A Wait-Free Sorting Algorithm

Nir Shavit * Eli Upfal † Asaph Zemach ‡

May 14, 1998

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. We show two variants of the algorithm, one deterministic and one randomized and prove that, with high probability, the latter suffers no more than $O(\sqrt{P})$ contention when run synchronously. Known sorting algorithms, when made wait-free through previously established transformation techniques, have complexity $O(\log^3 N)$. The algorithm we present here, when run in the CRCW PRAM model, executes with high probability 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.

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 high probability the algorithm runs in optimal time and has maximum memory contention of $O(\sqrt{P})$. Herlihy [27] 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 possible failure of other processors. Contention is the empirically observed phenomenon [5, 21] that when several processors attempt to access the same memory location (e.g. a variable) at the same time, a delay occurs, since the hardware can only handle a limited number of simultaneous requests. Keeping contention low is therefore a high priority in terms of program performance.

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 [26] 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 one can begin the sort by spawning a thread for each idle processor in the machine. If during the execution a processor is needed elsewhere, one 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

*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.

†Computer Science Department, Brown University.

‡Tera Computer Company and 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.

processors become free, one 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. One 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, any such thread can summarily be destroy. 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, this leads 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 [30] of which there have been a number of parallel implementations. For the CRCW PRAM, there is the algorithm of Martel and Gusfield [38], 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 [17] 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 [43] present an algorithm with a running time of $O((\log P + N/P)\log N)$. This was later improved upon by Brown and Xiong [12] 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 [27] Herlihy also gave a general method for the construction of wait-free objects [28]. Unfortunately, the algorithm resulting from implementing a "sorting-object" using this method (or the improvements of Afek et al. on it [1]) is inefficient. Processors wishing to update the shared object will have to 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 $k = P$ and $O(PN\log N)$ cost per operation, which will not yield good performance. Similar objections apply to Shavit and Touitou's software implementation [39] of Herlihy and Moss' transactional shared memory [29], while proposed hardware implementations are limited in size [9]. 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' [10] 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 [31] priority queue, besides requiring a non-standard two word Compare&Swap operation also employs a "helping" method which limits concurrency (this is discussed in [39]). 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 [4, 11, 18] and apply a transformation technique which simulates a reliable PRAM on a faulty one. This idea was first introduced by Kanellakis and Shvartsman in [32], and later improved upon by Kedem et al. [33]. Both of these results are for the fail-stop model. In the general asynchronous model the results of Anderson and Woll [6] and Buss et al. [16] 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. [35] 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 [6], since they require synchronization at the end of every PRAM step.

A previous result in fault-tolerant sorting is given by Yen et al. [42]. It 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 of fault tolerant sorting networks, see for example the articles by Assaf and Upfal [7], Ma [34], and Sun and Gecsei [40]. These papers deal 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 [19] proposed the APRAM model for designing parallel algorithms to work in an asynchronous setting. Zhou et al. [44] present a sorting algorithm for asynchronous machines that is not wait-free. Neither is the recent sorting algorithm of Gibbons et al. [24] for the QRQW asynchronous PRAM. This sample-sort based algorithm is somewhat similar to the one we use here in that it uses binary search trees built by a sampling phase, where we construct binary pivot trees. The main differences being that we use $O(N)$ space as opposed to their $O(N \log N)$, and that our algorithm is guaranteed to run to completion once it is started, whereas theirs might require a restart phase if failure is detected during the run. All of these models avoid making any timing assumptions, but they do not allow processor failures, and hence do not produce wait-free algorithms. These results indicate the need to develop an efficient sorting algorithm designed specifically for the wait-free case.

1.2 Contention

Contention is a phenomenon observed in multiprocessors [21, 25] 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 memory access operations per cycle some processors might have their access operations deferred to later cycles, forcing them to wait. On real machines, contention can account for a large portion of a program's run time [2] and ignoring this issue can lead the creation of algorithms that appear sound but perform poorly. Dwork et al. present the first formal complexity model for contention [20]. 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. Much of the subsequent work using formal contention models has dealt with amortized contention of counting networks [3, 13, 14] and

[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.

is based on the fact that networks have a regular, well defined structure. Since there is no bound on the number of tokens that may pass through a network, measuring amortized contention is a natural choice much in the same way that measuring the amortized complexity of operations on serial data structures.

We use a similar definition which we feel is more suited to an algorithm with well defined start and finish. We define contention as the maximum number of concurrent accesses to any single variable that occur 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?"

1.3 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 achieved with high probability under the assumption that the execution is "normal", that is, all processors participate in the algorithm and incur no delays. This form of run time analysis was proposed by Attiya et al. [8], as a way of capturing the overhead of wait-free algorithms. The idea being that since the common basis for comparison is the PRAM, which is faultless and synchronous, we should apply the same standard to other algorithms as a way of measuring any extra work performed to achieve wait-freedom.

We are able to achieve optimal running time by not using a standard PRAM sorting algorithm which generally requires $O(\log N)$ 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 a simple, deterministic version of the algorithm we turn our attention to the issue of contention and show how randomization can be used to reduce contention. We first present a simple low contention work allocation scheme that when combined with low contention winner selection and approximate write-all [32] (actually, write-most) yields a randomized wait-free sorting algorithm with contention $O(\sqrt{P})$ with high probability.

2 A Wait Free Sorting Algorithm

One of the challenges of writing wait-free code for manipulating a number of objects concurrently 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 Kanellakis and Shvartsman [32]: 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. These structures have been used extensively in the literature (see for example [6, 16, 32, 36]). Our implementation in Figure 1 follows Algorithm X of Buss et al. in [16]. The second operand of the routine `next_element`, `i`, will usually be the leaf of the WAT whose work the calling processor has just completed. The routine climbs the tree from `i` until it

```
 1  function next_element(tree: WAT(N), i: integer) :
                                        returns integer
 2  begin
 3    tree[i] := DONE
 4    repeat
 5      s := sibling(i)
 6      p := parent(i)
 7      if tree[s] = DONE then
 8         tree[p] := DONE
 9         i := p
10         if p = ROOT return DONE
11      endif
12    until tree[s] <> DONE
13    i = s
14    while not leaf(i)
15      if tree[ left_child(i) ] <> DONE then
16         i := left_child(i)
17      else if tree[ right_child(i) ] <> DONE then
18         i := right_child(i)
19      else
20         return i
21      endif
21    endwhile
22    return i
23  end
```

Figure 1: Work-Assignment-Tree algorithm

finds a node for whom one child is not yet marked DONE. During the climb at a given node one can determine the status of its parent by examining the status of its sibling. This is because the current node (in fact, the entire current subtree) is known to be marked DONE. If the routine reaches the root and marks it DONE it means that all the leaves have been handled and the special value DONE is returned as an indicator. If a sibling, s, not marked DONE is found, the routine descends the tree and normally returns an un-DONE leaf the tree rooted at s. A special case occurs if during the descent the processor discovers that the information at a node is outdated i.e. even though both children are DONE the node was not yet marked as completed. In this case the processor stops its descent and returns that inner node. The routine could have been written to return to the ascent phase to search for a different leaf, but we found this version makes the proofs simpler.

Lemma 2.1 *The routine* next_element *is wait-free and completes in* $O(\log N)$ *time.*

Proof: The routine contains two loops, the one in line 4–12 climbs the tree at each iteration, and so cannot climb more than $\log N$ steps before stopping. Similarly the second loop, in lines 14–20, descends the tree at each iteration. ■

Corollary 2.2 *Let* S *be set of all calls to* next_element *which have completed before a given call at time* t*. Let* S' *denote the set of all initial starting nodes (the second operand) of the calls in* S*. The call at time* t *will return a node of the tree not in* S'*, or* DONE *if* S' *contains all of the tree's nodes.*

Using Lemma 2.1 and its corollary 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()

```
procedure wait-free-algorithm
shared variables
  work: WAT(N)
processor private variables
  i: integer
begin
  i := leaf number N * PID / P + 1
  repeat
    if leaf(i) func(i)
    i := next_element(work, i)
  until i = DONE
end
```

Figure 2: A skeleton wait free algorithm

with the operation `B[i]:=1` for some array `B` of size `N`, we get a solution for the write-all problem.

Lemma 2.3 *Let $P = N$, and assume the routine* `func()` *can take no more than K time steps to complete. Then the skeleton wait-free algorithm of Figure 2 when run on a faultless CRCW PRAM completes in $O(K + \log N)$ time steps.*

Proof: Initially each processor is assigned a different leaf of the WAT, within K time steps all processors will complete working on their leaf, and all that remains to show is that it takes $\log N$ additional steps for this information to propagate to every node of the tree. When the last processor completes the leaf it was assigned, all nodes at depth $\log N$ will be marked `DONE`, though there may be processors who completed their own leaf early and are still working on other leaves. Within at most K additional time steps all processors will be in the WAT at depth at most $\log N$ and will never be assigned a leaf again. Let S denote the set of processors at depth $\log N$ at the $2K$-th time step. Each processor in S examines its node's sibling, notices that it is `DONE`, ascends to depth $\log N - 1$, and sets the node there to `DONE`. For every node at depth $\log N - 1$ the following is true, either there is a processor who has just set its value to `DONE`, or a processor has passed through that node previously and set its value to `DONE`, thus all processors at level $\log N - 1$ rise to level $\log N - 2$ in the next time step. The same logic continues to hold, and at the $2K + i$-th time step, all processors at level $\log N - i - 1$ raise to level $\log N - i$. Within $\log N$ time steps all processor reach the tree root and leave the tree. ■

2.2 The sorting algorithm

We now present in detail our wait-free algorithm for sorting an array `A` of N elements using P processors. 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 to each record of `A` to point to subtrees of smaller and larger nodes. Initially, all pointers have the distinct value `EMPTY`. The first phase follows the scheme outlined in Figure 2 with the routine `build_tree` of Figure 4 replacing the call to `func`. First we note the fact that A[1], being the first pivot need not be inserted into the tree (line 5). A processor p which is inserting record i first compares its key to the key of the 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 14). After the call either p or some other processor will have managed to install its records as the child of the root.

```
const
  BIG = 0
  SMALL = 1

type Element is
  key:    any-type
  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

```
 1  procedure build_tree(i: integer)
 2  processor private variables
 3    parent,side: integer
 4  begin
 5    if i = 1 return
 6    parent := 1
 7    while true
 8      if  A[parent].key > A[i].key then
 9          side := SMALL
10      else
11          side := BIG
12      endif
13      compare_and_swap(A[parent].child[side],EMPTY, i)
14      if A[parent].child[side] = i then
15         return
16      else
17         parent := A[parent].child[side]
18      endif
19    endwhile
20  end
```

Figure 4: Core of phase 1 of the sort: building the Quicksort tree

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. Successful installation of i (either by p or by some other processor simultaneously working on i) terminates the routine. 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 routine with the same value for `parent`.
2. For a given pair of values of `i` and `parent`, the comparison in line 8 always yields the same results.
3. For a given pair of values of `parent` and `side` the read operation in line 14 always returns the same value, which is never `EMPTY`.

4. The sequence of values of the variable `parent` determines a path along the nodes of the Quicksort tree. As a direct consequence of facts 1–3, we get that two processors with the same value for `i` would get the same value of `parent` in each iteration of the loop in lines 7–19, and therefore follow the same path down the tree.

5. From fact 3 follows that the value of `i` determines a unique path down the tree into which insertion attempts (line 13) are made, so 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 7–19 is done with a different value of `parent`.

6. Each time the `compare_and_swap` in line 13 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 2.4 *The loop in lines 7–19 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 5). 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 5 and 6), eventually either the `compare_and_swap` succeeds, or a processor encounters its own value in the tree and exits. ■

We interpret Lemma 2.4 to mean that the routine `build_tree` is wait-free and requires no more than $O(N)$ operations to run. This along with Lemma 2.3 proves that the entire first phase of the algorithm is wait-free and can be completed in $O(N(\log N + N)) = O(N^2)$ operations. We now show that it builds the pivot tree correctly.

Lemma 2.5 *When the first processor completes the first phase of the algorithm 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 8 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. ■

```
function tree_sum(i: integer, d: 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
     side = d-th bit of PID
     sum := tree_sum( A[i].child[side] , d+1)
     sum := sum + tree_sum( A[i].child[1 - side] , d+1 )
  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, d: 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 d-th bit of PID == SMALL then
     find_place( A[i].child[SMALL], sub, d+1)
     find_place( A[i].child[BIG], sub + s + 1, d+1)
  else
     find_place( A[i].child[BIG], sub + s + 1, d+1)
     find_place( A[i].child[SMALL], sub, d+1)
  endif
end
```

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

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 each processor's unique processor ID (assumed to be in the range $0, \ldots, P-1$) to spread the processors around the tree. Code for this phase appears in Figure 5.

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`, `sub = 0`, and `d = 0`.

Since tree based algorithms have been dealt with extensively in the literature, we state the following without proof (see for example Kanellakis and Shvartsman [15]).

Lemma 2.6 *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.

Lemma 2.7 *Let K be a bound on the running time of* `build_tree`*. The first phase of the algorithm, when run on a faultless CRCW PRAM has running time of $O(N(\log N + K)/P)$.*

Proof: Our work allocation scheme initially assigns each processor a leaf of of the WAT spaced N/P leaves apart. Processors are assigned work in a localized manner, meaning

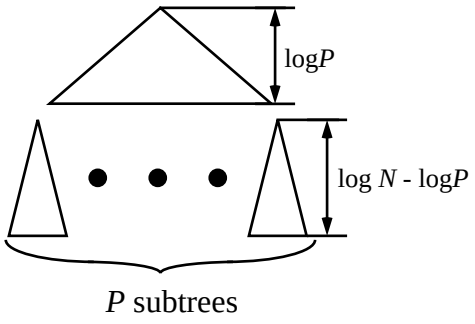

Figure 7: Division of subtrees between processors: after descending $\log P$ steps from the root every processor is in a sense assigned its own subtree of depth $\log N - \log P$.

that if T_1 and T_2 are two disjoint subtrees of the WAT, and processor p is currently assigned a leaf of T_1, p will complete every leaf of T_1 (and mark all its nodes `DONE`) before being assigned any leaf of T_2. We can therefore view processors as initially being assigned subtrees of depth $D = \log N - \log P$ rather than leaves (see Figure 7). Processors complete working on their subtrees in time $O(2^D(D+K)) = O(N/P(\log N + K))$. We can therefore view running the first phase of the algorithm as a special case of Lemma 2.3 where the running time of `func` is $O(N/P(\log N + K))$ and there are P processors. Total running time will then be $O(N/P(\log N + K) + \log P) = O(N/P(\log N + K))$ for $P \leq N$. ■

The bound K on the running time of `build_tree` corresponds to the depth of the Quicksort tree being built. Therefore, second and third phases require traversing a binary tree whose depth is K. A processor p in the second (summation) phase, operating on a node n at depth d, checks first whether the node is a leaf, and if so it marks its size as "1" and ascends to the node's parent. Otherwise, it visits both of n's children, calculates each of their sizes, and sets n's size accordingly. The order in which p visits n's children corresponds to the d-th bit of p's processor ID (PID). To avoid duplicating other processor's work, p will not descend to children of nodes whose size is known. In this phase information propagates bottom-up: only when the size of a node's children is known can the size of the node be determined. Let us assume this phase runs on a complete binary tree of K levels. Initially, all processors start at the root, odd numbered ones will then descend to the root's `SMALL` child, while even numbered ones will go to the `BIG` child. This process continues until all processors are at depth $\log P$, at which point each one is alone at the root of a tree of depth $K - \log P$. Processors finish summing their subtrees in unison and begin ascending the tree. Since the algorithm avoids descending into trees which have already been summed, all processors now ascend to the root simultaneously. The total running time is therefore $\log P + 2^{K-\log P} + \log P = O(\log P + M/P)$ where $M = 2^K \leq N$. If the binary tree of the depth K is not complete, the algorithm will not take longer to run: the processor (or set of processors) which finishes its tree last will move monotonically up as soon as its tree is complete since all nodes it encounters can be shown to be complete. The proof is essentially the same as that of Lemma 2.7. The algorithm's third phase is analogous to the second, and has the same running time.

Lemma 2.8 *Assuming that the elements in the initial array are in random order, the wait-free sorting algorithm, when run on a faultless CRCW PRAM has a running time of $O(N \log N/P)$ with high probability.*

Proof: The sum of the running times of the first, second and third phases calculated above is: $O(N/P(\log N + K))$ for a Quicksort tree of depth K. If the elements of A are in random order, the Quicksort tree can be shown to have depth $O(\log N)$ with high probability (see for example [17]). The resulting total running time is then $O(N/P \log N)$. ■

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 will pick one of the elements of `A` uniformly at random. If the element is not `DONE` the processor will insert it into the tree, and propagate its `DONE` value up the tree via a sequence of operations like that of lines 4–12 of `next_element`. This continues until a processor has randomly chosen `DONE` elements $\log N$ times in a row. From this stage elements are chosen using `next_element`. This change guarantees that with high probability all nodes in the first $\log N - \log\log N$ levels of the Quicksort tree are chosen uniformly at random. Thus, with high probability all nodes at level $\log N - \log\log N$ are roots of a subtree with $O(\log N)$ nodes, and the total sorting time remains $O(N/P \log N)$.

3 Dealing with Contention

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 [22] 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 randomized 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 $O(\log P/\log\log P)$ contention with high probability.

The code in Figure 8 follows the work allocation scheme of Martel et al. [37], but has been modified for low contention. In the algorithm of [37] processors must constantly check the root to find out whether all the work of the tree has been completed. 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 $O(\log P)$ with high probability, 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 with high probability the LC-WAT algorithm 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 by Upfal in packet routing analysis [41]. 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

```
procedure low_contention_work
shared variables
  tree: LCWAT(n)
processor private variables
  i: integer
begin
  while true
    i := random_node ( tree )
    if tree[i] = EMPTY  then
      if leaf(i) then                              // found un-DONE leaf
         func(i)                                   //    do the leaf's work
         tree[i] := DONE                           //    mark the leaf done
      else
        if tree[ left_child(i) ] = DONE and        // found node with
           tree[ right_child(i) ] = DONE then      // both children DONE
            if root(i) then                        //   it is the root
              tree[i] = ALLDONE                    //     ALL work DONE
            else                                   //   else, it is inner
              tree[i] = DONE                       //     mark it DONE
            endif
        endif
      endif
    else
      if tree[i] = ALLDONE and not leaf(i) then // found ALLDONE marker
        tree[ left_child(i) ] := ALLDONE        // propagate to children
        tree[ right_child(i) ] := ALLDONE
        return                                  // quit
      endif
    endif
  endwhile
end
```

Figure 8: Low Contention Work Assignment

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. A similar argument bounds the probability that dissemination of the ALLDONE mark through the tree 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 [23] with $\sqrt{P}$ copies of each 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 (This is similar to the approach used by Gibbons et al. [24]).

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 we assume that all processors arrive at the winner selection phase simultaneously, are assigned delays as above and operate in PRAM-like synchrony then the root will be acquired by a single processor after $\log P$ steps with $O(1)$ contention. That processor 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. In the context of our algorithm processors cannot be guaranteed to arrive at this phase simultaneously and will generally arrive within a span of $O(\log P)$ time steps. Instead of pre-assigning delays we will have processors choose delay times randomly so that the expected distribution will be the same. Figure 9 provides pseudo-code for this phase.

Lemma 3.2 *If all the processors reach the winner selection routine of Figure 9 within a span of $O(\log P)$ time steps then for an appropriate constant K, with high probability the routine selects a winner in time $O(\log P)$ with expected contention $O(\log P)$.*

Proof: The winner selection routine consists of two phases: the wait phase, and the tree traversal phase. Since a processors spends no more than $K \log p$ steps in the wait phase, and then traverses a path on the tree of no more than $\log P$ nodes the routine run-time is $O(\log P)$.

```
function select_winner(candidate: integer) : returns integer
shared variables
  winner : tree of P leaves initialized to EMPTY
processor private variables
  i: integer
  j: a node of winner
begin
  s := 0
  while toss_coin() = Heads and s < log(P)
    s := s + 1
  endwhile
  for i = 1 to K*(log(P)-s)
  endfor
  j = leaf number PID of winner
  while winner[j] = EMPTY and not root(j)
     j = parent(j)
  endwhile
  if root(j) then
     compare_and_swap(winner[j] , EMPTY, candidate)
  endif
  winner[ left_child(j) ] = winner[j]
  winner[ right_child(j) ] = winner[j]
end
```

Figure 9: Low Contention Winner Selection

To bound the contention we observe that contention can occur only between processors that exit the wait phase at the same step. Let z_i be the number of processors leaving the wait phase at time iK. Let $z_i = x_i + y_i$, where x_i are processors that had zero wait (s$= \log P$) and $y_i = z_i - x_i$. Since for any given processor $Prob(\mathrm{s} = \log P) = 1/P$, even if all processors start the routine at time iK, with high probability $x_i = O(\log P)$. Next we observe that if a processor was in the wait phase at time $(i-1)K$, the probability that it will exit the wait at time $(i-1)K$ is exactly half the probability of exiting the wait at time iK. Thus, either $z_i \leq \log P$, or with high probability $y_i \leq 2(1+\epsilon)z_{i-1} + \log P$, for some small constant ϵ. Thus, when the z_i processors traverse the tree at least $\frac{1-\epsilon}{2} z_i - O(\log P)$ nodes are not EMPTY. Since the z_i processors are randomly distributed between these nodes, with high probability the contention is $O(\log P)$. ■

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: $\sqrt{P}$ copies of the value at each node are made. Recall that the total number of nodes in the tree is $\sqrt{P}$ so that all in all we have P copies. 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 there are $\sqrt{P}$ nodes, the expected number of processors reading from the same location in A concurrently is $P/\sqrt{P} = \sqrt{P}$. This way we can fill the fat tree with high probability in time $\log P$, with contention $\sqrt{P}$. The main difference between our fat-tree and that of Gibbons et al. [24] (other than the fact that the sizes are different), is that [24] use binary broadcast to fill the tree, a method that is not wait-free, while we employ randomized write-most to ensure independence of actions 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. This reduces contention. 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 are divided into groups of expected size $\sqrt{P}$, each group operating on a different node.

3.3 Completing the sort

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 8, 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. Again nodes are repeatedly picked at random, and the following actions are taken:

1. For the root: the root's place can be calculated immediately using the size of its `SMALL` subtree. If both of the root's children are marked `DONE` the root is marked `ALLDONE`.
2. For an inner node whose parent's `place` is known: the node's `place` is calculated.
3. For a leaf whose parent's `place` is known: the leaf's `place` is calculated and the leaf is marked `DONE`.
4. For a node for which both children are marked `DONE`: the node is marked `DONE`.
5. For a node marked `ALLDONE`: the value is propagated to the node's children (if there are any) and the processor quits.

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.

Lemma 3.3 *For both second and third phases contention is the same as for the LC-WAT algorithm of Figure 8.*

Proof: The second phase is nearly word for word identical to the algorithm of Figure 8. The third phase follows the same "blueprint", the only difference being that processors access a constant number of additional memory locations at each node. Since these locations are in the vicinity of a processor's randomly chosen node (either a parent or the children) knowing the number of processors choosing the same node is enough to bound the contention (up to a constant factor), so Lemma 3.1 applies. ■

4 Conclusions

This paper presented the first run-time optimal wait-free sorting algorithm. The algorithm completes the sort in $O(N \log N/P)$ time, with high probability, when run on a CRCW PRAM. 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 [20]. Still, it would be interesting to present an analysis of our contention reduced variant in the face of a weaker adversary.

While this algorithm is not immediately practical, it does make use of some interesting general constructions. These constructions for solving work-allocation, write-most, and winner-selection suffer contention of no more than $O(\log P)$ with high probability. We believe they represent building-blocks that are simple, efficient and of low enough contention that they could form the basis of other, more practical, wait-free algorithms.

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] A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 396–406, May 1989.

[3] W. Aiello, R. Venkatesan and M. Yung. Coins, Weights and Contention in Counting Networks. In *13th ACM Symposium on Principles of Distributed Computing*, pp. 193–205, August 1994.

[4] 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.

[5] T.E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[6] 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.

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

[8] H. Attiya, N. Lynch and N. Shavit. Are Wait-Free Algorithms Fast? *JACM*, 41(4): pp. 725-763, July 1994.

[9] 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.

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

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

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

[13] C. Busch, N. Hardavellas and M. Marvonicolas Contention in Counting Networks In *13th ACM Symposium on Principles of Distributed Computing*, pp. 404, August 1994.

[14] C. Busch and M. Marvonicolas A Combinatorical Treatment of Balancing Networks. *JACM*, 43(5): pp. 794-839, September 1996.

[15] Kanellakis, P. C., and Shvartsman, A. A. Fault-Tolerant Parallel Computation. Kluwer Academic Publishers, Boston, 1997. ISBN 0-7923-9922-6.

[16] 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.

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

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

[19] 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.

[20] 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

[21] D. Gawlick. Processing 'hot spots' in high performance systems. In *Proceedings IEEE COMPCON'85*, Feb. 1985.

[22] 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.

[23] Gibbons, P. B., Matias, Y., and Ramachandran, V. Efficient Low-Contention Parallel Algorithms In *Journal of Computer and System Sciences*, 53(3):417–442, Dec. 1996.

[24] 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

[25] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75, April 1989.

[26] 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.

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

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

[29] 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.

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

[31] Israeli, A., and Rappoport, L. Efficient wait-free implementation of a concurrent priority queue. *Lecture Notes in Computer Science 725* (1993), 1–??

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] Martel, and C., Subramonian On the Complexity of Certified Write-All Algorithms, *J. of Algorithms* 16(3): pp. 361–387 (May 1994)

[37] 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).

[38] Martel, C. U., and Gusfield, D. A fast parallel Quicksort algorithm. *Information Processing Letters 30*, 2 (Jan. 1989), 97–102.

[39] 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).

[40] Sun, J., and Gecsei, J. A multiple-fault tolerant sorting network. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)* (1991), 274–81.

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

[42] 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.

[43] Zhang, W., and Rao, N. Optimal parallel Quicksort on EREW PRAM. *BIT: BIT 31* (1991).

[44] 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.