



Wait-free Parallel Algorithms for the Union-Find Problem

Richard J. Anderson*

University of Washington

Heather Woll†

University of California, San Diego

Abstract

We are interested in designing efficient data structures for a shared memory multiprocessor. In this paper we focus on the union-find data structure. Our machine model is asynchronous and allows stopping faults. Thus we require our solutions to the data structure problem have the *wait-free* property, meaning that each thread continues to make progress on its operations, independent of the speeds of the other threads. In this model efficiency is best measured in terms of the total number of instructions used to perform a sequence of data structure operations, the *work* performed by the processors. We give a wait-free implementation of an efficient algorithm for union-find. In addition we show that the worst case performance of the algorithm can be improved by simulating a synchronized algorithm, or by simulating a larger machine if the data structure requests support sufficient parallelism. Our solutions apply to a much more general adversary model than has been considered by other authors.

1 Introduction

In this paper we study the problem of designing efficient parallel data structures for a shared memory multiprocessor. We are interested in the amortized complexity of a series of data structure operations.

*Supported by an NSF Presidential Young Investigator Award CCR-8657562, Digital Equipment Corporation, NSF CER grant CCR-861966, and NSF/Darpa grant CCR-8907960. Email address anderson@cs.washington.edu.

†Partially supported by an NSF Research Initiation Award CCR-9009657 and a UCSD Faculty Career Development grant. Email address woll@cs.ucsd.edu.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 089791-397-3/91/0004/0370 \$1.50

For a p processor machine, we make the assumption that there are always at least p data structure requests that we can execute simultaneously. The specific case that we look is the union-find problem, where we maintain a collection of disjoint sets, and support the operations of merging subsets and testing if pairs of elements are in the same subset. Our solutions apply to a much more general adversary model than has been considered by other authors.

1.1 Motivation

The type of parallel machine that motivates our work is the shared memory multiprocessor. Machines of this type generally have a modest number of independent processors that communicate through a shared global memory. The operating system provides the user with a set of threads¹ which are executed by the processors. The threads are time-shared amongst the processors, so at a given time only a subset of the threads are being executed. When a thread is being executed it is subject to additional delays such as interrupts and page faults. The user's view of this is that he has a set of processors available, which run at highly variable rates. An algorithm designer attempts to find an algorithm that efficiently takes advantage of the computer time that is made available to the user.

An important aspect of this type of machine is that it is asynchronous. The asynchrony arises at both the hardware and the software levels. At the hardware level, asynchrony is caused by factors such as variable delay for bus access, and different costs for accessing cache versus global memory. At the software level, the asynchrony arises from interruptions such as page faults, and from threads being swapped out so that other threads may be run. The delays due to software are potentially much

¹We use the term *thread* in contrast to *process* to emphasize that a user's threads have access to a shared address space.

longer, and exert a stronger influence on our abstract model.

On this type of machine, it is expensive to have the threads synchronize. In particular, it is not practical to synchronize after every instruction to simulate a synchronized machine. To avoid synchronization overhead, it is often desirable to have the threads as independent as possible. A natural approach in designing algorithms for a shared memory multiprocessor is to have each thread execute a separate data structure request. When this is done, some method is needed to arbitrate over access to shared data structures. One way that this is commonly done is to use locks to give threads exclusive access to particular portions of the data. However, in this asynchronous domain locks can cause severe problems, since after a thread sets a lock, it may be delayed indefinitely, while all of the other threads wait at the lock.

1.2 Approach

In this work we want to use a simple model, that can capture the complexities of an asynchronous environment where it is not unusual for threads to suffer long delays. A key notion, which we borrow from the field of distributed computing, is a *wait-free* implementation of a data structure. A data structure is wait-free if any thread is guaranteed to complete an operation in a finite number of steps, independent of the processing speeds of the other processors. By requiring that our data structures be wait-free, we rule out the use of locks. The main thrust of research on wait-free objects has been to relate the powers of various primitives. It has been shown that there are some primitives, referred to as being *universal*, which are sufficiently powerful that they may be used to implement any wait-free data structure [Her88]. We include a universal primitive in our model.

In this paper, we are interested in measuring the performance of data structure operations as well as guaranteeing that they have wait-free implementations. We consider the cost of executing a series of data structure requests. This means that we want to measure the amortized cost of the sequence. We measure the cost of an algorithm by the amount of *work* that it performs. The work is defined to be sum of the active times of the threads. This measure gives us a convenient comparison with a sequential algorithm, since we can compare the work with p threads with the work with only one thread. (See section 3.1 explains why this is the most ap-

propriate measure for this model.)

A major difference between this work and many recent papers on parallel computation is that we are motivated by parallel machines with a small number of processors, as opposed to considering massively parallel machines. This means that we shall generally assume that the size of the problem is substantially larger than the number of processors. We will be considering performing a set of n data structure operations using p threads. We give our results parameterized in terms of both n and p . We are more interested in getting good results when n is substantially larger than p , than in maintaining good performance as p gets close to n .

1.3 Outline

In section 2, we survey related work, discussing work on wait-free objects, models of asynchronous computation, and parallel data structures. In section 3 we introduce our model and discuss our choice of primitives and the adversary used for worst case bounds. Section 4 gives our wait-free solution to the union-find problem and analyzes the algorithm's run time. Section 5 gives two methods for improving the performance if we have greater parallelism available than the number of threads. We consider both running our original algorithm where we randomly choose among the available requests and using additional parallelism to simulate a synchronized algorithm. The final section discusses directions in which this work could be extended.

2 Related Work

This paper has been influenced by two distinct bodies of work, work done in distributed computing on wait-free objects, and work done in the theory of parallel computing on identifying models of asynchronous computation. The notion of a wait-free object captures the idea that the algorithm must be resilient to the delays of threads. The work on parallel computation provides a basis for our performance measures. We give a brief survey of relevant papers in both fields.

2.1 Wait-free data structures

We say that an implementation of a data structure is wait-free if in a finite number of steps a thread is guaranteed to complete an operation, independent of the actions of the other threads. Any implementation that relies on locks cannot be wait-free since

after setting a lock, a thread may be delayed indefinitely while the other threads spin idly waiting for the lock.

There has been a substantial amount of study of wait-free objects. The emphasis of this work has primarily been on simple objects such as atomic registers and *test&set*. A series of papers has established relations between the primitives, either showing that one primitive could simulate another, or that such a simulation is impossible. Herlihy [Her88] unified much of this work by relating primitives to the a consensus problem and establishing the existence of universal primitives. A primitive is universal if any wait-free object can be constructed from it. Primitives such as an atomic append to a list and *compare&swap* are universal while *test&set* is not. A main drawback to the universality results is that the simulations of one primitive with another are often inefficient.

It can be shown that there is a hierarchy of wait-free primitives, with each level strictly more powerful than the preceding level. The universal primitives form the top of the hierarchy. Some common data structures, such as stacks and queues fall in intermediate levels. However, most of the complicated data structures used in programming are in the top level. The data structures at the top level have received relatively little attention. Herlihy [Her90] studied the implementation of a number of data structures in terms of *compare&swap*. He showed how to implement functional data structures, and discussed wait-free memory management.

2.2 Asynchronous P-RAM models

Researchers have recently begun to consider asynchronous versions of the P-RAM [Gib89,RZ89,RZ90,Nis90,MSP90]. The papers have introduced several different models, with differing notions of run time. Cole and Zajicek introduced an asynchronous P-RAM model where the run time is measured in rounds, where a round is a minimal interval of time that allows each processor to complete one step. This in effect measures time in terms of the slowest processor. This is not an appropriate measure when processors are subject to long delays, since while one processor is delayed, all other computation takes place for free. Nishimura [Nis90] and Cole and Zajicek [RZ90] analyze run time assuming random interleaving of processors. This model applies best when processors run at close to the same speed as opposed to stopping for long delays.

Cole and Zajicek do not specify the atomic primitives for the A-PRAM although implicitly assume that an atomic assignment can be done to several variables simultaneously. Aspnes and Herlihy [AH90] consider an A-PRAM that does not support universal primitives. They study the limitations of such machines and classify the computations that can be performed.

The works that present a model that is closest to ours are a series of papers considering fault tolerant P-RAMs [KS89,KPS90,MSP90]. These papers consider a model where processors may fail, and the goal is to simulate the computation using the surviving processors. The performance measure used in the papers is the amount of work done, the same measure as used in this paper. The model used by Martel et al. differs from ours in two respects. First, they use a primitive that appears to be somewhat weaker than *compare&swap*, and second, they use a much weaker adversary model. We shall return to the adversary below.

3 Model

Our basic model is an asynchronous shared memory machine. Each processor can execute the instructions of a standard Random Access Machine. There is a shared global memory, and each processor also has its own local memory. The instruction set also includes the *compare&swap* primitive. All of the instructions are atomic. An execution of this machine is an interleaving of the atomic instructions. Since the instructions are serialized, the issue of whether the reads and writes are concurrent or exclusive does not arise. All of the processors can perform a simultaneous write to a memory cell, but the execution will cause these writes to occur in some serial order.

3.1 Measure of Performance

To measure the complexity of algorithms we consider the total number of instructions executed by active threads. The *work* of an algorithm is the maximum such total over all possible execution sequences. It is clear that the work measure is equivalent to the time measure in the sequential model of computation and that work is the product of time and the number of processors in a synchronous model of parallel computation. In the asynchronous world the work measure for wait-free algorithms is a generalization of the time-processor product measure. For a fixed distribution of activation periods of the threads, a minimum work algorithm gives us

a minimum time algorithm. Since we do not have control over the activation periods, designing an algorithm to minimize the work, gives us one that maximizes efficiency of the processors.

3.2 Data structure queries

We are interested in finding a minimum work algorithm for executing a series of n data structure operations. We assume that there are always p operations that can be executed concurrently. Every time a thread completes a data structure request, it calls a routine to generate the next request. We assume that there is some underlying algorithm generating and making use of these data structure operations. When there are no more operations to perform, a requesting thread is able to deactivate itself and no longer counts towards the total work.

3.3 Correctness

The standard approach for defining correctness of an asynchronous algorithm is to assume that the atomic instructions of all of the threads are interleaved in some linear order. In order for an algorithm to be correct it must behave properly for all interleavings [HW87]. Our definition of what it means to correctly answer the data structure operations is also done in terms of serializability. The answers to the queries are said to be correct if the answers agree with some serialization of *atomic* queries. This serialization is also required to be consistent with the order in which the threads execute the queries.

3.4 Adversary

We express our worst case bounds in terms of an adversary. The adversary both chooses the queries and establishes the interleaving order. If our algorithm is randomized, then the adversary also may look at the results of the random coin before deciding on which thread has the next instruction in the interleaving. This adversary model has been used in [And90] and [BR90].

The strength of the adversary plays an important role in the results that one can establish. The adversary used by Martel et al. [MSP90] is substantially weaker than the adversary used here. Their adversary sets the complete interleaving order *before* it sees the results of any of the random numbers. The work of the algorithm given in the Martel et al. paper increases from $\Theta(n)$ to $\Theta(pn)$ against the more powerful adversary.

3.5 Primitives

In this work, we are willing to use a universal primitive. Although universal primitives are all equivalent in terms of computational power, the simulations between universal primitives are not necessarily very efficient. This means that our choice of primitive will have a strong influence on our results. We want to use a primitive that is simple, possible to implement in hardware or software², and has been accepted by other researchers. Our choice is *compare&swap*, which is an assignment that succeeds only if we know the current value of the variable. The *compare&swap* primitive was used by Herlihy in his work on wait-free data structures, and is included in the instruction set of the IBM 370 [Her90].

The following code fragment defines *compare&swap*. Our main use of this primitive is to ensure that a variable is not overwritten by another thread between reading and update the variable.

```
Compare&Swap( $x, a, b$ )
  if  $x = a$  then  $x := b$ ; return SUCCESS;
  else return FAILURE;
```

It is possible to extend *compare&swap* to apply to records instead of single words. This is done by making copies of records, and then applying *compare&swap* to a pointer to the record to make sure that the record has not been updated by another thread. We give our version of *compare&swap* that applies to records below.

4 Union-Find

The union-find data structure maintains a collection of disjoint subsets of $\{1, \dots, n\}$, and supports a *Union* operation which merges a pair of subsets, and *Find* which identifies the current subset containing the given element. We augment the operation to include an operation *Sameset* which tests if two elements are contained in the same subset. *Sameset* is important to have in a concurrent implementation, since the names of sets can change, making this test difficult to perform if it is not provided as a primitive. In this paper we give an implementation based on a *ranked union* with *path halving* [Tar83]. (We can also adapt *path compression* to a

²A software implementation of a primitive might guarantee that the operating system will not deactivate a thread while it is executing the primitive.

wait-free implementation.) The sequential cost for a sequence of n Union-Find operations with ranked union and either path halving or path compression is $n\alpha(n)$.

There are a number of difficulties that arise in developing a concurrent Union-Find data structure. For example, care must be taken to make sure that simultaneous Unions do not interfere and destroy the tree structure. When we prove that our algorithm correctly implements the Union-Find data structure, we restrict ourselves to Union and Same-set operations. The reason for this is that the value of the Finds depend upon the structure of the trees, and it is possible that the structure of the trees may differ between a concurrent, and a serialized set of queries. Most applications of Union-Find, such as constructing minimum spanning trees, only use Finds to test if elements are equivalent.

4.1 Basic Data Structure

In order to give a precise definition of our algorithm, it is necessary to present the algorithm at a very low level. We use a notation derived from the C language to express our manipulation of pointers and arrays. The standard sequential data structure is to have a set of records that point to each other to form a forest. Each record has a *next* field which is a pointer to another record, and a *rank* field which is used to keep information that aids in balancing the trees. If a record's next pointer points back to the record, then the record is the root of a tree. In our implementation, we introduce an additional level of indirection. We maintain an array $A[1 \dots n]$, where each entry is a pointer to a record with a next field, and a rank field. The next field is an array index. In our C-like notation, the rank field for element x is: $A[x] \rightarrow \text{rank}$.

We use an ordering of the records based on their ranks. We say that the record for x is less than the record for y , if $A[x] \rightarrow \text{rank} < A[y] \rightarrow \text{rank}$ or if $A[x] \rightarrow \text{rank} = A[y] \rightarrow \text{rank}$ and $x < y$. We use the notation $x \prec y$ when the record for x is strictly less than the record for y , and $x \preceq y$ when equality is allowed.

In order to achieve wait-free algorithms, we must be able to perform an atomic update of both the rank and next fields of a record. Instead of relying on a general operation for an atomic update of a record, we use a subroutine that updates the next and rank fields of a record that is the root of a tree. The reason we use a restricted operation is that it allows us to simplify the code that we present. If x

is a root with rank *oldrank* then an atomic update of x 's next and rank fields are performed, otherwise FAILURE is returned.

```
UpdateRoot( $x$ ,  $oldrank$ ,  $y$ ,  $newrank$ )
   $old := A[x]$ ;
  if  $old \rightarrow next \neq x$  or  $old \rightarrow rank \neq oldrank$ 
    return FAILURE;
   $new := CreateRecord()$ ;
   $new \rightarrow next := y$ ;
   $new \rightarrow rank := newrank$ ;
  return Compare&Swap( $x$ ,  $old$ ,  $new$ );
```

4.2 Basic algorithms

We now present our implementations in full detail. We give the code for Find, SameSet, and Union in the following subsections.

In designing the routines, there were a number of pitfalls to avoid. The main difficulties were to prevent one thread's updates from disrupting another thread, and to make sure separate threads did not have inconsistent view of the data. The main tool to arbitrate between threads is the *compare&swap* primitive. If one thread's update is blocked by another thread, then the operation is retried.

4.2.1 Find

We implement our find using *path-halving*. As we traverse a path to a root, we make each node we visit point to its grandparent. The benefit of this method is that it halves the distance of each node to the root.

We implement this heuristic by using *compare&swap* to assign the new value to the next field. This solves the problem of two or more threads concurrently updating the same pointer. We note that we can use *compare&swap* instead of the more expensive operation *UpdateRoot*, since we only update non-root nodes in Find, and *UpdateRoot* is only used on root nodes.

```
Find( $x$ )
  while  $x \neq A[x] \rightarrow next$  do
     $t := A[x] \rightarrow next$ ;
    Compare&Swap( $A[x] \rightarrow next$ ,  $t$ ,  $A[t] \rightarrow next$ );
     $x := A[t] \rightarrow next$ ;
  return  $x$ ;
```

4.2.2 Sameset

The operation Sameset tests if two elements are in the the same subset. The Sameset operation is unnecessary in the sequential case, since it can be constructed from a pair of finds. However, in the concurrent case, it is important to provide it as a primitive, since the names of sets may change, making it difficult to tell whether or not a pair of elements are in the same set. Our algorithm locates the roots for x and y . If the algorithm can give a definite answer, then it does, otherwise, it finds new roots for the elements and tries again. The subtlety is that the elements may cease to be roots, and we must guard against thinking the elements are in different sets just because the root x and y are different. If $x \neq y$ and $A[x] \rightarrow next = x$, we can safely say that x and y were in separate components when y was identified as the root.

```

Sameset( $x, y$ )
  TryAgain:
     $x = Find(x)$ ;  $y = Find(y)$ ;
    if  $x = y$  then return TRUE;
    if  $A[x] \rightarrow next = x$  then return FALSE;
    go to TryAgain;

```

4.2.3 Union

Our Union algorithm is an implementation of *Ranked Union*. Each record maintains a rank, and links are made from a record of smaller rank to a record of larger rank. If records of equal rank are linked, then the new root has its rank incremented. It is an easy exercise to show that the height of a tree formed by ranked unions is logarithmic in the number of vertices. In the sequential case, along any path to the root, the ranks are strictly increasing.

Several difficulties arise in a concurrent implementation of Union. The key idea that prevents cycles is to perform links in a direction consistent with our ordering on records. If x and y are roots with $x \prec y$, then we link from x to y . A subtler problem that arises is to ensure that different threads have consistent views of the data. We must prevent one thread from viewing $x \prec y$ and another viewing $y \prec x$. We do this by aborting a link from x to y if the rank of x changes between when it was identified as the lesser of the two roots, and when the assignment is to take place.

When we update the rank, we must guard against several threads incrementing the rank simultaneously. By using *UpdateRoot*, updating the rank only

succeeds if a thread is the first to update the rank of the new root, and if the new root is still a root. If the new root ceases to be a root before its rank is updated, then we can get adjacent nodes of the same rank. As we traverse a path from a node to the root, the ranks are non-decreasing, but not necessarily strictly increasing.

```

Union( $x, y$ )
  TryAgain:
     $x := Find(x)$ ;  $y := Find(y)$ ;
    if  $x = y$  then return ;
     $xr := A[x] \rightarrow rank$ ;  $yr := A[y] \rightarrow rank$ ;
    if  $xr > yr$  or ( $xr = yr$  and  $x > y$ )
      Swap( $x, y$ ); Swap( $xr, yr$ );
    if UpdateRoot( $x, xr, y, yr$ ) = FAILURE
      go to TryAgain;
    if  $xr = yr$  then
      UpdateRoot( $y, yr, y, yr + 1$ );

```

Lemma 4.1 *The pointer structure is a forest of in-trees.*

We can now give a formal proof that the concurrent implementation correctly implements union-find. The basic idea behind the proof is that we can identify some particular instruction in the interleaving where we can view each operation as occurring. We restrict ourselves to Union and Sameset queries.

Theorem 4.2 *Let R be a set of Union and Sameset operations. Let I be an interleaving of the atomic instructions of the threads executing R . There exists a serialization of R , consistent with the threads' order of request, giving exactly the same responses to the queries as I .*

4.3 Performance

The implementation above suffers from a major performance flaw: there is a sequence of requests and an interleaving of instructions that creates arbitrarily long chains of nodes with identical rank.

Lemma 4.3 *For any k , there exists a sequence of requests, and an interleaving of instructions that creates a chain of k records with equal rank. This can occur if there are only two threads.*

There is a relatively simple fix that alleviates this problem of creating long chains. The problem arises because one thread can initiate a second operation

before other processors detect that it has completed the first operation. The fix is to perform the routine *SetRoot* on one of the elements involved in the link at the end of the Union. (If the Union links x to y , then the call *SetRoot*(x) is made.) *SetRoot* is just like Find, except that it also makes sure that the root has greater rank than the node immediately before the root on the find path. For our purposes, it is important the *SetRoot* both compresses the path and updates the rank of the root.

```

SetRoot( $x$ )
 $y := x$ ;
while  $y \neq A[y] \rightarrow next$  do
     $t := A[y] \rightarrow next$ ;
    Compare&Swap( $A[y] \rightarrow next, t, A[t] \rightarrow next$ );
     $y := A[t] \rightarrow next$ ;
UpdateRoot( $y, A[x] \rightarrow rank, y, A[x] \rightarrow rank + 1$ );

```

We now show that by adding *SetRoot* to the Union, we reduce the length of a chain of equal ranked records to $3p$. We do not know what the correct upper bound is. Our best guess is that the bound is $2p$.

Lemma 4.4 *The modified algorithm cannot create a chain of records of equal rank of length greater than $3p$.*

These chains of length p can be constructed by a set of simultaneous unions performed by all processors. This sets the stage for somewhat disappointing worst case performance. After creating a chain of length p we can have p threads simultaneously traverse the chain. If the threads work in lock step, each thread does p work in spite of the path halving. By repeatedly creating chains and traversing them we have worst case behavior.

There is a second case that causes $\Omega(pn)$ work. When a Union is executed, a thread reexecutes the two Finds if the *UpdateRoot* fails. The *UpdateRoot* fails if either $A[x] \rightarrow next$ or $A[x] \rightarrow rank$ has changed between when the Finds were performed and when the link was attempted. We refer to these as *next failures* and *rank failures* respectively. To bound the total amount of work, we must bound the number of times that *UpdateRoot* can fail. We can bill each next failure to the next step performed in a Find, so we only need to concern ourselves with rank failures.

Theorem 4.5 *In the worst case, there are $O(pn)$ rank failures.*

Proof: A rank failure occurs when x is to be linked to y , and the rank of x changes before the link is performed. The worst case occurs if $p - 1$ threads are simultaneously performing a Union that has a rank failure. This means that each change in rank can account for $p - 1$ rank failures. The total number of rank changes is $O(n)$, giving a bound of $O(pn)$ for the number of rank failures. ■

We can now describe the worst case performance of union-find.

Theorem 4.6 *The worst case work for a series of n union-find operations is $\Theta(pn + n\alpha(n))$.*

The worst case bound of $\Theta(pn)$ work is not very good when it is compared with the sequential bound of $\Theta(n\alpha(n))$. This says that in the worst case, we get essentially no speedup using p processors. We can put a better spin on the result. The worst case only arises from a very tight interleaving of instruction where there is a high degree of contention in updating the fields of records. On a real multiprocessor, there will be enough variation in the rate that the instructions are executed, so that the contention will be reduced, and performance will not be as bad as our bound suggests, even on a worst case set of requests. We can discuss this in a formal setting by counting the number of times that this contention occurs, and parameterizing our work bounds by this quantity.

We have defined a *rank failure* to occur when a link fails because the rank of the record changes before the update is made. We now define a *compression failure* to occur when *Compare&Swap* fails inside of a Find. This type of failure occurs when two threads are traversing a chain in lock step, and the threads duplicate each other's work in splicing out elements of the chain. We shall let F denote the total number of rank and compress failures. Our bounds improve substantially when we account for these failures separately.

Theorem 4.7 *The worst case work for a series of n union-find operations when there are at most F rank and compress failures is $O(np^{1/2} + n\alpha(n) + F)$.*

Proof: The dominant term is still the cost of traversing long chains. However, if we count the compress failures separately, we can treat the updating of pointers during Finds as atomic operations that splice elements out of a list. The result follows from a bound given in [And91]. ■

4.4 Adversary strength

The adversary that we are considering has the power to decide upon the requests and to set the interleaving of threads' instruction streams. If the threads have access to random numbers, the adversary can base the interleaving on the random numbers. This gives the adversary considerable power. We present a pair of results to indicate the power that the adversary has. The results show why some natural approaches to improving the worst case bounds of our algorithms do not work.

One of the main difficulties we encounter in union-find is in having a set of processors simultaneously traverse a long chain. We can prove that in a pointer based model (where processors can only access memory cells to which they have links to), the adversary can force p threads to expend $\Theta(p^2)$ work in traversing a chain of length p . This result applies even if the threads are allowed to use randomization.

Theorem 4.8 *Suppose p threads simultaneously traverse a chain of length p . The adversary can force $\Theta(p^2)$ work to be performed.*

Proof: The adversary forces all of the threads to remain in lock step. Suppose that the chain is $x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_p$. All threads start out at x_1 . The adversary picks a thread and allows it to execute until it reads x_2 . The thread is then halted until all threads have read x_2 . The thread is then reactivated and allowed to read x_3 . This construction proceeds in phases, so that during the j -th phase threads proceed from x_j to x_{j+1} . Since the adversary keeps the processors in lock step, they are not able to benefit any path compression that might be performed. ■

Since the adversary can make the traversal of long chains expensive, we would like to avoid construction of long chains. We show that one natural randomized algorithm fails. Suppose we adopt the following strategy: whenever a Union is performed on elements of equal rank, the direction of the link is chosen at random. We can show that an adversary can create a chain of length $\frac{p}{2 \log p}$ with high probability.

The results show the power that the adversary has in defeating various algorithms. One can achieve far better bounds for the union-find problem if the adversary is weakened, or if we change our performance measures. The average case run time

(over a worst case set of requests) improves substantially if we take the random interleaving model of Nishimura [Nis90]. Under a random interleaving model, it is safe for a thread to stop working when contention is detected, since the other threads are not likely to be stopped indefinitely. If the adversary is not allowed to base its interleavings on the results of a random number generator, then we can get a better worst case bound. Martel et al. [MSP90] prevent the adversary taking advantage of the random number generator by having the adversary set the interleaving order before the random numbers are generated. Against this weaker adversary it is possible to prevent long chains from arising by techniques such as having the threads probabilistically decide whether or not their links succeed.

5 Improved Work Bounds

In this section, we discuss two very different techniques for improving the worst case performance of the Union-Find algorithm. The first of these is based upon simulating a synchronous algorithm with asynchronous threads. Although this gives the strongest theoretical result, it is unsatisfying since it is clearly not practical on a real machine. Our other approach requires having a more queries available for processing than we have threads. We are able to trade excess parallelism for improved worst case performance. This approach is more attractive, since it is course grained and hence more likely to be practical when implemented.

5.1 Step by step simulations

The first approach we consider is based on having our p threads simulate a synchronous algorithm also running on p processors. Each round simulates exactly one instruction of each of the p synchronous thread. This type of simulation is discussed for other asynchronous models in several papers [KS89, KPS90, MSP90].

It is not difficult to design a synchronized Union-Find algorithm that is almost as efficient as the sequential algorithm. The source of inefficiency in our asynchronous algorithm arises from long chains being created. In a synchronized algorithm we can avoid long chains by breaking them up as they are formed. We do this by simply applying the Cole-Vishkin algorithm for finding an independent set in a list [CV86]. This causes us to spend $\log^* p$ extra time for each linking step. This gives us the following lemma.

Lemma 5.9 *A synchronous algorithm can answer n Union-Find queries with $O(n(\alpha(n) + \log^* p))$ work.*

In order to simulate a synchronous algorithm, we have each thread execute the instructions of each processor. The abstraction that is used to capture this is a problem introduced by Kanellakis and Shvartsman [KS89] called the write-all problem. The write-all problem is to write ones into an array of size p . When a thread writes a one into location j it executes the instruction associated with thread j in the current round. When ones have been written into all locations, then the round is complete, so the next round can be performed.

There are a number of important details to take care of to turn an algorithm for the write-all problem into a simulation of a synchronous algorithm. First of all, it is possible that two threads will attempt to execute the same instruction at the same time. It is possible to set up the rounds so that multiple executions of instructions do not change the result. This is referred to as making the computation *idempotent*. A second problem to take care of is to ensure that all threads are working on the same round. It is possible for a thread to be delayed in the middle of executing an instruction, and by the time it is reactivated the other processors have finished off the round and started a new round. We can solve this problem by associating a tag with each memory cell giving the last time that it was written. When we update a tag, we use the *Compare&Swap* operation to avoid difficulties with concurrent updates. The fact that we have a more powerful model makes it easier for us to resolve this difficulty than have others using weaker models [MSP90].

In the next subsection, we show that one round of the synchronous algorithm can be simulated with work $O(p^{1+\epsilon})$ for any fixed $\epsilon > 0$. This gives us the following result:

Theorem 5.10 *By simulating a synchronous algorithm, a sequence of n Union-Find operations can be executed with $O(np^\epsilon(\alpha(n) + \log^* p))$ work for $\epsilon > 0$.*

This type of simulation can be done in our framework, and gives an improved result, so it is appropriate to consider. However, a step by step simulation is not a practical solution for a shared memory multiprocessor, so it does suggest a gap between our theoretical framework and the situation that we are attempting to model. The constant factors involved in this solution are too large to be considered practical. The first problem is involved in

the synchronous Union-Find algorithm. The step of performing a deterministic coin-tossing for each linking step introduces a large overhead. However, the real problem comes from the simulation. There are three separate problems with the simulation, each one of them probably sufficient to rule out practical implementation. First of all, requiring the tagged memory for every write would double storage, and greatly slow down the computation. Secondly, the simulation would require synchronizing at every time step. This is a far finer granularity of synchronization than is feasible on the type of machine being modelled. Finally, the process of having each thread execute the instructions of a synchronized algorithm means that each thread is acting as an interpreter, thus the advantage of compiled code is abandoned.

5.2 The write-all problem

In this section we give a solution to the write-all problem. Our result differs from the result of Martel, in that our algorithm is applicable to a stronger adversary. The write-all problem for the stronger adversary model has been previously considered by Buss and Ragde [BR90]. They give an upper bound of $O(p^{\log_2 3})$ work. Our algorithm includes their's as a special case. They also show an $\Omega(p \log p)$ work lower bound for the write-all problem.

Let S_n denote the set of permutations over $[n] (= \{1, \dots, n\})$, and call a subset R of S_n a *random k -subset* of S_n if it is constructed by choosing each of its k elements randomly and uniformly from S_n . We calculate the *contention* of this order π with respect to the actual order α in which the queries are processed as follows: Suppose the elements of $[n]$ are in a stack in the order π , and these elements disappear one by one in the order α . The *contention* of π with respect to α , $\text{contention}(\pi, \alpha)$, is the number of times that the next element to disappear is on the top of the stack. For a set R of permutations over S_n , we define the *contention* of R with respect to α as $\text{contention}(R, \alpha) = \sum_{\pi \in R} \text{contention}(\pi, \alpha)$, and the *maximum contention* of R as $\text{contention}(R) = \max_{\alpha \in S_n} \text{contention}(R, \alpha)$.

The lemmas that follow motivate our algorithms.

Lemma 5.11 *There is a constant c such that if α is a fixed element of S_n and π is a random element of S_n then the expected value of $\text{contention}(\pi, \alpha)$ is at most $c \log n$.*

Lemma 5.12 *There is a constant c such that if R is a random n -subset of S_n then the probability that $\text{contention}(R) > cn \log n$ is less than $\frac{1}{n!}$.*

Corollary 5.13 *There is a constant c such that for each n there is an n -subset of S_n with contention at most $cn \log n$.*

Lemma 5.14 *Suppose there are q high level instructions, n threads partitioned into q groups (g_1, \dots, g_q) of size $\frac{n}{q}$ and a q -subset $R = \{\pi_1, \dots, \pi_q\}$ of S_q with $\text{contention}(R) \leq cq \log q$. If each processor in group g_i performs unfinished instructions in the order π_i then the amount of work used to execute the q instructions is $O(cq \log q W(\frac{n}{q}))$, where $W(\frac{n}{q})$ is the amount of work used when $\frac{n}{q}$ threads execute a single instruction.*

Lemmas 5.12 and 5.14 motivate a randomized algorithm for the case where there are p threads and p^2 instructions. Let the instructions be partitioned into p groups $\{g_1, \dots, g_p\}$ of size p and define the high level instructions $\text{Inst}(1), \dots, \text{Inst}(p)$ to consist of the sequential execution of the instructions in g_1, \dots, g_p , respectively. Each thread randomly chooses an element π of S_p , and then attempts to execute the high level instructions in the order $\text{Inst}(\pi(1)), \dots, \text{Inst}(\pi(p))$. Once a thread has successfully completed a low level instruction, it marks the instruction as completed, so no thread that follows need consider the instruction. Similarly, the once a thread has completed all of the instructions in a group it marks the group as completed so no following thread need consider that group. Thus the following theorem holds.

Theorem 5.15 *There exists a randomized algorithm for p threads that simulates p^2 instructions with $O(p^2 \log p)$ work with high probability.*

A deterministic algorithm can be constructed using the basic ideas contained in corollary 5.13 and lemma 5.14. Unfortunately, this deterministic algorithm is not as efficient, but no extra processors are required.

Suppose $n = q^d$ for some integers q and d . We can label each of the q^d requests (or the q^d threads) by unique string of length d over the alphabet $A = \{1, \dots, q\}$. We can imagine that the requests are stored at the leaves of a q -ary tree, with internal nodes labelled by strings of length less than

d , such that if v is a parent of w then $\text{label}(v) = a_1 a_2 a_3 \dots a_i$ and $\text{label}(w) = a_1 a_2 a_3 \dots a_i a_{i+1}$ for some nonnegative $i < d$ and some labels a_1, \dots, a_{i+1} in A . (The root of the tree is labelled by the empty string.)

The basic code is that of the *Traverse* procedure for a fixed values of q and d and fixed set $\{\pi(1), \dots, \pi(q)\}$ with contention at most $cq \log q$. Each thread makes the following initial call to the *Traverse* procedure: *Traverse*(0, λ , $\text{label} = a_1 \dots a_d$), where λ is the empty string and the label of the thread is $a_1 \dots a_d$. The procedure *Traverse* calls a *Process* procedure that causes the thread to execute the instruction labelled by the *root* string. It also calls a *Mark* procedure to mark the tree rooted at *root* as completed, and a *ReadMark* procedure accordingly. (Read the symbol $*$ as a string concatenation operator.)

```

Traverse( $i$ ,  $\text{root}$ ,  $\text{label} = a_1 a_2 \dots a_d$ );
if ( $i = c$ ) then Process( $\text{root}$ )
else begin
  for  $j := 1$  to  $q$  do
    if ReadMark( $\text{root} * \pi_{a_{i+1}}(j)$ ) = false then
      Traverse( $i + 1$ ,  $\text{root} * \pi_{a_{i+1}}(j)$ ,  $\text{label}$ );
  Mark( $\text{root}$ );

```

The algorithm *Traverse* above together with corollary 5.13 and lemma 5.14 can be used to derive the following theorem.

Theorem 5.16 *For every $\epsilon > 0$, there exists a deterministic algorithm for p threads that simulates p instructions with $O(p^{1+\epsilon})$ work.*

5.3 Coarse grained algorithm

Instead of simulating a synchronized algorithm, we can take advantage of additional parallelism in a different way. If we have more operations available than threads, we have to allocate operations to threads. It turns out that if each thread selects operations *at random*, then the worst case performance improves dramatically. This method of defeating an adversary is much more appealing than simulating synchronization, since it is a method that could easily be implemented in practice.

Our proof shows that we can get an improvement in worst case performance of a factor of roughly $p^{1/2}$ by this method. Our result is for a general adversary that chooses the data structure operations and

sets the interleaving. The adversary also has full access to the state of the threads when deciding on the interleaving. The idea behind this theorem is that if the adversary attempts to create a bad case by building long chains, then the adversary has to decide on a large number of the requests. After the adversary has decided on these requests, most of them will turn out to be redundant, and hence easy to process.

Theorem 5.17 *The expected work for n union-find operations is $O(n(p^{1/2} \log p + \alpha(n)))$ when threads randomly select their current operations from a set of p^2 available operations.*

We believe that it should be possible to tighten the proof, to give a total work of approximately $O(n \log p)$. This would show that the course grained approach is almost as good as the synchronous approach.

6 Research Directions

We believe that there is still much work to be done in studying data structures for asynchronous parallel machines. The natural direction to extend this work is to look at other data structures. There are many interesting questions on how several different data structures can be maintained simultaneously, such as in a minimum spanning tree algorithm, where we want to keep track of disjoint sets, each of which has an associated heap. Another interesting direction of work would be to consider other universal primitives instead of *compare&swap*. Possible primitives to consider include an atomic splice out primitive, and versions of *compare&swap* that allow multiple variables to be updated.

References

- [AH90] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Second Annual Symposium on Parallel Algorithms and Architectures*, pages 340–349, 1990.
- [And90] R. J. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *Second Annual Symposium on Parallel Algorithms and Architectures*, pages 95–102, 1990.
- [And91] R. J. Anderson. Wait-free primitives for list compression. Work in progress, 1991.
- [BR90] J. Buss and P. Ragde. Certified write-all on a strongly asynchronous PRAM. Preliminary Report, 1990.
- [CV86] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Computation*, 70:32–53, 1986.
- [Gib89] P. Gibbons. A more practical PRAM model. In *1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.
- [Her88] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 276–291, 1988.
- [Her90] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, 1990.
- [HW87] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 13–26, 1987.
- [KPS90] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proceedings of the 22nd ACM Symposium on Theory of Computation*, pages 138–148, 1990.
- [KS89] P. Kanellakis and A. Shvartsman. Efficient parallel algorithms can be made robust. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 211–222, 1989.
- [MSP90] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *31st Symposium on Foundations of Computer Science*, pages 590–599, 1990.
- [Nis90] N. Nishimura. Asynchronous shared memory parallel computation. In *Second Annual Symposium on Parallel Algorithms and Architectures*, pages 76–84, 1990.
- [RZ89] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.
- [RZ90] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Second Annual Symposium on Parallel Algorithms and Architectures*, pages 85–94, 1990.
- [Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.