Experiments with Parallel Pointer-Based Algorithms

Margaret Reid-Miller
May 1998
CMU-CS-98-127

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Thesis Committee:
Guy E. Blelloch, Chair
Randy Bryant
Bruce Maggs
Daniel Sleator
Vijaya Ramachandran, U. Texas

School of Computer Science

DOCTORAL THESIS
in the field of
COMPUTER SCIENCE

Experiments with Parallel Pointer-Based Algorithms

MARGARET REID-MILLER

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

_____ THESIS COMMITTEE CHAIR   ____5/10/98____ DATE

_____ DEPARTMENT HEAD   ____5/19/98____ DATE

APPROVED:

_____ DEAN   ____5/19/98____ DATE

Abstract

Although parallel pointer-based algorithms have been studied extensively by the research community, implementations have met with marginal success, even for the simplest applications. In a careful implementation study of parallel list ranking (and the related list-scan operation) and parallel dynamic balanced trees operations, I show that indeed parallel pointer-based algorithms can have substantial speedup over fast workstations. List ranking is over 200 times faster on a CRAY C90 than on a high-end DEC Alpha workstation, and the balance tree algorithms are 6.3 to 6.8 times faster on eight processors than on one of a SGI Power Challenge, and 4.1 to 4.4 times faster on five processors than on one of a SUN Ultra Enterprise 3000. List ranking is a primitive in many parallel tree and graph algorithms, while dynamic balanced trees are important for maintaining databases, providing ordered set operations, and index searching. The parallel algorithms for list ranking and balanced trees are new; the key to their success is that they are both work optimal and very simple. In fact, the algorithms for set operations seem simpler than any previous sequential algorithms with the same work bounds, and might, therefore, be useful in a sequential context.

The algorithms as implemented, however, do not have optimal depth (parallel time). This dissertation shows how to reduce the depth of the tree algorithms to make them optimal by using pipelining. Pipelining has been used previously, but the method used here allows for asynchronous execution and for pipeline delays that are data dependent and dynamic. Rather than making the coding more difficult, the method lets the user write the algorithms using futures (a parallel language construct) and leaves the management of the pipelining to an efficient runtime system. To determine the runtime of algorithms, I first analyze the algorithms in a language-based cost model in terms of work and depth of the computations, and then show universal bounds for implementing the language on various machines.

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Linked lists and trees are two of the most fundamental data structures in computer science. They are so elementary and important that they appear early in introductory computer science courses and are ubiquitous in serial algorithms and applications. In addition, there has been much theoretical parallel algorithm work. But surprisingly, these structures are rarely used in parallel implementations, and there has been little performance experimentation. For example, not one of the NAS benchmarks uses lists or trees [10]. One reason is that early work has concentrated on scientific computing which, in large part, uses dense, regular, and static data structures such as arrays. Another is that the use of pointers in lists and trees makes them seem inherently sequential; developing parallel algorithms for them requires major rethinking. Finally, because accessing pointer-based data is communication intense, early parallel implementations have been disappointing, barely matching the then-current fastest workstation speeds.

There are several reasons, however, why pointer-based algorithms are likely to become more common. Scientific computing is moving to irregular, sparse, and dynamic applications and algorithms. For example, the space and time complexity of sparce problems using regular representations can be substantially reduced using pointer-based representations. Because shared-memory systems with a small number of processors are becoming quite common, there is more demand to move single processor applications to multiple processors. Thus, parallel computing is absorbing more applications, including those that use pointer-based objects. Finally, more general-purpose software solutions are available to improve irregular computation performance, such as high-level language support [58, 21], improved compilation techniques [14, 104, 80], advanced run-time sytems [57, 108, 26, 90], and efficient communication protocols [112, 106].

Over the last 20 years there has been much theoretical work in parallel pointer-based algorithm design [100]. Initially, algorithm designers developed algorithms that were asymptotically fast but not processor efficient. That is, the total work (the work efficiency) of the processors can be far greater than a constant factor more than the sequential time complexity of the problem [75]. Because work-efficient algorithms use far fewer processors or the same number of processors with each processor doing less work, algorithm designers now ensure that their algorithms are work-efficient. Often these algorithms are asymptotically very fast but are so complex and have such large constant factors that they are useful only for extremely large data sets.

Most of this theoretical work is based on the Parallel Random Access Memory (PRAM) machine model. Critics complain that the model is not realistic because it assumes that the time for memory access is comparable to the time for other operations and there is no cost for synchronization. On most multiprocessor machines memory access can be 100 times slower than a floating point operation. There are many reasons why there is such a time difference. In some cases there is high

communication overhead in which a processor is idle every time it sends and receives data requests. The physical distances of the data transmission and number of intermediate routing steps can vary greatly and result in high latencies (the time between the request for the data and the receipt or delivery of the data). Many machines have limited bandwidth on the assumption that application can take advantage of data locality (data either can be reused or is near other recently used data). Congestion on the network wire, at a router, or at a memory bank can cause further delays. To deal with these widely varying physical characteristics of multiprocessor machines, researchers have proposed a plethora of more realistic machine models and designed new algorithms for these models. But these models require the designer to consider more detail and often result in more complex algorithms than the corresponding optimal PRAM algorithms.

Since machines have finite resources, however, it is important to consider factors other than asymptotic running times, which rarely can be achieved in practice. This dissertation shows that the most important criteria for practical-size problems are work efficiency and small constants: Given that current supercomputer and Massively Parallel Processor (MPP) machines usually are used for problems that are much larger than the number of processors, the running time of an algorithm is dominated by the total work and its associated constants and not so much by asymptotics. Because there has been little theoretical work on the constants involved in the algorithms and because it is hard to take into account physical machine parameters in the machine models on which the algorithms are based, the best sources for comparison are actual implementations.

Considering that there have been hundreds of theoretical papers on pointer-based algorithms, implementation work has been quite limited. Hillis and Steele [61] implemented list ranking on the CM-2. Narayannan implemented data-parallel and replicated algorithms for single-source shortest path on the CM-2 and MP-1 [89]. Anderson and Setubal [4] implemented Goldberg's maximum flow algorithm on a Sequent Symmetry. Greiner [51] compared several parallel connected-components algorithms for general graphs on the CM-2 and the CRAY C90. Lumetta et al. followed with an implementation of a hybrid parallel/serial connected components algorithm on the distributed memory machines Cray T3D, Thinking Machines CM-5, and Meiko CS-2 [81, 74]. Hsu et al. [63, 65, 64] built a library of pointer-based algorithms for the MasPar MP-1, including connected components, open ear decomposition, and list ranking among others. Hsu et al. [66] further fine-tuned four connected-components algorithms for the MasPar MP-1 while Goddard et al. [78, 77] developed connected-components algorithms that use the faster 2-D mesh communication links of the MasPar MP-1. Most other connected-component implementations are restricted to grid graphs or specialized applications. Messeguer implemented skip lists on a CM-200 [85], while Gabarró and Petit implemented 2-3 trees on CM-200 [47].

An active area of research, however, is experimental work on parallel N-body algorithms for a variety of architectures (see [15] for an overview of some previous work). These algorithms simulate the movement of particles under some type of force. Although they use a quad-tree in 2-D and an oct-tree in 3-D to partition space, and thus can be classified as pointer-based algorithms, N-body problems are more coarse-grain and computationally intensive than the fine-grain low-computation applications studied above. Consequently, the problems and techniques needed for good performances are quite different for the two types of applications. This dissertation focuses on the fine-grain pointer-based algorithms, for which there is little performance data.

1.1   Thesis Statement

If the PRAM is unrealistic, then do PRAM algorithms have any value? What is the performance of PRAM pointer-based algorithms in practice? Do they necessarily have poor performance? Can

carefully hand-coded implementations on the best hardware be fast? What hardware and software requirements are necessary for good performance? Which parallel algorithms perform well and how well, and what are their characteristics? These questions motivated the work in this dissertation, and the results herein provide some answers.

Thesis Hypothesis: Pointer-based algorithms, used so commonly in serial computation, can be fast on parallel machines with good memory subsystems.

Previous work implementing pointer-based PRAM algorithms have been disappointing, barely matching serial algorithms implemented on fast workstations available at the same time as the parallel machines. One problem is that machines, such as the CM-2, CM-5, and MP-1, are multiprocessor machines that were built using custom processor chips that were much slower than contemporary commodity processor chips found in workstations. In addition, these machines have long memory access latencies and and insufficient bandwidth for such fine-grain algorithms. On the other hand, even commodity processors are not necessarily sufficient. The connected component implementation on the Cray T3D [81] shows good performance on graphs that can be easily be partitioned between processors, but poor performance on graphs that are hard to partition because the machine has distributed memory. The best overall performance for connected components was obtained on the CRAY C90 [51], which has extremely high memory performance.

Many PRAM algorithms have large constants. With sequential algorithms it is often the simpler algorithms with the same (expected, amortized) asymptotic work that give the best performance, for example quicksort and splay trees. It is not clear that previous parallel implementations have sufficiently emphasized finding the simplest algorithms. Even with seemingly simple parallel algorithms, frequent synchronization, contention avoidance, and many-way case statements can increase constants dramatically. Frequent synchronization, for example, to keep processors in lock step, can ruin performance, especially on machines without specialized hardware; frequent contention avoidance, used by some pointer-based algorithms, can significantly reduce performance; and frequent testing of a number of cases, especially ones that cannot overlap, can make an algorithm impractical.

Measure of success: To develop parallel pointer-based algorithms and their implementations that show reasonable speedup and are significantly faster than serial implementations on fast workstations.

By examining actual performance of parallel pointer-based algorithms we can start to understand what makes good parallel-algorithm design. If the algorithms that have the best performance are quite different from the ones being designed by the theory community, then my work may influence future theoretical work. For example, Ranade [97] recently designed a list-ranking algorithm that uses ideas I present in this dissertation. In addition, if algorithm developers place more emphasis on designing simple parallel algorithms with small constants, some designs may also provide new fast sequential algorithms. Finally, by demonstrating that parallel pointer-based algorithms can be fast, my work may show the potential for a wide variety of important applications. If there is an increased demand for pointer-based solutions, then there also may be increased pressure on hardware manufacturers to provide platforms that address the needs of these algorithms.

1.2 Methodology

In this dissertation I consider two fundamental pointer-based problems, list ranking and maintaining dynamic balanced trees. They pose the same challenges common to most fine-grain pointer-based parallel algorithms: the parallel algorithms can be quite different from their serial counterparts and often having much higher overheads, memory accesses that are hard to optimize and cannot be scheduled at compile time, and huge communication bandwidth requirements.

The work involves appraising various algorithms and data structures, designing and analyzing new algorithms, implementing the algorithms on high-performance hardware, and performing experiments that compare the sequential algorithm with one or more parallel algorithms. The two implementation studies are:

- List ranking on a CRAY C90 vector multiprocessor. List ranking is the simplest of all pointer-based problems serially, while one of the hardest to get good parallel performance. The parallel algorithms are quite different from the serial one. On the other hand, the CRAY C90 is one of the fastest machines on which to run list ranking. Thus, in this setting we can see what the best performance we can expect is for the problem.

- Maintaining balanced trees on shared-memory multiprocessors. The implementation is on an SGI Power Challenge and SUN Ultra Enterprise 3000 and uses the Cilk [26] runtime system to maintain and schedule multiple threads efficiently. While balanced trees have more obvious parallel implementations, the hardware platforms do not provide as good performance as the CRAY C90. However, even this problem shows reasonable speedup.

In both cases the algorithms do not provide optimal asymptotic depth*(parallel time): The algorithms are $O(\log^2 n)$ depth, when $O(\log n)$ depth is possible.[†] They are work efficient, however, in that they use the same number of operations, within constants, as their serial counterparts. In addition, the algorithms have small constants relative to optimal parallel algorithms, and hence are competitive with the serial algorithms when the problem size is large compared to the number of processors. The parallel balanced-tree algorithms are so simple that they may also be practical for serial implementations.

On a more theoretical side, I show that the balanced binary tree operations can be made optimal using pipelining to provide $O(\log n)$ depth algorithms. The pipelining framework is more general, though, in that it can be applied to a variety of algorithms. The novelty of the pipelining, for this algorithm and others, is that the depth of the pipeline is data dependent and dynamic. Even so, I show, in joint work with Guy Blelloch, that by using futures (a parallel language construct) a user implementing the algorithms does not need to manage the pipelining explicitly. A runtime system manages it implicitly. The result is much simpler code and more asynchronous execution.

The next three subsections gives an overview of the results of the list ranking implementations, the balanced tree implementations, and the pipelining framework. Chapters 2, 3, and 4 provides further details of the work. Finally, Chapter 5 discusses my contributions, future work, and some conclusions.

---

*If we represent an algorithm by a bounded fan-in circuit (directed acyclic graph) then the time to evaluate the circuit in parallel is its *depth* and the total computational work done by the circuit corresponds to its *size*, the number of noninput nodes. In Chapter 3 I include subgraphs in the circuit that have unbounded fan-in. The number of edges in the subgraphs, however, is asymptotically the same as the number of nodes. Therefore, work still corresponds to the number of noninput nodes.

[†]I use ln for $\log_e$, lg for $\log_2$, and log when the base is arbitrary.

1.3   List Ranking and List Scan

The first problem this dissertation considers is finding the rank or position of each node in a linked list, known as *list ranking,* and the related problem of finding the prefix sum of a linked list, know as *list scan.* Although, list ranking is one of the simplest of serial pointer-based algorithms, it is hard to get vector or parallel performance that is significantly faster than the serial performance. Because of data dependencies, there are no trivial work-efficient parallel algorithms. Most tend to be quite complex and have large constants. In contrast, the serial algorithm has very small constants. But even with a good parallel algorithm a compiler cannot schedule the data movement because the access patterns cannot be determined until runtime. Finally, the ratio of communication to computation is extremely high and, hence, any implementation is at the mercy of the memory system.

List ranking has been studied extensively by the theory community and is used as a primitive for many tree and graph algorithms [1, 49, 48, 72, 76, 86, 87, 95, 101]. Table 1.1 gives a comparison of list-ranking algorithms. Wyllie's algorithm [115] is not work efficient since it takes $O(n \log n)$ operations on a n vertex linked list, whereas a serial algorithm takes $O(n)$ operations. But, because Wyllie's algorithm is simple it works well for short lists or when we can increase the number of processors according to the linked-list size. On the other hand, work-efficient randomized pointer-jumping techniques suffer from having to take multiple trials on average before being able to perform a pointer jump and, therefore, results in larger constants. The known optimal deterministic parallel PRAM list-ranking algorithms have very large constants, which makes them impractical.

Algorithm	Time	Work	Constants	Scratch Space
Serial	$O(n)$	$O(n)$	small	c
Wyllie [115]	$O(\frac{n \log n}{p})$	$O(n \log n)$	small	$n + c$
Blelloch/Reid-Miller	$O(\frac{n}{p} + (\log n)^2)$	$O(n)$	small	$5p + c$
Randomized [86, 6]	$O(\frac{n}{p} + \log n)$	$O(n)$	medium	$> 2n$
Optimal [40, 41, 5]	$O(\frac{n}{p} + \log n)$	$O(n)$	very large	$> n$

Table 1.1   Comparison of several list ranking algorithms, where n is the length of the list, p is the number of processors, and c is a constant.

1.3.1   Performance

I implemented a few of the simplest list-ranking algorithms on the CRAY C90 to compare their performance with the parallel algorithm I developed with Guy Blelloch. They are the serial algorithm, Wyllie's algorithm, and randomized pointer-jumping algorithms of Miller/Reif [86] and Anderson/Miller [6]. Other work-efficient algorithms have much larger constants than the ones I implemented and, therefore, are likely to have worse performance. On one processor the parallel algorithms were vectorized, while the serial algorithm used no vectorization. Only my algorithm is faster than the serial algorithm and faster only when the list length was greater than a thousand. Wyllie's algorithm is almost as fast as the serial algorithm for moderate length lists. But for long lists, the longer the list is the poorer its performance. This performance is expected, since the algorithm has small constants but is not work efficient. The Anderson/Miller randomized pointer-jumping algorithm is only sightly slower than the serial and the Miller/Reif algorithm as about

three times slower. All the parallel algorithms, however, outperform the serial algorithm given enough processors and a sufficiently large linked list.

My implementation of list ranking and list scan on the CRAY C90 has continued, for four years, to be the world's fastest implementation, to the best of my knowledge. In addition, it is the first implementation of which I am aware that significantly outperforms fast workstations. For example, Table 1.2 shows that my list ranking algorithm is 200 times faster on 8 processors of the CRAY C90 than on a high-end DEC 3000/600 Alpha workstation. Also, on one processor of the CRAY C90 the vectorized algorithm is over eight times faster than the simple serial algorithm on the CRAY C90, which is significant since CRAY vector computers are very fast scalar machines (see Section 7.8 of [59]). On 8 processors the vector parallel algorithm achieves a 50 fold speedup over the CRAY C90 serial code.

Algorithm	DEC Alpha		Cray C-90				
	cache	memory	Serial	Vectorized	2 Proc.	4 Proc.	8 Proc.
List Rank	98	690	177	21.3	10.9	5.8	3.1
List Scan	200	990	183	30.8	16.1	8.5	4.6

Table 1.2 Comparison of the asymptotic execution time (nsec) per vertex of our vector parallel list scan and list ranking algorithm on a CRAY C90 and the serial algorithm on a CRAY C90 and a DEC 3000/600 Alpha workstation. On the Alpha, the table show times for when all the data can fit in the cache and when it must be stored in main memory.

1.3.2   Reasons for success

The primary advantage of my algorithm is that it is both work efficient and has small constants. It is also fully vector parallel. On the other hand, it uses a large number of rounds of communication. To perform well, therefore, it requires a machine that has either low latency or moderate latency with latency-hiding capabilities. Due to the nature of the problem, the machine also needs to have high memory bandwidth. The Cray vector multiprocessors have these capabilities and, as I argue later, closely resemble a PRAM. Thus, on the CRAY C90 our algorithm scales almost linearly with the number of processors. There is, however, some degradation in performance as the number of processors increases because the available memory bandwidth per processor decreases.

The implementation of my list ranking algorithm is successful for several reasons:

- I use high performance hardware. The most successful implementations of irregular and pointer-based algorithms have been on the Cray YMP class of vector multiprocessors [24, 117, 98]. This class of machines has the best communication hardware available.

- I use latency-hiding techniques to minimize bottlenecks in the hardware. By using many more virtual processors than physical processors and treating each element of a vector as a virtual processor, vectorization is possible. Whenever vector lengths are long, latencies to the functional units and memory can be hidden through pipelining.

- I designed an algorithm and used my analysis of it to minimize overheads. Most importantly, the algorithm has small constants. It uses randomization to break symmetry and to minimize memory contention. In addition, randomization minimizes the need for load balancing between processors (the implementation does none). Another form of load balancing, however, is required by vector processing. Completed virtual processors need to be removed from

the computation by compressing vectors. I minimize the frequency of this compression by analyzing the algorithm analytically and empirically.

1.4 Balanced Trees

After linked lists, the next most common pointer-based data type is the tree. Trees are used extensively for data bases, parsing, computational geometry, text compression, and operating-system applications, to name a few. A common use of balanced trees is to provide ordered set operations or to maintain dynamic dictionaries. Sequentially these trees, such as AVL, red-black, and splay trees, provide $O(\log n)$ time insert, delete, and search operations. Some also provide $O(\log n)$ time split and join operations. They also provide aggregate operations *intersection, union,* and *difference*, which can be performed sequentially or in parallel. These operations play an important role in, among other applications, databases queries and index searching [114, 83]. Like list ranking, the access patterns of balanced trees are data dependent and, therefore, can only be scheduled at runtime. In addition, as data locality is low and there is little computation for each memory access, the communication to computation ratio is very high. On the other hand, by their nature trees have obvious divide-and-conquer algorithms that make them easier to parallelize.

As with list ranking, previous optimal parallel balanced-tree algorithms are far too complex to get good performance for realistic data sizes on existing hardware. Paul et al. [92] developed the first $O(\log n + \log m)$ depth $O(m \log n)$ work EREW PRAM algorithms for searching m keys in, inserting m keys into, and deleting m keys from a 2-3 tree of n nodes. Highan and Schenk [60] extended the work to B-trees. Both sets of algorithms use pipelining to reduce the depth of the algorithms from $O(\log n \log m)$ to $O(\log n + \log m)$. But the pipelining requires that the operations are carefully timed and adds complexity. Even without pipelining, although conceptually simple, whenever node splitting or merging is required the parallel algorithms need to consider more cases (especially for delete) than the corresponding sequential algorithms. Katajainen removes the need for pipelining in his EREW PRAM algorithms for set union, intersection, and difference using 2-3 trees. He claims that they are optimal $O(\log n + \log m)$ depth and $O(m \log(n/m))$ work algorithms, but the work analysis is not correct. It may be possible to modify the algorithms, however, to meet the bounds. Ranade [96] gives algorithms for processing least-upper-bound queries and insertion on distributed memory networks. Bäumker and Dittrich [13] give algorithms for search, insertion, and deletion into BB*(a) trees for the BSP* model. These algorithms require $O(m \log n)$ work and appear to have large constants.

For this thesis, I chose to use treaps [103] to develop parallel balanced-tree algorithms for several reasons. The structure of the trees is determined by the keys and random values used for balancing and not by the order in which the nodes are inserted and deleted or by global factors. The serial operations are simple and have competitive running times with other balanced trees. But most importantly the corresponding parallel operations are also simple and thus have small constants. In particular, the parallel delete operation is especially simple compared with other parallel balanced-tree delete operations.

1.4.1 Features of parallel treap algorithms

Treaps have only been studied in the sequential context and little is known about their performance. I extend previous work of treaps by providing and analyzing parallel algorithms on treaps and by presenting sequential and parallel experimental results that demonstrate their utility. Although the algorithms use two of the same building blocks used in the sequential algorithms (split and join),

the design and analysis of these new algorithms is quite different. The algorithms I present have the following properties, which together make them attractive from both a theoretical and practical point of view.

- For two sets of size m and n with $m \leq n$, the algorithms for union, intersection, and difference run in expected $O(m \log(n/m))$ serial time or parallel work. This is optimal.

- The algorithms are significantly simpler than previous parallel and serial algorithms with matching work bounds. I therefore expect that the algorithms could be useful even in a sequential setting. This dissertation includes the full C code for these algorithms.

- The parallelism in the algorithms comes purely from their divide-and-conquer nature. The algorithms are therefore easy to implement in parallel and well suited for asynchronous execution. I implemented the parallel versions of the algorithms in Cilk [26] with only minor changes to the sequential C code.

- The algorithms are fully persistent—they create the resulting set while leaving the input sets untouched.‡ Such persistence is important in database and indexing applications.

- The algorithms are efficient when the results are from interleaving blocks of the ordered input sets. For example, a variant of the union algorithm requires expected serial time (or parallel work) that is only logarithmic in the input sizes when the inputs have only a constant number of blocks.

- The algorithms use the same representation of treaps as Seidel and Aragon [103] so that all their algorithms can be used on the same data without modification.

1.4.2   Performance

The SGI Power Challenge and SUN Ultra Enterprise 3000 multiprocessors that I used to implement these algorithms are somewhat different than the CRAY C90 vector multiprocessor I used for list ranking. All are shared-memory machines. But the CRAY has no cache and has fine-grain pipelined memory access, while the SGI and Sun machines use a coarser grain, cache-coherent memory management system with cache lines of 128 bytes and 64 bytes, respectively. The bandwidth-to-memory relative to processor speeds for the SGI and Sun are not as high as for the CRAY. My implementation of the tree algorithms uses the Cilk [26] runtime system that schedules multithreaded computations using work-stealing. Cilk uses asynchronous threads, whereas vector processing on the CRAY is synchronous. Therefore, because of the lower bandwidth and coarser granularity, we cannot expect the performance on the SGI and Sun to be as good as on the CRAY for these pointer-based algorithms. On the other hand, asynchronous processing is better suited to these tree algorithms.

Sequentially, I show that search, insertion, and delete on treaps has a similar performance to red-black, splay trees, and skip lists. My experiments also demonstrate that the aggregate operations, intersection, union, and difference have sublinear performance when the tree sizes are unequal and even better sublinear performance when the results are from interleaved blocks of the ordered input sets. In parallel these operations have reasonable speedup: 6.3 to 6.8 speedup on 8 processors of the SGI Power Challenge, and 4.1 to 4.4 speedup on 5 processors of the SUN Ultra Enterprise 3000.

---

‡Note that just copying the inputs does not work since copying would violate the $O(m \log(n/m))$ work bounds—it would require $O(n + m)$ work.

1.5 Pipelining With Futures

The balanced-tree algorithms and my implementation of them as presented in the previous section have $O(\log n \log m)$ depth. I also show how the same algorithms can have $O(\log n + \log m)$ depth using pipelining. Pipelining in parallel algorithms takes a sequence of tasks each with a sequence of steps and overlaps in time the execution of steps from different tasks. Due to dependencies between the tasks or the required resources, pipelined algorithms are designed such that each task is some number of steps ahead of the task following it. I refer to this number of steps as the pipeline depth. The idea of using pipelining is not new. Paul, Vishkin, and Wagener [92] used pipelining to reduce the computation depth of their EREW PRAM algorithms on 2-3 trees and Cole [37] used it to reduce the computation depth of merge sort. But previous PRAM pipelining algorithms have been synchronous, and the depth of the pipeline is fixed throughout the execution of the algorithm. The pipelining of the treap algorithms, however, is asynchronous and the pipeline depth can vary throughout the computation: The time (depth) from the start of a task to when the first result of the task is ready to be processed by the next task is data dependent and may differ from task to task. For these algorithms, and another tree algorithm I give, there would be no reduction in the depth of the algorithms if the pipeline depth was fixed.

Although pipelining has led to theoretical improvements in algorithms, from a practical point of view pipelining can be cumbersome for the programmer—managing the pipeline involves careful timing among the pipeline tasks and asynchronous processing adds another layer of complexity. In this dissertation, I show that many algorithms can be automatically pipelined using futures, a construct designed for parallel languages. If the parallelism in the algorithm is expressed using the futures, the algorithm code need not change to get this pipelined depth reduction; the pipelining is expressed implicitly and managed automatically by a runtime system. In addition, if code is of a restricted form, called linearity, the algorithms have no concurrent memory access and the runtime system is simpler.

To analyze the running times of algorithms I use a two step process. I first consider a language-based cost model based on futures and analyze the algorithms in this model. The model is a slight variation of the PSL model [52]. Then, together with Blelloch, I show universal bounds for efficiently implementing the model on various machine models.

I describe and analyze four algorithms with the language-based model. The first is a merging algorithm. It takes two binary trees with the keys sorted in-order within each tree and merges them into a single tree sorted in-order. The code is very simple and, assuming both input trees are of size n, the nonpipelined parallel version requires $O(\log^2 n)$ depth and $O(n)$ work. I show that, by using the same code but implementing it with futures, the depth is reduced to $O(\log n)$, which meets previous depth bounds. The next two algorithms use a parallel implementation of the treap data structure. I show randomized algorithms for finding the union and difference of two treaps of size m and $n, m \leq n$ in $O(\log n + \log m)$ expected depth and $O(m \log(n/m))$ expected work. Like the merge algorithm, the code is simple. There are no previous parallel or pipelined results for treaps of which I am aware. These three algorithms require a dynamic pipeline, which varies its depth depending on the input data. As such asynchronous algorithms have not been considered before, I developed a new technique for analyzing their computation depth. The fourth algorithm is a variant of Paul, Vishkin and Wagener's (PVW) 2-3 trees [92]. Because the bottom-up insertion used in the PVW algorithm does not map naturally into the use of futures, I describe a top-down variant that does. As with the PVW algorithm, the pipelining improves the algorithm complexity for inserting m keys into a tree of size n from $O(\log n \log m)$ to $O(\log n + \log m)$ depth and $O(m \log n)$ work. It can be implemented synchronously and with a fixed pipeline depth.

To complete the analysis I next consider implementations of the language-based model on various machines. The work and depth costs along with Brent's scheduling principle [29] imply that, given a computation with depth d and work w, there is a schedule of actions onto processors such that the computation will run in $w/p + d$ time on a p-processor PRAM. This principle, however, does not tell us how to find the schedule online—in particular it does not address the costs of dynamically assigning threads to processors nor the cost of handling the suspension and restarting required by futures at runtime. Since many of the algorithms are dynamic, the schedule cannot be computed offline. I show how a single runtime system can efficiently manage the pipeline and schedule the threads of any pipelined algorithm coded with futures. The runtime system is guaranteed to be efficient and, if the algorithm is of a certain restricted type, to use no concurrent memory accesses. Furthermore, the schedule can be optimized, for example, for space efficiency or locality.

Chapter 2

List Ranking and List Scan on the Cray C90

In this chapter I consider one of simplest pointer-based operations, list ranking (and the related list scan), and its parallel implementation.* My aim was to understand the problems involved in implementing parallel pointer-based algorithms and their possible solutions, and to see how well these kinds of algorithms perform in practice. First I introduce list ranking and compare the performance of several list ranking algorithms. Then I describe vector multiprocessors and their close relationship to the PRAM model. In Section 2.2 I highlight the five list-ranking algorithms I implemented. In Section 2.3 I describe my implementation of Blelloch/Reid-Miller list ranking algorithm in more detail and give timing data. Then, in Section 2.4 I give a cost model of my algorithm, analyze its expected performance, and describe how I tuned the parameters. In Section 2.5 I describe the multiprocessor version of the algorithm and its performance. In Section 2.6 I review some other PRAM list-ranking algorithms. Finally, in Section 2.7 I discuss my conclusions and future directions. In an appendix to the chapter, I present pseudo-C code for list scan and give a detailed account of the vector operations required to execute the code.

2.1 Introduction

List ranking finds, for each vertex in a linked list, the number of vertices that precede that vertex in the list. This information, for example, can be used to reorder the vertices of a linked list into an array in one parallel step. *List scan* computes, for each vertex in the linked list, the "sum" of the values of all prior vertices in the list, where "sum" is a binary associative operator. List ranking and list scan are related in that list ranking is the list scan where integer addition is the operator and the values to be summed are all equal to one. Because list ranking needs to read the link data only, whereas list scan needs to read both the link and value data, list ranking is usually faster than list scan (see Table 1.2). In addition, the scan operator can be more costly to compute than the increment operator used in list ranking. List-ranking and list-scan algorithms are otherwise the same.

For this dissertation I introduce a new list-ranking algorithm and its implementation on a CRAY C90 vector multiprocessor. I chose the CRAY C90 because, compared to other commercial architectures, it has very high memory bandwidth and fine-grain access to memory, and I wanted to see how well list ranking could perform under the best of circumstances. Because list ranking

---

*This chapter in large part is taken from [98]

makes great demands of the memory system, it would certainly have much worse performance on other architectures. To hide memory latency I used virtual processing. To avoid memory bank contention I used randomization. To get the best performance possible I developed a cost model of the algorithm, empirically determined the execution time of each parallel loop, and then analytically tuned the parameters.

Although list ranking is simple, it typifies the kinds of problems for which it is hard to get good vector or parallel performance. In particular, it uses an irregular data structure, it is communication intensive, and its communication patterns are data dependent and dynamic. From an algorithmic point of view it is interesting because it has features common to many problems: symmetry breaking and load balancing. Furthermore, because the serial algorithm is so simple, the parallel overhead must be kept small so that the parallel implementation compares well with a serial implementation. The problem is that no previous parallel algorithm is simultaneously work efficient and has small constants.

Given that current supercomputers and massively parallel processor machines are usually used for problems that are much larger than the number of processors, the running time of an algorithm is dominated by the total work and its associated constants and not so much by the asymptotic time. Therefore, my goal was to design an algorithm that both is work efficient and has small constants, even if it meant sacrificing optimal depth. To evaluate the performance of my algorithm, I implemented the list-ranking algorithms that are likely to be the most competitive: serial, Wyllie, Miller/Reif [86], and Anderson/Miller [6]. The latter two use randomized pointer jumping. Figure 2.1 shows the list-scan execution times on one CRAY C90 vector processor. Wyllie's algorithm is faster than mine for lists shorter than 1000 vertices. But for longer lists, my algorithm outperforms other algorithms. In addition, the space required by my algorithm, beyond what is needed for the list, depends on the number of (virtual) processors, which can be substantially less than the space required by other algorithms (see Table 1.1).
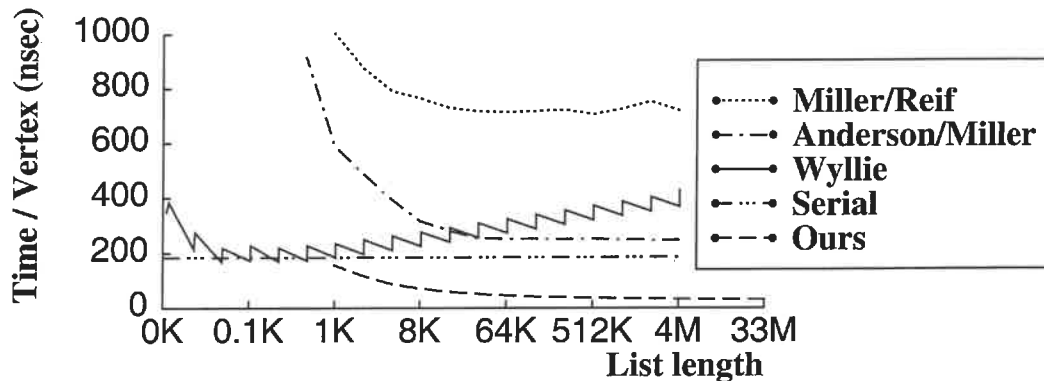
Figure 2.1:   Time per vertex for several list-scan algorithms on one CRAY C90 vector processor. The times for Wyllie's algorithm and our algorithm were obtained on a dedicated machine. The sawtooth shape of the Wyllie curve is due to the discontinuity of $\lceil \log n - 1 \rceil$, which is the number of rounds of pointer jumping done by the algorithm.

2.1.1   Vector Multiprocessors as PRAMs

I chose to implement list ranking on a vector multiprocessor because these machines, such as the CRAY family of vector computers, closely approximate the abstract EREW PRAM machine (see

Figure 2.2). They use a shared memory model, have fine-grain access to memory, have high global-communication bandwidth, and can hide functional and memory latencies through vectorization. The most important features that distinguish these machines from massively parallel processor machines are the high global-communication bandwidth and the pipelined memory access. Processors communicate to memory via a multistage butterfly-like interconnection network. As long as there are no memory-bank conflicts, the network can service one memory request per clock cycle for each memory pipe. Thus, the PRAM model assumption that often is cited as unrealistic, namely unit-time memory access, holds on vector multiprocessors as long as the algorithm can avoid memory-bank conflicts and hide latencies.
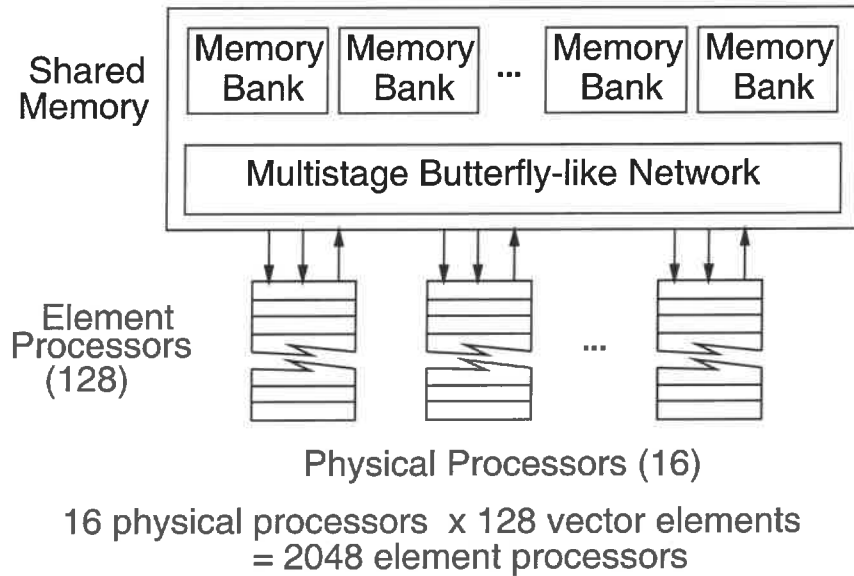
Figure 2.2: Vector multiprocessors viewed as a PRAM. Numbers are those for the CRAY C90.

Zagha [116] proposes several vector multiprocessing programming techniques for avoiding memory-bank conflicts and hiding latencies. To address bank conflicts he proposes a data distribution technique to manage the memory system explicitly. To address memory and functional-unit latencies he proposes that programs are written for sufficiently more *virtual processors* than physical processors. For vector multiprocessors, virtual processing allows for vectorization so that computation and communication can be pipelined to hide latencies. Virtual processing has been a common approach for hiding latencies [109, 84].

Using Zagha's programming model, I implemented the PRAM algorithms by treating a vector processor as a SIMD (distributed memory) multiprocessor. The ith elements of the vector registers of length L act as the local data for the ith element processor of an L-processor SIMD machine, see Figure 2.2. I call the processors *element processors* to distinguish them from a full vector processor. As processors in data-parallel algorithms do not use the results of another processor in the same time step, there are no recurrences to worry about in the corresponding vectorized implementation.

To amortize latencies, I attempt to keep the vector lengths long, close to the length of the vector registers or longer. When the work load is imbalanced, so that processors finish at different time steps, I use either strip-mining [91] or loop-raking [117, 22] to assign the work of several *virtual processors* to a single element processor. That is, element processor i is assigned virtual processors $j \cdot l + i$ in strip-mining and $i\lfloor n/(l-1)\rfloor + j$ in loop-ranking, where $i = 0, \ldots, l-1$ and $j =$

$0, \ldots, \lfloor n/(l-1) \rfloor - 1$ and $l \le L$. As processors complete, I reassign virtual processors to the element processors. Extending the vectorized algorithm to vector multiprocessors is straightforward: If the machine is SIMD, I simply treat the p vector processors as an $L \times p$ SIMD multiprocessor and apply the vectorized algorithm. If the machine is MIMD, I can treat it the same way except that, for efficiency, I minimize the frequency of load balancing across physical processors and processor synchronization.

2.2   The List-Ranking and List-Scan Algorithms

To evaluate the performance of Blelloch/Reid-Miller parallel algorithm, I implemented several list-ranking algorithms. I chose algorithms that have the smallest constants because they are the most likely to have the best performance for practical linked-list lengths. In Section 2.6 I conclude that other work efficient algorithms have much larger constants than the ones I implemented and, therefore, would have worse performance. Below I describe the main features of the five algorithms I implemented. Since list-ranking and list-scan algorithms are essentially the same I generally describe the list-scan algorithm only. For simplicity I use integer addition as the "sum" operator.

2.2.1   The serial algorithm

The serial list-scan algorithm simply walks down the list storing the accumulated values of the previous vertices until it reaches the end of the list. From Table 1.2 we see that the Cray C90 serial list-scan and list-ranking times are very nearly the same. The times are similar because the Cray C90 has two input ports that can bring in both the link and value data simultaneously. On workstations the list-ranking execution times are substantially faster than the list-scan times and both times depend on whether all the linked-list data can be placed in the cache or not.

2.2.2   Wyllie's algorithm

The first parallel list-ranking algorithm was introduced by Wyllie [115]. The algorithm uses a technique common to most parallel list-ranking algorithms, "pointer jumping" or "shortcutting". A processor is associated with every vertex and each processor repeatedly replaces its successor pointer with its successor's successor pointer in unison with the other processors. The new value for the vertex is its old value plus the value of its successor. For a list of length n and after $\lceil \lg(n-1) \rceil$ rounds of pointer jumping, every vertex points to the tail of the list and its value is the sum of the values from the vertex to the tail of the list. Although this algorithm is simple and has an $O(\log n)$ running time, it is not work efficient since the total number of operations is $O(n \log n)$, whereas the serial algorithm takes $O(n)$ operations.

Figure 2.1 shows the vectorized execution times of Wyllie's algorithm on one processor of the Cray C90. The sawtooth shape of the curves is due to the addition of another round of pointer jumping whenever the value of $\lceil \log(n-1) \rceil$ changes. The negative slope between a pair of teeth is due to the amortization of the additive constant terms over larger size lists. As one can see from Figure 2.1, Wyllie's algorithm quickly degrades in performance as the list lengths grow. On the other hand, as it scale almost linearly with the number of processors, it is faster than the serial algorithm when run on multiple processors on moderately long lists.

2.2.3 Miller/Reif random mate

One of the simplest work-efficient parallel algorithms was devised by Miller and Reif [86, 99]. It uses randomization for symmetry breaking so that processors at neighboring vertices do not attempt to dereference their successor pointers simultaneously. Each processor flips an unbiased male/female coin. If it is a female and its successor is a male (a "random mate") then it "splices out" its successor. The processor for the successor vertex becomes idle. On each round processors splice out only 1/4 of the remaining vertices on average. With high probability after $O(\log n)$ rounds all the vertices of the list are spliced out. Finally, there is a reconstruction phase, in which spliced-out vertices are reintroduced in reverse order from which they were removed. The constants for this algorithm are greater than for Wyllie's algorithm because it needs to generate random numbers and perform extra communication to establish random mates with successor vertices, takes on average 4 attempts to splice out each vertex, requires load balancing, and has both a reduction phase and a reconstruction phase.

I implemented this algorithm using the vector units on a single processor of the CRAY C90. My version removes idle processors on every round by compressing the remaining vertices into contiguous vector elements (an operation I call "pack"). This algorithm is 20 times slower than Blelloch/Reid-Miller algorithm and 3.5 times slower than the serial algorithm for long linked lists.

2.2.4 Anderson/Miller random mate

Anderson and Miller [6, 99] modified the above algorithm so that it avoids load balancing (packing). The p processors are assigned the work of splicing out a queue of n/p vertices. At each round a processor attempts to remove one vertex in its vertex queue. After a processor splices out a vertex, on the next round it attempts to splice out the next vertex in its queue. In this simple way processors remain busy without load balancing. When there are fewer than p vertices left, it compresses these vertices in memory and applies Wyllie's algorithm. Finally, there is reconstruction phase to reintroduce spliced out vertices.

To determine if a processor can splice out a vertex, all the vertices are set to female, except those at the top of the queue. These vertices are assigned male or female by a random toss of a coin. A vertex can be spliced out if it is a male pointed to by a female. If the coin is unbiased then in the initial rounds the processors remove on average up to $p/2$ vertices each round. But as the processing proceeds, the average number of processors that can remove a vertex each round drops to 1/4 of those left. Anderson and Miller show that if the number of processors, p, is $n/\log n$ then after a little over $4\log n$ rounds p vertices are left.

I attempted to optimize the Anderson/Miller algorithm to see how well it could perform. The most important optimization was changing the coin bias so that the probability of assigning male was 0.9. The effect was that almost 90% of the active processors could splice out on every round, even when the total number of remaining vertices was small. This high rate continued because as some processor queues completed and the remaining queues continued to have several (female) vertices on them. The result was to reduce the number of rounds and the run time by about 40%. Switching to Wyllie's algorithm was not useful because the number of rounds needed to complete without Wyllie's was not much greater than with Wyllie's. However, I did switch to the serial algorithm when only a few queues remained. This result is in contrast to that found by Hsu and Ramachandran[63] on the MasPar MP-1. They found that switching to Wyllie's algorithm greatly reduced the number of rounds needed. The difference is that they had 16,383 processors and were using an unbiased coin whereas I had only 128 element processors and was using a biased coin. For long lists the Anderson/Miller algorithm is three times faster than the Miller/Reif algorithm, but

still 7 times slower than Blelloch/Reid-Miller algorithm and 35% slower than the serial algorithm. However, because it scales almost linearly, for long lists it is faster on multiple physical processors than the serial algorithm or Wyllie's algorithm.

2.2.5 Blelloch/Reid-Miller sublist algorithm

Many other work-efficient PRAM algorithms have been developed for list ranking. Most use contract-scan-expand phases and address two considerations. One is how to find vertices on which to work to keep all the processors busy and the second is how to break symmetry so that two processors are not working on neighboring vertices [5]. I break symmetry by randomly dividing up the linked list of length n into $m + 1$ sublists that can be processed independently and in parallel. I briefly describe the three phases of the algorithm below.

Phase 1 Randomly divide the list into $m + 1$ sublists. Traverse each sublist computing the "sum" of the vertex values. Form a new linked list of length $m + 1$ that links the sublists sums in the order the sublists appear in the original list.

Phase 2 Find the list scan of the reduced list found in Phase 1. These values become the scan values for the heads of the sublists.

Phase 3 Traverse each sublist computing the list scan of each vertex as the sum of the value and list scan of the previous vertex.

Phases 1 and 3 are parallel. I pick $m \leq n/\log n$ so that Phase 1 reduces the problem size by at least a factor of $\log n$. The list scan in Phase 2 can be done recursively for large m, using Wyllie's pointer jumping technique for moderate size m, or serially for small m. For small m serial list scan works best because it avoids the overhead associated with the parallel versions (see Figure 2.1). Wyllie's algorithm performs best on moderate size lists where it can take advantage of vectorization and multiprocessing and where $\lg n$ is small. For large m I use the sublist algorithm recursively, until the number of sublists becomes small enough to use either the serial or Wyllie's algorithm. I empirically determined when I should switch between algorithms.

There are two problems with the sublist algorithm that make it theoretically inferior:

- The sublists lengths vary widely, from a small fraction of the mean $\frac{n}{m}$, namely $\frac{n}{m} \ln \frac{m+1}{m+.5}$ on average, to a large factor of the mean, namely $\frac{n}{m} \ln(2m+2)$ on average. Thus, the processors' work is very imbalanced.

- Since the expected length of the longest sublist is approximately $\frac{n}{m} \ln(2m + 2)$ the parallel running time can be no better than that, i.e., $O(\frac{n}{p} + \frac{n}{m} \log m), m \leq n/\log n$. In contrast, there are many parallel algorithms that have an $O(\frac{n}{p} + \log n)$ running time.

I ameliorate both problems by requiring m to be much greater than the number of processors, p. In this way a processor is responsible for several sublists, namely $(m + 1)/p$. Periodically I perform load balancing to regroup the lists, which addresses the first problem. When $p < m/\log m$ the running time is dominated by n/p and the length of the longest sublist is not a problem.

The primary advantage of the sublist algorithm is that it is both work efficient and has small constants. The algorithm is fully vector parallel and scales almost linearly with the number of processors. We can expect, however, some degradation in performance as the number of processors increases, because the available memory bandwidth per processor decreases. Figure 2.3 shows the speedup relative to one processor for various size lists.
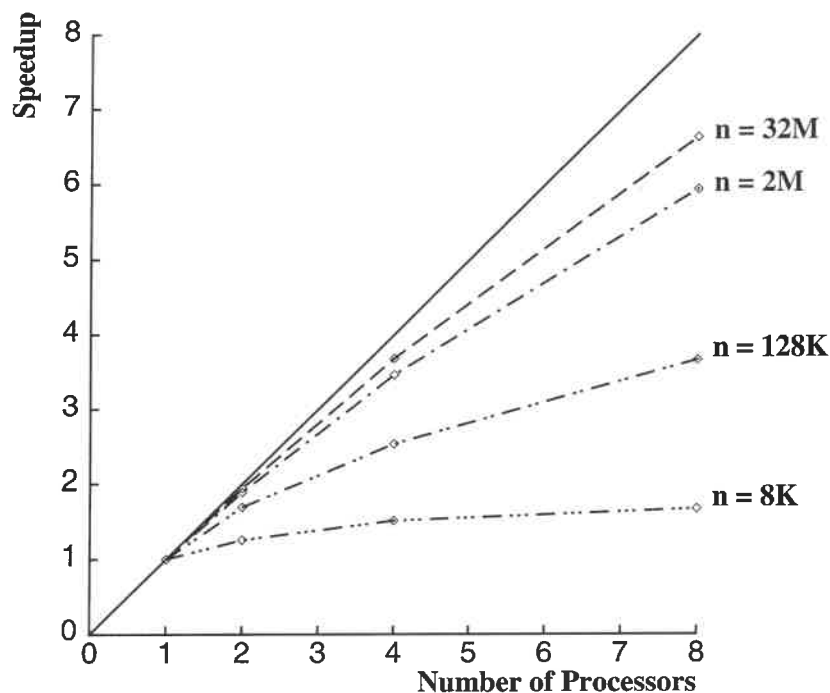
Figure 2.3: Relative speedups of Blelloch/Reid-Miller list-scan algorithm on the CRAY C90.

2.3 Blelloch/Reid-Miller Algorithm on a Vector Processor

This section describes the Blelloch/Reid-Miller list scan (for a more detailed description see the chapter appendix 2.8). In the following description I assume that there is one virtual processor for every sublist. By using strip-mining, the (vector) element processors are assigned the work of an equal number of virtual processors. I represent the linked list as a pair of arrays. The value array contains the value of each vertex of the list and the link array contains the index of the next vertex in the list. The tail of the list is a self-loop, i.e., the link at the tail is the index of the tail vertex. I used the C programming language and the standard CRAY C compiler on a CRAY C90. The time equations I give in this section are in CRAY C90 clock cycles (4.2 nsec).

2.3.1 Initialization

Each virtual processor except one designated one, P_0, picks a random vertex in the linked list to be the tail of a sublist and makes the successor vertex the head of its sublist. P_0 takes the head of the whole list as its sublist head. Then each virtual processor sets the sublist tail to a self loop and initializes its sublist sum to *zero*, where *zero* is the identity of the list-sum operator. Figure 2.4 shows the linked list that is the input of the list-scan algorithm and the result of the initialization step.

It is possible that two virtual processors pick the same random position at which to break the list. Then the two processors duplicate each other's actions and cause contention. To remove duplicate random numbers the processors can have a competition among themselves. Each processor writes its index at its random location and waits for all the processors to complete their writes. Then each processor reads the index at its random location. If the index is not its own it knows that it is a duplicate processor and can drop out of the computation.
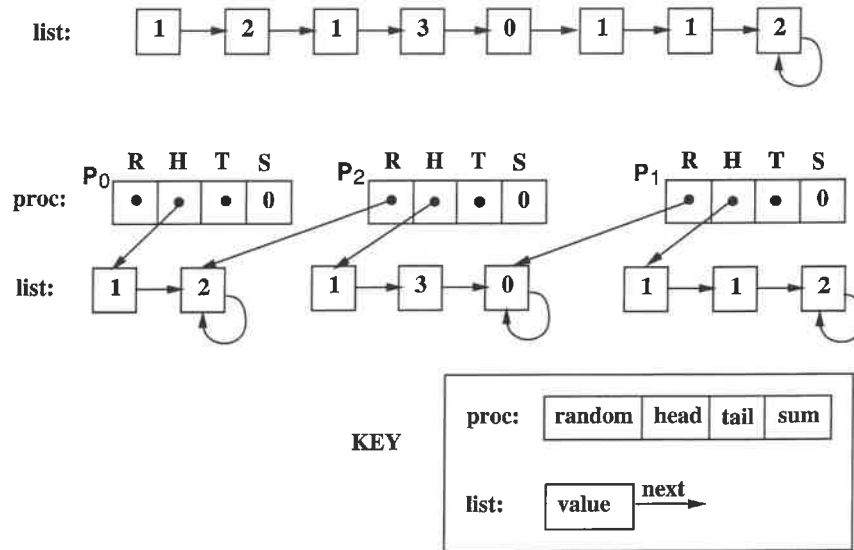
Figure 2.4: List scan initialization. The top of the figure shows the initial linked list with its values at each vertex. The bottom of the figure shows the results of the initialization step. The linked list is divided into 3 sublists, one for each processor, and each sublist is terminated with a self loop. Processor P_0 sets its sublist head, H, to the head of the whole list. Each remaining processor, P_1 and P_2, saves two values: its chosen random position, R, and the successor of the random position in the original linked list, which becomes the head of its sublist, H. Each processor also initializes its sublist sum, S, to *zero*, the identity of the scan operator.

The standard model [62] for the performance of vector operations on vectors of length n is:

$$T(n) = t_e(n + n_{1/2}),$$

where t_e is the incremental time per vector element and $n_{1/2}$ is the vector half-performance length (the vector length that achieves half the peak performance). Based on this model I found that the number of clock cycles needed for `Initialize` to set up $m + 1$ sublists is:

$$T_{\text{Initialize}}(m + 1) = 22(m + 1) + 1800.$$

2.3.2   Phase 1

Phase 1 alternates between summing the values along the links of the sublists and load balancing (vector pack). Because of the predictable sizes of the sublists lengths I can determine how many links to traverse between load balancing steps and adjust this number as the procedure progresses (see Section 2.4). Figure 2.5 shows the status after the processors have found their sublist sums.

To illustrate how streamlined the list traversal is, I show the code written in pseudo-C.
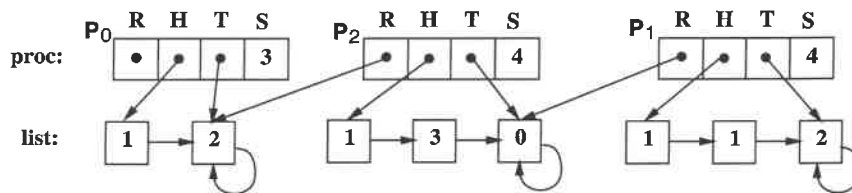
Figure 2.5: Results of computing the sum of each sublist in Phase 1. Each processor traverses its sublist until it reaches the sublist tail, T, and accumulates the "sum" of the values along the sublist, S.

```
Initial_Scan (vp, ll, n_links) {
    /* vp      — virtual processor data    */
    /* ll      — linked list               */
    /* n_links — number of links to traverse */
    for (j = 0; j < n_links; j++) {        /* Traverse n_links links of each sublist */
        for (i = 0; i < vp->n; i++) {      /* Vectorized loop                        */
            vp->sum[i] += ll->value[vp->next[i]]; /* Gather value and increment sum  */
            vp->next[i] = ll->next[vp->next[i]];  /* Gather successor link           */
        }
    }
}
```

The number of clock cycles needed for the inner loop is:

$$T_{\text{Initial\_Scan}}(x) = 3.4x + 35,$$

where x is the number of sublists remaining in the computation. I use the variable x, as opposed to $m + 1$, to indicate that the value of x changes during the course the execution of the algorithm.

Notice that all the loop does is gather the data necessary to compute the sum and store the results; there are no conditional tests or additional computation. To avoid conditional tests **Initialization** destructively set the sublist tail values to *zero*. In this way, **Initial_Scan** can repeatedly add the tail value without changing the sum. Because this loop (and the corresponding one in Phase 3) traverses every link in the linked list the time of this loop dominates the overall time and every economy is critical. For list ranking, I was able to improve the performance of the loop further by reducing the number of gather operations to one, which is important because the CRAY C90 can only perform one gather or scatter operation at a time. One gather is sufficient because I encode the link and value data for a vertex into a w-bit integer value, which I can do as long as the list length (and therefore the maximum rank) is no more than $2^{w/2}$.

To load balance, the sum and tail indices for the completed sublists are saved and the incomplete virtual processor data are packed into contiguous array locations. The number of clock cycles needed to load balance x sublists is:

$$T_{\text{Initial\_Pack}}(x) = 8.2x + 1200.$$

Once the virtual processors have reached the tails of their sublists, they create the reduced list of sublists sums. Each processor knows the tail of the previous sublist because it is the random position the processor chose during initialization. By writing the virtual processor's index into the tail of the previous sublist and then reading the index at the tail of its own sublist, the processor determines the virtual processor index of its successor's sublist. This index becomes the successor

link from its sublist sum to its successor's sublist sum and forms a new shorter linked list (see Figure 2.6). For example, consider the tail of the first sublist in Figure 2.6. Processor 2 writes **2** in the tail of the first sublist. Then Processor 0 reads the **2** at the tail of its sublist. Thus, Processor 0 links its sum to the sum at Processor 2. A processor can determine whether its sublist is the tail sublist because no processor wrote its index in the tail. Thus, the tail sublist processor, P_1 in Figure 2.6, sets its successor link to its own index.
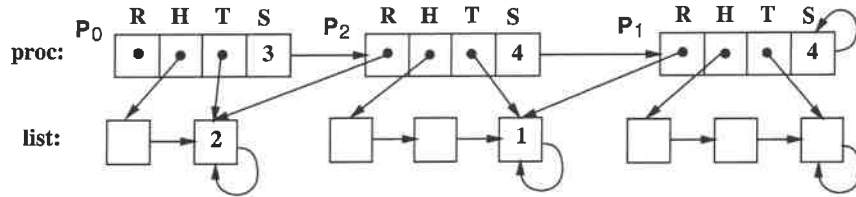
Figure 2.6: Creating the reduced list of sublist sums during Phase 1. Each processor writes its index at its random position in the linked list, R, and reads the index written at the tail of its sublist, T. This index is the index of the processor with the successor sublist. P1 finds no index at the tail of its sublist and therefore is the tail sublist.

The number of clock cycles needed to create the reduced list of length $m + 1$ is:

$$T_{\text{Find\_Sublist\_List}}(m + 1) = 11(m + 1) + 650.$$

2.3.3   Phase 2

Depending on the size of this new linked list, the algorithm finds the list scan of the reduced linked list recursively, using Wyllie's algorithm, or serially. Note that in this phase both the list ranking and list scan algorithms find the list scan and not the sum as in Phase 1. Figure 2.7 shows the result of Phase 2 of the algorithm.
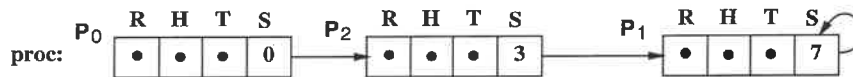
Figure 2.7: List scan on the reduced list of sublist sums after Phase 2.

2.3.4   Phase 3

The scan value found in Phase 2 becomes the scan value for the head of a virtual processor's sublist, see Figure 2.8.
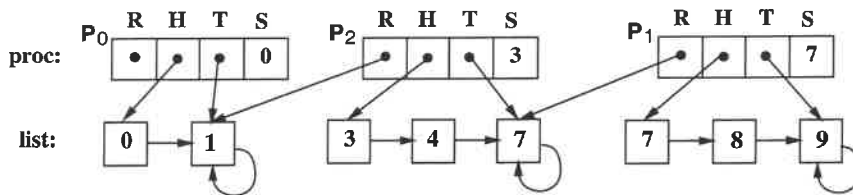
Figure 2.8: The final scan values after Phase 3. The scan values for the heads of the sublists are the scan values of the reduced list found in Phase 2. Phase 3 finds the remaining scan values.

As in Phase 1 each virtual processor alternates between traversing its sublist and load balancing. But this time it finds the scan of each vertex as the sum of the previous vertex scan and value. The number of clock cycles needed to traverse one link of the x sublists remaining in the computation is:

$$T_{\text{Final\_Scan}}(x) = 4.6x + 28,$$

and to load balance the x sublists is:

$$T_{\text{Final\_Pack}}(x) = 7.2x + 950.$$

2.3.5 Restoration

Finally, each virtual processor reconnects the sublists to form the original linked list, using the values saved during initialization. The number of clock cycles required to reconnect $m + 1$ sublists is:

$$T_{\text{Restore\_List}}(m + 1) = 4.2(m + 1) + 300.$$

2.4 Analysis of the Algorithm

In Phases 1 and 3 the algorithm periodically performs load balancing to removed completed sublists from the computation. If it load balances too frequently it removes none or only a few sublists, and when there are many sublists load balancing is expensive. If it does not load balance often enough it may have many processors performing needless work, repeatedly chasing completed sublists' tails. To determine when are good times to load balance we first need a better understanding of what the expected distribution of the sublists lengths is. I derive the expected distribution in Section 2.4.1. In Section 2.4.2 I next determine the overall cost of performing the algorithm, given the timing data in Section 2.3. In Section 2.4.3 I used the expected distribution to minimize the execution time of the algorithm, given the number of sublists and number of times to load balance. Finally, I explain how I compute the parameters values to minimize the overall execution time. The main theorem is:

Theorem 2.1 *The list-ranking algorithm has expected time $O(\frac{n}{p} + \frac{n}{m} \log m)$ on p processors, when $m < n/\log n$.*

2.4.1 Analysis of sublist lengths

In this section I show that the distribution of the lengths of the sublists is approximately a negative exponential distribution, when n and m are large. First consider the following situation. Let $X_1, \ldots, X_m$ be m random numbers in the range (0,1). For truly random numbers $\text{Prob}(X_j = X_k) = 0$ for $j \neq k$. Therefore, the numbers partition (0,1) into $m+1$ subintervals. Let $X_{(1)}, \ldots X_{(m)}$ denote the Xs ordered by their sizes from smallest to largest.

Proposition 2.2 (Feller [45]) *If $X_1, \ldots, X_m$ are independent and uniformly distributed over the range (0,1) then as $m \to \infty$ the successive intervals in our partition behave as though they are mutually independent exponentially distributed variables with $\mathbf{E}[X_{(j+1)} - X_{(j)}] = \frac{1}{m}$.*

In our case, we are choosing m random positions in a list of length n. If $n \gg m$, $n \to \infty$, and $m \to \infty$ then the lengths of the sublists tend to behave as mutually independent exponential variates with expectation $\frac{n}{m}$. That is, if L is a sublist length, then

$$\mathrm{Prob}\{L > x\} \approx e^{-mx/n} = a(x). \tag{2.1}$$

We can estimate the expected length of the jth shortest sublist by setting $a(x) = (m-j+.5)/(m+1)$ and solving for x. This estimate seems to be reasonable for n and m as small as $n > 1000$ and $m > 100$. The expected length of the shortest sublist is:

$$\mathbf{E}[L_{(0)}] \approx \frac{n}{m} \ln \left(\frac{m+1}{m+.5} \right),$$

and the longest sublist is:

$$\mathbf{E}[L_{(m)}] \approx \frac{n}{m} \ln(2m+2).$$

Figure 2.9 shows the expected length of the jth shortest sublist for several values of m when $n = 10,000$ and compares it to some actual trial data averaged over 20 samples. Notice that as m increases the expected length of the longest sublist decreases and there is less variation in list lengths. Therefore, to reduce the parallel running time we want to make m large. However, as m increases so do the costs due to load balancing, initialization, and Phase 2.
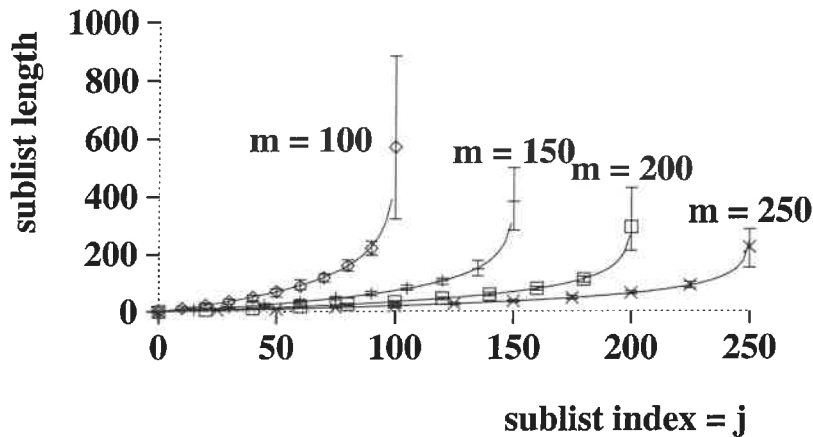
Figure 2.9:   The function curves are the expected length of the jth shortest sublist when $n = 10000$ for several values of m, the number of sublists. The observed lengths were found by taking 20 samples of dividing a list of size $n = 10000$ into m sublists randomly. The error bars show the minimum, average, and maximum lengths of the jth shortest sublists of the 20 samples.

2.4.2   Cost of the algorithm

Using the timing equations of each piece of the algorithm given in Section 2.3 we can determine the cost of the complete algorithm, assuming we know the exact lengths of the sublists and when to perform load balancing. Let S_i be the total number of links traversed in each list before the load balancing for the ith time. Let $g(x)$ be the expected number of sublists that have length greater than x. From Eq. (2.1) we get

$$\begin{aligned} g(x) &= (m+1) \times Prob(L > x) \\ &\approx (m+1)e^{-mx/n}. \end{aligned} \tag{2.2}$$

The dotted line in Figure 2.10 shows $g(x)$ when $n = 10000$ and $m = 200$. The x-axis is the sublist length and the y-axis is the number of sublists with that length. You can think of each sublist as being laid out from left to right and placed one above the other from longest to smallest, each starting at $x = 0$. That is, the y-axis is the number of sublists that are still active in the computation, namely the vector lengths of the computations, while the x-axis is the number of links traversed in each list. As we proceed from left to right, we are traversing links on a vector of length equal to the height of the step function. Every time we load balance (at the corner of a step) the vector length decreases. The area under the step function in Figure 2.10 is the expected total number of links traversed in either Phase 1 or Phase 3. If $S_i = i$ then the area under the step function would be n, the area under the curve $g(x)$. Our aim is to minimize the area under the step function that is above the dotted line, while keeping the cost of load balancing down. The cost of load balancing is proportional to the sum of the heights of the steps in the step function.
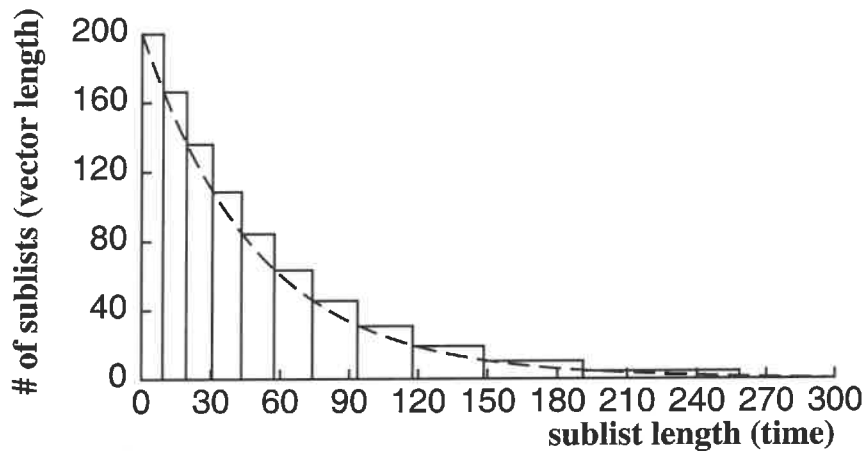
Figure 2.10: The expected number of active sublists. The dotted function is $g(x)$ the expected number of sublists that have length greater than x, where $n = 10000$ and $m = 199$. If we load balance 11 times the expected execution time on the CRAY C90 is minimized by load balancing at the vertical lines. The step function shows the expected number of sublists that are currently active at every link traversal. The drop in step size is the expected number of sublists that completed since the last time load balancing was done.

From the equations for each part of the algorithm given in Section 2.3, the expected number of CRAY C90 cycles for Phases 1 and 3 of the algorithm is:

$$T_{P1+P3} = \left(\sum_{i=0}^{l-1} (S_{i+1} - S_i)(a \cdot g(S_i) + b) \right) + \left(\sum_{i=0}^{l-1} (c \cdot g(S_i) + d) \right) + e(m+1) + f, \qquad (2.3)$$

where $T_{\text{Scan}}(x) = ax + b$, $T_{\text{Pack}}(x) = cx + d$, and $T_{\text{Other}}(x) = ex + f$, and $g(S_i)$ is the expected number of sublists remaining after traversing S_i links in each list. The first summation is the total time to traverse the links in Phases 1 and 3, the second summation is the time for load balancing in Phases 1 and 3, and the final $e(m+1) + f$ is the time for creating the sublists, forming the reduced linked lists from the sublist sums, and restoring the linked list to its original form after the final phase.

2.4.3   Minimizing the time given fixed parameters

Consider n, m, and l fixed. We want to minimize the execution time with respect to $S_0, S_1, S_2, \ldots, S_l$, where $S_0 = 0$ and $S_i, i > 0$, is the number of links traversed on each sublist before load balancing

for the ith time. We can minimize Eq. (2.3) by taking partial derivatives for each S_i and setting the partial to zero to obtain a set of l simultaneous equations:

$$
\begin{aligned}
S_{i+1} &= S_i - \frac{g(S_{i-1}) - g(S_i)}{g'(S_i)} - \frac{c}{a} \\
&= S_i + \frac{g(S_{i-1}) - g(S_i)}{\frac{m}{n}g(S_i)} - \frac{c}{a},
\end{aligned}
\tag{2.4}
$$

using $g'(S_i) = -\frac{m}{n}g(S_i)$. That is, if we know S_{i-1} and S_i then we can determine S_{i+1}. Since we know $S_0 = 0$ and if we know S_1 we can compute $S_2, \ldots, S_l$ iteratively.

Notice in Figure 2.10 that the S_i's become increasingly further apart for larger i's because the rate sublists complete slows down over time. The factor c/a in Eq. (2.4) reflects the relative cost of load balancing and traversing links. If we increase c and keep the number of times we load balance constant, load balancing would occur less frequently during the initial iterations and occur more frequently during later iterations. Initially load balancing is expensive because the vector lengths are long and becomes less expensive as vector lengths shorten. As we increase c relative to a eventually we find that the execution time is reduced by decreasing the number of times we load balance even though we increase the total number links we traverse. In the next section we consider how to determine the best value for S_1 which in turn determines the number times to load balance.

We can simplify Eq. (2.3) by using Eq. (2.4) to substitute for $S_{i+1} - S_i$. That is,

$$
\begin{aligned}
&\sum_{i=0}^{l-1}(S_{i+1} - S_i)(a \cdot g(S_i) + b) \\
&= aS_1 g(S_0) + a\sum_{i=1}^{l-1}(S_{i+1} - S_i)g(S_i) + b\sum_{i=0}^{l-1}(S_{i+1} - S_i) \\
&= aS_1(m+1) + a\sum_{i=1}^{l-1}[\frac{n}{m}(g(S_{i-1}) - g(S_i)) - \frac{c}{a}g(S_i)] + bS_l \\
&\approx aS_1(m+1) + an - c\sum_{i=1}^{l-1}g(S_i) + bS_l.
\end{aligned}
$$

Thus, the expected time for Phases 1 and 3 is:

$$
T_{P1+P3}(S_1, \ldots, S_l) \approx an + b\frac{n}{m}\ln m + (aS_1 + c + e)(m+1) + ld + f,
$$

since $\mathbf{E}[S_l] \approx \frac{n}{m}\ln m$.

Because the time for the second phase is no worse than the serial time (44 cycles/vertex) and given the timing data in Section 2.3, the expected number of clock cycles on a one processor CRAY C90 for the algorithm is:

$$
T(n) \approx 8n + 62\frac{n}{m}\ln m + (8S_1 + 96)(m+1) + 2150l + 2750.
\tag{2.5}
$$

where $m + 1$ is the number of sublists, S_1 is the number of links traversed before load balancing the first time, l is the number of times load balancing is done and is a function of S_1, m, and n.

2.4.4 Overall vector performance

From the previous discussion we have a way to determine when we should load balance, as long as we know the length of the whole linked list, n, the number of sublists, m, and the number of links traversed before load balancing the first time, S_1. However, all we know is the length of the whole linked list, namely n. We need to find good choices for m and S_1. My approach was to estimate the running time of the algorithm using Eq. (2.3) for various values of m, S_1 and n, and use Eq. (2.4) to determine the corresponding $S_2, \ldots, S_l$ values. Then, for each value of n I found values of m and S_1 that minimized the running time within about two percent. Finally, I fitted functions to m *vs.* n and S_1 *vs.* n. It appears that m and S_1 are approximately cubic polynomials of $\log n$. I used these fitted polylog functions in my implementation to determine m and S_1 given n, and used Eq. (2.4) to find successive values of S_i. When I used these estimates of m and S_1, I found that Eq. (2.3) accurately predicts and Eq. (2.5) overestimates the actual execution time on one CRAY C90 vector processor.

2.5 Vector Multiprocessor List Scan

To extend the algorithm to multiple vector processors I divided the virtual processors equally among the physical processors and let vectorization proceed on the data assigned to the physical processors. The CRAY C compiler makes parallelizing relatively easy. I modified loops to be tasked loops with compiler directives that direct the compiler to divide the loops into equal size chunks, one chunk per processor, and to vectorize the chunks within the processors. Ignoring the time for synchronization, from Eq. (2.3) the expected vector-parallel time for the algorithm is:

$$
\begin{aligned}
T(n) &\approx \sum_{i=0}^{l-1}(S_{i+1} - S_i)(ag(S_i)/p + b) + \sum_{i=0}^{l-1}(cg(S_i)/p + d) + e(m+1)/p + f + T_{P2}(m+1) \\
&\approx O\left(\frac{n}{p} + \frac{n}{m}\log m\right), \qquad \text{if } m < n/\log n.
\end{aligned} \tag{2.6}
$$

I derived Eq. (2.6) using an analysis similar to that used in the previous section and that the time for Wyllie's algorithm in Phase 2 is $T_{P2} \leq O(n/p + \log n)$ if $m < n/\log n$. From Eq. (2.6) we get Theorem 2.1.

Data parallel algorithms assume that each data parallel step is synchronized, whether or not it is necessary. However, the most the processors need to synchronize is after every innermost vectorized loop. If we treat the vector multiprocessor strictly as an $l \times p$ multiprocessor then, in particular, the processors need to synchronize each time Phases 1 and 3 load balance, so that load balancing can proceed globally across the physical processors. Instead, the implementation assigns virtual processors to physical processors once at the beginning and only load balances locally within each physical processor. In this way each processor completes all of Phase 1 and Phase 3 independently of the other processors. In effect, the processors synchronize only a *constant* number of times and do no load balancing across processors. Eliminating synchronization avoids needless delays at each synchronization point. Avoiding global load balancing across processors is important because most compilers do not know how to parallelize a pack operation across processors and global load balancing requires extra communication.

Because the algorithm uses randomization, significant load imbalance is unlikely when the processors load balance only locally. Even if an imbalance should become a problem as the procedure progresses, only one across-processor load balance should be necessary. My results are excellent without any global load balancing.

Unfortunately, to tune the parameters m and S_1 we need to minimize for every possible number of processors. For a highly or massively parallel machine, tuning the parameters for every number of processors would not be practical. But since the CRAY C90 has 16 processors there are only 16 sets of equations. I tuned the parameters for 1, 2, 4, and 8 processors, resulting in the list-scan asymptotic performance of 7.4, 3.9, 2.0, and 1.1 clock cycles per vertex, respectively, where a clock cycle is 4.2 nsec. The asymptotic times for list ranking are 5.1, 2.6, 1.4, and .75 clock cycles per vertex, respectively. Figure 2.11 shows a graph of the list scan execution times in nanoseconds.
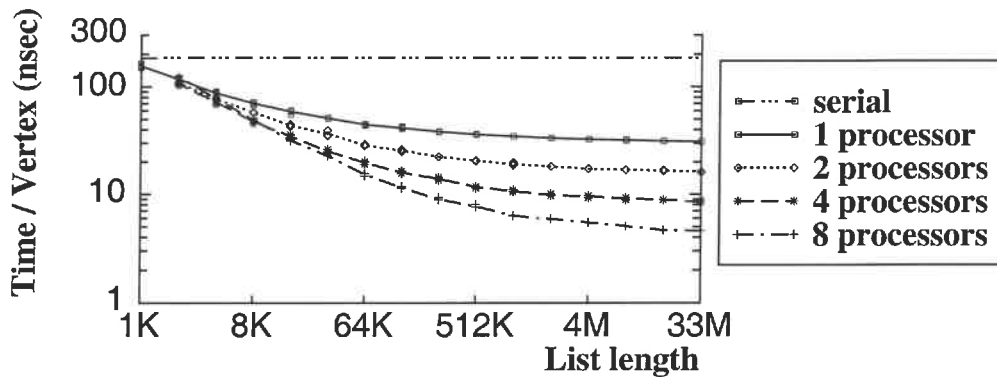
Figure 2.11:   Time per vertex for list scan on 1, 2, 4, and 8 processors of the CRAY C90.

2.6   Other work-efficient list-ranking algorithms

On one CRAY C90 vector processor the sublist algorithm takes about 7.4 clock cycles per vertex asymptotically to find the scan of a linked list. If other algorithms are to be competitive, they must be able to use no more than 7.4 cycles per vertex on average. Below I discuss some other deterministic algorithms that have been described in the literature. Except for Wyllie's pointer-jumping algorithm on short linked lists I conclude that other algorithms are unlikely to be competitive.

Cole and Viskin devised a parallel deterministic coin-tossing technique [38] that they used to develop an optimal deterministic parallel list-ranking algorithm [39, 9]. This algorithm breaks the linked list into sublists of two or three vertices long (the heads of the sublists are called 2-ruling sets); reduces the sublists to a single vertices; and then compacts these single vertices into contiguous memory to create a new linked list. It recursively applies the algorithm to the new linked list until the resulting linked list is less than $n/\log n$, at which point it applies Wyllie's algorithm. In the final phase it reconstructs the linked list by unraveling the recursion in the first phase and filling in the rank values of the removed vertices. The algorithm runs in $O(\log n \log \log n)$ parallel time and uses $O(n)$ steps. Later they modified their algorithm to give a $O(\log n)$ time optimal deterministic algorithm [41, 111]. The problem is algorithms for finding 2-ruling sets that give either of these time bounds are quite complicated and have large constants.

Cole and Viskin also developed the first $O(\log n)$ optimal deterministic list-ranking algorithm based on assigning processors to jobs and using expander graphs [40]. Its constants are far too large to be practical. In addition, they give a much simpler 2-ruling set algorithm that is not work efficient but has smaller constants (see [9]). Because it is not work efficient and its constants are larger than Wyllie's or ours, I chose not to implement it.

Anderson and Miller combined their randomized algorithm with the Cole/Viskin deterministic coin tossing to get an optimal $O(\log n)$ time deterministic list-ranking algorithm [5]. As with their

randomized algorithm, the processors are assigned $\log n$ vertices to process. On each round each processor executes a case statement that either breaks symmetry, splices out a vertex in its queue, or splices out a vertex at another processor's queue. To break symmetry it finds a $\log \log n$-ruling set. Finding $\log \log n$-ruling sets are much simpler ($O(1)$ time) than finding 2-ruling sets ($O(\log n)$ time with large constants). But because each round involves a nonparallel three way case statement, where each case needs to be completed by all the processors before the next case can be executed, its constants are also much larger than ours.

The problem with these work-efficient algorithms is that on each of the $O(\log n)$ rounds, they attempt to to find a large number of, at least $n/\log n$ and as many as $n/2$, nonadjacent elements in the linked list to maximize the parallelism in the algorithm. In contrast, the sublist algorithm has only few symmetry breaking rounds, and finds only a moderate number of such elements each round. Thus, the algorithm greatly reduces the overhead associated with symmetry breaking.

2.7   Conclusions and Future Directions

In this chapter I described a new parallel algorithm for list ranking and list scan and its implementation. The implementation is surprisingly fast, especially when compared with the limited speedups of other implementations of pointer-based algorithms. The key to the success is twofold. First, I was willing to sacrifice fast asymptotic depth for reduced parallel work, only a small constant factor greater than the serial work. By reducing the constants in the work, my algorithm performs better than other work-efficient algorithms, which have larger constants. Second, I used hardware with the highest-performance memory system available, and programmed it with virtual processing to hide latency. Because list ranking is so memory bound, its performance is directly related to the bandwidth of the memory system. Even though vector supercomputers, such as the CRAY C90 are becoming less common, there is evidence that multiprocessor systems are moving to higher bandwidths and reduced overhead. Both small constants and high bandwidth are necessary for good performance. High bandwidth is not sufficient because the algorithms with even moderate-size constants are not much better than the serial implementation as shown in Figure 2.1. Small constants are not sufficient because reduced bandwidths results in longer parallel times, as indicated in [82] and seen in the reduced speedup of my algorithm for larger numbers of processors (see Figure 2.3).

As with any implementation, there are a multitude of possible modifications and enhancements that could improve its performance. A large part of the performance loss is due to short vector lengths. As sublists drop out of the computation the vector lengths shorten. Not only are the vector lengths short, the number of iterations remaining with short vector lengths can be relatively large, since the longest sublists can be much longer that the other sublists. Short vectors are inefficient because of the latency due to filling the vector pipelines. John Reif suggested I use oversampling to further subdivide the remaining long sublists when the vector lengths become short. The cost, however, of maintaining which subdivisions remain relevant would slow down the two major list-scan loops of the algorithm and likely slow down the overall performance.

Finally, the question still remains whether having a fast list-ranking implementation helps in making other pointer-based applications practical. If so, I may have opened up major classes of PRAM algorithms that can have reasonable implementations. In addition, it also would be interesting to see whether my analysis and techniques can be applied to other pointer-based algorithms.

2.8   Vector Implementation of List Scan

I implemented the list scan algorithm on a CRAY C90, a vector multiprocessor. Vector multiprocessor machines consist of multiple scalar processors, each augmented with a bank of vector registers, pipelined functional units, and vector instructions. The functional units divide their operation into several stages so that the clock speeds can be increased. On every clock cycle another element from the vector register enters the pipeline of the functional unit while one results exits the pipeline. The delay between the time the first operands enters the pipeline until the first result leaves the pipeline is call the *latency* or *start up* time of the functional unit. If the functional units are fully pipelined, they can accept new operands every clock cycle. Multiple functional units can process data simultaneously, and if the hardware permits, the results of one functional unit can be *chained* directly to the input of another unit. Thus, to execute operands of length n on a fully pipelined functional unit with start up time s takes $s+n$ clock cycles, where $n \leq v$, the vector register length. To hide the latency s we want n as close to v as possible.

The vector multiprocessors are typically connected through a multistage interconnection network to a common memory. Memory is composed of multiple memory banks that can access different addresses in parallel using a single global address space. Once a memory bank has been accessed it cannot be access again until there is a delay, called the *cycle time*. Usually the memory subsystem is optimized for accessing sequential memory addresses. That is, banks are fully interleaved so that successive addresses are on successive memory banks and the number of memory banks is greater than the cycle time. In this way, sequential memory addresses can be accessed one element per clock cycle. *Load* operations load data to vector registers from memory and and *store* operations store data to memory from vector registers. These operations use consecutive (stride = 1) memory locations or use every k^{th} element (stride = k) of memory. Bad choices for k can result in accesses to the the same memory bank at a rate higher than the cycle time and *memory-bank conflicts* occur, causing memory stalls. Memory can also be accessed at arbitrary locations using an index vector. Often such loads are called *gather* operations and such stores are called *scatter* operations. Because of memory-bank conflicts, each element during a gather or scatter typically is accessed at a rate lower than the machine clock cycle (about 2 clock cycles/element for random access patterns on the CRAY Y-MP machines). In addition to cycle-time delays, there are *access-latency* time delays. The latency for a load is the time to get the first word from memory to the register and often is much greater than the latency of the functional units. However, for CRAY Y-MP machines-memory access latencies are about the same, but are getting longer as the machines get bigger. The CRAY Y-MP machines have two read ports and one write port to memory. Thus, they can perform two vector load operations and one vector store operation concurrently. The scatter/gather hardware, however, can service only one scatter or gather operation at a time. But a gather operation can run concurrently with one load and one store, and a scatter operation can run concurrently with two loads.

To implement a PRAM algorithm on vector multiprocessor, I treat the ith element in the vector registers as the data for i^{th} *element processor* of a SIMD machine. Recall, I call the vector element an element processor to distinguish it from a full vector processor. Any data parallel algorithm can be vectorized and parallelized by having an element processor do the work of a virtual processor in the algorithm. For example, on a CRAY C90 each processor has a vector-register length of 128 and there are 16 processors. Therefore, it has 2048 element processors. However, by using strip mining [91] or loop raking [117, 22], a single element processor is assigned the work of several virtual processors.

I used the C programming language and the standard CRAY C compiler on a CRAY C90.

Because many of the vector operations use indirect addressing, I needed to give compiler directives to get the compiler to vectorize the loops; the only portion that is not vectorizable is the serial list scan in Phase 2. I attempted to reorder the statements within vectorized loops to fill the multiple functional units for concurrent operations, to avoid contention between input/output memory ports and the gather/scatter hardware, and to avoid write after read dependencies [55]. With nested loops I unrolled the inner loop up to eight times to do eight iterations of the outer loop to avoid unnecessary loading and storing of vector registers on each execution of the inner loop. Chaining is also possible within loops. I made no attempt to avoid memory bank conflicts. However, since the algorithm chooses random positions for the heads of the sublists, systematic memory-bank conflicts are unlikely.

Below I give pseudo-C code to illustrate my implementation of the single vector-processor version of my list ranking algorithm. I use four structures containing sets of vectors to simplify the presentation. However, in the actual implementation I use individual vectors. For each subroutine I discuss the vectorization and present the C-pseudo code.

List_Scan The pointer `ll` points to a linked-list structure that contains a pair of arrays, where one array `ll->next` gives the indices of the successive nodes in the linked list and the other array `ll->value` gives the values of the nodes. The scalar `ll->head` is the index of the head of the linked list and the scalar `ll->n` is the length of the linked list. The linked list terminates with a self loop. The resulting list scan is stored in the array `ll_sum`.

The pointer `vp` points to a structure that maintains the local data for the currently active virtual processors. Periodically, the algorithm packs the arrays as processors drop out. The array `vp->next` provides the current next indices for the sublists, the array `vp->sum` maintains the current sums of the sublists, and array `vp->id` contains the virtual-processor identifiers. The scalar `vp->n` is the number of currently active virtual processors.

The result of Phase 1 is a reduced linked list, to which `rl` points. It has the same fields as `ll`.

Finally, the pointer `sl` points to a structure that saves information about the sublists. The random indices, which are the tails of the previous sublists, are in `sl->random`, the values of these tails are in `sl->value`, and the successor links at these tails, namely the heads of the sublists, are in `sl->head`. When a virtual processor reaches the tail of its sublist it saves the index of the tail in the array `sl->tail` and the sum of the sublist in reduced linked list value array `rl->value`.

`List_Scan`, shown below, starts by calling `Initialize`, which sets up the sublists and returns the number of sublists it creates. In Phase 1 `List_Scan` alternates between traversing each sublist and packing out completed sublists until no sublists remains. `Initial_Scan` traverses `s[l]` links of each sublist, summing the values at each node (in Section 2.4.3 I discussed how to determine the values of `s[l]`). Then `Initial_Pack` load balances the remaining lists by removing the completed sublists from `vp`. It saves the results of the completed sublists in `sl` and removes them from `vp` by packing the remaining sublists to the initial portion of the arrays. By packing the arrays it effectively reassign virtual processors to element processors. Finally, `Initial_Pack` returns the number of incomplete sublists remaining. After all the sublists have completed, `Find_Sublist_List` forms a linked list, `rl->next`, of the sublist sums, `rl->value`. In Phase 2 it finds the list scan of this linked list by calling either `List_Scan` recursively, the vector multiprocessor routine `Wyllie`, or the serial routine `Serial_List_Scan`. It puts results in the virtual processor array `vp->sum`. Phase 3 is like Phase 1 and proceeds by alternating between traversing the sublists for `s[l]` nodes and packing out finished sublists until no sublists remain. Finally, `Restore_List` puts back the original links and values at the tails of the sublists. All the routines are vectorized except `Serial_List_Scan`.

```
List_Scan(ll_sum, ll)
{
   /* Phase_1 */
   l = 0;                                          /* Initialize the pack counter       */
   vp->n = rl->n = Initialize(vp, sl, ll);         /* Initialize the virtual processors */
   rl->head = 0;                                   /* Head of rl's list is index 0      */
   while (vp->n > 0) {                             /* Find the sublist sums             */
      Initial_Scan(vp, ll, s[l++]);
      vp->n = Initial_Pack(vp, sl, ll, rl);
   }
   Find_Sublist_List(rl, ll, sl);                  /* Links the sums into a linked list */

   /* Phase_2 */
   if (sl->n > wyllie_cutoff)
      List_Scan(vp->sum, rl);
   elseif (sl->n > serial_cutoff)
      Wyllie(vp->sum, rl);
   else
      Serial_List_Scan(vp->sum, rl);

   /* Phase_3 */
   l = 0;                                          /* Reset the pack counter            */
   vp->n = sl->n;                                  /* Reset number of virtual processors*/
   while (vp->n > 0) {                             /* Find the scan of the sublists     */
      Final_Scan(ll_sum, vp, ll, s[l++]);
      vp->n = Final_Pack(ll_sum, vp, sl, ll);
   }
   Restore_List(sl, ll)                            /* Restore values at sublist tails   */
}
```

Initialize: To avoid having to check whether a processor has reached the end of a sublist at every pointer dereference, at the tail of each sublist `Initial_Scan` destructively sets `ll->next` to its own index to create a self loop and sets `ll->value` to *zero*, where *zero* is the identity value of the scan operator. In this way, it can repeatedly add the tail value to the sublist sum without affecting the sum.

Initialization starts by finding m, the appropriate number of sublists to use given the length of the linked list. In Section 2.4.3 I discuss how to determine what is an appropriate value for m. `Gen_Tails` finds m pseudo-random positions in the linked list, `sl->random`, which are to be the tails of the sublists. It also ensures that none of these positions are the tail of the whole list. To simplify the implementation I chose to use equally spaced positions and assumed that the linked lists are randomly ordered. If the ordering of the links is random then we can expect sublist lengths to follow the same distribution as when the heads of the lists are chosen randomly. Next `Initialize` saves the links and values at the tails. The head of the first sublist is the head of the whole linked list. Because the links are retrieved from random positions, to retrieve `ll->next` requires loading `sl->random` and a gathering `ll->next` using `sl->random`, and to save `sl->head` requires a store. Then `Initialize` gathers `ll->value` at `sl->random` and stores them in `sl->value`.

Next `Initialize` turns the linked list into a set of sublists by setting the sublist tails to self

loops and tail values to *zero*. As the tails are at random positions these assignments require two scatter operations. In Phase 1 it need not worry about the value of the tail of the whole list because the algorithm does not use tail sublist sum when finding the scan in Phase 2. Finally `Initialize` initializes the virtual processor vectors: it stores the heads, stores *zero*'s at the sums, and stores the processor identifiers (index).

```
Initialize(vp, sl, ll)
{
    m = Compute_Num_Sublists(ll->n);
    Gen_Tails (sl->random, m, ll->n);           /* Find random positions        */
    sl->head[0] = ll->head;                      /* Set head of first sublist    */

    for (i=1; i < sl->n; i++) {                  /* Save tails of sublists       */
        sl->head[i] = ll->next[sl->random[i]];       /* Gather heads and save        */
        sl->value[i] = ll->value[sl->random[i]];      /* Gather values at tails and save */
                                                 /* Set up sublists              */
        ll->next[sl->random[i]] = sl->random[i];      /* Scatter self loops at tails  */
        ll->value[sl->random[i]] = ZERO;             /* Scatter zero at tails values */
    }
    for (i = 0; i < sl->n; i++) {                /* Initialize virtual processors */
        vp->next[i] = sl->head[i];                   /* Store heads                  */
        vp->sum[i] = ZERO;                           /* Initialize sums              */
        vp->id[i] = i;                               /* Assign processor id's        */
    }
    return m;
}
```

Initial_Scan: `Initial_Scan` traverses each sublist for `n_steps` times computing the sum of the links. Because the weight of the tail is *zero*, it can repeatedly accumulate the sum at the tail without affecting the sum. Each traversal of the array of links requires retrieving the values and links at arbitrary locations in `ll`. Thus, it uses two gather operations. To increment the sum requires loading, adding to, and storing `vp->sum`. Finally it needs to store the current link `vp->next`.

```
Initial_Scan (vp, ll, n_steps)
{
    for (j = 0; j < n_steps; j++) {              /* Traverse n_steps links       */
        for (i = 0; i < vp->n; i++) {
            vp->sum[i] +=ll->value[vp->next[i]];     /* Gather value and add to sum  */
            vp->next[i] = ll->next[vp->next[i]];     /* Gather successor link        */
        }
    }
}
```

The CRAY only allows strip mining on an inner loop; the vector registers first hold the first v elements of the arrays, then the next v elements, and so on, where v is the vector register length. It is more efficient, however, to do strip mining of the inner loop outside the outer loop. To get that effect, the implementation unrolls the loop eight times so that it traverses eight links of each sublist before going on to the next set of virtual processors. In this way, the CRAY avoids the loads and stores of `vp` and keeps intermediate results in vector registers for each set of eight iterations.

Initial_Pack: After traversing the sublists s[1] steps List_Scan packs out any completed list. Packing requires saving the results of the completed lists according to their processor identifiers and then packing the remaining lists in vp so that they are contiguous. A sublist is complete if the virtual processor has reached the tail of its sublist, which is a self loop. To test for a self loop requires loading vp->next, gathering ll->next at vp->next, and testing whether the two are equal. There are two ways I could get the compiler to save the completed lists. One is to compute the indices of the completed lists and then using these indices to gather vp->sum, vp->next, and vp->id. Then it can scatter vp->sum to rl->value and vp->next to sl->tail at the indices in vp->id. The other way is to load vp->sum, vp->next, and vp->id, and change them so that all active sublists use one of the completed sublist values by using a vector merge operation. Then, as before, it can scatter vp->sum to rl->value and vp->next to sl->tail at the indices in vp->id. The effect is to have all active sublists write to the same location in rl->value and sl->tail. The latter approach causes much memory contention because most of the sublists are active. Clearly, the former approach is better and is what I used.

```
Initial_Pack(vp, sl, ll, rl)
{
  j = 0;
  for (i = 0; i < vp->n; i++) {
    if (vp->next[i] == ll->next[vp->next[i]]) { /* Save completed sublists      */
      rl->value[vp->id[i]] = vp->sum[i];            /* Gather and scatter sum       */
      sl->tail[vp->id[i]] = vp->next[i];            /* Gather and scatter tail      */
    } else {                                   /* Pack remaining sublists      */
      vp->id[j] = vp->id[i];                         /* Gather and store processor ids */
      vp->sum[j] = vp->sum[i];                       /* Gather and store current sum   */
      vp->next[j++] = vp->next[i];                   /* Gather and store current link  */
    }
  }
  return j;                                    /* Number of remaining sublists  */
}
```

It packs the remaining active sublists by computing the indices of the active sublists and for each array, vp->next, vp->sum and vp->id, gathering the data using the active indices and then storing the data contiguously.

Find_Sublist_List: At this point each sublist has reached its tail and is ready to start Phase 2. Recall that sl->tail holds indices for the sublist tails, while sl->random holds indices for the tails of the previous sublists. Therefore, when Find_Sublist_List writes the sublist index to ll->next at sl->random, then it is writing the index of the successor sublist to the tails of the sublists. (The implementation writes to ll->next because it can easily regenerate the self loops there.) This write requires loading sl->random and then scattering the index to ll->next at sl->random. Note that sl->random does not contain the index of the tail of one sublist, namely the tail of the whole list. Therefore, if it writes negative indices it can distinguish between values set at sl->random and the original self loops in ll->next. Next Find_Sublist_List gathers these indices from ll->next, but this time using sl->tail. These indices are the indices of the successor sublists as long as they are negative. Only one index is positive and it is the index of the tail of the whole list. Notice that it does the writing and reading of the indices in separate loops because the reading of the indices

may not be in the same order as the writing. That is, it needs to be sure that the write is complete before the read starts and there is no chaining.

```
Find_Sublist_List(rl, ll, sl)
{
    for (i=1; i<rl->n; i++)
        ll->next[sl->random[i]] = -i;          /* Scatter index of next sublist   */
    for (i=0; i < sl->n; i++) {                 /* Create list of sublists         */
        next = ll->next[sl->tail[i]];           /* Gather index of next sublist    */
        rl->next[i] = -next;                    /* Store the index                 */
        if (next > 0){                          /* Found tail of ll                */
            rl->next[i] = i;                    /* Set rl tail to self loop        */
            sl->random[0] = next;               /* Save tail index of ll           */
            sl->value[0] = ll->value[next];     /* Save tail value                 */
            ll->value[next] = ZERO;             /* Set tail value to the identity  */
        }
    }
    for (i=0; i<sl->n; i++) {
        ll->next[sl->tail[i]] = sl->tail[i];    /* Scatter self loops at tails     */
        rl->value[i] +=sl->value[rl->next[i]];  /* Gather tail values and add to sum*/
        vp->next[i] = sl->head[i];              /* Reset virtual processor heads   */
    }
}
```

Once Find_Sublist_List finds the tail sublist it sets the tail of rl->next to a self loop, saves the tail of the whole list in sl->random[0] and its value in sl->value[0], and sets the value of tail of the whole list to *zero*. Note that it was not necessary to set the value of the tail of the whole list to *zero* for the Phase 1 because it did not use the tail sublist sum to find the scan of the reduce list. But in Phase 3 it needs the tail value set to *zero* because, otherwise, it may repeatedly add the tail value to the scan at the tail.

Finally, Find_Sublist_List returns the tails of the sublists to self loops, which requires loading sl->tail and scattering it to ll->next. Since the tail values were never added to the sublist sums during Initial_Scan, it next adds the tail values to the sublist sums. The successor sublist saved the tail values in sl->value during Initialize and, therefore, must be indexed by rl->next. It loads rl->value, gathers sl->value using rl->next, adds sl->value to the sums in rl->value and then stores rl->value. Lastly, it reinitializes the virtual processor heads in anticipation of Phase 3. Reinitializing the heads requires loading sl->head and storing it in vp->next.

Scan of the Reduced List: Next List_Scan finds the list scan on the sublists sums rl->value using the list rl->next. If the list is large then it finds it recursively. If list length lies between the recursive cutoff and the serial cutoff it uses Wyllie. If the list length is small it uses Serial_List_Scan. The time for Serial_List_Scan is:

Final_Scan: In Phase 3 List_Scan repeatedly calls Final_Scan to traverse the sublists for n_steps steps. Final_Scan finds the scan of each sublist in the same manner as in Phase 1. The only difference is that Final_Scan scatters the resulting scan vp->sum to ll_sum at the current positions in vp->next.

```
Final_Scan(ll_sum, vp, ll, n_steps)
{
  for (j = 0; j < n_steps; j++)
    for (i = 0; i < vp->n; i++) {
      ll_sum[vp->next[i]] = vp->sum[i];          /* Load and scatter sums           */
      vp->sum[i] +=ll->value[vp->next[i]];       /* Gather value and add to sum     */
      vp->next[i] = ll->next[vp->next[i]];       /* Gather successor link and store */
    }
  }
}
```

Final_Pack: The packing step in Phase 3 is a little simpler than the packing step in Phase 1. Only the completed lists need to write their sums to ll_sum because the active sublists will write their sums on the next call to Final_Scan. However, it is faster to simply load all of vp->sum and scatter to ll_sum than it is to compute the indices of the completed sublists, to gather vp->sum and then scatter them to ll_sum. For active sublists, Final_Pack packs vp->sum and vp->next as in Initial_Pack. However, it does not need to save vp->id.

```
Final_Pack(ll_sum, vp, sl, ll)
{
  j = 0;
  for (i = 0; i < vp->n; i++) {
    ll_sum[vp->next[i]] = vp->sum[i];                    /* Load and scatter sums */
    if (vp->next[i] != ll->next[vp->next[i]]) { /* Pack remaining sublists */
      vp->sum[j] = vp->sum[i];                           /* Gather and store sums */
      vp->next[j++] = vp->next[i];                       /* Gather and store links */
    }
  }
  return j;
}
```

Restore_List: Finally each processor returns the original links and values at the sublist tails. This requires loading sl->random, sl->head, and sl->value and scattering to ll->next and ll->value using sl->random. Because the tail of the whole list is a self loop anyway, it does not set ll->next at the tail.

```
Restore_List(sl, ll)
{
  ll->value[sl->random[0]] = sl->value[0];        /* Reset value of list tail */

  for (i = 1; i < sl->n; i++) {
    ll->next[sl->random[i]] = sl->head[i];        /* Scatter links at tails */
    ll->value[sl->random[i]] = sl->value[i];      /* Scatter values at tails */
  }
}
```

Chapter 3

Fast Set Operations Using Treaps

In this chapter I consider another common pointer-based application, operations on balanced trees.* Instead of trying to obtain the absolute fastest solutions, I used the more modest shared-memory multiprocessors, which are becoming quite common. Again, I was interested in the kinds of problems that pointer-based algorithms present when implemented in parallel and how the architecture affects their performance. First, in Section 3.1, I introduce random balanced binary trees (treaps [103]), which I used for my algorithms and experiments, and relate my algorithms with previous work. In Section 3.2 I review the definition of treaps and the two sequential operations split and join, which I use for aggregate operations. Then I present the aggregate operations, union, intersection, and difference. Section 3.2.3 presents a refinement to union and implementation variations of the operations. In Section 3.3 I show bounds on the work and depth of the algorithms and the union refinement. Section 3.4 shows the results of my implementation experiments both sequentially and in parallel on the SUN Ultra Enterprise 3000 and in parallel on the SGI Power Challenge. I finish with a discussion of the work and how the algorithms might be extended to skip lists.

3.1 Introduction

Balanced trees provide a wide variety of low-cost operations on ordered sets and dynamic dictionaries. Of the many types of balanced trees that have been developed over the years, treaps [103] have the advantage of being both simple and general—in addition to insertion and deletion they easily support efficient finger searching, joining, and splitting. Furthermore, by using appropriate hash functions they require no balance information to be stored at the nodes. Treaps, however, have only been studied in the context of sequential algorithms, and there has been little study of their performance.

I extend previous work on treaps by describing and analyzing parallel algorithms on treaps and by presenting experimental results that demonstrate their utility. I focus on the aggregate set operations *intersection*, *union*, and *difference* since these operations play an important role in databases queries, index searching [114, 83], and other applications. The techniques I describe can also be applied to searching, inserting and deleting multiple elements from a dictionary in parallel.

Treaps are randomized search trees in which each node in the tree has a key and an associated random priority. The nodes are organized so that the keys appear in in-order and the priorities appear in heap-order. Seidel and Aragon showed that insertion, deletion, searching, splitting, and

---

*This chapter is in large part taken from [25]

joining[†] can all be implemented on treaps of size n in $O(\log n)$ expected time, and that given a "finger" in a treap (a pointer to one of the nodes), one can find the key that is d away in the sorted order in $O(\log d)$ expected time (although this "finger searching" requires extra parent pointers). They also showed that one need not store the priorities at the nodes, but instead can generate them as needed using a hashing function. The simplicity of the algorithms led them to conjecture that the algorithms should be fast in practice. Although they did not present experimental results, I ran experiments that compared treaps to splay trees [105], red-black trees, and skip lists [94] and the results show that treaps have around the same performance as these other fast data structures. These results are briefly presented in Section 3.4.1.

My interest is in developing fast parallel algorithms for set operations. Set operations are used extensively for index searching—each term (word) can be represented as a set of the "documents" in which it appears and searches on logical conjunctions of terms are implemented as set operations (intersection for **and**, union for **or**, and difference for **and-not**) [114, 83]. Most web search engines use this technique. The set-union operation is also closely related to merging—it is simply a merge with the duplicates removed. It is well known that for two sets of size m and n ($m \leq n$) merging, union, intersection and difference can be implemented sequentially in $O(m \log(n/m))$ time [30]. In applications in which the sets are of varying and different sizes, this bound is much better than using a regular linear time merge ($O(n + m)$ time) or inserting the smaller set into the larger set one element at a time using a balanced tree ($O(m \log n)$ time). Previous sequential algorithms that achieve these bounds, however, are reasonably messy [30, 31]. The only parallel version I know of that claims to achieve these work bounds [68] is much more complicated than mine.

Seidel and Aragon did not directly consider merging or set operations using treaps. It is straightforward, however, to use their finger search to implement these operations within the optimal time bounds (expected case). In particular, if p_i is the position at which element i of set A would be in set B, the expected time to insert all of A into B, by inserting one element at a time using the previous element as a finger, is $O(\sum_{i=1}^{|A|} (1 + \log(1 + p_i - p_{i-1})))$. With $|A| = m$, $|B| = n$, and $m \leq n$ this sum is bounded by $O(m \log(n/m))$. Although the algorithm is probably simpler than previous techniques, it requires parent pointers and does not parallelize.

In this chapter I describe algorithms for union, intersection and difference on ordered sets that use pairs of treaps directly. The algorithms run in optimal $O(m \log(n/m))$ work and in $O(\log n)$ depth (parallel time) on an EREW PRAM with unit-time scan operations (used for load balancing)—all expected case. This bound is based on automated pipelining (see Chapter 4)—without pipelining the algorithms run in $O(\log^2 n)$ depth. As with the sequential algorithms on treaps, the expectation is over possible random priorities, so the bounds do not depend on the key values. The algorithms are very simple (although the analysis of their bounds is not) and can be applied either sequentially or in parallel. All the algorithms have a similar divide-and-conquer structure and require no extensions to the basic treap data structure. I also show that, for two ordered sets of size n and m, if k is the minimum number of blocks in a partitioning of the first set that contains no elements of the second, a variant of the union algorithm requires only $O(k \log((n + m)/k))$ expected work. This is optimal with respect to the block measure [34].

To analyze the effectiveness of the algorithms in practice I ran several experiments. I was interested in various properties including how well sequentially treaps compare with other balanced trees such as red-black trees, how well the treap algorithms perform for various number of blocks of keys, and how well the algorithms parallelize. The serial experiments show that treaps are competitive with splay trees, skip lists, and red-black trees. They also show that the algorithms

---

[†]Join requires that the largest key in the first tree is smaller than the smallest element of the second tree, while union has no such restriction.

perform well when the keys are clustered into blocks. I used Cilk [26], a parallel extension of C, for the parallel implementation, and ran the experiments on a SUN Ultra Enterprise 3000 and an SGI Power Challenge. The algorithms achieve speedups of between 4.1 and 4.4 on 5 processors of the Sun and between 6.3 and 6.8 on 8 processors of the SGI. I feel that this is reasonably good considering the high memory bandwidth required and the irregular access patterns used by the algorithm.

Related Work

Merging and the related set operations (union, intersection and difference) have been well studied. Merging two ordered sets N and M of size n and m requires at least $\lceil \lg \binom{n+m}{n} \rceil$ comparisons in the worse case since an algorithm needs to distinguish between the $\binom{n+m}{n}$ possible placements of the n keys of N in the result. Without loss of generality I henceforth assume $m \leq n$, in which case $\lceil \lg \binom{n+m}{n} \rceil = \Theta(m \log(n/m))$. Hwang and Lin [67, 71] described an algorithm that matches this lower bound, but the algorithm assumes the input sets are in arrays and only returns cross pointers between the arrays. To rearrange the data into a single ordered output array requires an additional $O(n + m)$ steps. Brown and Tarjan gave the first $\Theta(m \log(n/m))$ time algorithm that outputs the result in the same format as the inputs [30]. Their algorithm was based on AVL-trees and they later showed a variant based on finger searching in 2-3 trees [31]. The same bounds can also be achieved with skip-lists [93].

The lower bound given above makes no assumptions about the input. When using a measure of the "easiness" of the input the bounds can be improved. In particular, suppose set A is divided into blocks of elements $A_1, A_2, \ldots, A_k$ and set B is divided into blocks of elements $B_1, B_2, \ldots, B_l$ such that the merged set is an alternation of these blocks from A and B ($k = l \pm 1$). Such inputs can be merged in $O(k \log((n + m)/k))$ time [34, 93]. In fact any data structure that takes $O(\log k)$ time for a split on the kth element of an ordered set, or to append (join) k elements to an ordered set can be used to achieve these bounds. Pugh used this approach for skip lists [93] and although not discussed directly, the approach can also be applied to treaps when using the "fast" versions of splits and joins. However, as with finger searches, these "fast" versions require parent pointers.

In the parallel setting previous work has focused either on merging algorithms that take $O(n+m)$ work and are optimal when the two sets have nearly equal sizes or on multi-insertion algorithms that take $O(m \log n)$ work and are optimal when the input values to be inserted are not presorted.

Anderson et al. [7], Dekel and Ozsvath [43], Hagerup and Rüb [54], and Varman et al. [110] provide $O(n/p + \log n)$ time EREW PRAM algorithms for merging. Guan and Langston [53] give the first time-space optimal algorithm that takes $O(n/p + \log n)$ time and $O(1)$ extra space on an EREW PRAM. Katajainen et al. [69] gives a simpler algorithm with the same time and space bounds for the EREW and an optimal space-efficient $O(n/p + \log \log m)$ time and $O(1)$ space algorithm for the CREW PRAM.

Paul et al. provide EREW PRAM search, insertion, and deletion algorithms for 2-3 trees [92], and Highan and Schenk have extended these results to B-trees [60]. Ranade [96] gives algorithms for processing least-upper-bound queries and insertion on distributed memory networks. Bäumker and Dittrich [13] give algorithms for search, insertion, and deletion into BB*(a) trees for the BSP* model that are 1-optimal and 2-optimal, where c-optimal means that as the problem size grows the speedup-up tends to p/c, where p is the number of processors, and that the communication time is asymptotically smaller than the computation time. All these algorithms require $O(m \log n)$ work and appear to have large constants.

In 1975 Gravil gave the first CREW PRAM merge algorithm that requires only $O(m \log(n/m))$

comparisons. However, as with Hwang and Lin's serial algorithm, it requires an additional $O(n+m)$ operations to return the sorted merged results in an array. Katajainen describes EREW PRAM algorithms for union, intersection and difference that use the same input and output representations [68]. The algorithms are an extension of Paul et al.'s 2-3 tree algorithms and he claims they run in optimal $O(\log n + \log m)$ depth and $O(m \log(n/m))$ work. The algorithms as described, however, do not actually meet these bounds since the analysis incorrectly assumes a 2-3 tree of depth $\log m$ has $O(m)$ leaves. It may be possible to modify the algorithms to meet the bound and the general approach seems correct.

3.2   Treaps

Treaps use randomization to maintain balance in dynamically changing search trees. Each node in the tree has an associated *key* and a random *priority*. The data are stored in the internal nodes of the tree so that the tree is in in-order with respect to the keys and in heap-order with respect to the priorities. That is, for any node, x, all nodes in the left subtree of x have keys less than x's key and all nodes in the right subtree of x have keys greater than x's key (in-order), and all ancestors of x have priorities greater than x's priority and all descendents have priorities less than x's (heap-order). For example, Figure 3.1a shows a treap where each (letter, number) pair represents a node and the letter is the key value and the number is the priority value. When both the keys and the priorities are unique, there is a unique treap for them, regardless of the order the nodes are added to or deleted from the treap.

The code shown in this section uses a `node` type, which points to a structure that contains the data for a node in a treap. This code and the code I used in my experiments implements persistent versions of the operations. That is, rather than modifying the input trees, the code makes copies of nodes that need to be modified using the function `new_node`. This function fills a new node with the key and priority data given in its arguments. All code in this section along with nonpersistent and parallel versions are available at `http://www.cs.cmu.edu/~scandal/treaps.html`.

3.2.1   Sequential algorithms

Seidel and Aragon showed how to perform many operations on treaps [103]. I quickly review the operations split and join, which the set operations use to manipulating pairs of treaps.

$(L, x, G) = \textbf{split}(T, \textit{key})$ Split T into two trees, L with key values less than *key* and G with key values greater than *key*. If T has a node x with key value equal to *key* then x is also returned.

$T = \textbf{join}(T1, T2)$ Join $T1$ and $T2$ into a single tree T, where the largest key value in $T1$ is less than the smallest key value in $T2$.

Below I give the recursive top-down C code for persistent `split` and `join`. The `split` and `join` I use in my experiments are the slightly more efficient iterative versions. In addition, there are bottom-up algorithms that use rotations for these operations [103].

Split   To split a tree rooted at r by key value a `split` follows the in-order access path with respect to the key value a until it reaches either a node with key value a or a leaf node. If the root key is less than a, the root becomes the root of the "less-than" tree. Recursively, `split` splits the right child of the root by a, and then makes the resulting tree with keys less than a the new right child of the root and makes the resulting tree with keys greater then a the "greater-than" tree. Similarly, if the

root key is greater than a split recursively splits the left child of the root. If the root key is equal to a split returns the root and the left and right children as the "less-than" and "greater-than" trees, respectively. Figure 3.1 shows the result of a split on a treap. The expected time to split treaps into two treaps of size n and m is $O(\log n + \log m)$ [103]. The following code returns the less and greater results by side effecting the first two arguments. It returns an equal key, if present, as the result.

```
node split(node *less, node *gtr, node r, key_t key)
{
  node root;
  if (r == NULL) {*less = *gtr = NULL; return NULL;}

  root = new_node(r->key, r->priority);
  if (r->key < key) {
    *less = root;
    return split(&(root->right), gtr, r->right, key);
  } else if (r->key > key) {
    *gtr = root;
    return split(less, &(root->left), r->left, key);
  } else {
    *less = r->left;
    *gtr = r->right;
    return root;
  }
}
```
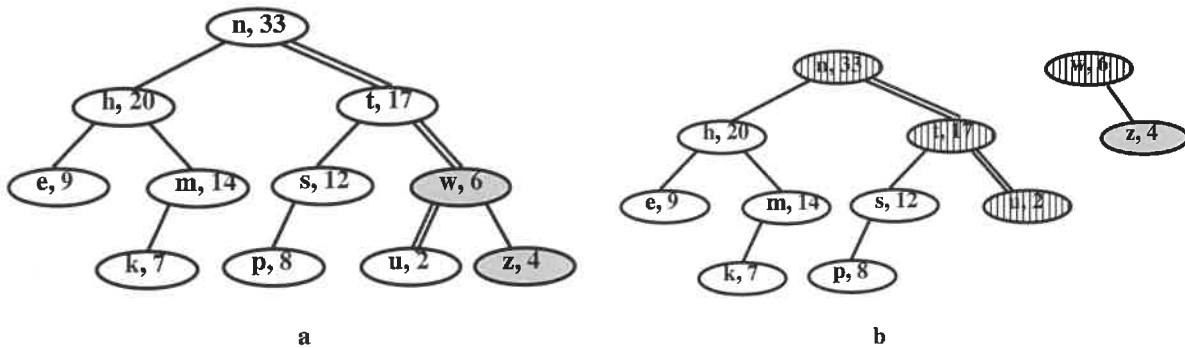

Figure 3.1: Treap Split. **a:** Input tree for split with each node's (key,priority) values. When splitting by the key v, the unshaded region becomes the less-than tree and the shaded region becomes the greater-than tree. The double links show the path split follows. **b:** The result trees. The striped nodes are the new nodes that the persistent version of split creates.

Join To join two treaps $T1$ with keys less than a and $T2$ with keys greater than a join traverses the right spine of $T1$ and the left spine of $T2$. A left (right) spine is defined recursively as the root plus the left (right) spine of the left (right) subtree. To maintain the heap order join interleaves pieces of the spines so that the priorities descend all the way to a leaf. The expected time to join two treaps of size n and m is $O(\log n + \log m)$ [103].

```
node join(node r1, node r2)
{
  node root;
  if (r1 == NULL) {return r2;}
  if (r2 == NULL) {return r1;}

  if (r1->priority < r2->priority) {
    root = new_node(r1->key, r1->priority);
    root->left = r1->left;
    root->right  = join(r1->right, r2);
  } else {
    root = new_node(r2->key, r2->priority);
    root->left  = join(r1, r2->left);
    root->right = r1->right;
  }
  return root;
}
```

3.2.2   Parallel algorithms

In the parallel setting we view each treap as an ordered set of its keys and we consider the following operations:

$T = \mathbf{union}(T1, T2)$ Find the union of treaps $T1$ and $T2$ to form a new treap T.

$T = \mathbf{intersect}(T1, T2)$ Find the intersection of treaps $T1$ and $T2$ to form a new treap T.

$T = \mathbf{diff}(T1, T2)$ Remove from $T1$ nodes that have the same key values as node in $T2$, returning its new root T.

All three algorithms have a similar divide-and-conquer structure in which they use the key from the larger priority root to split the tree with the smaller priority root, and then make two recursive calls (which can be parallel) on the values less and greater than the key. The algorithms differ in how they combine the results. The code I show below is for the sequential C versions. To make them parallel using Cilk, one needs only put the cilk keyword before each function definition, the spawn keyword before each recursive call, and the sync keyword after both recursive calls. As discussed in 3.4.2 the versions used in the experiments also terminate the parallel calls at a given depth in the tree to reduce the overhead of spawning.

Union: To maintain the set in heap- order with respect to the priorities, union makes r, the root with the largest priority, the root of the result treap. If the key of r is k then, to maintain the set in-order with respect to the keys, union splits the other treap into a "less-than" treap with key values less than k and "greater-than" treap with key values greater than k, and possibly a duplicate node with a key equal to k. Then, recursively (and in parallel) it finds the union of the left child of r and the less-than tree, and finds the union of the right child of r and the greater-than tree. The result of the two union operations become the left and right subtrees of r, respectively. The following C code implements the algorithm.

```
node union(node r1, node r2)
{
  node root, less, gtr, duplicate;

  if (r1 == NULL)  return r2;
  if (r2 == NULL)  return r1;

  if (r1->priority < r2->priority) swap(&r1, &r2);
  duplicate = split(&less, &gtr, r2, r1->key);

  root = new_node(r1->key, r1->priority);
  root->left  = union(r1->left, less);
  root->right = union(r1->right, gtr);
  return root;
}
```

Intersection: As with union, `intersection` starts by splitting the treap with the smaller priority root using k, the key of the root with the greater priority. It then finds the intersection of the two left subtrees, which have keys less than k, and the intersection of the two right subtrees, which have keys greater than k. If k appeared in both trees then these results become the left and right children of root used to split. Otherwise it returns the join of the two recursive call results.

```
node intersect(node r1, node r2)
{
  node root, less, gtr, left, right, duplicate;

  if ((r1 == NULL) || (r2 == NULL)) return NULL;

  if (r1->priority < r2->priority) swap(&r1, &r2);
  duplicate = split(&less, &gtr, r2, r1->key);

  left  = intersect(r1->left, less);
  right = intersect(r1->right, gtr);

  if (duplicate == NULL) {
    return destruct_join(left, right);
  } else {
    root = new_node(r1->key, r1->priority);
    root->left = left;
    root->right = right;
    return root;
  }
}
```

Notice that because the nodes returned by the intersection are all copies of input tree nodes, `intersect` can use a destructive version of join, one that modifies the nodes of the tree.

Difference: To find the difference of two treaps $T1$ and $T2$ diff splits the treap with the smaller priority root using k, the key of the root of the other treap. Then it finds the difference of the two left subtrees, which have keys less than k, and the difference of the two right subtrees, which have keys greater than k. Although not necessary, to simplify the code below I use the boolean r2_is_subtr to distinquish when $T2$ is the subtrahend (the set specifying what should be removed) and when $T2$ is not. If $T2$ is the subtrahend and it does not contain k, then it sets the left and right children of the root of $T1$ to the results of the recursive calls and returns this root. Otherwise it returns the join of the results of the recursive calls.

```
node diff(node r1, node r2, bool r2_is_subtr)
{
  node root, less, gtr, left, right, duplicate;

  if ((r1 == NULL) || (r2 == NULL))
    return r2_is_subtr ? r1 : r2;

  if (r1->priority < r2->priority) {
    r2_is_subtr = !r2_is_subtr;
    swap(&r1, &r2);
  }
  duplicate = split(&less, &gtr, r2, r1->key);

  left  = diff(r1->left, less, r2_is_subtr);
  right = diff(r1->right, gtr, r2_is_subtr);

  /* Keep r1 if no dupl. and subtracting r2 */
  if ((duplicate == NULL) && r2_is_subtr) {
    root = new_node(r1->key, r1->priority);
    root->left = left;
    root->right = right;
    return root;
  } else {                        /* Delete r1 */
    return join(left, right);
  }
}
```

3.2.3 Extensions

Using Fast Split and Join: The versions of split and join that the above algorithms use can split a treap into two treaps of size m and n or join two treaps of size m and n with $O(\lg \max(n, m))$ expected work. Seidel and Aragon also describe "fast" versions of split and join that use $O(\lg \min(n, m))$ expected work. These versions use parent pointers for quick access from the two ends of a set, and are similar to finger searching. The use of such fast versions of join and split does not effect the asymptotic work bounds assuming a general ordering of the input sets, but they do allow us to generate a parallel algorithm that is optimal with respect to the block measure. As described in the introduction if set A is divided into blocks of $A_1, A_2, \ldots, A_k$ and set B is divided into blocks of elements $B_1, B_2, \ldots, B_l$ such that the merged set is an alternation of these blocks from A and B, then A and B can be merged in $O(k \log((n+m)/k))$ time. This bound

also applies to union, although k is defined as the minimum number of blocks of A such that no value of B lies within a block. These times are optimal but previous algorithms are all sequential.

Section 3.3.2 shows that the parallel union algorithm achieves the same work bounds if it uses fast splits and joins. The modified algorithm also requires another change, at least for the analysis. In this change, if split(&less,&grt,r2,r1->key) returns an empty tree in less, then the algorithm executes the following instead of making the two recursive calls

```
km = minkey(r2);
dup = split(&ll,&rr,r1,km);
root = join(ll,union(rr,r2));
```

where minkey(r2) returns the minimum key in the set represented by r2. A symmetric case is used if grt is empty.

Note that although the analysis uses the fast splits and joins to prove the bounds, the algorithm with the slow versions still seem to work very well experimentally with respect to number blocks. I give some experimental results in Section 3.4.2.

Nonpersistent versions: All the code I show and use in my experiments is fully persistent in that it does not modify the input treaps. Persistence is important in any application where the input sets or intermediate results might be reused in future set operations. Certainly in the application of index searching we do not want to destroy the sets specifying the documents when manipulating them. In other applications, such as updating a single dictionary kept as a treap, nonpersistent versions are adequate. In such applications we typically view one of the treaps as the treap to be modified and the other as the set of modifications to make. The only changes that need to be made to my code to make such nonpersistent versions is to have them modify the nodes rather than create new ones, and to explicitly delete nodes that are being removed.

The relative performance of the persistent and nonpersistent versions are discussed in Section 3.4.2. The persistent code is not as space efficient as the more sophisticated method of Driscoll et al. [44], but their solution is much more complex and does not apply to operations that combine two trees.

3.3 Analysis

In Section 3.3.1 I analyze the expected work to find the union, intersection, and difference of two treaps t_n and t_m of size n and m, with $m \leq n$. In Section 3.3.2 I analyze union with fast splits and show that it is optimal with respect to interleaving blocks of keys.

3.3.1 Union, intersection, and difference

Consider union first. Without loss of generality, assume that the key values for the two trees are $1, 2, 3, \ldots, n + m$. Let N and M be the sets of keys for t_n and t_m, respectively, such that $N \cup M = 1, 2, 3, \ldots, n + m$ and $N \cap M = \emptyset$. (The expected work for the split operation in union is maximized when $N \cap M = \emptyset$.)

Since the priorities for the two treaps are chosen at random, arranging the keys so that their priorities are in decreasing order results in a random permutation $\sigma \in S_{n+m}$, where S_{n+m} is the set of all permutations on $n + m$ items. Along with N and M this permutation defines the result treap and the parallel work and depth required to find it. Therefore, we define $\mathbf{W}(N, M, \sigma)$ to be the work required to take the union of two *specific* treaps, which depends both on the interleaving of the key values in N and M and on the permutation σ defined by their priorities.

We define $\mathbf{E}\left[\mathbf{W}(N, M)\right]$ to be the expected work to take the union of N and M averaged over all permutations (*i.e.*, $1/(n + m)! \sum_{\sigma \in S_{n+m}} \mathbf{W}(N, M, \sigma)$). Even when the sizes of N and M are fixed this expected work can depend significantly on the interleaving of N and M. We define $\mathbf{E}\left[\mathbf{W}(n, m)\right] = \max\{\mathbf{E}\left[\mathbf{W}(N, M)\right], |N| = n, |M| = m\}$. This is the worst case work over the interleavings and expected case over the permutations, and is what we are interested in.

The work for a permutation $\sigma = (a_1 = i, a_2, a_3, \ldots, a_{n+m})$ is the time to split one treap using i plus the work to take the union of the treaps with keys less than i and the union of the treaps with keys greater than i. We use the notation $N < i$ to indicate all keys in the set N which are less than i. Since it is equally likely that any i will have highest priority, we can write the following recurrence for the expected work for a given N and M averaged over all permutations

$$(n + m)\mathbf{E}\left[\mathbf{W}(N, M)\right] = \sum_{i=1}^{n+m} \left(\mathbf{E}\left[\mathbf{W}(N < i, M < i)\right] + \mathbf{E}\left[\mathbf{W}(N > i, M > i)\right]\right) +$$
$$\sum_{i \in N} \mathbf{E}\left[\mathbf{T}_{split}(M, i)\right] + \sum_{i \in M} \mathbf{E}\left[\mathbf{T}_{split}(N, i)\right] + (n + m)d,$$

where d is a constant. From [103] we know that $\mathbf{E}\left[\mathbf{T}_{split}(N, i)\right] = O(\log n)$. These lead to the following lemma.

Lemma 3.1 *The expected work to take the union of two treaps of size n and m is bound by*

$$(n + m)\mathbf{E}\left[\mathbf{W}(n, m)\right] \leq \max_{\{p_i\}} \sum_{i=1}^{n+m-1} \{\mathbf{E}\left[\mathbf{W}(p_i, i - p_i)\right] + \mathbf{E}\left[\mathbf{W}(n - p_i, m - i + p_i)\right]\}$$
$$+ 2\mathbf{W}(0, 0) + nO(\log m) + mO(\log n) + (n + m)d, \qquad (3.1)$$

where $\{p_i\}$ denotes the set $\{p_1, \ldots, p_n\}$ such that $0 \leq p_i \leq n$, $0 \leq i - p_i \leq m$, and $p_i \leq p_{i+1}$.

Proof. Assume $|N| = n$ and $|M| = m$.

$$(n + m)\mathbf{E}\left[\mathbf{W}(n, m)\right] = (n + m)\max_{N,M} \mathbf{E}\left[\mathbf{W}(N, M)\right]$$
$$\leq \max_{N,M} \sum_{i=1}^{n+m} \left(\max_{N<i,M<i} \mathbf{E}\left[\mathbf{W}(N < i, M < i)\right] + \max_{N>i,M>i} \mathbf{E}\left[\mathbf{W}(N > i, M > i)\right]\right)$$
$$+ nO(\log m) + mO(\log n) + (n + m)d$$
$$= \max_{N,M} \sum_{i=1}^{n+m} \left(\mathbf{E}\left[\mathbf{W}(|N < i|, |M < i|)\right] + \mathbf{E}\left[\mathbf{W}(|N > i|, |M > i|)\right]\right)$$
$$+ nO(\log m) + mO(\log n) + (n + m)d \qquad (3.2)$$

Let $p_i = |N \leq i|$, the number of items in N that are less than or equal to i. Clearly, if $i \in N$ then $p_i = p_{i-1} + 1$, and if $i \in M$ then $p_i = p_{i-1}$. Then

$$\sum_{i=1}^{n+m} \mathbf{E}\left[\mathbf{W}(|N < i|, |M < i|)\right] = \mathbf{W}(0, 0) + \sum_{\substack{i=2 \\ i \in N}}^{n+m} \mathbf{E}\left[\mathbf{W}(p_i - 1, i - p_i)\right]$$
$$+ \sum_{\substack{i=2 \\ i \in M}}^{n+m} \mathbf{E}\left[\mathbf{W}(p_i, i - p_i - 1)\right]$$

$$= \mathbf{W}(0,0) + \sum_{i=2}^{n+m} \mathbf{E}\left[\mathbf{W}(p_{i-1}, i - p_{i-1} - 1)\right]$$

$$= \mathbf{W}(0,0) + \sum_{i=1}^{n+m-1} \mathbf{E}\left[\mathbf{W}(p_i, i - p_i)\right], \tag{3.3}$$

and

$$\sum_{i=1}^{n+m} \mathbf{E}\left[\mathbf{W}(|N>i|, |M>i|)\right] = \sum_{i=1}^{n+m-1} \mathbf{E}\left[\mathbf{W}(n - p_i, m - i + p_i)\right] + \mathbf{W}(0,0) \tag{3.4}$$

Substituting Equations 3.3 and 3.4 in Equation 3.2 we get

$$
\begin{aligned}
(n+m)\mathbf{E}\left[\mathbf{W}(n,m)\right] \;\leq\; & \max_{N,M} \sum_{i=1}^{n+m-1} \left\{\mathbf{E}\left[\mathbf{W}(p_i, i - p_i)\right] + \mathbf{E}\left[\mathbf{W}(n - p_i, m - i + p_i)\right]\right\} \\
& + 2\mathbf{W}(0,0) + nO(\log m) + mO(\log n) + (n+m)d \\
=\; & \max_{\{p_i\}} \sum_{i=1}^{n+m-1} \left\{\mathbf{E}\left[\mathbf{W}(p_i, i - p_i)\right] + \mathbf{E}\left[\mathbf{W}(n - p_i, m - i + p_i)\right]\right\} \\
& + 2\mathbf{W}(0,0) + nO(\log m) + mO(\log n) + (n+m)d.
\end{aligned}
$$

∎

Now we apply induction on the recurrence to show the bound on the expected work. If we assume $\mathbf{E}\left[\mathbf{W}(n,m)\right] = a\log\binom{n+m}{n} - b\log(n+m)$ and $\mathbf{W}(n,0) = \mathbf{W}(0,m) = c$, where $a, b,$ and c are constants, and substitute in Equation 3.1 we get

$$
\begin{aligned}
(n+m)\mathbf{E}\left[\mathbf{W}(n,m)\right] \;\leq\; & \max_{\{p_i\}} a \sum_{i=1}^{n+m-1} \left\{\log\binom{i}{p_i} + \log\binom{n+m-i}{n-p_i}\right\} \\
& - \min_{\{p_i\}} b \sum_{i=1}^{n+m-1} \left\{\log i + \log(n+m-i)\right\} \\
& + 2c + nO(\log m) + mO(\log n) + (n+m)d. \tag{3.5}
\end{aligned}
$$

Consider the expression containing the maximum. First we find the integral p_i values that maximize the sum with the a constant preceding it. Then we will place an upper bound on this expression by using a nonintegral approximations to p_i and Sterling's approximation for factorials.

Lemma 3.2 *The expression* $\log\binom{i}{p_i} + \log\binom{n+m-i}{n-p_i}$ *is maximized when* $p_i = \hat{p}_i$, *where* $\hat{p}_i = \left\lfloor \frac{(n+1)(i+1)}{n+m+2} \right\rfloor$.

Proof. We want to find integer p_i such that the expression using $p_i - 1$ or $p_i + 1$ is no greater than that with p_i. That is, find p_i such that

$$\binom{i}{p_i - 1}\binom{n+m-i}{n-p_i+1} \leq \binom{i}{p_i}\binom{n+m-i}{n-p_i}$$

and

$$\binom{i}{p_i + 1}\binom{n+m-i}{n-p_i-1} \leq \binom{i}{p_i}\binom{n+m-i}{n-p_i}$$
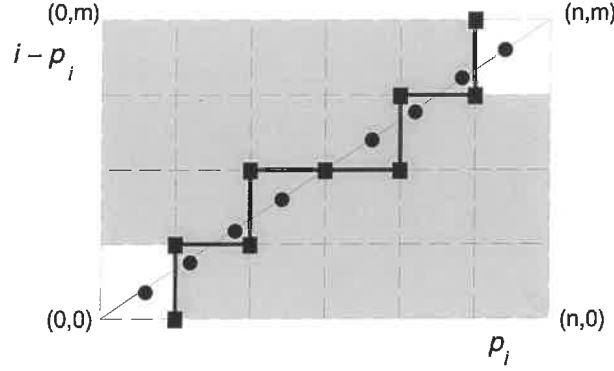
Figure 3.2: Maximizing the expected work for union. The shaded region is the region of possible values for $(p_i, i - p_i)$. The points (squares) on the step function maximize $\binom{i}{p_i}\binom{n+m-i}{n-p_i}$. The points (circles) on the straight line are the continuous approximation to the points on the step function.

This occurs when

$$\frac{1}{(i - p_i + 1)(n - p_i + 1)} \leq \frac{1}{p_i(m - i + p_i)}$$

$$p_i(m - i + p_i) \leq (i + 1)(n + 1) - (i + 1)p_i - (n + 1)p_i + p_i^2$$

$$p_i \leq (i + 1)(n + 1)/(m + n + 2)$$

and when

$$\frac{1}{(p_i + 1)(m - i + p_i + 1)} \leq \frac{1}{(i - p_i)(n - p_i)}$$

$$p_i(m - i + 2 + p_i) + (m - i + 1) \geq in - p_i(n + i - p_i)$$

$$p_i \geq (in - m + i - 1)/(m + n + 2)$$

$$= (i + 1)(n + 1)/(m + n + 2) - 1$$

Therefore $\hat{p}_i$ must be as in the statement of the lemma.   ∎

Lemma 3.3 *When p_i is an integer for all i and $0 \leq p_i \leq n$, $0 \leq i - p_i \leq m$, and $p_i \leq p_{i+1}$*

$$\max_{\{p_i\}} a \sum_{i=1}^{n+m-1} \left\{ \log \binom{i}{p_i} + \log \binom{n + m - i}{n - p_i} \right\} \leq a(n + m) \left\{ \log \binom{n + m}{n} + 2 \log n + 5 \right\}. \quad (3.6)$$

Proof.   Because it is hard to work with floor values we use $\tilde{p}_i = ni/(n + m)$ as a continuous approximation to $\hat{p}_i$. Figure 3.2 graphically shows the relationship between the $\tilde{p}_i$, which lies on a straight line between $(0, 0)$ and (n, m), and $\hat{p}_i$, which is on a step function that follows this line. In the following expressions, we use $\binom{i}{p}$ to denote $i!/\Gamma(p + 1)\Gamma(i - p + 1)$, when p is not an integer. Then

$$\log \left[\binom{i}{\hat{p}_i} \binom{n + m - i}{n - \hat{p}_i} \right] \leq \log \left[nm \binom{i}{\tilde{p}_i} \binom{n + m - i}{n - \tilde{p}_i} \right].$$

When we use $\gamma + (n + 1/2) \log n - n \leq \log \Gamma(n + 1) \leq \gamma + (n + 1/2) \log n - n + 1/12n$, where $\gamma = \log \sqrt{2\pi}$, for n real (see [70] exercise 1.2.11.2-6) to substitute for the binomial coefficients, we

get

$$\max_{\{p_i\}} a \sum_{i=1}^{n+m-1} \left\{ \log \binom{i}{p_i} + \log \binom{n+m-i}{n-p_i} \right\}$$

$$= \max_{\{p_i\}} a \sum_{i=1}^{n+m-1} \left\{ \log \binom{i}{p_i} + \log \binom{i}{n-p_{n+m-i}} \right\}$$

$$\leq 2a \sum_{i=1}^{n+m-1} \left\{ \log \binom{i}{\tilde{p}_i} + \log(nm) \right\}$$

$$\leq 2a \sum_{i=1}^{n+m-1} \left\{ [\gamma + (i+1/2)\log i - i + 1/12i] - [\gamma + (\tilde{p}_i + 1/2)\log \tilde{p}_i - \tilde{p}_i] \right.$$
$$\left. - [\gamma + (i - \tilde{p}_i + 1/2)\log(i - \tilde{p}_i) - (i - \tilde{p}_i)] + \log(nm) \right\}$$

$$= 2a \sum_{i=1}^{n+m-1} \left\{ -\gamma + (\tilde{p}_i + 1/2)\log\left(\frac{i}{\tilde{p}_i}\right) + (i - \tilde{p}_i + 1/2)\log\left(\frac{i}{i - \tilde{p}_i}\right) \right.$$
$$\left. - \frac{\log i}{2} + \frac{1}{12i} + \log(nm) \right\}$$

$$\leq a \sum_{i=1}^{n+m-1} \left\{ -2\gamma + \left(\frac{2ni}{n+m} + 1\right)\log\left(\frac{n+m}{n}\right) + \left(\frac{2mi}{n+m} + 1\right)\log\left(\frac{n+m}{m}\right) \right.$$
$$\left. - \log i + \frac{1}{6i} + 2\log(nm) \right\}$$

$$\leq a\{(n+m-1)\left[-2\gamma + (n+1)\log((n+m)/n) + (m+1)\log((n+m)/m)\right]$$
$$- [\gamma + (n+m-1/2)\log(n+m-1) - (n+m-1)]$$
$$+ \log(n+m)/6 + 2(n+m-1)\log(nm)\}$$

$$\leq a(n+m)\left\{ -\gamma + \left(n + \frac{1}{2}\right)\log\left(\frac{n+m}{n}\right) + \left(m + \frac{1}{2}\right)\log\left(\frac{n+m}{m}\right) - \frac{1}{2}\log(n+m) \right.$$
$$- \frac{1}{12n} - \frac{1}{12m} - \gamma + \frac{1}{2}\log\left(\frac{n+m}{n}\right) + \frac{1}{2}\log\left(\frac{n+m}{m}\right) + \frac{1}{2}\log(n+m) + \frac{1}{12n} + \frac{1}{12m}$$
$$\left. + 1 + \frac{\log(n+m)}{6(n+m)} + 2\log(nm) \right\}$$

$$\leq a(n+m)\left\{ \log\binom{n+m}{n} + 6\log(nm) \right\}.$$

$\blacksquare$

Next we consider the second summation in Equation 3.5.

$$b \sum_{i=1}^{n+m-1} [\log i + \log(n+m-i)] \geq 2b[\gamma + (n+m-1/2)\log(n+m) - (n+m)] \qquad (3.7)$$

Theorem 3.4 *The expected work to take the union of two treaps t_n and t_m of size n and m ($m \leq n$) is*

$$\mathbf{E}\left[\mathbf{W}(n,m)\right] = O(m\log(n/m))$$

Proof.   Substituting Equations 3.6 and 3.7 in Equation 3.5 gives

$$
\begin{aligned}
(n+m)\mathbf{E}\left[\mathbf{W}(n,m)\right] \;\le\;& a(n+m)\left[\log\binom{n+m}{n}+6\log(nm)\right] \\
&-2b\gamma-2b(n+m)\log(n+m)+b\log(n+m)+2b(n+m) \\
&+2c+gn\log m+gm\log n+d(n+m) \\
\le\;& (n+m)\left[a\log\binom{n+m}{n}-b\log(n+m)\right] \\
&+(n+m)[-b\log(n+m)+b\log(n+m)/(n+m)+2b \\
&+(g+6a)\log(nm)+d]+2c-2b\gamma \\
\le\;& (n+m)\left[a\log\binom{n+m}{n}-b\log(n+m)\right] \\
=\;& (n+m)O(m\log(n/m))
\end{aligned}
$$

for sufficiently large b, n, and m, and $m \le n$.   ∎

Corollary 3.5 *The expected work to take the intersection of two treaps t_n and t_m of size n and m, respectively, and $m \le n$ is:*

$$
\mathbf{E}\left[\mathbf{W}(n,m)\right]=O(m\log(n/m))
$$

Proof.   The only additional work that intersection does that union does not do is a join when there is no duplicate key. But since the join must be on trees that are no larger than the result trees of the split prior to the recursive calls, the additional join only changes the constants in the work bound.   ∎

Corollary 3.6 *The expected work to take the difference of two treaps t_n and t_m of size n and m, respectively, and $m \le n$ is:*

$$
\mathbf{E}\left[\mathbf{W}(n,m)\right]=O(m\log(n/m))
$$

Proof.   As with intersection the only additional work difference does that union does not is a join. When a subtree of t_n was split this join takes no more work than the split preceding it. When a subtree of t_m was split the join may take work proportional to the log of the size of the corresponding subtree of t_n. However, as this join only takes place when the key occurs in both t_n and t_m, the work over all permutations associated with the join is $mO(\log n)$. Thus, the work bound for difference is same as for union.   ∎

Theorem 3.7 *When $p = m/\log m$ the expected depth to take the union, intersection, or difference of two treaps t_n and t_m of size n and m, respectively, and $m \le n$ is*

$$
\mathbf{E}\left[\mathbf{D}(n,m)\right]=O(\log m\log n)
$$

Proof. Let h_n and h_m be the heights of t_m and t_n, respectively. Every time t_m is split it takes $O(h_m)$ time. Although the heights of the resulting trees may not be smaller than the original tree, the height of the subtrees from t_n are reduced by one. Similarly, when t_n is split the heights of the subtrees from t_m are reduced by one. Thus, after $O(h_m h_n)$ steps the algorithm completes. Since the expected heights of t_m and t_n are $O(\log m)$ and $O(\log n)$, the expected depth of each operation is $O(\log m \log n)$. ■

In Chapter 4 I show that the depths of the operations can be reduced to $O(\log m + \log n)$ using pipelining. Furthermore, I show that since the recursive calls in the algorithms are independent (never access the same parts of the trees), the algorithms run with exclusive reads and writes. Using Brent's scheduling principle these results together with the work bounds imply the algorithms run in $O(\frac{m \log(n/m)}{p} + T_s \log n)$ time on an EREW PRAM, where T_s is the time for a scan, which is needed for scheduling the tasks to the processors. On a plain EREW PRAM $T_c = \log p$ and on a PRAM with scan operations $T_c = 1$ [19].

3.3.2 Analysis of union with fast splits

Here I prove bounds on the work for union using fast splits when using the block metric. I do not know how to pipeline this version, so the depth of the algorithm on two sets of size n and m is $O(\log n \log m)$.

Lemma 3.8 *Assuming the expected cost to cut a sequence into two sequences of nonzero length n and m is $1 + \lg(\min(n, m))$, any set of cuts that partitions a sequence of length n into k blocks has a total expected cost $T_p \leq 2k(1 + \lg(n/k))$*

Proof. For a sequence of blocks N we will denote the lengths of the blocks as $\{n_1, n_2, \ldots, n_k\}$ and the sum of the lengths as n. We define $T_s(N) = \sum_1^k (1 + \lg n_i)$. Since the logarithm is concave downward, for a fixed k and n this sum is maximized when all the blocks are the same length, giving $T_s(N) \leq k(1 + \lg\lceil n/k \rceil)$. We can model a set of cuts that partitions a sequence into blocks as a tree with the blocks at the leaves and each internal node representing one of the cuts. We use $T_p(v)$ to denote the total expected cost of the cuts for a tree rooted at v. We use $T_s(v)$ to refer to $T_s(N)$ where N are the blocks at the leaves of the tree rooted at v. By our assumption of the cost of a cut we can write the recurrence

$$T_p(v) = \begin{cases} 0 & v \text{ a leaf} \\ T_p(l(v)) + T_p(r(v)) + 1 + \lg(\min(|l(v)|, |r(v)|)) & \text{otherwise} \end{cases} \quad (3.8)$$

where $l(v)$ and $r(v)$ are the left and right children of v, and $|v|$ is the sum of the sizes of the blocks in the tree rooted at v. The following is also true, by definition

$$T_s(v) = \begin{cases} 1 + \lg(|v|) & v \text{ a leaf} \\ T_s(l(v)) + T_s(r(v)) & \text{otherwise.} \end{cases}$$

Now we prove by induction on the tree that $T_p(v) \leq 2T_s(v) - \lg(|v|) - 2$. In the base case it is true for the leaves since $0 \leq 2(1 + \lg|v|) - \lg|v| - 2$. For the induction case we substitute T_s into the right hand side of Equation 3.8 giving

$$\begin{aligned} T_p(v) &\leq 2T_s(l(v)) - \lg(|l(v)|) - 2 + 2T_s(r(v)) - \lg(|r(v)|) - 2 \\ &\quad + 1 + \lg(\min(|l(v)|, |r(v)|)) \\ &= 2T_s(v) - \lg(n) - 3 \\ &\leq 2T_s(v) - \lg(n + m) - 2 \\ &= 2T_s(v) - \lg(|v|) - 2, \end{aligned}$$

where $n = \max(|l(v)|, |r(v)|)$ and $m = \min(|l(v)|, |r(v)|)$. Since $T_s(N) \le k(1 + \lg(\lceil n/k \rceil))$ we have for any set of cuts $T_p(N) \le 2k(1 + \lg(\lceil n/k \rceil))$. ∎

Theorem 3.9 *The parallel union algorithm using fast splits and joins on two ordered sets A and B runs in $O(k \lg((n+m)/k))$ expected work where $|A| = n$, $|B| = m$, and k is the minimum number of blocks in a partitioning of A such that no value of B lies within a block.*

Proof. We count all the work of the algorithm against the cost of cutting A into k blocks, cutting B into $k \pm 1$ blocks, and joining the $2k \pm 1$ blocks.[‡] Since the fast versions of split and join take work bounded by what is required by Lemma 3.8, the total expected work of cutting and joining is bound by $O(k \lg((n + m)/k))$.

We consider two cases. First, when the split in the union returns two nonempty sets. In this case we count the cost of the split and the constant overhead of the union against the partitioning of either A or B (whichever is being split). Note that on either input set there can be at most k cuts due to calls to the split function. Union can make additional cuts within a block when it removes the root of the tree (when it has the higher priority) and divides the tree into its left and right branches. But these trivial cuts will only reduce the cost of the split cuts and are charged against a split in the other tree. Second, consider when the split in the union returns an empty set. In this case, even though the split only takes constant work, we cannot count it or the union overhead against the k cuts of A or B. Assume that $(\emptyset, T_2) = \mathtt{split}(T_2, r_1)$ (*i.e.*, T_2 is being split by the root of T_1). Recall, that when the fast-split version of union gets an empty set, it then splits T_1 by the first value in T_2 giving T_{11} and T_{12} and executes $\mathtt{join}(T_{11}, \mathtt{union}(T_{12}, T_2))$. Finding the minimum value of T_2 takes constant time using a finger search. We count the cost of the split against the partitioning of the set corresponding to T_1 (unless T_{12} is empty, in which case we count the constant cost against the join). Since $\mathtt{minkey}(T_2) < \mathtt{minkey}(T_{12})$, the $\mathtt{join}$ is along one of the cuts between blocks of the result. We can therefore count the cost of the join and the constant overhead in the union against joining the $2k \pm 1$ result blocks. The constant work of any calls to union with an empty set (base of the recursion) are counted against their parent's split or join. ∎

3.4 Implementation

To evaluate the performance of these set-based operations, I implemented the serial treap algorithms in Gnu C and the parallel ones in Cilk 5.1 [26]. Cilk is a language for multithreaded parallel programs based on ANSI C, and is designed for computations with dynamic, highly asynchronous parallelism, such as divide-and-conquer algorithms. It includes a runtime system that schedules the multithreaded computation using work-stealing and provides dag-consistent distributed shared memory. I ran my experiments on an SGI Power Challenge with 16 195MHz R10000 processors and 4 Gbytes of memory running the IRIX 6.2 operating system, and on a Sun Ultra Enterprise 3000 with six 248MHz UltraSPARC II processors and 1.5Gbytes of memory running the SunOS 5.5.1 operating system. For each timing experiment I used at least five different data sets; the relative standard error of the times were less than 2% for the SUN Ultra Enterprise 3000 and less than 3% for the SGI Power Challenge.

[‡]To be precise, we count against a constant multiple of the plain split and join times since we include some constant-work overheads in the union function in their times.

3.4.1 Sequential experiments

Since speedup is a common measure of the performance of parallel algorithms, it is important to compare the parallel performance with a good sequential algorithm. My first step was to verify that sequential treaps compare reasonably well with other good sequential balanced tree algorithms. I implemented and compared the performance of red/black trees [102], splay trees [105] (Sleator's code), skip lists [93] (Pugh's code), and treaps on an SGI Power Challenge and SUN Ultra Enterprise 3000.

To evaluate the performance of the algorithms I performed a series of tests that create a tree of size n, and insert, delete and search for k keys in a tree of size n. For each test I also used four data distributions. One distribution inserts, searches, and deletes random keys. The remaining distributions insert, search, and delete consecutive keys in various orders. Figure 3.3 shows the time to create a tree from random keys. The union version for treaps creates the tree using recursive calls to union organized as in mergesort, instead of inserting one at a time. My other experiments give similar results, and in all cases all four data structures give running times that are within a factor of 2 of each other.
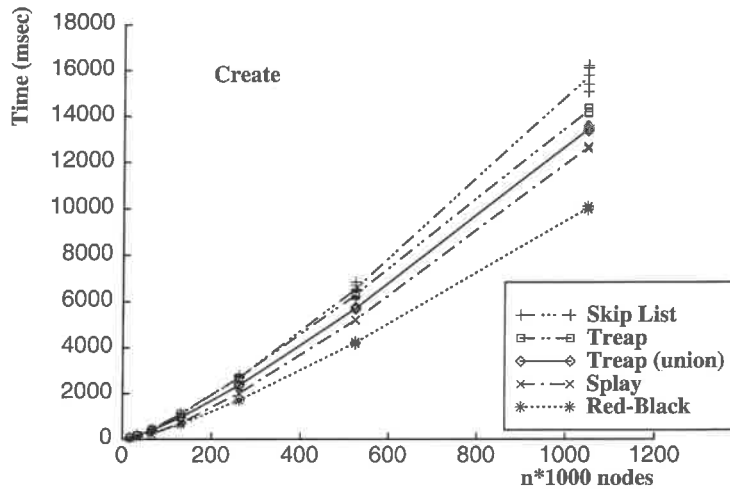
Figure 3.3: Time to create various data structures from n random keys on a single processor of a SUN Ultra Enterprise 3000.

Next I show the results of union on one processor. Figure 3.4 shows the runtime for the union operation on treaps of various input sizes n and m, where the keys are random integers over the same range of values. Notice that the x-axis specifies the sum of the treap sizes $n + m$ and each curve is for unions where one tree size stays fixed. As the graph shows the lines rise rapidly until $m = n$ and then rise more slowly. This change reflects the symmetry of the changing roles as the one tree switches from being the smaller to the larger. The envelope of the lines give the linear union times when $n = m$. Thus, one can see the sublinear times when the tree sizes are unequal.

Another advantage to the treap algorithms is that they take advantage of the "easiness" of the data distributions. Union, intersection and difference take less time on sets that do not have finely interleaving keys than sets that do, even when they are not the versions using fast splits and joins. Figure 3.5 shows the runtimes on one processor to take the union (without fast splist and joins) of a tree with a million nodes and a tree with 16 thousand nodes for a varying number of blocks. Each tree has equal size blocks such that the k blocks from each each interleave in the set union. The maximum number of blocks is the size of the smaller tree.

Finally, Figure 3.6 shows the times on one processor for union, intersection, and difference. The
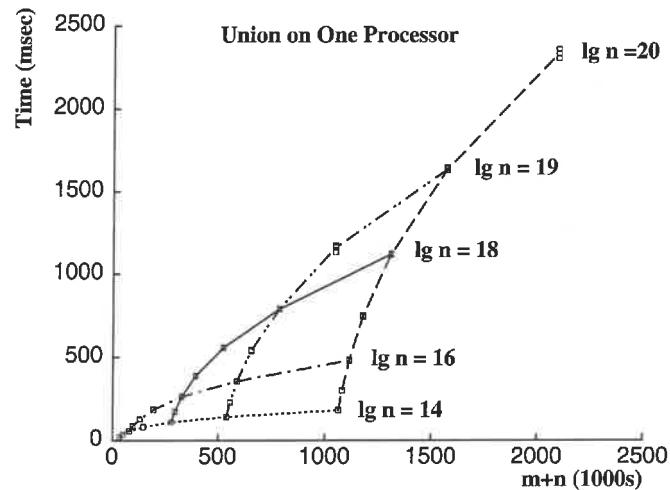
Figure 3.4: The sublinear time to find the union of two treaps of size m and n. Each line specifies the times on one processor of a SUN Ultra Enterprise 3000 for fixed n, as indicated, while m varies.
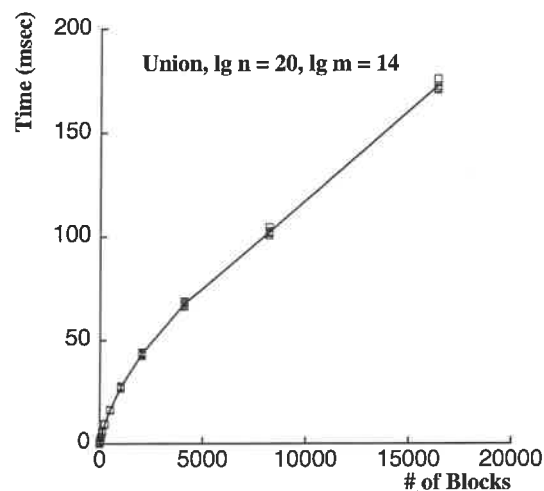
Figure 3.5: Time on one processor of a SUN Ultra Enterprise 3000 to find the union of a treap of size 1M nodes ($\lg n = 20$) and a treap of size 16K nodes ($\lg m = 14$) for a varying number of blocks.

keys are from two uniform random distributions of the same range. Again we see the sublinear times similar to those for union.

3.4.2 Parallel experiments

The parallel versions of the treap operations are simple extensions to the C code in which it spawns threads for the two recursive calls. In Cilk this requires adding the `cilk` keyword before each function definition, the `spawn` keyword before each recursive call, and the `sync` keyword after the pair of recursive calls so that the parent thread waits for both calls to complete before continuing. As discussed below, I also use my own memory management and stop making parallel calls when reaching a given depth in the tree.

In my first effort to implement the parallel algorithms in Cilk, I used nonpersistent versions and relied on the Cilk memory-management system to allocate and deallocate the nodes of the trees. The results were disappointing, especially for the difference operation where times were worse on more processors than on fewer processors. The slow down was due to two factors. One factor was
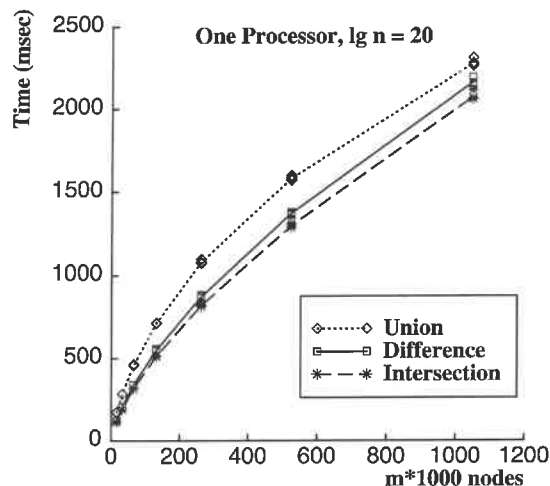
Figure 3.6: Time on one processor of a SUN Ultra Enterprise 3000 to take the union of a treap of size 1M ($\lg n = 20$) with a treap of size m and the intersection and difference of the result with the second treap in the union. Each treap has keys drawn randomly from the same range of values.

due to small-grain memory allocations/deallocations, the size of a treap node. To ensure that the memory operations are atomic and independent, Cilk uses a memory lock. Because the granularity is small, there was high contention for the memory lock. The second factor was that the cache lines on the SGI are long, 128 bytes (32 words), so that 8 treap nodes share the same cache line. In the nonpersistent version of the code when two processors write to different nodes on the same cache line, called "false sharing", the processors need to exchange the cache line even though they are not sharing nodes. The treap operations result in a large amount of false sharing.

To solve the first problem, I wrote my own memory management system for the tree nodes on top of Cilk's. It allocates tree nodes from a large area of memory allocated by Cilk. It divides this memory into smaller blocks of consecutive memory, and gives each processor one block from which to allocate tree nodes. Every time a processor runs out of nodes in its block, it gets a new block. In this way, a processor only acquires a lock when it needs a new block. This system assumes some form of copying garbage collection (either automatic or called by the user explicitly) since memory is allocated consecutively and never explicitly deallocated. I did not implement a garbage collector for the experiments but I did measure the time to copy a treap and it is very much less than the time to create it. In addition, my memory management is specialized to allocate only one fixed-size space, a tree node. Cilk's memory management is more general and needs to manage allocations of any size.

To solve the false sharing problem I converted my initial nonpersistent implementations into persistent versions. These versions write only to newly allocated nodes within a processor's own block of memory (they never modify an existing node). The cost of persistence varies. Consider the cost on a single processor. For union, which allocates the $O(m \log(n/m))$ new nodes, the persistent version is slower than the nonpersistent version by a modest 9%, when m is small compared to n, and 50%, when $m = n$. When the result of an intersection is small relative to the input sizes, the persistent version is 35% faster than the nonpersistent version. When the result is large the persistent version is 30% slower. For set difference the persistent version is 12%–23% faster than the nonpersistent version. For multiple processors the speedup is consistently better for the persistent version. For example, for intersection on 8 processors of the SGI Power Challenge the speedup is about 6.5 for the persistent version and 5.0 for the nonpersistent version.

Another improvement I made was to revert to sequential code after spawning threads to a certain recursive call depth. That is, every time the execution of the code applies another recursive call, it decrements a counter. Once the counter reaches zero it applies a recursive call to a non-Cilk procedure that only makes calls to C procedures. The C procedure is exactly like the Cilk one except it does not use Spawn and Sync. In this way, the C procedure avoids the overhead of the Cilk function call, which is about three times the overhead of a C function call [107]. In Figure 3.7 I show the times for union when I vary the depth at which the code reverts to C. Notice that as the depth decreases the times improve and then get worse. If it reverts to C too soon, the problem size on which the threads have to work can vary greatly and there are not enough threads to get sufficient load balancing. The run times for different data sets, therefore, vary quite widely. For larger depths, the execution incurs more Cilk overhead; the run times over different data sets, however, are more consistent than for smaller call depths.
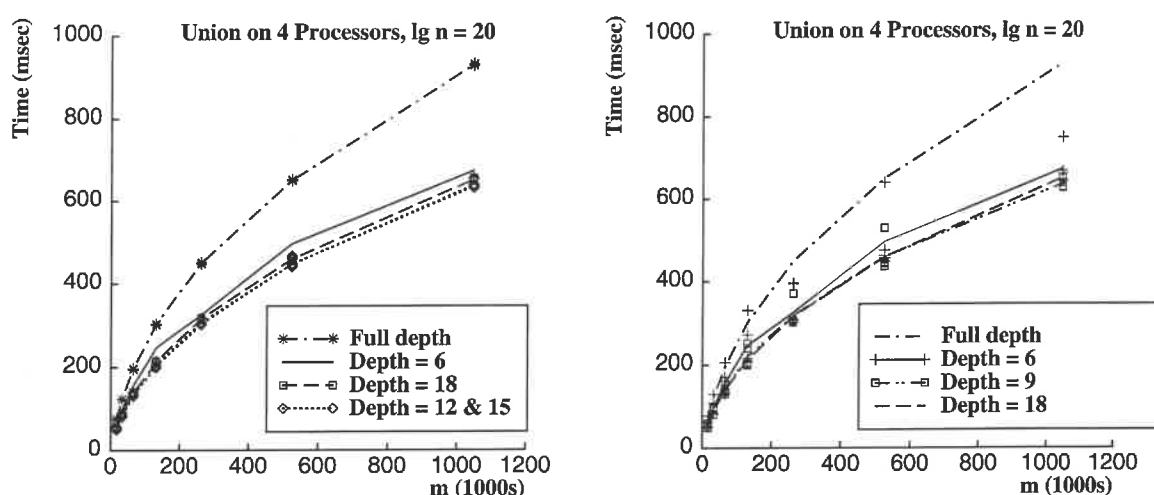
Figure 3.7:   Times on four processors of a SUN Ultra Enterprise 3000 to find the union of a treap of size 1M ($\lg n = 20$) with a treap of size m. Each line shows a different call depth at which the Cilk run reverted to sequential code. The left graph shows that the times for different data sets do not vary much when the depths are large. The right graph shows that the times for different data sets vary quite widely when the depth is 6 and 9.

Finally, Figure 3.8 shows speedups for up to 5 processors of the SUN Ultra Enterprise 3000 and up to 8 processors of the SGI Power Challenge. The times are for finding the union of two 1-million node treaps, and finding the intersection and difference of the result treap with one of the input treaps to union. The speedups are between a factor of 4.1 and 4.4 on 5 processors of the Sun and a factor of 6.3 and 6.8 on 8 processors of the SGI, which is quite reasonable considering the high bandwidth requirements of the operations.

3.5   Discussion

I considered parallelizing a variety of balanced trees (and skip lists) for operations on ordered sets. I selected treaps because they are simple, fully persistent (if implemented appropriately), and easy to parallelize. It is hard to make a definitive argument, however, that one data structure is simpler than another or that the "constant factors" in runtime are less since it can be very implementation and machine dependent. I therefore supply the code and experiments as data points. In terms of asymptotic bounds I believe I present the first parallel algorithms for set operations that run in
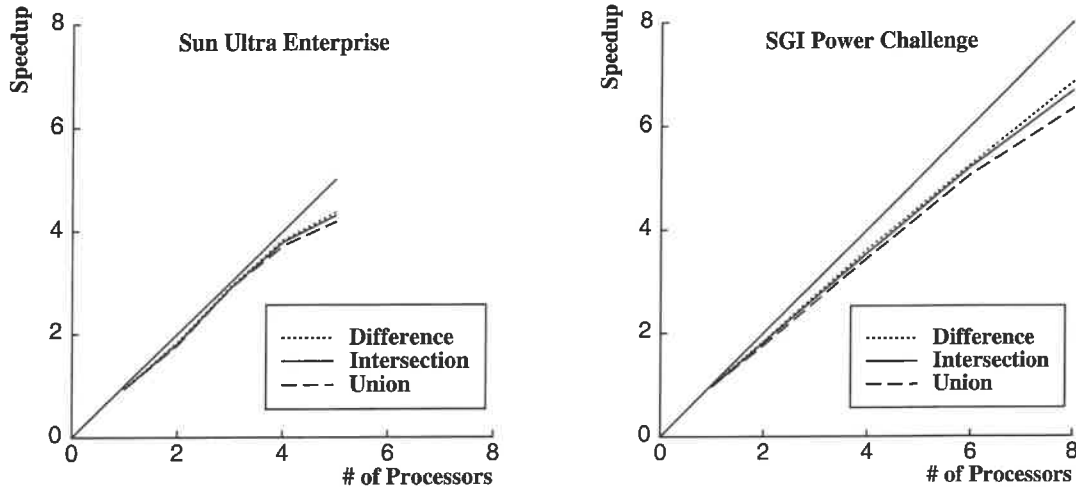
Figure 3.8: Speedup for the union of a 1M node treap with a 1M node treap, and the intersection and difference of the result with the same second 1M node treap.

$O(m \log((n+m)/m)))$ expected work and polylogarithmic depth, and similarly for the block metric with k blocks the first parallel algorithm that runs in $O(k \log((n+m)/k)))$ expected work.

I finish by briefly describing a parallel union algorithm for skip-lists that uses an approach similar to my algorithm on treaps. Recall that in a skip list every element is assigned a height $h \geq 1$ with probability 2^{-h}, and has h pointers $p_i, 1 \leq i \leq h$ which point to the next element in sorted order with height $\geq i.$[§] To merge two sets represented as skip lists, pick the set with the greater maximum height (or an arbitrary one if the heights are the same) and call this A and the other set B. Split A and B using the elements of height h in A, recurse in parallel on the split regions, and join the results. For two sets of size n and m this algorithm has expected parallel depth $O(\lg n \lg m)$. With the appropriate extensions (including back pointers) splits and joins can be implemented to run with the same bounds as fast joins and splits in treaps. Using a similar argument as used in Theorem 3.9 I conjecture that the total expected work for this union with skip lists is $O(k \log((n+m)/k))$ with k being the number of blocks. The main disadvantage with skip lists is that they are much more difficult to make persistent.

[§]More generally for a parameter $0 < p < 1$, the probability of being assigned height h is $(1-p)p^{h-1}$.

Chapter 4

Pipelining with Futures

In this chapter I show how the depth of the algorithms in the previous chapter can be reduced by a log factor, by using pipelining.* The approach, however, is more general and can be applied to a variety of algorithms. More importantly, the pipelining model allows the pipeline to be managed automatically. Managing the pipeline does not have to be coded explicitly, making the code much simpler. A runtime system schedules and manages the pipeline implicitly.

4.1 Introduction

Pipelining in parallel algorithms takes a sequence of tasks each with a sequence of steps and overlaps in time the execution of steps from different tasks. Due to dependences between the tasks or the required resources, pipelined algorithms are designed such that each task is some number of steps ahead of the task following it. Pipelining has been used to improve the time of many parallel algorithms for shared-memory models. Paul, Vishkin and Wagener described a pipelined algorithm for inserting m new keys into a balanced 2-3 tree with n keys [92]. They first considered a nonpipelined algorithm that has $O(\log m)$ tasks, each of which takes $O(\log n)$ parallel time (steps), for a total time of $O(\log n \log m)$ on an EREW PRAM. Each task works its way up from the bottom of the insertion tree to the top, one level at a time. They then showed how to reduce the time to $O(\log m + \log n)$ by pipelining the tasks through the tree. The idea is that when task i is working on level j of the tree, task $i + 1$ can work on level $j - 1$, and so on.

Cole used a similar idea to develop the first $O(\log n)$ time PRAM sorting algorithm that was not based on the AKS sorting network [37]; the AKS sorting network [3] has very large constants and is therefore considered impractical. The algorithm is based on parallel mergesort, and it uses a parallel merge that takes $O(\log n)$ time. The natural implementation would therefore take $O(\log^2 n)$ time—the depth of the mergesort recursion tree is $O(\log n)$ and the merge task at level i from the top takes $O(\log n - i)$ time. Cole showed, however, that the merge tasks can be pipelined up the recursion tree so that each merge can pass partial results to the node above it before it completes, and that this leads to a work-efficient algorithm that takes $O(\log n)$ time. The basic idea of Cole's mergesort was later used in a technique called cascading divide-and-conquer, which improved the time of many computational geometry algorithms [8].

Although pipelining has lead to theoretical improvements in algorithms, from a practical point of view pipelining can be very cumbersome for the programmer—managing the pipeline involves careful timing among the pipeline tasks and assumes a highly synchronous model. The central idea

---

*This chapter is in large part taken from [18]

56

is to show that many algorithms can be automatically pipelined using futures, a construct designed for parallel languages [46, 12]. Using futures, coding the pipelined algorithms is remarkably simple; we push the complexity of managing the pipeline and scheduling the threads to a single provably-efficient runtime system. In addition, my approach is the first that addresses asynchronous pipelined algorithms where the pipeline depth is dynamic and depends on the input data. I present and analyze several algorithms that require such an asynchronous pipeline. The approach also gives a natural way to restrict algorithms so they have no concurrent memory accesses.

The futures construct was developed in the late 70s for expressing parallelism in programming languages and has been included in several programming languages [56, 73, 32, 35, 33]. Conceptually, the *future* construct forks a new thread t_1 to calculate a value (evaluate an expression) and immediately returns a pointer to where the result of t_1 will be written. This pointer can then be passed to other threads. When a thread t_2 needs the result of t_1, it uses its pointer to request the value. If the value is ready (has been written) it is returned immediately, otherwise t_2 waits until the value is ready. To avoid deadlocks and for efficiency t_2 is typically suspended while waiting so that other threads can run.

To analyze the running time of algorithms programmed with futures I use a two step process. I first consider a language-based cost model based on futures and analyze the algorithms in this model. I then show universal bounds for efficiently implementing the model on various machine models. The language-based model uses a slight variation of the PSL model [52]. In this model computations are viewed as dynamically unfolding directed acyclic graphs (DAGs), where each node is a unit of computation (action) and each edge between nodes represents a dependence implied by the language. There are three types of dependence edges in the DAG, *thread edges* between two successive actions in a thread, *fork edges* from the node that creates a future to the first node of the future's thread, and *data edges* from the result of a future to all the nodes that request the result. The cost of a computation is then calculated in terms of total *work* (number of nodes in the DAG) and the *depth* (longest path length in the DAG). Analyzing an algorithm in the model involves determining the work and depth of the algorithm as a function of the input size.

As an example of the use of futures and of the DAG cost model consider Figure 4.1. This example has a producer that produces a list of decreasing integers from n down to 0, where each cons cell is created by its own thread. In parallel, a consumer consumes these values by summing them. This code pipelines producing and consuming the values.

I describe and analyze four algorithms with the language-based model. The first is a merging algorithm. It takes two binary trees with the keys sorted in-order within each tree and merges them into a single tree sorted in-order. The code is very simple and, assuming both input trees are of size n, the nonpipelined parallel version requires $O(\log^2 n)$ depth and $O(n)$ work. I show that, by using the same code but implementing it with futures, the depth is reduced to $O(\log n)$, which meets previous depth bounds. The next two algorithms use a parallel implementation of the treap data structure [103]. I show randomized algorithms for finding the union and difference of two treaps of size m and $n, m \leq n$ in $O(\log n + \log m)$ expected depth and $O(m \log(n/m))$ expected work. Like the merge algorithm, the code is very simple. There are no previous parallel or pipelined results for treaps of which I am aware. The fourth algorithm is a variant of Paul, Vishkin and Wagener's (PVW) 2-3 trees [92]. As the bottom-up insertion used in the PVW algorithm does not map naturally into the use of futures, I describe a top-down variant that does. As with the PVW, algorithm the pipelining improves the algorithm complexity for inserting m keys into a tree of size n from $O(\log n \log m)$ to $O(\log n + \log m)$ depth. In both cases the work is $O(m \log n)$.

To complete the analysis I next consider implementations of the language-based model on various machines. The work and depth costs along with Brent's scheduling principle [29] imply

```
datatype list = cons of int*list | null;

fun produce(n) =
  if (n < 0) then null
  else cons(n,?produce(n-1));

fun consume(sum,null) = sum
  | consume(sum,cons(a,l)) = consume(a+sum,l);

consume(0,?produce(n));
```
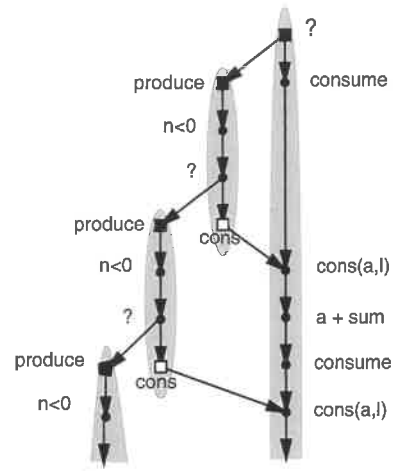

Figure 4.1:   Example code and the top of the corresponding computation DAG. The code syntax is based on ML described at the end of this chapter. Futures are marked with a question mark (?). Each node represents an action and each vertical sequence of actions represents a thread. The vertical edges are thread edges, the edges going to the left are fork edges and the edges going to the right are data edges. The `cons(a,l)` in the `consume` code refers to pulling apart the `cons` cell into its two components, `a` and `l`.

that, given a computation with depth d and work w, there is a schedule of actions onto processors such that the computation will run in $w/p + d$ time on a p processor PRAM. This principle, however, does not tell us how to find the schedule online—in particular it does not address the costs of dynamically assigning threads to processors nor the cost of handling the suspension and restarting required by futures at runtime. Since many of the algorithms are dynamic, the schedule cannot be computed offline. In addition, Brent's scheduling principle in general assumes concurrent memory access, requiring an implementation on a CRCW PRAM. The two key points are that all the scheduling and managing of futures can be handled by a runtime system in an algorithm-independent fashion with provable time bounds, and that by placing a restriction on the program type, we can guarantee the computation will require no concurrent memory accesses. The universal results place bounds on the time taken by an implementation on various machine models, including all online costs for scheduling and management of futures.

Previous results on implementing a model similar to the one use here [52] have shown that any computation with w work and d depth can be implemented online on an CRCW PRAM in $O(w/p + d \cdot T_f(p))$ time, where $T_f(p)$ is the time for a fetch-and-add (or multiprefix) on p processors. The fetch-and-add is used to manage queues for threads that are suspended waiting for a future to complete. For programs that are converted to a form called *linear code*, any computation can be implemented on the EREW PRAM model in $O(w/p + d \cdot T_s(p))$ time, where $T_s(p)$ is the time for a scan operation (all-prefix-sums) used for load balancing the tasks. The implementation also implies time bounds of $O(gw/p + d(T_s(p) + L))$ on the BSP [109], where g is the BSP gap parameter and is inversely related to bandwidth and L is the BSP periodicity parameter and is related to latency, $O(w/p + d \log p)$ on an Asynchronous EREW PRAM [42], and $O(w/p + d)$ on the EREW Scan model [19]. The conversion to linear code is a simple manipulation that can be done by a compiler. Although this conversion can potentially increase the work and/or depth of a computation, it does not for any of the algorithms described herein. In fact, linear code seems to be a natural way to define EREW algorithms in the context of a language model.

When mapping algorithms onto a PRAM, this approach loses some time over previous pipelined algorithms. For example, the $O(\log n)$ depth, $O(m \log n)$ work 2-3 tree algorithm mapped onto the PRAM becomes an $O(m \log n/p + \log n \cdot T_s(p))$ time as opposed to $O(m \log n/p + \log n)$ time for the PVW algorithm. Note, however, that when mapped directly onto more realistic models, such as the network models or the asynchronous PRAM, the algorithms perform equally well as the PRAM algorithms and with much simpler code: In the more realistic models, compaction using prefix sums has the same latency as either the memory read or write (network models) or the synchronization between steps (asynchronous PRAM). Furthermore, the approach can easily handle dynamic pipelines in which the structure and delays of the pipeline depends on the input data, such as the treap algorithms we describe. This would be considerably difficult to do by hand and I know of no previous PRAM algorithms with dynamic pipelines.

4.2 The Model

As with the work of Blumofe and Leiserson [27, 28], I model a computation as a set of threads and the cost as the size of the computation DAG. Threads can fork new threads using a future, and can synchronize by requesting a value written by another thread. A computation begins with a single thread and completes when all threads have terminated.

A *future* call in a thread t_1 starts a new thread t_2 to calculate one or more values and allocates a *future cell* for each of these values.[†] The thread t_1 is passed *read pointers* to each future cell and continues immediately. These read pointers can be copied and passed around to other threads, and at any point any thread that has a pointer can read its value. The thread t_2 is passed *write pointers* to each future cell, which is where the results values are to be written as they are computed. The write pointers can also be passed around to other threads, but each can only be written to once. When a thread reads the value from a read pointer, sometimes called a *touch operation*, it must wait until the write to the corresponding cell has completed. As discussed in Section 4.4, the read is implemented by suspending the reading thread and reactivating it when the write occurs. Note that, although a future cell can be written to at most once, in general it can be read from multiple times. In Section 4.4 I show that when the code meets a certain condition called linearity the future cell is read at most once.

To specify when it is necessary to read from a read pointer I distinguish between strict and nonstrict operations. An operation is *strict* on an argument if it needs to know the value of that argument immediately. For example, all the arithmetic operations are strict on their arguments, and an operation that extracts an element from a cell is strict on that cell. An operation is *nonstrict* on an argument if it does not need to know the value of that argument immediately. For example, passing an object to a user-defined function or placing an object in a cell are nonstrict because the actual value is not needed immediately and a pointer to the value can be used instead. Whenever an operation is strict on an argument and that argument is a read pointer to a future cell, executing the operation will invoke a read on that future cell. I also assume that writing to a future cell is strict on the value that is being written. This means that a read pointer cannot be written into a future cell, which prevents chains of future cells. This restriction is important for proving bounds on the implementation.

Note that when building a data structure out of multiple cells, such as in a linked list or tree, operations are strict on the individual cells, not on the whole data structure. For example, if an

---

[†]The ability to return multiple values and have separate future cells created for a single fork is actually quite important for some of the algorithms I present.

operation examines the head of a linked list to get a pointer to the second element, the operation is strict on the head but not the second or any other element. I will make significant use of this property in the algorithms in this chapter.

To describe the algorithms, I use a subset of ML [88] extended with futures. The syntax is defined at the end of this chapter (see Figure 4.12). The subset I use is purely functional (no side effects), and I use arrays only for the 2-6 tree algorithm described in Section 4.3.4 and otherwise I just use trees. Futures are created by placing a ? (question mark) before an expression, which will create a thread to evaluate the expression. The number of variables in an ML pattern determines the number of futures that an expression creates. I make significant use of the ML pattern matching capabilities, and have, therefore, included a quick description in the appendix.

I now consider the DAGs that correspond to computations in the model. The DAGs are generated dynamically as the computation proceeds and can be thought of as a trace of the computation. Each node in a DAG represents a unit-time action (the execution of a single instruction) and the edges represent dependencies among the actions. As mentioned in the introduction, there are three kinds of dependence edges in the DAGs: thread edges, fork edges, and data edges. A thread is modeled as a sequence of actions connected by *thread edges*. When an action a_1 within a thread uses a future to start a thread t_2, a *fork edge* is placed from a_1 to the first action in t_2. When an action a_1 reads from a future-cell, a *data edge* is placed from the action a_2 that writes to that cell to a_1. The cost of a computation is then measured in terms of the number of nodes in the DAG, called the *work*, and the longest path length in the DAG, called the *depth*. In analyzing algorithms the goal is to determine the work and depth in terms of the input size. Determining the work is often simple since it is the time a computation would take sequentially if futures were not used. Determining the depth can be more difficult. As an aid I will refer to the *time stamp* of a value as the depth in the DAG at which it is computed, and will then find upper bounds on the time stamps of the results to determine the depth of the computation.

The model, as defined here, is basically the PSL (Parallel Speculative λ-Calculus) [52], augmented with arrays as in NESL [23]. Although the PSL only considered the pure λ-Calculus with arithmetic operations, the syntactic sugar I include only affects work and depth by a constant factor. I am actually assuming a slightly simplified model by considering only a first-order language (it cannot pass functions) since I do not need the more general case. I also explicitly mark where futures are to be created, while in the PSL model all expressions are implicitly made into futures.

4.3 Pipelining Applications

In this section I show four applications that use pipelining to reduce the depth of the algorithms. The first three applications require a dynamic pipeline because the time at which data becomes available for the next task in the pipeline varied from task to task. The last application is sychronous and the pipeline depth can be fixed. For each application I give the parallel algorithm, explain how to modify the algorithm to pipeline the computation, and give an analysis of the depth.

4.3.1 Merging binary trees

The first algorithm I discuss is a simple divide-and-conquer algorithm that takes two binary trees T_1 and T_2, where the keys in each tree are unique and sorted when traversed in-order, and merges them into a new sorted binary tree, T_m. The code is shown in Figure 4.2. The function $\text{split}(s, T)$ splits a tree T into two trees, one with keys less than the splitter s and one with keys greater than or equal to s. The function traverses a path down to a leaf, separating subtrees based on the

```
1   datatype tree = node of int*tree*tree | empty;

2   fun split(s,empty) = (empty,empty)
3     | split(s,node(v,L,R)) =
4       if s < v then
5         let val (L1,R1) = ?split(s,L)
6         in (L1,node(v,R1,R))
7         end
8       else
9         let val (L1,R1) = ?split(s,R)
10        in (node(v,L,L1),R1)
11        end;

12  fun merge(empty,T2) = T2
13    | merge(T1,empty) = T1
14    | merge(node(v,L,R),T2) =
15      let val (L2,R2) = ?split(v,T2)
16      in node(v,?merge(L,L1), ?merge(R,R1))
17      end;
```
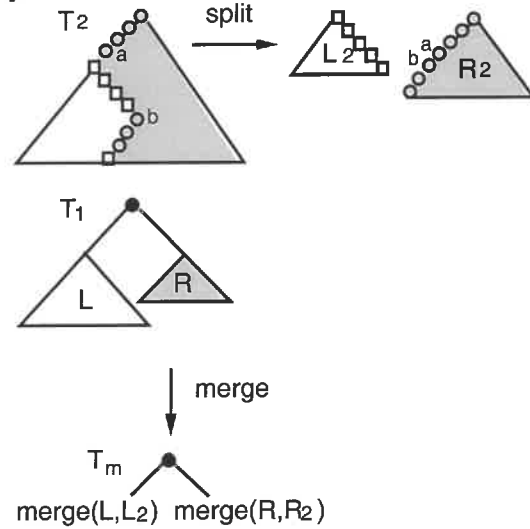
Figure 4.2: Code for merging two binary search trees and a corresponding figure. The shaded regions are keys that are greater than the key at the root of T_1.

splitter to form the two result trees (see Figure 4.2). It requires work that is at most proportional to the depth of the tree. The function `merge` makes the root of T_1 the root of the result tree T_m and splits T_2 by the key at the root of T_1. It then calls `merge` recursively twice to make the left and right subtrees.

The code is a natural sequential implementation for merging two binary trees, if we exclude the futures. Futures provide two forms of parallelism. First, they provide parallelism by allowing the two recursive `merge` functions to execute in parallel. If T_1 is balanced and of size n then the `merge` will be called recursively to a depth of $O(\log n)$. If T_2 is also balanced and of size m then the split operation has $O(\log m)$ depth. Therefore, the overall depth of the algorithm is easily bounded by $O(\log n \log m)$. Second, and more importantly, futures provide pipelining by allowing the partial results of `split` (i.e., nodes higher in the tree) to be fed into the two `merge` calls, thereby allowing for the overlap in time of multiple split calls at different levels of the recursion tree. With such pipelining `merge` has depth $O(\log n + \log m)$.

To appreciate this claim, let us consider the time (depth in the DAG) at which all nodes of the result trees, $(L_2, R_2) = \mathtt{split}(v, T_2)$, are computed. If the roots of both L_2 and R_2 are created in constant time, and each child at a constant time after its parent, it is not hard to see that the algorithm would pipeline within $O(\log n + \log m)$ depth. The problem, however, is that one root may only be ready after a considerable delay. For example, in Figure 4.2 the root of L_2 is ready only after traversing five nodes in T_2. In addition, there may be further delays at lower levels of the tree. For example, there is a delay going from node a to node b in R_2; b is created only after four nodes of L_2 have been created. In general, the rightmost path of L_2 and the leftmost path of R_2 are made from the nodes of T_2 that `split` traversed, and the time stamp for a node in these paths is proportional to its depth in T_2. These delays can accumulate when one split is pipelined into the next. To prove the bounds, however, I show that when there is a delay there is a corresponding decrease in the depth of the result tree.

Theorem 4.1 *Merging two balanced binary trees of size n and m, $m < n$, with keys sorted in-order takes $O(\log n + \log m)$ depth and $O(m \log(n/m))$ work.*

Proof.   Given in the next section. It is a simplification of the proof for taking the union of two treaps.   ∎

A problem with the merge algorithm described is that even though the input trees may be balanced the resulting merge tree may have depth up to $\log n + \log m$. I now briefly describe how using pipelining, again, the unbalanced result can be balanced with $O(\log n + \log m)$ depth and $O(n + m)$ work. First, the algorithm makes a pass through the tree computing the size of every subtree, which it stores at the root of the subtree. From the size data it next finds the rank of each node (its inorder index). Both steps take $O(\log n + \log m)$ depth and $O(n + m)$ work and do not require pipelining. Next, it rebalances the tree using a parallel pipelined algorithm similar to `merge`. But this time it uses a split operation (similar to `splitm` in the next section) that takes a rank argument and splits the tree into nodes with rank less than the argument and nodes with rank greater than the argument. It returns these two trees along with the node with equal rank. The rebalancing algorithm takes four arguments: a tree, a rank, and the number of lesser and the number of greater rank nodes in the tree. It calls this split operation on the tree and the rank. It uses the node returned by the split operation as the root and then recursively balances the two subtrees. The recursive call for the left (right) subtree supplies a rank that is the old rank minus (plus) half the lesser (greater) subtree size. The analysis of the depth of the algorithm is similar to the analysis of union in the next section.

4.3.2   Treap Union

Treaps [103] are balanced search trees that provide for search, insertion, and deletion of keys and can be used for maintaining a dynamic dictionary. Associated with each key in a treap is a random priority value. The keys are maintained in-order and the priority values are maintained in heap order, thus the name treap. The key with the highest priority is the root of the treap. Because the priorities are random, this key is a randomly chosen key. Similar to quicksort recursion depth, treaps, therefore, have an expected depth of $O(\log n)$ for a tree with n keys. Treaps have the advantage over other balanced tree techniques in that they allow for simple and efficient union. As we will see, they have the added advantage that it is easy to parallelize them.

I present two pipelined parallel operations on treaps—a *union* operation that takes the union of two treaps and can be used to insert a set of keys into a treap; and a *difference* operation that removes the values in one treap from another and can be used to delete a set of keys. Figure 4.3 shows the code for finding the union of two treaps. It is similar to merge in the previous section except that it removes any duplicate values and maintains the treap conditions so that the result treap is balanced. It uses a modified `split` operation, `splitm`, where the splitter can be a key in the treap. When the splitter is in the treap, `splitm` excludes it from the resulting treaps and returns it along with the two split treaps. Otherwise, it simply returns the two resulting treaps. Notice that `splitm` completes as soon as it finds the splitter in the treap.

To maintain the heap order `union` makes the root with the largest priority the root of the result treap (compare with merge, which always uses the root of the first tree). To maintain the keys in-order `union` splits the treaps by the key value of the new root. For one treap these are trivially the left and right children of the root. For the other treap the algorithm uses `splitm` with futures. It then recursively finds the union of the two treaps that have keys less than the root, and finds the union of the two treaps that have keys greater than the root. I show that the expected depth to

```
1   datatype treap = node of real*int*treap*treap
2                        | empty;
3   datatype midval = mid of int | none;

4   fun splitm(v1,empty) = (empty,none,empty)
5     | splitm(s,node(p,v,l,r)) =
6       if s=v then (l,mid(v),r)
7       else if s<v then
8         let val (l1,m,r1) = ?splitm(s,l)
9         in (l1,m,node(p,v,r1,r))
10        end
11      else
12        let val (l1,m,r1) = ?splitm(s,r)
13        in (node(p,v,l,l1),m,r1)
14        end;

15  fun union(empty,b) = b
16    | union(a,empty) = a
17    | union(node(p1,k1,l1,r1), node(p2,k2,l2,r2)) =
18      if p1 > p2 then
19        let val
20          (l,m,r) = ?splitm(k1,node(p2,k2,l2,r2))
21        in node(p1, k1, ?union(l1,l), ?union(r1,r))
22        end
23      else
24        let val
25          (l,m,r) = ?splitm(k2,node(p1,k1,l1,r1))
26        in node(p2, k2, ?union(l,l2), ?union(r,r2))
27        end;
```

Figure 4.3: Code for treap union

find the union of two treaps of size n and m is $O(\log n + \log m)$. Without pipelining the expected depth would be $O(\log n \log m)$.

To analyze the depth of the algorithm let us consider time stamps $t(v)$ for each node v of a tree. The *time stamp* of a node is the depth in the DAG at which the node is created. For a tree T I use the notation $v \in T$ to be a node in T, $h(v)$ to indicate the height of the subtree rooted at the node v (longest path length to any of its leaves), and $l(v)$ and $r(v)$ to indicate the left and right children of the node v, respectively. I use $t(T), h(T), l(T), r(T)$ to mean $t(v), h(v), l(v), r(v)$, respectively, where v is the root of T.

Definition 1 *A τ-value is valid if, for all $v \in T$, $t(v) \le \tau + k_s(h(T) - h(v))$, where k_s is a constant.*

A *τ-value* of a tree is some value that places an upper bound on each of the time stamps in the tree depending on the height of the subtree at the node. This definition means that $\tau \ge \max_{v \in T}\{t(v) - k_s(h(T) - h(v))\}$. These τ-values capture a relationship between the height of subtrees and their time stamps which is important for the proofs of the time bounds. Notice, for example, that a τ-value places the same upper bound on the time stamps for all leaves in the tree regardless of how far down they are in the tree. In the following theorem I show that, for each result treap of `splitm`, we can find a valid τ-value that depends only on the result treap height,

the input treap height, and the input treap's τ-value. In the analysis of **union** I keep track of the τ-values of the input treaps to recursive calls to bound the time stamps in these treaps.

Property 4.2 *If τ is a valid τ-value for a tree T, then a valid τ-value for a subtree T' is*

$$\tau + k_s(h(T) - h(T'))$$

Property 4.3 *If τ_l and τ_r are valid τ-values for $l(T)$ and $r(T)$, respectively then a valid τ-value for T is*

$$\max\{t(T), \tau_l - k_s, \tau_r - k_s\}$$

Lemma 4.4 (Splitm τ-values) *Consider any split value s and any treap T with associated τ-value τ and let k_s be the time between two successive recursive calls to* splitm. *If we call the* splitm(s, T) *function at a time t then, for each of the two results $T' \in \{L', R'\}$, a valid τ-value for T' is $\tau' = \max\{t, \tau\} + k_s(1 + h(T) - h(T'))$.*
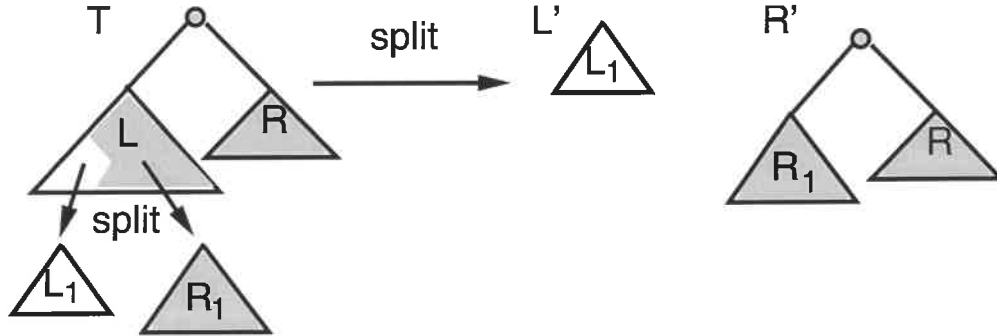
Figure 4.4:   Split of treap T into L' and R'. The shaded areas are keys that are greater than the splitter.

Proof.   Assume that the splitter does not appear in the treap since this is the worst case (if the splitter is found then the split will return earlier). The proof uses induction on the height of the input treap. The lemma is clearly true when $h(T) = 1$. Assume it is true for treaps of height less than or equal to $h-1$. We show it is true when $h(T) = h$. Let $L = l(T)$ and $R = r(T)$. Without loss of generality, assume that s is less than the key at the root of T, and let $(L_1, R_1) = $ splitm(s, L) (see Figure 4.4). First, find a valid τ-value for the greater than result treap, R', by finding the time stamps for all its nodes. Consider the root of R'. Once the root of T is available **union** can obtain R and L, which may be futures, compare the key at the root with s, and call splitm, which returns immediately since it returns three futures. Thus, **union** has all the information needed to create the root of R' in constant time, k_s, and $t(R') = \max\{t, t(T)\} + k_s$. Because $r(R') = R$, a valid τ-value for $r(R')$ is $\tau + k_s(h(T) - h(r(R')))$ by property 4.2. Next we find upper bounds of the times in L_1 and R_1.

The recursive call to splitm on L can be called at time $\max\{t, \tau\} + k_s$ and, by property 4.2, a valid τ-value for L is $\tau + k_s(h(T) - h(L))$. Therefore, by the induction hypothesis a valid τ-value τ'' for the resulting treap $T'' \in (L_1, R_1)$ is

$$\begin{aligned}
\tau'' &= \max\{\max\{t, \tau\} + k_s, \tau + k_s(h(T) - h(L))\} + k_s(1 + h(L) - h(T'')) \\
&\leq \max\{t, \tau\} + k_s(1 + h(T) - h(T''))
\end{aligned} \tag{4.1}$$

Since $l(R') = R_1$ and by property 4.3, a valid τ-value for R' is

$$\begin{aligned}
\tau' &= \max\{\max\{t, t(T)\} + k_s, \tau + k_s(h(T) - h(r(R')) - 1), \max\{t, \tau\} + k_s(h(T) - h(l(R')))\} \\
&\leq \max\{t, \tau\} + k_s(1 + h(T) - h(R'))
\end{aligned}$$

Finally, since $L' = L_1$, a τ-value of L' is a τ-value for L_1 as given in Equation 4.1. ∎

Note that **union** only creates new treaps by dividing a treap into its left and right children or by running the **splitm** operation on it. Given the above lemma, we can find τ-values for the treaps in all the recursive calls, and use these τ-values to find upper bounds $\hat{t}(v)$ for $t(v)$, the time stamps on the nodes v of the union result treap.

Theorem 4.5 (Depth bound on Union) *Consider two treaps T_1 and T_2 with τ-values τ_1 and τ_2. If we call* **union**(T_1, T_2) *at time t, then the maximum time stamp on any of the nodes of the result T_m will be* $\max\{t, \tau_1, \tau_2\} + O(h(T_1) + h(T_2))$.
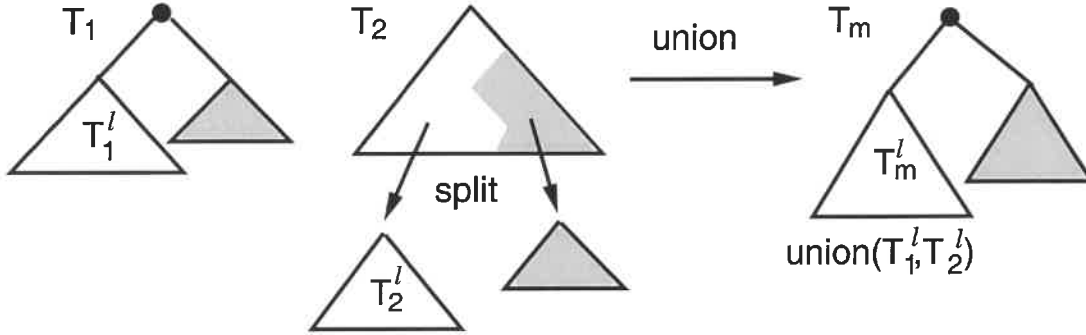

Figure 4.5: Union of treaps T_1 and T_2 into T_m, when the priority at the root of T_1 is greater than the priority at the root of T_2. T_2 is split by k_1, the key at the root of T_1. The subtreap $l(T_m)$ is the union of the subtreaps (not shaded) with keys less than k_1 and the subtreap $r(T_m)$ is the union of the subtreaps (shaded) with keys greater than k_1.

Proof. Once the two roots of T_1 and T_2 are ready, **union** can compare their priorities, start up **splitm** and the two recursive **union**s, and create the root of the result treap T_m with pointers to the futures for its two children. This all takes constant time, k_m, because **splitm** and **union** are called with futures. Thus, $t(T_m) \leq k_m + \max\{t, \tau_1, \tau_2\}$. This upper bound $k_m + \max\{t, \tau_1, \tau_2\}$ on the time stamp of the root of the result treap will be referred to as $\hat{t}(T_m)$.

We now calculate $\hat{t}(l(T_m))$, an upper bound on the time stamp of the left child of the root of the result treap, in terms of $\hat{t}(T_m)$. Consider the two treaps T_1^l and T_2^l, which are the inputs to the left call to **union**, and $T_m^l = l(T_m)$ which is the result of the call. Without loss of generality consider the case when the priority of T_1 is greater than the priority of T_2. Then $T_1^l = l(T_1)$ and T_2^l is the left result of **splitm**(k_1, T_2), where k_1 is the key at the root of T_1, see Figure 4.5. Due to the previous bound on the **splitm** operation, a τ-value for T_2^l is

$$\begin{aligned}
\tau_2^l &= \max\{t(T_m), \tau_2\} + k_s(1 + h(T_2) - h(T_2^l)) \\
&\leq \hat{t}(T_m) + k_s(1 + h(T_2) - h(T_2^l))
\end{aligned}$$

By property 4.2, a τ-value for T_1^l is

$$\begin{aligned}
\tau_1^l &= \tau_1 + k_s(h(T_1) - h(T_1^l)) \\
&< \hat{t}(T_m) + k_s(h(T_1) - h(T_1^l)).
\end{aligned}$$

These, along with the condition at the beginning of the proof, give an upper bound on the time stamp of T_m^l:

$$
\begin{aligned}
t(T_m^l) &\leq k_m + \max\{t(T_m), \tau_1^l, \tau_2^l\} \\
&\leq \hat{t}(T_m) + k_m + k_s \max(h(T_1) - h(T_1^l), 1 + h(T_2) - h(T_2^l)).
\end{aligned}
$$

That is, the only way the bound on the time stamp of a child can be $k_m + \delta \cdot k_s$ more than its parent's bound is by a corresponding height decrease of either δ in the depth of T_1 or $\delta - 1$ in T_2. Because union removes the root of T_1, $\delta \geq 1$. We can show the same bound for $r(T_m)$.

Now consider a path in T_m from the root to a leaf. Let $\Delta_i = \hat{t}(c) - \hat{t}(v)$, where c is a child of v and v is a node at depth $i - 1$. Let $h_j^i, j = 1, 2$ be the height of the input treaps of the union that created c. From the above discussion and $j = 1(2)$ and $k = 2(1)$

$$
\begin{aligned}
\Delta_i &\leq k_m + k_s \max(h_j^{i-1} - h_j^i, 1 + h_k^{i-1} - h_k^i) \quad\quad\quad (4.2) \\
&\leq k_m + k_s(h_1^{i-1} - h_1^i + h_2^{i-1} - h_2^i + 1).
\end{aligned}
$$

Since the algorithm terminates whenever one of the input treaps has height 0, and the height of at least one of the treaps decreases by one for each recursive call, the depth of the recursion treap is at most $O(h(T_1) + h(T_2))$. Therefore, the total increase in the bound on the time stamps along the path to any new node is $\sum \Delta_i \leq (k_m + 2k_s)(h(T_1) + h(T_2))$. Since the time stamp on the root is bound by $k_m + \max\{t, \tau_1, \tau_2\}$ and the path bound is true for all paths, this bounds the time stamp on any new node in T_m by $\max\{t, \tau_1, \tau_2\} + O(h(T_1) + h(T_2))$. The untouched nodes are also clearly similarly bounded.   ∎

Corollary 4.6 (Expected union depth) *The expected depth to find the union two treaps of size n and m is $O(\log n + \log m)$.*

Proof.   We assume that the treaps are "ready" when union is called at time t. That is, the treaps have valid τ-values, τ_1 and τ_2, with $\tau_1 < t$ and $\tau_2 < t$. Since the expected heights of the two treaps is $O(\log n)$ and $O(\log m)$ [103], the expected depth to find the union is $O(\log n + \log m)$.   ∎

I now return to the proof of depth on the merge computation described in the previous section.

Proof.   (of Theorem 4.1) The proof for the depth bound on merge is the same as for the depth bound on union, except that we do not need to consider the case when T_1 is split. Thus, in Equation 4.2, $j = 1$ and $k = 2$. Since $h(T_1) = \log n$ and $h(T_2) = \log m$, to merge the two trees takes $O(\log n + \log m)$ depth. The proof the the work bound for merge is easier than for union because the input trees are balanced. Union requires an expected case analysis.   ∎

4.3.3   Treap Difference

The inverse operation to taking the union of two treaps is taking their difference; remove any keys from the first treap that appear in the second treap. The diff algorithm is, again, quite simple and uses two operations splitm (shown previously in Figure 4.3) and join (shown in Figure 4.6). The join operation is the inverse of split—it takes two treaps, T_1 and T_2, where the largest key in T_1 is less than the smallest key in T_2, and joins them into a single treap, T'. A join only requires $O(h(T_1) + h(T_2))$ work since it need only descend the rightmost path of T_1 and the leftmost path of T_2, interleaving the nodes depending on their associated priorities.

The function diff takes two treaps, T_1 and T_2, and returns a treap T_d which is T_1 with any keys in T_2 removed. First, it calls splitm on T_2 and the key at the root of T_1 as the splitter to

```
 1  fun join(empty,b) = b
 2    | join(a,empty) = a
 3    | join(node(p1,k1,l1,r1),
 4            node(p2,k2,l2,r2)) =
 5      if p1 > p2 then
 6        node(p1,k1,l1,?join(r1,node(p2,k2,l2,r2)))
 7      else
 8        node(p2,k2,?join(node(p1,k1,l1,r1),l2),r2);

 9  fun diff(empty,b) = empty
10    | diff(a,empty) = a
11    | diff(node(p1,k1,l1,r1),t2) =
12      let val (l2,m,r2) = ?splitm(k1,t2);
13          val l = ?diff(l1,l2);
14          val r = ?diff(r1,r2);
15      in if m = none then node(p1,k1,l,r)
16          else ?join(l,r)
17      end;
```

Figure 4.6: Code for taking the difference of two treaps.

obtain two treaps, l_2 and r_2, and possibly the splitter. Next, `diff` recursively finds the difference of $l(T_1)$ and l_2 and the difference of $r(T_1)$ and r_2. If the root key of T_1 was not in T_2 the results of the recursive calls become the left and right branches of the root. Otherwise, the root and its subtreap is replaced by the join of the two treaps resulting from the recursive calls. As in `union`, without pipelining it takes $O(h(T_1)h(T_2))$ depth to descend to the bottom of the recursion call tree. On the way back up, a path may contain as many as $\min(h(T_1), m)$ nodes to delete, where m is the size of T_2. Each such node can add $O(h(T_d))$ depth due to the required `join`. Thus, the overall depth for `diff` not considering pipelining is $O((h(T_1)h(T_2) + h(T_d) \min(h(T_1), m))$.

The pipelining for `diff` is notably different from the pipelining for `union` because the algorithm requires work after the recursive calls (the join) as well as before them (the split). Interestingly, if we make the algorithm tail recursive by doing the join before the recursive call, the algorithm does not pipeline (besides having an increased work measure) because the join can not take place until after the split is complete. The pipelining while descending T_1 is much like the tree `merge`, except no actual merging takes place, and therefore that part of the computation DAG has $O(h(T_1) + h(T_2))$ depth. I next show that the ascending phase of the algorithm takes $O(h(T_1) + h(T_d))$ depth. First I show the worse-case time stamps on the results of a `join`. Then, I show the worse case time stamps on the final result treap. I use the same definitions as in Section 4.3.2, except I replace τ-values with a similar concept of ρ-values.

Definition 2 *Let $d_T(v)$ of a node $v \in T$ be the depth of the node in the tree, such that the $d_T(T) = 0, d_T(l(T)) = d_T(r(T)) = 1, \ldots$. A ρ-value is valid for a tree T if, for all $v \in T$, $t(v) \leq \rho + k\,d_T(v)$, where k is a constant.*

That is, a *valid ρ-value* for a tree T defines upper bounds for the time-stamps of the tree, namely for all $v \in T$, $t(v) \leq \rho + k\,d_T(v)$, where k is a constant. In contrast to τ-values, ρ-values are independent of the heights of the subtrees.

Property 4.7 *If ρ is a valid ρ-value for T then ρ is a valid τ-value for T.*

Property 4.8 *If τ is a valid τ-value for T then $\tau + kh(T) - 2$ is a valid ρ-value for T.*
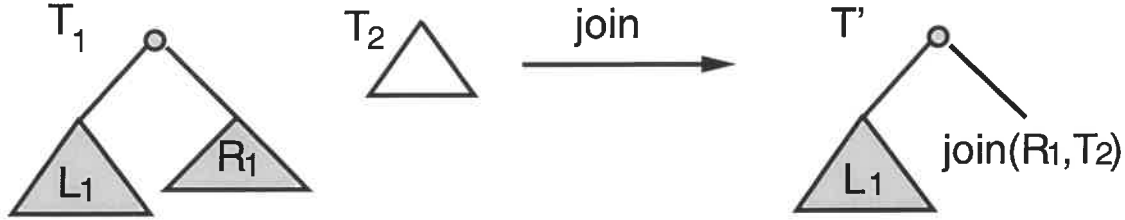

Figure 4.7: Join of treaps T_1 and T_2 into T', when the priority at the root of T_1 is greater than the priority at the root of T_2.

Lemma 4.9 (Join ρ-values) *If* join *is called at time t on two treaps T_1 and T_2 with valid ρ-values ρ_1 and ρ_2, then a valid ρ-value for the resulting joined treap T' is $\rho' = \max\{t, \rho_1, \rho_2\} + k$, where k is a constant at least as large as the maximum computation DAG depth between successive recursive calls to* join.

Proof. We find upper bounds of the time stamps of each node of the T' by induction on the size of T'. Let n be the size of T'. The lemma is clearly true when the size of the result treap is 1. Assume it is true for result treaps of size $n-1$. We show it is true for result treaps of size n. Since join can test the root priorities, receive a pointer to the future which is the result of the recursive call to join, and create the root node of T' in constant depth k, once the roots of T_1 and T_2 are ready, $t(T') = \max\{t, \rho_1, \rho_2\} + k$. Call this value ρ'. Without loss of generality, assume that the priority of the root of T_1 is greater than the priority of the root of T_2 (see Figure 4.7). Because $l(T_1) = l(T')$, then for all $v \in l(T'), t(v) \leq \rho_1 + k\, d_{T_1}(v) \leq \rho' + k\, d_{T'}(v)$, since the depth of v is the same in T_1 as in T'. By the induction hypothesis we can find the time stamps on $r(T') = $ join$(r(T_1), T_2)$, since the size of $r(T_1)$ is less than n. A valid ρ-value for $r(T_1)$ is $\rho_1 + k$. Therefore, a valid ρ-value for $r(T')$ is $\max\{\rho', \rho_1 + k, \rho_2\} + k = \rho' + k$. Since v's depth in $r(T')$ is 1 less than its depth in T', $t(v) \leq \rho' + k\, d_{T'}(v)$ for all $v \in r(T')$. Thus, ρ' is a valid ρ-value for T'. ∎

Theorem 4.10 (Bound on difference depth) *If* diff(T_1, T_2) *is called at time t and valid ρ-values for T_1 and T_2 are ρ_1 and ρ_2, then the maximum time stamp on the result treap T_d is* $\max\{t, \rho_1, \rho_2\} + O(h(T_1) + h(T_2) + h(T_d))$.

Proof. Let k be a constant greater than the maximum computational DAG depth between successive recursive calls to splitm, join, and diff. Since ρ_1 and ρ_2 are valid τ-values for T_1 and T_2 by Property 4.7 and using the same arguments as in Theorem 4.5, after $\max\{t, \rho_1, \rho_2\} + O(h(T_1) + h(T_2))$ depth in the computation DAG, diff has reached the bottom of every recursive path (either lines 9 or 10 in Figure 4.6 applies) and every future result of splitm has been computed. Thus, by property 4.8 there exists a constant $\rho' = \max\{t, \rho_1, \rho_2\} + O(h(T_1) + h(T_2))$ which is a valid ρ-value for all trees (treaps l and r on lines 13 and 14) that are the result of these calls at the leaves of the call tree. At this point we can find ρ-values for the results of each recursive call to diff. Let ρ_l and ρ_r be valid ρ-values for the results treaps l and r. Because the recursive calls to diff are called with futures, the call to join is always made by $\max\{\rho_l, \rho_r\}$. By Lemma 4.9 a valid ρ-value for result of the diff recursive call is $\max\{\rho_l, \rho_r\} + k$ (compare with the definition of the height of a tree). But since all the result treaps at the leaves of the recursive call tree have ρ' as a valid ρ-value and the height of the recursive call tree is no more than $h(T_1)$, a valid ρ-value for the treap
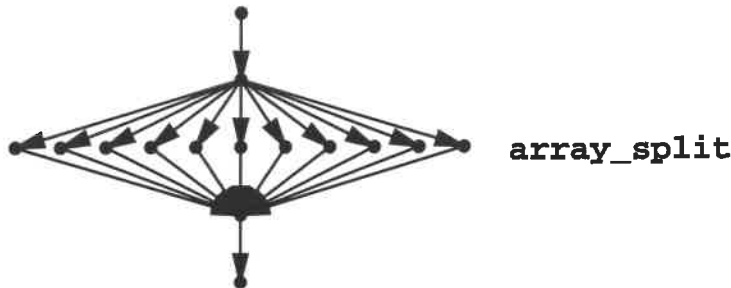
Figure 4.8: The DAG for an `array_split` on an array of length 11.

at the root of the call tree must be $\rho' + k\,h(T_1)$. By definition of ρ-values, the time stamp of the deepest node in that treap is $\rho' + O(h(T_1) + h(T_d)) = \max\{t, \rho_1, \rho_2\} + O(h(T_1) + h(T_2) + h(T_d))$.
∎

Corollary 4.11 (Expected difference depth) *The expected depth to find the difference of two treaps of size n and m is $O(\log n + \log m)$.*

Proof. Since the expected height of the two input treaps are $O(\log n)$ and $O(\log m)$ and the expected height of the result treap is $O(\log(n - m))$, the expected depth to find the difference is $O(\log n + \log m)$. ∎

4.3.4   2-6 Trees

We can obtain a pipelined variant of top-down 2-3-4 trees using 2-6 trees. It is analogous to the bottom-up pipelined 2-3 trees of Paul, Vishkin and Wagener [92]. Each node of a 2-6 tree has 1 to 5 keys in increasing value and one child for each range defined by the keys. The children are 2-6 trees with key values within their range. Every key appears only once, either in internal nodes or at the leaves, and all leaves are at the same level. I refer to the keys in the tree as *splitters*.

I consider the problem of inserting a set of sorted keys into a 2-6 tree. For this problem I use an array primitive `array_split`, which splits a sorted array of size m into two arrays, one with values less than the splitter and one with values greater than the splitter. In my cost model I define this operation to have $O(1)$ depth and $O(m)$ work—in the DAG I view the operation as a DAG of depth 2 and breadth m (see Figure 4.8).[‡] First consider inserting an ordered set of keys in which there is at least one key in the 2-6 tree between each pair of keys to be inserted. I call such an array a *well-separated* key array. Later, I show how to insert any ordered set of keys.

If the root of the 2-6 tree has more than three children, the algorithm `insert` splits the root into two 2-3 nodes (nodes with 2 or 3 children) and creates a new root using the "middle" splitter and these new 2-3 nodes as children. From now on `insert` maintains the invariant that the root of the tree into which it is inserting is a 2-3 node. It does so by always splitting any child, as necessary, before applying a recursive call on that child. Every time it splits a child it needs to include one of the child's splitters into the root. But since the root has at most two splitters and three children (by the invariant), the resulting root will have at most five splitters and six children.

To insert an ordered well-separated key array, `insert` first splits the keys by the smallest splitter at the root into two arrays using the `array_split` primitive. It will insert the first of the two arrays

---

[‡]The reader might argue that the split operation should have depth greater than $O(1)$ because of the need to collect the two sets of values. I show in Section 4.4, however, that the cost of the `array_split` is fully accounted for in the implementation.

into the left child. If there is no second splitter, it will insert the second key array into the right child. Otherwise, it splits the second array by the second splitter and will insert the resulting key arrays into the middle and right children. Before recursively inserting a key array into a child, `insert` first checks whether the child needs to be split to maintain the 2-3 root node invariant. When a child is split, it obtains a new splitter and two new children. It uses the new splitter to split the key arrays into two arrays that it will insert into the two new children. Next it recursively inserts the key arrays into the appropriate children to obtain new children for the root. Eventually, `insert` will reach a leaf node, which must be a 2-3 node by the invariant. Because of the requirement that there is always at least one key in the 2-6 tree between each key to be inserted, there can be at most 3 keys that need to be inserted in any one leaf; these keys can be included in the leaf without having to split the node. Note that the height of the tree increases by at most one, when the root of the tree was split.

If `insert` uses futures when making its recursive calls, then it traverses the different paths down the tree in parallel by forking off new tasks for each recursive call. Since the paths are at most $\log n$ long, inserting an ordered well-separated key array of size m into a 2-6 tree of size n takes $O(\log n)$ depth and $O(m \log n)$ work. No pipelining is needed.

To insert an arbitrary ordered set of keys of size m, `insert` first forms a balanced binary tree of the keys (conceptually), and then creates a list of arrays of keys, where each array is made up of the keys from one level of the tree. Thus, the first array contains the median key, the next array contains the first and third quartiles, and so on. It then successively inserts each array into the 2-6 tree using the tree returned by the previous insertion, see Figure 4.9. By inserting the keys in this manner, `insert` guarantees that for any array of keys, there is at least one key in the 2-6 tree between each pair of keys in the array, because it has inserted such keys previously. Without pipelining, inserting the $\log m$ arrays into a tree of size n would require $O(\log n \log m)$ depth and $O(m \log n)$ work.
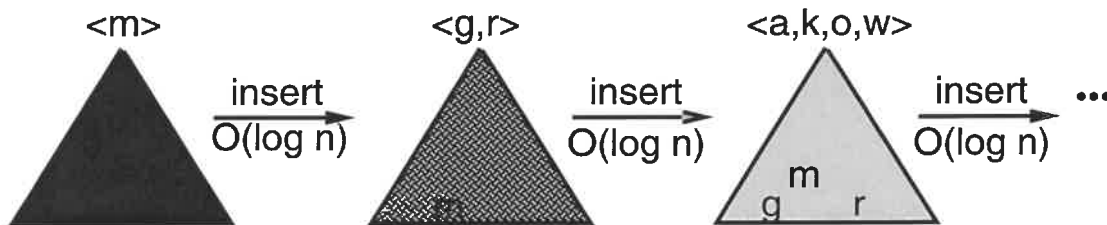

Figure 4.9: Inserting an ordered set of keys into a 2-6 tree of size n. The array (items inclosed in angle brackets <>) at the root of a tree is the well-separated key array to be inserted in the tree. First insert the median `<m>` into the tree (dark shading). Next insert the first and third quartile `<g,r>` into the resulting tree (medium shading). Then insert the next well-separated array into the next resulting tree (light shading) and so on. Inserting each well-separated key array takes $O(\log n)$ depth.

By simply making the recursive call that inserts a well-separated key array return a future (in addition to the futures used in its recursive calls), `insert` can pipeline inserting each array of keys into the 2-6 tree—no other changes to the code need to be made. The crucial fact that makes the pipelining work is that, in constant depth, `insert` can return the root node with its keys values filled in, although its children may be futures, see Figure 4.10. It can then insert the next well-separated key array in the list into this new root, which is the root of the 2-6 tree that will eventually contain the original and previous well-separated key arrays. With this structural information in the root the next insertion can also return the root in constant depth. Although it may need to wait a constant depth before the children nodes are ready, from then on the children

of all descendents will be ready when it reaches them. In this way there can be an array of keys being inserted at every second level and possibly every level of the 2-6 tree.
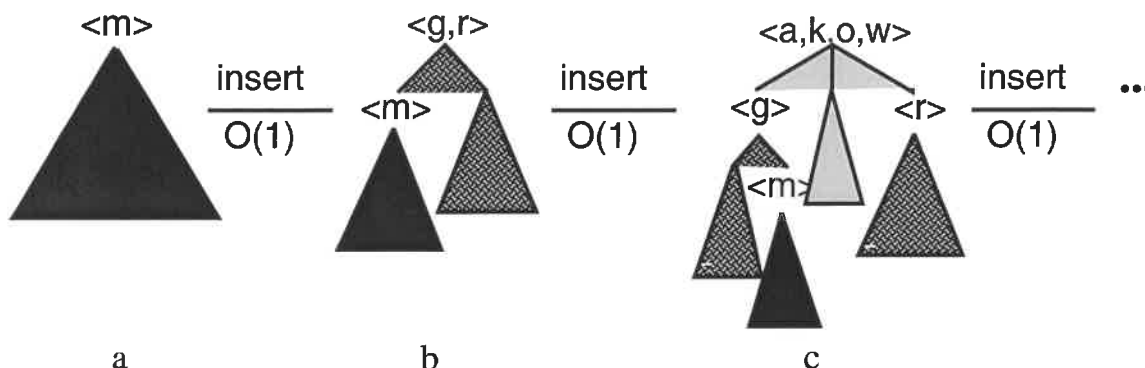

Figure 4.10: Inserting an ordered set of keys into a 2-6 tree of size n using pipelining. An array (items inclosed in angle brackets <>) at the root of a tree is the well-separated key array to be inserted in the tree and refers to a future in the computation. **a:** First, the median <m> is inserted into the original tree (dark shading). **b:** As soon as the root node of the resulting tree is ready (medium shading), the first and third quartile <g,r> are inserted into it. The root is ready in $O(1)$ depth. The median <m> still needs to be inserted into a child of the original tree root (dark shading), the result of which is a future. When the future value is available it becomes a child in the second result tree (medium shading). **c:** The next well-separated array is inserted into the next resulting tree (light shading) and so on.

Definition 3 v *is a valid v-value for a 2-6 tree T, if for all $v \in T$, $t(v) \leq v + k_b d_T(v))$, where $t(v)$ is the time stamp for v, $d_T(v)$ is the depth of v in T, and k_b is constant.*

Theorem 4.12 (Insertion into a 2-6 Tree) *A set of m ordered keys can be inserted in a 2-6 tree of size $n > m$ in $O(\log n + \log m)$ depth and $O(m \log n)$ work.*

Proof. First note that we can create a pipeline of well-separated key arrays from an arbitrary array of sorted keys. Each successive well-separated key array can be found in constant time, k_w, given the indices of the keys that made up the previous key array. That is, the time stamp for i^{th} key array is $k_w \cdot i$. Let T_0 be the original 2-6 tree we are inserting into, and v_0 be its associated valid v-value. Let T_i be the resulting 2-6 tree after inserting the ith well-separated key array into T_0. We will show that

$$v_{i+1} = v_i + 3k_b \tag{4.3}$$

are valid v-values for $T_{i+1}, i = 0, \ldots, \log m$.[§]

Assume v_i is a valid v-value for T_i and k_b is large enough such that $v_i > k_w(i+1)$. The `insert` function can start to insert the $(i+1)^{st}$ well-separated array once both it and the root of T_i are available; that is, at time $min(k_w(i+1), v_i) = v_i$. In the worse case the root of T_i needs to be split. It can do so in constant depth k_r, since it has all the structural information it needs to create the new root and its two children. Again we assume k_b is large enough such that $k_b > k_r$. This splitting result in a new intermediate tree T_i', with a valid v-value $v_i + k_b$. By induction on d we will find upper bounds on the time stamps of nodes at depth d of T_{i+1}.

First we find $t(T_{i+1})$. Once the root of T_i' or T_i and its children are available `insert` can do all the work necessary to create the root of T_{i+1}. These nodes have time stamps at most $v_i + 2k_b$. Then,

---

[§]It is also possible to show that $v_{i+1} = v_i + 2k_b$

in constant depth, `insert` can split the keys, determine which children need to be split, determine any new keys and children that need to be added to the root of T_i', split the key arrays by the new keys, and proceed with the recursive calls, which return futures to the children of the new root. Let this constant depth be k_b. Thus, it has the structural information needed to create and return the root so that $t(T_{i+1}) = v_i + 3k_b$. The recursive calls on nodes at depth 1 of T_i' are made by $v_i + 3k_b$ and these nodes and their children have time stamps no more than that. Therefore, by $v + 4k_b$ it can create the nodes at depth 1 of T_{i+1} and proceed with the recursive calls on nodes at depth 2. In general, the recursive call on nodes at depth d occur by $v_i + (d + 2)k_b$ and the nodes of T_i' at level d and level $d + 1$ are also available at that time. Thus, the time stamps for a node at level d of T_{i+1} is at most $v_i + (d + 3)k_b$, proving equation 4.3 holds. Since there are $\log m$ well-separated key arrays, the final 2-6 tree has a valid v-value $v + O(\log m)$ and the tree has depth $O(\log(m + n))$. Therefore, the largest time stamp is no more than $v + O(\log m + \log n)$.

It is easy to see that inserting m keys into a tree of size n using the above algorithm does no more work, within constants, than inserting the m keys one at a time. Since the latter takes $O(m \log n)$ work so does the former. ∎

4.4 Implementation

In this section I describe an implementation of futures and give provable bounds on the runtime of computations based on this implementation. The bounds include all costs for handling the suspension and reactivation of threads required by the futures and the cost of scheduling threads on processors. The implementation is an extension of the implementation described in [52] which allows us to improve the time bounds and avoid concurrent memory access.

The main idea of the implementation is to maintain a set of active threads S, and to execute a sequence of steps repeatedly, each of which takes some threads from S, executes some work on each, and returns some threads to S. The interesting part of the implementation is handling the suspension and reactivation of threads due to reading and writing to future cells. As suggested for the implementation of Multilisp [56], a queue can be associated with each future cell so that when a thread suspends waiting for a write on that cell, it is added to the queue, and when the write occurs, all the threads on the associated queue are returned to the active set S. Since multiple threads could suspend on a single cell on any given time step, the implementation needs to be able to add the threads to a queue in parallel. Previous work [52] has shown that by using dynamically growing arrays to implement the queues in parallel, any computation with w work and d depth will run in $O(w/p + d \cdot T_f(p))$ time on a CRCW PRAM, where $T_f(p)$ is the latency of a work-efficient fetch-and-add operation on p processors.

By placing a restriction on the code called linearity, we can guarantee that every future cell is read at most once so that only a single thread will ever need to be queued on a future cell. This greatly simplifies the implementation and allows us to replace the fetch-and-add with a scan operation. A further important advantage of linearity is that it guarantees that the implementation only uses exclusive reads and writes to shared memory. The linearity restriction is such that any code can easily be converted to be linear, although this can come at the cost of increasing the work or depth of an algorithm.

The linearity restriction on code is based on ideas from linear logic [50]. In the context of this dissertation linearizing code implies that whenever a variable is referenced more than once in the code a copy is made implicitly for each use [79]. The copy must be a so-called deep copy, which copies the full structure (e.g., if a variable refers to a list, the full list must be copied, not just the

head).[¶] Linearized code has the property that at any time every value can only have a single pointer to it [79]. This implies that there can only be a single pointer to a future cell and it can therefore only be read from once. Similarly it implies that there can only be exclusive read access to any value, even if it is not a future cell. Linear code has been studied extensively in the programming language community in the context of various memory optimizations, such as updating functional data in place or simplifying memory management [79, 113, 11, 2, 36].

Linearizing code does not affect the performance of any of the algorithms I considered in this chapter. For example, consider the body of the *split* code in Figure 4.2, lines 4–11. Figure 4.11 shows the linearized version of the same code. The only variables that are read more than once refer to keys and splitters(v and s). Since it is no more expensive to copy v and s than to compare them, such copying does not affect the costs. The trees themselves are never referenced more than once—although, L and R appear once each in the **then** or the **else** part of the **if** statement, only one of these branches can be executed. The trees L_1 and R_1 appear twice in both **then** and **else** parts, but one case is simply defining them (lines 5 and 9) while the other actually references them (lines 6 and 10).

```
 1   datatype tree = node of int*tree*tree | leaf;

 2   fun split(s,leaf) = (leaf,leaf)
 3     | split(s,node(v,L,R)) =
 4     let val (sa,sb) = copy(s);
 5         val (va,vb) = copy(v);
 6     in if sa < va then
 7         let val (L1,R1) = ?split(sb,L)
 8         in (L1,node(vb,R1,R))
 9         end
10       else
11         let val (L1,R1) = ?split(sb,R)
12         in (node(vb,L,L1),R1)
13         end;
14     end;
```

Figure 4.11: Linearized code for splitting two binary trees. Two copies of **s** and **v** are made so that no variable is referenced more than once. A variable that is referenced once in the **then** clause and once in the **else** clause of an **if** statement is referenced once overall because only one of the two clauses is executed. Similarly a variable must be referenced at most once in each function body.

I now consider the main result of this section. The bounds are in terms of the EREW scan model [19], which is the EREW extended with a unit-time scan (all-prefix-sums) operation. The bounds on the scan model imply bounds of $O(w/p + d \log p)$ time on the plain EREW PRAM, $O(gw/p + d(T_s + L))$ on the BSP [109], and $O(w/p + d \log p)$ on an Asynchronous EREW PRAM [42] using standard simulations.

Lemma 4.13 (Implementation of Futures) *Any linearized future-based computation with w work and d depth can be simulated on an EREW scan model in $O(w/p + d)$ time.*

Proof. In the following discussion we say that an action (node in the computation DAG) is *ready* if all its parents have been executed and that a thread is *active* if one of its actions is ready. We

---

[¶]Note that to copy the structure, the copy must be strict on the full structure—all futures must be written before they can be copied.

store threads as bounded-sized structures containing a code pointer and pointers to local variables. We store each future cell as a structure that holds a flag and a pointer. Initially the flag is unset; when the pointer is filled the flag is set. The pointer points to either a value or a suspended thread. We store the set of active threads S as an array.

The algorithm takes a sequences of steps, each of which takes $m = \min\{|S|, p\}$ threads from S, executes one action on each thread, and returns the resulting active threads to S. We treat S as a stack so that threads are removed and added to the top of the stack. Let t be the largest index of an active thread on S.

To take threads from S:

1. Remove m threads from the top of S. That is, processor i takes thread $S[t-i]$, unless $t-i < 0$, in which case it does nothing this step.

2. Decrement the stack pointer by m ($t = t - m$).

The above operations take constant time.

Next we show that each action takes constant time.

1. If a thread with a read pointer to a future cell wants to read the future then

    (a) if the future cell has been set then dereference the pointer,

    (b) otherwise set the flag, write a pointer to itself into the future cell, and suspend.

2. If a thread with a write pointer to a future cell wants to write a result then

    (a) if the future cell has been set then, read the future cell pointer, write a new pointer to the result, and reactivate the thread pointed to in the pointer just read,

    (b) otherwise write a pointer to the result and set the flag.

3. Otherwise execute the action.

To prevent both the writer and reader from accessing the flag concurrently we can assign even steps to the reader and odd steps to the writer.[||] Thus, reading from and writing to a future cell takes constant time and, by definition of the DAG in our model, actions not involving a future cell take constant time.

After executing one action, each thread can return zero, one or two threads to S (zero if it dies or suspends, one if it continues, and two if it forks or reactivates another thread). To return the new active threads to S:

1. Compute the scan of the number of threads each processor returns.

2. If the maximum scan value, j_{max}, is greater than the allocated size of S, then allocate an array double the size of S and copy S to the newly allocated space using a scan.

3. For scan result j for a result thread, place the thread at $S[t+j]$.

4. Increment the top of the stack by j_{max} ($t = t + j_{max}$).

[||]A test-and-set operation will suffice, but we don't have such an operation in an EREW PRAM.

Although the size of S may be much greater than p, we can amortize the cost of the copy to the cost it took to place threads on S in the first place, in the same manner as in [52]. Because we double the array on each expansion, the cost of the copy is at most two times the maximum size of S. Since each processor has at most two threads to return to S, the implementation can place the threads back in S in constant (amortized) time using the unit-time scan primitive assumed in the machine model.

In summary, since the algorithm can remove $\min\{|S|, p\}$ threads from the top of S in constant time, can execute one action of each thread in constant time, and can place resulting active threads back on S in constant time, the whole step takes constant time. Since, on each step, the implementation processes $\min\{|S|, p\}$ threads, and S holds all the active threads (by definition), the implementation executes a greedy schedule of the computation DAG. The number of steps is therefore bounded by $w/p + d$ [27] and the total time by $O(w/p + d)$. Note that for the time bounds it does not matter which threads are taken from S on each step, allowing the implementation some freedom in selecting a schedule that is space or communication efficient. The stack discipline we describe is above is probably much better for space than a queue discipline, for example.

We now outline how to handle the `array_split` operation used in the 2-6 trees. We first consider implementing a simpler `array_scan` which, given an array of integers of length n, returns the plus_scan of the array in $O(n)$ work and $O(1)$ depth (remember that n could be much larger than p). As with the `array_split` we account for the cost of the `array_scan` in our cost model as a DAG of depth 2 and breadth n. When coming to an `array_scan` in the code the implementation spawns n threads and places them in the set of active threads. Since creating n threads could take more than constant time on p processors, they are created lazily using a stub as described in [16]—threads are expanded when taken from S instead of when inserted. For each block of p or less threads that are scheduled from the set in a particular step, we can use the unit-time scan primitive assumed in the machine model to execute the scan across that subset and place the new running sum back into the stub. When the last thread finishes, it reactivates the parent thread and the scan is complete. If we associate each created thread as a node in the breadth n DAG then each node of this DAG can be executed in constant work, and the sink node (bottom node of the $2 \times n$ DAG) is ready as soon as the last thread is done. Since the schedule remains greedy (on each step the implementation always schedules $\min\{|S|, p\}$ threads), the number of steps is bounded by $O(w/p + d)$, where w is now the total number of nodes in the DAG including the expanded DAGs for each `array_scan` (i.e., we are including $O(n)$ work for each `array_scan`). Each step of the scheduling algorithm still takes constant time so the total time on the EREW scan model is also bound by $O(w/p + d)$.

The `array_split` can be implemented by broadcasting the pivot, comparing the array elements to it, executing two scans to determine the final locations of the array elements, and writing the values to these locations (see [19] for example). Each step can be implemented with $O(n)$ work and $O(1)$ depth in a similar way as described above. ∎

4.5 Conclusions

This chapter suggests an approach for designing and analyzing pipelined parallel algorithms using futures. The approach is based on working with an abstract language-based model that hides the implementation of futures from the user. Universal bounds for implementing the model are then shown separately.

The main advantages of this approach over pipelining by hand is that it leaves the management of pipelining to the runtime system, greatly simplifying the code. The code I gave for merging and

for treaps is indeed very simple, and is just the obvious sequential code with future annotations added in a few places. I expect that it would be very messy to pipeline the treaps by hand because of the unbalanced and dynamic nature of the tree structures. In particular, the depth at which subtrees returned by the `split` function become available is data dependent, and to maintain the depth bounds an implementation must start the next computation as soon as a node becomes available. The immediate reawakening of suspended tasks is therefore a critical part of any implementation. The code for the 2-6 trees is somewhat more complicated, but still significantly simpler than a version in which the pipelining is done by hand.

Another important advantage of the approach is that it gives more flexibility to the implementation to generate efficient schedules. The algorithms of Cole and PVW specify a very rigorous and synchronous schedule for pipelining while the specification of pipelining using futures is much more asynchronous—the only synchronization is through the future-cells themselves and there is no specification in the algorithms of what happens on what step. This gives freedom to the implementation as to how to schedule the tasks. The implementation, for example, could optimize the schedule for either space efficiency [27, 16, 17] or locality [28]. On a uniprocessor the implementation could run the code in a purely sequential mode without any need for synchronization.

I am not yet sure how general the approach is. I have not been able to show, for example, whether the method can be used to generate a sort that has depth $O(\log n)$. I conjecture that a simple mergesort based on the merge in section 4.3.1 has expected depth (averaged over all possible input orderings) close to $O(\log n)$, perhaps $O(\log n \log \log n)$. This algorithm has three levels of pipelining (i.e., has depth $O(\log^3 n)$ without pipelining).

This work is part of a larger research theme of studying language-based cost models, as opposed to machine-based models, and is an extension of the work on the NESL programming language and its corresponding cost model based on work and depth (summarized in [20]).

4.6   ML Code

All code in this chapter is a subset of ML [88] augmented with future notation, a question mark (?). The syntax I use is summarized in Figure 4.12. The LET VAR *pattern* = *exp* IN *exp* END notation is used to define local variables and is similar to Let in Lisp. The DATATYPE notation is used to define recursive structures. For example, the notation

```
datatype tree = node of int*tree*tree | leaf;
```

is used to define a datatype called `tree` which can either be a `node` with three fields (an integer, and two trees), or a leaf.

Pattern matching is used both for pulling datatypes apart into their components (e.g., separating a list into its head and tail) and for branching based on the subtype. For example, in the pattern:

```
fun merge(leaf,B) = B
  | merge(A,leaf) = A
  | merge(node(v,L,R),B) = .....
```

the code first checks if the first argument is a `leaf` type, and returns B if it is, it then checks if the second argument is a `leaf` type, and returns A if it is, otherwise it pulls the first argument, which must be a `node` into its three components (the integer v and the two subtrees L and R) and executes the remaining code.

defn	::=	FUN **body** [**|** **body**]*;	function def'n
	::=	DATATYPE **name** = *sumtype*;	datatype def'n
body	::=	**name** *pattern* = *exp*	function body
exp	::=	*const*	constant
		name	variable
		IF *exp* THEN *exp* ELSE *exp*	conditional
		LET VAR *pattern* = *exp*	local bindings
		IN *exp* END	
		name (*exp*,...)	fn application
		exp binop exp	binary op
		(*exp*)	paren expr'n
		? *exp*	future
pattern	::=	**name**	var or datatype
		pattern,*pattern*	tuple
		name(*pattern*)	datatype
		(*pattern*)	paren pattern
sumtype	::=	**name** [OF *prodtype*]	sum type
		[**|** *sumtype*]	
prodtype	::=	**name** [* *prodtype*]	product type

Figure 4.12:   The ML syntax used in this chapter.

Chapter 5

Discussion

Parallel computing seems to have temporarily settled down to embarrassingly parallel applications on distributed memory machines and networks of workstations, and large server applications on shared-memory multiprocessors. The latter applications include financial and business analysis, scientific computing, seismic processing, data mining, computer-aided engineering, animation, and electronic design. Soon all types of applications will be on shared-memory multiprocessors, including applications that use pointer data structures. But until pointer-based algorithms perform well, there will be little incentive to build such pointer-based applications. And until such applications are in use, there will be little incentive for hardware manufacturers to address the needs of these applications. The work presented in the dissertation is an early step in an effort to understand the potential and limitations of pointer-based applications.

5.1   Limitations

I did not design the algorithms presented here for distributed memory systems with moderate performance interconnect. That is, I did not attempt

- to minimize the number of rounds of communication between processors and memory or among processors with local memory;

- to minimize contention at routers or on communications links;

- to minimize the average or maximum bandwidth required; or

- to do load balancing or data distribution;

Quite different techniques and algorithms are likely to be required to get reasonable performance on such loosely connected systems, and the question remains as to whether truly good performance is possible for some problems.

5.2   Contributions

I have made the following contributions to parallel pointer-based algorithm design and implementation.

- I developed a new list-ranking algorithm that is quite different and has smaller constants than other list-ranking algorithms. The solutions to list ranking seem to be wide and varied; my algorithm seems particularly well suited to vector multiprocessors.

78

- I was first to consider treaps for balanced trees in the parallel setting. The advantage of treaps is that the simplicity of the sequential algorithms has carried over to the parallel ones.

- I designed new algorithms for union, intersection, and difference on sets of size n and m that run in expected $O(m \log(n/m))$ serial time or parallel work. This is optimal.

- I developed two implementations of pointer-based problems: list ranking and dynamic balanced trees.

- I demonstrated that simple work-efficient parallel pointer-based algorithms can be faster than optimal ones on practical problem sizes and much faster than serial implementations on fast workstations.

- I showed how pipelining several tree-based algorithm could reduce the asymptotic depth of the algorithms by a logarithmic factor. For some of these algorithms, traditional pipelining with fixed delays would not reduce the algorithms' asymptotic depth.

- I analyzed the performance of the algorithms presented here. The analysis of the list ranking included the constants associated with the implementation. The analysis technique for the dynamic pipelining algorithms is new.

5.3 Future Work

It would be interesting to investigate whether these parallel linked list and tree algorithms can be used on single processor workstations to close the growing speed gap between memory subsystems and high-performance processors.

5.4 Conclusions

Although we a long way from pointer-based algorithms being common on parallel machines, the trend towards multiprocessor workstations and servers in the general computing market will put pressure on bringing such applications onto them. The advantages may be two fold: better utilization of the multiple processors and better latency hiding to the memory subsystems.

My implementation results show that optimal depth algorithms do not always provide the best performance solutions. Previous optimal algorithms are much more complex than the ones I presented here, and are likely to have much poorer performance on practical size problems. Even an optimal list-ranking algorithm derived from mine is likely to be slower, except possibly when there are many processors compared to the problem size. This algorithm by Ranade [97] has a single compaction phase followed by Wyllie's algorithm. But to know which elements need to be compacted requires marking completed elements during the main loop, increasing the constants of the algorithm. On the other hand, his algorithm has a better randomization method and is less likely exhibit worse-case behavior. The pipelining method for balanced trees is also likely to be slower than the nonoptimal depth algorithm I implemented. Pipelining results in many more short computation threads, increasing the overhead of the algorithms.

As with many sequential algorithms, it appears that randomized parallel algorithms have a significant performance benefit over deterministic algorithms. Not only do randomized algorithms have much simpler solutions, they often solve other problems related to parallel computing, such as memory contention and load balancing.

Virtual processing and multithreading play a crucial role in parallel pointer-based implementations. Without them, there would be no latency hiding and performance would be dismal. Work-efficiency and small constants are what distinguish good performance parallel algorithms from those that do not perform as well on the same hardware. This observation is not too surprising, since it also holds true for sequential processing. Of course, there may be other algorithmic solutions that reduce some specific hardware bottleneck and provide even better performance on certain machines.

Bibliography

[1] K. Abrahamson, N. Dadoun, D. G. Kirpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

[2] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[3] M. Ajtai, J. Komlos, and E. Szemeredi. An $O(n \lg n)$ sorting network. In *Proceedings ACM Symposium on Theory of Computing*, pages 1–9, Apr. 1983.

[4] R. Anderson and J. ao C. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June/July 1992.

[5] R. Anderson and G. L. Miller. Deterministic parallel list ranking. In J. H. Reif, editor, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC88*, volume 319 of *Lecture Notes in Computer Science*, pages 81–90. Springer-Verlag, June/July 1988.

[6] R. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, 1990.

[7] R. J. Anderson, E. W. Meyer, and M. K. Warmuth. Parallel approximation algorithms for bin packing. *Information and Computation*, 82:262–277, 1989.

[8] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM Journal of Computing*, 18(3):499–532, June 1989.

[9] S. Baase. Introduction to parallel connectivity, list ranking, and Euler tour techniques. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 61–114. Morgan Kaufmann, 1993.

[10] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results 10-93. Technical Report RNR-94-006, NASA Ames Research Center, Mar. 1994.

[11] H. Baker. Lively linear lisp — 'Look Ma, no garbage!'. *ACM SIGPLAN Notices*, 27(8):89–98, Aug. 1992.

[12] H. G. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12(8):55–59, Aug. 1977.

[13] A. Bäumker and W. Dittrich. Fully dynamic search trees for an extension of the BSP model. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 233–242, 1996.

[14] A. Bik and A. Sijshoff. Compilation techniques for sparse matrix computation. In *Supercomputing '93*, pages 416–424, Tokyo, Japan, July 1993.

[15] D. Blackston and T. Suel. Highly portable and efficient implementations of parallel adaptive n-body methods. In *Proceedings of Supercomputing '97*, 1997.

[16] G. Blelloch, P. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, July 1995.

[17] G. Blelloch, P. Gibbons, Y. Matias, and G. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 12–23, Newport, RI, June 1997.

[18] G. Blelloch and M. Reid-Miller. Pipelining with futures. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 249–259, Newport, RI, June 1997.

[19] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C–38(11):1526–1538, Nov. 1989.

[20] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, Mar. 1996.

[21] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[22] G. E. Blelloch, S. Chatterjee, and M. Zagha. Solving linear recurrences with loop raking. In *Proceedings Sixth International Parallel Processing Symposium*, pages 416–424, Mar. 1992.

[23] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[24] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, Aug. 1993.

[25] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June 1998.

[26] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, CA, July 1995.

[27] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 362–371, May 1993.

[28] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Nov. 1994.

[29] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the Association for Computing Machinery*, 21(2):201–206, Apr. 1974.

[30] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the Association for Computing Machinery*, 26(2):211–226, Apr. 1979.

[31] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9(3):594–614, Aug. 1980.

[32] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In D. Padua, D. Gelernter, and A. Nicolau, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 95–113. MIT Press, 1990.

[33] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with OLDEN (parallel programming). In *Proceedings 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 1–20. Springer-Verlag, Aug. 1993.

[34] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural merge sort. In *Proceedings of the International Symposium on Algorithms SIGAL'90*, pages 251–260, Tokyo, Japan, Aug. 1990.

[35] R. Chandra, A. Gupta, and J. Hennessy. COOL: A Language for Parallel Programming. In D. Padua, D. Gelernter, and A. Nicolau, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 126–148. MIT Press, 1990.

[36] J. L. Chirimar, C. A. Gunter, and J. G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, Mar. 1996.

[37] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):770–785, Aug. 1988.

[38] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *Proceedings ACM Symposium on Theory of Computing*, pages 206–219, 1986.

[39] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):31–53, 1986.

[40] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal of Computing*, 17(1):128–142, Feb. 1988.

[41] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334–352, June 1989.

[42] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, June 1989.

[43] E. Dekel and I. Azsvath. Parallel external merging. *Journal of Parallel and Distributed Computing*, 6:623–635, 1989.

[44] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, Feb. 1989.

[45] W. Feller. *An Introduction to Probability Theory and Its Applications, Volume 2*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, New York, 1971.

[46] D. P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C–27(4):289–296, Apr. 1978.

[47] J. Gabarró and J. Petit i Silvestre. ParaDict, a data parallel library for dictionaries. In *5th EUROMICRO Workshop on Parallel and Distributed Processing*, 1997.

[48] H. Gazit. Optimal EREW parallel algorithms for connectivity, ear decomposition and st-number of planar graphs. In *Proceedings International Conference on Parallel Processing*, pages 84–91, Aug. 1991.

[49] H. Gazit, G. L. Miller, and S.-H. Teng. Optimal tree contraction in the EREW model. In S. K. Tewsburg, B. W. Dickinson, and S. C. Schwartz, editors, *Concurrent Computations*, pages 139–156. Plenum Publishing Corporation, 1988.

[50] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[51] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 16–25, Cape May, NJ, June 1994.

[52] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proceedings of the ACM Symposium on Principals of Programming Languages*, pages 309–321, Jan. 1996.

[53] X. Guan and M. A. Langston. Time-space optimal parallel merging and sorting. *IEEE Transactions on Computers*, 40:592–602, 1991.

[54] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, 1989.

[55] A. R. Hainline, S. R. Thompson, and L. L. Halcomb. Vector performance estimation for CRAY X-MP/Y-MP supercomputers. *Journal of Supercomputing*, 6:49–70, 1992.

[56] R. H. Halstead. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[57] J. Hardwick. *A Portable Toolbox for Nested Data-Parallel Algorithms on Distributed-Memory Multiprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1997.

[58] L. Hendren and G. Gao. Designing programming languages for the analyzability of pointer data structures. *Computer Languages*, 19(2):119–134, 1994.

[59] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[60] L. Highan and E. Schenk. Maintaining B-trees on an EREW PRAM. *Journal of Parallel and Distributed Computing*, 22:329–335, 1994.

[61] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, Dec. 1986.

[62] R. W. Hockney. Characterization of parallel computers and algorithms. *Computer Physics Communications*, 26:285–291, 1982.

[63] T.-s. Hsu and V. Ramachandran. Efficient implementation of virtual processing for some combinatorial algorithms on the MasPar MP-1. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Computing*, pages 154–159, San Antonio, TX, Oct. 1995.

[64] T.-s. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on the MasPar. In *Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 15*, pages 165–198. American Mathematical Society, 1994. Also available as TR-92-38, Dept. of Comp. Sci., University of Texas at Austin, February 1992.

[65] T.-s. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing. In *Proceedings of the International Parallel Processing Symposium*, pages 106–112, Santa Barbara CA, Apr. 1995.

[66] T.-s. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components in graphs. In *Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 30*. American Mathematical Society, 1997.

[67] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal of Computing*, 1:31–39, Mar. 1972.

[68] J. Katajainen. Efficient parallel algorithms for manipulating sorted sets. *Proceedings of the 17th Annual Computer Science Conference, Australian Computer Science Communications*, 16(1):281–288, 1994.

[69] J. Katajainen, C. Levcopoulos, and O. Petersson. Space-efficient parallel merging. In *Proceedings of the 4th International PARLE Conference (Parallel Architectures and Languages Europe)*, volume 605 of *Lecture Notes in Computer Science*, pages 37–49, 1992.

[70] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, MA, 1968.

[71] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, MA, 1973.

[72] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computation by ranking (extended abstract). In J. H. Reif, editor, *VLSI Algorithms and Architectures: 3rd Aegean Workshop on Computing, AWOC88*, volume 319 of *Lecture Notes in Computer Science*, pages 101–110. Springer-Verlag, June/July 1988.

[73] D. A. Krantz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 81–90, 1989.

[74] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In *Proceedings of the 3rd Dimacs Implementation Challenge Workshop*, Oct. 1994.

[75] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, 1990.

[76] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5(1):43–64, 1990.

[77] S. Kumar, S. M. Goddard, and J. F. Prins. Connected components algorithms for mesh-connected parallel computers. In *Proceedings of the 3rd Dimacs Implementation Challenge Workshop*, pages 37–51, Oct. 1994.

[78] S. Kumar, S. M. Goddard, and J. F. Prins. Connected components algorithms for mesh-connected parallel computers. In S. Bhatt, editor, *Parallel Algorithms*, pages 43–48. American Mathematical Society, Providence RI, 1997.

[79] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[80] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'96*, pages 222–233, Oct. 1996.

[81] S. S. Lumetta, A. Krishnamurthy, and D. E. Culler. Towards modeling the performance of a fast connected components algorithm on parallel machines. In *Proceedings Supercomputing '95*, 1995.

[82] Y. Mansour, N. Nisan, and U. Vishkin. Trade-offs between communication throughput and parallel time. In *Proceedings ACM Symposium on Theory of Computing*, pages 372–381, Montreal, P.Q., May 1994.

[83] M. L. Mauldin. Lycos: Design choices in an Internet search service. *IEEE Expert*, 12(1), Jan. 1997. (www.computer.org/pubs/expert/1997/trends/x1008/mauldin.htm).

[84] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memory. *Acta Informatica*, 21:339–374, 1984.

[85] X. Messeguer. A sequential and parallel implementation of skip lists. Technical Report LSI-94-41-R, Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics, Barcelona, Spain, 1994.

[86] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings Symposium on Foundations of Computer Science*, pages 478–489, Oct. 1985.

[87] G. L. Miller and J. H. Reif. Parallel tree contraction. Part 2: Further applications. *SIAM Journal of Computing*, 20(6):1128–1147, Dec. 1991.

[88] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.

[89] P. Narayanan. Single source shortest path problem on processor arrays. In *Proceedings Frontiers of Massively Parallel Computation*, pages 553–556, McLean, VA, Oct. 1992.

[90] G. Narlikar. *Space-Efficient Multithreading*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. To appear.

[91] D. A. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.

[92] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In *Lecture Notes in Computer Science 143: Proceedings of the 10th Colloquium on Automata, Languages and Programming, Barcelona, Spain*, pages 597–609, Berlin/New York, July 1983. Springer-Verlag.

[93] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, University of Maryland Institute for Advanced Computer Studies, Dept. of Computer Science, University of Maryland, June 1990.

[94] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[95] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 275–340. Morgan Kaufmann, 1993.

[96] A. Ranade. Maintaining dynamic ordered sets on processor networks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 127–137, San Diego, CA, June-July 1992.

[97] A. Ranade. A simple optimal list ranking algorithm. 1998.

[98] M. Reid-Miller. List ranking and list scan on the Cray C-90. *Journal of Computer and System Sciences*, 53(3), Dec. 1996. Also in *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 104-113, Cape May, NJ, June 1994.

[99] M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 115–194. Morgan Kaufmann, 1993.

[100] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, CA, 1993.

[101] B. Schieber. Parallel lowest common ancestor computation. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 259–274. Morgan Kaufmann, 1993.

[102] R. Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, MA, 1990.

[103] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[104] J. Sipelstein. *Data Representation Optimizations for Collection-Oriented Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University. To appear.

[105] D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. *Journal of the Association for Computing Machinery*, 32(3):652–686, 1985.

[106] T. Stricker. *Direct Deposit: A Communication Architecture for Parallel and Distributed Programs.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1997. To appear.

[107] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk-5.0 (Beta 1) Reference Manual*, Mar. 1997.

[108] A. Sussman, J. Saltz, S. Gupta, D. Mavriplis, and C. R. Oibbysant abd J. Parti primitives for unstructured and block structured problems. *Computing Systems in Engineering*, 3(2):73–86, 1992.

[109] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[110] P. J. Varman, B. R. Iyer, D. J. Haderle, and S. M. Dunn. Parallel merging: Algorithm and implementation results. *Parallel Computing*, 15:165–177, 1990.

[111] U. Vishkin. Advanced parallel prefix-sums, list ranking and connectivity. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 215–257. Morgan Kaufmann, 1993.

[112] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[113] P. Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluations and Semantics-Based Program Manipulation*, pages 255–273, New Haven, Connecticut, June 1991. Also in SIGPLAN Notices; vol.26, no.9; September 1991.

[114] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, 1994.

[115] J. C. Wyllie. The complexity of parallel computations. Technical Report TR-79-387, Department of Computer Science, Cornell University, Ithaca, NY, Aug. 1979.

[116] M. Zagha. *Efficient Irregular Computation on High-Bandwidth Pipelined-Memory Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1998.

[117] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, Nov. 1991.