# Parallelized QuickSort with Optimal Speedup

**David M. W. Powers**[1]

Fachbereich Informatik
Universität Kaiserslautern
6750 KAISERSLAUTERN WEST GERMANY

# 1 Overview

This paper introduces a parallel sorting algorithm based on QuickSort and having an *n*-input, *n*-processor, time complexity of O(log *n*) exhibited using a CRCW PRAM model. Although existing algorithms of similar complexity are known, this approach leads to a family of algorithms with a considerably lower constant. It is also significant in its close relationship to a standard sequential algorithm.

## 1.1 Sorting

Knuth (1973,pp2-3) notes that sorting is estimated to take up 25% of the world's computer time. With the advent of the microcomputer this may well have changed, but it is nonetheless a both practically and theoretically interesting task. Sorting, in the sense of bringing together related things, has now been subsumed by.the more specific task of ordering, and has spawned an enormous number of serial sorting algorithms.

Whilst for specific cases, faster algorithms are known, in general sorting requires O(*n* log *n*) comparisons, and hence time. The algorithms meeting this expected complexity are based on ideas of either partitioning or merging, usually with the aid of explicit or implicit list and/or tree data structures. These are prototypically represented by the algorithms QuickSort and MergeSort (Knuth,1973).

Logically these algorithms require two phases: placement into the tree, and extraction from the tree. In some cases, one or other of these phases can be left implicit. As the tree has logarithmic depth, and each element needs to be placed and/or extracted, the O(*n* log *n*) complexity follows immediately.

---

[1]The work reported was undertaken while the author was at Macquarie University NSW 2009, AUSTRALIA, and was supported by IMPACT Ltd, PETERSHAM NSW 2049, AUSTRALIA. The author is currently supported under ESPRIT BRA 3012: COMPULOG.

## 1.2 Parallel Sorting Algorithms

In view of this sequential result, we would hope for an optimal linear speedup to O(log *n*) when we move to parallel processing on *n* processors. In practise, however, this is not easily achieved.

The infamous AKS result (Ajtai et al, 1983), which was the first to demonstrate the theoretical feasibility of O(log *n*) time on O(*n*) processing elements, is based on a sorting network entailing a fixed sequence of comparisons - but with a rather impractical constant of 6100! This is for achievable datasets ($<2^{6100}$ elements) actually worse than the performance of the older networks/algorithms due to Batcher (1968) - with O($\log^2 n$) and a subunit constant (Parberry,1989).

## 1.3 Parallelized Sorting Algorithms

Several authors have noted that the standard O(*n* log *n*) serial algorithms seem to have serial constraints which prohibit linear parallelization. Knuth (1973,p114) proposed that a "strategy which uses the result of each comparison to determine what keys are to be considered next ... is inappropriate for parallel computations". Only recently has this state of affairs changed.

Cole (1988) has achieved the O(log *n*) bound with a variation of MergeSort - and a considerably smaller constant (normalized at 12). This makes clever use of a sorted sample to partition the streams to be merged into bounded chunks which may be merged in parallel, thus avoiding a linear cost in the merging/extraction process.

The obvious parallel form of QuickSort exhibits an O(*n*) time complexity, and although a number of variants have been proposed, none have hitherto proven to have the desired O(log *n*) complexity (Francis and Pannan, 1989).

## 1.4 A Parallelized QuickSort with Linear Speedup

The present work has achieved the O(log *n*) bound with a variant of QuickSort - achieving a still smaller constant of 2.4 (in terms of the *expected* number of comparisons). The algorithm performs the partitioning implicitly, avoiding the linear number of overt moves normally performed at each level of the tree. The implicitly sorted elements are eventually moved to their desired position (if necessary) using standard techniques.

# 2 Algorithm

The problems of QuickSort and its parallel variants lie in the partition procedure - the recombination is trivial and can be implemented as an implicit concatenation step.

The problems with the partition step have three aspects: the choice of *pivot* (from or appropriate to the present partition); the division of the partition into two (one containing elements "*less than*" and the other those "*greater than*" the *pivot*); and the handling of the *equality* case (do elements *equal* to the *pivot* go into a partition, and if so which).

The choice of *pivot* is important, as a poor choice will result in little or no reduction in the size of one of the partition. In a serial context, this is responsible for the $O(n^2)$ worst case of QuickSort. In the parallel context, it reduces the utilization of processors, leading to a worst case linear number of parallel partitioning steps. If these cost linear time, this can result in a similar quadratic time execution for a parallel algorithm!

The partitioning step is normally conceived of in the context of an *in situ* partitioning, or perhaps in terms of operations on *linked lists*. In both cases the procedure tends to be serial, as the placement of an element to be moved can only be determined in relation to those which have already been processed. If the partioning step requires linear time in the size of the partition, then the linear time for the initial partition step will dominate any later savings due to the parallelism.

The handling of equal elements can also lead to a worst case situation if elements that compare equal to the pivot are naïvely placed consistently in the same partition (always in the righthand partition in terms of the above description).

## 2.1 Description

The worst cases resulting from poor choice of *pivot* or poor handling of *equality* correspond to fairly common situations - and can be attacked with *ad hoc* heuristic techniques applicable only to specific distributions (e.g. use of a sample median where a linear distribution is expected), or by *randomizaition* techniques (i.e. the pivot is chosen randomly and any element equal to it is distributed randomly).

In the algorithm described here, the latter approach is used, and a worst case performance would imply that the *random* choices have been a perfect *oracle* for either the sort or the reverse ordering, with equal elements being consistently distributed to the same (longer) side. The expected length of the longer partition is in any case $0.75p$ (where $p$ is the

prior length), giving rise to an expectation of $2.4 \log n$ parallel partitioning steps ($n * 0.75^{2.4 \log n} = 1$).

Once the *pivot* is chosen, it is straightforward to decide in parallel which partition an element should go into. The problem of choosing the *pivot* can be easily solved in parallel in the CRCW model, as all processors working in a partition can simultaneously propose themselves as the new pivot by *Concurrently Writing* to an agreed location, and can subsequently obtain the actual selected *pivot* by *Concurrently Reading* that location. In view of the preceding discussion, it is desirable that the choice of which of the contending processors succeeds in writing the values be *non-systematic* so that it resembles a *random* choice rather than a likely *oracle* for *systematic* data.

The final partitioning problem concerns the moving of the partitioned data to suitable locations without meeting a sequential bottleneck. The solution proposed here, which is the primary augmentation of the algorithm, is that the movement of the elements is not performed at this stage, but sufficient information about the partitioning structure is retained to allow logarithmic time relocation of the elements in a subsequent phase.

Making the partition tree structure explicit provides this information. The algorithm exhibited here relocates the elements in unit time following two logical logarithmic phases. The first sizes the partitions, giving the*relative location* of each *pivot*. The second adds in the sizes of the left partitions at each level to give the implicit *absolute location* of each *pivot*. Each element is at some level a pivot, so it is now possible to read each element in parallel from its original location and relocate it directly to the specified *absolute location*.

## 2.2 Code

In this section, extracts from the code of a PASCAL procedure implementing the described algorithm are shown, and in the following section results from an instrumented version are given as empirical evidence of the number of parallel steps required. Note that as no machine implementing the CRCW PRAM model is available, the algorithm has been simulated in a sequential environment and the *Concurrent Read* and *Concurrent Write* steps are expressed in separate pseudo-parallel loops - the **par** in the code shown is precisely equivalent to a PASCAL **for** executed in *random* order. The **par** and **for** statements are all to be executed in parallel.

Associated with each *pivot* is its immediate *parent* , *which* partition of the *parent* it is in, and pointers to the two partitions of the *next* subdivision. The initialization of these variables

is not shown, but *which*is initialized to *right* and the remainder to *undef*. At each level each processor remembers its current *pivot* as *parent* and determines which new partition it is in. Then with a *Concurrent Write* to the corresponding*next* pointer of the *parent*, it nominates itself as the *pivot* for this new partition. The result of the ballot is discovered with a *Concurrent Read* and stored away as its new *pivot*.

```
repeat                    { ** PHASE A ** form binary tree by partitioning}
   finish:= true;

   par r:= n-1 downto 0 do                    {random synchronization}
      if pivot <> r then begin
         finish:= false;
         if elt[pivot] = elt[r]                    {fair<= >= comparison}
           then which:= boolean(random(2))
           else which:= elt[pivot] < elt[r];;
         parent:= pivot;
         proc[pivot].next[which]:= r;              {concurrent write}
      end;

   for p:= n-1 downto 0 do                   {selection synchronization}
   with proc[p] do
   if pivot <> p then
      pivot:= proc[pivot].next[which];             {concurrent read}

until finish;
```

**Figure 1.** *Procedure powersort (PHASE A)*

At the end of this process, the partition tree is explicitly represented with bidirectional pointers, *parent* and *next*. The number of parallel *fair* comparisons is clearly equal to the height of this tree, namely 2.4 log *n*.

In the next phase we first calculate (bottom up) the size of each partition which gives us the relative position of each element (*pivot*) in the partition of which it is the pivot. The variable *root* is intialized to the index of the first pivot chosen. The leaves have trivially defined counts, and calculation of further counts is possible as the defined count condition

propogates up the tree. The parallel overhead here is clearly also proportional to the height of the tree.

```
while proc[root].count = undef do     { ** PHASE B ** calc size of partition}
begin

    for p:= n-1 downto 0 do
    with proc[p] do
    begin
        if next[left] = undef then
            pos:= 0
        else
            if proc[next[left]].count <> undef then
              pos:= proc[next[left]].count;
        if pos <> undef then
          if (next[right] = undef) then
              count:= pos + 1
          else
              if (proc[next[right]].count <> undef) then
                count:= pos + proc[next[right]].count + 1
    end;

end;
```

**Figure 2.** *Procedure powersort (PHASE B).*

For the final pass, it is necessary that the *root* processor (owning the first *pivot*) has *pivot* and *parent* initialized to *undef*. and -1 respectively, indicating that it is the root and has the imaginary element -1 to its left (recall that we initialized the input partition as a right partition). The undefined elements are those which are processed and defined on each iteration (working top down). The defined absolute position of the parent plus one (since we count from zero and need to count the parent too) is added to the processed element's position relative to the parent. This phase also requires time proportional to the height of the tree.

The final move of each element from the input vector to the output vector is also shown. This is clearly achievable in constant parallel time.

```
    repeat                          { ** PHASE C ** calc absolute position}
        finish:= true;

        for p:= n-1 downto 0 do
        with proc[p] do
        if pivot = undef then
        begin
            pos:= pos + parent + 1;
            proc[next[left]].parent:= parent;
            proc[next[right]].parent:= pos;

            proc[next[left]].pivot:= undef;
            proc[next[right]].pivot:= undef;
            pivot:= p;
            finish:= false;
        end;
    until finish;

    for p:= n-1 downto 0 do sort[proc[p].pos]:= elt[p];
end;
```

**Figure 3.** *Procedure powersort (PHASE C).*

## 2.3 Results

The parallelized QuickSort procedure '*powersort*' was run on a series of distributions with results for Phase A that accorded well with the expected 2.4 log *n* parallel comparison time. Note that the algorithm and results for Phases B and C have been included only for completeness as the logarithmic complexity of parallel counting operations is well known. For this reason, no effort has been made to enforce simultaneous simulated parallel instances of the **for** loops, and the sequential simulation results in fortuitous premature propagation of counts and positions within a single iteration of the main loop, so on average half the degenerate subtrees are pruned away faster.

| **Size** | **Distribution** | **Phase A** | | **Phase B** | | **Phase C** | |
|---|---|---|---|---|---|---|---|
| *n* | | ***c*** | ***c*/log *n*** | *c* | *c*/log *n* | *c* | *c*/log *n* |
| 25 | sorted | **7** | **1.8** | 5 | 1.3 | 7 | 1.8 |
| 25 | reversed | **9** | **2.3** | 7 | 1.8 | 5 | 1.3 |
| 25 | normal | **10** | **2.5** | 5 | 1.3 | 7 | 1.8 |
| 25 | linear | **7** | **1.8** | 4 | 1.0 | 6 | 1.5 |
| 25 | quadratic | **8** | **2.0** | 6 | 1.5 | 5 | 1.3 |
| 25 | quartic | **8** | **2.0** | 4 | 1.0 | 7 | 1.8 |
| 100 | sorted | **12** | **2.0** | 8 | 1.3 | 8 | 1.3 |
| 100 | reversed | **12** | **2.0** | 8 | 1.3 | 11 | 1.8 |
| 100 | normal | **12** | **2.0** | 7 | 1.2 | 8 | 1.3 |
| 100 | linear | **15** | **2.5** | 7 | 1.2 | 10 | 1.7 |
| 100 | quadratic | **14** | **2.3** | 8 | 1.3 | 8 | 1.3 |
| 100 | quartic | **11** | **1.8** | 7 | 1.2 | 7 | 1.2 |
| 400 | sorted | **19** | **2.4** | 14 | 1.8 | 11 | 1.4 |
| 400 | reversed | **16** | **2.0** | 12 | 1.5 | 13 | 1.6 |
| 400 | normal | **16** | **2.0** | 9 | 1.1 | 10 | 1.3 |
| 400 | linear | **24** | **3.0** | 12 | 1.5 | 16 | 2.0 |
| 400 | quadratic | **20** | **2.5** | 9 | 1.1 | 13 | 1.6 |
| 400 | quartic | **16** | **2.0** | 9 | 1.1 | 10 | 1.3 |
| 1600 | sorted | **24** | **2.4** | 14 | 1.4 | 15 | 1.5 |
| 1600 | reversed | **21** | **2.1** | 13 | 1.3 | 15 | 1.5 |
| 1600 | normal | **20** | **2.0** | 12 | 1.2 | 12 | 1.2 |
| 1600 | linear | **25** | **2.5** | 16 | 1.6 | 15 | 1.5 |
| 1600 | quadratic | **27** | **2.7** | 14 | 1.4 | 17 | 1.7 |
| 1600 | quartic | **21** | **2.1** | 12 | 1.2 | 13 | 1.3 |

**Table 1.** Empirical measurements.

# 3 Conclusions

## 3.1 Parallelized QuickSort

The algorithm, analysis and results here all indicate that QuickSort can easily be executed in logarithmic expected parallel time. The simple CRCW algorithm of the first

phase is the key, and there is considerable scope for utilization or relocation of the elements in ways other than that used here in the subsequent phases.

To give an example of such an alternative, it is noted here that this algorithm is actually a biproduct of research into Parallel Logic Programming (Powers,1988). The input in this case is a list, and the output is a relinking of this list. Preliminary results indicate that the standard PROLOG QuickSort admits a logarithmic expected parallel logical inference complexity, that the unification involved could also be executed in logarithmic parallel time, and that heuristics can be added to a parallel theorem prover which will guide it into such a proof.

It was noted earlier that no CRCW parallel computer was available for the empirical verification of the algorithm. The use of a CRCW model as opposed to CREW is not itself a bar to achieving the expected efficiency, as they can be shown equivalent within a constant factor. Cole (1986) however translates his original CREW sorting algorithm to a more complex algorithm for an EREW model of the same theoretical complexity. The present author is of the opinion that in the case of this QuickSort algorithm it makes more sense to implement the required concurrent operations, and that EREW is in practice little easier to implement.

A more serious bar to achieving the logarithmic time is the additional overhead of communication cost between processors, or processors and memory. Assuming that it is necessary to use constant-degree processors, this introduces an $O(\log^2 n)$ overhead according to the conventional wisdom (Parberry,1989). By contrast Batcher's Bitonic Exchange requires only constant-degree communication and thus incurs only an $O(\log n)$ overhead, and with its smaller constant will always be ahead.

However, the $O(\log^2 n)$ overhead is not a lower bound, and using a pipelined $O(\log n * \log \log n)$ communication scheme, the present algorithm would be well ahead of Batcher's by $n = 10^6$ elements/processors.

## 3.2 Parallelized RadixSort

One major disadvantage of QuickSort in general is its terrible worst case performance. As discussed above, the problem can largely be avoided by randomization techniques, but we can never guarantee that the unlikely '*oracle*' behaviour will not occur.

In the above discussion, it was implicitly assumed that the $\log n$ characterization of communication costs included not only addressing but the messages themselves. Suppose the

keys are less than 2.4 log *n* bits long. Instead of choosing an element as pivot, an implicit pivot can be used by making the *left–right* decision on the basis of successive bits, which guarantees the tree will have a height equal to this bound.

Moreover, even in the general case, it is now no longer necessary to transmit entire keys around the network and RadixSort (Knuth, 1973) may be a viable alternative. Whereas QuickSort will in fact have a *max*(log *n*, *w*) overhead hidden in the communication costs, RadixSort will avoid this potential deterioration of the communication, and any further undesirable ramifications of that overloading, but will have a guaranteed O(*w*) parallel complexity.

Thus, whatever the key length, and assuming no compression of messages is performed, parallelized RadixSort will have the same expected overall time complexity as the parallelized QuickSort, but will not suffer an insidiously lurking potential worst case.

# References

AJTAI, M; KOMLÓS, J; & SZEMERÉDI (1983): **Sorting in *c* log *n* parallel steps.** *Combinatorica,* **3** #1 pp.1–19.

BATCHER, K E (1968):.**Sorting Networks and their Applications.** *Proc. AFIPS Spring Joint Comp. Conf.* pp.303–314.

COLE, R (1986): **Parallel Merge Sort.** *Proc. 27th Ann. IEEE Symp. FOCS*, pp.511–516.

FRANCIS, R S; & PANNAN, L J H (1989): **Parallelism in QuickSort.** *Proc. 12th Aust. Comp. Sci. Conf.* pp.353–361.

HOARE, C A R (1962): **QuickSort**, *Computer Journal* , **5** pp.10–15.

KNUTH, D E (1973): **Sorting and Searching.** *The Art of Computer Programming*, **3.** Addison–Wesley, Reading MA USA.

PARBERRY, I (1989):**Scholarly Review.** *Computing Reviews*, #8911 pp.578–580.

POWERS, D M W (1988): **Implementing Connection Graphs for Logic Programming.** *Proc. 9th European Meeting on Cybernetics, Vienna* pp957–964.