Investigating the Practical Value of Cole's $O(\log n)$ time CREW PRAM Merge Sort Algorithm

Lasse Natvig

Division of Computer Systems and Telematics, The Norwegian Institute of Technology,
The University of Trondheim, NORWAY
E-mail: `lasse@idt.unit.no`

1 Introduction

The work reported in this paper is motivated by a wish to learn more about the *practical* value of parallel algorithms taken from the field of *theoretical* computer science.

The literature on parallel complexity theory contains a large, and rapidly increasing amount of parallel algorithms. Most of these algorithms are described for the PRAM model, a computational model originally presented by Fortune and Wyllie [9, 15]. Today, most researchers consider this model as unrealistic with respect to realization with present technology. However, there is no clear consensus about whether we should continue to use the PRAM model, or turn to more realistic models. The proceedings from the Workshop on *"Opportunities and Constraints of Parallel Computing"* [14] contains many different and interesting opinions about these issues. Many researchers argue that the simplicity and generality of the PRAM model should be offered to the programmer, and that the programs should be made executable on more realistic machines by compile time mapping or run time emulation.

Now, let us *assume* that the PRAM model is an useful *programming model*—and you are supposed to use it on *practical problems*. Where should you start to read for finding the right algorithms? It is clear that a lot of the PRAM algorithms from theoretical computer science are very fast, at least for (very) large problem sizes.

This paper presents a comparison of Cole's $O(\log n)$ time parallel merge sort algorithm [4, 6] with Batcher's $O(\log^2 n)$ time bitonic sorting method [3]. Both algorithms are implemented[1] as complete synchronous MIMD programs on a CREW PRAM simulator that has been developed by the author [12, 11]. The implementations are used to test and measure the two algorithms on various problem sizes. The simplicity of the PRAM model and the synchronous behaviour of the programs made it possible to derive *exact* expressions for the time used by the algorithms. It was found that a straightforward *implementation of Batcher's bitonic sorting is faster than the implementation of Cole's parallel merge sort as long as the number of items to be sorted, n, is less than* 1.2×10^{21}, *i.e. more than 1 Giga Tera items!*

Sorting is used as an essential part of many parallel algorithms. In parallel complexity theory, algorithm descriptions often contain phrases like: "This step is done by sorting the n items in $O(\log n)$ time by using Cole's parallel merge sort (or ... by using the AKS sorting network [2])". This is completely legal for proving membership in some complexity class for a given problem. However, the result reported in this paper shows that a much simpler $O(\log^2 n)$ time sorting method will give faster algorithms for all *practical* problem sizes. It seems that little work has been reported that compares theoretical parallel algorithms with simpler algorithms based on *implementations*, *exact analysis*, and *finite* problems.

This paper describes Cole's parallel merge sort algorithm with emphasis on how it is based on merging in $O(1)$ time. We explain parts of the algorithm which not are given in Cole's description [6], but must be understood to be able to implement it. The comparison demonstrates the importance of simplicity (*i.e.*, small complexity constants), when we do not allow the problem size to be infinite.

[1] This is *probably* the *first* implementation of Cole's parallel merge sort.

2 Cole's Parallel Merge Sort

2.1 Cole's Algorithm—an Important Contribution

In 1983, a sorting network using only $O(\log n)$ time[2] and $O(n \log n)$ comparators was presented by *Ajtai, Komlós and Szemerédi* [2], often called the AKS-network. Based on that work, Leighton [10] obtained in 1984 the first parallel sorting algorithm which is *cost optimal.* Unfortunately, the AKS-network is generally considered to be of little practical value, due to its huge complexity constants.[3] However, the optimal *asymptotical* behaviour initiated a search for improvements for closing the gap between its theoretical importance and its practical use.

In 1986, *Richard Cole* presented a new parallel sorting algorithm called *parallel merge sort* [4]. Cole's algorithm is an important contribution, since it is the second $O(\log n)$ time cost optimal sorting algorithm—the first was the one implied by the AKS-network [10]. Further, it is claimed to have complexity constants which are much smaller than in the AKS-network. A revised version of the original paper was published in the *SIAM Journal on Computing* in 1988 [6]. Cole did also write a technical report about the algorithm [5], but has informally expressed that it is less good than the SIAM paper [7].

2.2 Cole's Algorithm—Main Principles

Cole's parallel merge sort assumes n *distinct items.* These are distributed one per leaf in a complete binary tree—it is assumed that *n is a power of 2.* The computation proceeds up the tree, level by level from the leaves to the root. Each internal node u merges the sorted sets computed at its children. The algorithm is based on the following $\log n$ merging procedure: ([6] page 771.)

> "The problem is to merge two sorted arrays of n items. We proceed in $\log n$ stages. In the ith stage, for each array, we take a sorted sample of 2^{i-1} items, comprising every $n/2^{i-1}$th item in the array. We compute the merge of these two samples".

Cole made two key observations. 1) *Merging in constant time:* Given the result of the merge from the $(i-1)$'th stage, the merge in the ith stage can be done in $O(1)$ time. 2) *The merges at the different levels of the tree can be pipelined:* This is possible since merged *samples* made at level l of the tree may be used to provide samples of the appropriate size for merging at the next level above l without losing the $O(1)$ time merging property.

2.3 Using Ranks to Obtain Constant Time Merging

2.3.1 The Task of Each Node u

Each node u should produce a list $L(u)$, which is the sorted list containing the items distributed initially at the leaves of the subtree with u as its top node (root). During the progress of the algorithm, each node u stores an array $\mathrm{Up}(u)$ which is a sorted subset of $L(u)$. Initially, all leaf nodes y have $\mathrm{Up}(y) = L(y)$. At the termination of the algorithm the top node of the whole tree, t, has $\mathrm{Up}(t) = L(t)$ which is the sorted sequence. In the progress of the algorithm, $\mathrm{Up}(u)$ will generally contain a sample of the items in $L(u)$.

Up(u) , SampleUp(v) , SampleUp(w) , and NewUp(u)

Before we proceed, we need some simple definitions. A node u is said to be *external* if $|\mathrm{Up}(u)| = |L(u)|$, otherwise it is an *inside* node. Initially, only the leaf nodes are external. In each stage of the algorithm, a new array, called $\mathrm{NewUp}(u)$ is made in every inside node u. $\mathrm{NewUp}(u)$ is formed in the following manner.

(Phase 1) Samples from the items in $\mathrm{Up}(v)$ and $\mathrm{Up}(w)$ are made and stored in the arrays $\mathrm{SampleUp}(v)$ and $\mathrm{SampleUp}(w)$, respectively.

---

[2] Throughout this paper, $\log n$ means $\log_2 n$.

[3] ([10] p. 346: " ... the AKS sorting circuit, which behaves terribly for "small" values of N (*e.g.*, $N < 10^{100}$).")

(Phase 2) NewUp(u) is made by merging SampleUp(v) and SampleUp(w) *with help* of the array Up(u).

For all nodes, the NewUp(u) array made in stage i is the Up(u) array at the start of stage $i+1$. At external nodes, Phase 2 is not performed—since these nodes have already produced their list $L(u)$.

2.3.2 Phase 1: Making Samples

The samples SampleUp(u) are made in parallel by all nodes u according to the following rules:

1. If u is an *inside* node, SampleUp(u) is the sorted array comprising of every fourth item in Up(u). The items are taken from the positions 1,5,9, etc.. If Up(u) contains less than four items, SampleUp(u) becomes empty.

2. At the first stage when u is external, SampleUp(u) is made in the same way as for inside nodes. At the second stage as external node, SampleUp(u) is every second item in Up(u), and in the third stage, SampleUp(u) is every item in Up(u). SampleUp(u) is always in sorted order.

The algorithm has $3 \log n$ stages
Initially, only the leaves are external nodes. In the third stage as external, a node v has used all the items in $L(v)$ to produce SampleUp(v) in Phase 1, and its parent node u has merged SampleUp(v) and SampleUp(w) into Up(u) in Phase 2. Node u has now received all the items in $L(u)$, and the child nodes v and w have finished their tasks and may "terminate". In the next stage, the parent node u will have Up(u) $= L(u)$, and this is the first stage as external for node u.

We see that the level with external nodes moves up one level every third stage. After $3 \log n$ stages the top node t becomes external, it has Up(t) $= L(t)$, and the algorithm may stop (Lemma 2 in [6]).

The size of Up(u) is doubled in each stage
The size of the samples made by the external nodes double in each stage, because the sample rate is halved in each stage. As a consequence, the size of Up(u) in the inside nodes one level higher does also double in each stage. At this level, the sample rate is fixed (4) as long as the nodes are inside nodes. But, since $|\text{Up}(u)|$ doubles in each stage, the samples made at this level will also double in each stage. By induction, we see intuitively that the size of Up(u) doubles in each stage for each inside node with $|\text{Up}(u)| > 0$.

2.3.3 Phase 2: Merging In $O(1)$ Time

The most complicated part of Cole's algorithm is the merging of SampleUp(v) and SampleUp(w) into NewUp(u) in $O(1)$ time. It constitutes the major part of the algorithm description in [6], and about 90% of the code in the implementation. The merging is described in the following. Proofs and some more details are found in [6].

Using ranks to compute NewUp(u)
The merging is based on maintaining a set of *ranks*. Assume that A and B are sorted arrays. Informally we define A to be ranked in B (denoted $A \rightarrow B$) if we for each item e in A know how many items in B that are smaller or equal to e. Thus, knowing $A \rightarrow B$ means that we know where to insert an item from A in B so as to keep B sorted.

At the start of each stage we know Up(u), SampleUp(v) and SampleUp(w). In addition, we do the following assumption;

Assumption 2.1
At the start of each stage, for each inside node u with child nodes v and w we know the ranks Up(u) $\rightarrow$ SampleUp(v), and Up(u) $\rightarrow$ SampleUp(w).

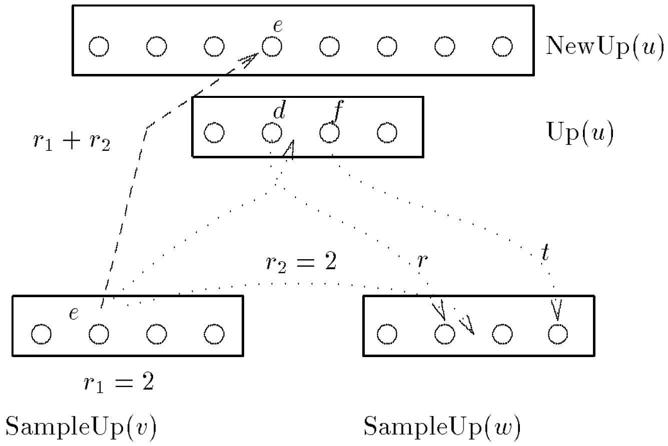

Figure 1: The use of *ranks* in Cole's $O(1)$ time merging procedure.

The making of NewUp(u) may be split into two main steps; the merging of the samples from its two child nodes, and the maintaining of Assumption 2.1. The calculation of the ranks constitutes a large fraction of the total time consumption of Cole's algorithm. However, as we will see, they are crucial for making it possible to merge in constant time.

2.3.4 Phase 2, Step 1: Merging in Constant Time

We want to merge SampleUp(v) and SampleUp(w) into NewUp(u), see Figure 1. This is easy if we for each item in SampleUp(v) and SampleUp(w) know its position in NewUp(u). The merging is then done by simply writing each item to its right position.

Consider an arbitrary item e in SampleUp(v), see Figure 1. We want to compute the position of e in NewUp(u), *i.e.*, the rank of e in NewUp(u), denoted R(e,NewUp(u)). Since NewUp(u) is made by merging SampleUp(v) and SampleUp(w) we have

$$\mathrm{R}(e, \mathrm{NewUp}(u)) = \mathrm{R}(e, \mathrm{SampleUp}(v)) + \mathrm{R}(e, \mathrm{SampleUp}(w)) \tag{1}$$

The problem of merging has been reduced to computation of R(e,SampleUp(v)) and R(e,SampleUp(w)). Since e is from SampleUp(v), R(e,SampleUp(v)) is just the position of e in SampleUp(v). Computing the rank of e in SampleUp(w) is much more complicated. Similarly, for items e in SampleUp(w) it is easy to compute R(e,SampleUp(w)), but difficult to compute R(e,SampleUp(v)).

The computation of the ranks SampleUp(v) $\rightarrow$ SampleUp(w) and SampleUp(w) $\rightarrow$ SampleUp(v) is termed computation of *crossranks.*

2.3.5 Computing Crossranks

Up(u) and Assumption 2.1 may be to a great help in computing the crossranks. Consider Figure 1 and the item e from SampleUp(v). We want to find the position of e in SampleUp(w) if it had to be inserted according to sorted order in that array.

Note that it is only the case e is from SampleUp(v) that is described in this and the following sections (2.3.6 and 2.3.7). The other case, e is from SampleUp(w), is handled in a completely symmetric manner.

First, for each item e in SampleUp(v) we compute its rank in Up(u). This computation is done for all the items in SampleUp(v) in parallel, and is described as *Substep 1* in Section 2.3.6 below. R(e,Up(u)) gives us the items d and f in Up(u) which would *straddle*[4] e if e had to be inserted

[4] Let x, y and z be three items, with $x < z$. We say that x and z *straddle* y if $x \leq y$ and $y < z$, *i.e.*, $y \in [x, z\rangle$.

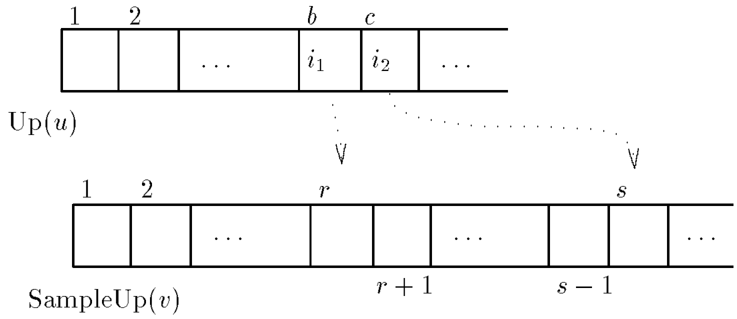

Figure 2: Substep 1: Computing SampleUp(v) $\rightarrow$ Up(u) by using Up(u) $\rightarrow$ SampleUp(v)

in Up(u). Further, Assumption 2.1 gives R(d,SampleUp(w)) and R(f,SampleUp(w))—the right positions for inserting d and f in SampleUp(w). These ranks are called r and t in Figure 1. Informally, since $e \in [d, f\rangle$ the right position for e in SampleUp(w) is bounded by the positions (ranks) r and t. It can be shown that r and t always specifies a range of maximum 4 positions, and the right position, R(e,SampleUp(w)), may thus be found in constant time. This is explained in more detail as *Substep 2* in Section 2.3.7.

2.3.6 Substep 1: Computing SampleUp(v) $\rightarrow$ Up(u)

SampleUp(v) $\rightarrow$ Up(u) is computed by using Up(u) $\rightarrow$ SampleUp(v). Consider an arbitrary item i_1 in Up(u), see Figure 2. The range $[i_1, i_2\rangle$ is defined to be the interval *induced* by item i_1. We want to find the set of items in SampleUp(v) which are contained in $[i_1, i_2\rangle$, denoted $I(i_1)$.

The items in $I(i_1)$ have rank in Up(u) equal to b, where b is the position of i_1 in Up(u), and the array positions are numbered starting with 1 as shown in the figure. Once $I(i_1)$ have been found a processor associated with item i_1 may assign the rank b to the items in the set. Simultaneously, a processor associated with i_2 should assign the rank c to $I(i_2)$.

Computing $I(i_1)$
The precise calculations necessary for finding $I(i_1)$ are not explained in Cole's SIAM paper [6]. However, we must understand how it can be done to be able to implement the algorithm. We want to find all items $\alpha \in$ SampleUp(v) with R(α, Up(u)) $= b$. These items must satisfy

$$(\alpha \geq i_1) \wedge (\alpha < i_2) \tag{2}$$

Let item(j) denote the item stored in position j of SampleUp(v). Assumption 2.1 gives R(i_1,SampleUp(v)) $= r$ and R(i_2,SampleUp(v)) $= s$, which implies

$$(\text{item}(r) \leq i_1) \wedge (\text{item}(s) \leq i_2) \tag{3}$$

We have the following observation

Observation 2.1
(See Figure 2.) If there exist items β in SampleUp(v) between (and not including) position r and s they must have R(β,Up(u)) $= b$.

Proof: Let item(r+1) denote an item which is to the right of item(r) and to the left of item(s), see Figure 2. The definition of the rank r implies that $i_1 <$ item(r+1). Hence, R(item(r+1), Up(u)) $\geq b$. Let item($s-1$) denote an item which is to the left of item(s) and to the right of item(r). The requirement of distinct items implies item(s) $>$ item($s-1$), and the definition of the rank s implies item(s) $\leq i_2$. Distinct items then gives item($s-1$) $< i_2$, so we have R(item($s-1$),Up(u)) $< c$.

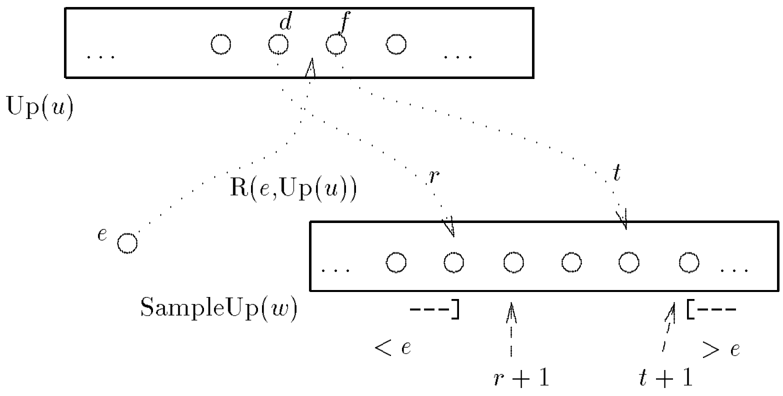

Figure 3: Substep 2: Computing SampleUp(v) $\rightarrow$ SampleUp(w) by using SampleUp(v) $\rightarrow$ Up(u)

Since $b = c - 1$, all items in SampleUp(v) in positions $r+1, r+2, \ldots, s-1$ have rank b in Up(u). $\square$

The items in positions r and s must be given special treatment, we have:

$$\begin{aligned}
&\text{item}(r) = i_1 \Rightarrow \text{R}(\text{item}(r), \text{Up}(u)) = b \\
&\text{item}(r) < i_1 \Rightarrow \text{R}(\text{item}(r), \text{Up}(u)) < b \\
&\text{item}(s) = i_2 \Rightarrow \text{R}(\text{item}(s), \text{Up}(u)) = c \\
&\text{item}(s) < i_2 \Rightarrow \text{R}(\text{item}(s), \text{Up}(u)) < c \Rightarrow \text{R}(\text{item}(s), \text{Up}(u)) = b
\end{aligned}$$

To conclude, $I(i_1)$ is found by comparing item(r) with i_1 and item(s) with i_2. A processor associated with item i_1 may do these simple calculations and assign the rank b to all items in $I(i_1)$ in *constant* time. This is possible because it can be proved that $I(y)$ for any item y in Up(u) will contain at most three items [6].

2.3.7 Substep 2: Computing SampleUp(v) $\rightarrow$ SampleUp(w)

See Figure 3. As described in Section 2.3.5, knowing R(e,Up(u)), which was computed in Substep1, gives the straddling items d and f in Up(u). Further, Assumption 2.1 gives the ranks r and t. We want the *exact* position of e if it had to be inserted in SampleUp(w). *Which* items in SampleUp(w) must be compared with e? The question is answered by the following observation.

Observation 2.2
All items in SampleUp(w) to the left of, and including, position r are smaller than item e. All items in SampleUp(w) to the right of position t are smaller than item e. *(Proof: See [13]).*

How many items must be compared with e? Observation 2.2 tells us that e must be compared with the items from SampleUp(w) with positions in the range $[r+1, t]$. Since Up(u) is a 3-cover for SampleUp(w) we know that $[d, f\rangle$ contains at most three items in SampleUp(w). But, the set of items in SampleUp(w) contained in $[d, f\rangle$ is not *necessarily* the same as the items with positions in the range $[r+1, t]$. However, we can prove the following observation by using the 3-cover property of $[d, f\rangle$.

Observation 2.3
(See Figure 3). Item e must be compared with *at most three items* from SampleUp(w) starting with item($r+1$), going to the right, but not beyond item(t). In other words, the items item($r+i$) for $1 \le i \le \min(3, t-r)$.

Proof: First, let us consider which items in SampleUp(w) may be contained in $[d, f\rangle$. We have four possible cases;

$$\begin{array}{ll} 1) & d = \text{item}(r) \Rightarrow \text{item}(r) \in [d, f\rangle \\ 2) & d \neq \text{item}(r) \Rightarrow d > \text{item}(r) \Rightarrow \text{item}(r) \notin [d, f\rangle \\ 3) & f = \text{item}(t) \Rightarrow \text{item}(t) \notin [d, f\rangle \\ 4) & f \neq \text{item}(t) \Rightarrow f > \text{item}(t) \Rightarrow \text{item}(t) \in [d, f\rangle \end{array} \tag{4}$$

which imply four cases for the "placement" of $[d, f\rangle$ in SampleUp(w);

$$\begin{array}{ll} i) & [\text{item}(r), \text{item}(t-1)] \\ ii) & [\text{item}(r), \text{item}(t)] \\ iii) & [\text{item}(r+1), \text{item}(t-1)] \\ iv) & [\text{item}(r+1), \text{item}(t)] \end{array}$$

Case *i)*: The 3-cover property implies that $|[\text{item}(r), \text{item}(t-1)]| \leq 3$. By deleting one and adding one item we get $|[\text{item}(r+1), \text{item}(t)]| \leq 3$. Case *ii)*: The 3-cover property gives $|[\text{item}(r), \text{item}(t)]| \leq 3$, which implies $|[\text{item}(r+1), \text{item}(t)]| \leq 2$.[5] Case *iii)*: $|[\text{item}(r+1), \text{item}(t-1)]| \leq 3$. At first sight, one may think that this case makes it necessary to compare e with 4 items in SampleUp(w). However, this case does only occur when item$(t) = f$ (see case 3 in equation 4), and since $e < f$ we know that $e <$ item(t), so there is no need to consider item(t). We need only consider items in $[\text{item}(r+1), \text{item}(t-1)]$ which contains at most 3 items. Case *iv)*: The 3-cover property implies $|[\text{item}(r+1), \text{item}(t)]| \leq 3$. Thus we have shown that e must be compared with at most three items for all possible cases. $\square$

Since these (at most) three items in SampleUp(w) are sorted, two comparisons are sufficient to locate the correct position of e. Therefore, R$(e, \text{SampleUp}(w))$ can be computed in $O(1)$ time.

When Substep 2 have been done (in parallel) for each item e in SampleUp(v) and SampleUp(w), every item knows its position (rank) in NewUp(u) by Equation 1, and may write itself to the correct position of that array.

2.3.8 Phase 2, Step 2 — Maintaining Ranks

We assumed above that the ranks Up$(u) \rightarrow$ SampleUp(v) and Up$(u) \rightarrow$ SampleUp(w) were available. We must therefore compute NewUp$(u) \rightarrow$ SampleUp(v) and NewUp$(u) \rightarrow$ SampleUp(w) at the end of each stage, so that the assumption is valid at the start of the next stage. Doing this in $O(1)$ time is about just as complicated as the merging outlined above. (See [6, 13].)

2.4 Implementation and Analysis

CREW PRAM programming

Cole's succinct description of the algorithm [6] is at a relatively high level giving the programmer freedom to choose between the SIMD or MIMD [8] implementation paradigms. The algorithm have been programmed in a *synchronous MIMD programming style*, as proposed for the PRAM model by Wyllie [15]. This paper gives only a brief description of the implementation, providing a crude base for discussing its time requirement. Figure 4 outlines the main program in a notation called "parallel pseudo pascal" (PPP) [12]. This notation is inspired by *parallel pidgin algol* as defined by Wyllie in [15]—with modernizations from the pseudo language notation ("Super Pascal") used by Aho, Hopcroft and Ullmann in [1].

When the algorithm starts, one single CREW PRAM processor is running, and the problem instance and its size, n, are stored in the global memory. Statement (1–3) are executed by this single processor.

[5] In this case Figure 3 is not correct; at most one item may exist between item(r) and item(t).

```
     CREW PRAM procedure ColeMergeSort
     begin
(1)      Compute the processor requirement, NoOfProcs;
(2)      Allocate working areas;
(3)      Push addresses of working areas and other facts on the stack;
(4)      assign NoOfProcs processors, name them P;
(5)      for each processor in P do begin
(6)          Read facts from the stack;
(7)          InitiateProcessors;
(8)          for Stage := 1 to 3 log n do begin
(9)              ComputeWhoIsWho;
(10)             CopyNewUpToUp;
(11)             MakeSamples;
(12)             MergeWithHelp;
             end;
         end;
     end;
```

Figure 4: Main program of Cole's parallel merge sort expressed in parallel pseudo pascal.

A sliding pyramid of processors

Cole describes that one should have one processor standing by each item in the Up, NewUp, and SampleUp arrays. Since the size of these arrays, for each node u, change from stage to stage—the processors must be *dynamically allocated* to the nodes (*i.e.* the array elements in Up(u), NewUp(u) and SampleUp(u)) as the computation proceeds from the leaves to the root of the tree. The maximum *processor requirement* is given by the maximum size of the Up(u), NewUp(u) and SampleUp(u) arrays for all nodes u during the computation. We have [6]:

$$\mathit{NoOfProcs} = \sum_u |\mathrm{Up}(u)| + \sum_u |\mathrm{NewUp}(u)| + \sum_u |\mathrm{SampleUp}(u)| \tag{5}$$

where $\sum_u |\mathrm{Up}(u)| \leq n + n/2 + n/16 + n/128 + \ldots = 11n/7$ and $\sum_u |\mathrm{NewUp}(u)| = \sum_u |\mathrm{SampleUp}(u)| \leq n + n/8 + n/64 + n/512 + \ldots = 8n/7$ which is *slightly less than* $4n$. The $\leq$ means that the total number of array elements is *bounded above* by the given sum.

Consider the sum given for the Up arrays. There are n processors (array elements) at the lowest active level, a maximum of $n/2$ processors at the next level above, and so on. This may be viewed as a *pyramid of processors*. Each time the lowest *active* level moves one level up—the pyramid of processors follows so that we still have n processors at the lowest active level. Similarly, the NewUp and SampleUp processors may be viewed as a "sliding pyramid of processors".

Analysis

For a given n, the *exact* calculation of *NoOfProcs* is done by a loop with $\log_8 n$ iterations. The time used by this sequential startup code is shown in Table 1. $t(i, n)$ denotes the time used on *one single* execution of statement i of the discussed program when the problem size is n. $t(j..k, n)$ is a short hand notation for $\sum_{i=j}^{i=k} t(i, n)$.

A general procedure for *processor allocation* is implemented in the CREW PRAM simulator by a real CREW PRAM algorithm which is able to allocate k processors in $\log k$ time utilizing the (standard PRAM [9, 15]) `fork` instruction in a binary tree structured "chain reaction". Thus, the time used for processor allocation (statement (4)) is as given in Table 1 and Equation 5.

Statement (3) and (6) illustrate that a dedicated area (a stack) in the global memory is used to pass variables (such as the problem size) to the processors allocated in statement (4), and

Table 1: Time consumption for the statements in the implementation of *ColeMergeSort*.

$t(1,n) = 34 + 8\lfloor\log_8(n/2)\rfloor + 8\lfloor\log_8 n\rfloor$	$t(7,n) = 224 + 36\lfloor\log_8(n/2)\rfloor + 72\lfloor\log_8 n\rfloor$
$t(2..3,n) = 83$	$t(8..10,n) = 159$
$t(4,n) = 42 + 23\lfloor\log \mathit{NoOfProcs}\rfloor$	$t(11,n) = 48$
$t(5..6,n) = 13$	$t(12,n) = 781$

activated in statement (5). Due to the *concurrent read* property of the CREW PRAM, statement (6) is easily executed in O(1) time.

InitiateProcessors computes the *static* part of the processor allocation information. Examples are what level (in the "pyramid" as discussed above) the processor is assigned to, and the local processor no within that level. It have been implemented by two "divide by 8 loops" resulting in the time consumption shown in the table.

The $3\log n$ stages each consists of four main computation steps. *ComputeWhoIsWho* performs the *dynamic* part of the processor allocation. Since both the active levels of the tree, and the size of the various arrays change from stage to stage, information such as the node no, and item no in the array for that node, must be recomputed for each processor at the start of each stage. The necessary computations are easily performed in $O(1)$ time.

CopyNewUpToUp is only a simple procedure that makes the NewUp arrays made in the previous stage to the Up arrays of the current stage. *MakeSamples* produce the array SampleUp(u) from Up(u) for all active nodes in the tree. It is a relatively straightforward task.

The $O(1)$ time merging performed by *MergeWithHelp* constitutes the major part of the algorithm. Of the time used by *MergeWithHelp* (781 time units), about 40% is needed to compute the *crossranks*, and nearly 43% is used to *maintain ranks*.

The time used to perform a *Stage* (9–12) is somewhat shorter for the six first stages than the numbers listed in Table 1. This is because some parts of the algorithm do not need to be performed when the sequences are very short. However, for all stages after the six'th, the time used is as given by the constants in the table. Stages 1–6 takes a total of 2525 time units. The total time used by *ColeMergeSort* on n distinct items, $n = 2^m$ may be expressed as

$$T(\mathit{ColeMergeSort}, n) = t(1..7,n) + 2525 + t(8..12,n) \times 3((\log n) - 2)$$

The reader is referred to [13] for further details about the implementation.

3 Comparison With a Much Simpler Algorithm

3.1 Bitonic Sorting on a CREW PRAM

Batcher's bitonic sorting network [3] for sorting of $n = 2^m$ items consists of $\frac{1}{2}m(m+1)$ columns each containing $n/2$ comparators (comparison elements). A natural emulation on a CREW PRAM is to use $n/2$ processors which are dynamically allocated to the one active column of comparators as it moves from the input side to the output side through the network.[6] The global memory is used to store the sequence when the computation proceeds from one step (*i.e.* comparator column) to the next. The main program and its time requirement are shown in Figure 5. *EmulationNetwork* is a procedure which computes the addresses in the global memory corresponding to the two input lines for that comparator in the current *Stage* and *Step*. *ActAsComparator* calculates which (of the two possible) comparator functions that should be done by the processor (comparator) in the current *Stage* and *Step*, performs the function, and writes the two outputs to the global memory.

[6] The possibility of sorting several sequences simultaneously in the network by use of pipelining is sacrificed by this method. This is not relevant in this comparison, since Cole's algorithm do not have a similar possibility.

```
    CREW PRAM procedure BitonicSort
    begin
(1)     assign n/2 processors, name them P;
(2)     for each processor in P do begin
(3)         Initiate processors;
(4)         for Stage := 1 to log n do
(5)             for Step := 1 to Stage do begin
(6)                 EmulateNetwork;
(7)                 ActAsComparator;
                end;
        end;
    end;
```

$t(1, n) = 42 + 23\lfloor\log(n/2)\rfloor$
$t(2..3, n) = 38$
$t(4, n) = 10$
$t(5..7, n) = 84$

Figure 5: Main program and time consumption of bitonic sorting emulated on a CREW PRAM.

Table 2: Performance data for $n = 256$. Time and cost in this table are given in *kilo* CREW PRAM (unit-time) instructions, reads and writes in *kilo* locations.

Algorithm	time	#processors	cost	# reads	# writes
ColeMergeSort	21.2	986	20863.8	104.6	65.2
BitonicSort	3.4	128	428.2	9.9	9.4

Both procedures are easily done in $O(1)$ time. The total time requirement becomes :

$$T(BitonicSort, n) = t(1..3, n) + t(4, n) \times \log n + t(5..7, n) \times \frac{1}{2} \log n(\log n + 1)$$

3.2 Comparison

Table 2 shows measured performance data obtained by running the two algorithms for $n = 256$. The number of reads and writes from/to the global memory is also shown. These and other test runs have been used to check the analytical model. Table 3 shows time usage and processor requirement for the two algorithms for $n = 64k (k = 2^{10})$, $n = 256k$, for the last value of n making $T(BitonicSort, n) < T(ColeMergeSort, n)$, and for the first value of n making *ColeMergeSort* to a faster algorithm. The exact numbers calculated are *not* especially important. What is *important, is that the method of algorithm investigation described in this paper makes it possible*

Table 3: Calculated performance data for the two CREW PRAM implementations.

Algorithm	n	time	#processors
ColeMergeSort	65536 (64k)	4.5×10^4	2.5×10^5
BitonicSort	65536 (64k)	1.2×10^4	3.3×10^4
ColeMergeSort	262144 (256k)	5.2×10^4	1.0×10^6
BitonicSort	262144 (256k)	1.5×10^4	1.3×10^5
ColeMergeSort	2^{69}	205972	2.3×10^{21}
BitonicSort	2^{69}	205194	3.0×10^{20}
ColeMergeSort	2^{70}	208958	4.6×10^{21}
BitonicSort	2^{70}	211107	5.9×10^{20}

to do such exact calculations. Also, the huge value of n gives room for a lot of improvements to Cole's algorithm before it beats bitonic sorting for practical problem sizes. There are also good possibilities to improve the implementation of bitonic sorting.

In fact, Cole's algorithm is even less practical than depicted by the described comparison of execution time; it requires about 8 times as many processors than bitonic sorting, and it has a far more extensive use of the global memory.

4 Concluding Remarks

The way of investigating PRAM algorithms exemplified by this paper might contribute to lessen the gap between theory and practice in parallel computing. Reducing this gap was recently emphasized as a very important research area by several prominent researchers at the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing [14].

In practice, we do not have infinite problem sizes. This makes asymptotical analysis less useful for comparing the relative performance of algorithms on practical problems. On finite problems, the size of the various complexity constants becomes more important. In my view, the best way to find representative complexity constants is to implement the algorithms. Measurements, and detailed analysis may then be used to provide a more accurate comparison. The comparison reported in this paper was based on exact analysis of implementations and tested algorithms. It was demonstrated how the *simplicity* of $O(\log^2 n)$ time bitonic sorting *makes it faster in practice* than Cole's $O(\log n)$ time algorithm.

Acknowledgements

This work has been done during my Ph.D. studies at The Norwegian Institute of Technology (NTH) in Trondheim. I wish to thank my supervisor Professor Arne Halaas for constructive criticisms and for continuing encouragement. The work has been financially supported by a scholarship from The Royal Norwegian Council for Scientific and Industrial Research (NTNF).

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. *Combinatorica*, 3(1):1–19, 1983.

[3] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[4] Richard Cole. Parallel Merge Sort. In *Proceedings of 27th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 511–516, 1986.

[5] Richard Cole. Parallel Merge Sort. Technical Report 278, Computer Science Department, New York University, March 1987.

[6] Richard Cole. Parallel Merge Sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[7] Richard Cole. New York University. Private communication, 16 March and 27 April 1990, 1990.

[8] Michael J. Flynn. Very High-Speed Computing Systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.

[9] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10'th ACM Symposium on Theory of Computing (STOC)*, pages 114–118. ACM, NewYork, May 1978.

[10] Tom Leighton. Tight Bounds on the Complexity of Parallel Sorting. In *Proceedings of the 16th Annual ACM Symposium on Theory Of Computing (May)*, pages 71–80. ACM, New York, 1984.

[11] Lasse Natvig. Crew pram simulator—users's guide. Technical Report 39/89, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, The University of Trondheim, Norway, December 1989.

[12] Lasse Natvig. The CREW PRAM Model—Simulation and Programming. Technical Report 38/89, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, The University of Trondheim, Norway, December 1989.

[13] Lasse Natvig. Cole's Parallel Merge Sort Implemented on a CREW PRAM Simulator. Technical Report 3/90, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, The University of Trondheim, Norway, 1990. Still in preparation.

[14] Jorge L. C. Sanz, editor. *Opportunities and Constraints of Parallel Computing.* Springer-Verlag, London, 1989. Papers presented at the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing, San Jose, California, December 1988. (ARC = IBM Almaden Research Center, NSF = National Science Foundation).

[15] J. C. Wyllie. *The Complexity of Parallel Computations.* PhD thesis, Dept. of Computer Science, Cornell University, 1979.