A Wait-Free Sorting Algorithm

Nir Shavit * Eli Upfal † Asaph Zemach ‡

Abstract

Sorting is one of a set of fundamental problems in computer science. In this paper we present the first wait-free algorithm for sorting an input array of size N using $P \leq N$ processors to achieve optimal running time. Known sorting algorithms, when made wait-free through previously established transformation techniques have complexity $O(\log^3 N)$. The randomized algorithm we present here, when run in the CRCW PRAM model executes in optimal $O(\log N)$ time when $P = N$ and $O(N \log N/P)$ otherwise. The wait-free property guarantees that the sort will complete despite any delays or failures incurred by the processors. This is a very desirable property from an operating systems point of view, since it allows oblivious thread scheduling as well as thread creation and deletion, without fear of losing the algorithm's correctness. We further present a variant of the algorithm which is shown to suffer no more than $O(\sqrt{P})$ contention when run synchronously.

1 Introduction

Sorting is a basic algorithmic building block and has attracted the attention of many researchers. In this paper we present a wait-free algorithm for sorting an array of N elements, in the CRCW PRAM model with processor failures and undetectable restarts. Herlihy [17] defines a wait-free data structure as one on which any operation by any processor is guaranteed to complete within a bounded number of steps, regardless of the actions or failures of other processors. By extension, a wait-free algorithm for some fixed-size problem is guaranteed to arrive at the solution within a bounded number of steps, even in the face of processor failures and delays. Wait-free algorithms have the appealing property that correct completion of the algorithm is assured despite any problematic scheduling imposed by the system. Greenwald and Cheriton [16] note that such algorithms are well suited for implementing operating system kernels since they free the operating system from many book-keeping tasks. Consider the case of sorting a large data set in the background of other ongoing computations. Using the wait-free algorithm given here we can begin the sort by spawning a thread for each idle processor in the machine. If during the execution a processor is needed elsewhere we can reap the thread associated with it without fear of leaving the program's internal data structures in an inconsistent state. On the other hand if other processors become free, we can spawn more threads to speed up the sorting process. An interesting special case is when one of the sorting algorithm's own threads must wait for some time-consuming operation such as a page fault. We can immediately spawn a new sorting thread for the same processor and continue working on available elements of the array, soaking up otherwise wasted cycles. When the page fault is handled, we can summarily destroy any such thread. From the point of view of the operating system, wait-free algorithms are desirable since they allow oblivious allocation of processors to threads, creation of new threads, and destruction of redundant threads as needed, leading to better utilization of system resources.

1.1 Related work

The number of articles dealing with sorting in a parallel environment is too large to allow mentioning them all, so we will restrict discussion to those that are directly related to our work. The sorting technique we use is based on Hoare's serial Quicksort [20] of which there have been a number of parallel implementations. For the CRCW PRAM, there is the result of Martel and Gusfield [28], with an $O(\log N)$ running time that may require as much as $O(N^3)$ memory. This is improved upon by Chlebus and Vrto [10] to achieve $O(\log N)$ time and $O(N)$ space, using a method that is very similar to the one we use here. For EREW PRAMs, Zhang and Rao [33] present an algorithm with a running time of $O((\log P + N/P) \log N)$. This was later improved upon by Brown and Xiong [8] to achieve $O((N/P) \log N)$ for the case where $P \leq N/\log N$. All of these algorithms work in the PRAM model, making strong use of processor synchronization, and are not wait-free.

In [17] Herlihy also gave a general method for the construction of wait-free objects [18]. Unfortunately, trying to

*MIT and Tel-Aviv University. Contact Author: shanir@theory.lcs.mit.edu. Work supported by the following grants: ARPA: F19628-95-C-0118, AFOSR-ONR: F49620-94-1-0199, NSF: 9225124-CCR and CCR-9520298.

†The Weizmann Institute, Israel, and IBM Almaden Research Center, California. Work at the Weizmann Institute supported in part by the Norman D. Cohen Professorial Chair of Computer Science, a MINERVA grant, and a grant from the Israeli Academy of Science.

‡Tel-Aviv University. Supported by an Eshkol Scholarship from the Israeli Ministry of Science. Part of this work was done while the author was at MIT.

1997 PODC 97 Santa Barbara CA USA

implement a "sorting-object" using this method (or the improvements of Afek et al. on it [1]) is liable to become inefficient. Processors wishing to update the shared object must first post the changes they are about to make. If they fail before these changes are completed another processor can complete them, ensuring the object remains consistent. This can be detrimental to parallelism as often only one process performs all pending work. For example, using the methods of [1], the complexity of a wait-free operation is $O(kf \log f)$, where k is the number of processors accessing the object concurrently, and f is the complexity of the update operation. Using any straight-forward sorting algorithm, we can expect $k = P$, which will not yield good performance. Similar objections apply to Herlihy and Moss' transactional shared memory [19], since Shavit and Touitou's lock-free software implementation [29] suffers the same drawbacks mentioned above, and proposed hardware implementations are limited in size [5]. Some special purpose wait-free data structures have also been introduced, of which the most suitable for sorting are heaps and priority queues. Both data structures use a scheme for announcing pending operations similar to the one proposed by Herlihy, and tend to perform at least part of each pending operation in a serial manner. For Barnes' [6] wait-free heap the complexity is $O(Mk \log N)$ for performing M operations by k threads on a heap with N elements. Israeli and Rappoport's [21] priority queue besides requiring a non-standard two word Compare&Swap operation also employs a "helping" method which limits concurrency (this is discussed in [29]). In any event, simply providing a wait-free data structure which can order its inputs does not immediately imply a wait-free solution to the sorting problem. One must still allocate processors to values, handle duplicate insertions and deletions of the same value, and make sure values aren't lost even if the processor assigned to them fails.

Another possible approach comes from research into fault tolerant systems. For a fixed sized array, an algorithm which sorts in a failure model which allows processors to fail, and later possibly revive and proceed (in an undetectable manner) would also sort under wait-free assumptions. It is possible to convert any PRAM algorithm to work in this failure model. However such transformations are expensive. One might start with an $O(\log N)$ sorting algorithm [2, 7, 11] and apply a transformation technique which simulates a reliable PRAM on a faulty one. This idea was first introduced by Kanellakis and Shvartsman in [22], and later improved upon by Kedem et al. [23]. Both of these results are for the fail-stop model. In the general asynchronous model the results of Anderson and Woll [3] and Buss et al. [9] apply, and would mean an increase in the complexity of the sort to at least $O(\log^3 N)$, and cost a multiplicative $\log N$ factor in memory. The method of Martel et al. [25] would also work, and would increase running time by only a $\log N$ factor. However, it supports only limited asynchrony through the use of the non-standard FTS instruction.[1] The above simulations would not be efficient, as was noticed by [3], since they require synchronization at the end of every PRAM step.

These results indicate the need to develop a sorting algorithm designed specifically for the wait-free case. A previous result in fault-tolerant sorting is given by Yen et al. [32], which employs the Batcher sorting network, giving a complexity of $O(\log^2 N)$. This result supports only the fail-stop failure model and requires non-standard hardware components. It is possible to transform this algorithm into a wait-free sorting algorithm with a complexity of $O(\log^3 N)$, but it would require an $O(\log^2 N)$ factor memory increase. There has also been much study in fault tolerant sorting networks [4, 24, 30]. This work deals with networks whose comparator-gates may be faulty but whose connections do not fail. This is akin to a computation model where processors do not fail, but may sometimes return the wrong result for a comparison.

Related work has also been done on asynchronous computing models. Cole and Zajicek [12] proposed the APRAM model for designing parallel algorithms to work in an asynchronous setting. Zhou et al. [34] present a sorting algorithm for asynchronous machines that is not wait-free. Neither is the recent sorting algorithm of Gibbons et al. [15] for the QRQW asynchronous PRAM. While these models avoid making any timing assumptions, they also do not allow processor failures, and hence do not produce wait-free algorithms.

1.2 Our algorithm

Our parallel Quicksort algorithm is the first wait-free algorithm for the sorting problem to achieve optimal running time of $O(N \log N/P)$ or $O(\log N)$ in the case where $P = N$. These running times are under the assumption that all processors participate in the algorithm and incur no delays. We are able to achieve these times by not using a standard PRAM sorting algorithm which generally require $O(\log N)$ synchronized steps. As was previously noted, the cost of simulating $O(\log N)$ PRAM steps in a wait-free manner is $O(\log^3 N)$. In contrast, our algorithm consists of three phases, each of which requires logarithmic time. Since wait-freedom is inherently incorporated into the algorithm, the $\log N$ cost of tracking completed work can be made additive (as opposed to multiplicative when using simulation techniques). After presenting the algorithm we turn our attention to the issue of contention and show a simple low contention work allocation scheme. This scheme, when combined with low contention winner selection and approximate write-all (actually, write-most) yields a wait-free sorting algorithm with contention $O(\sqrt{P})$.

2 A Wait Free Sorting Algorithm

One of the challenges of writing wait-free code for manipulating a number of objects is to make sure that all objects are dealt with. Since processors may fail, one cannot assume that just because work has been assigned to a processor – it will indeed complete that job. This situation is modeled by the write-all problem of [22]: given an array B of N elements and P fault-prone processors, devise an algorithm that fills every element of B with "1". A standard solution is to assign work to processors using binary trees.

2.1 Work assignment trees

Work Assignment Trees (WATs) are binary trees that store jobs in the leaves and use the inner nodes to track progress in subtrees rooted at those nodes. Figure 1 illustrates an implementation of WATs. The first routine, `undone_element` finds the next job to be done in the tree, starting the search at `start`. The routine `propagate_done` marks an element as completed, and continues up the tree along the path to the root, marking nodes whose subtrees are done. For simplicity we assume that for the root of the tree the routine `parent()`

[1] The FTS Fetch-Test-Store instruction is a stronger version of Read-Modify-Write which can read one location and, based on the value read, modify a *different* location.

returns a unique value EMPTY for which leaf(EMPTY) = TRUE and tree[EMPTY] <> DONE.

```
function undone_element(tree: WAT(N),
                        start: integer
                       ) : returns integer
begin
  i := start
  repeat
     while tree[i] = DONE do
        i := parent( i )
     end
     while not leaf(i) do
        if tree[ left_child(i) ] <> DONE then
           i := left_child(i)
        else
           i := right_child(i)
     end
  until tree[i] <> DONE
  return i
end

procedure propagate_done(tree: WAT(N),
                         i: integer )
begin
  tree[i] := DONE
  if not root(i) and tree[ sibling(i) ] = DONE
  then
     propagate_done( tree, parent(i) )
  endif
end
```

Figure 1: Work-Assignment-Tree algorithm

Since this method is well established we state the following lemma without proof (see for example [3, 9, 22, 26]).

Lemma 2.1 *Let S be the set of leaves in the WAT for which* done[] $\neq$ DONE, *at the time the routine* undone_element *is called. If S is not empty the routine will return one of the elements of S, otherwise it will return* EMPTY. *In either case, the routine runs in $O(N)$ time.*

```
procedure wait-free-algorithm
processor private variables
  i: integer
shared variables
  work: WAT(N)
begin
  i := undone_element(work , START )
  repeat
    func(i)
    propagate_done(work, i)
    i := undone_element(work, i)
  until i = EMPTY
end
```

Figure 2: A skeleton wait free algorithm

The procedure propagate_done can take no more than $O(\log N)$ iterations since each iteration goes up one level in the binary tree. Given this fact and Lemma 2.1 it is easy to see that the algorithm of Figure 2 is wait-free, provided the function func() is wait-free. If we replace the call to func() with the operation B[i]:=1 for some array B of size N, we get a solution for the write-all problem.

2.2 The sorting algorithm

```
type Element is
  key:    any-type
  parent: integer initialized to 1
  child:  array [BIG,SMALL] of integer
             initialized to EMPTY
  size:   integer initialized to 0
  place:  integer initialized to 0
end

A:  array [1..N] of Element
```

Figure 3: Data structure used for sorting

We now present our wait-free algorithm for sorting an array A of N elements using P processors in detail. The algorithm is divided into three phases: tree building, tree summation and element shuffling. In the first phase we construct a sorted binary tree whose nodes contain the records of A. For this purpose we attach two child pointers and one parent pointer to each record. Initially, all pointers point to the first element of A, this will be the pivot element for the root of the tree. The first phase is shown in Figure 4 and proceeds as follows. First we note the fact that A[1], being the first pivot need not be inserted into the tree (line 7). Initially, we assign the p-th processor to insert the $\lceil Np/P \rceil$-th element (line 8). A processor p which is inserting record i first compares its key to the key of root element, setting side to the result of the comparison. We assume that no two keys are the same, which can easily be accomplished by using an element's index to break ties. Now p tries to establish i as the appropriate child of the root node (line 18). Since the compare_and_swap operation will succeed only if the child is EMPTY, p can re-read the child's value after the operation to check success. By now either p or some other processor has managed to install its records as the child of the root (line 20). If i was installed, (either by p or by some other processor simultaneously working on i), p fixes i's parent pointer (line 21), updates the work tree to mark the fact that i is done, and chooses another element (lines 22-23). If i was not installed, it follows that some other processor, q preceded p in installing its element, j, as the root's child. So p must now try to install i as a child of j. It does so by updating its local parent pointer to j, and going through the loop again. Eventually, p will install i somewhere in the tree, and go on to the next element.

We make the following observations about the procedure build_tree.

1. All processors begin the algorithm with the same value for parent.

2. For a given pair of values of i and parent, the comparison in line 13 always yields the same results.

3. For a given pair of values of parent and side the read operation in line 19 always returns the same value, which is never EMPTY.

4. As a direct consequence of facts 1–3, we get that two processors with the same value for i would follow the same path down the tree. For this reason the same value cannot be successfully inserted twice into the tree. Which also means that, for a given processor and value of i, each iteration of the loop in lines 11–20 is done with a different value of actual.

```
1  procedure build_tree
2  processor private variables
3    i,actual,parent,side: integer
4  shared variables
5    work: WAT(N)
6  begin
7    propagate_done(work,1)
8    i := undone_element(work , N*MY_MID/P + 1)
9    repeat
10     actual := A[i].parent
11     repeat
12       parent := actual
13       if  A[parent].key > A[i].key then
14           side := SMALL
15       else
16           side := BIG
17       endif
18       compare_and_swap(A[parent].child[side],
                          EMPTY, i)
19       actual := A[parent].child[side]
20     until actual = i
21     A[i].parent := parent
22     propagate_done(work, i)
23     i := undone_element(work, i)
24   until i = EMPTY
25 end
```

Figure 4: Phase 1 of the sort: building the Quicksort tree

5. Each time the compare_and_swap in line 18 succeeds, it is with a different value for i. This follows directly from the fact that processors working on the same element follow the same path down the tree.

Lemma 2.2 *The loop in lines 11–20 will be performed no more than $N-1$ times.*

Proof: The proof is by the pigeon-hole principle. At each iteration a processor attempts the compare_and_swap on a different location (fact 4). There are N possible locations, and only N-1 possible different values of i (no processor is assigned i=1). Since no value can be encountered twice (facts 4 and 5), eventually either the compare_and_swap succeeds, or a processor encounters its own value in the tree and exits. ∎

Lemma 2.3 *When the first processor completes the procedure* build_tree, *after at most $O(N^2)$ operations, the tree defined by the* child *pointers will be a sorted binary tree containing all the records of* A.

Proof: A node's child pointers, once set, are never changed. This assures the comparison in line 13 is consistent for all processors. Since key values don't change during the course of the algorithm and all processors start by comparing their key to the same value, the resulting tree is correctly sorted. ∎

The previous two lemmas prove that the first phase of the algorithm is wait-free and builds the pivot tree correctly. Any processor that completes the first phase immediately goes on to the second phase.

In the second phase of the algorithm we calculate the size of the subtree rooted at each element. Since our binary trees are not complete we must count the elements directly. The algorithm follows the standard tree summation method except that it uses randomization to spread the processors around the tree.

```
function tree_sum(i: integer) returns integer
processor private variables
  sum:  integer
begin
  if i = EMPTY then
     return 0
  else if A[i].size > 0 then
     return A[i].size
  else if CoinToss = Heads then
     sum := tree_sum( A[i].child[BIG] )
     sum := sum + tree_sum( A[i].child[SMALL] )
  else
     sum := tree_sum( A[i].child[SMALL] )
     sum := sum + tree_sum( A[i].child[BIG] )
  endif
  A[i].size = sum+1
  return sum+1
endif
```

Figure 5: Phase 2 of the sort: summing the subtrees

```
procedure find_place(i: integer, sub: integer)
processor private variables
  s: integer
begin
  if i = EMPTY or A[i].place > 0 then
     return
  endif
  if A[i].child[SMALL] <> EMPTY then
     s := A[ A[i].child[SMALL] ].size
  else
     s := 0
  endif
  A[i].place := s + sub + 1
  if CoinToss = Heads then
     find_place( A[i].child[SMALL], sub)
     find_place( A[i].child[BIG], sub + s + 1)
  else
     find_place( A[i].child[BIG], sub + s + 1)
     find_place( A[i].child[SMALL], sub)
  endif
end
```

Figure 6: Phase 3 of the sort: putting the elements in their right place

Any processor which completes the second phase advances without delay to the third phase. Using the results from the second phase, calculating the location of each element in the sorted array is now a simple matter. We use the following rule in the routine `find_place`. Let j be some element whose left and right children, $l(j)$ and $r(j)$ correspond to the larger and smaller child respectively. We denote by $P(j)$ j's rank among the elements of `A` after sorting, and by $S(j)$ the size of the subtree rooted at j. Then $P(l(j)) = P(j) + S(r(l(j))) + 1$ and $P(r(j)) = P(j) - S(l(r(j)))$. The routine `find_place()` is initially called with `i = 0` and `sub = 0`.

Since tree based algorithms have been dealt with extensively in the literature, we state the following without proof.

Lemma 2.4 *The second and third phase of the algorithm are both wait-free and require no more than $O(N)$ operations to complete.*

2.3 Run-time analysis

We analyze the running time of the algorithm in the synchronized case, where it is essentially running on a CRCW PRAM. The first phase of the algorithm is a simple parallel Quicksort implementation similar to the one given in [10], which is shown to run in optimal time on a CRCW PRAM. The second and third phases require traversing a binary tree of depth $O(\log N)$. For the synchronous case, it is easy to see that only $O(\log N)$ steps are required (e.g. [27]). We state the following lemma leaving the proof for the full paper.

Lemma 2.5 *Assuming that the elements in the initial array are in random order, each of the algorithm's three steps, when running on a CRCW PRAM has a running time of $O(N \log N/P)$.*

The assumption that elements in the initial array are in random order is needed only for the first phase. We can eliminate this assumption by employing the following work allocation strategy in the first phase of the algorithm. Instead of calling `undone_element` a processor picks one of the elements of `A` uniformly at random. If the element is not `DONE` the processor inserts it into the tree, and calls `propagate_done` as usual. This process continues until a processor has randomly chosen `DONE` elements $\log N$ times in a row. From this stage elements are chosen using `undone_element`. This change guarantees that w.h.p. all nodes in the first $\log N - \log\log N$ levels of the Quicksort tree are chosen uniformly at random. Thus, w.h.p. all nodes at level $\log N - \log\log N$ are roots of a subtree with $O(\log N)$ nodes, and the total sorting takes $O(\log N)$ time.

3 Dealing with Contention

Contention is a phenomenon observed in multiprocessors that occurs when several processors attempt to access the same location in memory at the same time. Since current hardware can only service a constant number accesses per cycle some processors might have their accesses deferred to later cycles, forcing them to wait. Dwork et al. present the first formal complexity model for contention [13]. In their model, if two or more processors attempt to access the same memory location concurrently, one will succeed and the others will *stall.* They differentiate between the *contention of an algorithm,* defined as total number of stalls which can be induced by an adversary scheduler divided by the number of processors, and the *variable-contention*, defined as the worst case number of concurrent accesses to any single variable. They further prove that an adversary scheduler can always cause the variable-contention of a wait-free algorithm running on P processors to be $O(P)$, so we cannot use this measure directly. Also, for randomized algorithms contention depends on the random choices made by the processors. For these reasons we define contention as the maximum number of concurrent accesses to any single variable that occurs with non-negligible probability when the algorithm is run on a CRCW PRAM. This is a natural measure since it makes no assumptions about how the machine handles concurrent accesses, it simply asks "How many are there likely to be?"

The algorithm presented in the previous section suffers $O(P)$ contention, for example, at the very start when all processors attempt to install the element they are working on at the root. Once the tree contains $O(P)$ levels, the random nature of element selection will reduce the expected contention at each element to $O(1)$. If $P \ll N$ initial contention is less of an issue, even under QRQW [14] assumptions since the running time of the algorithm will be dominated by N. As N approaches P contention begins to play a greater role in determining running time. In this section we try to overcome this to some extent by presenting a method for lowering contention to $O(\sqrt{P})$.

3.1 Low contention WATs

We begin by introducing low contention work assignment trees (LC-WATs), which solve the write-all problem in time $O(\log P)$ with expected $O(\log P/\log\log P)$ contention.

```
Repeat forever
  i = a random node of the tree
  If i is an unmarked leaf Then
     Do the work for i
     Mark i DONE
  Else If i is an unmarked inner node Then
     If both of i's children are marked DONE Then
        Mark i DONE
        If i is the root of the tree Then
           Mark the root ALLDONE
        Endif
     Endif
  Else If i is an inner node marked ALLDONE Then
     Mark both of i's children ALLDONE
     Quit
  Endif
Endrepeat
```

Figure 7: Low Contention Work Assignment

The code in Figure 7 follows the work allocation scheme of [27], but has been modified for low contention. In the algorithm of [27] processors must constantly check the root to find out whether all the work of the tree has been done, this causes the root to be a source of $O(P)$ contention. We modify the algorithm by having the processor that would have set the root to `DONE` set it instead to `ALLDONE`. This `ALLDONE` value propagates down the tree, till in time $O(\log P)$ w.h.p. most of the tree is marked `ALLDONE`. We thus trade an additive log factor in time for low contention completion discovery.

Lemma 3.1 *Assuming $O(1)$ work per tree leaf. Under synchronous execution assumptions w.h.p the LC-WAT algo-*

rithm given above terminates in $O(\log P)$ time, with maximum contention $O(\log P/\log\log P)$.

Proof: We first bound the run-time of the algorithm. A node can be marked DONE only after its two children are marked DONE. Once the two children are marked the probability that the node is not marked in the next t steps is bounded by $(1-\frac{1}{2P})^{tP}$. We bound the probability that the root was not marked after T steps using a *delay sequence* argument similar to the one used in packet routing analysis [31]. Let $x_0,...,x_{\log P}$ be a sequence of nodes such that (1) x_0 is the root of the tree; (2) x_i is the last child to be marked DONE among the two children of x_{i-1} (ties are broken arbitrarily). Let t_i be the time node x_i was marked DONE, let $t_{\log P+1}=0$. If the root was not marked after T steps then

$$\sum_{i=0}^{\log P} t_i - t_{i+1} \geq T.$$

Let $s_i = t_i - t_{i+1}$, then x_i was marked s_i steps after its two children had been marked. If the root was marked after T steps then there is a root to leaf path for which

$$\sum_{i=0}^{\log P} s_i \geq T.$$

The probability that such a path exists for $T = b\log P$ is bounded by

$$\binom{b\log P-1}{\log P}(1-\frac{1}{2P})^{P(b-1)\log P} \leq \frac{1}{P}$$

for a sufficiently large constant b. Similar argument bounds the probability that dissemination the ALLDONE mark takes more than $b\log P$ steps.

To bound the contention we observe that at each iteration P processors choose randomly between $2P$ locations causing an average of $O(1/2)$ contention per node per step. The probability that through the execution of the algorithm any node experiences a contention of at least $c\log P/\log\log P$ is bounded by

$$4Pb\log P\binom{P}{c\log P/\log\log P}(\frac{1}{2P})^{c\log P/\log\log P} \leq \frac{1}{P}$$

for a sufficiently large constant c. ∎

3.2 Building the Quicksort tree

We now show how to deal with contention in the tree building phase of the algorithm, we assume that work is distributed using LC-WATs. The method we use is based on splitting the sort into three major phases, the first and last of which are based on the sort of the previous section and the middle phase serves as a "glue" between them. For simplicity we will present the algorithm for the case where $P=N$, extending it to other cases is straightforward. Here is a high level view of the sort.

1. Split the P processors into $\sqrt{P}$ groups of $\sqrt{P}$ processors each. Each group sorts a different slice of size $\sqrt{P}$ of the original array in parallel, using the algorithm of section 2.
2. One group, the *winner*, is selected, most likely the first group to finish sorting its slice. This sorted slice is transformed into a fat balanced binary tree with $\sqrt{P}$ copies of the root node.
3. The entire array is sorted using the algorithm of section 2, the only difference is that node values of elements with depth $\leq \log\sqrt{P}$ are read from the fat tree of the previous phase (see also [15]).

The second phase of the algorithm has two new parts: winner selection and fattening of the tree. Low contention winner selection can be achieved using a balanced binary tree (e.g. implemented as an array) whose nodes are all initially set to EMPTY. Processors begin at the tree's leaves and advance towards the root till they reach a node with a value (one that is not EMPTY), they then copy this value to the node's two children. If the root is reached, the processor attempts to acquire it using compare-and-swap. Low contention is achieved by having processors enter the tree in waves with appropriate constant spacing between them. The first wave has a single processor, each successive wave has twice as many processors as the last, till the $\log P$-th wave has $P/2$ processors. If processors advance without delays, the root will be acquired by a single processor with $O(1)$ contention, who will also write its value to the root's two children. Each child will in turn be read by a single processor who will continue the propagation towards the leaves. In this way we can select the winner in $O(\log P)$ time with $O(1)$ contention, for the synchronous case.

Once a winner is selected, we use its sorted slice as the base for a fat balanced binary tree which will serve for the top levels of the Quicksort tree. A balanced binary tree is a binary tree, where each node has two children. The tree is made fat by duplicating the values at its nodes. We make k copies of the value at the root node, ck copies of each of the values at the root's children, and c^ik copies of the values of the children at the i-th level. We choose $k=\sqrt{P}$ and c such that the total number of values in the tree is approximately P. Recall that the total number of nodes in the tree is $\sqrt{P}$. To fill the fat tree with values we will use an approximation of the write-all problem, *write-most*. Each processor reaching this stage will choose $\log P$ values of the fat tree at random, and write into them values taken from the sorted slice of A chosen in the previous stage (the winning slice). Any two processors choosing the same node of the fat tree, even if they choose different duplicate values in that node must read from the same element of A. Since the largest node has $\sqrt{P}$ values, the expected number of processors choosing that node at any one time is $P/\sqrt{P}=\sqrt{P}$. Thus the greatest expected read contention for any value in A is also $\sqrt{P}$. This way we can fill the fat tree w.h.p in time $\log P$, with contention $\sqrt{P}$. The main difference between our fat-tree and that of Gibbons et al. [15] (other than the fact that the sizes are different), is that they use binary broadcast to fill the tree, a method that is not wait-free, while we employ randomized write-most to ensure independence between processors.

We can now apply the first stage of the sorting algorithm, build_tree, to the entire array and construct the Quicksort tree with expected contention at most $\sqrt{P}$. Processors reading the fat tree have access to multiple copies, which reduce contention. The value of c must satisfy the equation $(2c)^{\log\sqrt{P}+1}-1=2cP$, which can be shown to imply $c>\frac{1}{2}$. Therefore the node with the largest ratio of processors to duplicate values will be the root which is accessed by P processors and has $\sqrt{P}$ duplicates, leading to $\sqrt{P}$ contention. Once out of the fat tree, the processors have been split into groups of expected size $\sqrt{P}$ with each group operating on a different node.

3.3 Completing the sort

```
Repeat forever
  i = a random node of the tree
  If i is not marked
     If i is the root, or
        i's parent's PLACE is set Then
        Set i's PLACE based on the parent
        If i is a leaf
           Mark i as DONE
        Endif
     Endif
     If both i's children are marked DONE Then
         Mark i as DONE
         If i is the root Then
            Mark the root ALLDONE
         Endif
     Endif
  Else If i marked ALLDONE Then
      Mark both of i's children ALLDONE
      Quit
  Endif
Endrepeat
```

Figure 8: Low Contention Place Finding

We complete the sort by giving low contention versions of the second and third phases of the sort: `tree_sum` and `find_place`. Tree summation follows the algorithm for LC-WATs in figure 7, with the following minor changes:

1. The work for each leaf is simply setting its SUM value to 1.

2. Before marking an inner node as DONE we set its SUM value to the sum of each of its children's SUM values plus 1.

We can find an element's location using a similar method as detailed in figure 8. We set a node's PLACE based on its parent's location using the equations in section 2. When processors are all participating, this phase takes $O(\log P)$ time, in three passes: first PLACE values are written going down the tree, then DONE values propagate up the tree, and finally, ALLDONE values spread back down the tree.

4 Conclusions

This paper presented the first run-time optimal wait-free sorting algorithm. The algorithm, which employs randomization, completes the sort in $O(N \log N/P)$ time when run on a CRCW PRAM and is guaranteed to complete the sort in the face of any adversary scheduler. A detailed analysis of the work performed by the algorithm in the asynchronous case is still required. Using low contention randomized solutions for winner selection and work allocation we have shown how to reduce the contention suffered by the algorithm to $O(\sqrt{P})$ in the synchronous case. In the asynchronous case it has been shown that an omnipotent adversary can always cause a wait-free algorithm to suffer $O(P)$ contention [13]. Still, it would be interesting to present an analysis of our contention reduced variant in the face of a weaker adversary.

5 Acknowledgments

Thanks are due to Alex Shvartsman and to The Collection of Computer Science Bibliographies maintained by Alf-Christian Achilles at
http://liinwww.ira.uka.de/bibliography/index.html.

References

[1] Afek, Y., Dauber, D., and Touitou, D. Wait-free made fast (extended abstract). In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing* (Las Vegas, Nevada, 29 May–1 June 1995), pp. 538–547.

[2] Ajtai, M., Komlós, J., and Szemerédi, E. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing* (Boston, Massachusetts, 25–27 Apr. 1983), pp. 1–9.

[3] Anderson, R. J., and Woll, H. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing* (New Orleans, LS, May 1991), B. Awerbuch, Ed., ACM Press, pp. 370–380.

[4] Assaf, S. and Uppal, E. Fault tolerant sorting networks. *SIJDM: SIAM Journal on Discrete Mathematics 4* (1991).

[5] Banatre, M., Muller, G., Rochat, B., and Sanchez, P. Design decisions for the FTM: a general purpose fault tolerant machine. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)* (1991), 71–8.

[6] Barnes, G. Wait-free algorithms for heaps. Technical Report 94–12–07, University of Washington, Department of Computer Science and Engineering, 1994.

[7] Batcher, K. E. Sorting networks and their applications. *Proceedings of AFIPS Spring Joint Computer Conference* (1968), 307–314.

[8] Brown, T., and Xiong, R. A parallel Quicksort algorithm. *Journal of Parallel and Distributed Computing 19*, 2 (Oct. 1993), 83–89.

[9] Buss, J. F., Kanellakis, P. C., Ragde, P. L., and Shvartsman, A. A. Parallel algorithms with processor failures and delays. *Journal of Algorithms 20*, 1 (Jan. 1996), 45–86.

[10] Chlebus, B. S., and Vrto, I. Parallel Quicksort. *Journal of Parallel and Distributed Computing 11*, 4 ([4] 1991), 332–337.

[11] Cole, R. Parallel merge sort. *SIAM J. Comput. 1*, 4 (Aug. 1988), 770–785.

[12] Cole, R., and Zajicek, O. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures* (Santa Fe, NM, June 1989), A.-S. ACM-SIGARCH, Ed., ACM Press, pp. 169–178.

[13] DWORK, C., HERLIHY, M., WAARTS, O. Contention in Shared Memory Algorithms. In *Proceedings of the Twenty Fifth Annual ACM Symposium on Theory of Computing* (1993), pp. 174–183

[14] GIBBONS, P. B., MATIAS, Y., AND RAMACHANDRAN, V. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. In *Proceedings of the 5th ACM-SIAM Symp. on Discrete Algorithms*, January, 1994, pp. 638–648.

[15] GIBBONS, P. B., MATIAS, Y., AND RAMACHANDRAN, V. The queue-read queue-write asynchronous PRAM model. *Lecture Notes in Computer Science 1124* (1996), pg. 279

[16] GREENWALD, M. AND CHERITON, D. . The Synergy Between Non-blocking Synchronization and Operating System Structure. In *Proceedings of the Second Symposium on Operating System Design and Implementation.* USENIX, Seattle, October, 1996, pp 123–136.

[17] HERLIHY, M. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[18] HERLIHY, M. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems 15*, 5 (Nov. 1993), 745–770.

[19] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, May 17–19, 1993), ACM SIGARCH and IEEE Computer Society TCCA, pp. 289–300.

[20] HOARE, A.. Quicksort. In *C. A. A. Hoare and C. B. Jones (Ed.), Essays in Computing Science, Prentice Hall.* 1989.

[21] ISRAELI, A., AND RAPPOPORT, L. Efficient wait-free implementation of a concurrent priority queue. *Lecture Notes in Computer Science 725* (1993), 1–??

[22] KANELLAKIS, P. C., AND SHVARTSMAN, A. A. Efficient parallel algorithms can be made robust. In *Proceedings of the 8th Annual Symposium on Principles of Distributed Computing* (Edmonton, AB, Canada, Aug. 1989), P. Rudnicki, Ed., ACM Press, pp. 211–222.

[23] KEDEM, Z. M., PALEM, K. V., AND SPIRAKIS, P. G. Efficient robust parallel computations (extended abstract). In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing* (Baltimore, Maryland, 14–16 May 1990), pp. 138–148.

[24] MA, Y. An $O(n \log n)$-size fault-tolerant sorting network (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, 22–24 May 1996), pp. 266–275.

[25] MARTEL, C., SUBRAMONIAN, R., AND PARK, A. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science* (St. Louis, MS, Oct. 1990), IEEE, Ed., IEEE Computer Society Press, pp. 590–599.

[26] MARTEL, AND C., SUBRAMONIAN On the Complexity of Certified Write-All Algorithms, *J. of Algorithms* 16(3): pp. 361–387 (May 1994)

[27] MARTEL, C., SUBRAMONIAN, R., AND PARK, A. Work Optimal Asynchronous Algorithms For Shared Memory Parallel Machines *SIAM J. Computing* 21(6): pp. 1070–1099, (1992).

[28] MARTEL, C. U., AND GUSFIELD, D. A fast parallel Quicksort algorithm. *Information Processing Letters 30*, 2 (Jan. 1989), 97–102.

[29] SHAVIT, N., AND TOUITOU, D. Software Transactional Memory. In *Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing (PODC'95)* (Aug. 1995).

[30] SUN, J., AND GECSEI, J. A multiple-fault tolerant sorting network. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)* (1991), 274–81.

[31] Upfal, E. "Efficient schemes for parallel communication." *J. ACM* 31(3)1984 507–517.

[32] YEN, I.-L., BASTANI, F., AND LEISS, E. An inherently fault tolerant sorting algorithm. In *Proceedings of the 5th International Parallel Processing Symposium* (Anaheim, CA, Apr.–May 1991), V. K. P. Kumar, Ed., IEEE Computer Society Press, pp. 37–42.

[33] ZHANG, W., AND RAO, N. Optimal parallel Quicksort on EREW PRAM. *BIT: BIT 31* (1991).

[34] ZHOU, B. B., BRENT, R. P., AND TRIDGELL, A. Efficient implementation of sorting algorithms on asynchronous mimd machines. Tech. Rep. TR-CS-93-06, Australian National University, Computer Science Department, May 93.