# Wait-free Parallel Algorithms for the Union–Find Problem

Richard J. Anderson*
University of Washington
Heather Woll†
University of California, San Diego

November 1, 1994

**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. We consider a fully asynchronous model of computation where arbitrary delays are possible. 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.

A preliminary version of this paper was presented at the 23rd STOC, 1991, [AW91].

*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`.

# 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. 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 at in this paper 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 set. Our solutions apply to a much more general adversary model than has been considered by other authors.

This paper makes a number of contributions. We use a model of asynchronous parallel computation that incorporates ideas developed in the field of distributed computing. We feel that these ideas have a very important role to play in understanding algorithms for parallel machines that are not fully synchronous. In the paper we give a simple asynchronous implementation of the Union-Find data structure that incorporates the well known sequential heuristics. We provide a correctness proof for our algorithm. The performance of the algorithm is close to the performance of the sequential Union-Find, except for certain pathological cases. We study two methods for improving the worst case performance. One method involves simulating a sequential machine. Our results include an improved result for the Write-All problem [BR90, KS89] with respect to a general adversary. We also introduce a coarse grained technique for improving the worst case performance of the algorithm that does not involve synchronization. The latter approach would be far more reasonable in a real implementation.

## 1.1 Motivation

The class of parallel machine that motivates our work is the shared memory multiprocessor. This is perhaps the simplest design for a parallel computer, and there are a number of shared memory multiprocessors that are commercially available. Examples of machines in this class include: the Sequent Symmetry, the DEC Firefly and the Encore Multimax. 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 threads[1] 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 may be active. 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

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

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

On this type of machine, it is expensive to synchronize threads. 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 a simple model that captures the complexities of an asynchronous environment where it is not unusual for threads to suffer long delays. A key idea, 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 for an argument as to why this is the most appropriate performance 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 worst case performance of our algorithm. One method is to simulate a synchronized machine and involves giving a new solution to the Write-All problem. The other approach is a coarse grained approach which requires having extra parallelism available, but has far lower overheads and would be far more reasonable in practice. The coarse grained approach is to run the original algorithm on a random selection of active requests. Section 6 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 models of asynchronous computation. An implementation of an object or data structure is wait-free if active threads can make progress in the computation independent of the behavior of the the other 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 relationships between the primitives, either showing that one primitive can simulate another, or that such a simulation is impossible. Herlihy [Her88] unified much of this work by relating primitives to a family of consensus problems 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. Plotkin [Plo89] showed that there is a single bit primitive

(called a sticky bit) that is also universal. 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 measure of run time has also been used in the field of distributed computing [Lam78, LF81] One way of viewing this is that time is measured with respect to 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, Nis91] 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 periods of time.

Cole and Zajicek do not specify the atomic primitives for the A-PRAM although they 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. They measure performance in terms of *work*, the sum of the active times of the processors, the same measure as used in this paper. These papers all look at the problem of how to simulate a single step of an $n$ processor algorithm on a fail-stop or asynchronous machine. Kanellakis and Shvartsman [KS89] introduced the Fail-Stop P-RAM model and the write-all problem. Kedem et al. [KPS90] gave a very general simulation result based upon a robust implementation of a variant of the write-all problem. Martel et al. [MSP90] extended the result to an asynchronous model by giving an asynchronous algorithm for the write-all problem. Their main result is a probabilistic simulation of an $n$ processor algorithm using $\frac{n}{\log n \log^* n}$ processors and $O(n)$ work. The model used by Martel et al. differs from ours in two important 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. In our exposition, it will be convenient to view certain operations as occurring simultaneously. When we say a set of operations occur simultaneously, we mean that the operations occur consecutively in the interleaving, with no other operations between them.

## 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. .

Our formal model of this is that there is collection of global memory locations that communicate the data structure requests to the threads. The memory location $q_i$ contains the current query for thread $i$. When thread $i$ completes the query, it writes the answer to

location $a_i$. This causes a new query to appear in $q_i$. After $n$ queries have been assigned to threads, all subsequent queries are *Halt*, which allows a thread to enter a special halt state.

## 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, Lam79]. 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.

We can formalize this definition of correctness. Let $Q_1, \ldots, Q_n$ be the queries performed (listed in order of request by the threads), and let $A_i$ be the answer returned from $Q_i$. The answers correspond to a serialization of the queries if there exists a permutation $Q_{\sigma_1}, \ldots, Q_{\sigma_n}$ of the queries, such if the queries are executed in the order $Q_{\sigma_1}, \ldots, Q_{\sigma_n}$ on a single processor, the sequence of answers is $A_{\sigma_1}, \ldots, A_{\sigma_n}$. To be consistent with the order that the threads executed the queries, if in the parallel execution $Q_i$ was finished before $Q_j$ was started, we must have $\sigma_i < \sigma_j$. The ordering of queries starting and finishing is given by the serialization of the instructions of the threads.

## 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 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 may have a strong influence on our results. We want to use a primitive that is simple, possible to

implement in hardware or software[2], 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 updating 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. This does rely on having a storage allocator that returns "clean" storage. 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 $\{1, \ldots, n\}$, and supports a *Union* operation which merges a pair of subsets, and *Find* which identifies the current subset containing the given element. We also 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. The Union-Find data structure is usually implemented by a collection of in-trees, where the root of a tree gives the current name of a subset. Union is implemented by making the root of one tree point to the root of the other tree, and Find is implemented by following a path from the given element to the root of its tree. There are a number of implementations of both Union and Find [Tar75, TvL84]. In this paper we give an implementation based on a *ranked Union* with *path halving* [Tar83]. (We can also adapt *path compression* to a 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 Sameset 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

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

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 precise definitions of our algorithms, it is necessary to present the algorithms at a very low level. We use a notation derived from the C language to express our manipulation of pointers and arrays [KR78]. The standard sequential data structure for Union-Find is a collection 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 $A[1 \ldots n]$, 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: $A[x]\text{->}rank$.

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 $A[x]\text{->}rank < A[y]\text{->}rank$ or if $A[x]\text{->}rank = A[y]\text{->}rank$ 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 called *UpdateRoot* 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. We assume that we have a storage allocator, *CreateRecord*, that returns a record of the appropriate size that is not pointed to by any live pointer.

```
UpdateRoot(x, oldrank, y, newrank)
    old := A[x];
    if old->next ≠ x or old->rank ≠ oldrank return FAILURE;
    new := CreateRecord();
    new->next := y;    new->rank := newrank;
    return Compare&Swap(A[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 views of the data. The main tool to arbitrate

between threads is the *compare&swap* primitive. If one thread's update is disrupted 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** $x$;

We can also give a version of Find that uses path compression. The difficulty in developing a concurrent Find with path compression is that the path between $x$ and the root of its tree might be altered between the pass that identifies the root, and the pass that makes all pointers along the path point to the root. It is possible to create cycles in the graph if two processors simultaneously perform compressing Finds. The key to getting Find to work correctly is to terminate the update step as soon as a vertex is reached that could have been an ancestor of the root that was originally identified. If $y$ is the vertex identified as the new root, then we traverse the path from $x$ until we encounter a vertex $t$ that has greater rank than $y$, or has the same rank and is greater than or equal to $y$.

*CompressFind*($x$)
  $y := x$;
  **while** $x \neq A[x]\rightarrow next$ **do**
    $x := A[x]\rightarrow next$;
  **while** $y \prec x$ **do**
    $t := A[y]\rightarrow next$;
    $Compare\&Swap(A[y]\rightarrow next, t, x)$;
    $y := A[t]\rightarrow next$;
  **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 its result can be computed from a pair of Finds. However, in the concurrent case, it is important to provide SameSet 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 subset. 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]→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. 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.

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

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.

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

**Proof:** We say that the pointers are *consistent* if they all agree with the ordering on the records, i. e., if $A[x]\rightarrow next = y$, then $y \succeq x$. If the pointers are consistent, then the record structure is a forest. Initially, each record points to itself, so the pointers are consistent. We show that every update to the *rank* and *next* fields preserves consistency.

There are only three places where the rank and next fields are changed: when a path is shortcut during a Find, when two trees are linked by a Union, and when the rank is updated in the Union. We begin by making the following observation about ranks: ranks are never decreased, and the only time a rank is changed is when the record is the root of a tree. We first consider the case of a Find. Suppose that the assignment $A[x]\rightarrow next := t$ is made in a Find, and when the instruction is initiated, the pointers are consistent. In either version of Find, we have that $x \prec t$. Since $x$ is not a root, $A[x]\rightarrow rank$ remains unchanged, so when the assignment succeeds, we still must have $x \prec t$. The second case where a *next* field is changed is when $x$ is made to point to $y$ in a Union. Suppose that the state is consistent when the assignment $yr := A[y]\rightarrow rank$ is made. We must verify that when the assignment $A[x]\rightarrow next := y$ is made, that $x \prec y$. If the assignment succeeds, then we must have $A[x]\rightarrow rank = xr$. At this point, we either $xr < yr$ or $xr = yr$ and $x < y$. Since $yr \leq A[y]\rightarrow rank$, we must have $x \prec y$. To complete our proof, we show that when the *rank* field is updated during a Union, consistency is preserved. Suppose that $A[z]\rightarrow next = y$ and an attempt is made to update the rank of $y$. If the state is consistent when the update is started, $z \prec y$, so consistency is maintained. Since the update of the rank only succeeds if $y$ is still a root, we do not introduce a violation in the case where $A[y]\rightarrow next = w$, $y \neq w$. ∎

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 $I$.*

**Proof:** We construct a serialization of the queries that gives the same answers as the interleaving $I$ of the instructions. We do this by specifying where each of the operations "occurs" in this sequence.

Suppose a Union links the record $x$ to the record $y$. This Union *occurs* when the *compare&swap* inside of *UpdateRoot* succeeds. If a Union discovers that the elements are in the same component, then it *occurs* when when the Union procedure returns. A Sameset

operation occurs when $Find(y)$ is executed for the last time. The precise moment that the operation occurs is when the root is identified inside the Find, i. e., when $x \neq A[x]{\rightarrow}next$ becomes false.

Let $Q_1, \ldots, Q_n$ be the data structure operations. We order the data structure operations $S = Q_{\sigma_1}, Q_{\sigma_2}, \ldots, Q_{\sigma_n}$ by the order in which they occur in the sequence of instructions. This order of operations is clearly consistent with the parallel execution of the queries. We must show that we get the same set of answers. Let $A_i$ be the answer returned by query $Q_i$ in the interleaving $I$, and let $A'_i$ be the answer to $Q_i$ in $S$, the serialization of queries. We must show that $A_i = A'_i$.

Let $CC_i$ be the equivalence relation induced by the first $i$ operations in the interleaving, and $CC'_i$ be the equivalence relation induced by the first $i$ operations of the serialization. We claim that $CC_i = CC'_i$. This can be shown by induction, noting that the Sameset operations do not change the relation, and Union merges the same components under both the interleaving and serialization. By definition of the Sameset operation, the answer to a Sameset query under the serialization is consistent with $CC'_i$. We now must just verify that a Sameset query in the interleaving is consistent with $CC_i$. If Sameset returns true, then the test $x = y$ was true, which shows that the two elements are in the same component. If Sameset returns false, then the test $x = A[x]{\rightarrow}next$ was true, meaning that $x$ was still a root. This means that $x$ was also a root when $Find(y)$ completed. Since at that moment $x$ and $y$ were distinct roots, the two elements were in different components when the Sameset operation occurred. ∎

It is not possible to extend this theorem to sequences of operations that include Finds, because it is possible that different elements will end up as the roots of the sets when the operations are interleaved than under any serialization of the operations. It is possible to perform the three Unions: $Union(a, b)$, $Union(b, c)$, and $Union(c, d)$, where $a < b < c < d$ in such a way that a chain $a \rightarrow b \rightarrow c \rightarrow d$ is created with $d$ having rank one, and $a$, $b$, and $c$ having rank zero. Now suppose we perform $Union(e, f)$, giving $f$ a rank of one. If $Union(d, f)$ is performed, and $d < f$, the root of the tree will be $f$, since they had identical ranks. However, by inserting a set of Sameset queries, it is possible to force a valid serialization of the first three Unions to result in $d$ being to root and having rank two. This means $Union(d, f)$ will also be the root, so $Find(a)$ will give a different answer.

This negative result only applies if we take as our definition of Union that it names the resulting set by a fixed rule. In our case, the naming is given by the ranks of the two roots that are joined. If instead, the type of Union is not specified, then the theorem can be extended to include Finds as well. To extend it, we just need to make sure that the Union used in the serialization joins the elements in exactly the same manner as does the Union in the interleaving.

## 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

| Thread 1 | Thread 2 |
|---|---|
| $Identify(x_1, x_2)$ | |
| $Link(x_1, x_2)$ | |
| | $Identify(x_2, x_3)$ |
| | $Link(x_2, x_3)$ |
| $Rank(x_1, x_2)$ | |
| $Identify(x_3, x_4)$ | |
| $Link(x_3, x_4)$ | |
| | $Rank(x_2, x_3)$ |
| $Identify(x_4, x_5)$ | |
| $Link(x_4, x_5)$ | |

Figure 1: Constructing a chain from the back

identical rank. We call a path of nodes in a tree with equal ranks an *equal ranked chain.* Equal ranked chains do not arise in the sequential case because the rank of the root is incremented when a node is linked to another with the same rank. However, we shall see that it is possible to construct equal ranked chains with an interleaving of Unions. It turns out that chains of unbounded length can be constructed even with only two threads. The Union operations can be divided into three operations: identifying the link, performing the link, and updating the rank. We denote these three operations as $Identify(x, y)$, $Link(x, y)$, and $Rank(x, y)$. when $Union(x, y)$ is performed. We now show how the long chains can be constructed.

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

**Proof:** Suppose that we wish to perform the Unions: $Union(x_1, x_2)$, $Union(x_2, x_3)$, ..., $Union(x_{k-1}, x_k)$. Let the ranks of $x_1, \ldots, x_k$ all be zero, and let $x_1 < x_2 < \cdots x_k$. Figures 1 and 2 show two interleavings of length $k$, with all records except for $x_k$ having rank zero. Both of the constructions work on the principle of having the links of the two threads leapfrog over each other.

The first construction builds the chain from the back. Thread one makes a link from $x_i$ to $x_{i+1}$. Before thread one sets the rank of $x_{i+1}$, thread two makes a link from $x_{i+1}$ to $x_{i+2}$. Since the two records still appear to have the same rank, the link is allowed to succeed. After the link is complete, thread one attempts to update the rank of $x_{i+1}$ and fails. Thread one then goes on to make the link from $x_{i+2}$ to $x_{i+3}$. This careful interleaving results in a chain of length $k$. The other construction works along a similar principle, but builds the chain from the front. ∎

| Thread 1 | Thread 2 |
|---|---|
| $Identify(x_{k-1}, x_k)$ | |
| | $Identify(x_{k-2}, x_{k-1})$ |
| $Link(x_{k-1}, x_k)$ | |
| $Rank(x_{k-1}, x_k)$ | |
| $Identify(x_{k-3}, x_{k-2})$ | |
| | $Link(x_{k-2}, x_{k-1})$ |
| | $Rank(x_{k-2}, x_{k-1})$ |
| | $Identify(x_{k-4}, x_{k-3})$ |

Figure 2: Constructing a chain from the front

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 ≠ A[y]→next do
      t := A[y]→next;
      Compare&Swap(A[y]→next, t, A[t]→next);
      y := A[t]→next;
    UpdateRoot(y, A[x]→rank, y, A[x]→rank + 1);
```

We now show that by adding SetRoot to the Union, we reduce the length of an equal ranked chain to at most $3p$. (This bound is not known to be tight, we guess that the correct bound is $2p$.)

**Lemma 4.4** *The Union–Find routines that incorporate SetRoot do not create equal ranked chains of length greater than* $3p$.

**Proof:** Long equal ranked chains arise because a single thread can add many links to a chain. The SetRoot call restricts a single thread to adding links to an equal ranked chain from the front to the back, meaning that if a thread links $w$ to $x$ and later links $y$ to $z$, where $w$, $x$, $y$ and $z$ are all in the same chain with the same rank, $z$ must be an ancestor of $w$. The reason for this is that when $w$ is linked to $x$, the SetRoot call ensures that the root for $w$ has greater rank than $w$.

An equal ranked chain is *terminated* if it ends with a link to a higher ranked node (as opposed to ending with an incomplete Union). Any equal ranked chain of length more than $p$ is terminated. We now show that a terminated equal ranked chain cannot have an element of distance more than $3p$ from the end of the chain. Before $x$ can gain $p$ descendents of equal rank, the distance from $x$ to the end of the chain must be cut in half. Let $x$ be of distance $2p$ from the end of the chain. Before $x$ gains $p$ descendents, $x$ has its distance reduced to at most $p$ to the end of the chain. This means that no node in an equal ranked chain can have distance of more than $3p$ to the end of the chain. ∎

Equal ranked 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 $\Theta(p)$ 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 additional work to be performed, but we shall see that it is not as significant a problem as simultaneous traversal of long chains. 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 bound the number of times that *UpdateRoot* can fail. We can bill each next failure to the cost of traversing the new link that was added. We now give a bound on the number of rank failures.

**Theorem 4.5** *In a collection of $n$ Unions, there are $\Theta(n \log p)$ 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. We use a simple accounting argument to bound the number of rank failures. If a rank failure occurs when $x$ has rank less than $\log p$, we bill the failure to the Union. If the rank of $x$ is at least $\log p$, we bill the failure to the thread that updated the rank of $x$. Since each update in rank increases the rank, a Union is billed at most $2 \log p$ (since both of the elements involved in the Union may have their ranks changed). Each change in rank can cause at most $p - 1$ rank failures, since the worst case is if all other threads fail on the same Union. The total number of rank changes on nodes with rank at least $\log p$ is at most $n/p$. This means that the total amount billed to the threads is at most $n$. Thus, the number of rank failures is $O(n \log p)$.

We can construct a set of Unions that causes $\Omega(n \log p)$ rank failures. We divide the elements into $n/2p$ groups of $2p$ elements each. In each group, the threads will conspire to cause $(p - 1) \log p$ rank failures. All but one of the threads attempt Union$(x, y)$ where $x$ initially has rank 0 and $y$ has rank at least $\log p$. The other thread performs Unions on all of the other elements in the group and on $x$ so that every time the threads attempt to make a link, they find the rank of $x$ has changed. The number of ranks performed in a group is $p + \log p$ and the number of rank failures is $(p - 1) \log p$. By summing the failures over all of the groups we get the lowerbound. ∎

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 $\Theta(pn+n\alpha(n))$.*

**Proof:** We begin with the lowerbound. The $n\alpha(n)$ term comes from the sequential lower bound for ranked Union with path-halving [TvL84]. The other term arises from having groups of threads simultaneously traverse long chains. The adversary gives a set of $n$ operations, and an interleaving of instructions that takes $\Omega(pn)$ work. $p/2$ threads perform Unions, and $p/2$ threads perform Finds. The adversary divides the queries into $n/p$ phases, where a phase consists of partially completing a set of Unions, followed by performing a set of Finds, and then finishing off the Unions. In the $j$-th phase, Unions are performed to construct a chain

$$x_1^j \rightarrow x_2^j \rightarrow \cdots \rightarrow x_{p/2}^j \rightarrow x_{p/2+1}^j.$$

The Unions are halted when their links are complete, before any path compression is done. The Find processors then all execute $Find(x_1^j)$. The Finds are done in lock step, so that one thread does not gain from another's path compression. The work for a phase is $\Omega(p^2)$, yielding $\Omega(np)$ work overall.

The $pn$ term in the upperbound follows from considering both the rank failures, and the amount of work in traverse chains of equal rank. The $n\alpha(n)$ term follows from the sequential upperbound. (The results rely on a sequential bound that applies when the number of Finds is less than the number of Unions [TvL84].) ∎

The worst case bound of $\Theta(pn)$ work is not very good when 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

In our model, we consider a very powerful adversary. The adversary specifies the queries and decides upon the interleaving of instructions. If an algorithm uses random numbers, then the adversary may base its decisions on the random numbers. (This type of an adversary has been called an adaptive adversary, in contrast to an oblivious adversary which is required to makes its decisions before seeing the random numbers.) We present a pair of results to indicate the power that the adversary has. These results show why some natural approaches for improving the worst case bounds of our algorithms do not work.

The main source of the worst case performance of our algorithm is when a set of threads 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 $x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_p$. 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 from 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. (There are some implementation difficulties, such as dealing with cycles, but we show that this method yields poor results) This strategy looks attractive since it appears that long chains will be very rare. For example, if the $p$ Unions: $Union(x_1, x_2)$, $Union(x_2, x_3) \ldots Union(x_p, x_{p+1})$, are executed, the expected length of the longest segment is only $\log p$. We show that the adversary can defeat this strategy and can cause long chains to be created with high probability.

**Theorem 4.9** *There exists an adversary that can give a set of $p$ Union requests that causes the random Union algorithm to construct a chain of length $\frac{p}{2 \log p}$ with probability at least $1 - \frac{1}{2p}$.*

**Proof:** The difficulty that an adversary has against the random Union algorithm is that the direction of the links is not determined. To solve this problem, the adversary causes a duplicate set of links to be performed, and then choose the link that is made in the desired direction. For $i = 1, \ldots, \frac{p}{2 \log p}$, the adversary makes $2 \log p$ requests of $Union(x_i, x_{i+1})$. The

random Union algorithm chooses a link from $x_i$ to $x_{i+1}$ with probability $\frac{1}{2}$ and a link from $x_{i+1}$ to $x_i$ with probability $\frac{1}{2}$. This means that of the $2 \log p$ Unions of $x_i$ and $x_{i+1}$ at least one of them results in a link from $x_i$ to $x_{i+1}$ with probability at least $1 - \frac{1}{p^2}$. The adversary has all threads execute their Unions until they decide upon the directions for their links. The adversary then chooses one thread from each of the groups of Unions. The threads are chosen so that $x_i$ is linked to $x_{i+1}$. With probability at least $1 - \frac{1}{2p}$ such links exist for $1 = 1, \ldots, \frac{p}{2 \log p}$. If all links exist, then the adversary constructs a chain of length $\frac{p}{2 \log p}$. ∎

The results show the power that the adversary has in defeating various algorithms. One can achieve far better bounds than ours for the Union-Find problem is the adversary is weakened, or if the performance measures are changed. 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 a practical approach on a real machine. Our other approach requires having 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 coarse 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 applying the Cole-Vishkin algorithm

for finding an independent set in a linked list[CV86]. This causes us to spend $O(\log^* p)$ extra time for each linking step. This gives us the following lemma:

**Lemma 5.10** *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 a fixed value into every cell of an array of size $p$. When a thread writes a value into location $j$ it executes the instruction associated with processor $j$ in the current round. When values have been written into all locations the round is complete, and the next round can be started.

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 for two threads to 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 of the computation. This is referred to as making the computation *idempotent*. A second problem 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 the round and started a new round. We can solve this problem by associating a tag with each memory cell indicating 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 5.11** *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. There are a number of problems with this approach that lead to overheads that 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 larger problems come 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 modeled. 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 our solution to the write-all problem. Since the write-all problem is central to the simulation algorithms, it has been considered previously in different models. Under the fail-stop model, Kanellakis and Shvartsman [KS89] gave an $O(p \log^2 p)$ work upperbound for an $p$-processor, $p$-cell write-all algorithm. Martel et al. gave an $O(p)$ work bound for an $\frac{p}{\log p \log^* p}$-processor, $p$-cell write-all algorithm. Martel's algorithm is randomized and assumes that the adversary must set the instruction interleaving prior to viewing the random numbers. Our result differs from Martel's, in that our algorithm is applicable against a much stronger adversary. The write-all problem for the general (adaptive) adversary model has been previously considered by Buss and Ragde[BR90]. They give an algorithm with $O(p^{\log_2 3})$ work ($\log_2 3 \approx 1.79$). Our algorithm includes their result as a special case. They also show an $\Omega(p \log p)$ work lowerbound for the write-all problem. Our main result is that for any $\epsilon > 0$ there exists an $O(p^{1+\epsilon})$ work, $p$-processor, $p$-cell writeall algorithm.

### 5.2.1 The contention of a set of permutations

The difficulty in performing a write-all operation is that since the rates of the threads are variable, it is not possible to know in advance which thread will succeed in writing to a particular location. Each thread must have the ability to perform all of the work since it is possible for all of the other threads to be delayed indefinitely. We begin our discussion of algorithms for the write-all problem by considering *oblivious* algorithms. An algorithm is oblivious if the order that it attempts to write to the cells is independent of the operations of the other processors. (Oblivious algorithms will be used in our solution, but do not by themselves provide a good solution since a $p$ processor oblivious write-all algorithm for $p$ memory cells takes $\Theta(p^2)$ work.) An oblivious algorithm can be viewed as a set of $p$ permutations on $\{1, \ldots, p\}$, where the $i$-th permutations gives the order that processor $i$ writes to the memory cells. In our application of oblivious algorithms, we will be concerned with the number of times values are first written to cells, where if several processors simultaneously perform the first write to a cell we count each of these writes separately. We define this to be the contention of an algorithm.

We formally define contention by fixing the order that the cells are written to, and then computing the maximum number of concurrent first writes that are consistent with this order. For permutations $\pi$ and $\alpha$, the contention of $\pi$ with respect to $\alpha$ (denoted $\mathrm{Cont}(\pi, \alpha)$) is calculated as follows: Suppose the elements $\{1, \ldots, p\}$ are in a list in the order $\pi$ and these elements disappear one-by-one in the order $\alpha$. $\mathrm{Cont}(\pi, \alpha)$ is the number of times that the element at the front of the list is removed. For example, if $\pi = (4, 3, 1, 5, 2, 6)$ and $\alpha =$

$$
\begin{array}{cccccc}
4 & 3 & 1 & 5 & 2 & 6 \\
 & \underline{4} & 1 & 5 & 2 & 6 \\
 & & 1 & 5 & 2 & 6 \\
 & & & \underline{1} & 5 & 2 \\
 & & & & 5 & 2 \\
 & & & & & \underline{5}
\end{array}
$$

Figure 3: Computing $\mathrm{Cont}((4,3,1,5,2,6),(3,4,6,1,2,5))$.

$(3,4,6,1,2,5)$, then $\mathrm{Cont}(\pi,\alpha)$ is 3 since 4, 1, and 5 are at the front of the list when they are removed. Figure 3 shows the computation of contention in more detail, with the underlined elements counting towards the contention. For a set $S$ of permutations and a permutation $\alpha$, the contention of $S$ with respect to $\alpha$ is defined $\mathrm{Cont}(S,\alpha) = \sum_{\pi\in S}\mathrm{Cont}(\pi,\alpha)$. The maximum contention of $S$ is $\mathrm{Cont}(S) = \max_\alpha \mathrm{Cont}(S,\alpha)$.

If $\alpha$ is the identity permutation, then $\mathrm{Cont}(\pi,\alpha)$ is just the number of *left-to-right maxima* in $\pi$. This connection can be generalized to an arbitrary permutation $\alpha$.

**Lemma 5.12** *For permutations $\pi$ and $\alpha$, $\mathrm{Cont}(\pi,\alpha)$ is equal to the number of left-to-right maxima in $\alpha^{-1}\pi$.*

**Proof:** The $i$-th element of the permutation $\alpha^{-1}\pi$ is $(\alpha^{-1}\pi)_i = \alpha^{-1}_{\pi_i}$. Thus, $(\alpha^{-1}\pi)_i$ is the position of $\pi_i$ in $\alpha$. Our definition of $\mathrm{Cont}(\pi,\alpha)$ is that $\pi_i$ counts towards the sum if it is at the front of the list when it is removed. $\pi_i$ is at the front of the list if $\pi_i$ comes after $\pi_1,\ldots,\pi_{i-1}$ in $\alpha$. This occurs when $(\alpha^{-1}\pi)_i$ is a left-to-right maximum in $\alpha^{-1}\pi$. ∎

With $\pi = (4,3,1,5,2,6)$ and $\alpha = (3,4,6,1,2,5)$, $\alpha^{-1}\pi = (2,1,4,6,5,3)$. The left-to-right maxima are 2, 4, and 6, giving a value of 3 for $\mathrm{Cont}(\pi,\alpha)$. Relating the contention to the number of left-to-right maxima makes it quite easy to compute the expected contention for a single permutation or a set of permutations. However, we will need to have bounds on the probability that the contention differs substantially from the expected value which requires a little more work. We begin with a simple lemma giving the expected contention of a uniformly random permutation $\pi$ with respect to a fixed ordering $\alpha$. (The same result holds if we fix $\pi$ and choose $\alpha$ at random.)

**Lemma 5.13** *Let $\alpha$ be a fixed ordering, and suppose that $\pi$ is a random permutation over $\{1,\ldots,p\}$. The expected value of $\mathrm{Cont}(\pi,\alpha)$ is $H_p = \ln p + O(1)$.*

**Proof:** Since $\pi$ is a random permutation, $\alpha^{-1}\pi$ is also a random permutation, so the problem is reduced to computing the expected number of left-to-right maxima in a random permutation. The value $i$ is a left-to-right maximum if $i$ appears before $i+1,\ldots,p$.

This occurs with probability $\frac{1}{p-i+1}$. The expected number of left-to-right maxima is thus $\sum_{i=1}^{p} \frac{1}{p-i+1} = \sum_{i=1}^{p} \frac{1}{i} = H_p$. ∎

We now extend this result to the case where we select $p$ permutations at random. In this case we need a bound on the probability that the contention is more than a constant multiple of the expected value. The number of left-to-right maxima in a permutation on $\{1, \ldots, p\}$ is a random variable with mean $H_p$ and standard deviation approximately $\sqrt{H_p}$ [Knu68]. Since the variables are independent, a natural proof would be to apply the central limit theorem. This approach runs into difficulties because the random variables change for each $p$. In order to apply the central limit theorem rigorously, it is necessary to bound the rate of convergence to normal distribution. We get a simpler proof by viewing the random variables as a sum of independent Bernoulli trials and applying a bound established by Raghavan [Rag86]. For convenience, we restate his result.

**Lemma 5.14** *(Raghavan) Let $X_1, X_2, \ldots, X_r$ be a sequence of independent Bernoulli trials and let $\Psi = X_1 + X_2 \cdots + X_r$. Suppose* $\mathrm{Exp}[\Psi] = m$. *For $\delta > 0$,*

$$\mathrm{Prob}[\Psi > (1+\delta)m] < \left( \frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right)^{m}.$$

**Proof:** See [Rag86], Theorem 1. ∎

**Lemma 5.15** *There is a constant $c$ such that if $\alpha$ is any permutation and $S$ is a random set of $p$ permutations over $\{1, \ldots, p\}$ then* $\mathrm{Prob}[\mathrm{Cont}(S, \alpha) \geq cp \log p] < \frac{1}{p!}$.

**Proof:** Let $\alpha$ be a permutation and $S$ a random set of $p$ permutations. Let $S' = \{\alpha^{-1}\pi \mid \pi \in S\}$ and let $\Psi$ the random variable for the total number left-to-right maxima of the permutations in $S'$. By Lemma 5.12 the distribution of $\mathrm{Cont}(S, \alpha)$ is exactly the same as the distribution of $\Psi$.

Let $X_{ij}$ be a random variable that is one if $j$ is a left-to-right maximum in the $i$-th permutation of $S'$. $\Psi = \sum_i \sum_j X_{ij}$. The key observation for our proof is that this is a set of independent random variables. It is clear that random variables that differ on the first subscript are independent, since they arise for different permutations chosen independently. We must argue that $X_{i1}, X_{i2}, \ldots, X_{ip}$ are independent. In a permutation, $j$ is a left-to-right maximum if $j$ comes before all of the elements greater than $j$. The order of the elements greater than $j$ does not matter. The position of the elements less than $j$ do not influence whether or not $j$ is a left-to-right maximum. This means the probability that $j$ is a left-to-right maximum is independent of the other elements being left-to-right maxima, so the corresponding random variables are also independent.

The probability that $X_{ij}$ is one is $\frac{1}{p-j+1}$, so $\mathrm{Exp}[\Psi] = pH_p < p \ln p + p$. Applying Lemma 5.14 with $\delta = 2$:

$$\mathrm{Prob}[\Psi > 3p \ln p + 3p] < \left( \frac{e^2}{3^3} \right)^{p \ln p} < \left( \frac{1}{e} \right)^{p \ln p} = \frac{1}{p^p} < \frac{1}{p!}.$$

It follows that we can choose $c = 6$ to complete our proof. ∎

The result we are after is to show that there is a set of permutations with low contention with respect to all orderings $\alpha$.

**Lemma 5.16** *There is a constant $c$ such that for each $p$ there is a set $S$ of $p$ permutations over $\{1, \ldots, p\}$ with* $\mathrm{Cont}(S) < cp \log p$.

**Proof:** Let $c$ be the constant from Lemma 5.15. An ordering $\alpha$ is bad for a set $S$ of permutations if $\mathrm{Cont}(S, \alpha) \geq cp \log p$. From Lemma 5.15, it follows directly that for a random set $S$ of permutations, the expected number of bad orderings is less than one. This means that there exists a set $S$ with no bad orderings, in other words, that $Cont(S) < cp \log p$. ∎

### 5.2.2 Randomized algorithm for the write-all problem

The basic idea behind our write-all algorithms is to have threads execute operations in an order given by a set of permutations. In order to make the algorithm more efficient each operation consists of a block of writes. Each block has an additional memory location indicating whether or not it has been completed. We refer to these cells as the *completion bits*. When a thread is to write to a block, it looks at the completion bit for the block, and if it has not been set, it writes to each memory location in the block and then to the location indicating the block is complete. The cost of writing to a block is one if the completion bit was set at the start of the write, and is equal to the size of the block otherwise.

If two threads work on the same block at the same time, then they both perform all the writes in the block. The advantage of this method is that if one thread considers blocks $B_1$ and $B_2$ in order $B_1$, $B_2$ and another thread considers the blocks in order $B_2$, $B_1$, then at least one of the blocks is written to by only one thread. By processing tasks in a different order the threads reduce the total amount of work. The following lemma relates our notion of contention to the amount of work that is performed. A block access is said to occur if thread $t$ writes to block $B$.

**Lemma 5.17** *Let $S = \{\pi^1, \ldots, \pi^p\}$ be a set of permutations on $\{1, \ldots, p\}$. Suppose threads $t_1, \ldots, t_p$ write to blocks $B_1, \ldots, B_p$. Thread $t_i$ processes the blocks in the order given by $\pi^i$. The total number of block accesses is at most* $\mathrm{Cont}(S)$. *If each block has size $b$, then the total number of writes by the $p$ threads is at most $(b+1)\mathrm{Cont}(S)$ and the total work is $O(b\mathrm{Cont}(S) + p^2)$.*

**Proof:** Consider a run of the algorithm. We can construct an order in which we view the blocks to have been processed by taking the order of the successful writes to the completion bits. Since the algorithm is an interleaving of atomic instructions, this gives us a permutation $\alpha$.

Suppose that thread $t_k$ succeeds in writing to the block $B_i$, so $t_k$ accesses $B_i$. We show that this access can be counted as a contribution of $t_k$ to $\mathrm{Cont}(S)$. Suppose $t_k$ attempts to

write to $B_j$ before the write to $B_i$. Clearly this means that $B_j$ was completed before $B_i$, so $j$ comes before $i$ in $\alpha$. Thus, all elements coming before $i$ in $\pi_k$ are completed before $i$, so $i$ contributes to $\mathrm{Cont}(S)$ from $t_k$.

If $j$ precedes $i$ in $\pi^k$, then $j$ must also precede $i$ in $\alpha$. This means that $i$ contributes to $\mathrm{Cont}(S, \alpha)$ from the permutation $\pi^k$. Thus, $\mathrm{Cont}(S)$ is an upperbound for all interleavings of the number of block accesses. The maximum number of writes to the cells of the blocks is $(b+1)\mathrm{Cont}(S)$, including the write to the completion bit. The $p^2$ term in the work is included to account for all of the reads of the completion bits, including the case when the block is already complete. ∎

We can apply the lemma to get a simple randomized algorithm for the write-all problem. The first algorithm is for the case where the size of the array is greater than the number of threads being used. This is applicable if we are simulating a larger synchronous machine with a smaller machine.

**Lemma 5.18** *There exists a randomized $p$-thread algorithm for a write-all to $p^2$ memory cells that takes $O(p^2 \log p)$ work with high probability.*

**Proof:** We divide the memory cells into $p$ blocks of size $p$. We choose a random set $S$ of $p$ permutations over $\{1, \ldots, p\}$. With high probability, $\mathrm{Cont}(S) \leq cp \log p$. Thread $i$ writes to the blocks in the order given by the $i$-th permutation of $S$. A direct application of Lemma 5.17 gives the work bound. ∎

We can improve the result in the sense that the number of cells written to is reduced, at the cost of a slight increase in the amount of work per cell. The idea is to use a two level recursive scheme, where we subdivide each block into a set of smaller blocks.

**Lemma 5.19** *There exists a randomized $p$-thread algorithm for a write-all to $p^{3/2}$ memory cells that takes $O(p^{3/2} \log^2 p)$ work with high probability.*

**Proof:** Suppose the threads are $\{t_{ij} \mid 1 \leq i, j \leq \sqrt{p}\}$. We divide the memory cells into $\sqrt{p}$ blocks $B_1, \ldots, B_{\sqrt{p}}$, each of size $p$, and then subdivide each block $B_i$ into $\sqrt{p}$ subblocks $B_{i1}, \ldots, B_{i\sqrt{p}}$ of size $\sqrt{p}$. We choose a random set $S = \{\pi^1, \ldots, \pi^{\sqrt{p}}\}$ of $\sqrt{p}$ permutations over $[\sqrt{p}]$. For some fixed $c$, $\mathrm{Cont}(S) < c\sqrt{p} \log p$ with high probability. Thread $t_{ij}$ processes the blocks in the order given by $\pi^i$. If the block $B_k$ does not have its completion bit set when $t_{ij}$ processes it, then $t_{ij}$ processes $B_{k1}, \ldots, B_{k\sqrt{p}}$ in the order $\pi^j$.

We now establish the work bound. Fix an interleaving for the instructions. We say that $i$ writes to block $B_k$, if for some $j$, $t_{ij}$ writes successfully to block $B_k$. An application of Lemma 5.17 says that the total number of times that $i$ writes to $B_k$, summed over $i$ and $k$ is bounded by $\mathrm{Cont}(S)$. If $i$ writes to $B_k$, we make the pessimistic assumption that all $t_{i1}, \ldots, t_{i\sqrt{p}}$ succeed in writing to $B_k$. Applying the proof of Lemma 5.2.2, the amount of work done by $t_{i1}, \ldots, t_{i\sqrt{p}}$ is $O(\sqrt{p}\mathrm{Cont}(S) + p)$. Putting these two bounds together, we get a work bound of $O(\sqrt{p}(\mathrm{Cont}(S))^2 + p^{3/2})$ which is $O(p^{3/2} \log^2 p)$ with high probability. ∎

### 5.2.3 Deterministic algorithm for the write-all problem

We can adapt the ideas used in the randomized algorithm to a deterministic algorithm. The only use of randomness is that we do not know how to construct a set of permutations with low contention, so we rely on using a random set, which is good with overwhelming probability. In our deterministic algorithm we increase the number of levels of recursion. As the number of levels of recursion increases, the size of the permutations used decreases. When we reduce the size of the permutations used to a constant, we can claim that we can find a good set in constant time by a brute force search.

We show that for any $\epsilon > 0$, there exists a deterministic $p$-thread algorithm for writing to $p$ memory cells that takes $O(p^{1+\epsilon})$ work. Suppose $p = q^d$ for integers $q$ and $d$. We view the computation taking place on a $q$-ary tree of height $d$. Each internal node of the tree contains a single bit, which corresponds to the completion bit of a block, and the leaf nodes contain the memory cells. We begin with a set $S = \{\pi^0, \ldots, \pi^{q-1}\}$ of permutations over $[q]$ with $\mathrm{Cont}(S) \leq cq \log q$. The existence of such a set is guaranteed by Lemma 5.16. Since $q$ will turn out to be a constant, we do not need to worry about the complexity of finding the set $S$. When a thread is at a node, it visits the children of that node in an order given by a permutation of $S$. Each thread chooses a permutation to use for all of the nodes at a particular level of the tree. The choice of permutation is based upon looking at the digits of the $q$-ary expansion of the number of the thread. Suppose that $i = q_1 q_2 \ldots q_d$ when represented in $q$-ary. Thread $t_i$ considers the children of a node on level $j$ in the order given by $\pi^{q_j}$.

We give the code for a routine *Traverse* which does a write-all to array $A$. To increase clarity, we describe the routine using a tree, although the implementation could be done more efficiently with arrays. We use a C-like notation to describe the tree. For the node $n$, $n \rightarrow c$ is a vector of pointers to $n$'s children and $n \rightarrow c[i]$ is the pointer to the $i$-th child of $n$. The completion bit of an internal node $n$ is $n \rightarrow bit$, and the array index associated with the leaf node $n$ is $n \rightarrow index$.

```
Traverse(l, node, q_1 ... q_d)
  if l = d then
    A[node->index] := 1
  else
    if node->bit = 0
      for i = 1 to q do
        Traverse(l + 1, node->c[pi_i^{q_l}], q_1 ... q_d);
      node->bit := 1;
```

Each thread begins with an initial call $\mathit{Traverse}(1, \mathit{root}, q_1 \cdots q_d)$, where *root* is the root of the tree and $q_1 \cdots q_d$ is the $q$-ary expansion of the thread number.

Since each thread will visit every node of the tree in the absence of other threads, the algorithm performs a write-all to the array $A$. We must show that it completes within the desired work bounds. The key to a formal proof of the work bound is that we can characterize the worst case behavior. In order to maximize the work of the processors, an adversary will

keep the threads together in groups, so that the threads of each group work in lock step. For example, at the top level of the tree, the adversary will divide the threads into $q$ groups of $p/q$ threads each. The adversary will choose an ordering on the groups to maximize the contention of $q$ threads working on a block of $q$ cells. We will formalize these ideas in our proof.

The threads are divided into groups based upon the $q$-ary expansions of their indices. If threads $t_1$ and $t_2$ agree on the first $k$ digits of their $q$-ary expansions, then we say they are in the same $k$-group. A group of threads is in lock step on level $j$ if they access the same set of nodes on level $j$, and for each node on level $j$ accessed, the entire group simultaneously writes to the completion bit. We say that the threads are *synchronized by groups* if for each $k$, all of the $k$-groups are in lock step on level $k$.

**Lemma 5.20** *There exists a maximum work interleaving such that the threads are synchronized by groups.*

**Proof:** The proof is a straightforward reordering argument. We take a maximum work interleaving, and reorder the instructions to make it synchronized by groups without decreasing the work performed.

Suppose that threads $t_1$ and $t_2$ are in the same $k$ group, and $t_1$ writes to the completion bit of block $B$ before $t_2$ does. $t_2$ processes blocks in levels 1 through $k$ in exactly the same order that $t_1$ does. Since $t_1$ has completed block $B$, $t_2$ will not encounter any incomplete cell until it reaches the completion bit for $B$. This means that the instructions of $t_2$ (up to writing to $B$) can be executed without interfering with other threads. Since thread $t_2$ will only be accessing completed nodes, executing all of these instructions immediately will not decrease the work performed. Thus, $t_2$ can execute all instructions upto the write to $B$, and do a simultaneous write to $B$'s completion bit with $t_1$. This reordering argument can be repeated until all groups are working in lock step. ∎

**Lemma 5.21** *The sum over all $k$-groups of the number of blocks accessed at level $k+1$ is at most $(\mathrm{Cont}(S))^k$.*

**Proof:** The proof is by induction on $k$. The base case holds, since the entire set of threads is a single 0-block accessing the block at level one. Suppose there are at most $(\mathrm{Cont}(S))^{k-1}$ block accesses by $k-1$ groups on blocks at level $k$. By Lemma 5.20 we assume that the threads are synchronized by groups. Consider a $k-1$ group that accesses the block $B$ at level $k$. This $k-1$-group consists of $q$ $k$-groups. These groups consider the blocks at level $k+1$ descended from $B$ in the order given by the permutations in $S$. Lemma 5.17 gives a bound of $\mathrm{Cont}(S)$ for the number of accesses of the $k$ groups to the blocks at level $k+1$ descended from $B$. This means that the total number of accesses by $k$-groups to blocks at level $k+1$ is at most $(\mathrm{Cont}(S))^k$. ∎

We can now prove the main theorem of this section.

**Theorem 5.22** *For every $\epsilon > 0$, there exists a deterministic algorithm for $p$ threads that simulates $p$ instructions with $O(p^{1+\epsilon})$ work.*

**Proof:** The total amount of work performed is proportional to the total number of blocks accessed. When a $k$-group accesses a block at level $k+1$, a total of $p/q^k$ threads may participate. Using lemmas 5.16 and 5.21 we get the following bound on $W$, the total number of blocks accessed:

$$W \leq \sum_{k=0}^{\log_q p} \frac{p}{q^k}(\mathrm{Cont}(S))^k \leq \sum_{k=0}^{\log_q p} p(c \log q)^k < 2p(c \log q)^{\log_q p}.$$

Manipulating the logarithms we have:

$$(c \log q)^{\log_q p} = 2^{\log(\log q^c)^{\log_q p}} = p^{\log \log q^c / \log q}.$$

Since $\log \log q^c / \log q \to 0$, with the appropriate choice of $q$, we can make $W < p^{1+\epsilon}$. ∎

## 5.3 Coarse grained algorithm

We now consider a coarse grained approach to improving the performance of the algorithm. This method requires that the number of queries that can be performed simultaneously is greater than the number of threads. 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 and has a very low overhead.

We alter our query model to accommodate extra queries, and to implement our randomized strategy[3]. We assume that the algorithm issues queries in batches of $p^2$ queries. When a thread requests a query, it is given a randomly selected query from the set of available queries. When the $p^2$ queries have been answered, a new set of $p^2$ queries is made available. This method forces the adversary to generate $p^2$ queries at once, and allows our randomization to mitigate the worst case behavior.

We assume that a minor modification is made to the Union procedure so that the roots of the trees have their ranks set before a link is attempted. If Union$(x, y)$ is attempted, and $w$ is the root for $x$, with $x \neq w$, and $v$ immediately precedes $w$ on the path from $x$, we shall make sure that the rank of $w$ is greater than the rank of $v$.

The source of inefficiency in our Union-Find algorithm is that it is possible for long chains (chains of length $p$) to arise, and then traversing the chains is expensive. We show that the randomized algorithm has better performance since it is unlikely for long chains to be created. Long chains arise by having threads simultaneously making links on the same path. If no other thread is linking to $u$ or from $v$, then linking from $u$ to $v$ does not result in adjacent nodes in the tree having the same rank. In our adversary model, if the adversary attempts to have many links on the same path, it will turn out that many operations become

[3]A model that might be more natural would be to have threads select randomly from a collection of $p^2$ query registers. We believe that this model gives the same results.

very easy. On the other hand, if the adversary does not try to concentrate the links, long paths will not be constructed.

The basis of the proof is to bound the number of edges in chains of equal rank. Once we have a bound on the number of edges in chains of equal rank, we can separately bound the amount of work traversing these edges, and the amount of work traversing other edges to get our result.

We keep track of the connected components formed by the Unions, and use them to define sets of edges. (We often view Union$(x, y)$ as the undirected edge between $x$ and $y$.) Initially, Union$(x, y)$ is associated with the component $\{x\}$ and the component $\{y\}$. The weight of a component is the number of Unions associated with the component. As Unions are performed, the weights of the components increase. When a component's weight exceeds $p^{1/2}$, the Unions are put in a *heavy set* and no longer associated with a component. Once a heavy set is formed, the edges in the set are not changed.

**Lemma 5.23** *The number of edges in chains from heavy sets is $O(p^{3/2} \log n)$.*

**Proof:** Let $S = \{U_1, \ldots, U_k\}$ be the Unions in a heavy set which was formed from a component $C$. Consider the Unions from $S$ that succeed in linking $C$ to another component, or linking another component to $C$. These are to or from vertices $v_1, v_2, \ldots, v_j$ in order of linking. We claim that these vertices are strictly increasing in rank. The reason for this is that before a link is attempted from $x$, the thread makes sure $x$ has greater rank than the child $x$ was reached from. This means that at most $\log n$ Unions from $S$ succeed. There are at most $2p^{3/2}$ heavy sets (since each Union may appear in two sets). Thus, at most $O(2p^{3/2} \log n)$ edges come from heavy sets.

∎

**Lemma 5.24** *The expected number of edges in equal ranked chains that came directly from components (without being in a heavy set) is $O(p^{3/2} \log n)$.*

**Proof:** We say a component is *active* if a thread is in the process of linking the component to another component. An edge can become part of a chain of equal rank in three ways: if it is adjacent to an edge from a heavy set, if it is adjacent to an active component, or if it is in an active component adjacent to a later query edge. The first case is covered by Lemma 5.23. We now show that the expected number of edges that arise from the other two cases is $O(p^{3/2})$. The number of active components is less than $p$. Each active component is adjacent to at most $p^{1/2}$ queries since additional queries are set aside in heavy sets. Thus, when a query is chosen at random, it has probability at most $p^{1/2}$ of being adjacent to an active component. Thus, the expected number of the $p^2$ queries adjacent to active queries is $O(p^{3/2})$. ∎

**Theorem 5.25** *The expected work for $n$ Union-Find operations is $O(np^{1/2} \log n)$ when the queries are executed in batches of $p^2$ queries and the queries within a batch are executed in a random order.*

**Proof:** Lemmas 5.23 and 5.24 show that while processing one batch of $p^2$ queries, the expected number of edges put into chains of equal rank is $O(p^{3/2} \log n)$. This means that during a total of $n$ queries, the number of edges put in chains of equal rank is $O(np^{-1/2} \log n)$. The amount of work $n$ Finds could expend in traversing these equal ranked chains is $O(np^{1/2} \log n)$. The amount of work exclusive of traversing the chains is $O(n \log n)$ ∎

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 coarse 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.

There is room for improvement of our results on the Union-Find problem. We would like to see an improvement of our worst-case bound of $O(np)$ work that does not rely on simulating a synchronous algorithm. 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.

# 7 Acknowledgements

The authors wish to thank Greg Barnes, Paul Beame, Krishna Palem, Ashok Subramanian and Orli Waarts for helpful comments on this work.

# 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.

[AW91] R. J. Anderson and H. S. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the 23rd ACM Symposium on Theory of Computation*, pages 370–380, 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.

[Knu68] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley, 1968.

[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.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[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.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[LF81] N. A. Lynch and M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[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.

[Nis91] N. Nishimura. *Asynchrony in Shared Memory Parallel Computation.* PhD thesis, Department of Computer Science, University of Toronto, June 1991.

[Plo89] S. A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 159–175, 1989.

[Rag86] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. In *27th Symposium on Foundations of Computer Science*, pages 10–18, 1986.

[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.

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(4):215–225, 1975.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, 1983.

[TvL84] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.