d[u] + w[u, v] % this is a decrease key operation
12              end
13  end Dijkstra
```

Figure 5.8: Dijkstra's single source shortest path algorithm

heap has slightly greater speedup than the concurrent priority pool on large number of processors (around ten). This could be caused by the fact that the decrease key operation on the parallel Fibonacci heap is more efficient.

5.5.2 Vertex Cover Problem

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \in V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ or both. The size of a vertex cover is the number of vertices in it. The vertex cover problem (VCP) is finding a minimal vertex cover for G [Vor87, PS82, CLR90, KRR88]. VCP is an NP-complete problem [CLR90]. As many other NP-complete problems, VCP can be attacked with branch-and-bound algorithms [LW66, jLW84, LS84].

Figure 5.11 shows a parallel branch-and-bound algorithm for VCP. In line 1 of the master procedure, an upper bound C_0 of the VCP is found by using a greedy algorithm, i.e., picking vertices with larger degree first to get a cover. We start from an empty cover and fork off some workers to search the state space of the VCP. The priority queue Q keeps track of all the partial subcovers that have better lower bound than C_0. Each worker repeatedly takes subcovers out of Q and puts bigger subcovers that have lower bounds smaller than C_0 into Q until the smallest vertex cover is found. In the pseudocode for the workers, line 2 extracts a subcover C. Line 6 finds a vertex x not in C that covers edges not already covered by C. We generate C's two successors C_1 and C_2 by either including x or excluding x in lines 7-10. Excluding x is equivalent to including all x's neighbors into the cover. In line 11, we compute the lower bounds for the newly generated subcovers. A lower bound b for a subcover C means that every vertex cover for G that contains C will be of size at least b. Intuitively, $b = |C|$ + the least number of vertices that have to be added into C to form a cover. We compute the second item by finding a match M of the graph uncovered by C[3]. Because at least one of the two endpoints of each edge in M has to be included in a vertex cover, $b = |C| + |M|$. In line 12, if we find a vertex cover that has better bound than the global bound C_0, then we replace C_0 with the new cover. We insert subcovers that have better lower bound than C_0 back into Q.

Figure 5.12 shows the speedup graph of VCP on a 50 vertex graph with degree randomly chosen from 0 to either 10 or 16. The sequential binary heap is used as the basis to compute speedup. The concurrent priority pool and the parallel Fibonacci heap

[3]A match is a set of independent edges, i.e., edges that do not share common vertex. We can use any kind of match to compute the lower bound here; the maximal match gives the best bound, but takes more time to find. In the experiments, a simple greedy match is used.

both have good scalable speedup whereas the concurrent binary heap saturates when the number of processors is more than ten. The graph also shows that the concurrent priority pool has slightly greater speedup than that of the parallel Fibonacci heap. Both the concurrent priority pool and the parallel Fibonacci heap have greater throughput when the degree upper bound of the vertices is bigger (i.e., 16 in the graph). The results are quite consistent with the synthetic data presented in the last few sections.

5.6 Summary

Some experimental results on different concurrent priority queues have been presented in this chapter. For the parallel Fibonacci heap, the parameters *buffersize* and *strictness* do not affect the running time much. In fact, the parallel Fibonacci heap performs fairly well in the quite strict case, when $buffersize = 1$ and $strictness = 1$. For the concurrent priority pool, the effects of the parameters *segnum* and *segsize* do not seem to affect the throughput much either. The comparison of different concurrent priority queues, as shown in Figure 5.6, indicates that the parallel Fibonacci heap has linearly scalable throughput; the concurrent priority pool has the largest throughput and at the same time it has a linearly scalable performance. The throughput of the concurrent binary heap saturates when the number of processes accessing it is more than about eight. The sequential binary heap's throughput decreases as the number of processors increases.

Two different types of applications of concurrent priority queues, namely single source shortest path problem and vertex cover problem, have been implemented. The single source shortest path problem is in the computational class P and can be efficiently solved by using sequential binary heaps. Both the parallel Fibonacci heap and the concurrent priority pool have good scalable speedup, although it is around 0.3 with 1 processor and 4.5 with 15 processors. The vertex cover problem is an NP-complete problem. Both the parallel Fibonacci heap and the concurrent priority pool have good scalable speedup. When the degrees of vertices in the graph are relatively large, the speedup is close to linear. The concurrent binary heap's speedup saturates when the number of processors is more than about ten. The results on applications are quite consistent with the synthetic data.

```
%% pseudocode for the single source shortest path problem
%% Find the shortest paths from source s to all other nodes in the graph
%% Data structure: the graph is represented as an adjecent ...

proc worker(q)

1   loop
2       n := extract-min(q)
3       if n = nil then
              %% q is empty
4              Termination test; see if the worker can quit
5       else
6              mark[n] := not-in-queue
               %% n has been taken out of q
7              For each neighbor in adj[n] do
8                     lock neighbor
9                     if d(n) + w(n, neighbor) < d(neighbor) then
10                          if mark[neighbor] = not-in-queue then
11                                insert neighbor into q with new-distance
12                          else
13                                decrease-key(neighbor, new-distance)
14                          end
15                    end
16                    unlock neighbor
17      end
18  end
19  end worker



    proc master

1   Q := ∅
2   Put source s in Q with priority 0
3   Fork off some workers to work on q
4   end master
```

Figure 5.9: Parallel single source shortest path algorithm

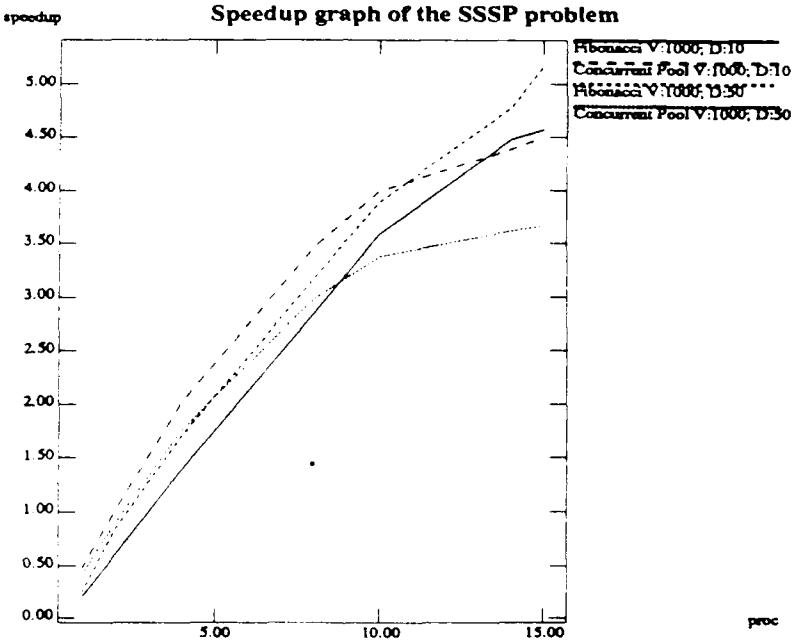


Figure 5.10: The speedup graph for the SSSP problem

```
    proc worker(Q)

1   loop
2         subcover := extract-min(Q)
          % subcover = (C, b) where C is the set of vertices and
          %      b is the lower bound (i.e., the key in Q).
3         if subcover = nil then
                % Q is empty
4               Termination test; see if the worker can quit
5         else
6               Find a vertex x not in the cover C such that x covers
7                     edges that are not already covered by C
8               Generate two subcovers C1 and C2
9               C1 includes vertex x
10              C2 includes x's neighbors
11              Compute the corresponding lower bounds b1 and b2
12              if one of the new subcovers forms a vertex cover that
13                    is smaller than the current cover C0 then
14                    replace the current cover with the new one

15              if newly generated subcovers have better bound than the current
16                    one then  insert them into Q
17  end
18  end worker


    proc master(G)

1   Generate an initial cover C0 using greedy algorithm
2   Q := empty cover with bound 0
3   Fork off some worker(Q)

4   end master
```

Figure 5.11: The branch-and-bound algorithm for the vertex cover problem

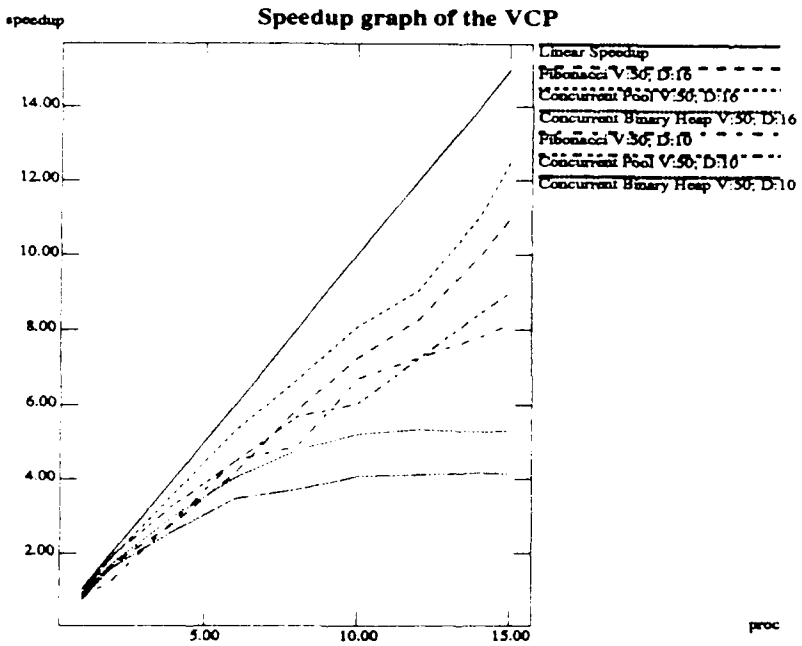


Figure 5.12: The speedup graph of the VCP

Chapter 6

Conclusion and Future Directions

6.1 Contributions

This thesis presented two novel concurrent priority queues: the parallel Fibonacci heap and the concurrent priority pool, both of which have non-strict semantics (see section 3.1). The parallel Fibonacci heap is based on the sequential Fibonacci heap, theoretically the most efficient data structure for sequential priority queues. This scheme employs distributed small critical sections so that it has linearly scalable throughput. The experimental results in Chapter 5 showed that the parallel Fibonacci heap has linearly scalable throughput that is larger than that of the concurrent binary heap with even small number of processors. A concurrent access scheme for a doubly linked list was described as part of the Fibonacci heap.

The concurrent priority pool, based on the concurrent B-tree and the concurrent pool, has the largest throughput among all of the priority queues tested, besides providing a easy way to control the quality of extracted nodes. The experiments showed that the concurrent priority pool also has linearly scalable throughput. The three kinds of concurrent priority queues, namely the parallel Fibonacci heap, the concurrent priority pool, and the concurrent binary heap, were evaluated on an Encore machine using the language Mul-T.

Two different types of applications of concurrent priority queues have been tested. One is the single source shortest path problem, which belongs to the computational class *P*. The other one is the vertex cover problem, an *NP*-complete problem. The results show that the parallel Fibonacci heap and the concurrent priority pool both have good scalable speedup on the applications whereas the concurrent binary heap saturates quickly. The speedup is larger on VCP than on SSSP.

6.2 Future Directions

6.2.1 More experiments

More experiments will be done when the simulator asim becomes practically usable.

6.2.2 Distributed Memory Model

The concurrent priority queues discussed in this thesis are mainly based on the shared memory model. Here, we discuss see how they can be modified to use a distributed memory model.

The parallel Fibonacci heap is nicely divided into many sections. In a distributed memory model, each processor can have a section in its local memory and the promising list may be replicated. The promising list does not have to be updated synchronously on all processors. The insert operation can insert in the process' local section, or randomly pick up a remote section to insert in depending on the network communication cost. The extract operation first tries to extract a local promising node. If there are no local promising nodes, the extract process finds remote promising nodes through the promising list. If the consolidation process finds that the quality of local nodes is not as good as nodes at remote processors, then some trees can be moved to balance the quality of nodes on different processors. Since a parallel Fibonacci section is a forest of trees linked in a doubly linked list, it is easier to move data around than if a section were a binary heap.

For the concurrent priority pool whose skeleton is a concurrent B-tree, we can implement each B-tree interior node and segment as an object. Since all the insert operations go through the B-tree root, we may want to replicate interior nodes close to the root on different processors to diffuse the traffic on the upper part of the B-tree [1]. Similarly, because all extract-min operations go through the leftmost leaf, it would be desirable to put different segments in the leftmost leaf on different processors.

6.2.3 Other Related Research

Kumar et al [KRR88] introduced several distributed binary heaps. They used three kinds of communication methods among processors to balance load: *blackboard*, *random*, and *ring*, and pointed out that the blackboard approach is the best.

[1] This problem is examined in Paul Wang's thesis[Wan90].

Driscoll et al [DGST88] have proposed a parallel priority queue for SIMD machines that is called a "relaxed heap". Van Emde Boas presented sequential priority queues [vEB75] that support insert, extract, delete and other operations in worst-case time $O(lg\ lg\ n)$, if all the keys in the priority queue are restricted in the set {1, 2, ..., n}. It would be interesting to see if a more efficient parallel priority queue can be built using this as a base.

Bibliography

[BB87] Jit Biswas and James C. Browne. Simultaneous update of priority structures. In *Proceedings of International conference on Parallel Processing*, pages 124–131, 1987.

[Bir89] Andrew D. Birrell. An introduction to programming with threads. Technical Report Research Report 35, Digital System Research Center, January 1989.

[BS77] S. Bayer and E. M. Schkolnick. Organization and maintenance of large ordered indices. *Acta Informatica*, 9:1–21, 1977.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithm*. The MIT Press and McGraw-Hill Book Company, 1990.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[DGST88] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, November 1988.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[jLW84] Guo jie Li and Benjamin W. Wah. Computational efficiency of parallel approximate branch-and-bound algorithms. In *1984 International Conference on Parallel Processing*, pages 473–480, 1984.

[KE89] David Kotz and Carla Schlatter Ellis. Evaluation of concurrent pools. *Proceedings of the International Conference on Distributed Computing Systems*, pages 378–385, June 1989.

[KHM89] D. Kranz, R. Halstead, and E. Mohr. Mul-t: A high-performance parallel lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implementation*, 1989.

[KRR88] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *National Conference of Artificial Intelligence*, pages 122–127, August 1988.

[LS84] T. H. Lai and S. Sanhni. Anomalies in parallel branch and bound algorithms. *Communications of the ACM*, 27(6):594–602, June 1984.

[LS86] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *1986 Fall Joint Computer Conference*, pages 380–389, 1986.

[LSS87] V. Lanin, D. Shasha, and J. Schmidt. An analytical model for the performance of concurrent b-tree algorithms. NYU Ultracomputer Note 311, NYU Ultracomputer Lab, 1987.

[LW66] E. L. Lawler and D. Woods. Branch-and-bound methods: a survey. *Operations Research*, 14:699–719, 1966.

[LY81] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations of B-trees. *ACM Transactions on Computer Systems*, 6(4):650–670, 1981.

[Man86] Udi Manber. On maintaining dynamic information in a concurrent environment. *SIAM Journal on Computing*, 15(4):1130–1142, November 1986.

[MR85] Y. Mond and Y. Raz. Concurrency control in B^+-tree databases using preparatory operations. In *11th International Conference on Very Large Databases*, pages 331–334. Stockholm, August 1985.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Press, 1980.

[Pea84] Judea Pearl. *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., 1982.

[RK88a] V. Nageshwara Rao and Vipin Kumar. Concurrent access of priority queues. Technical Report TR-88-06, The University of Texas at Austin, Department of Computer Sciences, February 1988.

[RK88b] V. Nageshwara Rao and Vipin Kumar. Concurrent insertions and deletions in a priority queue. In *Proceedings of the 1988 Parallel Processing Conference*, pages 207–211, 1988.

[Sen89] Stephanie Seneff. Tina: a probabilistic syntactic parser for speech understanding systems. In *Darpa Speech and Natural Language Workshop Proceedings*, February 1989. A LCS/MIT technical report version is forthcoming.

[vEB75] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, October 1975.

[Vor87] O. Vornberger. Load balancing in a network of transputers. In *2nd International Workshop on Distributed Algorithms*, pages 116–126, 1987.

[Wan90] Paul Wang. An in-depth analysis of concurrent B-tree algorithms. Master's thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Sq., Cambridge, MA 02139, 1990. Forthcoming.

[Wed74] H. Wedekind. On the selection of access paths in a database system. In J. W. Klimbie and K. L. Koffeman, editors, *Database Management*, pages 385–397. North Holland Publishing Company, 1974.

[Win84] Patrick H. Winston. *Artificial Intelligence*. Addison-Wesley, 2nd edition, 1984.

[WW90] W. E. Weihl and P. Wang. Multi-version memory: Software cache management for concurrent B-trees. To appear in the second IEEE Symposium on Parallel and Distributed Systems, Dallas, Texas, December 1990.