Logarithmic Time Cost Optimal Parallel Sorting is *Not Yet* Fast in Practice!

Lasse Natvig
Division of Computer Systems and Telematics

The Norwegian Institute of Technology, The University of Trondheim
N–7034 Trondheim, NORWAY
(E-mail: lasse@idt.unit.no)

November 29, 1996

Abstract
When looking for new and faster parallel sorting algorithms for use in massively parallel systems it is tempting to investigate promising alternatives from the large body of research done on parallel sorting in the field of theoretical computer science. Such "theoretical" algorithms are mainly described for the PRAM (Parallel Random Access Machine) model of computation [13, 26]. This paper shows how this kind of investigation can be done on a simple but versatile environment for *programming and measuring* of PRAM algorithms [18, 19]. The practical value of Cole's Parallel Merge Sort algorithm [10,11] have been investigated by comparing it with Batcher's bitonic sorting [5]. The $O(\log n)$ time consumption of Cole's algorithm implies that it must be faster than bitonic sorting which is $O(\log^2 n)$ time—if n is large enough. However, we have found that bitonic sorting is faster as long as n is less than 1.2×10^{21}, *i.e.* more than 1 Giga Tera items!. Consequently, Cole's logarithmic time algorithm is *not fast in practice*.

1 Introduction and Motivation

The work reported in this paper is an attempt to lessen the gap between theory and practice within the field of parallel computing. Within theoretical computer science, parallel algorithms are mainly compared by using asymptotical analysis (O-notation). This paper gives an example on how the analysis of *implemented* algorithms on *finite* problems provides new and more practically oriented results than those traditionally obtained by asymptotical analysis.

Parallel Complexity Theory—A Rich Source for Parallel Algorithms
In the past years there has been an increased interest in parallel algorithms. In the field of parallel complexity theory the so called *Nick's Class* (NC) has been given a lot of attention. Problems belonging to this complexity class may be solved by algorithms with polylogarithmic (i.e. of $O(\log^k(n))$ where k is a constant and n is the problem size) running time and a polynomial requirement for processors. A lot of parallel algorithms have recently been presented to prove membership in this class for various problems. This kind of algorithms are frequently denoted NC-algorithms. Although the main motivation for these new algorithms often is to show that a given problem is in NC, the algorithms may also be taken as proposals for parallel algorithms that are "fast in practice".

The Gap Between Theory and Practice
Unfortunately, it may be very hard to assess the practical value of NC-algorithms.

Asymptotical analysis Order–notation is a very useful tool when the aim is to prove membership of complexity classes or if asymptotic behavior for some other reason is "detailed enough". It makes it possible to describe and analyze algorithms at a very high level. However, it also makes it possible to hide (willingly or unwillingly) a lot of details which cannot be omitted when algorithms are compared for "realistic" problems of large but *limited* size.

Unrealistic machine model Few of these theoretical algorithms are described as implementations on real machines. Their descriptions are based on various computational models. A computational model is the same as an *abstract machine*. One such model is the CREW PRAM.

Details are swept under the rug Most NC-algorithms originating from parallel complexity theory are presented on a very high level and in a compact manner. One reason is probably that parallel complexity theory is a field that to a large extent overlaps with mathematics—where the elegance and advantages of compact descriptions are highly appreciated. Another reason may be found in the call for papers for the 30'th FOCS Symposium; A strict limit of 6 pages was enforced on the submitted papers. Considering the complexity of most of these algorithms, a compact description is therefore a necessity.

Traditional Use of the CREW PRAM Model

The certainly most used [16] model for expressing parallel algorithms in theoretical computer science is the P-RAM model proposed by Fortune and Wyllie in 1978 [13]. Its simplicity and generality makes it possible to concentrate on the algorithm without being distracted by the obstacles caused by describing it for a more specific (and realistic) machine model. Its synchronous operation makes it easy to describe and analyze programs for the model. This is exactly what is needed in parallel complexity theory, where the main focus is on the parallelism inherent in a problem.

Implementing On an Abstract Model is Worthwhile

Implementing algorithms on the PRAM model may be regarded as a paradox. One of the reasons for using such abstract machines has traditionally been a wish to avoid implementation details. However, it may be a viable step in an attempt to lessen the gap between theory and practice. The following is achieved by implementing a "theoretical" parallel algorithm on a CREW PRAM model:

- *A deeper understanding.* Making an implementation enforces a detailed study of all aspects of an algorithm.
- *Confidence in your understanding.* Verification of large parallel programs is very difficult. In practice, the best way of getting confident with one's own understanding of a complicated algorithm is to make an implementation that works. Of course, testing can not give a proof, but elaborate and systematic testing may give a larger degree of confidence than program verification (which also may contain errors).
- *A good help in mastering the complexity involved in implementing a complicated parallel algorithm on a real machine.* A CREW PRAM implementation may be a good starting point for implementing the same algorithm on a more realistic machine model. This is especially important for complicated algorithms. Going directly from an abstract mathematical description to an implementation on a real machine will often be a too large step. The existence of a CREW PRAM implementation may reduce this to two smaller steps.
- *Denying the practical value of an algorithm.* How can an unrealistic machine model be used to answer questions about the practical value of algorithms? It is important here to differentiate between negative and positive answers. If a parallel algorithm $\mathcal{A}$ shows up to be inferior as compared with sequential and/or parallel algorithms for (more) realistic models, the use of an *unrealistic* parallel machine model for executing $\mathcal{A}$ will only *strengthen* the conclusion. On the other side, a parallel algorithm cannot be classified as a better alternative for practical use if it is based on a less realistic model.

These advantages of implementing parallel algorithms on the CREW PRAM model are independent of whether a parallel computer that resembles the CREW PRAM model ever will be built.

2 The CREW PRAM Model — Programming and Simulation

This section gives a brief introduction to the CREW PRAM model, and how algorithms may be programmed, executed and measured on a CREW PRAM simulator.

The Original CREW PRAM

Unfortunately, there is a lot of different opinions on how the (CREW) PRAM should be programmed. My work is originally inspired by James Wyllie's well-written Ph.D. thesis *The Complexity of Parallel Computations* [26]. The thesis gives a high-level and succinct description of how a PRAM may be programmed.

Background The *P-RAM (parallel random access machine)* was first presented by Steven Fortune and James Wyllie [13]. It was further elaborated in Wyllie's Ph.D. thesis [26]. The P-RAM is based on random access machines (RAMs) operating in parallel and sharing a common memory. Thus it is in a sense the model in the world of parallel computations that corresponds to the RAM (Random Access Machine) model, which certainly is the prevailing model for sequential computations.

Today, the mostly used name on the original P-RAM model is probably CREW PRAM. CREW is an abbreviation for the very central concurrent read exclusive write property. (Several processors may at the same time step read the same variable (location) in global memory, but they may not write to the same global variable simultaneously.)

Main Properties A CREW PRAM is a very simple and general model of a parallel computer. It has an unbounded number of equal processors. Each has an unbounded local memory, an accumulator, program counter, and a flag indicating whether it is running or not. A CREW PRAM has an unbounded global memory shared by all processors. An unbounded number of processors may write into global memory as long as they write to different locations. An unbounded number of processors may read any location at any time. All processors execute the same program. At each *global* time step in the computation, each running processor executes the instruction given by its own *local* program counter in one unit of time—and the model is therefore best classified as a synchronous MIMD machine. More details may be found in one of [13, 19, 26].

Parallel Programming May be Easy! The algorithms are implemented as PIL programs and executed on a CREW PRAM simulator prototype. PIL is a very simple extension of the high-level, object oriented programming language SIMULA [8, 22]. PIL includes features for processor allocation, activation and synchronization, for interaction with the simulator etc. The powerful and flexible standard SIMULA source code level debugger may be used on the PIL programs [15]. The simplicity and generality of the PRAM model, combined with the high-level language and the debugger—have made the development of *synchronous parallel MIMD programs* to a surprisingly easy task. The system has also been used in connection with teaching of parallel algorithms.

About the Simulator and Time Modelling The CREW PRAM simulator has been developed in SIMULA with good help of the DEMOS discrete event simulation package [7]. It runs on SUN workstations under the UNIX operating system.

Time is measured in number of CREW PRAM unit time instructions—denoted CREW PRAM time units or simply time units. Each processor is assumed to have a rather simple and small instruction set. (Nearly all instructions use one time unit, some few (such as divide) use more. The instruction set time requirement is defined as parameters in the simulator—and therefore easy to change.) The time consumption is specified as part of the PIL program.

Monitoring facilities The simulator provides the following features for producing data necessary for doing algorithm evaluation:

- *Global clock.* The synchronous operation of a CREW PRAM implies that one global time concept is valid for all the processors. The global clock may be read and reset. The default output from the simulator gives information about the exact time for all events which produce any form of output.

- *Specifying time consumption.* In the current version of the simulator, the time used on local computations inside each processor must be explicitly given in the PIL program by the user. This makes it possible for the user to choose an appropriate level of "granularity" for the time modelling. Further, the explicit time modelling implies the advantage that the programmer is free to assume any particular instruction set or other way of representing time consumption.

- *Processor and global memory requirement.* The number of processors used in the algorithm is explicitly given in the PIL program, and may therefore easily be reported together with other performance data. The amount of global memory used may be obtained by reading the user stack pointer.

- *Global memory access statistics.* The simulator counts and reports the number of reads and the number of writes to the global memory.

- *DEMOS data collection facilities.* The general and flexible data collection facilities provided in DEMOS are available; `COUNT` may be used to record incidences, `TALLY` may be used to record time independent variables, with various statistics (mean, estimated standard deviation etc.) maintained over the samples, and `HISTOGRAM` provides a convenient way of displaying measurement data.

Note that the monitoring facilities does *not* interfere with the algorithm being evaluated. Algorithm or experiment specific monitoring are easily defined by the user.

Further details about PIL and the simulator may be found in [18].

3 Investigating the Practical Value of Cole's Parallel Merge Sort Algorithm

This section starts by explaining why Richard Cole's parallel merge sort algorithm is an important contribution to the field of parallel sorting. We describe the main principles of the algorithm, and outlines the implementation. A straightforward CREW PRAM implementation of Batcher's bitonic sorting is presented and compared with Cole's algorithm. The comparison shows how the *relative simplicity* of bitonic sorting makes it to a better algorithm *in practice*—in spite of being inferior to Cole's algorithm "in the theory".

3.1 Parallel Sorting—The Continuing Search for Faster Algorithms

The literature on parallel sorting is very extensive. A good survey is *A Taxonomy of Parallel Sorting* by Bitton, DeWitt, Hsiao and Menon [9]. Detailed descriptions of many parallel sorting algorithms is found in Akl's book devoted to the subject [3]. In 1986, there was published a bibliography containing nearly four hundred references [24]. Nevertheless, it is still appearing new interesting parallel sorting algorithms.

There are mainly two (theoretical) computational models that have been considered for parallel sorting algorithms — the circuit model and the PRAM model. An early and important result for the circuit model was the odd-even merge and bitonic merge sorting networks presented by Batcher in 1968 [5]. A bitonic merge sort network sorts n numbers in $O(\log^2 n)$ time.

The *cost* of a parallel algorithm is commonly defined as the product of its execution time and processor requirement. A *parallel sorting algorithm* is often said to be *optimal* (with respect to cost), if its cost is $O(n \log n)$.

The AKS $O(n \log n)$ sorting network The first parallel sorting algorithm using only $O(\log n)$ time was presented by *Ajtai, Komlós and Szemerédi* in 1983 [2]. This algorithm is often called the three Hungarians's algorithm, or the AKS-network. The original variant of the AKS-network used $O(n \log n)$ processors and was therefore not cost-optimal. However, Leighton showed in 1984 that the AKS-network can be combined with a variation of the odd-even network to give an optimal sorting network with $O(\log n)$ time and $O(n)$ processors [17]. Leighton points out that the constant of proportionality of this algorithm is immense and that other parallel sorting algorithms will be faster as long as $n < 10^{100}$.

In spite of being commonly thought of as a purely theoretical achievement, the AKS-network was a major breakthrough; it *proved the possibility* of sorting in $O(\log n)$ time, and *implied* the first cost optimal parallel sorting algorithm described by Leighton in 1984 [17]. The optimal *asymptotical* behavior initiated a search for improvements for closing the gap between its theoretical importance and its practical use. One such improvement is the simplification of the AKS-network done by Paterson [21]. However, the algorithm is still very complicated and the complexity constants remain impractically large [14].

Cole's CREW PRAM sorting algorithm The PRAM model is more powerful than the

circuit model. (Even an EREW PRAM may implement a sorting circuit without loss of efficiency.) Also for the PRAM model, there has been a search for a parallel sorting algorithm with optimal cost. (Some of the important results are reported by Cole [11].)

In 1986, *Richard Cole* presented a new parallel sorting algorithm called *parallel merge sort* [10]. This was an important contribution, since Cole's algorithm is the second $O(\log n)$ time $O(n)$ processor sorting algorithm—the first was the one implied by the AKS-network. Further, it is claimed to have complexity constants which are much smaller than that of the AKS-network.

3.2 Cole's Parallel Merge Sort

A revised version of the original paper is [11]—which has been used as the main reference for my implementation of the CREW PRAM variant of the algorithm.

Cole's algorithm—main principles

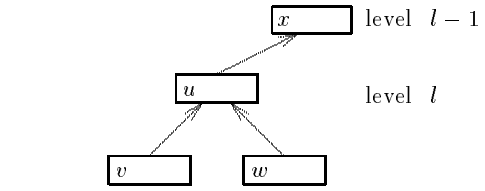
Cole's parallel merge sort assumes *n distinct items*. These are distributed one per leaf in a complete binary tree—it is assumed that *n is a power of 2*. The computation proceeds up the tree, level by level from the leaves to the root. Each internal node u merges the sorted sets computed at its children. The algorithm is based on the following $\log n$ merging procedure: ([11] 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".

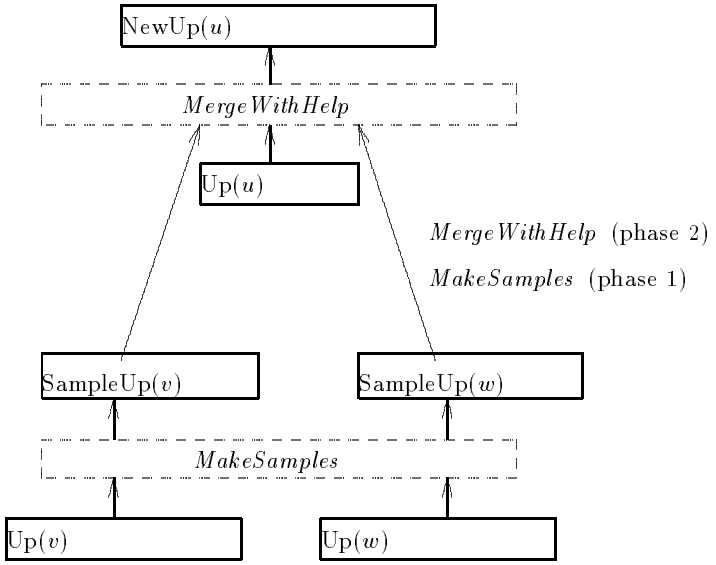
Cole made two key observations:

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.



(a)



(b)

Figure 1: Cole's parallel merge sort algorithm. Part (a): Arbitrary node u in the binary computation tree. Part (b): Computation of NewUp(u) in two phases.

During the progress of the algorithm, each node u stores an array $\mathrm{Up}(u)$ of items. The goal of each node u is to make $\mathrm{Up}(u)$ into a sorted list containing the items initially stored in the leaves of the subtree rooted at u. Each stage of Cole's merging procedure consists of two phases, see Figure 1. In phase 1, the $\mathrm{Up}(u)$ arrays are sampled in a systematic manner to produce the arrays $\mathrm{SampleUp}(u)$. In phase 2, the two samples $\mathrm{SampleUp}(v)$ and $\mathrm{SampleUp}(w)$ (from u's two child nodes) are merged into a new sequence $\mathrm{NewUp}(u)$ *with help* of the array $\mathrm{Up}(u)$.

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 $\mathrm{Up}(u)$, $\mathrm{NewUp}(u)$ and $\mathrm{SampleUp}(u)$) as the computation proceeds from the leaves to the root of the tree. The processor requirement is slightly less than $4n$. At the end of every third stage the lowest active level moves one level up towards the top—and the algorithm has exactly $3 \log n$ stages. Also, the highest active level will move one level upwards every second stage. Because of this difference in "speed" the total number of active levels will increase during the computation, until the top level has been reached. The reader is referred to Cole's paper [11] for further details about the algorithm.

The simplicity of the CREW PRAM model and the nice properties of synchronous programs made it relatively easy to develop an exact analytical model of the time requirement for the implementation of Cole's algorithm.

About the implementation

Cole's succinct description of the algorithm given in [11] is at a relatively high level giving the programmer freedom to choose between the SIMD or MIMD [12] implementation paradigms. The algorithm have been programmed in a *synchronous MIMD programming style*, as proposed for the PRAM model by Wyllie [26]. This paper gives only a brief description of the implementation, providing a crude base for discussing its time requirement. Figure 2 outlines the main program in a notation called "parallel pseudo pascal" (PPP) [19]. This notation is inspired by *parallel pidgin algol* as defined by Wyllie in [26]—with modernizations from the pseudo language notation used by Aho, Hopcroft and Ullmann in [1].

```
    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 2: Main program of Cole's parallel merge sort expressed in parallel pseudo pascal.

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.

The maximum *processor requirement* is given by the maximum size of the $\mathrm{Up}(u)$, $\mathrm{NewUp}(u)$ and $\mathrm{SampleUp}(u)$ arrays for all nodes u during the computation. We have [11]:

$$\mathit{NoOfProcs} = \sum_u |\mathrm{Up}(u)| + \sum_u |\mathrm{NewUp}(u)| + \sum_u |\mathrm{SampleUp}(u)| \quad (1)$$

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

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

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

still have n processors at the lowest active level. Similarly, the NewUp and SampleUp processors may be viewed as a "sliding pyramid of processors".

For a given n, the *exact* calculation of *NoOfProcs* is done by a loop with $\log_8 n$ iterations. (Throughout this paper, $\log n$ means $\log_2 n$.) 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 [13, 26]) `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 1.

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 *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 the array Up(u) for all active nodes in the tree, as was depicted in Figure 1. It is a relatively straightforward task.

In contrast, the $O(1)$ time merging performed by *MergeWithHelp* is a relatively complicated affair. It constitutes the major part of the algorithm description in [11], and about 90% of the code in the implementation. Of the time used by *MergeWithHelp* (781 time units), about 40% is needed to compute the so called *cross ranks* (Substep 1 and 2, p. 773, [11]), and 43% is used to *maintain ranks*(Step 2, p. 774).

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

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

The reader is referred to [20] for further details about the implementation.

3.3 Bitonic Sorting on a CREW PRAM

Batcher's bitonic sorting network [5] 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. (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

```
    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 3: Main program and time consumption of bitonic sorting emulated on a CREW PRAM.

have a similar possibility.) 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 3.

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

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

3.4 Comparison

Figure 4 shows the time used to sort n integers by Cole's algorithm compared with 1-processor insertion sort, a $n/2$ processor version of odd-even transposition sort, and our bitonic sorting algorithm. For all algorithms, time is measured in number of CREW PRAM instructions. It includes the time used on processor allocation. The algorithms start with the input in global memory and delivers the output at the same place. Note that we have logarithmic scale on both axes.

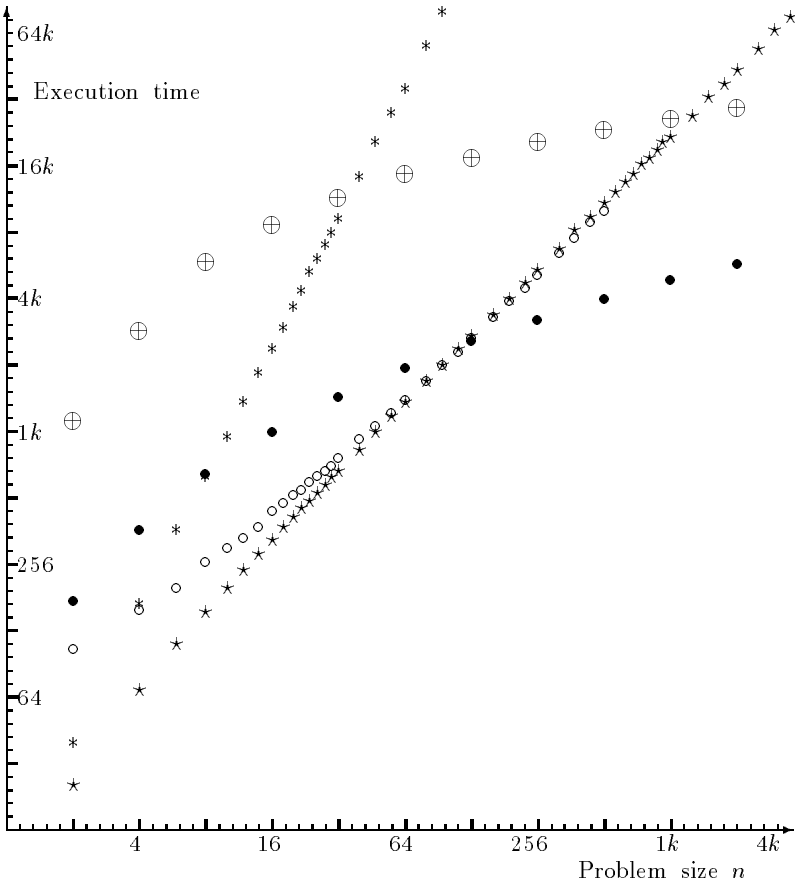


Figure 4: Time consumption (in number of (parallel) CREW PRAM instructions) measured by running parallel sorting algorithms on the CREW PRAM simulator for various problem sizes n (horizontal axis). Note the scale on both axes. Legend: $\oplus$ = Cole's algorithm ($O(\log n)$), $\bullet$ bitonic sorting ($O(\log^2 n)$), $\circ$ = odd-even transposition sort ($O(n)$), $*$ = insertion sort (worst case, $O(n^2)$), and $\star$ = insertion sort (best case, $O(n)$).

Cole's algorithm is the CREW PRAM algorithm described in Section 3.2. The implementation counts about 2000 PIL lines and was developed and tested in about 40 days of work.

Bitonic sorting is the algorithm outlined in Section 3.3. The implementation counts about 200 PIL lines and was developed and tested in about 2 days.

Odd-even transposition sort is perhaps the simplest parallel sorting algorithm. Our CREW

Table 2: Performance data for the CREW PRAM implementations of the studied sorting algorithms. Problem size n is 128. P is short for number of processors used. #R is short for total number of read operations from the global memory, and #W is the total number of writes. Time and cost is given in *kilo* CREW PRAM (unit-time) instructions, reads and writes in *kilo* locations.

Algorithm	time	P	cost	#R	#W
Cole	18.2	493	8959.3	44.7	28.2
Bitonic	2.6	64	169.0	3.9	3.7
Odd-Even	2.8	64	176.5	16.6	8.0
Insert-worst	148.7	1	148.7	8.3	8.3
Insert-Average	76.2	1	76.2	4.3	4.2
Insert-best	2.8	1	2.8	0.3	0.1

Table 3: Same table as above but with problem size $n = 256$.

Algorithm	time	P	cost	#R	#W
Cole	21.2	986	20863.8	104.6	65.2
Bitonic	3.4	128	428.2	9.9	9.4
Odd-Even	5.3	128	683.7	65.9	34.8
Insert-worst	592.4	1	592.4	32.9	32.9
Insert-Average	303.3	1	303.3	17.0	16.8
Insert-best	5.6	1	5.6	5.1	0.3

Table 4: Calculated performance data for the two CREW PRAM implementations. P is short for number of processors.

Algorithm	n	time	P
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}

PRAM implementation uses $n/2$ processors which acts as "odd" and "even" processors in an alternating style. Readers unfamiliar with the algorithm are referred to one of [4, 9, 23].

Insertion sort is the algorithm called Insertion Sort 2 in *Programming Pearls* by John Bentley [6] and presented at page 108 of that book. It was implemented as a 1-processor CREW PRAM algorithm.

The time used by the three parallel algorithms are independent of the actual problem instance (when the problem size is fixed). However, insertion sort use $O(n^2)$ time in the worst case, and $O(n)$ time in the best case (both shown in the figure). We see that bitonic sorting is fastest in this comparison in a large range of n starting at about 256.

Table 2 and 3 shows some central performance data for small test runs, $n = 128$ and $n = 256$. The two rightmost columns show the quantity of the memory use.

Bitonic sorting is faster in practice We have developed exact analytical models for the implementations of Cole's algorithm and bitonic sorting. The models have been checked against the test runs, and have been used to find the point where Cole's $O(\log n)$ algorithm becomes faster than the $O(\log^2 n)$ bitonic algorithm. The results are summarized in Table 4. The table shows time and processor requirement for the two algorithms for $n = 64k$, $n = 256k$, ($k = 2^{10}$), for the last value of n making bitonic sorting faster than Cole's algorithm, and for the first value of n making Cole's algorithm to a faster algorithm. We see that our 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 $2^{70} \approx 1.2 \times 10^{21}$, i.e. more than 1 Giga Tera items!*

A lot may be learned from medium-sized test runs Highly concurrent algorithms with polylogarithmic running time are often relatively complex. One might think that evaluation of such algorithms would require processing of very large problem instances. So far, this have not been the case. In studying the relatively complex Cole's algorithm, some hundreds of processors and small sized memories have been sufficient to enlighten the main aspects of the algorithm. In many cases, the need

for brute force (*i.e.*, huge test runs) may to a large extent be reduced by the following "working rules":

1. The *size* of the problem instance is used as *parameter* to the algorithm which is made to solve the problem for all possible problem sizes.

2. *Elaborate testing* is performed on all problem sizes that are within the limitations of the simulator.

3. A detailed *analysis* of the algorithm is performed. The possibility of making such an analysis with a reasonable effort *depends strongly on* the fact that the algorithm is *deterministic* and *synchronous*.

4. The analysis is confirmed with *measurements* from the test cases.

Together, this have made it possible to use the analytical performance model by extrapolation for problem sizes beyond the limitiations of the simulator.

4 Concluding Remarks

We can conclude that Batcher's well known and simple $O(\log n^2)$ time bitonic sorting is faster than Cole's $O(\log n)$ time algorithm *for all practical values of* n. The huge value of n reported in the previous section gives also 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. This is because it requires about 8 times as many processors than bitonic sorting, and it has a far more extensive use of the global memory.

The method for 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 at the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing [25].

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] Selim G. Akl. *Parallel Sorting Algorithms*. Academic Press, Inc., Orlando, Florida, 1985.

[4] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall International, Inc., Englewood Cliffs, New Jersey, 1989.

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

[6] Jon L. Bentley. *Programming Pearls*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[7] Graham M. Birtwistle. *DEMOS — A System for Discrete Event Modelling on Simula*. The MacMillan Press Ltd., London, 1979.

[8] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA BEGIN, second edition*. Van Nostrand Reinhold Company, New York, 1979.

[9] Dina Bitton, David J. DeWitt, David K. Hsiao, and Jaishankar Menon. A Taxonomy of Parallel Sorting. *Computing Surveys*, 16(3):287–318, September 1984.

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

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

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

[13] 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.

[14] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, 1988.

[15] Per Holm and Magnus Taube. SIMDEB User's Guide for UNIX. Technical report, Lund Software House AB, Lund, Sweden, 1987.

[16] Richard M. Karp. A Position Paper on Parallel Computation. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([25])*, pages 73–75, 1989.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] M. S. Paterson. Improved sorting networks with $O(\log n)$ depth. Technical report, Dept. of Computer Science, University of Warwick, England, 1987. Res. Report RR89.

[22] R. J. Pooley. *An Introduction to Programming in SIMULA*. Blackwell Scientific Publications, Oxford, England, 1987.

[23] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill Book Company, New York, 1987.

[24] D. Richards. Parallel sorting—a bibliography. *ACM SIGACT News*, pages 28–48, 1986.

[25] 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).

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