Constants in parallel sorting

Jyrki Katajainen
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East, Denmark
E-mail jyrki@diku.dk

Abstract

The constant factors in the time complexity of three parallel mergesort variants, running on an EREW PRAM, are analysed under a simple cost model. The algorithms studied are linear-time mergesort, odd-even mergesort and polylogarithmic mergesort. The processors of a PRAM are assumed to be word-addressable machines that can execute three-address instructions of our hypothetical assembly language in unit time. For the sake of comparison, sequential quicksort, heapsort and mergesort are examined within the same framework. The number of operations performed by linear-time mergesort, i.e. its work, is at most four times the time required by sequential mergesort. The work of polylogarithmic mergesort is about twenty times that of sequential mergesort. Moreover, polylogarithmic mergesort seems to perform relatively well compared to odd-even mergesort, but there are still place for improvements.

Keywords mergesort, sorting, parallel algorithms, constant factors.

1 Introduction

The constant factors in the running time of parallel algorithms are seldom analysed carefully, even though these are significant. For example, consider two algorithms A_1 and A_2 that both use N processors to solve a problem of size N, but A_1 requires $72\log_2 N + O(1)$ time and A_2 $2(\log_2 N)^2 + O(1)$ time. Now the theoretically superior $O(\log_2 N)$-algorithm A_1 is faster than $O((\log_2 N)^2)$-algorithm A_2 only if N is larger than about 2^{36}. Hence, in practice, one should use A_2 even though it is not asymptotically optimal.

In this paper we explore the constant factors in the time complexity of several parallel sorting algorithms, running in the EREW PRAM model. For the definition of various PRAM variants, see, for example, [17]. The main complications encountered in this work were the following:

1. It was difficult to find a satisfactory definition of the EREW PRAM model, which could be used as the basis for a detailed analysis of the constant factors.
2. The sorting algorithms were seldom described at such a level of detail that the constant factors could be analysed directly.

In Section 2 we define the cost of various primitive operations on an EREW PRAM. The cost of some typical constructs of a high-level programming language is also calculated there. In later sections we exploit these calculations when analysing the performance of various sorting algorithms.

For the sake of simplicity, we assume that the N elements to be sorted are integers. However, we want that the sorting algorithms are general so that they will not try to take advantage of this assumption. To be more specific, moves and comparisons are the only operations allowed for the input elements. This means that the number of operations performed by any sorting algorithm, deterministic or randomized, must be $\Omega(N \log_2 N)$ [22]. We say that a sorting algorithm is *work-optimal* if its running time is $O(N \log_2 N/P)$ with P processors.

In Section 4 we analyse the constant factors in the running time of three parallel mergesort variants. A simple parallel adaptation of mergesort running in linear time with a logarithmic number of processors is examined in Section 4.1, odd-even mergesort in Section 4.2, and a work-optimal polylogarithmic adaptation of mergesort in Section 4.3. For the sake of comparison, we calculate the cost of some sequential sorting algorithms within our framework in Section 3.

Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 31–February 2 1996.

The results of this paper are summarized in Table 1 (see back page). Before telling how these results are obtained, we take a review of the sorting algorithms that are work-optimal and run in polylogarithmic time.

Several efficient algorithms for parallel merging have been proposed (see, *e.g.*, [1, 4, 10, 13, 14, 18]). Any of these could be used in the realization of the standard mergesort. Our analysis in Section 4.3 is based on the merge algorithm presented by Katajainen *et al.* [18]. If P, the number of processors is $O(N/\log_2 N)$, the mergesort algorithm (*e.g.*, in the form described by Hagerup and Rüb [14]) runs in $O(N \log_2 N/P)$ time and $O(N)$ space on an EREW PRAM.

However, if speed DOES really matter and if we have more than $\lceil N/\log_2 N \rceil$ processors, sorting can be accomplished faster with optimal speed-up. Cole [8] (see also [9]) proposed a pipelined mergesort algorithm, which runs in $O(N \log_2 N/P)$ time and $O(N)$ space with $P \leq N$ processors on an EREW PRAM. The main drawback of this algorithm is that it is complex. We have not been able to code it at the same level of detail as the other parallel mergesort variants.

Martel and Gusfield [21] developed a parallel version of quicksort which runs (with high probability) asymptotically as fast as pipelined mergesort. The space consumption of the algorithm was later improved to $O(N)$ by Chlebus and Vrťo [6]. The implementation of this algorithm seems, however, to require the stronger CRCW PRAM model.

Yet another alternative is flashsort[1] which is known to run (with high probability) in $O(N \log_2 N/P)$ time even in a bounded-degree network with $P \leq N$ processors, as shown by Reif and Valiant [24]. The main idea in flashsort is to take a random sample of size, say, $\sqrt{N}$, sort the sample in a brute-force manner, divide the input into disjoint buckets induced by the sample, and sort the buckets recursively. But it is not at all obvious how this algorithm is implemented without concurrent reads and writes by using only linear space. (For example, the implementation given in [17] requires concurrent reads and superlinear space.)

2 Cost model

To be able to analyse the constant factors in the running times of various sorting algorithms, we have to specify the cost of all primitive operations in the underlining model of computation. In this section, we first define a simple cost model and then, as an example of its use, analyse the performance of some algorithms for computing prefix sums.

2.1 Target computer

The model of computation used throughout this paper is an EREW PRAM. The basic components of the model are random access machines (RAMs) which operate synchronously on a shared memory. On an EREW PRAM, only one processor has access to a memory cell at a time.

When programming an EREW PRAM, we use a Pascal-like programming language described in the survey paper by Auramo *et al.* [2]. The data types of the language are *integer*, *interval* and *array*. An interval $[j..k]$ denotes simply the set of integers $\{j, j+1, \ldots, k\}$. The control structures of the language are similar to those in Pascal, except the parallel **for**-statement. The effect of "**for** $i \in [j..k]$ **pardo** S" is that each processor P_i, $i \in [j..k]$, executes the statement S with the value i, which is its own number. A processor whose number is not in the interval $[j..k]$ does nothing.

In order to determine the cost of the statements of the high-level language, we specify the assembly language of a PRAM and show how various statements are compiled to this assembly language. We assume that the execution of any instruction involving at most three addresses takes one unit of time. Let a, b, c, i, j be (private or shared) integer variables or constants, A and B integer arrays, and ℓ a label pointing to some line of the program under execution. All the unit-cost instructions are listed below.

1. Simple assignments: $a := b$, $a := b \oplus c$, where $\oplus \in \{+, -, *, \text{DIV}\}$;
2. Array assignments: $a := A[i]$, $A[i] := b$, $A[i] := b \oplus b$, $A[i] := A[i] \oplus b$, $A[i] := A[j]$, $A[i] := B[i]$, $A[i] := A[i] \oplus A[j]$, where $\oplus \in \{+, -, *, \text{DIV}\}$;
3. Comparisons: IF $a \lhd b$ THEN GOTO ℓ, where $\lhd \in \{<, >, =, \leq, \geq, \neq\}$;
4. Jumps: GOTO ℓ; SKIP;
5. Random number generation: $a :=$ RANDOM($[i..j]$);
6. Signature register access: $a :=$ LOADINDEX.

[1]This algorithm is sometimes called samplesort. However, in sequential samplesort introduced by Frazer and McKellar [11] the buckets are sorted by quicksort. In flashsort these are sorted recursively.

The instruction SKIP is a dummy operation that simply skips to the next line of the program. The instruction RANDOM takes as its argument an interval $[i..j]$ and returns a random number from the uniform distribution over $[i..j]$. The instruction LOADINDEX fetches the processor number from the *signature register*. The meaning of the other instructions should be obvious.

Each of the above instructions involve at most three addresses. For example, to execute $A[i] := A[i] + A[j]$ we have to know the start address of array A, and the address of indices i and j. Observe that, according to our model, the cost of assignments $A[i] := A[j] + A[k]$ and $A[i] := B[j]$ is two because both of these involve four addresses.

2.2 Some translations

Let $\text{cost}(S)$ denote the *cost* of statement S, that is, the number of instructions needed for executing S in our target computer. Next we consider how some statements of the high-level language of Auramo *et al.* [2] could be compiled and we also determine the cost of these statements.

Let a and b be integer variables or constants, and S_1 and S_2 statements. The compilation of the **if**-*statement* "**if** $a < b$ **then** S_1 **else** S_2" is given in Fig. 1. From the figure we get that the cost of this statement is $\max\{\text{cost}(S_1), \text{cost}(S_2)\} + 2$. Observe that the cost of the statement will increase if the condition following the **if** is more complicated.

$$
\begin{array}{rl}
 & \text{IF } a < b \text{ THEN GOTO } \ell_1 \\
 & S_2 \\
 & \text{GOTO } \ell_2 \\
\ell_1 : & S_1 \\
 & \text{GOTO } \ell_2 \\
\ell_2 : &
\end{array}
$$

Figure 1. The translation of statement "**if** $a < b$ **then** S_1 **else** S_2" in which a and b are integer variables or constants, and S_1 and S_2 statements.

Actually, it would be possible to compile the above **if**-statement with a more compact code. However, if the statements S_1 and S_2 can be compiled such that their cost is the same, then the cost of the **if**-statement will not depend on the value of the condition. This means that no explicit syncronization is necessary when several processors are executing the **if**-statement simultaneously.

Let i be a variable, j and k integer variables or constants, and S a statement. The translation of the *sequential* **for**-*statement* "**for** $i \in [j..k]$ **do** S" is given in Fig. 2. The cost of this statement is seen to be $\max\{0, k - j + 1\} * (\text{cost}(S) + 2) + 2$. If a similar loop is programmed by using a **while**- or **repeat**-statement, the cost turns out to be the same, except one instruction is saved for the **repeat**-statement since in that case k is known to be larger than j.

$$
\begin{array}{rl}
 & \text{IF } k < j \text{ THEN GOTO } \ell_2 \\
 & i := j \\
\ell_1 : & S \\
 & i := i + 1 \\
 & \text{IF } i \leq k \text{ THEN GOTO } \ell_1 \\
\ell_2 : &
\end{array}
$$

Figure 2. The translation of statement "**for** $i \in [j..k]$ **do** S" in which i is a variable, j and k are integer variables or constants, and S is a statement.

Both **while**- and **repeat**-statements should be used with care, because the compiler can have difficulties in analysing the cost of these loops statically, that is based on the program only. To avoid the possible syncronization problems, it may be necessary for the compiler to add some syncronization code, which again may lead to some loss of efficiency. Of course, at this point the programmer can help the compiler, for instance, by ensuring that all the **while**- and **repeat**-statements executed in parallel have the same cost. Then no explicit syncronization is necessary.

As earlier, let i be a variable, j and k integer variables or constants, and S is a statement. The translation of the *parallel* **for**-*statement* "**for** $i \in [j..k]$ **pardo** S" is given in Fig. 3. The cost of this statement is $\text{cost}(S) + 4$. For the syncronization reasons, the non-active processors will execute as many instructions as the active processors. To generate the corresponding code, the compiler must know the cost of statement S.

$$
\begin{array}{rl}
 & i := \text{LOADINDEX} \\
 & \text{IF } i \geq j \text{ THEN GOTO } \ell_1 \\
 & (\text{cost}(S) + 1) \text{ SKIPs} \\
 & \text{GOTO } \ell_3 \\
\ell_1 : & \text{IF } i \leq k \text{ THEN GOTO } \ell_2 \\
 & \text{cost}(S) \text{ SKIPs} \\
 & \text{GOTO } \ell_3 \\
\ell_2 : & S \\
 & \text{GOTO } \ell_3 \\
\ell_3 : &
\end{array}
$$

Figure 3. The translation of statement "**for** $i \in [j..k]$ **pardo** S" in which i is a variable, j and k are integer variables or constants, and S is a statement.

In addition to the statements discussed above, we assume that *procedure calls* require constant time. We will not determine the exact value of the constant, because this will not have any significant influence on the time complexities of the algorithms to be analysed. The *storage allocation* in our algorithms can be done statically by one processor. Therefore, these costs are also omitted in our calculations. Some authors assume that it takes $O(\log_2 P)$ time to start P processors on an EREW PRAM, but we omit the *start-up* cost. Moreover, we assume that at the beginning of computation all processors know the start address and the size of the input array, as well as the number of processors.

2.3 Prefix sums

In the *prefix-sums* problem, we are given an array $A[0..n-1]$ of n elements and the task is to compute all partial sums $\bigoplus_{i=0}^{j} A[i]$, $j \in [0..n-1]$. Here $\oplus$ can be any associative operation that keeps the sum in the universe from which the input elements are drawn. It is well-known that this problem can be solved in $O(\log_2 n)$ time with $O(n/\log_2 n)$ processors on an EREW PRAM (see, *e.g.*, [5] or [20]).

Since a prefix-sums subroutine is frequently used in various parallel algorithms, it is important to implement it as efficiently as possible. As the basis of our implementation, we use the elegant (non-optimal) subroutine described by Hillis and Steele [15, p. 1173]. The implementation of their subroutine is given in Fig. 4(a) without using the functions $\log_2 x$ and 2^x, for these are not assumed to be unit-time operations in our model of computation.

By using the non-optimal procedure, a work-optimal procedure can be developed by following the guidelines of Kruskal *et al.* [20]. Here the standard partitioning technique is used. First, the input is divided into about P blocks each containing at most $\lceil n/P \rceil$ elements, where P is the number of processors available. Second, the sums within each block are computed in parallel by using a sequential algorithm. Third, the prefix-sums of the block sums are computed by using the non-optimal parallel routine. Fourth, the first element in each block is replaced by the sum of the prefix value of the preceeding block and itself. Finally, the prefix-sums within each block are computed in parallel by using a sequential algorithm. The implementation details are given in Fig. 4(b).

From Fig. 4 the cost of the prefix-sums procedures is readily obtained by using the costs derived for various statements in Section 2. The complexities depend on type T and operation $\oplus$. For usual integer arithmetic, the running time of the non-optimal procedure is $6\log_2 n + O(1)$ with n processors and that of the work-optimal procedure $7n/P + 6\log_2 P + O(1)$ with P processors. For $\lceil n/\log_2 n \rceil$ processors, the cost of the latter procedure is $13\log_2 n + O(1)$.

```
(a)
proc[0..n − 1] ⊕-PREFIX_5(A: array[0..n − 1] of T)
for i ∈ [0..n − 1] pardo
  j := 1
  while j < n do
    if i ≥ j then A[i] := A[i − j] ⊕ A[i]
    j := 2 ∗ j
(b)
proc[0..P − 1] ⊕-PREFIX_0(A: array[0..n − 1] of T)
shared X: array[0..P − 1] of T
for i ∈ [0..P − 1] pardo
  b := ⌈n/P⌉; p := P − ⌊(P ∗ b − n)/b⌋
  if i ∈ [0..p − 1] then
    x := A[i ∗ b]
    if i ≠ p − 1 then
      for j ∈ [i ∗ b + 1..(i + 1) ∗ b − 1] do
        x := x ⊕ A[j]
    else
      for j ∈ [i ∗ b + 1..n − 1] do
        x := x ⊕ A[j]
    X[i] := x
    ⊕-PREFIX_5[0..p − 1](X[0..p − 1])
    if i ≠ 0 then A[i ∗ b] := X[i − 1] ⊕ A[i ∗ b]
    if i ≠ p − 1 then
      for j ∈ [i ∗ b + 1..(i + 1) ∗ b − 1] do
        A[j] := A[j − 1] ⊕ A[j]
    else
      for j ∈ [i ∗ b + 1..n − 1] do
        A[j] := A[j − 1] ⊕ A[j]
```

Figure 4. (a) A non-optimal and (b) work-optimal procedure for computing prefix sums.

The *copying* problem where the task is to create n copies of an element is closely related to the prefix-sums problem. Efficient programs for copying can be directly written by using only parts of the procedures in Fig. 4 (*cf.* Fig. 5). It takes $3n/P + 6\log_2 P + O(1)$ time with P processors to create n copies of an integer.

2.4 Segmented computations

The prefix-sum procedures given in Fig. 4 compute the prefix sums of the elements in $A[0..n-1]$. In some cases, one wants to compute the prefix sums within a subarray $A[\ell..h]$. In a more general case, one perhaps wants to compute the prefix sums within many subarrays at the same time. In the literature this problem is called the *segmented prefix-sums* problem. Usually the segmented prefix-sums problem is solved (see, *e.g.*, [25]) by calling the standard procedure with a new associative operation $\odot$ defined for pairs $\langle e, x\rangle$, where x is an original element and e is its segment label, as follows:

(a)

```
proc[0..n − 1] COPY_6 (label: array[0..n − 1] of int,
                       B: array[0..n − 1] of T)
for i ∈ [0..n − 1] pardo
  j := 1
  while j < n do
    if i ≥ j then
      if label[i] = label[i − j] then B[i] := B[i − j]
    j := 2 ∗ j
```

(b)

```
proc[0..P − 1] COPY_0 (label: array[0..n − 1] of int,
                       B: array[0..n − 1] of T)
shared t: array[0..P − 1] of int
shared X: array[0..P − 1] of T
for i ∈ [0..P − 1] pardo
  b := ⌈n/P⌉; p := P − ⌊(P ∗ b − n)/b⌋
  if i ∈ [0..p − 1] then
    ℓ := label[i ∗ b]; k := i ∗ b
    if i ≠ p − 1 then
      for j ∈ [i ∗ b + 1..(i + 1) ∗ b − 1] do
        if label[j] ≠ ℓ then ℓ := label[j]; k := j
    else
      for j ∈ [i ∗ b + 1..n − 1] do
        if label[j] ≠ ℓ then ℓ := label[j]; k := j
    t[i] := label[k]; X[i] := B[k]
    COPY_6[0..p − 1](t[0..p − 1], X[0..p − 1])
    if i ≠ 0 then
      if t[i − 1] = label[i ∗ b] then B[i ∗ b] := X[i − 1]
    ℓ := label[i ∗ b]
    if i ≠ p − 1 then
      for j ∈ [i ∗ b + 1..(i + 1) ∗ b − 1] do
        if label[j] = ℓ then B[j] := B[j − 1]
        else ℓ := label[j]
    else
      for j ∈ [i ∗ b + 1..n − 1] do
        if label[j] = ℓ then B[j] := B[j − 1]
        else ℓ := label[j]
```

Figure 5. (a) A non-optimal and (b) work-optimal procedure for segmented copying.

$$\langle e, x\rangle \odot \langle f, y\rangle := \begin{cases} \langle f, x \oplus y\rangle & \text{if } e = f \\ \langle f, y\rangle & \text{if } e \neq f. \end{cases}$$

As a special case of the segmented prefix-sums problem, we consider the *segmented copying* problem. The input for this problem consists of two arrays *label* and B of size n. The labels are any integers such that, if $label[i] = label[k]$, then also $label[j] = label[i]$, for all $j \in [i..k]$. At the beginning of computation, if $label[i-1] \neq label[i]$, location $B[i]$ contains an element that has to be copied into other locations having the same label. A work-optimal procedure for accomplishing the segmentwise copying is given in Fig. 5(b). The cost of this procedure is $14n/P + 10\log_2 P + O(1)$, for P processors.

Lateron we need a solution for the *segmented multicopying* problem. In this problem the input consists of two two-dimensional arrays *label* and C with m rows and n columns, $m \leq n$. At each column the labels should be identical. The task is to solve a segmented copying problem at each row. The number of processors available for solving this problem is n.

Assume first that $m \geq n/3$. In this case we simply assign one processor for each row and solve the segmented copying problems in parallel by using a sequential algorithm at each row. The running time of the sequential algorithm is $7n + O(1)$, which is now bounded by $21m + O(1)$. Assume next that $m < n/3$. In this case we distribute the processors evenly over the rows and solve the segmented copying problems in parallel by using the parallel algorithm of Fig. 5(b) at each row. Each row gets $\lceil n/m\rceil$ or $\lfloor n/m\rfloor$ processors. Therefore, the overall running time is $14n/\lfloor n/m\rfloor + 10\log_2(n/m) + O(1)$, which is bounded by $21m + 10\log_2 n + O(1)$ when $m < n/3$. To sum up, for all $m \leq n$, the problem can be solved in $21m + 10\log_2 n + O(1)$ time with n processors.

In the above examples the segments were specified by labels. Another possibility is to let each processor know the boundaries of the segment it should operate with. The main complication in the segmented computations mentioned was that the segments may vary in size. If the segmented computations were of the same size, it would be much easier to carried out these in parallel.

3 Sequential sorting algorithms

In this section we analyse the constant factors in the time complexity of quicksort, heapsort and mergesort. For the purpose of our analysis, we use the Pascal-programs presented in Sedgewick's book [26]. Throughout this section, we let N denote the number of elements to be sorted. The running time of quicksort is a random variable that is bounded below by $3N\log_2 N - O(N)$ and, with high probability, above by $20N\log_2 N + O(N)$. The worst-case running times of heapsort and mergesort turn out to be $11N\log_2 N + O(N)$ and $12N\log_2 N + O(N)$, respectively. The reader is encouraged to compare these results with those found in Knuth's book [19].

3.1 Quicksort

Randomized quicksort proposed by Hoare [16] selects a random element from the input, called a *pivot*, partitions the input elements into two parts, the elements smaller than the pivot and those larger than the pivot, and sorts these parts recursively. To analyse the algorithm we use the implementation given in [26, p. 118]. Note that this implementation uses a deterministic pivot but the program is easily modified to use a random pivot.

The computation of quicksort can be seen as a binary tree where the root corresponds to the first pivot, its children the pivots of the subfiles, and so on. The depth of each element in this tree is the same as the number of partitionings this particular element takes part in. Thus, the *internal path length* of this tree, that is the sum of the depths of all nodes, can be used to estimate the complexity of quicksort. It is a well-known fact that the internal path length of a binary tree with N nodes is at least $N \log_2 N - O(N)$ (see, *e.g.*, [19]). It is not difficult to see that the partitioning procedure given in [26, p. 118] performs at least $3k$ three-address instructions when partitioning an integer array of size k. Hence, the cost of each partitioning is three per element and the running time of the algorithm is at least $3N \log_2 N - O(N)$.

Various researchers have analysed the *probable case* complexity of quicksort. Chlebus and Vrťo [6] proved that, with probability $1 - N^{-\alpha}$, no element takes part in more than $1 + (3 + \alpha) \log_2 N$ partitionings. When the maximal number of swaps is performed, the cost of a partitioning involving k elements is $4k + O(1)$. Since the total number of (non-trivial) partitionings is at most $N - 1$, we get that, with high probability, Sedgewick's implementation performs at most $4 \cdot (3 + \alpha) N \log_2 N + O(N)$ instructions. For example, at the reliability level $\alpha = 2$, the running time is less than $20N \log_2 N + O(N)$.

3.2 Heapsort

In practical applications quicksort seems to be considerably faster than other sorting algorithms. According to our analysis, the *heapsort* implementation given by Sedgewick [26, pp. 156 and 152] requires at most $11N \log_2 N + O(N)$ instructions in our target computer. This means that the probable case analysis gives a pessimistic picture of the performance of quicksort.

One drawback in Sedgewick's heapsort implementation is that the *downheap* operation restoring the heap is programmed with a **while**-statement. The heap condition can be satisfied somewhere in the interior of the heap before the bottom of the heap is hit. Therefore, if many downheap operations are executed in parallel on heaps of the same size, some operations can take considerably less time than others. To avoid explicit syncronization, the downheap operation should be rewritten such that the number of instructions executed is always logarithmic.

Another reason why we have not used heapsort in our parallel algorithms is that it is not suited for segmented computations. Usually heapsort is implemented such that the root of the heap is in position 1 of the input array. Of course, one could keep the root in any other position, but this would make the index arithmetic more complicated. This would also have an effect on the constant factor in the running time.

3.3 Mergesort

Mergesort is a typical divide-and-conquer algorithm. The input is first divided into two parts, these parts are sorted recursively, and then the two sorted parts are merged together to form the final output. The heart of the construction is the merge subroutine, the description of which can again be found from Sedgewick's book [26, pp. 164 or 166]. According to our calculations, the implementation given in [26, p. 166] merges two subarrays of respective size m and n, $m \leq n$, in $13m + 11n + O(1)$ time. This implies that the cost of mergesort [26, p. 166] is $12N \log_2 N + O(N)$ in the worst case.

The implementations of the merge and mergesort procedures given by Sedgewick are suited for segmented computations. Moreover, the merge procedure can be modified so that it merges two subarrays of size m and n, $m \leq n$, in precisely $24n + O(1)$ time. If a suitable number of SKIP-instructions is executed at the end of the merge procedure, an implementation is got that merges the subarrays in $24M + O(1)$ steps, for any $M \geq n$. A similar trick can be done for the mergesort procedure in order to obtain an implementation that sorts N elements in $12M \log_2 M + O(M)$ time, for any $M \geq N$. When these modified procedures are used as building blocks in our parallel mergesort adaptations, no explicit syncronization is necessary.

4 Parallel mergesort algorithms

Mergesort can be implemented in a parallel computer by sorting the two halves in parallel. An interesting question is how the merge routine is implemented efficiently. Depending on the merge routine used, three parallel variants of mergesort are got with running times $O(N)$ (using $O(\log_2 N)$ processors), $O((\log_2 N)^2)$ (using N processors), and $O((\log_2 N)^2)$ (using $O(N/ \log_2 N)$ processors), where N is the size of the input. Naturally, these algorithms can be slowed down to utilize only P processors. We assume throughout this section that P is a power of two.

4.1 Linear-time mergesort

We describe next a parallel algorithm, called here *linear-time mergesort*, that sorts the N input elements work-optimally if $P \leq O(\log_2 N)$. The computation proceeds as follows. First, the input is divided into P blocks such that the size of each block is $\lceil N/P \rceil$ or $\lfloor N/P \rfloor$. Second, the blocks are sorted in parallel by applying the sequential mergesort to each block. Third, $P/2$ processors are utilized so that each of them merges a pair of blocks. Fourth, $P/4$ processors are put to merge simultaneously two of the remaining $P/2$ blocks, and so on. When the first processor has done the last large merge, the whole input is in sorted order.

We leave the detailed implementation of this algorithm for the interested reader. At this point one should, however, observe that the leading term in the running time of the algorithm does not depend on the implementation details. The computational costs are dominated by those of the sequential mergesort and merge procedures.

By Section 3.3, the cost of the bottom-level sorts is $12N \log_2 N/P + O(N/P)$. After h merging levels, $h \in [0..\log_2 P - 1]$, the maximum size of any block merged is $2^h(N/P + 1)$. Hence, by Section 3.3, the overall cost of the merges is bounded by $\sum_{h=0}^{\log_2 P-1}(24 \cdot 2^h(N/P+1) + O(1))$, which is $24N + O(P)$. Thus, the algorithm uses never more than $12N \log_2 N/P + 24N + O(N/P + P)$ time. If P is equal to the largest power of two not larger than $\log_2 N$, the time bound is $48N + O(N/\log_2 N)$.

4.2 Odd-even mergesort

In *odd-even mergesort*, merging is carried out by the odd-even merge algorithm of Batcher [3], originally presented in the form of a merging network. An implementation of this algorithm on a PRAM can be found, for example, in [23, Appendix].

To be able to describe the merge algorithm properly, we assume that the input arrays X and Y are of equal size, and that their total size n is a power of two. Since we will use the procedure in mergesort, we assume that the input arrays are consecutive subarrays of the same array. Let us first consider how n processors can be used to merge X and Y.

The merge algorithm is recursive. First, the elements of X located at odd positions are merged with those located at the odd positions in Y. At the same time, the subarrays consisting of elements located at even positions are merged recursively. The final output is then obtained by shuffling the sorted subarrays returned by the recursive calls and possibly swapping pairs of adjacent elements.

```
proc[1..n] ◁-ODD-EVEN(A: array[1..n] of T)
% Assume that n = 2^k, k > 0.
for i ∈ [1..n] pardo
1. ℓ := 0; h := n
   d := 2; odd := (⌊i/2⌋ * 2 ≠ i)
   while d < n do
2.   x := A[i]
     if odd then A[⌈(ℓ + i)/2⌉] := x
     else A[(h + i)/2] := x
3.   m := (ℓ + h)/2
     if i ≤ m then h := m
     else ℓ := m
     d := 2 * d
   if odd then
4.   t := A[i]; u := A[i + 1]
     if u ◁ t then A[i] := u; A[i + 1] := t
   d := 2
   while d < n do
5.   x := A[i]
     j := ⌈i/d⌉; odd_group := (⌊j/2⌋ * 2 ≠ j)
     if odd_group then m := h; h := h + d
6.   else m := ℓ; ℓ := ℓ − d
     if i ≤ m then A[2i − ℓ − 1] := x
     else A[2i − h] := x
     if i < m then
7.     t := A[2i − ℓ]; u := A[2i − ℓ + 1]
       if u ◁ t then A[2i − ℓ] := u; A[2i − ℓ + 1] := t
     d := 2 * d
```

Figure 6. In-place odd-even merge.

In our iterative implementation given in Fig. 6, the operations before the recursive calls are done in a loop and those after the recursive calls in another. At each step, every processor moves some element to another location, and then it adjusts the local variables indicating the subarray to be considered next.

From Fig. 6 the cost of the merge procedure can be calculated. After some obvious optimization, this is seen to be at most $40 \log_2 n + O(1)$ with n processors. In the segmented form the cost is the same. If P, the number of physical processors is less than n, the actions of the n virtual processors should be simulated. The easiest way of doing this is to carry out the seven critical sections of the program indicated in Fig. 6 in a **repeat**-loop such that the ith physical processor simulates the virtual processors i, $i + P$, $i + 2P, \ldots$. In order to transfer information from one section to another the variables ℓ, h, m and x should be stored in arrays. This makes some of the assignments more expensive. After these modifications, the running time of the procedure with P processors turns out to be at most $54n \log_2 n/P + 9 \log_2 n + O(n/P)$.

Let us now switch to sorting. Assume first that N is a power of two. The structure of the odd-even mergesort algorithm is similar to that of the algorithm given in Section 4.1. After block sorting, the sorted blocks are repeatedly merged pairwise. Now the only difference its that at each merging level, all processors are in use. We omit the details of implementation here, since the maintanance of the boundaries of the merged subarrays does not affect the leading term in the time complexity of the algorithm.

As earlier the bottom-level sorts require $12N \log_2 N/P + O(N/P)$ time. The number of merging levels is $\log_2 P$. At level h, $h \in [0..\log_2 P - 1]$, the size of the blocks merged is $2^h N/P$ and, for each merge, 2^{h+1} processors are allocated. Thus, the overall cost of the merges is bounded by $\sum_{h=0}^{\log_2 P-1}(54\, 2^{h+1} N \log_2(2^{h+1}N/P)/(2^{h+1}P)$ $+9\log_2(2^{h+1}N/P) + O(N/P))$. To sum up, the running time is bounded by $54N \log_2 N \log_2 P/P + 12N \log_2 N/P + 9 \log_2 N \log_2 P + O(N \log_2 P/P)$.

Consider now the case that N is not a power of two. Let M be a power of two such that $N \leq M < 2N$. As a preprocessing step, we compute the maximum among the input and create M copies of it. After this we construct a new input that contains the original elements plus $M - N$ dummy elements, all larger than the maximum. Then these M elements are sorted by the odd-even sorting routine described above. As a postprocessing step, the dummy elements are discarded. The total cost of this preprocessing and postprocessing is at most $O(N/P + \log_2 N)$. This cost does not have any significant effect on the total running time. During the initial sorts the dummy elements need not be touched. Therefore, this odd-even mergesort algorithm runs in $108N \log_2 N \log_2 P/P + 12N \log_2 N/P + 9 \log_2 N \log_2 P + O(N \log_2 P/P)$ time.

4.3 Work-optimal polylogarithmic mergesort

In our third parallel mergesort variant, a fast work-optimal merge algorithm is used. Several such algorithms have been proposed (see, *e.g.*, [1, 4, 10, 13, 14, 18]). The analysis in this section is based on the merge algorithm by Katajainen *et al.* [18].

Let X and Y denote the two arrays that are to be merged. Further, let n denote their total length. Since the merge algorithm to be described is used in mergesort, we assume that the input arrays are subarrays of the same array. We assume also that $P \geq 8$. If this is not the case, *i.e.* P is 1,2, or 4, we allocate only one processor for the whole task and carry out it sequentially. By Section 3.3 the running time is $12n + O(1)$, which is less than $48n/P + O(1)$.

The parallel merge algorithm is based on the partitioning technique (*cf.* [17, Section 2.4.2]). First, the input arrays are divided into P blocks of size at most $\lceil n/(P-2) \rceil$, expect perhaps the first X-block and the last Y-block that can be smaller. Second, the first element x (y) in each block of X (Y) is marked. Call these *head-marks*. Third, the first element in Y (X) that is larger than x (y) is also marked. Call these *cross-marks*. The marked elements partition the input into disjoint *subblocks* of size at most $\lceil n/(P-2) \rceil$. Fourth, in parallel, each subblock started by a head-mark is merged with the subblock started by the corresponding cross-mark by using a sequential algorithm. The results of these merges are directed to the final destinations in the output array.

The most essential part in the algorithm is how the computation of the cross-ranks is carried out. When the positions of the cross-marks are known, the boundaries of the induced subblocks can be easily determined. For the details, see [18]. If concurrent reads were allowed, cross-marking could be done by parallel binary searches. On an EREW PRAM, a more careful organization of the computation is required. The solution proposed in [18] is described below.

First, for each head-marked element, the block in which its cross-mark lies is located by using the technique, called *searching by merging*, introduced by Preparata [23]. All elements with a head-mark are merged by using the odd-even merge algorithm and the final location of each element is used to determine the block of the corresponding cross-mark. In order to enable this calculation, the original index of each element should be transported with it. Call the block where the cross-mark of an element lies its *cross-block*. Second, for every head-marked element a copy of its cross-block is created. Hence, we have here an instance of the segmented multicopying problem introduced in Section 2.4. As labels we use the array indices of the cross-block. Third, the element that should be cross-marked is found within each cross-block by binary search. These searches are again done in parallel.

Due to lack of space, we shall not present the merging program. From the description above it should, however, be clear that the time complexity is dominated by the modules: odd-even merge, initiation of the data structures needed for segmented multicopying, segmented multicopying, binary search, and sequential merge. Odd-even merge was already analysed in Section 4.2, multicopying in Section 2.4, and sequential merge in Section 3.3. The initiation required for segmented multicopying is nothing but a sequential scan over the columns; this can be programmed in the very same way as done in Fig. 4(b). A binary-search routine suitable for our purposes can be found, for instance, from the book by Gonnet and Baeza-Yates [12, p. 38].

To summarize our calculations, the cost of the different modules is as follows: odd-even merge $48 \log_2 P + O(1)$ (recalling that element movements are now more costly), initiation for segmented multicopying $5n/(P-2) + O(1)$, segmented multicopying $21n/(P-2) + 10 \log_2 P + O(1)$, binary search $8 \log_2 n + O(1)$, and sequential merge $24n/(P-2) + O(1)$. All other computations take O(1) time. Since $P \geq 8$, the merge routine uses never more than $67n/P + 58 \log_2 P + 8 \log_2 n + O(1)$ time.

We use this merge routine in the elegant mergesort variation proposed by Hagerup and Rüb [14]. Their proposal makes the processor allocation easy and the algorithm runs efficiently even if N is not a power of two. The basic idea is to replace each element x by a pair $\langle e, x \rangle$, where e denotes the *segment label* of x. Now sorting can be accomplished by calling the merge routine $\lceil \log_2 N \rceil$ times and by adjusting the segment labels there between properly (for details, see [14]).

The only drawback with this approach is that it doubles the cost of element comparisons and element moves. A careful analysis shows that the cost of the critical modules in the merge routine increases to the following figures: odd-even merge $58 \log_2 P + O(1)$, initiation for segmented multicopying $8N/P + O(1)$, segmented multicopying $30N/P + 11 \log_2 P + O(1)$, binary search $10 \log_2 N + O(1)$, and sequential merging $38N/P + O(1)$. The manipulation of segment labels between the merges can be carried out in $O(1)$ time with N processors. When these computations are simulated with P processors, $14N/P + O(1)$ time is used. To summarize, the time requirement of this mergesort variation is $90 N \log_2 N/P + 10(\log_2 N)^2 + 69 \log_2 N \log_2 P + O(N/P + \log_2 N)$ with P processors. When P is the largest power of two not larger than $N/\log_2 N$, this equals to $259(\log_2 N)^2 + O(\log_2 N)$.

5 Conclusions

The performance of various sorting algorithms has been analysed under a simple cost model. The results of this study are summarized in Table 1. The computational cost of work-optimal polylogarithmic mergesort turns out to be about twenty times that of sequential mergesort. That is, assuming that twenty ideal processors are available this parallel algorithm performs as well as the sequential algorithm. It would be interesting to know whether there exists another work-optimal polylogarithmic sorting algorithm requiring even less overhead.

The goodness of various algorithms depends heavily on the total number of physical proccessors. For example, when only a few processors are utilized, linear-time mergesort is clearly superior to polylogarithmic mergesort.

In the analyses of this paper the assignments involving at most three addresses were assumed to take one unit of time. However, in existing parallel computers with no shared memory, communication between the processors costs often much more than local computations. For instance, the measurements performed at our institute [7] indicate that on the 16 processor Meiko CS2 computer the communication cost is about the same as that for 5 000 integer computations. Assuming that communication is really as expensive as it is on our Meiko CS2 and making the simplifying assumption that all steps of polylogarithmic mergesort require communication, which is a very pessimistic assumption, this parallel algorithm will be able to outperform the sequential algorithm only when more than 100 000 processors are available and when the size of the input is larger than about two millions. This shows the importance of new innovations in computer manufactoring. Without these, PRAM techniques will never be used in practical parallel computing.

Acknowledgements

The author would like to thank Jesper Larsson Träff for providing stimulus for this research. This work was partially supported by the Danish Natural Science Research Council under contract No. 9400952 (Project "Computational Algorithmics").

References

[1] R. J. Anderson, E. W. Mayr, M. K. Warmuth: Parallel approximation algorithms for bin packing. *Information and Computation* **82** (1989) 262–277.

[2] Y. Auramo, J. Katajainen, J. Kulmala: Finding the maximum in parallel random access machines. *Bulletin of the EATCS* 52 (1994) 315–334.

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

[4] G. Bilardi, A. Nicolau: Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal on Computing* **18** (1989) 216–228.

[5] G. E. Blelloch: Prefix sums and their applications. In J.H. Reif (Ed.): *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers (1993) 35–60.

[6] B. S. Chlebus, I. Vrťo: Parallel Quicksort. *Journal of Parallel and Distributed Computing* **11** (1991) 332–337.

[7] J. Clausen: Private communication, November 1995.

[8] R. Cole: Parallel merge sort. *SIAM Journal on Computing* **17** (1988) 770–785. Correction *SIAM Journal on Computing* **22** (1993) 1349.

[9] R. Cole: Parallel merge sort. In J. H. Reif (Ed.): *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers (1993) 453–495.

[10] E. Dekel, I. Ozsvath: Parallel external merging. *Journal of Parallel and Distributed Computing* **6** (1989) 623–635.

[11] W. D. Frazer, A. C. McKellar: Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM* **17** (1970) 496–507.

[12] G. H. Gonnet, R. Baeza-Yates: *Handbook of Algorithms and Data Structures in Pascal and C*. Addison-Wesley Publishing Company (1991).

[13] X. Guan, M. A. Langston: Time-space optimal parallel merging and sorting. *IEEE Transactions on Computers* **40** (1991) 596–602.

[14] T. Hagerup, C. Rüb: Optimal merging and sorting on the EREW PRAM. *Information Processing Letters* **33** (1989) 181–185.

[15] W. D. Hillis, G. L. Steele, Jr: Data parallel allgorithms. *Communications of the ACM* **29** (1986) 1170–1183. Update *Communications of the ACM* **30** (1987) 78.

[16] C. A. R. Hoare: Quicksort. *The Computer Journal* **5** (1962) 10–15.

[17] J. JáJá: *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company (1992).

[18] J. Katajainen, C. Levcopoulos, O. Petersson: Space-efficient parallel merging. *Informatique théorique et Applications* **27** (1993) 295–310.

[19] D. E. Knuth: *The Art of Computer Programming*, Volume 3: *Sorting and Searching*. Addison-Wesley Publishing Company (1973).

[20] C.P. Kruskal, L. Rudolph, M. Snir: The power of parallel prefix. *IEEE Transactions on Computers* **C-34** (1985) 965–968.

[21] C.U. Martel, D. Gusfield: A fast parallel quicksort algorithm. *Information Processing Letters* **30** (1989) 97–102.

[22] K. Mehlhorn: *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag (1984).

[23] F. P. Preparata: New parallel-sorting schemes. *IEEE Transactions on Computers* **C-27** (1978) 669–673.

[24] J. H. Reif, L. G. Valiant: A logarithmic time sort for linear size networks. *Journal of the ACM* **34** (1987) 60–76.

[25] J. T. Schwartz: Ultracomputers. *ACM Transactions on Programming Languages and Systems* **2** (1980) 484–521.

[26] R. Sedgewick: *Algorithms*. 2nd Edition, Addison-Wesley Publishing Company (1988).

Table 1. The performance of various sorting algorithms when sorting N integers on an EREW PRAM with P processors, $P \leq N$. The parallel algorithms require that P is a power of two. The running time of quicksort is a random variable, the value of which is within the mentioned interval with probability $1 - N^{\alpha}$, for any fixed $\alpha > 0$. For the other algorithms the upper bound of their worst-case running time is given.

Algorithm	Time complexity
quicksort	$[3N \log_2 N - O(N)..4(3+\alpha)N \log_2 N + O(N)]$
heapsort	$11N \log_2 N + O(N)$
mergesort	$12N \log_2 N + O(N)$
linear-time mergesort	$12N \log_2 N/P + 24N + O(N/P)$
odd-even mergesort	$108N \log_2 N \log_2 P/P + 9 \log_2 N \log_2 P + O(N \log_2 N/P)$
polylogarithmic mergesort	$90N \log_2 N/P + 10(\log_2 N)^2 + 69 \log_2 N \log_2 P + O(N/P + \log_2 N)$